PROCEEDINGS

OF THE

1988 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 15-19, 1988

Vol. III Algorithms and Applications

David H. Bailey, Editor

Sponsored by

Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

THE PENNSYLVANIA STATE UNIVERSITY PRESS
UNIVERSITY PARK AND LONDON

1988 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING
Vol. III Algorithms and Applications
Bailey
PENN STATE

PROCEEDINGS

OF THE

1988 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 15-19, 1988

Vol. III Algorithms and Applications

David H. Bailey, Editor

Sponsored by

Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

THE PENNSYLVANIA STATE UNIVERSITY PRESS
UNIVERSITY PARK AND LONDON

Library of Congress Catalog Card Number 79-640377
ISSN 0190-3918
ISBN 0-271-00654-4
IEEE Computer Society Order Number 889
IEEE Catalog Number 88CH2625-2

Additional copies may be obtained from:
Penn State Press
215 Wagner Building
University Park, PA 16802

PREFACE

Interest in the field of parallel processing continues to climb. This trend is evidenced by the sharp increase in papers submitted to the International Conference on Parallel Processing during recent years:

Year	Papers Submitted	Papers Accepted	Percent
1980	170	65	57
1983	240	136	57
1986	400	170	43
1987	487	174	36
1988	590	173	29

Although the number of submissions continues to increase, the number of accepted papers this year and in the past two years has remained relatively unchanged. This is due to the limitation imposed by the fixed number of hours available for the conference. As a result, a record number of papers had to be rejected. This year, the conference proceedings is being published in three volumes according to the subject category. The breakdown of submissions and acceptances in the three main categories of this conference is as follows:

Category	Papers Submitted	Papers Accepted	Percent
Architecture	264	74	28
Software	144	43	30
Algorithms and Applications	182	56	31

Of the 173 papers that were accepted, 79 were accepted as regular papers and 94 were accepted as short papers. Many papers that normally would have been accepted as long papers were accepted as short papers in order to meet the maximum number of paper-sessions allotted for the conference.

Finding sufficient numbers of qualified reviewers was a particularly challenging task this year, due to the record number of submissions. Over 1,000 professionals in the field participated in this process. This year the process of selecting referees was simplified by the use of questionnaires, which were mailed to previous participants in the conference. The information on the completed questionnaires were entered into databases, which then allowed the conference chairmen to select reviewers qualified in fairly specialized fields. Even so, numerous papers were so highly specialized that custom selection of referees was still required. It appears that an even more detailed breakdown of specializations will be needed for these questionnaires in the future. Greater effort will also be required in the future to find additional reviewers to adequately evaluate the increasing numbers of submissions.

I wish to thank the management of the Numerical Aerodynamic Simulation Systems Division at NASA Ames for providing me the opportunity to serve on the program committee this year. I also wish to thank the following persons on our staff who assisted in selecting referees and in handling the correspondence: Liviu Lustman, Martin Fouts, Julie Swisshelm, Horst Simon, Creon Levit, Gina Riley, Saundra Ramirez, and Reina Trinwith. I wish also to thank Prof. Tse-yun Feng for his support and encouragement in this effort.

David H. Bailey
NASA Ames Research Center
Moffett Field, CA 94035

LIST OF REFEREES

Abello, J.	U. C. Santa Barbara
Abileah, R.	SRI International
Adams, G. B.	Purdue Univ.
Agrawal, D. P.	North Carolina St. Univ.
Agrawala, A.	Univ. of Maryland
Ahmad, M. O.	Concordia Univ.
Alaghband, G.	Univ. of Colorado
Alexander, W. E.	North Carolina St. Univ.
Altmann, E.	Carnegie Mellon Univ.
Antony, R.	U. S. Army
Armstrong, J.	Convex Computer Corp.
Bailey, D. H.	NASA Ames Research Ctr.
Bappana, R. V.	Univ. of Southern Cal.
Baru, C. K.	Univ. of Michigan
Bastani, F. B.	Univ. of Houston
Berger, M.	Courant Inst.
Berkling, K.	Syracuse Univ.
Bermudez, M. E.	Univ. of Florida
Bhanu, B.	Honeywell Research Ctr.
Bhargava, B.	Purdue Univ.
Bhasker, J.	Honeywell, Inc.
Blelloch, G.	MIT AI Lab.
Bodorik, P.	Tech. Univ. Nova Scotia
Bose, P.	IBM Watson Research Ctr.
Bourbakis, N.	George Mason Univ.
Bowyer, K. W.	Univ. of South Florida
Braaten, M. E.	G. E. Res. and Dev. Ctr.
Breitkreutz, T.	Univ. of Alberta
Brenner, A. E.	IDA - SRC
Breuer, M. A.	Univ. of Southern Cal.
Browne, J. C.	Univ. of Texas
Bryant, B.	Univ. of Alabama Birm.
Bryant, R. M.	IBM Watson Research Ctr.
Buell, D.	IDA - SRC
Bui, T. D.	Concordia Univ.
Burn, R.	Systems Control
Buzbee, B. L.	NCAR
Cappello, P. R.	U. C. Santa Barbara
Cargo, D.	U. S. Dept. of Defense
Carlson, D. A.	IDA - SRC
Carty, F. G.	Goodyear Aerospace Corp.
Cazes, A.	IBM Watson Research Ctr.
Cerny, E.	Univ. de Montreal
Chan, M. Y.	Univ. of Texas at Dallas
Chan, T.	U. C. Los Angeles
Chandran, S.	Univ. of Maryland
Chang, C. K.	Univ. of Ill. Chicago
Chang, D.	Univ. of Louisville
Chang, H.	Univ. of Miami
Chang, P. R.	Purdue Univ.
Chang, S.	Univ. of Maryland
Chellappa, R.	Univ. of Southern Cal.
Chen, S.	Univ. of North Carolina
Chennagiri, R. K.	Univ. of Southern Cal.
Cherkassy, V.	Univ. of Minnesota
Christian, F.	IBM Almaden Research Ctr.
Chung, M. J.	Michigan St. Univ.
Coffman, E. G.	AT&T Bell Labs
Conroy, J. M.	IDA - SRC
Cuny, J. E.	Univ. of Massachusetts
Curoe, J. E.	Mobil Corp.
Cybenko, G.	Tufts Univ.
Cypher, R.	Univ. of Washington
Darema, F.	IBM Hawthorne Res. Lab.
De Forcrand, P.	Cray Research Inc.
De Young, G. E.	Winchester, MA
Dehne, F.	Carleton Univ.
Dekel, E.	Univ. of Texas, Dallas
Despain, A. M.	U. C. Berkeley
Dey, P.	Univ. of Alabama Birm.
Diamond, M. D.	FMC Corp.
Dixit, V. V.	Univ. of Southern Cal.
Downes, E. H.	Reston, VA
Dyer, C. R.	Univ. of Wisconsin
Egecioglu, O.	U. C. Santa Barbara
El-Sharkawy, M.	Bucknell Univ.
Ellis, C.	Duke Univ.
Eltgroth, P.	Lawrence Livermore Lab.
Ercegovac, M. D.	U. C. Los Angeles
Eshaghian, M. M.	Univ. of Southern Cal.
Fahlman, S. E.	Carnegie-Mellon Univ.
Fang, Z.	Concurrent Comp. Corp.
Fatoohi, R.	NASA Ames Research Ctr.
Felten, E. W.	Cal. Inst. of Tech.
Feo, J.	Lawrence Livermore Lab.
Ferguson, D.	Boeing Computer Serv.
Ferreira, A. G.	Grenoble, France
Fiduccia, C. M.	G. E. Res. and Dev. Ctr.
Fier, J.	Ametek
Finkel, R.	Lexington, KY
Fornberg, B.	Exxon Research and Eng.
Foulser, D.	Yale Univ.

Fouts, M. J.	NASA Ames Research Ctr.
Franklin, M.	Washington Univ.
Friesen, D.	Texas A&M Univ.
Fujimoto, R. M.	Univ. of Utah
Gallopoulos, E.	Univ. of Illinois Urbana
Gao, G. R.	McGill Univ.
Gaushell, D. J.	Westin Power Consultants
Gear, C. W.	Univ. of Illinois
Ghosh, A. K.	Univ. of Iowa
Goel, A.	Ohio St. Univ.
Goel, P.	Univ. of Michigan
Gonzalez, T. F.	U. C. Santa Barbara
Gorin, A. L.	AT&T Bell Labs
Graf, K.	SRI International
Greenbaum, A.	New York Univ.
Greening, D. R.	Nashua, NH
Grefenstette, J. J.	Vanderbilt Univ.
Grimes, R.	Boeing Computer Serv.
Guha, S.	Univ. of Michigan
Gupta, A.	Carnegie-Mellon Univ.
Hac, A.	AT&T Bell Labs
Hachtel, G. D.	Univ. of Colordo
Hadlock, F. O.	Tennessee Tech. Univ.
Haghighi, M.	Bradley Univ.
Han, J.	Southern Illinois Univ.
Han, Y.	Univ. of Kentucky
Hanson, F. B.	Argonne National Lab.
Heath, M.	Oak Ridge National Lab
Hewitt, C. E.	MIT AI Lab.
Ho, C.	Yale Univ.
Hong, Y.	U. C. Riverside
Hume, D.	Tennessee Tech. Univ.
Hyatt, R.	Univ. of Alabama Birm.
Ibarra, O. H.	Univ. of Minnesota
Ipsen, I.	Yale Univ.
Iyengar, S. S.	Louisiana St. Univ.
JaJa, J.	Univ. of Maryland
Janakiram, V. K.	North Carolina St. Univ.
Janardan, R.	Univ. of Minnesota
Janicki, R.	McMaster Univ.
Jones, J.	Air Force Inst. of Tech.
Josephson, J. R.	Ohio St. Univ.
Kailath, T.	Stanford Univ.
Kamath, C.	Digital Equipment Corp.
Karabeg, D.	U. C. San Diego
Karp, A.	IBM Scientific Ctr.
Keller, R. M.	Quintus Computer Sys.
Kender, J. R.	Columbia Univ.
Kim, M. H.	Michigan St. Univ.
Kirkpatrick, S.	IBM Watson Research Ctr.
Kocay, W.	Univ. of Manitoba
Kodeih, M.	Rensselaer Poly. Inst.
Koenig, E. C.	Univ. of Wisconsin
Kosaraju, S. R.	Johns Hopkins Univ.
Kountanis, D.	Western Michigan Univ.
Kowalik, J. S.	Boeing Computer Serv.
Krishnamoorthy, M. S.	Rensselaer Poly. Inst.
Krishnamurthy, B.	Tektronix Inc.
Kruskal, C.	Univ. of Maryland
Kumar, D.	St. Univ. of New York
Kumar, V.	Univ. of Texas
Kung, H. T.	Carnegie Mellon Univ.
Kung, S.	Princeton Univ.
Kurtzberg, J.	IBM Watson Research Ctr.
Ladner, R. E.	Univ. of Washington
Lai, T.	Ohio St. Univ.
Lander, E.	Whitehead Inst.
Lazowska, E. D.	Univ. of Washington
LeBlanc, T. J.	Univ. of Rochester
Lee, C. S. G.	Purdue Univ.
Lee, D. L.	Ohio St. Univ.
Lee, S.	Cornell Univ.
Leiserson, C. E.	MIT
Leite, T. R.	IMSL
Lesser, V.	Univ. of Massachusetts
Leu, D.	Univ. of Houston
Lewis, J.	Boeing Computer Serv.
Li, H.	IBM Watson Research Ctr.
Li, X.	Univ. of Alberta
Liao, Y.	Digital Equipment Corp.
Lin, A.	Temple Univ.
Lin, W.	Pennsylvania St. Univ.
Liu, W.	North Carolina St. Univ.
Livingston, M. L.	Southern Illinois Univ.
Logananthara j, R.	Univ. of S. Louisianna
Logothetis, G.	Univ. of Florida
Lowrie, M. B.	Univ. of Illinois Urbana
Lubachevsky, B. D.	AT&T Bell Labs
Lustman, L. R.	NASA Ames Research Ctr.
Lynch, N. A.	MIT Lab for Computer Sci.
Makhoul, J.	BBN, Inc.
Mandell, D.	Los Alamos National Lab
Manhardt, P. D.	COMCO, Inc.
Mann, T.	DEC Systems Research Ctr.
Marsland, T. A.	Univ. of Alberta
Mattheyses, R. M.	G. E. Res. and Dev. Ctr.
McMillin, B. M.	Michigan St. Univ.
Mei, G.	North Carolina St. Univ.
Mesirov, J. P.	Thinking Machines Corp.
Messerschmitt, D. G.	U. C. Berkeley

Miller, G. L.	Univ. of Southern Cal.
Miller, R.	SUNY Buffalo
Miller, T. K.	North Carolina St. Univ.
Moceyunas, P. H.	Univ. of Colordo
Moldovan, D. I.	Univ. of Southern Cal.
Molloy, M. K.	Univ. of Texas
Montry, G.	Sandia National Labs
Morgan, A. P.	General Motors Res. Lab.
Morris, R. A.	U. S. Dept. of Defense
Morris, R. J. T.	AT&T Bell Labs
Mudge, T.	Univ. of Michigan
Mueller, R. A.	Colorado St. Univ.
Mukherjee, A.	Univ. of Cent. Florida
Nakazawa, S.	MARC Analytical Res.
Nassimi, D.	Univ. of Delaware
Natarajan, K. S.	IBM Watson Research Ctr.
Newman-Wolfe, R.	Univ. of Florida
Ng, E.	Oak Ridge Natl. Labs
Nichols, K. M.	AT&T Bell Labs
Noga, M. T.	Lockheed Palo Alto
Norton, A.	IBM Watson Research Ctr.
Nuttal, L. A.	Univ. of Utah
O'Hallaron, D. R.	G. E. Res. and Dev. Ctr.
O'Leary, D. P.	Univ. of Maryland
Oh, S. J.	Syracuse Univ.
Oliger, J.	Stanford Univ.
Omiecinski, E.	Georgia Inst. of Tech.
Ortega, J. M.	Univ. of Virginia
Otto, S. W.	Cal. Inst. of Tech.
Ougouag, A. M.	Univ. of Illinois Urbana
Ozguner, F.	Ohio St. Univ.
Pargas, R. P.	Clemson Univ.
Park, S.	North Carolina St. Univ.
Patrick, M. L.	Duke Univ.
Pawagi, S.	SUNY Stony Brook
Payne, T. H.	U. C. Riverside
Peng, S.	Univ. of Maryland BC
Perry, R. J.	Villanova Univ.
Peskin, R. L.	Rutgers Univ.
Peterson, J. L.	MCC
Pham, Q. T.	BNR, Canada
Pong, T. C.	Univ. of Minnesota
Prabhu, G. M.	Iowa St. Univ.
Pramanik, S.	Michigan St. Univ.
Quinn, M. J.	Univ. of New Hampshire
Raefsky, A.	Palo Alto, CA
Raghavendra, C. S.	Univ. of Southern Cal.
Ramachandran, V.	Univ. of Illinois Urbana
Ramakrishnan, I. V.	SUNY Stony Brook
Ramamoorthy, C. V.	U. C. Berkeley
Ramesh, K.	Univ. of Texas
Ranka, S.	Univ. of Minnesota
Rao, V. N.	Univ. of Texas
Reddy, S. M.	Univ. of Iowa
Reed, D. A.	Univ. of Illinois Urbana
Reeves, A. P.	Univ. of Illinois
Reynolds, P. F.	The Univ. of Virginia
Rivest, R. L.	MIT
Robertson, G. G.	Thinking Machines Corp.
Robinson, J.	IBM Watson Research Ctr.
Rodrigue, G.	U. C. Davis
Rogers, E. H.	Rensselaer Poly. Inst.
Ruiu, L. A.	Griffis AFB
Saad, Y.	Univ. of Illinois
Sahni, S.	Univ. of Minnesota
Saltz, J.	Yale Univ.
Sangiovanni-V, A.	U. C. Berkeley
Sanz, J. L. C.	IBM Almaden Research Ctr.
Sarrafzadeh, M.	Northwestern Univ.
Sawafzadeh, M.	Northwestern Univ.
Schaper, G. A.	Univ. of Central Florida
Schwetman, H.	MCC
Seager, M.	Lawrence Livermore Lab.
Sen, A.	Arizona St. Univ.
Sen, S.	Univ. of Alabama
Sengupta, A.	Univ. of South Carolina
Shaffer, P.	G. E. Res. and Dev. Ctr.
Shannon, G. E.	Purdue Univ.
Shields, M. W.	Univ. of Kent, GB
Shokooh, A.	Tennessee Tech. Univ.
Shyu, W. C. H.	Old Dominion Univ.
Simmes, S. D.	Science Appl. Inc.
Simon, H. D.	NASA Ames Research Ctr.
Sinclair, B.	Rice Univ.
Singhal, M.	Ohio St. Univ.
Somani, A. K.	Univ. of Washington
Sorensen, D.	Argonne National Lab
Sridhar, M. A.	Univ. of S. Carolina
Srimani, P. K.	Southern Illinois Univ.
Srinidhi, H. N.	Univ. of Cent. Florida
Starzyk, J.	Ohio Univ.
Stiles, G. S.	Utah St. Univ.
Stojmenovic, I.	Univ. of Miami
Stolfo, S. J.	Columbia Univ.
Stotts, D.	Univ. of Maryland
Stout, Q. F.	Univ. of Michigan
Strader, R.	Texas A&M Univ.
Strong, R.	IBM Almaden Res. Ctr.
Stunkel, C. B.	Univ. of Illinois Urbana
Suk, M.	Syracuse Univ.

Suzuki, I.	Univ. of Wisconsin
Swarztrauber, P.	NCAR
Swisshelm, J. M.	NASA Ames Research Ctr.
Tang, W.	Univ. of Waterloo
Tanik, M. M.	Southern Methodist Univ.
Tao, L.	Univ. of Pennsylvania
Tham, K. Y.	Mentor Graphics Corp.
Thomborson, C. D.	Univ. of Minn. Duluth
Tokuta, A. O.	Univ. of South Florida
Tomboulian, S. J.	NASA Langley Res. Ctr.
Tong, Z.	Univ. of Minnesota
Tsin, Y. H.	Univ. of Windsor
Turner, C. J.	Science Appl. Inc.
Tymann, P. T.	SUNY Oswego
Ursein, A.	Los Angeles, CA
Van Loan, C.	Cornell Univ.
Varman, P.	Rice Univ.
Venkatesan, S.	Univ. of Minnesota
Vernon, M. K.	Univ. of Wisconsin
Vishwanathan, S.	Univ. of South Carolina
Visvanathan, V.	AT&T Bell Labs
Wagar, B.	Univ. of Michigan
Wah, B. W.	Purdue Univ.
Waid, B.	Glen Ellyn, IL
Wainer, M.	Southern Illinois Univ.
Walicki, J.	Colorado St. Univ.
Walton, S.	Cal. St. Northridge
Wang, C. Y.	Southern Illinois Univ.
Wang, C.	Cal. St. Sacramento
Waramahaputi, J.	Univ. of S. Louisiana
White, W.	Ohio St. Univ.
Willebeek-LeMair, M.	Cornell Univ.
Winter, C. L.	Science Appl. Inc.
Witten, M.	Univ. of Louisville
Wu, C.	Univ. of Col. Denver
Wunderlich, M. C.	U. S. Dept. of Defense
Young-Myers, H.	Columbia, MD
Yu, K.	Univ. of Alabama Birm.
Zargham, M.	Southern Illinois Univ.
Zeigler, B. P.	Univ. of Arizona
Zeigler, G. M.	Hewlett Packard
Zhang, C. N.	North Carolina A&T Univ.
Zhang, H.	Temple Univ.
Zhao, F.	MIT
Zyda, M. J.	Naval Postgrad. School

AUTHOR INDEX

TABLE OF CONTENTS

A GENERAL APPROACH TO MAPPING OF PARALLEL COMPUTATIONS UPON MULTIPROCESSOR ARCHITECTURES

S. J. Kim and J. C. Browne

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188

Abstract -- This paper defines and describes a broadly applicable approach to mapping of parallel computations upon multiprocessors, and briefly sketches the related mapping algorithms. The approach begins with a graph representation of a parallel computation and first generates a reduced graph by merging nodes with high internode communication cost through iterative use of a critical path algorithm. This graph is then mapped to a graphical representation of a multiprocessor architecture by the mapping algorithms. These algorithms attempt to minimize the total execution time including both computation and communication times. The algorithms, while they are heuristic rather than true optimal algorithms, are shown to yield excellent results in example applications and have modest execution costs.

1. INTRODUCTION

This paper defines and describes a broadly applicable approach to mapping of parallel computation structures (consisting of mutually dependent schedulable units of computations) upon MIMD multiprocessor architectures, and then sketches the related heuristic mapping algorithms. It also gives examples of the results obtained by application of the algorithms to different types of parallel computation structures and different multiprocessor architectures. The algorithms are based upon the mapping of a graphical representation of a parallel computation structure [4, 5] upon a graphical representation of a multiprocessor architecture. In fact, we consider a series of transformations and mappings between a computation graph and an architecture graph as illustrated in Fig. 1-1.

The algorithms are described informally herein but complete formal definitions can be found in [17]. The algorithms apply to a broad class of graphs which can be derived from programs with various types of loop structures, and to a wide class of architectures. The algorithms attempt to minimize the total execution time (computation time and communication time) of the parallel computation. Reduction of execution time is attained mainly by reduction of communication time by merging of schedulable units of computation. The first step of each of the algorithms is the reduction of the computation graph to a virtual architecture graph through transformations determined by iterative application of a critical path algorithm. This virtual architecture graph is then either transformed into another virtual architecture graph or mapped onto an abstracted graphical representation of a multiprocessor architecture (called physical architecture graph). Note that a computation graph is assumed to have one root node and one leaf node without loss of generality.

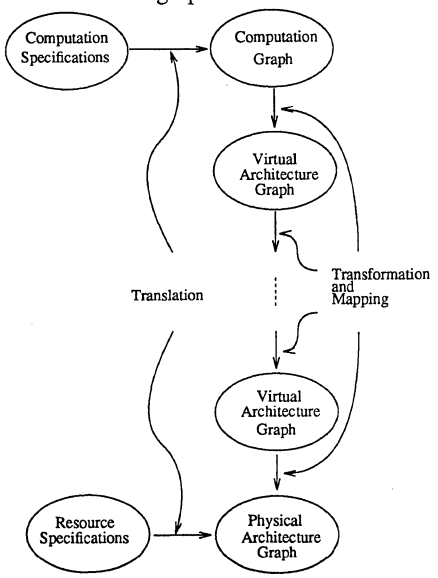


Figure 1-1 General Overview of Our Approach

The algorithms are heuristics with modest execution cost. True optimal algorithms for the scheduling problem as stated in Section 2 are known to be *NP−complete*. Three applications are given: mapping of the Sieve of Eratosthenes to an Intel iPSC/5 [15], mapping of a Gaussian (forward) elimination to a Sequent Balance and mapping of a molecular physics code to an emulated Intel iPSC/5 configuration with a mixture of fast and slow processors and communication channels. The results of the applications are surprisingly good. Near optimal total execution times are coupled with near minimal resource requirements and good workload balancing.

This paper is organized as follows: After giving the problem statement in Section 2, we briefly review previous work in Section 3. Then, in Section 4, after discussing our approach and the models for computation and architecture graphs, we explain mapping algorithms based on linear clusters. Section 5 gives a brief summary of performance results and Section 6 summarizes the status of the research.

2. PROBLEM STATEMENT

A parallel computation can be represented by a direct acyclic graph $G_C = (N_C, E_C)$, where $N_C = \{n_1, n_2, \cdots, n_l\}$ is a set of schedulable units of computation to be executed, and E_C specifies scheduling constraints and data dependencies defined on N_C. A multiprocessor architecture can be represented by an undirected graph $G_P = (N_P, E_P)$, where $N_P = \{p_1, p_2, \cdots, p_m\}$ is a set of processors, and E_P specifies interconnection topology among the processors. The basic problem is to find a mapping of G_C onto G_P which minimizes schedule length (or makespan) defined as:

$$\max_{1 \le k \le s} \sum_{i,j \in \phi_k} (comp_i + comm_{ij}),$$

where $\phi = \{\phi_1, \phi_2, \ldots, \phi_s\}$ represents a set of paths from the root node to the leaf node in G_C, node n_j (assigned to processor $p_y \in N_P$ $(1 \le y \le m)$) is a direct descendant of node n_i (assigned to processor $p_x \in N_P$ $(1 \le x \le m)$) in G_C, $comp_i$ is computation time of n_i, and $comm_{ij}$ is communication time from n_i to n_j ($comm_{ij} = 0$, if $p_x = p_y$ or n_i has no direct descendants).

An optimal schedule is one which meets the criteria of the minimum schedule length for a single parallel computation structure or the maximum total throughput for a set of

simultaneously executing parallel computation structures. It must integrate scheduling of computations and dependency relations to resources. An approach which integrates consideration of all the interacting factors is one which maps a computation graph defining the computation structure (including the resource requirements for execution of each element of the computation structure) onto an architecture graph which defines the capability and capacity of the resource set of the execution environment. From here on we use the term *task* and *schedulable units of computation*, which corresponds to a node in a computation graph, interchangeably.

3. PREVIOUS WORK

The problem of optimal scheduling (as defined in Section 2) of parallel computations upon multiprocessor architectures has received generous attention in the literature. Algorithms which yield true optimal solutions in the absence of resource constraints are well known to be *NP−complete* [12, 21]. In fact, it is proven by Kim [17] that the other interesting scheduling problems are also *NP−complete* or worse in computation complexity.

There have been many heuristic algorithms proposed in the past. Previous approaches have focused mainly on the development of specific mapping strategies for particular multiprocessor architectures. Some attempt to take advantage of the unique hardware characteristics such as interconnection topologies of multiprocessor architectures under consideration. Since each strategy is usually an ad-hoc scheme, it is in most cases applicable to some limited class of multiprocessor architectures (e.g., tightly-coupled homogeneous architectures [2], loosely-coupled homogeneous architectures [23], loosely-coupled heterogeneous architectures [11], or multicomputers connected in point-to-point fashion [6]).

Various simplifying assumptions are common. For example, Bokhari [3] studies the assignment of tasks to processors with the restriction that the number of tasks should be less than or equal to the number of processors. Shen and Tsai [20] propose a graph matching approach for solving task assignment to processors, but ignore dependency relations among tasks. Some approaches have limited scheduling objectives; they find the best schedule with respect to either the total computation time [13] or interprocessor communication time [14]. Other approaches are interested in balancing the workload of the total multiprocessor architecture [10, 22].

In most scheduling strategies for tightly-coupled architectures, specific interconnection networks such as the Butterfly switch, the Omega network, the SW-Banyan network or a composition of them [19] are assumed. On the other hand, most research has not taken into account scheduling constraints, resource limitations, and/or the current workloads of processors. It is frequently assumed that each processor is identical (i.e., all have the same processing speed, equal number of communication channels, and identical memory capacity). Finally, while most scheduling strategies make heavy use of busy-waiting as a synchronization mechanism, there is little attempt to reduce or avoid using it.

All in all, there are a myriad of multiprocessor scheduling strategies which can be applied to specific multiprocessor architectures. On the other hand, there is little research which attempts an integrated approach to multiprocessor scheduling which could be applicable to various multiprocessor architectures regardless of underlying architectural characteristics.

4. APPROACH AND ALGORITHMS

4.1. Approach

One of the contributions of this paper is to propose algorithms based on *linear clustering*. A linear cluster is a connected subgraph of a computation graph which is in the form of a linear list of schedulable units of computation. Linear clustering is an effectual heuristic to compromise between two conflicting goals of multiprocessor scheduling, minimization of interprocessor communication and maximization of potential parallelism, and to satisfy the other goals, throughput enhancement and workload balance, relatively well. The underlying idea of linear clustering is that the schedulable units of computation that are sequentially dependent on each other are to be assigned to one processor, while those that are mutually independent are to be allocated to separate processors. We select linear clusters on the basis of total execution time on an architecture with a processor for each node of the graph and a distinct communication channel per each edge of the graph. The critical restriction of linear clustering is that it expects a computation graph to be acyclic. To minimize this restriction, we identify cases in which cyclic computation graphs can be transformed into acyclic graphs in a straightforward manner [17].

A computation graph is transformed into a *virtual architecture graph* (*VAG*) by linear clustering. The *VAG* in fact represents an optimal multiprocessor architecture for the computation graph. The optimal architecture provides one processor to every linear cluster so that mutually independent tasks belonging to different linear clusters can be executed in parallel as long as possible. Furthermore, direct communication channels are always available for any adjacent linear clusters in the optimal architecture.

The *VAG* may be transformed into another *VAG* by *merging* two or more linear clusters into one cluster. Two linear clusters K_1 and K_2 are combined into one if K_2 may start only after K_1 finishes or may be executed only while K_1 is idle. It contributes to further balancing the workload of processors, and further reducing the amount of resources to be utilized and interprocessor communication overhead.

After constructing a *VAG* which represents the optimal multiprocessor architecture for a given computation graph, we then find an optimal mapping of the *VAG* onto a *physical architecture graph* (*PAG*) which represents the target architecture. This mapping is called a *physical mapping* as it is the final mapping of a computation graph onto a real physical multiprocessor architecture. We develop *homogeneous* and *heterogeneous* mapping algorithms for homogeneous and heterogeneous architectures, respectively.

These algorithms rely on not only local information but also on limited global information. The key issue is how to reduce the mapping complexity while sacrificing as little optimality as possible. A *dominant request tree* is a maximal spanning tree of a *VAG*. It provides limited global information on the *VAG* such as the mapping order of the nodes and the edges whose adjacency should be maintained. Both mapping algorithms utilize dominant request trees, but take quite different approaches to mapping the trees onto *PAG*'s. Most importantly, in the case of homogeneous mappings, the trees are directly mapped onto *PAG*'s. On the other hand, in the case of heterogeneous mappings, they are mapped onto dominant service trees. A *dominant service tree* is a maximal spanning tree of a *PAG*. For heterogeneous mappings, one of the important issues is how to identify and utilize resources with high performance. A dominant service tree provides such information.

4.2. Model of Computation

Browne [5] proposes a directed graph as a representation basis of a parallel computation, in which the nodes represent the bindings of operations to data and the edges represent

dependency relations between schedulable units of computation executed at the nodes. Our computation graph model is a triple $(G_c, f_c^{comp}, f_c^{comm})$, whose first component $G_c = (N_c, E_c)$ specifies a parallel computation. Computation graph G_c is a directed acyclic graph and defined as follows:

(i) A node set $N_c = \{n_1, n_2, \cdots, n_l\}$;

(ii) An edge set $E_c = \{e_1, e_2, \cdots, e_t\}$, where any given edge $e_p = (n_i, n_j)$ is directed from node n_i to node n_j.

To be specific, graph G_c defines computation steps by the nodes and sequencing among the steps by the edges. The remaining components provide information necessary for mapping the computation graph onto a target architecture. The second component f_c^{comp} is a function which maps each node in N_c onto a positive integer which is the expected computation time used by the schedulable unit of computation corresponding to the node. The next function f_c^{comm} maps each edge (n_i, n_j) in E_c onto a nonnegative integer which is the expected amount of internode communication from node n_i to node n_j. For example, if $f_c^{comm}(e_p) = N_{bytes}$ for $e_p = (n_i, n_j)$, then the total length of messages sent from n_i to n_j is N_{bytes} bytes.

Our computation graph is a restricted model in a couple of ways. The critical restriction that makes the model inappropriate for representing some parallel computations is that a set of edges entering and leaving a given node may not be joined by *or* conditions. The other restriction is that computation graphs are required to be static; neither new nodes nor new edges can be created during runtime. The main reason for these restrictions is to avoid ambiguity in determining the computation and communication requirements of the nodes and edges in a computation graph.

The model for architecture graphs provides a representation basis for the structural description of multiprocessor architectures. We consider three types of resources: processors, communication channels and memory. Our architecture graph model is also a triple $(G_a, f_a^{comp}, f_a^{comm})$, whose first component $G_a = (N_a, E_a)$ is an undirected graph defined as follows:

(i) An architecture node set $N_a = \{an_1, an_2, \cdots, an_l\}$;

(ii) An architecture edge set $E_a = \{ae_1, ae_2, \cdots, ae_t\}$, where any architecture edge $ae_p = (an_i, an_j)$ is undirected.

In an architecture graph, an architecture node represents a processor as well as a memory module, and an architecture edge represents a communication channel between two processors. The second component f_a^{comp} is a function which maps each architecture node in N_a onto a pair of positive integers which denote the level of computing power of a processor relative to the others in the architecture and the current local memory size. A common global memory may be specified by a *dummy* architecture node which is fully-connected with the other architecture nodes. The next function f_a^{comm} maps an architecture edge (an_i, an_j) in E_a onto a positive integer which represents the bandwidth of communication channel from an_i to an_j and vice versa.

It is assumed that an architecture graph is static; the resource configuration of a physical multiprocessor architecture will not be changed dynamically during runtime. Moreover, it maintains the exact current status of the architecture. The status includes the information on which processors are currently active/inactive, which communication channels are currently available and what is the current memory capacity available in each processor.

4.3. Mapping Based on Linear Clusters

Clustering techniques have been used in a variety of areas in computer science [1, 7]. In this section, we propose a new mapping technique based on linear clustering and linear cluster merging. After discussing linear clustering and merging, we explain how to iteratively refine linear clusters (if necessary) for the minimization of schedule length.

4.3.1. Linear Clustering

Linear clustering is a fundamental idea of our mapping algorithms discussed in Section 4.4. A cluster of $G_C = (N_C, E_C)$ is called a *linear cluster* K if it satisfies the following conditions:

- K is nonempty;
- K is a connected subgraph of G_C;
- Both indegree and outdegree of every node in K is less than or equal to 1.

Linear clustering is a special case of general clustering in that a linear cluster is a degenerate tree in which each node has at most one direct ancestor and/or one direct descendant, while a cluster, in general, is an arbitrary graph.

The following algorithm *LinearCluster* illustrates how to identify linear clusters:

```
LinearCluster (G, K)
/* G is a (cycle-free) computation graph. */
/* K is a set of linear clusters. */
Begin
  Let K = ∅;
  Find a longest path P from the root to a leaf node in G;
  During traversing path P backward
     from the leaf to the root node,
     cut all the incoming and outgoing edges
     except the one belonging to P;
  For each connected subgraph S of G,
     If both indegree and outdegree of each node in S
       is less than or equal to 1,
     Then
        K = K ∪ S
     Else Do
          LinearCluster (S, K');
          K = K ∪ K';
     End Do;
End LinearCluster.
```

A path $(n_1, n_2, \ldots, n_l)$ of graph $G_C = (N_C, E_C)$ such that $n_i \in N_C$ and $(n_i, n_{(i+1)}) \in E_C$ is considered the longest path if it maximizes the following function:

$$\sum_{i=1}^{l-1}(\omega_1 \cdot T_{comp_i} + (1-\omega_1)\cdot(\omega_2 \cdot T_{comm_{(i,i+1)}} + (1-\omega_2)\cdot \sum_{\substack{j \in N_{adj}^{(i,i+1)} \\ j \neq i+1}} T_{comm_{(i,j)}})) + \omega_1 \cdot T_{comp_l},$$

where T_{comp_k} is the computation time of node n_k $(1 \le k \le l)$, $T_{comm_{st}}$ is the communication time of node n_s with an adjacent node n_t $(1 \le s < l$ and $1 < t \le l)$, N_{adj}^t denotes a set of nodes adjacent to n_t $(1 < t \le l)$, and both ω_1 and ω_2 are normalization factors.

4.3.2. Linear Cluster Merging

In this section, we investigate a means to merge two or more linear clusters into one without affecting potential parallelism existing in a computation graph. It may contribute to further balancing the workload of processors. It may also contribute further reducing the amount of resources to be utilized and interprocessor communication overhead.

The level numbers may be used to identify potential parallelism [18] in a computation graph if defined as follows:

level $(T) = 1$ if T is a root node;

= [max(*level* (A) for each direct ancestor A of T)]

+ 1, otherwise.

Then, the same level number implies mutual independence. To be more specific, if a group of tasks have the same level number, they are mutually independent and may be simultaneously executable.

In order to define conditions for merging linear clusters, let L_i represent a set of level numbers assigned to tasks in linear cluster K_i. Two linear clusters K_i and K_j are said to be *sequentially strong-dependent* if they satisfy the following conditions:

1) $L_i \cap L_j = \varnothing$;
2) The trailer node of linear cluster K_i precedes the header node of linear cluster K_j.

Two linear clusters K_i and K_j are said to be *mutually strong-dependent* if they satisfy the following conditions:

1) $L_i \cap L_j = \varnothing$;
2) For two tasks T_1 and T_2 in K_i, T_1 is a direct ancestor of T_2, where the former is one of direct ancestors of the header node of K_j and has the largest level number among the direct ancestors, and the latter is one of direct descendants of the trailer node of K_j and has the smallest level number among the direct descendants.

If a pair of linear clusters satisfy any of the merging conditions, they can be merged into one cluster without affecting the execution time of the computation.

4.3.3. Iterative Refinement of Linear Cluster

In the previous sections, we discussed how to transform a computation graph G_C into a virtual architecture graph by linear clustering and merging. It is expected that a linear cluster consisting of schedulable units of computation on the critical path of G_C takes the longest time to finish in the *VAG* in most cases. In this case, we can make use of the *VAG* for the mapping onto a physical architecture graph without any modification. This may not be true if the computation graph has extremely heavy communication requirements on edges not on the initial critical path. If that is the case, we may need to iteratively refine linear clusters in the *VAG* so that we can further reduce the total length of schedule prior to mapping. It consists of two steps:

- Linear cluster labeling;
- Linear cluster refinement.

During labeling step, we label edges in a computation graph $G_C = (N_C, E_C)$. The level number $level_{edge}$ of edge $e_{ij} = (n_i, n_j)$ may be defined as follows:

$level_{edge}(e_{ij}) = \omega \cdot comp_j + (1-\omega) \cdot comm_{ij} + level_{node}(n_j)$,

where $level_{node}(n_j)$ is the level number of node n_j, $comp_j$ and $comm_{ij}$ are computation time of n_j and communication time from n_i to n_j, respectively, and ω is a normalization factor. Note that $level_{node}(n_j)$ is defined as $\max_{n_k \in D_j}(level_{edge}(e_{jk}))$ where D_j is a set of direct descendants of node n_j. These edge labels allow us to identify the longest path to be considered for the minimization of the total schedule length in a *VAG*.

After linear cluster labeling, we can determine if there are paths through a *VAG*, each of whose length is longer than the total computation time of a linear cluster corresponding to the critical path of the original computation graph G_C. If there exist such paths, we modify the current set of linear clusters in order to further reduce the total schedule length through iterative refinements of them.

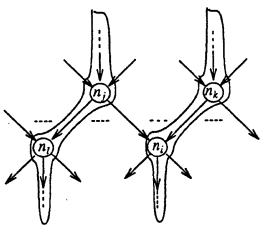


Figure 4-1 Linear Clusters

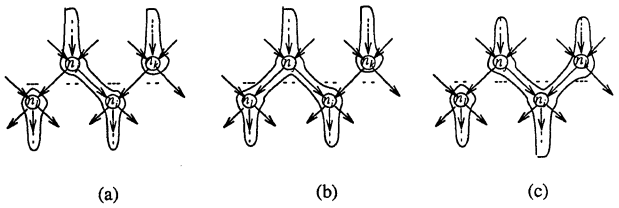


Figure 4-2 Possible Refinements of Linear Clusters

In Fig. 4-1, let us assume that a new longest path is passing through nodes n_j and n_i, i.e., the new longest path is $(\cdots, n_j, n_i, \cdots)$. The basic idea of linear cluster refinement is to locate a *cut* edge (n_j, n_i) on the longest path and to reduce the length by merging nodes n_i and n_j (belonging to separate linear clusters) into one. After the two nodes n_j and n_i are merged, linear clusters shown in Fig. 4-1 can be refined as shown in Fig. 4-2. In Fig. 4-2-a, we merge n_i and n_j into one cluster, and cut the edges like (n_j, n_l) and (n_k, n_i) so that all the clusters remain as linear clusters. In Fig. 4-2-b and Fig. 4-2-c, however, we merge them, but leave one of the edges uncut while we cut the other edge. This type of refinement may force us to sacrifice some potential parallelism since two or more nodes (e.g., n_l and n_i in Fig. 4-2-b, n_j and n_k in Fig. 4-2-c) executable in parallel are to be assigned to the same cluster. Nonetheless, it is worthwhile to merge two linear clusters in this way if internode communication overhead from n_j to n_l is larger than the schedule extension caused by sequential execution of nodes (e.g., n_l and n_i in Fig. 4-2-b, n_j and n_k in Fig. 4-2-c).

4.4. Mapping Algorithms

The subject of this section is how to map a *VAG* onto a *PAG*. The important goal of our proposed algorithms is to compromise between two extreme approaches [8, 18] by reducing the complexity of the mapping algorithms while sacrificing their optimality as little as possible. For physical mapping, we need to take into consideration as much global information as possible during mapping.

4.4.1. Dominant Request Tree

The basic idea of our algorithm is to find a subgraph isomorphism [12] from a *VAG* to a *PAG* which minimizes the total schedule length and satisfies given scheduling constraints. We can easily show that the subgraph isomorphism problem is *NP-complete*, making use of the fact that the Undirected Hamilton Circuit problem is *NP-complete*. This fact forces us to rely on heuristics. We map each node of a *VAG* one by one in a sequential order. The key issue is then how to determine the mapping order which leads to the minimization of the schedule length. For this purpose, we propose another transformation of a *VAG* into a tree called *Dominant Request Tree* (*DRT*). This transformation can be done independently of the target architecture (i.e., whether it is homogeneous or heterogeneous).

A *DRT* is a maximal spanning tree of a *VAG*. We construct the *DRT* starting from a node called the *Most Dominant Node* (*MDN*) rather than starting from an arbitrary node in the *VAG*. The *MDN* is that node n which maximizes the cost function defined as:

$$\omega \cdot T_{comp} + (1-\omega) \cdot T_{comm},$$

where T_{comp} is the computation time of n, T_{comm} is the total communication time of n with its adjacent node(s), and ω is a normalization factor. The *MDN* is considered to be the most important node in the *VAG* in the sense that it represents a linear cluster which includes all tasks on the critical path in a given computation graph. Since it is usually the case that the *MDN* requires the largest weighted sum of computation and communication times among nodes in the *VAG*, we would better assign the *MDN* to the most appropriate processor in a *PAG*.

Starting from the *MDN* of a *DRT*, we select a node with the highest binding power among unassigned nodes incident upon any already assigned node until all the nodes in the *DRT* are selected. The binding power of node n_i with respect to an adjacent node n_j is determined by:

$$\omega_1 \cdot T_{comp_j} + (1-\omega_1) \cdot (\omega_2 \cdot T_{comm_{ij}} + (1-\omega_2) \cdot \sum_{\substack{k \in A_j \\ k \neq i}} T_{comp_{jk}}),$$

where T_{comp_j} is the computation time of n_j, $T_{comm_{ij}}$ is the communication time of node n_i with node n_j, and A_j represents a set of nodes adjacent to n_j. ω_1 and ω_2 are again normalization factors. A *DRT* of a *VAG* has two types of edges: the primary and secondary edges. The former are edges belonging to the *DRT*, while the latter are edges belonging to the *VAG* but not to the *DRT*. Note that the order in which each cluster is included in the *DRT* determines the *priority list* L.

4.4.2. Homogeneous Mapping

The goal of homogeneous mapping is to find a subgraph in a *PAG* to which a *DRT* of a *VAG* is isomorphic, relying on various heuristics like connectivity, exclusion, perturbation, foster mapping, and restricted pairwise exchange. The basic approach of the homogeneous mapping is to try to maintain adjacency of each node in the *DRT* with its neighbors as far as possible; whenever there is a direct primary edge from cluster K_1 to cluster K_2, we choose processor P_{K_2} which has a direct channel from P_{K_1}. Note that P_K and K_{da} denote a processor onto which cluster K is to be mapped and the direct ancestor of K, respectively.

Each node of the *VAG* is assigned to a processor in the order determined during transforming the *VAG* into the *DRT*. For each cluster K in the order of the priority list L, if there are at least two clusters which form full-connectivity with K, we first apply *connectivity mapping*. This heuristic attempts to maintain full-connectivity among clusters during mapping. If it is not successful to maintain the connectivity or there exist no clusters which form full-connectivity with K, then we try to assign K to a free processor in *PAG* adjacent to $P_{K_{da}}$. During this mapping, we apply *exclusion mapping* to exclude processors in *PAG* which might be crucial to other clusters yet to be assigned.

Next, we consider the case that $P_{K_{da}}$ has no more free adjacent processors. Then, K may be mapped onto a processor which is not adjacent to $P_{K_{da}}$. For this case, we provide two heuristics: perturbation and foster mappings. In both heuristics, we first choose a processor which has the most appropriate number of channels among currently unassigned processors. If there is more than one, we choose the one which is the nearest to $P_{K_{da}}$. Those unassigned processors should be adjacent to at least one processor to which a cluster has already been assigned.

In *perturbation mapping*, we attempt to preempt a linear cluster which has already been assigned to a processor adjacent to $P_{K_{da}}$, and to assign K to the processor. There are two possible cases that a linear cluster may be preempted after being assigned to a processor. First, an adjacent processor (say, $P_{K_{adj}}$) of $P_{K_{da}}$ might be assigned to cluster K_{adj} which is not in fact adjacent to cluster K_{da} in the *VAG*. The other possible case is that all the clusters assigned to adjacent processors of $P_{K_{da}}$ are in fact neighbors of K_{da}, but K_{adj} might have less communication overhead with K_{da} than K in the *VAG*.

As long as perturbation mapping does not make any improvement, it is not possible to maintain adjacency using a primary edge for this particular mapping. That is, cluster K can not communicate directly with cluster K_{da}. In order to lessen the effect of the indirect communication, we first check whether there is another cluster adjacent to K through a primary edge which has already been assigned to a processor. If there is more than one, we choose a cluster K_{fa} which has the highest binding power (other than K_{da}) with K. After assuming K_{fa} as a direct ancestor of K, we reiterate the same mapping procedure mentioned above (i.e., finding the best mapping from processor $P_{K_{fa}}$). We call such a mapping *foster mapping*. The only difference is that K_{fa} is now assumed to be the direct ancestor of K for K_{da} in the *VAG*. If there does not exist such a primary edge, utilizing the secondary edges, we repeat the same procedure as we do for the primary edge.

Since the previous heuristics do not guarantee an optimal mapping, we try to further improve the result by applying *restricted pairwise exchange*; we do allow random pairwise exchange of clusters to which specific codes have been assigned during mapping [17]. Note that we keep track of such codes based on how the clusters have been assigned during mapping.

4.4.3. Heterogeneous Mapping

Heterogeneous mapping is a mapping of computation graphs onto architecture graphs which represent heterogeneous multiprocessors. For heterogeneous mappings, it is important to utilize resources with high performance as far as possible so that the total schedule length can be minimized and the workload balance can be achieved. We first need to distinguish resources with higher performance from those with lower performance. A *Dominant Service Tree* (*DST*) provides a limited amount of global information on resources in a heterogeneous multiprocessor lest our mapping algorithms become totally greedy based on local information. We can construct a *DST* by utilizing a maximal spanning tree algorithm. This may be considered as a transformation of a *PAG* into another *PAG*. In a sense, the transformation can be regarded as prescanning of architecture graphs prior to physical mapping. During the scanning, we collect information like which processors have more computing power and which communication channels have more bandwidth than others.

After the transformation of a *PAG* into a *DST*, the scheduling problems for heterogeneous multiprocessor architectures turn into the tree-to-tree mapping problems. The edges in the *PAG* are to be divided into two different types: the primary and secondary edges. Analogous to a *DRT*, the edges belonging to the *DST* are called the primary edges, while the edges belonging to the *PAG* but not to the *DST* are called the secondary edges. The main goal of heterogeneous mapping is to identify a mapping which maintains adjacency of the primary edges of the *VAG* with those of the *PAG*. When there are no primary edges available, however, we util-

ize secondary edges of the *PAG* during mapping. Specific scheduling constraints (e.g., available local memory size) are also to be applied on the fly during the mapping.

Since it is still an *NP−complete* problem to find an optimal mapping from one tree to another, the issue is how to develop efficient heuristic mapping algorithms between a *DRT* and a *DST*. We exploit sequential mapping order of nodes determined during constructing a *DRT*, and so-called *node information* [17] as a means to avoid exhaustive matching between two trees.

5. APPLICATIONS

The applications described here cover regular (Sieve of Eratosthenes, Gaussian elimination) and irregular (molecular physics code) computation graphs, and partitioned (Intel iPSC) and shared memory (Sequent Balance) multiprocessor architectures.

5.1 Mapping of Sieve of Eratosthenes to an Intel iPSC

We seek here decrease of the communication time component of the total execution time. The computation graph for the algorithm is shown in Fig. 5-1. The *VAG* for the computation graph is shown in Fig. 5-2. Fig. 5-3 shows the improvement in total execution time obtained by application of the algorithm together with the lower bound of total execution time for this execution environment.

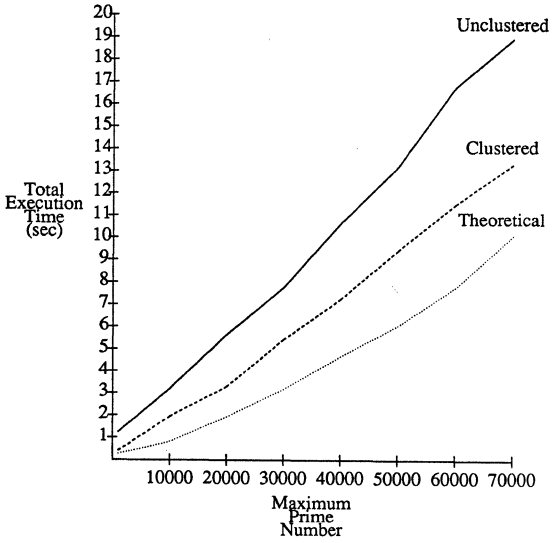


Figure 5-3 Comparison of Total Execution Times

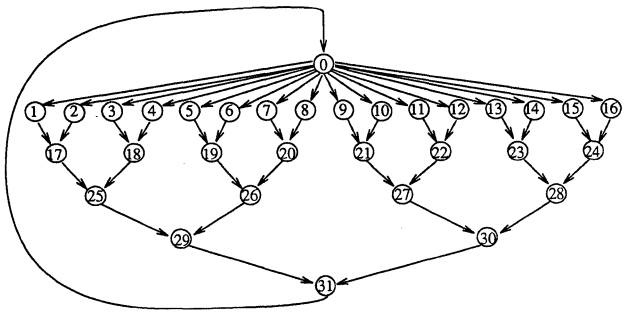


Figure 5-1 Computation Graph

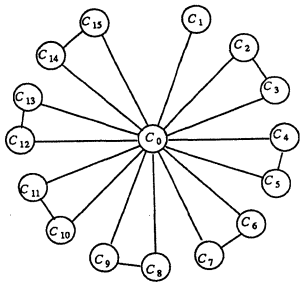


Figure 5-2 Virtual Architecture Graph

5.2. Gaussian Elimination on a Sequent Balance

The principal benefit to be obtained from application of one algorithm to scheduling for a shared memory multiprocessor is decrease in overhead without loss of parallelism caused by an optimal selection of schedulable units of computation. The computation graph for forward elimination is shown in Fig. 5-4. Each node $A_{k,l}$ in Fig. 5-4 represents the row operation to force $A_{k,l}$ to zero. The *VAG* is shown in Fig. 5-5. The saving in overhead is shown in Fig. 5-6 for 9 processors across a range of array sizes after linear clustering and merging. The gain is substantial (~15%-20%) for larger array sizes.

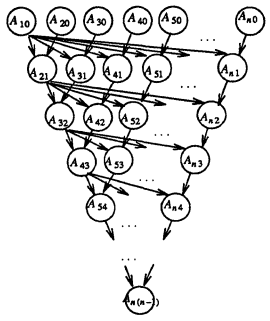


Figure 5-4 Computation Graph

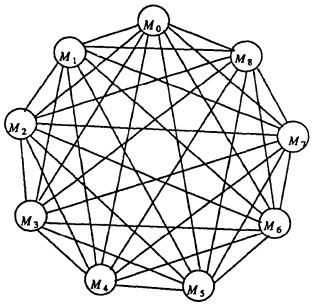


Figure 5-5 Virtual Architecture Graph

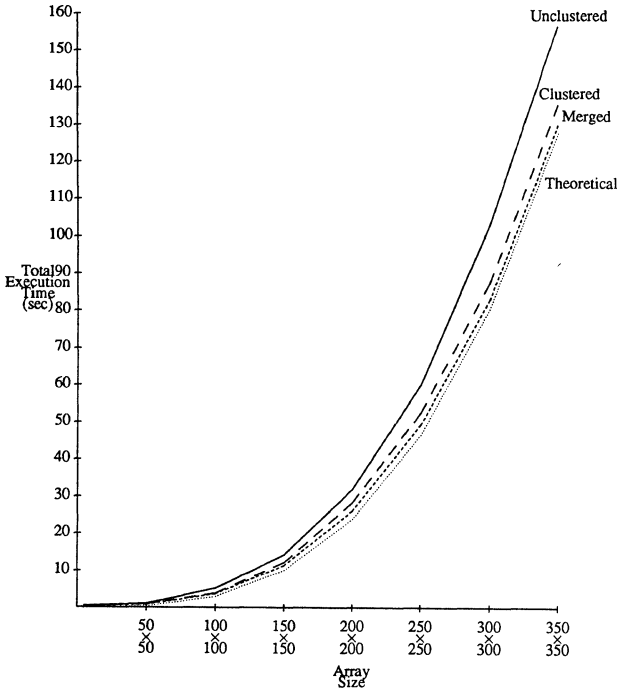


Figure 5-6 Comparison of Total Execution Times using 9 Processors

5.3. Modified Molecular Dynamics Code on a "Heterogeneous" Intel iPSC

The effects to be studied here are those of an irregular computation graph on a heterogeneous architecture. The computation graph is shown in Fig. 5-7 and the *VAG* in Fig. 5-8. In order to obtain the effect of a heterogeneous multiprocessor, we assume that 50% of processors and 20% of communication channels are twice as fast as real ones by setting computation times of nodes and communication times of edges in the *VAG* to ½ of their actual values if they are assigned to faster processors or channels, respectively. Fig. 5-9 compares total execution times for four cases (**x**,**y**) where **x** = (homogeneous, heterogeneous) and **y** = (measured, theoretical). It is not surprising that the improvement in execution time is greater for the heterogeneous architecture than for the homogeneous architecture since the clusters with greater resource requirements can be assigned to faster processors and channels of the heterogeneous one as far as possible.

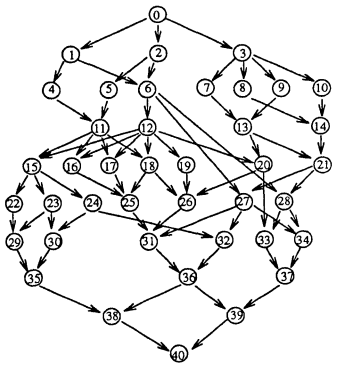


Figure 5-7 Computation Graph

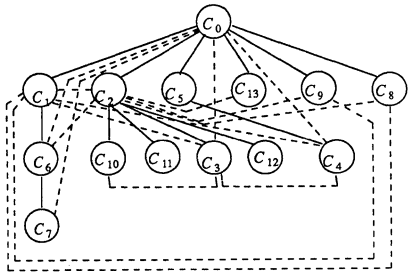


Figure 5-8 Virtual Architecture Graph

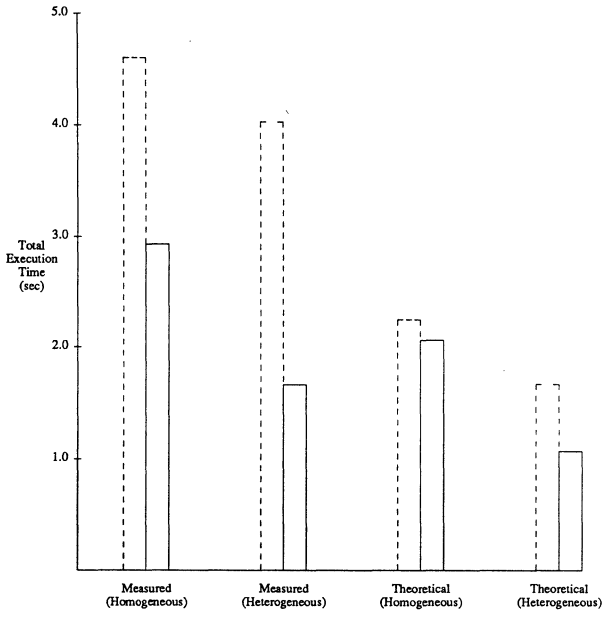


Figure 5-9 Comparison of Total Execution Times (Unclustered vs Clustered)

6. SUMMARY

The conceptually simple and computationally tractable heuristics based on linear clustering have been found in application to be effective and, so far as can be judged by the limited sample of applications, robust.

Future work will include test of a large number of applications, incorporation of various scheduling constraints into the model, and analytic definition of the class of graphs where the heuristics yield optimal schedules.

7. ACKNOWLEDGEMENT

We are grateful to Al Mok for pointing out a drawback in the original linear clustering algorithm. This research is partially supported by DARPA grant N00039-86-C-0167, and DOE grant DE-FG05-85ER-25010.

REFERENCES

[1] Banerjee, J., Kim, W., Kim, S. J., and Garza, J. F., "Clustering a DAG for CAD Databases," To appear in *IEEE-SE*.

[2] "Butterfly (TM) Parallel Processor Overview," Bolt Beranek and Newman Inc., Cambridge, MA, June 1985.

[3] Bokhari, S. H., "On the Mapping Problem," *IEEE-TC*, Vol. C-30, No. 3, Mar. 1981, pp. 207-214.

[4] Browne, J. C., "Formulation and Programming of Parallel Computations: A Unified Approach," *Proc. of Int'l Conf. on Parallel Processing*, Aug. 1985, pp. 624-631.

[5] Browne, J. C., "Framework for Formulation and Analysis of Parallel Computation Structures," *Parallel Computing* 3, 1986, pp. 1-9.

[6] Bryant, R. M., and Finkel, R. A., "A Stable Distributed Scheduling Algorithm," *2nd Int'l. Conf. on Distributed Computing Systems*, 1981, pp. 314-323.

[7] Chiang, W. P., "Optimal Graph Clustering Problems with Applications to Information System Design," Technical Report CRL-TR-30-84, The Univ. of Michigan, June 1984.

[8] Coffman, E. G., Jr., and Graham, R. L., "Optimal Scheduling for Two-Processor Systems," *Acta Informatica 1*, 1972, pp. 200-213.

[9] Coffman, E. G., Jr. (Ed.), *Computer and Job-Shop Scheduling Theory*, John Wiley and Son, N. Y., 1976.

[10] Eager, D. L., Lazowska, E. D., and Zahorjan, J., "Dynamic Load Sharing in Homogeneous Distributes Systems," *IEEE-SE*, Vol. SE-12, No. 5, May 1986, pp. 662-675.

[11] Forsdick, H., Schantz, R., and Thomas, R., "Operating Systems for Computer Networks," *IEEE Computer*, Vol. 11, Jan. 1978.

[12] Garey, M. R., and Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. M. Freeman and Company, New York, 1979.

[13] Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., and Snir, M., "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE-TC*, Vol. C-32, No. 2, Feb. 1983, pp. 175-189.

[14] Haessig, K., and Jenny, C. J., "Partitioning and Allocating Computational Objects in Distributed Computing Systems," *IFIP*, 1980, pp. 503-508.

[15] "iPSC User's Guide," Intel Corporation, Apr. 1987.

[16] Karp, R. M., and Miller, R. E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM J. Appl. Math.*, Vol. 14, No. 6, Nov. 1966, pp. 1390-1411.

[17] Kim, S. J., "A General Approach to Multiprocessor Scheduling," TR-88-4, The Univ. of Texas at Austin, Feb. 1988.

[18] Pathak, G. C., "Towards Automated Design of Multicomputer System for Real-time Applications," Ph. D. Thesis, North Carolina State Univ., 1984.

[19] Pfister, G. F., "The Architecture of the IBM Research Parallel Processor Prototype (RP3)," IBM Research Report RC 11210, June 1985.

[20] Shen, C.-C., and Tsai, W.-H., "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE-TC*, Vol. C-34, No. 3, Mar. 1985, pp. 197-203.

[21] Ullman, J. D., "NP-complete Scheduling Problem," *J. of Computer System Science*, Vol. 10, 1975, pp. 384-393.

[22] Wang, Y.-T., and Morris, R. J. T., "Load Sharing in Distributed Systems," *IEEE-TC*, Vol. C-34, No. 3, Mar. 1985, pp. 204-217.

[23] Wittie, L. D., and van Tilborg, A. M., "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer," *IEEE-TC*, Vol. C-29, No. 12, Dec. 1980, pp. 1133-1144.

Parallel Algorithms For River Routing[1]

(Extended Abstract)

Shing-Chong Chang
Department of Electrical Engineering
and
Systems Research Center
University of Maryland
College Park, MD 20742

Joseph JáJá
Department of Electrical Engineering
Institute For Advanced Computer Studies
and
Systems Research Center
University of Maryland
College Park, MD 20742

Abstract

We develop efficient parallel algorithms for several river routing problems. These algorithms can be implemented on the CREW-PRAM model in $O(\log n)$ or $O(\log^2 n)$ time with $O(n)$ processors, where n is the size of the input. Our algorithms have fast implementations on other parallel models such as the mesh or the hypercube.

1 Introduction

It is well-known that many of the optimization problems arising in VLSI routing are NP-complete (e.g. [KL],[L], [SB],[S]). One notable exception is the class of *river routing* problems associated with a hierechical layout strategy such as Bristle-Blocks([J]). See ([CS],[D et al],[LM],[LP], [M],[P],[SD],[T]) for more examples. In this paper, fast parallel algorithms for several river routing problems are presented. In particular, $O(\log n)$ or $O(\log^2 n)$ time algorithms with $O(n)$ processors are developed for the separation problem and for the routability problem around a rectilinear polygon ([P]).

The above problems are considered in the CREW-PRAM model, which is characterized by the presence of an unlimited number of processors which can access a shared memory unit. Concurrent read is allowed while concurrent write is not. We are aiming for efficient parallel algorithms that run in $O(\frac{T(n)}{p})$, where p is the number of processors and $T(n)$ is the running time of the best known sequential algorithm with input length n. In the rest of the paper, we assume that the reader is familiar with some of the basic parallel techniques such as path doubling, parallel prefix, and the Euler tour technique. Our algorithms can be mapped into fixed-interconnection parallel architectures such as the array architecture or the hypercube. For example, all the algorithms stated in this paper can be implemented on a $\sqrt{n} \times \sqrt{n}$ mesh in time $O(\sqrt{n})$, where n is the input length.

[1]Supported in part by NSA Contract No. MDA-904-85H-0015, NSF Grant No. DCR-86-00378 and by the Systems Research Center Contract No. OIR-85-00108.

The class of general river routing problems involves routing between ordered sequences of terminals such that the final layout is planar. One such problem is the wiring of two ordered sets of terminals $\{b_0, b_1, \ldots, b_{n-1}\}$ and $\{t_0, t_1, \ldots, t_{n-1}\}$ across a channel between the parallel boundaries of two rectangles. The width of the channel is the vertical distance between the two lines forming the channel. The *separation problem* is to find the minimum width of the channel necessary to wire all nets such that any two wires are separated by a unit distance. We will restrict ourselves to the case where the wires are rectilinear, i.e., there is a grid structure such that each wire consists of a set of grid line segments. Our methods generalize for all the other known variations ([SD],[T]).

A more general version of the river routing problem that is known to have an efficient serial algorithm is to perform planar routing where the ports lie on the boundary of a simple rectilinear polygon. In this case, we are interested in whether the routing is possible or not and, if it is possible, we have to provide the detailed routing. Several interesting subproblems such as finding the contour of the union of a set of rectilinear polygons or determining whether a set of nets can be wired within a set "passages" are also tackled.

2 The Separation Problem

Let $\{N_i =< b_i, t_i > \mid 1 \leq i \leq n\}$ be an instance of the channel separation problem. Notice that b_i and t_i will be also used to denote the horizontal coordinates of the terminals relative to an arbitrary origin. A net N_i is a *right* net if $b_i < t_i$. If $b_i > t_i$, then N_i is a *left* net. Otherwise, it is a *vertical* net. We can partition the nets into *right* blocks, *left* blocks and *vertical* blocks. A set of right nets $N_i, N_{i+1}, \ldots, N_p$ is a right block if it is a maximal block with the property $b_k < b_{k+1} \leq t_k$, for any $i \leq k < p$. We can similarly define left blocks and vertical blocks.

The wiring problem is reduced to wiring each block separately. We will concentrate on the wiring of right blocks. Obvious changes can be made to deduce the corresponding algorithm for left blocks.

The wiring of a net can be specified by the coordinates of its bend points. For example, net N_1 of Figure 1 has the bend points A_{11}, B_{11}. For each net N_i, we have $2k$ bend points, $A_{i1}, A_{i2}, \ldots, A_{ik}$ and $B_{i1}, B_{i2}, \ldots, B_{ik}$, for some k. Not all of these bend points are needed to determine the overall wiring. Let's call A_{i1} and B_{i1} (bend points closest to the bottom row) the *characteristic* bend points and all the others *ordinary* bend points. Notice that the characteristic bend points uniquely define the overall wiring since once we have the wiring of N_{i-1} and the characteristic bend points A_{i1} and B_{i1}, we can determine all the ordinary bend points of N_i very easily. Figure 1 shows an example of a river routing problem and a wiring achieving the minimum separation.

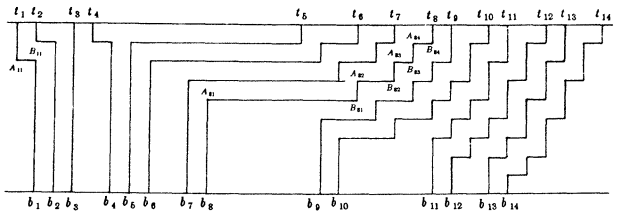


Figure 1: Basic river routing problem

The algorithm to find the minimum separation is based on the following lemma.

Lemma 1 *Let N_i be a net in a right block and let $\hat{j}$ be the minimum $j \leq i$ such that $t_j + (i - j - 1) \geq b_i$. Then the coordinates of the characteristic bend points of N_i are $A_{i1} = (b_i, i - \hat{j} + 1)$ and $B_{i1} = (t_{\hat{j}} + i - \hat{j}, i - \hat{j} + 1)$.*

We now show how to compute in parallel the index $\hat{j}(i)$ for each i.

Algorithm Index

input: A set of nets $< b_i, t_i >$, $1 \leq i \leq n$, forming a right block.

output: $\hat{j}(i)$ such that $\hat{j}(i)$ is the minimum j such that $b_i - t_j \leq i - j - 1$, for each $1 \leq i \leq n$.

1. Compute $b'_i = b_i - i$ and $t'_j = t_j - j - 1$ for each i and j.
2. Sort the t'_js, say $t_{p_1} \leq t_{p_2} \leq \ldots \leq t_{p_n}$.
3. For each p_i, determine $f(p_i) = min\{p_k | i \leq k \leq n\}$.
4. Sort the b'_is and the t'_js such that if a $b'_i = t'_j$, the b'_i is pushed to the lower rank.
5. For each b'_i, let t'_{p_j} be the closest $t'_{p_k} \geq b'_i$. Then $f(p_j) = \hat{j}(i)$.

Now we can find the minimum separation as well as the charactersitic bend points of all the nets by partitioning the nets into blocks and by using algorithm Index and Lemma 1.

Theorem 1 *The minimum separation and the characteristic bend points of n input nets can be found in $O(\log n)$ time with $O(n)$ processor on a CREW-PRAM. If all terminals lie in the range $[1, N]$, where $N = O(n)$, then the running time is $O(\frac{n}{p} + \log n)$ with p processors, for all $1 \leq p \leq n^{1-\epsilon}$ (any $\epsilon > 0$).*

3 Routing In a Simple Polygon

The routing problem of nets within a simple rectilinear polygon introduced in([P]) is a generalization of the standard river routing problem. In this case we are supposed to connect a set of terminals $a_1, a_2, \ldots, a_n$ on the boundary of a simple rectilinear polygon to another set of terminals $b_1, b_2, \ldots, b_n$ on the boundary of the same polygon such that all the wires lie within the polygon and no two wires intersect. *Routability testing* is to determine whether or not a one layer routing is possible and *detailed routing* is to specify the actual wiring of the n nets, if they are routable. We will restrict ourselves to the rectangle case. However all the algorithms can be generalized to any rectilinear polygon.

3.1 Detailed Routing

Let $N_i =< a_i, b_i >$ be an arbitrary net. The terminals a_i and b_i divide the boundary of a rectangle R into two parts. The part of smaller length will be called the *internal boundary* of N_i. The other part will be called the *external boundary*. A net N_i is *covered* by another net N_j if the terminals of N_j are in the external boundary of N_i and the terminals of N_i are in the internal boundary of N_j. A *representative net* is a net that is not covered by any other net. Figure 2 shows an example of a detailed routing problem such that N_1, N_6 and N_{14} are the representative nets. We can partition the nets into *groups* such that each group consists of a representative net and all the nets covered by it. The groups in Figure 2 are $\{N_1, N_2, N_3, N_4, N_5\}$,$\{N_6, N_7, N_8, N_9, N_{10}, N_{11}, N_{12}, N_{13}\}$, and $\{N_{14}, N_{15}\}$. One can show the following.

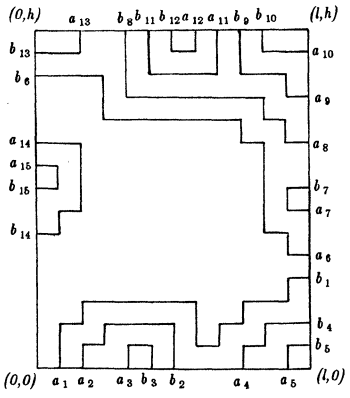


Figure 2: Basic river routing around a rectangle boundary

Lemma 2 *Suppose a given instance of the above problem is routable. Then the routing can be performed by routing each group of nets separately.*

The general strategy for specifying the routing will be the following: (i) identify the proper groups , (ii) find the representative nets, and (iii) specify the routing of each group. By the parallel techniques of sorting, path doubling and prefix computation, we can create a chain of the nets involved in each group such that a representative net is a sink and the chains have following properties:

Lemma 3 *Let $N_{r1}, N_{r2}, \ldots, N_{r_k}$ be all the representative nets and let $R(N_{r_i})$ be the number of nets in the internal boundary of N_{r_i}. Then $\sum_{i=1}^{k}(R(N_{r_i})+1) = n$. Moreover, there exists a wiring strategy such that N_{r_i} has at most $2(R(N_{r_i})+1)$ bend points.*

Corollary: The total number of bend points of all the representative nets is $O(n)$, where n is the number of nets.

Lemma 4 *Let n be the number of nets. Then all the groups and representative nets can be identified in time $O(\log\ n)$ with $O(n)$ processors on the PRAM. With p processors, we obtain $O(\frac{n}{p} + \log n)$, $1 \leq p \leq n^{1-\epsilon}$ and $\epsilon > 0$.*

We now turn to the problem of routing each group separately. Our goal here is to identify the bend points of each representative net. Let $N =< x, y >$ be a net in a group whose representative is N_r. Let k be the number of nets between N and N_r, including both N and N_r. The *bounding perimeter of rank k* is the rectilinear boundary of the region determined by N such that the wiring of N_r cannot lie inside it, i.e., this is the boundary of the region within the rectangle of all the points of distance $\leq k$ of the rectangle boundary determined by N. Consider again the case of Figure 2. Let $B_{k,i}$ be the bounding perimeter of rank k induced by net N_i. Figure 3 shows the contours $B_{3,3}$, $B_{3,5}$, $B_{2,2}$, $B_{2,4}$ and $B_{1,1}$. We claim that the following lemma is true.

Lemma 5 *The union of all the bounding perimeters of all the nets within a group determines the contour of the group and hence determines the wiring of the representative net.*

To determine the union, flatten the rectangle into a line. Suppose a terminal p gets mapped into $\bar{p}$. A bounding perimeter connecting p and q of rank k will get mapped into a simple rectangle with endpoints $\bar{p}$ and $\bar{q}$ and height k. Denote the mapped bounding perimeters by $\bar{R}_1, \bar{R}_2, \ldots, \bar{R}_l$. These rectangles determine a (union) contour R given by its extreme points. Then map these extreme points back to the original rectangle to get the wiring of the representative net. Few of these points

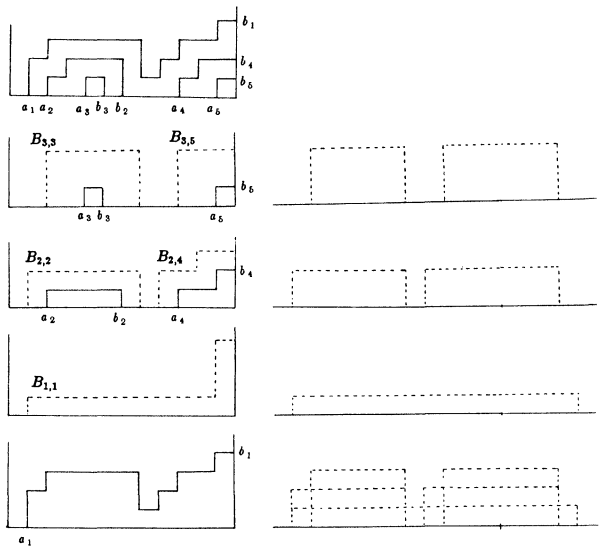


Figure 3: The union of all bounding perimeters

around the corners may not be mapped into extreme points of the contour within the rectangle, but rather onto the boundary. These can be determined quickly and then eliminated. We are now ready to state the algorithm.

Algorithm Contour

Input: A group of nets with their representative.
Output: The bendpoints of the corresponding contour.

1. Determine the rank of each net, i.e. the number of nets between itself and the representative net.
2. Determine all the bounding perimeters.
3. Flatten the rectangle boundary into a line. Map the bounding perimeters into this line . Each corresponding perimeter can be identified by $(\bar{p}, \bar{q}, k)$.
4. Sort the triplets $(\bar{p}, \bar{q}, k)$ according to k. For each k, determine the union of line segments at distance k.
5. From each line segment generated at step 4, determine the corresponding bendpoints. The overall contour can be specified by the bend points.
6. Map the bend points of the contour on the line back into the rectangle. Eliminate those points within the rectangle which are not bend points.

Lemma 6 *If the number of nets in the group is n, then algorithm contour can be implemented in time $O(\log\ n)$ with $O(n)$ processors.*

Theorem 2 *Detailed routing of the representative nets of n nets within a simple rectilinear polygon can be done in time $O(\log n)$ with $O(n)$ processors. With p processors, we have $O(\frac{n}{p} + \log n)$, $1 \leq p \leq n^{1-\epsilon}$ and $\epsilon > 0$.*

3.2 Routability Testing

The problem may be unroutable for one of the following reasons: (1) The graph determined by the nets when

restricted to lie within the rectangle is *nonplanar*. (2) The wiring of all the nets requires more area.

Lemma 7 *Whether the interconnection pattern of the given nets is planar can be determined in time $O(\log\ n)$ time with $O(n)$ processors on the PRAM model.*

A *single side* net is a net whose terminals lie on the same side of the rectangle. If the terminals lie on adjacent sides then the net is called *corner* net. It is a *cross* net if the terminals lie on opposite sides. Partition the single side nets corresponding to each specific side into *single side blocks* such that each net except one (*cover net*) is covered by one or more nets in the block. Moreover each such block is maximal. A *corner block* is a maximal set of corner nets corresponding to the same corner such that each net except one (*cover net*) is covered by one or more nets within the block. Moreover no other net outside a block is covered by the cover net. For example, in Figure 2, N_2 is a single side net, N_1 is a corner net and N_6 is a cross net. The single side blocks are $\{N_2, N_3\}$, $\{N_7\}$, $\{N_{11}, N_{12}\}$ and $\{N_{14}, N_{15}\}$, whose corresponding cover nets are N_2,N_7,N_{11} and N_{14}. $\{N_4, N_5\}$, $\{N_9, N_{10}\}$ and $\{N_{13}\}$ are the corner blocks with N_4,N_9, N_{13} as the corresponding cover nets.

To decide whether the above blocks are routable, first determine the wiring of all the cover nets by algorithm Contour then check whether there is any intersection between the wires of the cover nets.

Lemma 8 *Whether or not the single side blocks and the corner blocks can be wired within the rectangle can be determined in $O(\log n)$ time with $O(n)$ processors.*

Once the block cover nets are wired, it should checked if there is enough space to route the remaining nets. Our approach consists of determining *the wiring capacity* and the *wiring density* between blocks. The wiring capacity between two blocks is the number of nets that can be wired between these two blocks, while the wiring density is the number of wires that have to be wired between these two blocks. The capacity between blocks on two orthogonal sides of the rectangle boundary is computed as follows. Given a block B consider all the convex corners of B. Generate 45 degree "rays" from each such corner and determine the line segment where it intersects another block contour or the original rectangle boundary. Based on this information, one can determine the width of the narrowest passage between B and any other block. The details are given in the full paper.

Algorithm Intersection

Input: Contours of single side and corner blocks on two orthogonal sides of rectangle boundary.
Output: Intersection points of rays emanating from convex corners.

1. Consider the case of the the lower right corner. The other cases can be dealt with in a similar fashion. Sort all the line segments determined by the block contours and the right side of the rectangle R. Determine the projection of each line segment on the diagonal, say line segment i is projected into line segment $p(i)$ on the diagonal.
2. Sort the projections according to their order on the diagonal and compute $p'(i) = p(i) - \bigcup_{j=1}^{i-1} p(i)$.
3. For each ray y coming out of a corner of contour on the horizontal side of the original rectangle, find its intersection with the diagonal. If the intersection point lies in $p'(j)$, then ray y intersects segment j. Determine the intersection point of ray y and line segment j.
4. If a ray y intersects the original rectangle boundary, then rotate to find the intersection with the next line segment belonging to some block contour (see Figure 4 and ray y_B). Now determine the point of intersection.

For example, one can check that in Figure 4 $p'(CD) = C'D'$ and $p'(EF) = D'F'$. Hence rays y_A and y_B intersect CD and EF respectively. If we rotate y_B, we can find the intersection with the next line segment GF.

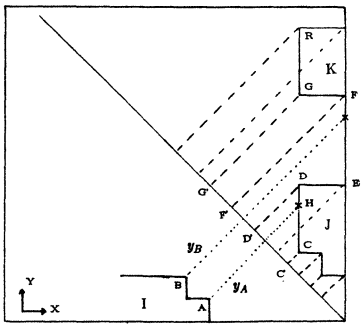


Figure 4: Intersection between rays and block contours

Lemma 9 *Algorithm Intersection finds the intersection points of rays emanating from convex corners with the line segments of contours on two orthogonal sides of the bounding rectangle in time $O(\log n)$ time with $O(n)$ processors.*

Use algorithm Intersection to compute the intersection point of each ray with a single side contour, corner block contour or the original boundary of the rectangle. The capacity between blocks can then be calculated easily. Then compare with the density between blocks to determine the routability between blocks.

Lemma 10 *Testing the routability of n nets between two orthogonal sides of a rectangle can be done in $O(\log\ n)$ time with $O(n)$ processors on the CREW PRAM model.*

We now address the routability problem between two opposite sides of the bounding rectangle. The genera-

tion of horizontal, vertical and 45 degree rays from each convex corner is not enough to determine the routability between two opposite sides. We will use a divide-and-conquer strategy to handle this case.

Assume without loss of generality that all cross nets are between the top and the bottom sides. Select two adjacent cross nets N_i and N_j that split the nets almost evenly. Let N_i be to the left of N_j. (Figure 5) Find the temporary wiring of N_i as close to the left as possible and the temporary wiring of N_j as close to the right as possible. Check whether any intersection will result. Repeat above procedure recursively for the cross nets to the left of N_i and for the cross nets to the right of N_j separately.

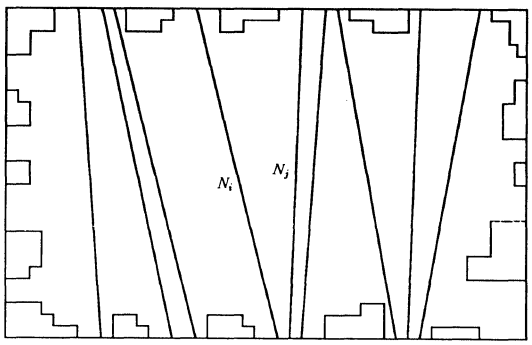


Figure 5: Routability between two blocks in opposite sides

Theorem 3 *Testing the routability of n nets within a simple rectilinear polygon could be done in $O(\log^2 n)$ time with $O(n)$ processors on the CREW PRAM model.*

4 References

[AH] M. Atallah and S. Hambrusch, " Solving tree problems on a mesh-connected processor array," Proceedings of the 26th Symp. FOCS, 1985, pp. 222-231.

[CS] R. Cole and A. Siegel, " River routing every which way, but loose," 25th FOCS, October 1984, pp. 65-73.

[D et al] D. Dolev, K. Karplus, A. Seigel, A. Strong and J. Ullman, "Optimal wiring between rectangles," Proc. 13th Annual ACM Symposuim STOC, May 1981, pp. 312-317.

[J] D. Johannsen, "Bristle blocks: a silicon compiler," Proc. 16th Design Automation Conference, June 1979, pp. 310-313.

[KL] M. Kramer and J. van Leeuwen, "Wire routing is NP-complete," technical report, University of Utrecht, the Netherlands, February 1982.

[L] A. LaPaugh, "Algorithms for integrated circuit layout: an analytic approach," Ph.D. dissertation, MIT, Cambridge, MA, November 1980.

[LM] C.E. Leiserson and F.M. Maley, "Algorithms for routing and testing routing of planar VLSI layout," 17th ACM STOC, May 1985, pp. 69-78.

[LP] C. Leieserson and R. Pinter, "Optimal placement for river routing," SICOMP 12(3), August 1983, pp. 447-462.

[M] A. Mirzaian, " Channel routing in VLSI," 16th ACM STOC, May 1984, pp. 101-107.

[P] R. Pinter, "River routing: methodology and analysis," Proceedings of the third CALTECH Conference on Very Large Scale Integration, March 1983, pp. 141-163.

[SB] S. Sahni and A. Bhatt, "Complexity of the Design Automation Problem," Proceedings of the 17th Design Automation Conference, June 1980, pp. 402-411.

[SD] A. Seigel and D. Dolev, "The separation for general single layer wiring barriers," Proceedings of the CMU Conference on VLSI Systems and Computations, October 1981, pp. 143-152.

[S] T. Szymanski, "Dogleg Channel routing is NP-complete," manuscript, Bell Laboratories, Murray Hill, NJ, September 1981.

[T] M. Tompa, "An optimal solution to a wire routing problem," Proceedings of the 12th Annual Symposuim on Theory of Computing, April 1980, pp. 161-176.

NEARLY OPTIMAL CLOCK SYNCHRONIZATION UNDER UNBOUNDED MESSAGE TRANSMISSION TIME

(Extended Abstract)

Kazuo Sugihara[1] *and Ichiro Suzuki* [2]

[1] Department of Information and Computer Sciences
University of Hawaii at Manoa
Honolulu, HI 96822

[2] Department of Electrical Engineering and Computer Science
University of Wisconsin–Milwaukee
Milwaukee, WI 53201

Abstract Consider a fully connected network of $n \geq 3$ processes in which a process can send messages to a set of other processes simultaneously. Messages sent from a process to other processes simultaneously at time t are guaranteed to be delivered in the time interval $[t+\delta, t+\delta+\epsilon]$ for some δ and ϵ, where ϵ is a constant but δ can vary and no upper bound on δ is known. We show that, under this assumption, the clocks of the n processes cannot be synchronized any more closely than $(1+\frac{1}{n(n-2)})\epsilon$, even if the clocks run at the rate of real time. A simple algorithm that synchronizes the clocks to within $(1+\frac{1}{n})\epsilon$ is presented. The $(1+\frac{1}{n})\epsilon$ upper bound on the imprecision of clock synchronization, together with the $(1-\frac{1}{n})\epsilon$ lower bound found in the literature for the case in which both δ and ϵ are known constants, implies that whether or not there exists a given upper bound on the message transmission time becomes less and less significant when the number of processes increases. This is the first known solution for clock synchronization under unbounded message transmission time.

1 Introduction

The problem of synchronizing clocks in a distributed system has been investigated under various assumptions. For example, in [7], Lundelius and Lynch considered the problem in an error-free system of n processes in which there is an uncertainty of ϵ in the message delivery time. That is, for some *known* constants δ and ϵ, a message sent by a process at time t is guaranteed to be delivered at the destination within the time interval $[t+\delta, t+\delta+\epsilon]$. They show that, under this assumption, it is impossible to synchronize the clocks of n processes any more closely than $(1-\frac{1}{n})\epsilon$, even if all clocks run at the rate of real time. They also present an algorithm that achieves this bound. Clock synchronization when processes and communication links can fail has been studied extensively in [1] [2] [3] [5] [6] [8].

All clock synchronization algorithms reported in the literature [1]–[8] have been obtained under the assumption that an upper bound on the message transmission time is given. Although this may be a reasonable assumption in many practical situations, achieving clock synchronization when no upper bound on the transmission time is available is interesting, not only from the theoretical point of view.

In this paper we consider the problem of clock synchronization in a fully connected, error-free network of $n \geq 3$ processes in which a process can send messages to any set of processes simultaneously. We assume that if a process P sends messages to a set S of processes simultaneously at time t, then

1. the messages addressed to the processes $P' \in S$ such that $P' \neq P$ are received within the time interval $[t+\delta, t+\delta+\epsilon]$ for some finite $\delta > 0$ and $\epsilon \geq 0$, where ϵ is a constant but δ can vary and no upper bound on δ is known, and
2. if $P \in S$, then the transmission time of the message from P to P itself may not be related to those of the messages addressed to the processes $P' \neq P$.

That is, messages sent by a process to *other* processes simultaneously are delivered within a time interval of size ϵ, but the message transmission times can be unbounded.

If messages sent by a process P simultaneously to a set S of processes such that $P \in S$ are *all* delivered within a time interval of size ϵ, then clock synchronization becomes a trivial problem. It is conceivable, however, that in certain systems messages sent by a process to itself are processed locally and delivered immediately, whereas messages sent to other processes are delivered

more or less simultaneously when the communication channel becomes available after an unpredictable delay.[1] The model we consider can be a close approximation of such a system.

It should be easy to see, at least intuitively, that synchronizing clocks without using an upper bound on the message transmission time is more involved compared to the case in which an upper bound is known. For example, in the algorithm of [7], a process which receives a message assumes that the transmission time of the message was exactly $\delta + \epsilon/2$, the average of the lower and upper bounds. If no upper bound is given, such a simple approximation is not possible.

We show that, under the assumption described above, the clocks of n processes cannot be synchronized any more closely than $(1 + \frac{1}{n(n-2)})\epsilon$ for any $n \geq 3$, even if the clocks run at the rate of real time. The proof is by the "many scenarios" techniques [1] [7] used commonly for this purpose. Next, we present a simple algorithm that synchronizes the clocks of n processes to within $(1 + \frac{1}{n})\epsilon$ for any $n \geq 3$. The algorithm achieves optimal clock synchronization for $n = 3$ and is nearly optimal for $n \geq 4$.

An interesting observation is in order. The $(1 - \frac{1}{n})\epsilon$ lower bound on the imprecision of clock synchronization proved in [7]–for the case in which the message transmission time is in the range $[\delta, \delta + \epsilon]$ for known constants δ and ϵ–increases and approaches ϵ when n becomes larger. In contrast, the $(1 + \frac{1}{n})\epsilon$ upper bound obtained under the assumption of this paper decreases and approaches ϵ when n becomes larger. This implies that whether or not there exists a given upper bound on the message transmission time becomes less and less significant when the number of processes increases.

2 The Model

Let $P_1, P_2, \ldots, P_n$ be $n \geq 3$ processes. We assume that messages sent by a process to other processes simultaneously at real time t are received in the time interval $[t + \delta, t + \delta + \epsilon]$ for some finite $\delta > 0$ and $\epsilon \geq 0$, where ϵ is known but δ can vary and no upper bound on δ is known. Other than this, the model we use is essentially that of [2] [7].

Process P_i has a *physical clock* C_i which is a real-valued function of real time. We assume that the physical clocks run at the rate of real time and they cannot be reset by the processes; that is, $C_i(t) = C_i(0) + t$ at every real time $t \geq 0$. The processes have no access to the real time.

[1] For example, in a local area network consisting of sites running UNIX™ connected by Ethernet, messages sent from a process to itself are routed through a local "loopback" interface whose delay is independent of the load of the Ethernet.

Destination	Transmission Time
P_1	$d + \epsilon$
P_2	$d + (1 - \frac{1}{n-2})\epsilon$
	$\ldots$
P_{i-1}	$d + (1 - \frac{i-2}{n-2})\epsilon$
P_{i+1}	$d + (1 - \frac{i-1}{n-2})\epsilon$
P_{i+2}	$d + (1 - \frac{i}{n-2})\epsilon$
	$\ldots$
P_{n-1}	$d + (1 - \frac{n-3}{n-2})\epsilon$
P_n	d

Table 1: Message transmission times from P_i to other processes in e_1.

Process P_i has a local variable A_i (for *adjustment*) which provides the difference between the logical and physical clock times of P_i. That is, the *logical time* $L_i(t)$ of process P_i at real time t is given by $L_i(t) = C_i(t) + A_i(t)$, where $A_i(t)$ is the value of A_i at t.

Following [2], we assume that a clock synchronization algorithm is a deterministic algorithm in which the state transition and the action of sending messages of process P_i at real time t is determined only by the value of $C_i(t)$ and the message history of P_i at t. Here, the *message history* of P_i at real time t is the sequence consisting of tuples of the form $< P_j, m, T, y >$ for every message P_i has sent or received before t, where $< P_j, m, T, y >$ represents that message m was either sent ($y =$ **sent**) or received ($y =$ **received**) to or from P_j when the value of C_i was T. An algorithm is said to *synchronize the logical times to within* γ if the algorithm eventually terminates, and when it terminates at real time t, $|L_i(t) - L_j(t)| \leq \gamma$ holds for any $i \neq j$.

3 Lower Bound

In this section, we show that no algorithm can synchronize the logical times of n processes any more closely than $(1 + \frac{1}{n(n-2)})\epsilon$ in our model. The proof is by the standard "many scenarios" techniques [1] [7].

Theorem 1 *No algorithm can synchronize the logical times of n processes to within γ, for any $\gamma < (1 + \frac{1}{n(n-2)})\epsilon$.*

Proof (Sketch) Fix an algorithm that synchronizes the logical times to within γ. Let e_1 be an execution of the algorithm in which the transmission times of messages from P_i to other processes are as given in Table 1, where $d > (1 + \frac{1}{n-2})\epsilon$ is a constant. The transmission time of a message from P_i to P_i itself is an arbitrary constant. Since in e_1 messages sent by a process to other processes at real time t are received within the time interval $[t + d, t + d + \epsilon]$ of size ϵ, e_1 is a valid execution.

Consider another execution e_2 which is obtained from e_1 by "shifting" [7] P_1 by $(1+\frac{1}{n-2})\epsilon$. That is, e_2 is identical to e_1 except that

1. at any given real time, the physical clock reading of P_1 in e_2 is larger than that in e_1 by $(1+\frac{1}{n-2})\epsilon$,
2. the transmission time of a message from P_1 to P_j $(j \neq 1)$ is increased by $(1+\frac{1}{n-2})\epsilon$,
3. the transmission time of a message from P_j $(j \neq 1)$ to P_1 is decreased by $(1+\frac{1}{n-2})\epsilon$, and
4. all state transitions and actions of sending messages of P_1 take place earlier in e_2 than in e_1 by $(1+\frac{1}{n-2})\epsilon$ in real time.

The execution e_2 is valid, since all messages sent by a process to other processes are received within a time interval of size ϵ. Similarly, for $2 < i \leq n$, we can obtain a valid execution e_i from e_{i-1} by shifting P_{i-1} by $(1+\frac{1}{n-2})\epsilon$.

Now assume that in e_1, the logical times of $P_1, P_2, \ldots, P_n$ are $T_1, T_2, \ldots, T_n$, respectively, at real time t_f when the algorithm has terminated at every process. By assumption we have

$$T_n \leq T_1 + \gamma.$$

Since during the execution of the algorithm each process has the same message history in e_1 and e_2 when its physical clock has the same value, the values of A_i computed by the algorithm are the same in e_1 and e_2. Thus the logical times of P_1 and P_2 at t_f in e_2 are $T_1 + (1+\frac{1}{n-2})\epsilon$ and T_2, respectively. Then by assumption we have

$$T_1 + (1+\frac{1}{n-2})\epsilon \leq T_2 + \gamma.$$

Similarly, for $2 < i \leq n$, the logical times of P_{i-1} and P_i at t_f in e_i are $T_{i-1} + (1+\frac{1}{n-2})\epsilon$ and T_i, respectively, and thus by assumption we have

$$T_{i-1} + (1+\frac{1}{n-2})\epsilon \leq T_i + \gamma.$$

By adding the n inequalities we obtain

$$\gamma \geq (1+\frac{1}{n(n-2)})\epsilon.$$

□

4 A Simple Algorithm

There exists a simple algorithm which synchronizes the logical times of n processes to within $(1+\frac{1}{n})\epsilon$ for any $n \geq 3$.

The concept of "view" introduced below is essential in describing the algorithm. Suppose that P_i sends the message SNAPSHOT$_i$ to $P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_n$ simultaneously, and let v_j $(j \neq i)$ be the value of the physical clock C_j at the moment SNAPSHOT$_i$ is received by P_j. Then the n-tuple

$$V = (v_1, v_2, \ldots, v_{i-1}, -, v_{i+1}, \ldots, v_n)$$

is called a *view* of P_i, where '$-$' represents "don't care." Since messages sent simultaneously by a process to other processes are received within a time interval of size ϵ, the following lemma is immediate.

Lemma 1 *Let $V = (v_1, v_2, \ldots, v_{i-1}, -, v_{i+1}, \ldots, v_n)$ be a view of P_i. There exist some real time t and $\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_n$ such that for each $j \neq i$, $0 \leq \alpha_j \leq \epsilon$ and $v_j = C_j(t) + \alpha_j$.*

The algorithm can be divided into the following two phases. It is a straightforward exercise to represent the algorithm in any given language in such a way that its execution will eventually terminate at every process.

Phase 1 Obtain a view

$$V_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,i-1}, -, v_{i,i+1}, \ldots, v_{i,n})$$

of P_i for each $1 \leq i \leq n$.

Phase 2 Compute $A_1, A_2, \ldots, A_n$ from $V_1, V_2, \ldots, V_n$ as follows. For $1 \leq i \leq n$,

$$A_i = \frac{1}{n} \sum_{1 \leq k \leq n} D_{k,i}$$

where for $1 \leq k, i \leq n$,

$$D_{k,i} = \begin{cases} \frac{1}{n-2} \sum_{1 \leq l \leq n, l \neq k,i} (v_{l,k} - v_{l,i}) & \text{if } k \neq i \\ 0 & \text{if } k = i. \end{cases}$$

For $k \neq i$, $D_{k,i}$ is the average of the differences between the physical clock readings of P_k and P_i observed in views V_l such that $l \neq k, i$. A_i is the average of $D_{k,i}$ over all k, including $D_{i,i} = 0$.

By Lemma 1, for each view

$$V_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,i-1}, -, v_{i,i+1}, \ldots, v_{i,n}),$$

there exist t_i and $\alpha_{i,1}, \alpha_{i,2}, \ldots, \alpha_{i,i-1}, \alpha_{i,i+1}, \ldots, \alpha_{i,n}$ such that $0 \leq \alpha_{i,j} \leq \epsilon$ and $v_{i,j} = C_j(t_i) + \alpha_{i,j}$ for $j \neq i$. In the following let t be any real time when the execution of the algorithm has terminated at every process.

Lemma 2 *For $1 \leq i \leq n$,*

$$L_i(t) = \frac{1}{n} \sum_{1 \leq k \leq n} C_k(t) + \frac{1}{n(n-2)} \sum_{1 \leq k \leq n, k \neq i} \sum_{1 \leq l \leq n, l \neq k,i} (\alpha_{l,k} - \alpha_{l,i}).$$

Proof Since $C_k(t_l) - C_i(t_l) = C_k(t) - C_i(t)$, for $k \neq i$ we have

$$\begin{aligned} D_{k,i} &= \frac{1}{n-2} \sum_{1 \leq l \leq n, l \neq k,i} (v_{l,k} - v_{l,i}) \\ &= \frac{1}{n-2} \sum_{1 \leq l \leq n, l \neq k,i} ((C_k(t_l) + \alpha_{l,k}) \\ &\qquad -(C_i(t_l) + \alpha_{l,i})) \\ &= C_k(t) - C_i(t) + \frac{1}{n-2} \sum_{1 \leq l \leq n, l \neq k,i} (\alpha_{l,k} - \alpha_{l,i}). \end{aligned}$$

Thus

$$\begin{aligned} L_i(t) &= C_i(t) + A_i(t) \\ &= C_i(t) + \frac{1}{n} \sum_{1 \leq k \leq n} D_{k,i} \\ &= C_i(t) + \frac{1}{n} \sum_{1 \leq k \leq n, k \neq i} (C_k(t) - C_i(t)) \\ &\quad + \frac{1}{n(n-2)} \sum_{1 \leq k \leq n, k \neq i} \sum_{1 \leq l \leq n, l \neq k,i} (\alpha_{l,k} - \alpha_{l,i}) \\ &= \frac{1}{n} \sum_{1 \leq k \leq n} C_k(t) \\ &\quad + \frac{1}{n(n-2)} \sum_{1 \leq k \leq n, k \neq i} \sum_{1 \leq l \leq n, l \neq k,i} (\alpha_{l,k} - \alpha_{l,i}). \end{aligned}$$

□

Theorem 2 *The algorithm synchronizes the logical times of the n processes to within* $(1 + \frac{1}{n})\epsilon$. *That is,* $|L_i(t) - L_j(t)| \leq (1 + \frac{1}{n})\epsilon$ *for any* $i \neq j$.

Proof (Sketch) By Lemma 2,

$$\begin{aligned} & L_i(t) - L_j(t) \\ &= \frac{1}{n(n-2)} \{ \sum_{1 \leq k \leq n, k \neq i} \sum_{1 \leq l \leq n, l \neq k,i} (\alpha_{l,k} - \alpha_{l,i}) \\ &\qquad - \sum_{1 \leq k \leq n, k \neq j} \sum_{1 \leq l \leq n, l \neq k,j} (\alpha_{l,k} - \alpha_{l,j}) \} \\ &= \frac{1}{n(n-2)} (X - Y), \end{aligned}$$

where

$$X = \sum_{1 \leq k \leq n, k \neq i,j} \alpha_{j,k} + (n-1) \sum_{1 \leq l \leq n, l \neq i,j} \alpha_{l,j} + (n-2)\alpha_{i,j}$$

and

$$Y = \sum_{1 \leq k \leq n, k \neq i,j} \alpha_{i,k} + (n-1) \sum_{1 \leq l \leq n, l \neq i,j} \alpha_{l,i} + (n-2)\alpha_{j,i}.$$

Since $0 \leq \alpha_{l,k} \leq \epsilon$ for $l \neq k$, we have

$$\begin{aligned} 0 \leq X, Y &\leq \{(n-2) + (n-1)(n-2) + (n-2)\}\epsilon \\ &= (n+1)(n-2)\epsilon. \end{aligned}$$

Thus

$$\begin{aligned} |L_i(t) - L_j(t)| &\leq \frac{(n+1)(n-2)}{n(n-2)}\epsilon \\ &= (1 + \frac{1}{n})\epsilon. \end{aligned}$$

□

5 Remarks

Since the $(1 + \frac{1}{n(n-2)})\epsilon$ lower bound and the $(1 + \frac{1}{n})\epsilon$ upper bound proved in this paper coincide with each other if $n = 3$, the algorithm achieves optimal clock synchronization for $n = 3$. Closing the small gap of $\frac{n-3}{n(n-2)}\epsilon$ between the two bounds for $n \geq 4$ remains as an open problem.

References

[1] D. Dolev, J. Halpern and R. Strong, "On the possibility and impossibility of achieving clock synchronization," *J. Computer and System Sciences* 32, 1986, pp. 230– 250.

[2] J. Halpern, N. Megiddo and A. Munshi, "Optimal precision in the presence of uncertainty," *Journal of Complexity* 1, 1985, pp. 170–196.

[3] J. Halpern, B. Simons, R. Strong and D. Dolev, "Fault-tolerant clock synchronization," *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, 1984, pp. 89–92.

[4] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Communications of the ACM* 21, No. 7, 1978, pp. 558–565.

[5] L. Lamport and P. M. Melliar-Smith, "Byzantine clock synchronization," *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, 1984, pp. 68–74.

[6] J. Lundelius and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, 1984, pp. 75–88.

[7] J. Lundelius and N. Lynch, "An upper and lower bound for clock synchronization," *Information and Control* 62, 1984, pp. 190–204.

[8] T. K. Srikanth and S. Toueg, "Optimal clock synchronization," *Proc. 4th Annual ACM Symposium on Principles of Distributed Computing*, Ontario, Canada, 1985, pp. 71–86.

Parallel Algorithms For Channel Routing in the Knock-Knee Model[1]

Shing-Chong Chang
Department of Electrical Engineering
and
Systems Research Center
University of Maryland
College Park, MD 20742

Joseph JáJá
Department of Electrical Engineering
Institute For Advanced Computer Studies
and
Systems Research Center
University of Maryland
College Park, MD 20742

Abstract

We consider the channel routing problem of a set of two-terminal nets in the knock-knee model. The known strategy to handle this problem seems to be inherently sequential. We develop a new approach to route all the nets within d tracks, where d is the density, such that the corresponding layout can be realized with three layers. Both the routing and the layer assignment algorithms have linear time sequential implementations. In addition, they both can be implemented on the CREW-PRAM model in $O(\log n)$ time with $O(n)$ processors, where n is the number of nets. With $1 \leq p \leq n^{1-\epsilon}$ processors, ϵ any positive constant, the running time of the algorithms is $O(\frac{n}{p} + \log n)$.

1 Introduction

The recent advances in the VLSI technology allow the fabrication of highly complex systems on single chips. Sophisticated software tools are needed to successfully design such systems. In particular, the routing phase is a critical and time-consuming part of the overall design process. Unfortunately, it turns out that most routing problems are NP-complete and hence no efficient solutions seem to be likely. There are few exceptions, however. For example, various river routing (one-layer) problems, the two-layer channel routing with no constraints, and few routing problems in the knock-knee model are known to have efficient solutions ([D et al],[MP],[O],[P], [PL]). Our goal is to develop a good set of techniques to obtain fast and efficient parallel routing algorithms.

In this paper, we consider the channel routing problem of two-terminal nets in the knock-knee model. A routing algorithm that uses d tracks, where d is the density, is presented in ([PL]) such that the routing can be realized with three layers. This algorithm can viewed as a nontrivial extension of the left edge algorithm ([O]) in which the routing is done row by row, left to right according to a greedy strategy. However, this method seems to be *inherently sequential* even for the case when each column has at most one terminal. We develop a novel strategy to obtain the optimal routing (which is in general different from the one obtained by the [PL] method) such that both the routing and the layer assignment algorithms have linear time sequential implementations. Moreove, they are both fully parallelilzable in the sense that they can be implemented on the CREW-PRAM model in $O(\log n)$ time with $O(n)$ processors, where n is the number of nets. If all the terminals lie in the range $[1, N]$, where $N = O(n)$, then these algorithms will run in time $O(\frac{n}{p} + \log n)$ time with $p \leq n^{1-\epsilon}$ processors, where ϵ is any positive constant.

The rest of the paper is organized as follows. The basic definitions needed for the rest of the paper are introduced in the next section, while in section 3 we develop a novel routing strategy and establish its correctness. The layer assignment algorithm is presented in the last section.

2 Definitions

We assume that the reader is familiar with the basic definitions related to channel routing (See for example [O],[PL]). In this paper, we restrict ourselves to two-terminal nets $N = < t, b >$, where t is the *top* terminal (on the top row) and b is the *bottom* terminal. t and b will also represent the integer displacements of these terminals relative to a fixed origin. N is a *left* (*right*) net if $t < b$ ($t > b$). Otherwise it is a *vertical* net. We will also represent a net N as $N = [l, r]$, where $l \leq r$, $l = min\{t, b\}$ and $r = max\{t, b\}$. We refer to l and r as the *left* and *right* terminals of N respectively. An instance of the channel routing problem (CRP) is a channel consisting of a rectangular grid and a set of nets whose terminals lie on the grid points of the (horizontal) parallel boundaries. The *local density* d_x at x is defined to be the number of nets $[l_i, r_i]$ such that $l_i \leq x < r_i$. The *density* d is given by $d = max_x\{d_x\}$. A routing in the knock-knee model consists of a set of edge-disjoint paths (made up of gridline segments) connecting the terminals of each net. Hence a shared grid point could be one of two types: crossing and knock-knee (Figure 1).

[1]Supported in part by NSA Contract No. MDA-904-85H-0015, NSF Grant No. DCR-86-00378 and by the Systems Research Center Contract No. OIR-85-00108

Figure 1: Types of shared grid points

Let $L_1, L_2, \ldots, L_t$ be a set of conduction layers stacked on top of each other such that L_1 is on the bottom and L_t is on the top. A *wiring layout* is an assignment of single layer to each routing segment such that (1) no two segments of two distinct nets share a grid point on the same layer, (2) a routing path may change layers at a via and (3) no wire can use a grid point on a layer which is between two layers with a via at that grid point. It is known that any routing in the knock-knee model can be realized with four layers ([BB]) and that three layers suffice for the channel routing problem ([PL]).

Given a routing of an instance of CRP, the *diagonal diagram* can be obtained by inserting a diagonal for each knock-knee, a half-diagonal for each bend. If we remove the half-diagonals, we obtain the *core* layout. It is known that a wire layout can be realized with three layers if its core can [PL]. A *partition grid* is a grid containing all the diagonals (see [PL] for a formal definition). A set P of edges of the partition grid is called a *legal partition* if the following properties hold:

1. Every internal vertex in incident on an even number of edges of P.
2. The set of diagonals in P is identical to that of the diagonal diagram.
3. None of the *forbidden patterns* in Figure 2 appear in P.

A legal partition of a core layout W exists if and only if W can be wired with three conducting layers.

We use the standard CREW (Concurrent Read Exclusive Write) shared memory model. All our results will be stated in this model. However, our algorithms have fast implementations on fixed-interconnection networks such as the mesh or the hypercube. For example, all the algorithms stated in this paper can be implemented on a $\sqrt{n} \times \sqrt{n}$ mesh in time $O(\sqrt{n})$, where n is the input length.

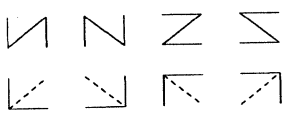


Figure 2: Forbidden Patterns

3 Channel Routing

Given an instance of CRP of density d, our goal is to determine a wiring of all the nets in d tracks. In addition, the resulting layout or a slight modification of it should be realizable in three layers.

The algorithm developed in [PL] constructs the wiring track by track by lying each track from left to right. The overall strategy can be viewed as a nontrivial extension of the *line packing* (or *left edge*) algorithm, where a mechanism is provided to solve conflicts arising in columns. This approach seems to be inherently sequential even if there is at most one terminal in each column. Our method is quite different and consists of two main steps:

1. Partition the nets into d chains satisfying certain properties to be outlined below. In particular, the nets in each chain define a set of nonoveralpping intervals.
2. Assign a track number to each chain. Then wire all the nets simultaneously.

We will outline how to perform each step next. The algorithm below creates chains of nets which will be modified later to satisfy all the desired properties. We will denote the successor (predecessor) of a net N by $succ(N)$ ($pred(N)$).

Algorithm Create Chains

Input: terminals l_i's and r_i's of all the nets $N_1, N_2, \ldots, N_n$.

Output: d chains of nets, where d is the density of the corresponding channel routing problem.

1. Mark all terminals as *active*. For each left terminal l_i of a net N_i, find the nearest right terminal r_j of some other net such that r_j is to the left (or in the same column) of l_i. If two such choices are possible, pick the one whose corresponding net is of the same type as N_i. Set $p(l_i) = r_j$. If no such r_j exists, then set $p(l_i)$ =nil. Similarily, define $p(r_i)$ for each right terminal.

2. If $p(l_i) = r_j$ and $p(r_j) = l_i$, then set $succ(N_j) = N_i$, and mark r_j and l_i as *inactive*. Create a reference point k between r_j and l_i.

3. Let $R_1, R_2, \ldots, R_m$ be the intervals determined by the reference points. For each R_i, create $L(R_i)$ consisting of all the active left terminals, and $R(R_i)$ consisting of all the active right terminals in R_i.

4. Find the corresponding terminal pairs in $R(R_i)$ and $L(R_{i+1})$ and create links as before. Mark all terminals used as inactive and merge intervals R_{2i-1} and R_{2i} for all i. Repeat this step until there is one interval left.

As an example, consider the channel routing instance of Figure 3. The chains produced by the above algorithm are given in Figure 4. We also have the following.

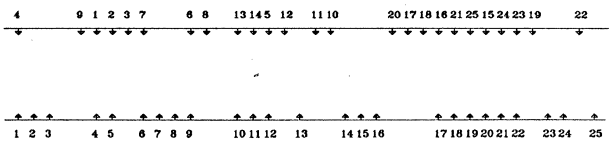


Figure 3: A channel routing problem

1. $N_1 \rightarrow N_{14} \rightarrow N_{15} \rightarrow N_{23}$
2. $N_4 \rightarrow N_7 \rightarrow N_8 \rightarrow N_{10} \rightarrow N_{16} \rightarrow N_{25}$
3. $N_2 \rightarrow N_5 \rightarrow N_{12} \rightarrow N_{18} \rightarrow N_{21} \rightarrow N_{24}$
4. $N_3 \rightarrow N_6 \rightarrow N_{13} \rightarrow N_{17} \rightarrow N_{19}$
5. $N_9 \rightarrow N_{11} \rightarrow N_{20} \rightarrow N_{22}$

Figure 4: The chains created by Algorithm Create Chains

Lemma1: The number of chains created by the above algorithm is exactly d, where d is the channel density. This algorithm can be implemented on the CREW-PRAM in time $O(\log n)$ with $O(n)$ processors, where n is the number of nets.

Proof : Let $R_1, R_2, \ldots, R_m$ be the intervals created by the above algorithm, prior to a set of merging operations of step 4, such that K_i is the reference point between R_{i-1} and R_i. Let n_{r_i}, n_{l_i} be respectively the numbers of active right and left terminals in R_i and let n_{k_i} be the number of nets with terminals on different sides of K_i.

Claim: The following inequalities hold true before each set of merging operations performed in step 4 of the above algorithm:

$$n_{r_i} + n_{k_{i+1}} \leq d$$
$$n_{l_i} + n_{k_i} \leq d$$

Proof of Claim: Notice that initially all active right terminals in R_i must be to the right of the rightmost left terminal l_j in R_i. If at the completion of step 3, $n_{r_i} + n_{k_{i+1}} > d$, then the density of the channel at a point between the right and left terminals of R_i is $\geq n_{r_i} + n_{k_{i+1}} > d$, which is impossible. Similarily we can establish the other inequality. We now show that after each set of merging operations, the inequalities will hold. Consider the merging of the intervals R_{2i-1} and R_{2i}. We know that $n_{r_{2i-1}} + n_{k_{2i}} \leq d$ and $n_{l_{2i}} + n_{k_{2i}} \leq d$. Let $c = min\{n_{l_{2i-1}}, n_{r_{2i-2}}\}$. We distinguish between two cases:

1. Suppose that $n_{l_{2i}} \geq n_{r_{2i-1}}$. Then the number of left terminals in the new merged interval $R_{i'}$ is given by $n_{l_{i'}} = n_{l_{2i-1}} + n_{l_{2i}} - n_{r_{2i-1}} - c$ and hence $n_{l_{i'}} + n_{k_{i'}} = n_{l_{2i-1}} + n_{l_{2i}} - n_{r_{2i-1}} + n_{k_{2i-1}} - c$. But $n_{l_{2i-1}} + n_{k_{2i-1}} = n_{r_{2i-1}} + n_{k_{2i}}$ and therefore $n_{l_{i'}} + n_{k_{i'}} = n_{l_{2i}} + n_{k_{2i}} - c \leq d$.

2. Suppose that $n_{l_{2i}} < n_{r_{2i-1}}$. Then the number of left terminals in the merged interval $R_{i'}$ will be $n_{l_{i'}} = n_{l_{2i-1}} - c$ and thus $n_{l_{i'}} + n_{k_{i'}} = n_{l_{2i-1}} + n_{k_{2i-1}} - c \leq d$.

In a similar fashion, we can establish the other inequality. This concludes the proof of the claim.

Let d' be the number of chains created by the above algorithm. Clearly, $d' \geq d$. At the termination of the algorithm, the number of chains is equal to the number of left terminals. Using the claim above, we deduce that $d' \leq d$ and hence $d' = d$.

We now establish the time and processor bounds. One can check that a couple of sorting steps and few simple operations will take care of step 1-3. Step 4 consists of $O(\log n)$ merging operations each of which can be done in $O(1)$ time.

The above chains can be used to wire all the nets in d tracks. However, the corresponding layout may not be realizable in three layers. We modify the above chains so that they have the following property. Let c be any column. Then either

1. c is empty, or
2. c contains one terminal, or
3. c contains two terminals of nets N_i and N_j. Let $N_i = < c, b_i >$ and $N_j = < t_j, c >$.
   - If both N_i and N_j are either right or left nets, then they both belong to the same chain and one is the successor of the other.
   - Suppose that N_i is a right net and N_j is a left net. The other case can be dealt with similarily. Let $N_i' = succ(N_i)$ and $N_j' = succ(N_j)$. Then they either share a column or the column of N_i' or N_j' which is closer to c has only one terminal (see Figure 5(b)).

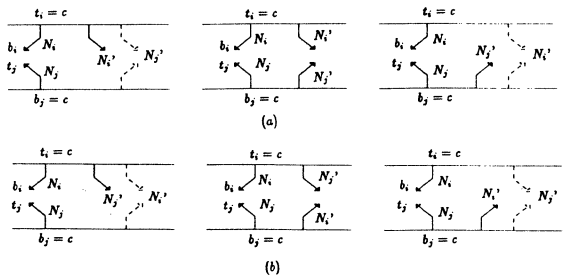


Figure 5: Possible successors of two nets with right terminals in the same column

The following algorithm outlines how to modify the chains so that the above property holds.

Algorithm Modify Chains

Input: A set of chains produced by the algorithm create chains.

Output: A set of chains satisfying the property stated above.

1. Mark each column with two right or two left terminals as active.

2. For each active column c with a top right terminal t_i and a bottom right terminal b_j, do the following:

- If the left terminals of $succ(N_i)$ and $succ(N_j)$ are in the same column c', then mark both c and c' as inactive.
- If the left terminals are in two distinct columns, say c' containing the left terminal of $succ(N_j)$ is the left one, then mark c inactive if c' has only one terminal.
- Otherwise, c' contains another left terminal b'_k. Let $N_k = pred(N'_k)$. Then create the pair $< N_i, N_k >$. Mark c and c' as inactive.

3. Group the pairs $< N_i, N_k >$ into maximal groups $< N_{k0}, N_{k1} >, < N_{k1}, N_{k2} >, \ldots, < N_{kt-1}, N_{kt} >$. Update the successors of these nets by setting the new successor of N_{ki} to be the previous successor of N_{ki+1} for all $0 \leq i < t-1$. In addition, set the new successor of N_{kt} to be the previous successor of N_{k0}.

4. Repeat procedure for active columns with two left terminals.

5. Adjust chains in such a way that whenever the configurations of Figure 5(a) occur, they will be replaced by the corresponding configurations of Figure 5(b) (similarily for columns with two left terminals).

As an example, consider the chains of Figure 4. Then the above algorithm creates the new set of chains given in Figure 6.

1. $N_1 \rightarrow N_6 \rightarrow N_{10} \rightarrow N_{16} \rightarrow N_{19}$
2. $N_4 \rightarrow N_7 \rightarrow N_8 \rightarrow N_{11} \rightarrow N_{20} \rightarrow N_{23}$
3. $N_2 \rightarrow N_5 \rightarrow N_{12} \rightarrow N_{18} \rightarrow N_{21} \rightarrow N_{24}$
4. $N_3 \rightarrow N_{14} \rightarrow N_{15} \rightarrow N_{22}$
5. $N_9 \rightarrow N_{13} \rightarrow N_{17} \rightarrow N_{25}$

Figure 6: New chains generated by Algorithm Modify Chains

Lemma2: The above algorithm modifies the chains generated by the algorithm Create Chains such that the new chains satisfy the desired properties. Moreover, the algorithm runs in $O(\log n)$ time with $O(n)$ processors on the CREW-PRAM model.

Proof: To simplify the presentation we will introduce a new graph called the *link graph*. There is vertex v_c corresponding to each column c. There is an edge between v_c and $v_{c'}$ if and only if c contains a terminal of a net whose successor or predecessor has a terminal in c'. Notice that the link graph of each of the groups created in step 3 has the form shown in Figure 7(a). If c'_0 has another link to a, then a cannot appear between c_0 and c_1. After the modifications performed in step 3 the link graph of the group will be of the form given in Figure 7(b) with 2 link loops or paths of length 2. Hence it is clear that after step 3 no column with two right terminals could cause any problem. Each group may have generated one column with two left terminals which donot satisfy the desired property. Then step 4 of the above algorithm takes care of all these columns (Figure 5). Step 5 insures that columns with two terminals will be of the form given in Figure 5(b). The time and processor bounds of the algorithm can be easily established.

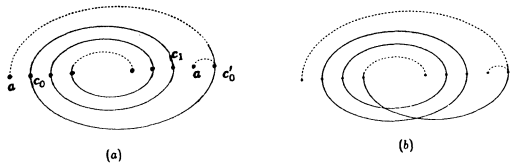


Figure 7: Forms of groups in the proof of Lemma2

The track assignment and the wire layout will be described next. Suppose that track k has been assigned to net $N = < t, b >$. Then the wire of N will consist of the interval $[t_k, b_k]$ on track k, a vertical line segment from b to b_k, and a vertical line segment from t plus a possible detour to t_k. Therefore the problem comes down to determining how to connect a terminal on the upper row down vertically to its track. The algorithm below describes how to achieve this.

Algorithm Wire Nets

Input: A chain of nets as modified by the algorithm Modify Chains.

Output: A wire layout for each net.

1. For each chain, assign the leftmost terminal l_i as the primary key, and, if l_i is a bottom terminal, assign 0 as the secondary key and 1 otherwise. Sort the chains according to their keys. The track number of each chain is its corresponding rank.
2. For each column c, do the following:
   1. if c contains one terminal of a net N, then connect that terminal vertically to the track of N.
   2. Suppose c contains two terminals of a single net. Then connect these two terminals vertically.
   3. Suppose that c contains two terminals of two distinct nets $N =< c, b >$ and $M =< t, c >$. If N and M have the same track number, then wire the terminals to this track using a knock-knee. Otherwise there is detour only if the track number of N is less than that of M. In this case, it is a left or right detour depending on whether c is a right or left terminal. The detour extends to either to the column of successor (for a right detour)or predecessor (for a left detour)of either N or M whichever is closer. All the cases that can arise and the corresponding routing are shown in Figure 8.

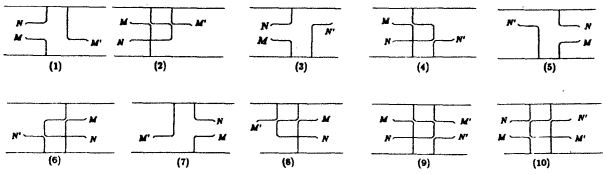


Figure 8: Possible detours of nets with terminals in the same column

Consider the example of Figure 2 again. Then the routing obtained by the above algorithm is given in Figure 9 .

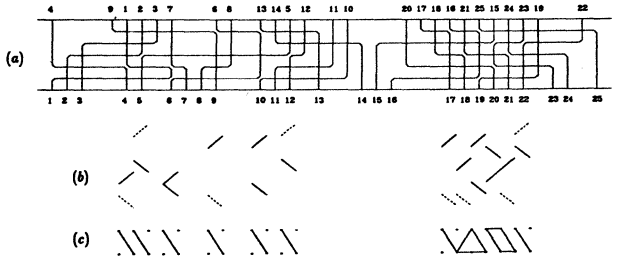


Figure 9: (a) The layout generated by Algorithm Wire Nets, (b) its corresponding diagonal diagram and (c) its corresponding constraint graph

Lemma3: Given an instance of the channel routing problem, the above algorithm provides a legal routing of all the nets in the knock-knee model.

Theorem1: Given an instance of the channel routing problem of density d, it is possible to wire all the nets in d tracks in time $O(\log n)$ time on the CREW-PRAM model with $O(n)$ processors, where n is the number of nets. If all terminals lie in the range $[1, N]$, where $N = O(n)$, then the above algorithm can be implemented in $O(n)$ sequential time and in $O(\frac{n}{p} + \log n)$ parallel time with p processors on the CREW-PRAM model, where $p \leq n^{1-\epsilon}$, and ϵ is any positive constant.

Proof: The first statement of the theorem follows from the previous lemmas. If all the terminals lie in the interval $[1, N]$, $N = O(n)$, then sorting (most expensive step) takes $O(n)$ sequential time. For the parallel implementation, the most expensive steps are sorting and traversing linked lists. Using the results of ([K et al]) we obtain the bounds stated in the theorem.

4 Layer Assignment

In this section, we show that a modified version of the routing produced by the algorithm of the previous section can be laid out in three layers. [PL] provides a necessary and sufficient conditions for the realization of a wiring in three layers. As stated in section2, the problem is essentially reduced to finding a legal partition of the core of the diagonal diagram. The routing layout produced by the algorithm in [PL] has a special property, namely every column is either empty or contains one diagonal or a diagonal \ on the bottom and a diagonal / above it. Their algorithm proceeds from left to right, looking at each column and making vertical connections (and possibly changing the routing) so that the resulting partition is legal. Unfortunately, we encouter a major difficulty in our case. Each column of our routing layout could have two diagonals (\ and /) in an arbitrary order (because our routing uses left and right detours). This makes it necessary to change the wire layout much more substantially than was done in [PL]. In the rest of this section, we outline how to overcome this difficulty.

By adding dummy diagonals if necessary, we can assume that each column is either empty or contains exactly two diagonals. As in [PL], our partition will be constructed by adding vertical edges only. Define a *reference line* as a vertical line that touches the endpoint of some diagonal. For each reference line, the diagonals touching this line will partition it into several line segments. Number these line segments starting from the top most segment. Notice that there are two possible ways of adding vertical segments (to create a legal partition): add the odd-numbered or the even-numbered segments. We have to choose (if possible) those segments that will not create a forbidden pattern.

We define the *constraint graph* as follows. The two possible choices of vertical segments corresponding to reference line L_i are represented by two vertices v_{2i-1} and v_{2i}. Two vertices are connected by an edge if and only if the corresponding choices create a forbidden pattern. Notice that forbidden patterns can be created only between adjacent reference lines.

Lemma4: The total number of the edges between the vertices corresponding to adjacent reference lines is ≤ 2.

Proof: Since the maximum number of diagonals between two adjacent vertical reference lines is 2, there are at most two "constraints" between $\{v_{2i-1}, v_{2i}\}$ and $\{v_{2i+1}, v_{2i+2}\}$, for each i.

Our goal is to pick for each reference line one of its vertices such that no two such vertices are connected by an edge. This may not be possible, in which case the routing layout has to be modified. We introduce the patterns that can create potential problems. A *forbidden column* is a pair of vertices corresponding to a reference line such that no selection of its vertices will lead to a legal partition. The set of configurations that *may* give rise to a forbidden column are shown in Figure 10.

Our goal is to modify the wiring layout if necessary so that the resulting constraint graph has no forbidden columns. We start by showing that any such graph will lead to a legal partition. The following algorithm shows how to select the proper set of vertices.

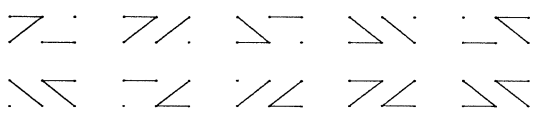


Figure 10: Configurations that may give rise to forbidden columns

Algorithm Select

Input: Reference lines and the corresponding constraint graph with no forbidden columns.

Output: A subset of the vertices which will induce a legal partition of the wiring layout.

1. Mark all reference lines as *active*. For each reference line L_i, select v_{2i} (v_{2i-1}) if v_{2i-1} (v_{2i}) is incident on two edges to a single adjacent column. If such a selection is made, mark L_i as inactive and assign weight 0 if v_{2i} is selected, otherwise assign weight 1.
2. Create a sorted list for each set of active reference lines between two inactive reference lines.
3. For each list created in step 2, do the following. Assign a weight 0 to each line L_k in the list if there is an edge between v_{2k-3} and v_{2k} or between v_{2k-2} and v_{2k-1}. Otherwise, assign a weight of 1 to L_k.
4. Calculate the rank of each reference line. Then select v_{2k} if the rank of L_k is even; otherwise select v_{2k-1}.

Lemma5: Given a partition graph with no forbidden columns, Algorithm Select will generate a subset of the vertices that determine a legal partition of the wiring layout.

Proof: Let's start by observing that the selection made in step 4 for inactive reference lines is consistent with that of step 1 because the graph contains no forbidden columns. For the rest of the proof, it is enough to show that there is a selected vertex for each reference line such that no two selected vertices are connected by an edge. The algorithm clearly selects exactly one vertex for each reference line. Suppose that there is an edge between two selected vertices, say v_{2k} and v_{k-2}. Then the weight of L_k must be 0 (because both have even ranks). But then either v_{2k} is connected to v_{2k-3} or v_{2k-1} is connected to v_{2k-2}. In the first case, v_{2k-1} would have been selected; in the second case, v_{2k-3} would have been selected. Similarily we can handle the other cases. Notice that the selection made in step 4 for inactive reference lines is consistent with that of step 1 because the graph contains no forbidden columns.

In the rest of this section, we will show how to modify the wiring in such a way that the corresponding constraint graph has no forbidden columns. We first introduce the following classification of reference lines (cf [PL]): *Trivial* (Figure 11), *Overlap* (Figure 12), *Disjoint* (Figure 13), *Inclusion* (Figure 14). Each type is shown with its possible constraint graph. The only possible forbidden columns could come from: D_1, D_3, D_6, D_8, I_2, I_4, I_6, I_8. In most of these cases, the wiring has to be modified by adding diagonals in such a way that no forbidden column could possibly arise. The procedure involves a detailed case study which is summarized by the following algorithm.

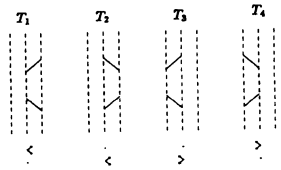


Figure 11: Trivial reference lines

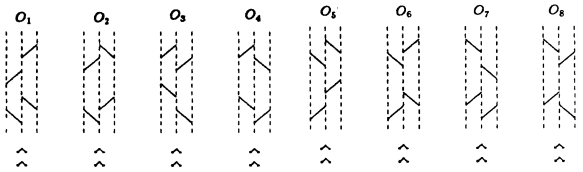


Figure 12: Overlap reference lines

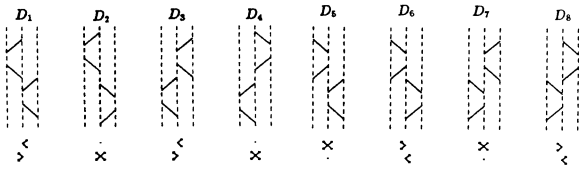


Figure 13: Disjoint reference lines

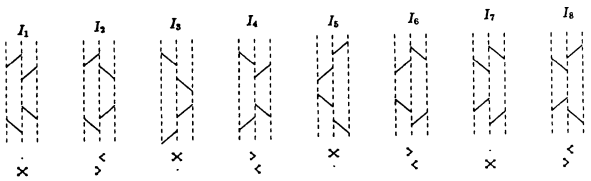


Figure 14: Inclusion reference lines

Algorithm Modify

Input: Wiring layout produced by Algorithm Wire Nets.

Output: A new wiring with its modified constraint graph and a set of selected vertices.

1. Generate the diagonal diagram, delete all half diagonals and add necessary dummy diagonals as follows. If there exists exactly one diagonal \, then add a dummy diagonal / in an additional row above all the rows. If there exists exactly one diagonal /, then add a dummy diagonal \ in an additional row below all the rows. Determine the constraint graph and mark all reference lines which may give rise to forbidden columns as active.

2. Handle type I_2 active reference lines as follows. Let $L_j, L_{j-2}, \ldots, L_{j-2k}$ be a maximal chain of active I_2's. We want to modify every other L_i starting with L_j in a way that depends on the type of its left neighbor L_{i-1}. All the cases that can arise are shown in Figure 15 with the corresponding modifications. In each such case, a vertex of L_{i-1} is selected (its degree is 0), edges between reference line L_{i-1} of selected vertex and its neighbors removed and the reference lines L_i, L_{i-1}, L_{i-2} are marked inactive. Handle type I_6 reference lines in a similar fashion.

3. Handle type active I_4 as shown in Figure 16. Select v_{2i} and remove edges between L_i and its neighbors. Mark L_i, L_{i-1}, L_{i+1} as inactive. Handle type I_8 similarily.

4. Handle active type D_1 as shown in Figure 17. Select v_{2i-1} and remove edges between L_i and its neighbors. Mark L_{i-1}, L_i, L_{i+1} as inactive. In Figure 18 a maximal chain of D_1's is considered. $L_i, L_{i+1}, \ldots, L_k$ are all of type D_1. If L_i or L_k can give rise to a forbidden column, then modify as shown and remove all edges of $L_i - L_k$. All the odd vertices of of $L_i - L_k$ are selected. As before edges are removed for selected columns and adjacent reference lines are marked inactive. Repeat the same procedure for types D_3, D_6 and D_8.

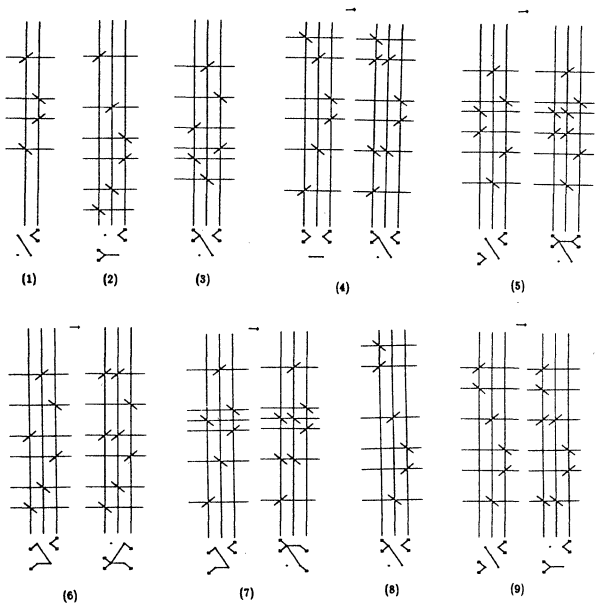


Figure 15: Transformations on type I_2 reference lines.

Lemma6: Algorithm Modify will change the wiring layout produced by Algorithm Wire Nets in such a way that the corresponding constraint graph contains no forbidden columns.

Proof: Consider the original constraint graph in which L_i was of type I_2 (hardest case). Then we have to show

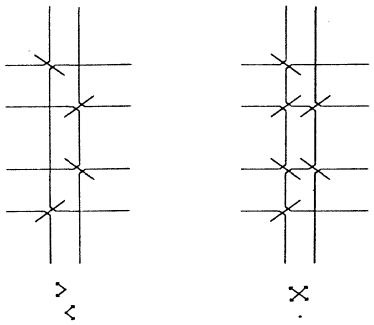


Figure 16: Transformations on type I_4 reference lines

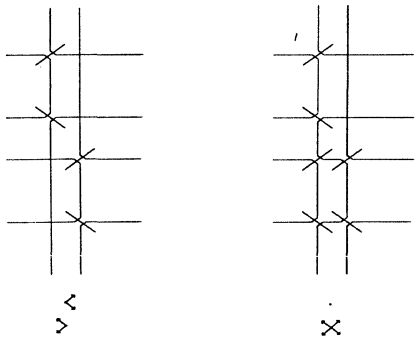


Figure 17: Transformations on type D_1 reference lines

that L_{i-3} will create no problems. The only nontrivial cases are the following:

1. L_{i-3} is of type I_2. In this case the algorithm selects vertices in the columns corresponding to L_{i-1} and L_{i-4} and hence there are no edges left between L_{i-2} and L_{i-1}, and between L_{i-3} and L_{i-4}.

2. L_{i-3} is of type I_6. Suppose that there are no dummy diagonals between L_{i-3} and L_{i-2} or between L_{i-1} and L_i. The only possible wiring configurations are shown in Figure 19 with their corresponding diagonal diagrams. If there is a dummy diagonal between L_{i-1} and L_i, then we can have one of the three possibilities shown in Figure 20. In each of these cases, one of L_i or L_{i-3} cannot generate a forbidden column.

3. L_{i-3} is of type I_4, I_8, D_1, D_3, D_6 or D_8. One can check that none of these cases can possibly generate a forbidden column.

The remaining cases can be dealt with similarily.

If we go back to the example of Figure 2, then the routing produced by the algorithm of the previous section is given in Figure 9. The layer assignment algorithm will change the wiring of N_{16} and N_{21} (Figure 21) and the final layout is shown in Figure 22.

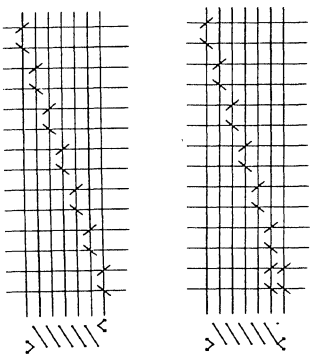


Figure 18: Maximal chain of D_1's.

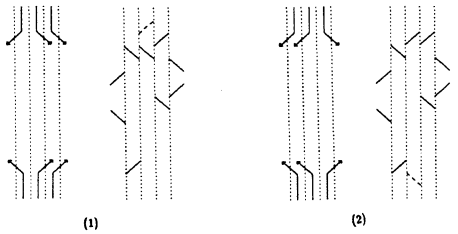


Figure 19: Possible wiring configurations for case 2 of lemma6

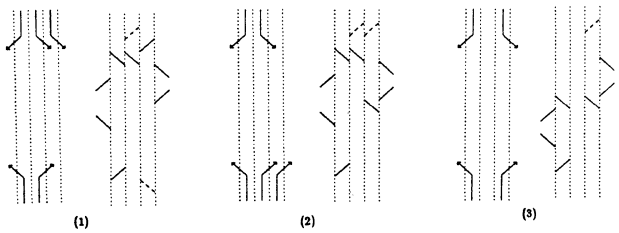


Figure 20: Possible configurations with dummy diagonals between L_i and L_{i-1}.

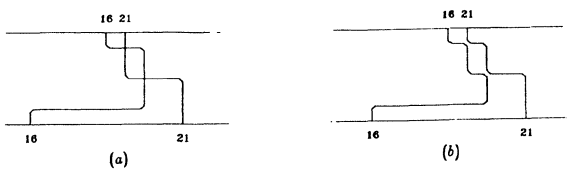


Figure 21: Changes in the wiring of N_{16} and N_{21}

Theorem2: Given an instance of the channel routing problem, it is possible to determine a three-layer assignment of the routing layout in time $O(\log n)$ time with $O(n)$ processors on the CREW-PRAM model. If all terminals lie in the range $[1, N]$, where $N = O(n)$, then the above algorithm can be implemented in $O(n)$ sequential time and in $O(\frac{n}{p} + \log n)$ parallel time with p processors on the CREW-PRAM model, where $p \leq n^{1-\epsilon}$, and ϵ is any positive constant.

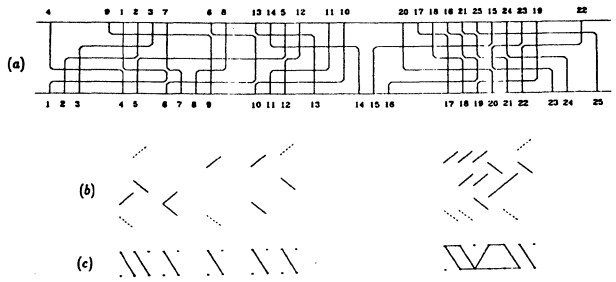


Figure 22: (a) The final layout after the modification of layer assignment algorithm, (b) its corresponding diagonal diagram and (c) its corresponding constraint graph

5 References

[BB] Brady, M. and D. Brown, "VLSI Routing: Four Layers Suffice," Advances in Computing Research 2 (VLSI Theory), ed. Preparata, JAI Press, Inc., Greenwich, CT, pp. 245-257, 1984.

[D et al] Dolev, D., K. Karplus, A. Seigel, A. Strong and J. Ullman, "Optimal Wiring Between Rectangles," Proc. 13th Annual ACM Symposuim STOC, May 1981, pp. 312-317.

[K et al] Kruskal, C., Rudolph, L. and M. Snir, "The Power of Parallel Prefix," IEEE Transactions on Computers, vol. C-34 (10), pp. 965-968, Oct. 1985.

[L] Lipski, W., "On the Structure of Three-Layer Wirable Layouts," Advances in Computing Research 2 (VLSI Theory), ed. Preperata, JAI Press, Inc., Greenwich, CT, pp. 231-243, 1984.

[MP] Melhorn, K. and F. Preparata, "Routing through a rectangle," JACM, vol. 33(1), Jan. 1986, pp.60-85.

[O] Ohtsuki, T., "Layout Design and Verification," Advances in CAD for VLSI, vol. 4, North-Holland, 1986.

[P] Pinter, R., "River Routing: Methodology and Analysis," Proceedings of the third CalTech conference on VLSI, March 1983, pp. 141-163.

[PL] Preparata, F. and W. Lipski, "Optimal Three-Layer Channel Routing," IEEE Trans. on Computers, C-33, pp. 427-437, 1984.

PARALLEL ALGORITHM FOR MINIMUM DUAL-COVER WITH APPLICATION TO CMOS LAYOUT

Y. M. Huang and M. Sarrafzadeh
Department of Electrical Engineering and Computer Science
The Technological Institude
Northwestern University
Evanston, IL 60208

Abstract — In a pair of planar graphs (G, G^d), with G^d being dual graph of G, a sequence of distinct edges is a dual-Euler trail if it is a trail both in G and in G^d. A set of disjoint dual-Euler trails that simultaneously cover G and G^d is called a dual-cover. We present an $O(\log n)$ time and $O(n)$ processors algorithm, in PRAM model, based on the graph separator theory, for obtaining a minimum cardinality dual-cover in a pair of series-parallel graphs (G, G^d), where n is the total number of edges. We employ the proposed algorithm to obtain a minimum-area VLSI layout of CMOS functional cells.

1 Introduction

Algorithm design is the development of better procedures and data structures to reduce the time to solve a given problem on a given computing system. Exploitation of a multiprocessor system requires a radical departure from the traditional Von Neumann environment. Detection of parallelism in sequential programs is essential to the discipline.

In the parallel-random-access memory model (PRAM) there is a group of processors, with access to a shared memory, cooperating to solve a given problem. An effective algorithm in PRAM model should aim to minimize the computation time and the number of processors.

Consider a planar graph $G = (V, E)$ along with its dual graph $G^d = (V^d, E^d)$, where there is a one-to-one correspondence between E and E^d, as shown in Figures 1a and 1b. A *trail* in G is a sequence of vertices $\tau = (v_a, v_{a+1}, \ldots, v_{b+1})$, where $e_i = (v_i, v_{i+1}) \in E$, $v_i \neq v_{i+1}$, and $e_i \neq e_j$ for $a \leq i, j \leq b$. To each trail τ we associate a label $\mathrm{L}(\tau) = (e_a, e_{a+1}, \ldots, e_b)$. Consider a trail τ of G and a trail τ^d of G^d. A pair $t = (\tau, \tau^d)$, with τ being a trail in G and τ^d being a trail in G^d, is called a *dual-Euler trail* (DET) if $\mathrm{L}(\tau)$=$\mathrm{L}(\tau^d)$. A set of disjoint DETs $\{ t_1, \ldots, t_s \}$ is called a *dual-cover* if $\mathrm{L}(t_i) \bigcap \mathrm{L}(t_j) = \emptyset$, for $i \neq j$, and $\bigcup_{i=1}^{s} \mathrm{L}(t_i) = E$. An optimal dual-cover of (G, G^d) is a minimum cardinality dual-cover, that is, a dual-cover with minimum s.

A CMOS functional cell consists of two parts: the p-part representing PMOS transistors, and the n-part representing NMOS transistors. Each transistor has a polysilicon strip; one side of the polysilicon strip being a source and the other side being a drain. The p-part is a series-parallel interconnection of PMOS transistors; similarly, the n-part is a series-parallel interconnection of NMOS transistors, and is the dual of the p-part. Representing the p-part and n-part interconnections by $G_p = (V_p, E_p)$ and $G_n = (V_n, E_n)$, respectively the $G_p = G_n^d$ and $G_n = G_p^d$. In CMOS circuits, it is possible to implement complex logic functions supported by complementary NMOS and PMOS transistors instead of by conventional NAND and NOR logic elements. The former implementation requires about half the area of the latter implementation, has less time delay and better performance.

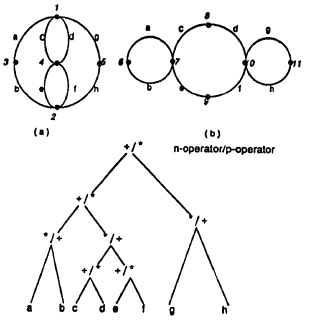


Figure 1: (a) NMOS graph. (b) PMOS graph. (c) Binary decomposition tree of (a)/(b).

A systematic approach to layouts of CMOS functional cells has been proposed by Uehara and VanCleemput [UV]; we will refer to it as *UV style*. A UV layout can be viewed as a set of vertical polysilicon lines corresponding to gates, and a set of horizontal metal lines, corresponding to interconnections among the transistors. A source or a drain of a transistor is connected to a source or a drain of another transistor either by horizontal metal lines or by adjoining their corresponding gates (their polysilicon vertical lines). The former requires metal connections; thus, it increases the height of the layout area. The latter does not require any connection.

Consider a UV layout. Let *a polysilicon pitch* be the minimum separation between two polysilicon lines and *a diffusion pitch* be the minimum separation between two diffusion regions. Two polysilicon strips with common source or drain have a polysilicon pitch separation; otherwise they have a polysilicon plus diffusion pitch separation. An optimal UV layout is obtained when the transistors are "chained" (i.e., placed adjacent to each other) in an "optimal" manner. It has been shown [UV] that an optimal UV layout corresponds to an optimal dual-cover of (G_p, G_n). A heuristic algorithm for obtaining a dual-cover of (G_p, G_n) has been proposed in [UV]. Subsequently, two optimal algorithms running in $O(|E_p|)$ time in the RAM model were proposed [NBR,MH]. If (G_p, G_n) does not have a single dual-cover, then the algorithm of [NBR] cannot produce a layout [WPF].

In this paper, we will show an $O(log|E_p|)$ time and

This work was supported in part by the National Science Fundation under Grant MIP-8709074.

$O(|E_p|)$ processors algorithm, in the PRAM model, for obtaining an optimal dual-cover of (G_p, G_n). As a subproblem, we will show how to separate a series-parallel graph $G = (V, E)$ using $O(1)$ time and $O(|E|)$ processors — an improvement over previous $O(log^2|E|)$ time and $O(|E|^{1+\epsilon})$ processors result, $\epsilon > 0$ [GM] (algorithm of [GM] works on arbitrary planar graphs). The proposed algorithm is based on the divide-and-conquer principle. Aim is to recursively partition (G_p, G_n) into two "equal-size" subgraphs using a dual-graph separation theory. Then the processors collectively obtain an optimal dual-cover in each subgraph and combine them to produce an optimal dual-cover of (G_p, G_n). The technique we use in the combination step is an extension of the Algebra proposed in [MH].

This paper is organized as follows. In Section 2 preliminary definitions and results are given. The proposed parallel algorithm, for obtaining an optimal dual-cover, is presented in Section 3. An application of the proposed parallel algorithm to optimal UV-style layout of CMOS functional cells is described in Section 4 and experimental results are included. Details of the proposed implementation are given in Appendix A.

2 Preliminaries

A *series-parallel* graph (SP graph) is constructed by recursively applying "series" and "parallel" connections. It is a subclass of *planar graph*. We will introduce an effective method for finding all dual-covers of a pair of SP graphs with a fixed topology (non-permutable topology).

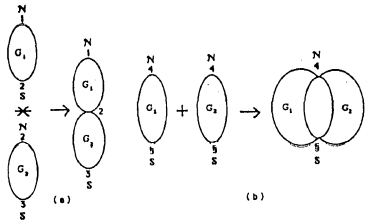


Figure 2: (a) A series connection. (b) A parallel connection.

2.1 Abstract Model

A Boolean logic function is modeled as a series-parallel graph $G = (V, E)$ with E corresponding to the input signals and V corresponding to the AND/OR operators. In each graph G, there are two *distinguished terminal vertices* labeled as N (the northern terminal) and S (the southern terminal).

Definition 1 : Two subgraphs G_1 and G_2 have a series connection if they have one common vertex, and have a parallel connection if they have two common vertices (see Figure 2).

Recursive combinations of a SP graph are described by a *binary decomposition tree* (BDT) T. Consider a SP graph $G = (V, E)$ and a BDT $T = (V_T, E_T)$. Each leaf of T corresponds to an *edge* of G and each internal vertex of T corresponds to a *combination* of two subgraphs G_1 and G_2, either in series (noted as $*$) or in parallel (noted as $+$). Let T_1 and T_2 be two BDTs corresponding to G_1 and G_2, respectively. The BDT T corresponding to SP graph $G = G_1 \# G_2$ has a vertex labeled $\#$ with T_1 and T_2 as its left subtree and right subtree, respectively (see Figure 1), where $\#$ *is used as a generic symbol for* $(+, *)$.

Consider a 2×2 *terminal-matrix* $\begin{bmatrix} N & S \\ N^d & S^d \end{bmatrix}$ corresponding to (G, G^d), where N and S are the two distinguished vertices of G, and N^d and S^d are the two distinguished vertices of G^d. Let two SP graphs G_1 and G_2 have terminal-matrices $\begin{bmatrix} N_1 & S_1 \\ N_1^d & S_1^d \end{bmatrix}$ and $\begin{bmatrix} N_2 & S_2 \\ N_2^d & S_2^d \end{bmatrix}$, respectively. A SP graph $G = G_1 * G_2$ has a terminal-matrix $\begin{bmatrix} N_1 & S_2 \\ N_1^d & S_1^d \end{bmatrix}$ if $S_1 = N_2$, $\begin{bmatrix} N_1 & N_2 \\ N_1^d & S_1^d \end{bmatrix}$ if $S_1 = S_2$, $\begin{bmatrix} S_1 & N_2 \\ N_1^d & S_1^d \end{bmatrix}$ if $N_1 = S_2$, $\begin{bmatrix} S_1 & S_2 \\ N_1^d & S_1^d \end{bmatrix}$ if $N_1 = N_2$. Since the dual SP graphs G_1^d and G_2^d are connected in parallel when G_1 and G_2 are connected in series, then $N_1^d = N_2^d$ and $S_1^d = S_2^d$. Similarly, a SP graph $G = G_1 + G_2$ has a terminal-matrix $\begin{bmatrix} N_1 & S_1 \\ N_1^d & S_2^d \end{bmatrix}$ if $S_1^d = N_2^d$, $\begin{bmatrix} N_1 & S_1 \\ N_1^d & N_2^d \end{bmatrix}$ if $S_1^d = S_2^d$, $\begin{bmatrix} N_1 & S_1 \\ S_1^d & N_2^d \end{bmatrix}$ if $N_1^d = S_2^d$, $\begin{bmatrix} N_1 & S_1 \\ S_1^d & S_2^d \end{bmatrix}$ if $N_1^d = N_2^d$. Since the dual SP graphs G_1^d and G_2^d have a series connection when G_1 and G_2 have a parallel connection, then $N_1 = N_2$ and $S_1 = S_2$.

2.2 Dual-Euler Trail

Consider a pair of graphs (G, G^d) and a dual-Euler trail t with $\mathrm{L}(t) = (e_a, e_{a+1}, \ldots, e_b)$. We call the starting and terminating vertices of a trail in G *boundary vertices*; similarly, we call the starting and terminating vertices of a trail in G^d *dual-boundary vertices* (or, for short, *d-boundary vertices*). Note that a DET t with $\mathrm{L}(t) = (e_a, e_{a+1}, \ldots, e_b)$ and its "reverse" t^r with $\mathrm{L}(t^r) = (e_b, \ldots, e_{a+1}, e_a)$ are equivalent. The boundary vertices v_a and v_{b+1}, and the d-boundary vertices v_a^d and v_{b+1}^d are used as the subscript of a DET label $\mathrm{L}(t) = (e_a, e_{a+1}, \ldots, e_b)_{(v_a, v_a^d) \rightarrow (v_{b+1}, v_{b+1}^d)}$.

Following [MH] we say (v_i, v_i^d) is a *terminal pair* if v_i is a boundary vertex of a DET t, v_i^d is a d-boundary vertex of the same DET t, and both v_i and v_i^d are distinguished terminal vertices of a pair of graphs (G, G^d). A DET is *distinguished* if it has at least one terminal pair and two DETs are *incompatible* if they cannot be "joined" with each other.

Each boundary vertex of a DET has type N, S, or I if it is the northern, the southern, or the internal vertex of the corresponding SP graph, respectively. A DET t has type $(\mathcal{T}_s, \mathcal{T}_s^d)/\ (\mathcal{T}_e, \mathcal{T}_e^d)$, where $\mathcal{T}_s$ and $\mathcal{T}_e$ are types of the boundary vertices, and, $\mathcal{T}_s^d$ and $\mathcal{T}_e^d$ are types of d-boundary vertices. A boundary and d-boundary vertices pair (v_i, v_i^d) can

be of type (N,N), (N,S), (S,N), (S,S), or (I,I) ((N,I), (S,I), (I,N), and (I,S) are included in (I,I)). Therefore, a DET has 25 possible types. Eliminating equivalent DET types (for example (N,S)/(S,S) is equivalent to (S,S)/(N,S)) and the (N,N)/(N,N), (N,S)/(N,S), (S,N)/(S,N), and (S,S)/(S,S) are four imposible DET types yields 11 possible types. Let Z denote incompatible DET types. The set of DET types are:

$$\Gamma = \{\ \begin{array}{lll} \text{(N,N)/(S,S)}, & \text{(N,S)/(S,N)}, & \text{(S,N)/(S,S)}, \\ \text{(N,N)/(S,N)}, & \text{(N,N)/(N,S)}, & \text{(N,S)/(S,S)}, \\ \text{(S,N)/(I,I)}, & \text{(S,S)/(I,I)}, & \text{(N,N)/(I,I)}, \\ \text{(N,S)/(I,I)}, & \text{(I,I)/(I,I)}, & \text{Z} \end{array} \ \}$$

Theorem 1[MH] : The triplet (Γ, +, $*$) form an Algebra.

Example : Consider Figure 1. A DET t_1 with $L(t_1)$= $(a,b)_{(1,6)\rightarrow(2,6)}$ has type (N,N)/(S,N), since its terminal-matrix is $\begin{bmatrix} 1 & 2 \\ 6 & 7 \end{bmatrix}$. Another DET t_2 with $L(t_2)$ = (c,d,f,e) $_{(4,7)\rightarrow(4,7)}$ has type (I,I)/(I,I), because its terminal-matrix is $\begin{bmatrix} 1 & 2 \\ 7 & 10 \end{bmatrix}$ and neither of the boundary and d-boundary vertices pairs is a terminal pair.

An (I,I)/(I,I) trail is called an *internal* DET. Note that an incompatible DET is not necessarily an internal DET, because two distinguished DETs cannot join together without compatible boundary and d-boundary vertices pairs.

Theorem 2 : There are at most four distinguished DETs in a dual-cover.

Proof : We recall the definition of a dual-cover. All the DETs in a dual-cover are disjoint incompatible DETs. Therefore, any two distinguished DETs t and t' in a dual-cover have types $(\mathcal{T}_s, \mathcal{T}_s^d)/(\mathcal{T}_e, \mathcal{T}_e^d) \neq (\mathcal{T'}_s, \mathcal{T'}_s^d)/(\mathcal{T'}_e, \mathcal{T'}_e^d)$. It causes a compatibility for two distinguished DETs t and t' when $(\mathcal{T}_s, \mathcal{T}_s^d)/(\mathcal{T}_e, \mathcal{T}_e^d) = (\mathcal{T'}_s, \mathcal{T'}_s^d)/(\mathcal{T'}_e, \mathcal{T'}_e^d)$, since the types $\mathcal{T}_s$, $\mathcal{T}_s^d$, $\mathcal{T}_e$, and $\mathcal{T}_e^d$ of a DET are constructed according to the same terminal-matrix. A graph can only have four distinct types of terminal pairs (N,N), (N,S), (S,N), and (S,S). Note that the (N,I) and (S,I) are not legal types of terminal pairs. These construct four distinct types of a maximum cardinality incompatible distinguished DETs (N,N)/(I,I), (N,S)/(I,I), (S,N)/(I,I), and (S,S)/(I,I) in a dual-cover. Any other distinguished DET in the same dual-cover is compatible with two of those DETs (e.g., a DET with (N,N)/(S,N) is compatible with the DET (N,N)/(I,I) and with the DET (S,N)/(I,I)), and this contradicts the definition of a dual-cover. We conclude that there are at most four distinguished DETs in a dual-cover. □

Let a *concatenation step* be the process of concatenating two dual-covers, that is, t_1 concatenates with t_2 if $L(t_1) \bigcap L(t_2) = \emptyset$ and t_1 and t_2 have a common vertex in G_1 and G_2 (or G_1^d and G_2^d). In the resulting DET t, $L(t)$=$L(t_1) \bigcup L(t_2)$.

Lemma 1 : An internal DET is not able to concatenate with any other DET.

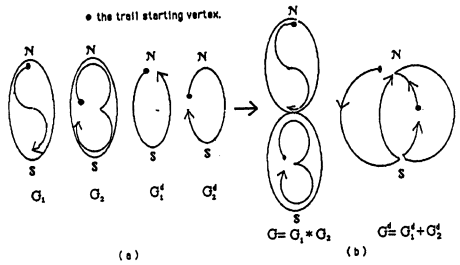


Figure 3: (a) An internal DET of (G_2, G_2^d) and a distinguished DET of (G_1, G_1^d).
(b) Incompatible DETs.

Proof : Consider an internal DET t_i of a pair of SP graphs and a distinguished DET t_j of another pair of SP graphs. The two pairs of graphs are joined at terminal vertices. Therefore, a DET must be distinguished and have compatible terminal vertices with other DET for concatenation. However, t_i has neither distinguished vertices nor compatible terminal vertices with t_j in the combination step. Therefore an internal DET is unable to concatenate with the other distinguished DET. □

An example showing incompatible DETs is depicted in Figure 3.

Let $Match(t_i, t_j) = 1$ if DET t_i is compatible with DET t_j ; otherwise $Match(t_i, t_j) = 0$. We define a *trail-match* to be the process of matching two distinguished DETs. According to Lemma 1 and Theorem 2, a concatenation step can be done in at most 16 trail-matches.

Let a *dual-cover type* δ represent a set of distinguished DETs types in a dual-cover. Each series-parallel operator # constitutes a pair of semigroup Algebras. Let Δ_0, Δ_1, Δ_2, Δ_3, and Δ_4 represent the five styles (consisting of 0, 1, 2, 3, 4 distinguished DETs) of dual-cover types. That is :

Δ_0 = { (I,I)/(I,I) }

Δ_1 = { (N,N)/(S,S), (N,S)/(S,N), (S,N)/(S,S), (S,S)/(I,I),
(N,N)/(N,S), (N,S)/(S,S), (S,N)/(I,I), (N,N)/(I,I),
(N,N)/(S,N), (N,S)/(I,I) }

Δ_2 = { [(N,N)/(S,N), (N,S)/(S,S)], [(N,S)/(S,N), (N,N)/(S,S)],
[(N,N)/(I,I), (S,S)/(I,I)], [(N,N)/(I,I), (N,S)/(I,I)],
[(N,N)/(I,I), (S,N)/(I,I)], [(N,N)/(I,I), (N,S)/(S,S)],
[(N,N)/(I,I), (S,N)/(S,S)], [(N,N)/(I,I), (S,N)/(N,S)],
[(S,S)/(I,I), (S,N)/(I,I)], [(S,S)/(I,I), (N,S)/(I,I)],
[(S,S)/(I,I), (N,N)/(S,N)], [(S,S)/(I,I), (N,N)/(N,S)],
[(N,S)/(I,I), (N,N)/(S,N)], [(N,S)/(I,I), (S,N)/(I,I)],
[(N,S)/(I,I), (S,N)/(S,S)], [(N,S)/(I,I), (N,N)/(S,S)],
[(S,N)/(I,I), (N,S)/(S,S)], [(S,N)/(I,I), (N,N)/(S,S)],
[(S,N)/(I,I), (N,N)/(N,S)], [(N,N)/(N,S), (S,N)/(S,S)] }

Δ_3 = { [(N,N)/(I,I), (N,S)/(I,I), (S,S)/(I,I)],
[(N,N)/(I,I), (N,S)/(I,I), (S,N)/(S,S)],
[(N,N)/(I,I), (N,S)/(I,I), (S,N)/(I,I)],
[(N,N)/(I,I), (S,N)/(I,I), (S,S)/(I,I)],
[(N,N)/(I,I), (S,N)/(I,I), (N,S)/(S,S)],
[(N,S)/(I,I), (S,S)/(I,I), (N,N)/(S,N)],
[(N,S)/(I,I), (S,S)/(I,I), (S,N)/(I,I)],

[(S,S)/(I,I), (S,N)/(I,I), (N,N)/(N,S)],
[(N,N)/(S,S), (N,S)/(I,I), (S,N)/(I,I)],
[(N,N)/(I,I), (S,S)/(I,I), (N,S)/(S,N)]}

Δ_4 = { [(N,N)/(I,I), (N,S)/(I,I), (S,N)/(I,I), (S,S)/(I,I)] }

Note that an internal-type DETs is not involved in Δ_1, Δ_2, Δ_3, and Δ_4. The type (I,I)/(I,I) in Δ_0 is a single-trail dual-cover.

Theorem 3[MH] : There are exactly 42 dual-cover types in a series-parallel combination.

Consider a set of SP graphs (G, G^d). Let a *dual-cover set* $\mathcal{D}$ be an optimal set of dual-covers with minimum cardinality. $\mathcal{D}$ is obtained by series or parallel combinations of two dual-cover sets $\mathcal{D}_1$ and $\mathcal{D}_2$ (i.e., $\mathcal{D} = \mathcal{D}_1 \# \mathcal{D}_2$).

Lemma 2 : No two dual-covers in a dual-cover set have the same dual-cover type except in Δ_0.
Proof : Each dual-cover set is an optimal set. Consider a set of dual-cover $\mathcal{D}$. As we mentioned in Lemma 1, the dual-cover $\mathcal{D}(i)$ with $\delta(i) \in \Delta_0$ is unable to combine with $\mathcal{D}(j)$, where $j \neq i$. If two dual-covers $\mathcal{D}(j)$ and $\mathcal{D}(k)$ have $\delta(j) = \delta(k)$ with $\delta(j) and \delta(k) \in \Delta_i$, where $i \in \{1,2,3,4\}$, they will have the same concatenations in the next combination step. This contradicts the definition of a dual-cover set. □

Lemma 3 : Each dual-cover has the smallest possible internal DETs.
Proof : Consider a dual-cover of a SP graph. Since it is a set of optimal disjoint DETs, then, except the distinguished DETs, all the internal DETs in the internal DETs set must be the smallest possible set and disjoint with each other. □

Lemmas 2 and 3, and Theorem 3 establish the following conclusion : *Every dual-cover set obtained by a combination step of two dual-cover sets contains at most 42 different dual-covers.*

We call the combinations of dual-covers $\mathcal{D}_1(i)$ and $\mathcal{D}_2(j)$ from two dual-cover sets $\mathcal{D}_1$ and $\mathcal{D}_2$ *a trailhunt* step, where $\mathcal{D}_1(i) \in \mathcal{D}_1$, $\mathcal{D}_2(j) \in \mathcal{D}_2$, $1 \leq i \leq |\mathcal{D}_1|$, and $1 \leq j \leq |\mathcal{D}_2|$. There are at most $42 \times 42 \times 16 = 28224$ trail-matches for a trailhunt. In fact, there is only one dual-cover with Δ_4 type in a dual-cover set. Moreover, usually much fewer than 42 dual-covers are included in an optimal set of dual-covers. Therefore, far fewer than 28224 trail-matches need to be performed in a trailhunt.

2.3 Graph Separator Theory

Consider a BDT $T = (V_T, E_T)$. Let a *cut-edge* e_c be an edge separating T into two "equal-size" sub-BDTs. There exists a cut-edge in every BDT [LT]. The edge e_c partitions T into $T_1 = (V_{T_1}, E_{T_1})$ and $T_2 = (V_{T_2}, E_{T_2})$, where $E_T = E_{T_1} \cup E_{T_2} \cup \{e_c\}$ and $\frac{1}{3}|V_T| \leq |V_{T_1}|, |V_{T_2}| \leq \frac{2}{3}|V_T|$.

Every vertex v_i in a BDT $T = (V_T, E_T)$, where $1 \leq i \leq |V_T|$ is the root of a (possibly empty) sub-BDT T_i. Consider the cut-edge $e_c = (v_c, v_d)$. We call v_c a *cut-vertex* if v_d is the parent of v_c. Two sub-BDTs T_1 and T_2 are obtained from T by removing the cut-edge. After the separation, the roots of T_1 and T_2 are v_c and the root of T (e.g., Figure 4a), or

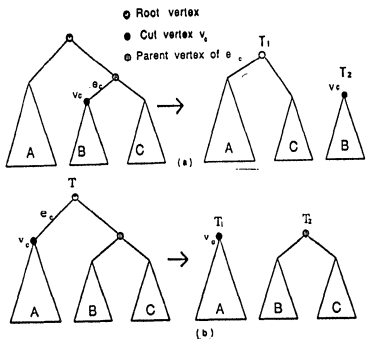


Figure 4: Two kinds of tree separation.

v_c and the other child of v_d (e.g., Figure 4b).

When T is separated into two "equal-size" sub-BDTs T_1 and T_2, the corresponding graph G is separated into two "equal-size" SP subgraphs G_1 and G_2 with T_1 being the BDT of G_1 and T_2 being the BDT of G_2. Subgraphs G_1 and G_2 have new terminal-matrices $\begin{bmatrix} N_1 & S_1 \\ N_1^d & S_1^d \end{bmatrix}$ and $\begin{bmatrix} N_2 & S_2 \\ N_2^d & S_2^d \end{bmatrix}$, respectively. Consider a SP graph $G = G_1 \# G_2$ with the terminal-matrix $\begin{bmatrix} N & S \\ N^d & S^d \end{bmatrix}$. If G_1 and G_2 have two common vertices, then # = '$*$', and, $N_1 = N_2$ and $S_1 = S_2$ which are not necessarily N or S of G (see Figure 5a). If G_1 and G_2 have one vertex in common, then # = '+', N_1 = N, $S_1 = N_2$, and S_2 = S (see Figure 5b). The same rules apply to G^d, G_1^d, G_2^d. For efficient implementation of trailhunt, the terminal-matrices of (G_1, G_1^d) and (G_2, G_2^d) have to be stored in order to decide the types of DETs.

A SP graph C=(V_C, E_C). Let $C = A \uplus B$ be the union of two SP graphs A=(V_A, E_A) and B=(V_B, E_B), where $V_C = V_A \cup V_B$ and $E_C = E_A \cup E_B$.

Example : Consider Figure 5a with G = A ⊎ B ⊎ C, when ⊎ denotes a composition of two graphs. When G is separated into G_1 = B and G_2 = A ⊎ C, we observe that the new boundary vertices of G_1 and G_2 are the same as the vertices being split by the separation line. Therefore, the terminal-matrices of G_1 and G_2 derived from G are described as follows : $\begin{bmatrix} 1 & 3 \\ 4 & 6 \end{bmatrix}_G \longrightarrow \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}_{G_1}$ & $\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}_{G_2}$. Again, consider Figuer 5b. As before, G = A ⊎ B ⊎ C. When G is separated into G_1 = A and G_2 = B ⊎ C, the separation line cuts G_1 and G_2 at vertex 2 of G. Hence, the terminal-matrices of G_1 and G_2 are not the same : $\begin{bmatrix} 1 & 3 \\ 4 & 6 \end{bmatrix}_G \longrightarrow \begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix}_{G_1}$ & $\begin{bmatrix} 1 & 2 \\ 4 & 6 \end{bmatrix}_{G_2}$.

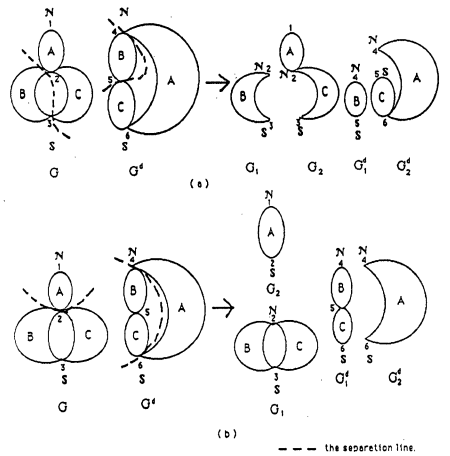


Figure 5: (a) A SP graph corresponding to Figure 4a.
(b) A SP graph corresponding to Figure 4b.

3 Parallel Algorithm for Minimum Dual-Cover

Utilizing the concepts discussed in Section 2, we will develop a parallel algorithm for solving subproblems of minimum dual-covers. After then, we integrate the algorithms to obtain a minimum dual-cover.

Here, we assume that a binary decomposition tree has been constructed (the construction of a BDT will be discussed in the next Section). We aim to employ the divide-and-conquer principle for separating the SP graphs and the corresponding BDTs. First procedure is called TREE SEPARATION which decomposes a BDT into two sub-BDTs, thus the corresponding SP graph will be separated into two subgraphs. The procedure TRAILHUNT combines two optimal dual-cover sets into one optimal dual-cover set. Each dual-cover set shows the optimal DETs of the corresponding SP graphs.

3.1 Tree Separation

In the procedure TREE SEPARATION, first we find a cut-edge and then separate the given BDT T= (V_T, E_T). In the last step we delete the leaves no longer belonging to the vertices on the path from cut-vertex up to the root. A formal description of TREE SEPARATION is given below.

Procedure TREE SEPARATION
begin
(1) **pardo** for all sub-BDTs T_i at vertices v_i
begin
if $\frac{1}{3}|V_T| \leq |V_{T_i}| \leq \frac{2}{3}|V_T|$
then $f_i :=$ TRUE;
else $f_i :=$ FALSE;
parend;
(2) select a cut-edge e_c from all v_i with $f_i =$ TRUE;
(3) separate the tree into two "equal-size" sub-BDTs by e_c;
(4) **pardo** for all tree vertices $v_j \in$ the path ($v_c \longrightarrow$ the root);
delete the leaves not belonging to T_j;
end;

Lemma 4 : TREE SEPARATION runs in $O(1)$ time and uses $O(|V_T|)$ processors.

Proof : Consider the BDT T= (V_T, E_T). Assume the tree path ($v_i \longrightarrow$ the root) and the leaves under v_i have been constructed, where $1 \leq i \leq |V_T|$. It is trivialy seen that Step 1 can be done in constant time using $|V_T|$ processors. Steps 2 and 3 run in constant time, as well. The last step takes constant time, for it involves cutting off the leaves under the sub-BDT T_C from the sub-BDT T_i while $i \in$ the path ($v_c \longrightarrow$ the root). Thus, we conclude that TREE SEPARATION runs in $O(1)$ time and uses $O(|V_T|)$ processors. □

The separation technique of [GM] can be used to separate the SP graph, too. But their algorithm, devised for arbitrary planar graphs, runs in $O(log^2|V_T|)$ time and uses $O(|V_T|^{1+\epsilon})$ processors, $\epsilon > 0$. In the series-parallel graph applications, our algorithm TREE SEPARATION is much better than the algorithm in [GM].

3.2 Trailhunt

Consider a SP graph $G = (V, E)$ and its BDT T=(V_T, E_T) with $|V_T| = 2|E| - 1$. When G is separated into $|E|$ single-edge subgraphs, T is decomposed into $|E|$ single-leaf sub-BDTs. The TRAILHUNT recursively combines two pairs of subgraphs and generates all possible concatenations from two optimal dual-cover sets. An optimal dual-cover covering new graph is thus obtained by applying TRAILHUNT recursively.

Consider two subgraphs G_1 and G_2 and their optimal dual-cover sets $\mathcal{D}_1$ and $\mathcal{D}_2$. A dual-cover $\mathcal{D}_1(i) \in \mathcal{D}_1$ with $\delta_1(i) \in \{\Delta_2, \Delta_3, \Delta_4\}$ might be compatible with $\mathcal{D}_2(k) \in \mathcal{D}_2$ having $\delta_2(k) \notin \Delta_0$ while a single-DET $\mathcal{D}_1(j) \in \mathcal{D}_1$ with $\delta_1(j) \in \Delta_1$ is incompatible with $\mathcal{D}_2(k)$. Consequently, except keeping the single-DET dual-covers we need to keep all the possible dual-covers $\mathcal{D}_a(i)$ that satisfy Lemmas 2 and 3 in a dual cover set $\mathcal{D}_a$.

For an optimal dual-cover set $\mathcal{D}_3 = \mathcal{D}_1 \# \mathcal{D}_2$, we define a function COMBINE($\mathcal{D}_1(i)$, $\mathcal{D}_2(j)$) for obtaining a combination of $\mathcal{D}_1(i)$ and $\mathcal{D}_2(j)$, where $1 \leq i \leq |\mathcal{D}_1|$ and $1 \leq j \leq |\mathcal{D}_2|$. Consider two distinguished DETs t_1 with $L(t_1) = (e_1, e_2, \ldots, e_n)_{(V_1, V_1^d) \rightarrow (V_{n+1}, V_{n+1}^d)}$ and t_2 with $L(t_2) = (e'_1, e'_2, \ldots, e'_m)_{(V'_1, V'^d_1) \rightarrow (V'_{m+1}, V'^d_{m+1})}$. A trail-match step checks the boundary and d-boundary vertices pairs of t_1 and t_2 (V_1, V_1^d), (V_{n+1}, V_{n+1}^d), (V'_1, V'^d_1), and (V'_{m+1}, V'^d_{m+1}). t_1 and t_2 are concatenated into one DET t if they match each other at the boundary vertices and the d-boundary vertices, that is, $(V_1, V_1^d) = (V'_1, V'^d_1)$, $(V_1, V_1^d) = (V'_{m+1}, V'^d_{m+1})$, $(V_{n+1}, V_{n+1}^d) = (V'_1, V'^d_1)$, or $(V_{n+1}, V_{n+1}^d) = (V'_{m+1}, V'^d_{m+1})$. Otherwise, they are incompatible.

After a COMBINE step, let d be the resulting dual-cover. In order for $\mathcal{D}_3$ to be an optimal set, every dual-cover in $\mathcal{D}_3$ needs to satisfy Lemmas 2 and 3. If any dual-cover $\mathcal{D}_3(k) \in \mathcal{D}_3$ has $\delta_3(k) = \delta(d)$, we choose the one with less internal DETs and discard the other non-optimal dual-cover. If no such dual-cover $\mathcal{D}_3(k)$ exists then d is included in $\mathcal{D}_3$.

In TRAILHUNT, first, COMBINE($\mathcal{D}_1(i)$, $\mathcal{D}_2(j)$) sequentially matches two distinguished DETs from $\mathcal{D}_1$ and $\mathcal{D}_2$ to generate a new dual-cover d. Then, d is checked against the restrictions imposed by Lemmas 2 and 3. If d satisfies the conditions then $\mathcal{D}_3 = \mathcal{D}_3 \bigcup \{d\}$; otherwise d is discared. Therefore, the optimality of $\mathcal{D}_3$ is ensured. Now, we give a formal description of TRAILHUNT algorithm.

Procedure TRAILHUNT($\mathcal{D}_1$, $\mathcal{D}_2$)
  begin
    for $i := 1$ to $|\mathcal{D}_1|$
      for $j := 1$ to $|\mathcal{D}_2|$
(1) **begin**
(2) d = COMBINE($\mathcal{D}_1(i), \mathcal{D}_2(j)$);
      if $\delta(d) = \delta_3(k), 1 \leq k \leq |\mathcal{D}_3|$ and
      |internal DETs of d| < |internal DETs of $\mathcal{D}_3(k)$|
      then begin
(2.1) $\mathcal{D}_3 := \mathcal{D}_3 - \{\mathcal{D}_3(k)\}$;
(2.2) $\mathcal{D}_3 := \mathcal{D}_3 \bigcup \{d\}$;
        end
      else if $\delta(d) \neq \delta_3(k), \forall k$, and $1 \leq k \leq |\mathcal{D}_3|$
(2.3) **then** $\mathcal{D}_3 := \mathcal{D}_3 \bigcup \{d\}$;
    end
  end;

Lemma 5 : TRAILHUNT runs in $O(1)$ time and uses one processor.

Proof : Size of a set of dual-cover, as proved in Theorem 3, is at most 42. Therefore, $|\mathcal{D}_1| \times |\mathcal{D}_2| \leq 1762$, that is, the number of iterations. Step 1 performs at most $4 \times 4 = 16$ trail-matches, because in Theorem 2 it was proved that there are at most 4 distinguished DETs in a dual-cover. Step 2 clearly takes constant time, for $|\mathcal{D}_3| \leq 42$. Steps 2.1 to 2.3 each takes constant time, as well. Thus total running time is $O(42 \times 42 \times (16 + 42)) = O(1)$. We conclude that TRAILHUNT takes $O(1)$ time and employs one processor.□

Example : Consider Figure 6. When the BDT in Figure 6a is separated into two sub-BDTs, the SP graphs in Figures 1a and 1b each is split into two subgraphs as shown in Figures 6b and 6c, respectively. Now, we take one possible dual-cover of (G_1, G_1^d) $\mathcal{D}_1(i)$ = { (b,a)$_{(2,7)\ \rightarrow\ (1,7)}$; (g,h)$_{(1,10)\ \rightarrow\ (2,10)}$ } and one possible dual-cover of (G_2, G_2^d) $\mathcal{D}_2(j)$= { (c,d) $_{(1,7)\ \rightarrow\ (1,10)}$; (e,f)$_{(2,7)\ \rightarrow\ (2,10)}$ } and combine them. As the SP graphs are described in DET forms, the trail-match steps are independent of whether the combination of two dual-covers is in series or is parallel. It is obvious that the boundary and d-boundary vertices pairs of $\mathcal{D}_1(i)$ and $\mathcal{D}_2(j)$ are matched : (1,7) of (a,b) and (1,7) of (c,d), (1,10) of (g,h) and (1,10) of (c,d), (2,10) of (g,h) and (2,10) of (e,f), and (2,7) of (b,a) and (2,7) of (e,f) are matched. A new dual-cover can be concatenated for example as (a,b,e,f,h,g,d,c)$_{(1,7)\ \rightarrow\ (1,7)}$ (see Figure 6d) or (b,a,c,d,g,h,f,e)$_{(2,7)\ \rightarrow\ (2,7)}$.

The following theorem is readily established by virtue of Lemmas 2 and 3.

Theorem 5 : Two dual-covers can be optimally combined in $O(1)$ time using one processor.

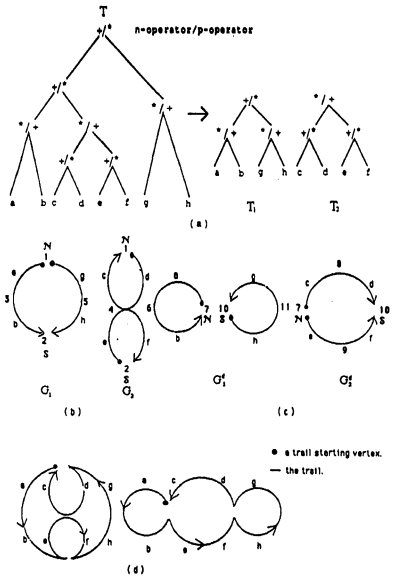


Figure 6: (a) The BDT of Figures 1a and 1b.
(b)(c) The separated SP subgraphs (G_1, G_1^d) and (G_2, G_2^d) of Figure(1a, 1b).
(d) One optimal dual-cover of Figures (1a, 1b).

3.3 Optimal Dual-Cover

We have derived a technique (TREE SEPARATION) for partitioning a pair of SP graphs in parallel. After log $|E|$ iterations of TREE SEPARATION the graphs $G = (E, V)$ and $G^d = (E^d, V^d)$ are partitioned into $|E|$ pairs of single-edge SP subgraphs. After then, these subgraphs are combined in parallel; after log $|E|$ iterations of TRAILHUNT the optimal dual-covers with minimum cardinality generated. In TREE SEPARATION and TRAILHUNT, the parallel separations and combinations are independent of the types of operations (series or parallel) in the corresponding SP graphs. The terminal vertices of SP graphs are of concern.

The algorithm OPTIMAL DUAL-COVER separates a BDT $T = (N_T, E_T)$ and the respective SP graphs $G = (V, E)$ and $G^d = (V^d, E^d)$ each $|E|$ sub-BDTs and $|E|$ pairs of subgraphs. Next, it combines the subgraphs to get the desired optimal dual-covers of (G, G^d). A one-edge dual-cover set is initialized as $\mathcal{D}$ = { (e)$_{(v_N, v_N^d)\ \rightarrow\ (v_S, v_S^d)}$; (e)$_{(v_N, v_S^d)\ \rightarrow\ (v_S, v_N^d)}$ }.

Procedure OPTIMAL DUAL-COVER(T)
  begin
    pardo for all active processors each associating
        with a sub-BDT T_i
    begin
(1) TREE SEPARATION;
(2) push terminal-matrix of T_i;
(3) set two new terminal-matrices of sub-BDTs T_{i_1} and T_{i_2};
(4) activate an available processor to perform the T_{i_2};
      pardo for T_{i_k} and $k \in \{1, 2\}$
(5) **if** $|V_{T_{i_k}}| > 1$
        then OPTIMAL DUAL-COVER(T_{i_k});
(6) **else** initialize the dual-cover set of T_{i_k};
      parend;

(7) pop terminal-matrix of T_i;
(8) TRAILHUNT($\mathcal{D}_{i_1}$, $\mathcal{D}_{i_2}$);
(9) release a processor;
end;
end;

Lemma 6 : OPTIMAL DUAL-COVER runs in $O(\log |E|)$ time and uses $O(|E|)$ processors with $|E|$ being the number of edges of the input SP graph.

Proof : With an input set of SP graphs $G = (V, E)$ and $G^d = (V^d, E^d)$, employing $|E|$ processors, the parallel algorithm OPTIMAL DUAL-COVER takes $O(\log |E|)$ iterations of TREE SEPARATION to get $|E|$ single-edge subgraphs. Additional $O(\log |E|)$ iterations of TRAILHUNT are required to combine these subgraphs to obtain the resulting dual-cover set. Step 5 is performed recursively. Steps 1 and 8 each takes constant time as proved in Lemma 4 and lemma 5, respectively. Steps 2 and 7 require constant time to access the shared memory. Step 3 utilizes the concepts introduced in Section 2.3 and is done in constant time. Steps 4 and 9 need constant time to acknowledge the processors, and Step 6 clearly requires constant time for initializations. We conclude that OPTIMAL DUAL-COVER runs in $O(\log |E|)$ time and uses $O(|E|)$ processors. □

4 Optimal Layout Of CMOS Functional Cells

In this Section, we will apply the algorithm OPTIMAL DUAL-COVER proposed in Section 3 to get optimal layouts of CMOS functional cells. We consider UV style [UV] layout of CMOS functional cells. As is customary, we assume the p-part and the n-part interconnections are series-parallel graphs with fixed topologies.

4.1 Graphs Models Of CMOS Circuits

Consider a pair of SP graphs (G, G^d) representing a CMOS circuit. Let G represent the *n-part* of CMOS circuit, and G^d represent the *p-part* of CMOS circuit. For instance, Figure 1a represents the NMOS transistors of Figure 7b and Figure 1b represents the PMOS transistors of Figure 7b.

4.2 Graph and Tree Transformations From Boolean Expressions

We assume the input is a Boolean expression representing the NMOS interconnections. In order to apply the OPTIMAL DUAL-COVER algorithm proposed in Section 3, the input function is transformed into a pair of series-parallel graphs (G_n, G_p) with $G_n = G_p^d$ and $G_p = G_n^d$, and the corresponding binary decomposition tree T is to be obtained. We find postfix notation most convenient for input representation.

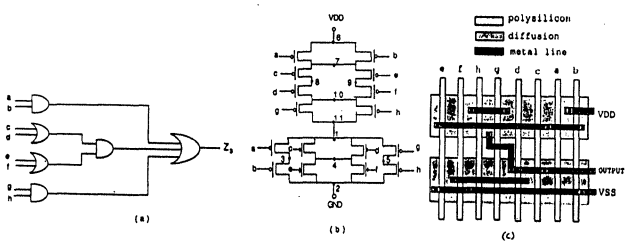


Figure 7: (a) A logic diagram with Z= $(a*b)+((c+d)*(e+f))+(g*h)$. (b) The CMOS circuit. (c) The optimal layout.

From the Boolean expression, we define $G_n = (V_n, E_n)$ representing the n-part, $G_p = (V_p, E_p)$ representing the p-part, and T= (V_T, E_T) representing the corresponding SP graphs (G_n, G_p). The number of transistors in p-part and n-part are $|E_p| = |E_n| = \frac{1}{2}(|V_T| + 1)$. Each vertex in (G_n, G_p) dictates the interconnection of sources and drains of a subset of transistors.

In BDT TRANSFORMATION, We define NODE[l] to be the lth input symbol, OP[i] to be the ith operator, and VAR[j] to be the jth variable. OP[i]'s two child- vertices can be of VAR[j−1] and VAR[j], VAR[j] and OP[i−1], or OP[i−1] and OP[i−k − 2] while one of the child-vertices dominates k operators. Each VAR[j] is a leaf of BDT. In BDT TRANSFORMATION, first, we scan the input string representing a Boolean function and store the symbols in OP[i] or in VAR[j] appropriately. Then each symbol links to its parent OP[p] with p $> i$ and p $\geq j - 1$, and points to its two child-vertices if this symbol is an operator.

Lemma 7 : BDT TRANSFORMATION runs in $O(\log n)$ time and uses $O(n)$ processors with n being CMOS gates.

The algorithm GRAPH TRANSFORMATION constructs a pair of SP graphs (G, G^d) by assigning a terminal-matrix to each BDT vertex, which is applied after the tree structures have been established. In the GRAPH TRANSFORMATION, M_T, M_T^l and M_T^r are defined as the terminal-matrices of a BDT T, of its left sub-BDT, and of its right sub-BDT, respectively.

Lemma 8 : GRAPH TRANSFORMATION runs in $O(1)$ time and uses $O(n)$ processors with n being CMOS gates.

By virtue of Lemmas 7 and 8, we conclude :

Theorem 4 : A set of SP graphs (G_n, G_p) and its corresponding BDT T are established in $O(\log |E|)$ time using $O(|E|)$ processors from an input Boolean expression, where $|E|$ is the CMOS gates.

Based on Theorem 4 and Lemma 6, OPTIMAL LAYOUT is used for obtaining the dual-cover set of SP graphs (G, G^d). The dual-covers with minimum cardinality of DETs minimize the CMOS layout area.

Procedure GRAPH TRANSFORMATION
begin
set the terminal-matrix of T_{root} to be $M_{root} := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$;
pardo for all $T_{OP[i]}$ with matrix $M_{OP[i]} := \begin{bmatrix} a & b \\ c & d \end{bmatrix}$;
begin
if OP[i] = '$*$'
then $M^l_{OP[i]} := \begin{bmatrix} a & f \\ c & d \end{bmatrix}$ & $M^r_{OP[i]} := \begin{bmatrix} f & b \\ c & d \end{bmatrix}$;
else $M^l_{OP[i]} := \begin{bmatrix} a & b \\ c & f \end{bmatrix}$ & $M^r_{OP[i]} := \begin{bmatrix} a & b \\ f & d \end{bmatrix}$, where $f \in V_{T_{OP[i]}}$;
parend;
end;

Procedure OPTIMAL LAYOUT
begin
(1) BDT TRANSFORMATION;
(2) GRAPH TRANSFORMATION;
(3) OPTIMAL DUAL-COVER;
(4) output the dual-covers with minimum cardinality of DETs;
end;

Lemma 9 : OPTIMAL LAYOUT runs in $O(\log n)$ time and uses $O(n)$ processors with n being the CMOS gates.

Figure 7c has different gates permutation from the layout in [MH], and has one metal tracks less than [MH]'s 6 tracks, which leads to a smaller area. Therefore, the optimal dual-cover is not unique. In fact some are preferred to others, and any arbitrary dual-cover may require a "large" number of tracks [S]. From Lemma 9, the following Theorem 6 is readily established.

Theorem 6 : An optimal UV style layout of a CMOS functional cell is obtained in $O(\log n)$ time using $O(n)$ processors in PRAM model with n being the CMOS gates.

4.3 Experimental Results

The divide-and-conquer algorithm outlined in this paper has been implemented in the C programming language on VAX/UNIX BSD 4.3 and the output is displayed on SILICON GRAPHICS IRIS 2400 work station. The bottleneck running time of this simulation program is TRAILHUNT. Therefore, we use *one processor* (VAX machine) to approximate the longest TRAILHUNT running time in OPTIMAL DUAL-COVER as an time unit, then multiply it by *log n* as the OPTIMAL DUAL-COVER running time shown in Figure 8. We also use the algorithm [GLL] which runs in $O(n \log n)$ time using one processor (RAM model) in our simulation program to compact the layout, that is, to compact the layout height.

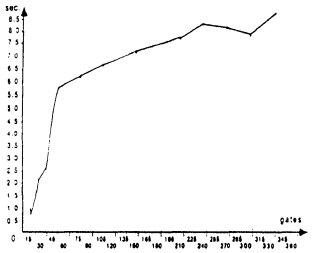


Figure 8: The OPTIMAL LAYOUT running time using n processors (n is the gates number).

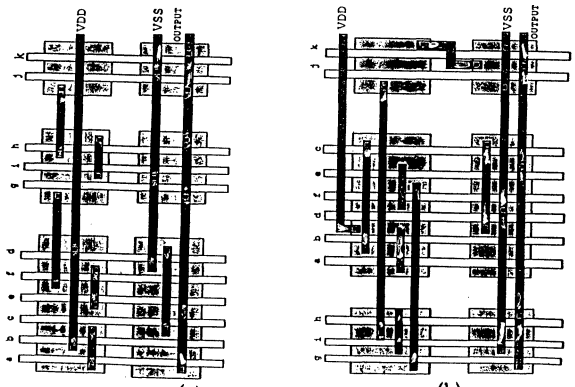


Figure 9: (a)An 11 gates CMOS layout using 6 tracks with input: a b * c + d * e f * g + h i * j + + k + + (b)A CMOS layout using 7 tracks in same input.

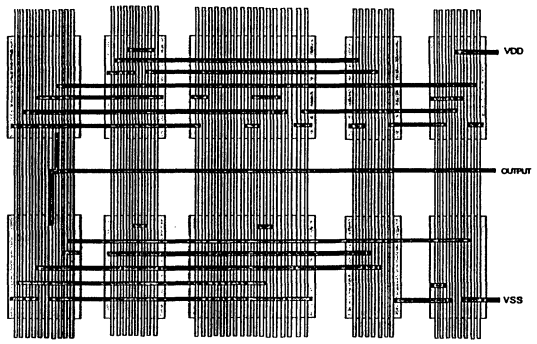


Figure 10: A 50 gates CMOS layout using 13 tracks.

Reference

[GLL] U. I. Gupta, D. T. Lee, and J. Y. Leung," An Optimal Solution for the Channel Assignment Problem", IEEE Transactions on Computers, Vol. C-28, No. 11, November 1979, pp. 807-810.

[GM] H. Gazit and G. L. Miller,"A Parallel Algorithm for Finding a Separator in Planar Graphs", Proceedings of 28*th* Symposium on Fundations of Computer Science, 1987, pp. 238-248.

[LT] R. J. Lipton and R. E. Tarjan, "A Separator Theorem for Planar Graphs", SIAM Journal on Applied Mathematics, Vol. 36 No. 2, April 1979, pp. 177-189.

[MH] R. L. Maziasz and J. P. Hayes, "Layout Optimization of CMOS Functional Cells", 24*th* ACM/IEEE Design Automation Conference, 1987, pp. 544-551.

[MR] G. L. Miller and J. H. Reif, "Parallel Tree Contraction and Its Application", manuscript, 1985.

[NBR] R. Nair, A. Bruss, and J. H. Reif, "Linear Time Algorithms for Optimal CMOS Layout", VLSI Algorithms and Architectures (P. Bertolazzi and F. Luccio, ed.), North-Holland, 1985, pp. 327-338.

[S] C. C. Su,"Optimal Gate-matrix Layout of CMOS Functional Cells", manuscript, Department of Electrical Engineering and Computer Science, Northwestern University,1987.

[TNS] K. Takamizama, T. Nishizeki, and N. Saito, "Linear Time Computability of Combinatorial Problems on Series-Parallel Graphs", Journal of ACM, Vol. 29, No. 3, July 1982, pp. 623-641.

[U] J. D. Ullman, Computational Aspects of VLSI, Computer Science Press, 1984.

[UV] T. Uehara and W. M. VanCleemput, "Optimal Layout of CMOS Functional Arrays", IEEE Transactions on Computers, Vol. C-30, No. 5, May 1981, pp. 305-312.

[WE] N. Weste and K. Eshraghian, Principles of CMOS VLSI Design A System Perspective, Addison Wesley, 1985.

[WPF] S. Wimer, R. Y. Pinter, and J. A Feldman, "Optimal Chaining of CMOS Transistors in a Functional Cell", IEEE transactions on Computer-Aided Design, Vol. CAD-6, No. 5, September 1987. pp. 795-801.

LOOKAHEAD IN PARALLEL DISCRETE EVENT SIMULATION

Richard M. Fujimoto[1]
Computer Science Department
University of Utah
Salt Lake City, UT 84112

Abstract

Empirical performance evaluations of parallel, discrete event simulation algorithms using deadlock avoidance and deadlock detection and recovery techniques developed by Chandy and Misra have been performed using the BBN Butterfly™ multiprocessor. Experiments using synthetic workloads reveal that the degree to which processes can look ahead in simulated time plays a critical role in the performance of distributed simulators using these algorithms. These results are applied to a queueing network simulation where as much as an order of magnitude improvement in performance is observed if the distributed simulator is programmed to fully exploit the lookahead available in the application. Performance measurements of several hypercube-based communication network simulators provide additional empirical data to support these claims. These results demonstrate that substantial improvements in performance are obtainable if the application can be programmed to have good lookahead characteristics. On the other hand, other applications *inherently* contain poor lookahead properties, and appear to be ill-suited for these simulation algorithms.

1. Introduction

Discrete event simulation has long been a task with computation requirements that challenge the fastest available computers. For example, simulations of communication networks, parallel computer architectures, and battlefield scenarios often require hours, days, or even weeks of CPU time using traditional, single processor techniques. Simulator performance may be improved using vectorizing techniques [Chan83a], processors dedicated to specific simulation functions [Comf84a], execution of independent trials on separate processors [Bile85a], or the execution of a single instance of a simulation program on a parallel computer. The last technique, referred to as distributed simulation, is the subject of this paper.

Simulation would initially appear to be a natural candidate for parallel processing because many of the aforementioned applications contain a high degree of parallelism. However, the exploitation of this parallelism is elusive because the global notion of simulated time does not easily map onto a distributed computer. This property distinguishes distributed simulation from other forms of parallel computation.

Several schemes have been proposed to solve this problem. A survey of the literature has been reported by Kaudel [Kaud87a]. One important class of distributed simulation algorithms is the so-called "conservative" mechanisms. Chandy and Misra developed a mechanism based on a deadlock avoidance technique where null messages are used to distribute clock information among the processes taking part in the simulation [Chan79a, Misr86a]. Another mechanism, also developed by Chandy and Misra, is based on a deadlock detection and recovery paradigm — the simulator runs until deadlock, the deadlock is detected, and an algorithm is executed to break the deadlock [Chan81a, Misr86a]. Other approaches to distributed simulation have been proposed, notably the Time Warp approach proposed by Jefferson [Jeff85a], but the work discussed here will be confined to deadlock avoidance and deadlock detection and recovery techniques.

In [Fuji88a] several experiments using synthetic workloads were described that were designed to evaluate the effectiveness of distributed simulation strategies using the deadlock avoidance and the deadlock detection and recovery algorithms. These experiments were performed on a distributed simulation testbed that was implemented on the BBN Butterfly,™ a shared-memory multiprocessor. Here, we apply these results to specific application problems to provide empirical data to support these results. In particular, parallel simulations of queueing networks and the communication subsystem of a hypercube-based multicomputer demonstrate the relationship between lookahead in the simulation application and performance of the parallel simulator.

[1]This work was supported by ONR contract number N00014-87-K-0184 and NSF grant number DCR-8504826.

2. Logical Processes, Activities, and Lookahead

Logical processes, activities, and lookahead form the basis for the synthetic workload model that is used here. The simulation program consists of some number of *logical processes*, each of which models some portion of the system being simulated. For example, in simulating a digital logic network, each gate (or some collection of gates) could be modeled by a logical process. Logical processes communicate exclusively by exchanging timestamped messages. Messages typically correspond to events that trigger a change in system state. Each logical process must process incoming messages in non-decreasing timestamp order to ensure that cause-and-effect relationships are faithfully reproduced by the simulator.

We informally define an *activity* as a sequence or thread of events that propagates among the logical processes in the simulation. These events model some sequence of cause-and-effect relationships in the system being simulated. For example, in a logic simulation, individual events are logic signal transitions and each activity corresponds to a signal propagating through a sequence of logic gates. In a queueing network simulation, each activity corresponds to a job traveling through the network. Activities are usually dynamic. A new activity is created in the logic simulation whenever an existing activity reaches a fanout point in the network. The activity disappears when (for instance) it reaches an AND gate with a logic zero on one of the other input lines. For our purposes, this informal definition of activities and logical processes will suffice.

Logical processes often "look ahead" into the simulated time future to schedule new events. For example, upon receiving a signal transition event in a logical process for an inverter gate, the process can predict and schedule a new event (a signal transition at the output of the gate) one gate delay later in simulated time. The lookahead abilities of the process determine how readily it will schedule new events. Processes such as the inverter with *good* lookahead abilities can "see" sufficiently far into the future that "effect" events can be scheduled as soon as the "cause" event is received. On the other hand, processes with poor lookahead ability must first wait until simulated time is advanced before they can schedule the effect event. For example, in a queueing network simulation with prioritized jobs, the "departure" event for a low priority job cannot be scheduled until it is first determined that no higher priority job will preempt it.

Quantitatively, lookahead is defined as follows: if a process has knowledge of all events that will occur up to simulated time T, and can predict all new events it will generate with timestamp $T+L$ or less, then the process is said to have lookahead L. In general, lookahead is a complex function that varies with time and the type of event, and is highly dependent on details of the simulation problem and the way it is programmed. A process can schedule a future event so long as the timestamp on that event is less than or equal to the process's local clock plus its lookahead. Such events are said to be within the "lookahead horizon" of the process.

Consider a "cause" event with timestamp T_{cause} that leads to an "effect" event with timestamp T_{effect}. The *absolute* value of lookahead is not as important as the lookahead *relative* to $T_{effect} - T_{cause}$, because this will determine how far the process must advance in simulated time to generate the new event. Therefore, we define a quantity referred to as the *lookahead ratio (LAR)*:

$$LAR = \frac{T_{effect} - T_{cause}}{lookahead}.$$

A *low* (e.g., 1.0) *LAR* corresponds to a *high* degree of lookahead.

3. The Distributed Simulation Testbed

An 18 processor BBN Butterfly multiprocessor was used for experimentation. Each processor node contains a 16 MHz MC68020 with MC68881 floating point coprocessor, 1 to 4 MBytes of memory, and a

Table 1. Hardware Parameters

Operation	Execution Time (microseconds)
Local memory reference	0.60
Remote memory reference	4.0
Register-to-register instruction	0.71
16 bit Load (Local Memory)	1.3
16 bit Load (Remote Memory)	6.3
Parameterless function call	6.9
Atomic inclusive OR	20

processor node controller (PNC), a microcoded engine that processes local and remote memory requests. The interconnection switch is configured as an Omega network. Atomic test-and-set like memory operations are also implemented in the PNC. Execution times of various instructions and operations are shown in table 1. Experimental data indicate that switch contention, and hot spot congestion in particular, is unlikely [Thom86a].

Each processor executes a single operating system process. This process is a scheduler that time multiplexes execution of the simulation processes mapped to the processor. This strategy avoids excessive context switching overhead, and allows more direct control over the process scheduling mechanism. Asynchronous message passing primitives were constructed using direct memory accesses to the mailbox in the receiving simulator process. Only a few simple Butterfly primitives, namely lock and atomic-add operations, are used by the testbed after initialization is complete.

4. The Simulation Algorithms

Two distributed simulation algorithms were implemented in the testbed: one based on deadlock avoidance and another based on deadlock detection and recovery. The shared memory architecture of the Butterfly was used to improve the efficiency of these algorithms, as described below. A single processor, event list implementation was also developed in order to compute speedup.

4.1 Deadlock Avoidance Strategy

The deadlock avoidance scheme developed by Chandy and Misra was implemented first. Each logical process sends a null message to each of its neighbors whenever it blocks. The timestamp on this message represents a lower bound of the timestamp on any message that will be sent to the receiver in the future. It is equal to the local clock value of the process plus the lookahead value because, by definition, the process cannot predict the occurrence (or non-occurrence) of events further into the future. Chandy and Misra have shown that this approach is sufficient to avoid deadlock [Chan79a].

In the testbed, one optimization was performed to streamline the processing of null messages. Rather than enqueueing each null message sent to another processor, a single variable is associated with each input link that contains the timestamp of the last null message that was received. This avoids unnecessary enqueue and dequeue operations and leads to more efficient memory utilization.

4.2 Deadlock Detection and Recovery Strategy

The second simulation approach is based on deadlock detection and recovery. The simulation runs until deadlock, the deadlock is detected, and an algorithm is initiated to break the deadlock [Chan81a]. A central controller is used to coordinate the deadlock recovery procedure.

Deadlock in the testbed is easily detected by maintaining a global counter indicating the number of processes that are either scheduled or running. The system is deadlocked whenever the counter reaches zero and there is at least one process that has not yet terminated (otherwise, the computation has terminated). Each scheduler checks the deadlock counter whenever it fails to find a process to run, and initiates a computation to break the deadlock if it finds the counter is zero.

The deadlock recovery algorithm locates the message in the system with the smallest timestamp and arranges for it to be processed next. A distributed algorithm is used to perform this computation. A central controller is used to coordinate this activity. By convention, the scheduler executing on PE 0 acts as the controller.

An alternative deadlock recovery algorithm was also implemented in which messages are propagated throughout the system in order to restart as many processes as possible. This algorithm is described in [Chan81a]. It was found, however, that the additional time required to execute this algorithm yielded a net loss in performance. The performance figures reported here are based on the former deadlock recovery approach.

4.3 Uniprocessor Simulation Algorithm

Finally, a single processor, event list simulator was developed to allow comparison of distributed simulation programs with sequential event list implementations. In order to obtain a fair comparison, the uniprocessor simulator was constructed by modifying the distributed simulator. Both implementations maintain the same overall structure, organization, programming style, and conventions. All code specific to parallel computation (e.g., synchronization locks) was eliminated.

The event list was implemented as a splay tree [Slea85a]. Empirical evidence suggests that splay trees are among the fastest methods for implementing an event list [Jone86a]. An alternative implementation using a singly linked linear list was also developed. It was found that this implementation yielded performance comparable to the splay tree for small simulations but, as expected, ran much more slowly for the larger simulations. The splay tree implementation is used in all comparisons with uniprocessor simulations reported here.

4.4 Performance Metrics

Three metrics are defined to evaluate the performance of the distributed simulation programs:

- **Speedup.** $SU(n)$, the speedup using n processors, is defined as the execution time of the single processor, event list implementation using a splay tree divided by the execution time of the distributed simulation program when n processors are used.
- **Null Message Ratio.** *NMR* is defined as the number of null messages processed by the simulator using deadlock avoidance divided by the number of real (non-null) messages processed. This measures the overhead of the deadlock avoidance approach.
- **Deadlock Ratio.** *DR* is the number of messages processed by the distributed simulator using deadlock detection and recovery, divided by the number of deadlocks that occur. This figure measures the efficiency of the deadlock detection and recovery algorithm.

The single processor execution times were obtained by running the splay tree simulator on a single node of the Butterfly. The same compiler as that used by the distributed simulator was used. Therefore, compiler and processor speed dependencies are factored out of the speedup figures.

The experiments were performed with no other applications running on the Butterfly. Facilities, such as the window manager, were run on processors different from those executing the simulation program. These measures were taken to minimize interference with the computation.

Experimental data were, for the most part, well behaved. The 95 percent confidence intervals for the measured data were typically less than one or two percent of the reported value. Only in a few instances were significant variations observed from one measurement to another. These were related to the avalanche effect described later, and do not affect the conclusions that follow from these experiments.

5. Experiments Using Synthetic Workloads

Synthetic workloads were constructed based on the notions of logical processes, activities, and lookahead, described earlier. Workloads contained 16 and 64 logical processes organized in 4 by 4 and 8 by 8 toroids, respectively (a toroid is a nearest neighbor mesh with wrap-around edge connections). Toroids were used because they do not contain inherent bottlenecks that might color the results, and because they are rich in cycles, and therefore represent a reasonably challenging configuration for the simulation algorithms. It is assumed that the number of activities in the simulation remains constant, and the lookahead of each process remains fixed throughout the simulation and does not depend on the type of event. Within each experiment, a fixed number of messages (the message population) circulates in a manner similar to jobs traveling throughout a closed queueing network. Simulation activity in each process was emulated using busy wait loops.

The experiments discussed next assume a message population of four messages per process and an average computation time of 1 millisecond (selected from a random variable with a negative exponential distribution) to process each incoming message. A static process to processor mapping

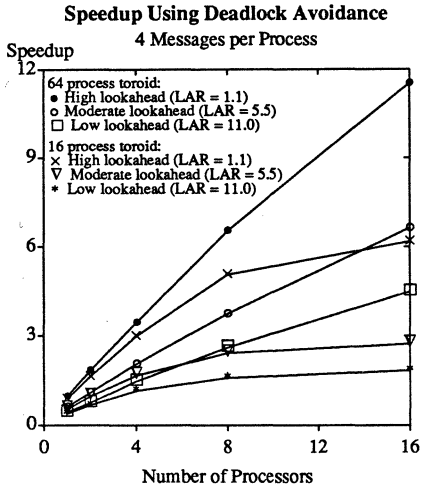


Figure 1. Speedup of synthetic workload as lookahead is varied.

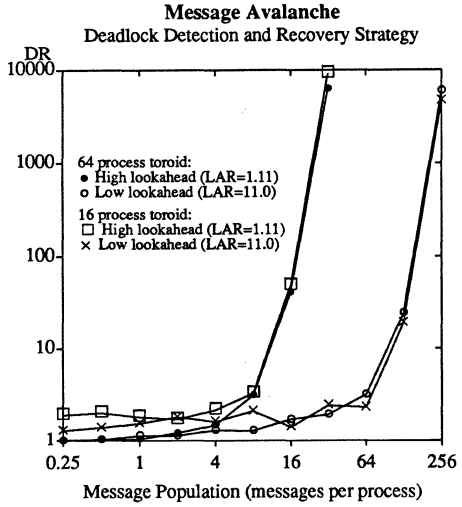


Figure 2. Message avalanche occurs as the message population is increased.

was used that balanced the workload assigned to the available processors while minimizing interprocessor communications.

Numerous experiments were conducted to examine the effects of computation granularity, dynamic load balancing, message population, message routing, and other factors. A detailed description of these results is beyond the scope of the present discussion, but is described elsewhere [Fuji87a, Fuji88a]. We will summarize some of these results and discuss how they can be applied to a specific application.

5.1 Effect of Lookahead

The speedup curves in figure 1 show the effect of varying lookahead in the deadlock avoidance simulator. As can be seen, lookahead plays a critical role in determining simulator performance. Performance degrades significantly as the lookahead ability of each process is reduced. Processes with poor lookahead characteristics must delay generating new events, reducing the amount of parallelism available in the simulation.

Performance of the 16 node toroid is somewhat less than the 64 node toroid because the simulation does not contain sufficient parallelism to keep all of the processors busy. In addition, as the number of processes per processor is decreased, each process is afforded less time to collect messages before it is executed by the scheduler. As a result, a process may be scheduled more often than if there were more processes mapped to the processor. The additional scheduling overhead and increased idle time lead to poorer performance in the 16 node simulator, particularly as the number of processors is increased.

5.2 Message Avalanche

Experiments using the deadlock detection and recovery strategy also revealed an "avalanche" phenomenon. This behavior is depicted in figure 2 where the deadlock ratio is plotted as a function of the message population. Performance remains poor (only a few messages processed between deadlocks) at low and moderate message populations, but then increases dramatically once message population reaches a certain critical level. It was found that message avalanche was a prerequisite for achieving good performance for this simulation strategy.

Message avalanche occurs when a message arriving at a process causes the transmission of one or more additional messages, which in turn trigger the transmission of still others, and so on. A multiplicative effect occurs whereby an "avalanche" of message traffic results from the original, accounting for the dramatic improvement in simulator efficiency.

As shown in figure 2, the message population required to induce avalanche was found to be dependent on the lookahead ability of the processes. Smaller populations were required to induce avalanche if processes were able to see far into the simulated future. This is again because poor lookahead characteristics reduce the amount of parallelism in the simulator.

5.3 Processes with Different Lookaheads

The experiments described above used homogeneous workloads where each process behaved in the same way as the others. Many real simulations contain a variety of logical processes with different lookahead characteristics. Additional experiments were performed in which some processes had poorer lookahead characteristics than the others.

Figures 3 and 4 show simulator overhead for the deadlock detection and recovery, and deadlock avoidance simulators, respectively, when some number of processes with poor lookahead characteristics are mixed with processes with good lookahead characteristics. Experiments were performed in which one, one fourth, one half, and finally all processes have poor lookahead (high *LAR*). Figure 3 indicates that the presence of a few processes with poor lookahead results in a perceivable performance degradation in the deadlock detection and recovery simulator (the avalanche point is moved to higher message populations). When a significant fraction of the processes have poor lookahead, performance is almost the same as that when all processes have poor lookahead. The deadlock avoidance simulator was found *not* to be as susceptible to such behavior (see figure 4), though some degradation results if a sufficiently high fraction have poor lookahead properties.

6. Queueing Network Simulations

To illustrate the applicability of the above results in a specific application, queueing network simulations were performed. A five process, central server network was simulated on the testbed. As shown in figure 5, this network contains three first-come-first-serve (FCFS) processes that service incoming jobs in the order in which they arrive, a fork process that stochastically routes each incoming job to one of its output ports (assume for now that either port is equally likely to be selected), and a merge process that combines streams of incoming jobs into a single output stream. Each server process also computes the average number of jobs in the server and reports this figure to the user.

Simulation and empirical studies by Seethalakshmi and Reed respectively concluded that the central server network is ill-suited for the conservative distributed simulation algorithms discussed here [Seet79a, Reed88a]. We reproduce and explain the poor results that these researchers observed in terms of message population and lookahead, and utilize this knowledge to improve performance.

The "classical" implementation of the FCFS process uses two types of events: arrival events (scheduled by other processes) denote jobs arriving at the server, and departure events (scheduled by the FCFS process itself) denote jobs completing service. The actions executed by the server process for each event type are shown in figure 6. *NJobs* indicates the number of jobs currently residing in the server, and *ServiceTime* indicates the time required to service each job. Code for computing statistics is not shown.

The classical server process has very poor lookahead properties. This is because it will not transmit an arrival event message with timestamp *TS*

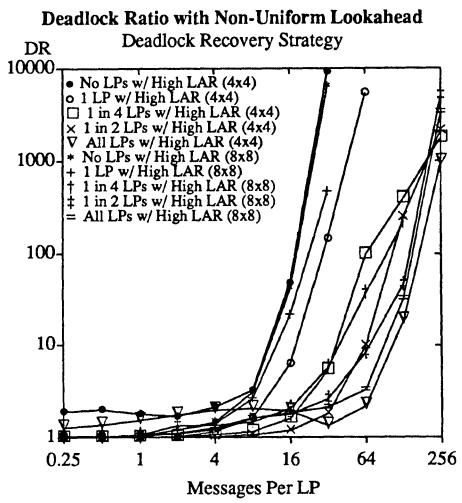


Figure 3. Overhead with non-uniform lookahead — deadlock recovery.

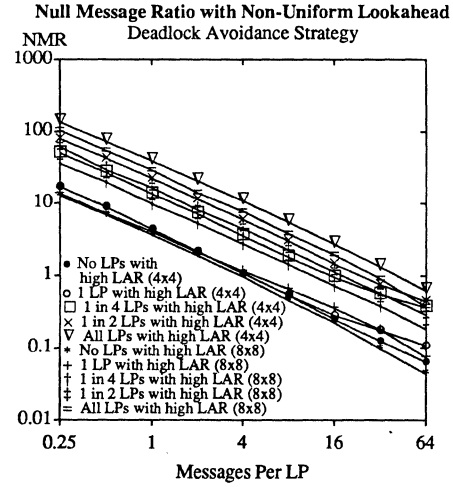


Figure 4. Overhead with non-uniform lookahead — deadlock avoidance.

until it has first advanced its local simulated time clock to *TS* by processing a departure event. In effect, it has a lookahead value of zero.

The lookahead properties of the FCFS process can be improved by eliminating the departure event, and generating a new arrival event as soon as one is received. Because an FCFS queueing discipline is used, the departure time can be determined as soon as the message is received. The optimized program is shown in figure 7. *EndService* denotes the time at which the server process will become idle if no additional jobs are received in the future. This program exhibits very good lookahead abilities because it can schedule events far into the simulated time future.

6.1 Performance Using Identical Servers

Simulators using each of these server programs were developed and executed on the Butterfly testbed. In all of the experiments described below, each logical process was mapped to a separate processor, and static scheduling was used. Service times for server processes were selected either deterministically or from a random variable with a negative exponential distribution.

The resulting speedup and simulator efficiencies for the central server queueing model using the deadlock detection and recovery strategy are shown in figures 8 and 9, respectively. The deadlock avoidance simulator yielded similar speedups. As can be seen, reprogramming the server to have better lookahead characteristics dramatically improves performance. Speedup is improved by as much as an order of magnitude. These results are consistent with those obtained using synthetic workloads.

The performance results of the *classical* server process are qualitatively similar to those reported by Reed and Seethalakshmi. The servers used in

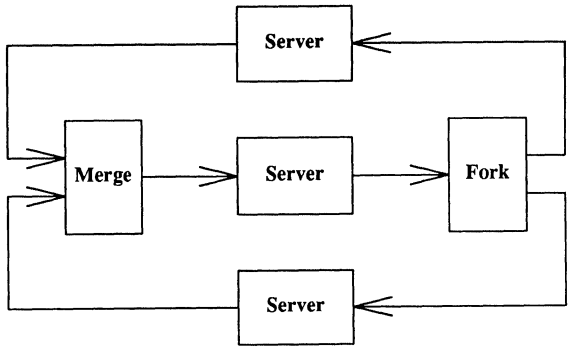


Figure 5. Central server queueing model.

```
ARRIVAL EVENT at TIME T:
    NJobs := NJobs + 1;
    IF (NJobs = 1) THEN  /* if server was previously idle */
        Schedule (local) Departure Event at time T + ServiceTime;

DEPARTURE EVENT at TIME T:
    Schedule (remote) Arrival Event at time T;
    NJobs := NJobs - 1;
    IF (NJobs > 0) THEN  /* if job(s) waiting in queue */
        Schedule (local) Departure Event at time T + ServiceTime;
```

Figure 6. "Classical" program for FCFS server (poor lookahead).

```
ARRIVAL EVENT at TIME T:
    IF (T < EndService) THEN  /* if server busy */
        BEGIN
        Schedule (remote) Arrival Event at time EndService+ServiceTime;
        EndService := EndService + ServiceTime;
        END
    ELSE  /* server idle */
        BEGIN
        Schedule (remote) Arrival Event at time T + ServiceTime;
        EndService := T + ServiceTime;
        END
```

Figure 7. Optimized program for FCFS server (good lookahead).

those studies are a variation of the classical server described above, and share the same (poor) lookahead properties — a message will not be forwarded until another message is first received with a timestamp at least as large as the departure time of the first. Therefore, lookahead provides an explanation for the poor performance that they observed.

Although the above results are encouraging, it is important to keep in mind that reprogramming the application to exhibit greater lookahead ability is not always possible. The above optimization relied on the servers using an FCFS scheduling discipline. As we shall soon see, many applications inherently contain poor lookahead properties.

Finally we note that, at first glance, reprogramming logical processes to maximize lookahead may complicate other aspects of the simulation, e.g., statistics collection. For example, the optimized server does not pause for departure events, so statistics that are most easily collected at job departure must be collected at other points in simulated time. This problem is easily reconciled by scheduling local departure events (as was done before) that are only used for statistics collection purposes.

6.2 Performance Using Mixed Servers

Additional experiments were performed to examine the effect of mixing processes with poor and good lookahead characteristics. Recall that experiments using synthetic workloads revealed that a small number of processes with poor lookahead could significantly degrade performance of the deadlock detection and recovery simulator. The deadlock avoidance simulator was found not to be as susceptible to such behavior.

The central server queueing network simulations were repeated where one of the three servers was implemented using the classical server

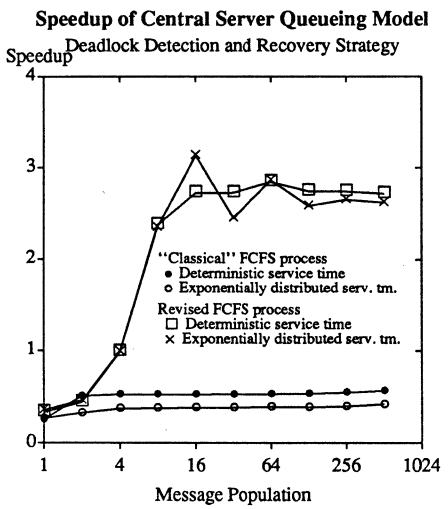


Figure 8. Speedup of central server queueing model.

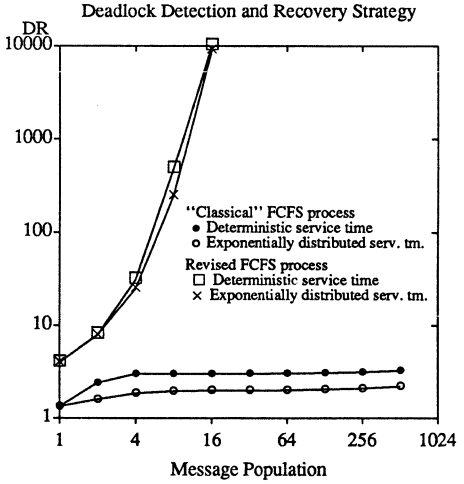


Figure 9. Overhead of central server queueing network simulator.

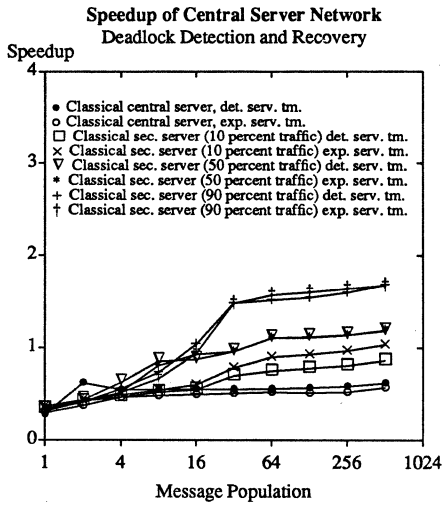


Figure 10. Speedup of detection and recovery simulator with one classical server.

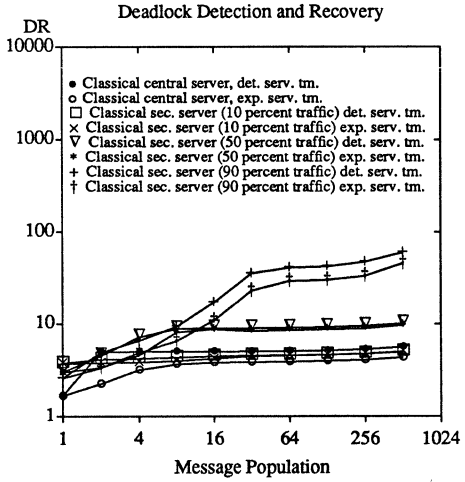


Figure 11. Overhead of detection and recovery simulator with one classical server.

program described earlier, and the remaining servers used the optimized program. The resulting simulator is not unlike one that would result if one of the servers was (say) a prioritized queue while the others were FCFS.

The speedup and efficiency of the deadlock detection and recovery simulator is shown in figures 10 and 11. When the central server (the process receiving messages from the merge process) has poor lookahead properties, performance is almost as poor as when *all* of the servers have poor lookahead. When one of the secondary servers (the servers receiving messages from the fork process) has poor lookahead, performance is better, but still well below that of the simulator using only optimized servers. These results are consistent with those obtained using synthetic workloads, and demonstrate that a few processes with poor lookahead can significantly degrade overall performance in the deadlock detection and recovery simulator.

When the classical program was used to implement a secondary server, the routing probabilities in the fork were modified so that 10, 50, and finally 90 percent of the message traffic was routed to the classical server. It is interesting to note that performance improves as *more* traffic is routed toward the server with poor lookahead. If little traffic is directed toward this server, the simulator is constantly deadlocking because the merge process is forced to block because it cannot determine whether or not it is safe to proceed without first receiving a message from this server. Routing additional message traffic toward this server helps the simulator to overcome (somewhat) the server's poor lookahead characteristics.

Speedup and overhead curves for the deadlock avoidance simulator are shown in figures 12 and 13. The deadlock avoidance simulator tends to be more forgiving of processes with poor lookahead. Poor performance results when the central server process has poor lookahead. However, performance begins to approach that of the optimized simulator in some situations where one of the secondary servers has poor lookahead. In particular, good performance is obtained if a significant fraction of the message traffic (50 to 90 percent) is routed *around* the process with poor lookahead. Unlike the deadlock detection and recovery simulator, null message traffic is generated by the classical server to allow the merge process to proceed. Because processes with poor lookahead tend to buffer messages rather than immediately forwarding them, it is best to minimize the amount of traffic routed to the classical server because this only detracts from the available parallelism.

7. Communication Network Simulations

Simulations of the message passing subsystem of a hypothetical multicomputer were also performed. The multicomputer is organized in a hypercube topology, and Sullivan's algorithm is used to route messages to their respective destinations [Sull77a]. Like the queueing network and synthetic workload experiments, a fixed message population was used to control the amount of available parallelism. Initially, each message is assigned a destination to which it is to be routed, and a message length. The destination is selected from a uniform distribution (excluding the

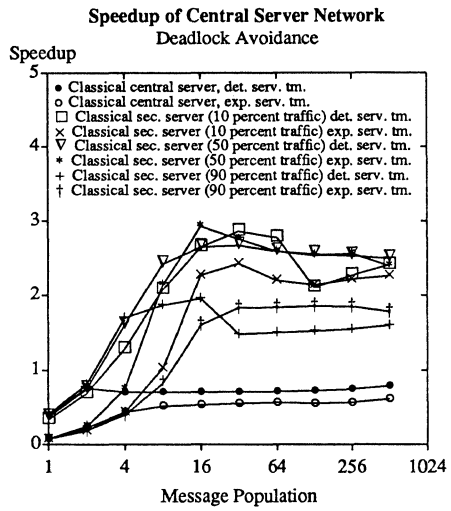


Figure 12. Speedup of deadlock avoidance simulator with one classical server.

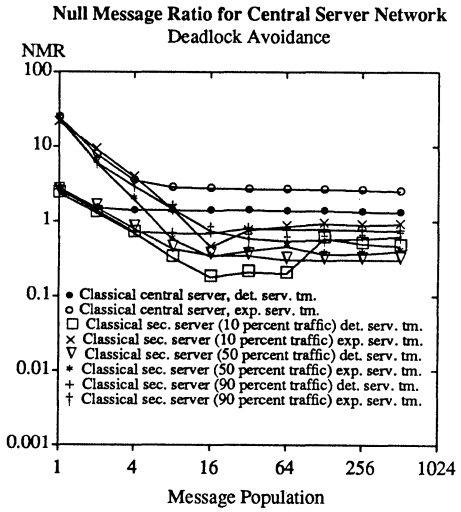


Figure 13. Overhead of deadlock avoidance simulator with one classical server.

processor where the message initially resides), and the message length is selected from an exponential distribution. When a message reaches its final destination, a new destination and message length are selected. All communication links in the hypercube are assumed to provide the same bandwidth. Three simulators were developed that contain varying degrees of lookahead, as will be described next.

7.1 A Simulator with High Lookahead

FCFS is a simulator in which messages are simply forwarded on the output link selected by the routing algorithm in FCFS order. Like the FCFS queueing network described earlier, this simulator has great lookahead ability because messages arriving at a logical process (with timestamp denoting the arrival time in the hypercube) can be immediately forwarded.

7.2 A Simulator with Moderate Lookahead

PRIO is a simulator with intermediate lookahead properties. Here, messages are classified as either high priority or low priority. Communication links in the hypercube give preference to high priority messages when selecting the next message to be transmitted. A low priority message is only forwarded if there are no high priority messages waiting to use the link. Messages within each priority level are processed in FCFS order. Each message is assigned a new priority whenever a new destination address and message length are selected and maintains this priority until it reaches the destination processor.

No preemption occurs in this simulator. Once the link begins forwarding a low priority message, it will continue to send it, even if a high priority message arrives before transmission is complete.

The parallel simulator for this system has intermediate lookahead properties. Logical processes have excellent lookahead for high priority messages, but poorer lookahead for those with low priority. Just as is the case for the *FCFS* simulator, high priority messages can be forwarded as soon as they arrive because the departure time can be immediately determined. However, a low priority message cannot be forward until simulated time in the logical process has advanced to the *departure* time (the time the hypercube *begins* sending the message) because it must first be determined that no high priority message will receive service ahead of it.

7.3 A Simulator with Poor Lookahead

The third simulator, *PREEMPT*, is identical to the *PRIORITY* simulator except that high priority messages preempt service of low priority messages. When a low priority message is preempted, it is assumed that the message must be completely resent once no other high priority messages remain that are waiting to use the link. The simulator for this system cannot forward a message to another logical process until simulated time has advanced to the *arrival* time (the time the *tail* of the message reaches the receiving hypercube node), so it has even poorer lookahead properties than the preceding simulator.

7.4 Performance Results

The hypercube simulations were performed on the Butterfly, and compared with execution of the sequential event list implementation. Unlike the previous experiments, these were performed on the Butterfly *Plus*, an upgraded version of the Butterfly that features 32 bit data paths (the original Butterfly has 16 bit data paths). The switch remains the same, so this effectively increases the cost of interprocessor communications. Because the simulation testbed already minimizes interprocessor communication, no program modifications were required. Experiments indicated that this hardware modification did not significantly affect the speedup measures derived earlier.

Overhead for these three simulators is shown in figures 14 and 15 for hypercubes of dimensions 4 and 6 (16 and 64 nodes respectively). Eight processors were used in these experiments. Upon reaching its destination, each message is assigned a high priority with probability P_{hprio}. In these experiments, P_{hprio} was selected to be either 0.01 or 0.50.

As predicted, the observed overhead steadily increases as the lookahead properties of the simulation are diminished. This is reflected in higher null message ratios in the deadlock avoidance simulator, and a larger message population required to induce avalanche in the detection and recovery simulator. Overheads are generally lower in the dimension four hypercube than the cube of dimension six for a fixed message population (as measured in messages per process) because there are fewer communication links; the simulators operate at peak efficiency when there is at least one message on each incoming link because no blocking occurs.

The lookahead properties of the simulator increase as P_{hprio} increases because more high priority messages are generated that can be forwarded as soon as they are received. This explains the lower overheads that were observed when P_{hprio} was increased.

Speedup curves for the hypercube simulators are shown in figures 16 and 17. Using eight processors, the parallel simulator executed anywhere from 5.7 times faster to nearly 20 times *slower* than the splay tree simulator, depending on the lookahead properties of the application. Some data points for very high message populations are missing because insufficient memory was available on a single processor to conduct an event list simulation.

The hypercube simulations provide additional evidence to support our contention that lookahead properties of the application are crucial to obtaining efficient performance for simulators using the deadlock avoidance and deadlock detection and recovery strategies. While the queueing network simulations demonstrated that it is possible to obtain dramatic speedups by reprogramming the simulation to fully exploit its lookahead properties, these experiments demonstrate that some simulations *inherently* contain poor lookahead, and cannot be improved by reprogramming. Such simulations appear to be poorly suited for the conservative simulation algorithms using deadlock avoidance and deadlock detection and recovery techniques, except in a few special circumstances such as networks that contain no feedback loops.

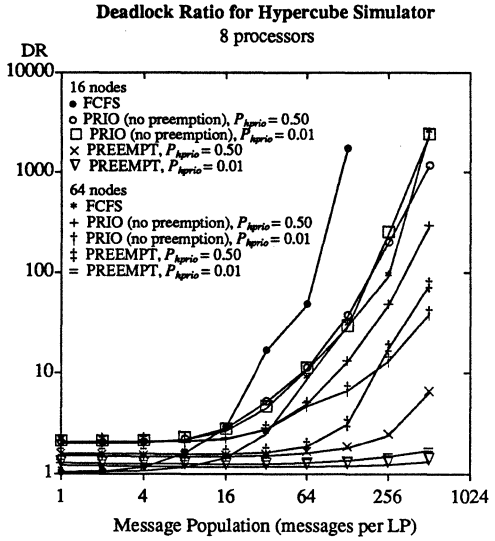


Figure 14. Overhead in hypercube simulator using deadlock recovery.

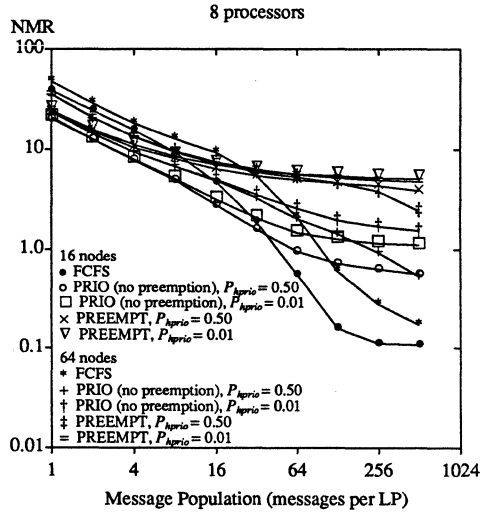


Figure 15. Overhead in hypercube simulator using deadlock avoidance.

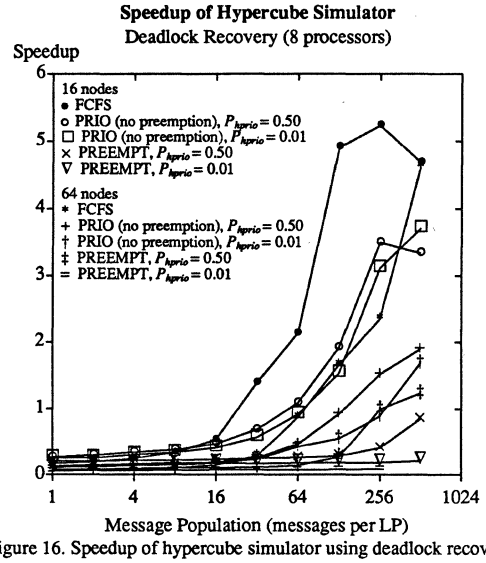


Figure 16. Speedup of hypercube simulator using deadlock recovery.

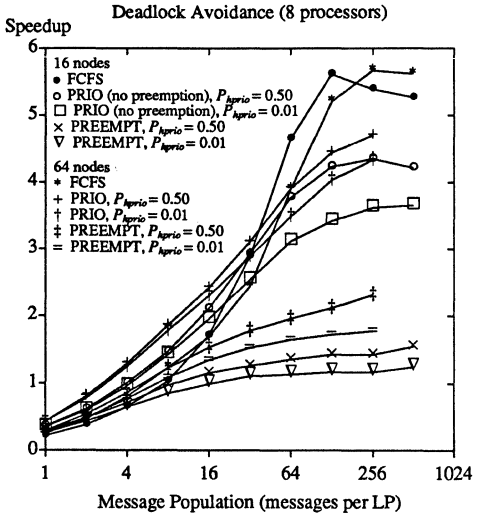


Figure 17. Speedup of hypercube simulator using deadlock avoidance.

8. A Perspective on Lookahead: Non-Events

The influence of lookahead on performance can be viewed from another perspective: processes with very good lookahead ability are able to act in a largely autonomous fashion; their behavior is not heavily influenced by the activities of other processes, so they can perform simulation work at "full speed," limited only by the rate at which they can be fed work, and the number of CPU cycles (or other resources) that they can obtain. The optimized queueing network server process is a good example of such autonomous behavior.

On the other hand, processes with poor lookahead ability must frequently obtain additional information from other processes before they can safely proceed. This is unfortunate because not only must such processes wait for real events to be generated by other processes (corresponding to data dependencies that cannot be circumvented), but often they must also wait to be sure other events will *not* occur. The fact that an airplane will *not* crash and close the airport in the next moment of simulated time must be discovered before the airport process can go about its business of deciding what *will* happen next. We call these "phantom" events that never materialize *non-events*. Chandy and Misra recently captured these notions in an elegant formalism called *conditional* and *unconditional* knowledge [Chan87a].

In the deadlock avoidance simulator, knowledge of non-events is passed explicitly through the use of null messages. In the deadlock detection and recovery simulator, this information is obtained by system deadlock — processes with messages waiting to be processed must wait until they can be certain that specific events will *not* occur. Certainty as to the eventuality of non-events comes about when the deadlock is broken, and the deadlock resolution protocol is invoked. Sequential, event list simulators incur little or no overhead for non-events.

If non-events are possible, but occur infrequently, the simulator is often forced to wait needlessly, leading to very poor performance. The hypercube simulator containing preemption and *few* high priority messages is one example of such behavior. Optimistic simulation methods such as Time Warp appear to offer the greatest potential for addressing this problem, if the associated state saving and rollback overheads can be overcome.

9. Conclusions

Extensive empirical performance evaluations of distributed simulation programs were performed using the deadlock avoidance and deadlock detection and recovery algorithms developed by Chandy and Misra. The principal results of these studies are:

- The lookahead ability of logical processes plays a critical role in determining the efficiency of the deadlock avoidance and deadlock detection and recovery algorithms. This is attributed to the fact that processes must spend an excessive amount of time waiting to be sure that certain events will *not* occur if their lookahead ability is poor.
- Message avalanche was observed in the deadlock detection and recovery simulator for moderate to high message populations, and was necessary to achieve efficient execution. The poorer the lookahead ability of a process, the larger the message population necessary to achieve avalanche. If lookahead is sufficiently poor, avalanche may never be observed for workloads of practical interest.
- Deadlock detection and recovery simulators containing different types of logical processes can be adversely affected by a small number of processes that exhibit poor lookahead ability. The existence of a few such processes can greatly increase the message population necessary to achieve avalanche, even if many other processes contain very good lookahead properties. The deadlock avoidance simulator is not as severely affected by this behavior if the bulk of the simulation activity avoids processes with poor lookahead.
- Queueing networks that contain cycles, previously thought to be ill-suited for conservative distributed simulation algorithms, can achieve good performance if servers are reprogrammed to take advantage of all available lookahead.
- Simulation applications such as those containing infrequent preemptive events *inherently* have poor lookahead properties, and appear ill-suited for these algorithms. Applications containing state dependent behavior (e.g., load balancing mechanisms) similarly contain moderate to poor lookahead properties.
- Simulations of several hypercube-based communication networks with varying degrees of lookahead provide empirical data to support the above conclusions.

These studies demonstrate that parallel simulation algorithms can achieve significant speedups over sequential event list implementations if a moderate to high degree of parallelism is present, even if there are many feedback loops in the logical process topology. However, good lookahead properties are essential to obtaining good performance in simulations using deadlock avoidance or deadlock detection techniques. The fact that a few processes with poor lookahead properties can significantly degrade performance also limits the usefulness of these approaches.

Because conservative simulation algorithms must continually predict what will *not* happen in order to be able to safely proceed, these studies raise considerable doubt as to whether *any* conservative parallel simulation algorithm can obtain significant speedup in applications containing poor lookahead properties. In these situations, optimistic simulation algorithms such as Time Warp appear to offer much greater potential for achieving significant speedups.

REFERENCES

[Bile85a] W. Biles, "Statistical Considerations in Simulation on a Network of Microcomputers," *1985 Winter Simulation Conference Proceedings*, pp. 388-393 (December 1985).

[Chan83a] A. Chandak and J. C. Browne, "Vectorization of Discrete Event Simulation," *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 359-361 (August 1983).

[Chan79a] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering* **SE-5**(5) pp. 440-452 (Sept. 1979).

[Chan81a] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM* **24**(4) pp. 198-206 (April 1981).

[Chan87a] K. M. Chandy and J. Misra, "Conditional Knowledge as a Basis for Distributed Simulation," Technical Report 5251:TR:87, Computer Science Department, California Institute of Technology (1987).

[Comf84a] J. C. Comfort, "The Simulation of a Master-Slave Event Set Processor," *Simulation* **42**(3) pp. 117-124 (March, 1984).

[Fuji87a] R. M. Fujimoto, "Performance Measurement of Distributed Simulation Strategies," Technical Report UUCS-87-026a, Computer Science Department, University of Utah, Salt Lake City, UT (November 1987).

[Fuji88a] R. M. Fujimoto, "Performance Measurement of Distributed Simulation Strategies," *Proceedings of the 1988 SCS Multiconference — Distributed Simulation, San Diego, California*, (February 1988).

[Jeff85a] D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems* **7**(3) pp. 404-425 (July 1985).

[Jone86a] D. W. Jones, "An Empirical Comparison of Priority-Queue and Event-Set Implementations," *Communications of the ACM* **29**(4) pp. 300-311 (April 1986).

[Kaud87a] F. J. Kaudel, "A Literature Survey on Distributed Discrete Event Simulation," *Simuletter* **18**(2) pp. 11-21 (June 1987).

[Misr86a] J. Misra, "Distributed-Discrete Event Simulation," *ACM Computing Surveys* **18**(1) pp. 39-65 (March 1986).

[Reed88a] D. A. Reed, A. D. Malony, and B. D. McCredie, "Parallel Discrete Event Simulation: A Shared Memory Approach," *IEEE Transactions on Software Engineering*, (to appear 1988).

[Seet79a] M. Seethalakshmi, "A Study and Analysis of Performance of Distributed Simulation," MS Report, University of Texas, Austin, Texas (May 1979).

[Slea85a] D. D. Sleator and R. E. Tarjan, "Self-Adjusting Binary Search Trees," *Journal of the ACM* **32**(3) pp. 652-686 (July 1985).

[Sull77a] H. Sullivan and T. R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine," *Proceedings of the 4th Annual Symposium on Computer Architecture* **5**(7) pp. 105-117 (March 1977).

[Thom86a] R. H. Thomas, "Behavior of the Butterfly Parallel Processor in the Presence pf Memory Hot Spots," *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 46-50 (August 1986).

A BLOCKED JACOBI METHOD FOR THE SYMMETRIC EIGENPROBLEM[a]

David E. Foulser
Computer Science Department
Yale University
Box 2158 Yale Station
New Haven, CT 06520

Abstract – A block matrix generalization of the Jacobi rotation method for computing the eigendecomposition of a symmetric matrix is presented. This Blocked Classical Jacobi (BCJ) algorithm selects for block rotation at each step the off-diagonal block(s) of largest mass. The BCJ algorithm exhibits substantially shorter runtimes than other Jacobi-like methods, even though it performs more work per iteration. A probabilistic analysis of the BCJ selection method is presented. Timings and other data are presented from experiments on random matrices.

1 Introduction

The class of Jacobi rotation methods [4,7,10,12] for computing the symmetric eigenvalue decomposition

$$A = UDU^T \tag{1}$$

of an $n \times n$ real matrix A, where U is orthogonal and D is diagonal, has generated substantial interest in recent years, particularly in the context of parallel computer architectures. Algorithms have been developed for systolic processor arrays as well as for more general purpose parallel computers. These methods differ principally from the original method of Jacobi in that they choose a fixed sequence of matrix elements for the necessary orthogonal rotations. Jacobi's method performs a rotation to zero out the largest off-diagonal element at each step; the sequence of rotations is data-dependent.

This paper presents a novel block matrix or "hypermatrix" adaptation [2,3,16] of the original algorithm, which we label the Blocked Classical Jacobi (BCJ) algorithm. The matrix A is treated as a smaller $m \times m$ matrix of $b \times b$ submatrices; computations work on entire submatrices rather than on scalars. Furthermore, the sequence of submatrices to be rotated is chosen to locally maximize the reduction of A to diagonal form by selecting the off-diagonal blocks of largest mass. BCJ reduces serial runtimes compared with other Jacobi methods and thus may prove useful where Jacobi methods are preferred over other eigensolvers, such as the QR algorithm.

For computers with a hierarchical memory system, in which successively larger yet slower memories are located at increasing distances from the arithmetic processor, many numerical calculations are efficiently structured in terms of block algorithms [3,8,15]. Rather than computing with scalar quantities, block algorithms operate on small square or rectangular submatrices of data. The resulting "surface-to-volume" effect of a single block data transfer followed by several computations allows a fast processor with local memory to achieve nearly full utilization even when supplied by a significantly slower bus or main memory.

The blocked organization of BCJ reduces the overhead cost of determining the maximum off-diagonal elements. It also makes BCJ especially well-suited for implementation on multiprocessors with a hierarchical memory system (*e.g.*, [8]). As well, BCJ is suitable for parallel implementation.

The organization of the paper is as follows. Section 2 gives a brief review of serial Jacobi methods for the symmetric eigenproblem. Section 3 gives the motivations for BCJ and presents the algorithm as implemented in this study. Section 4 lays out the numerical experiments with BCJ, including timings and numbers of iterations to convergence. Section 5 presents the analysis of the block selection method using the theory of order statistics, and discusses the implications for the experimental data. Concluding remarks and indications for parallel implementations are presented in section 6. Section 7 contains the proofs of two probabilistic results from section 5.

[a]Research supported by Office of Naval Research grant N00014-86-K-0310.

2 Review of Serial Jacobi Methods

The Jacobi method of solving (1) constructs a sequence of orthogonal rotations $U_1 = U(\theta_1, i_1, j_1, A^{(0)})$, $U_2 = U(\theta_2, i_2, j_2, A^{(1)}), \ldots$, such that $U = U_1 U_2 \cdots$ diagonalizes A (that is, $U^T A U$ is diagonal), $0 \le \theta_i \le \pi/4$, and $\lim_{i\to\infty} \theta_i = 0$. In practice the computation is terminated after a finite number of rotations, leaving $U = U_1 U_2 \ldots, U_N$. The rotation U_ν is selected to zero out the matrix elements in positions (i_ν, j_ν) and (j_ν, i_ν).

Given $(i,j) \equiv (i_\nu, j_\nu)$, the rotation angle θ_ν is computed so that $A^{(\nu)} = U_\nu^T A^{(\nu-1)} U_\nu$, according to

$$\begin{pmatrix} a_{ii}^{(\nu)} & a_{ij}^{(\nu)} \\ a_{ji}^{(\nu)} & a_{jj}^{(\nu)} \end{pmatrix} = \begin{pmatrix} c_\nu & s_\nu \\ -s_\nu & c_\nu \end{pmatrix}^T \begin{pmatrix} a_{ii}^{(\nu-1)} & a_{ij}^{(\nu-1)} \\ a_{ji}^{(\nu-1)} & a_{jj}^{(\nu-1)} \end{pmatrix} \begin{pmatrix} c_\nu & s_\nu \\ -s_\nu & c_\nu \end{pmatrix}, \tag{2}$$

with $a_{ij}^{(\nu)} = a_{ji}^{(\nu)} = 0$; here $A^{(0)} = A$. The cosine c_ν and sine s_ν of the angle θ_ν may be calculated by [9]

$$\tau = (a_{ii}^{(\nu-1)} - a_{jj}^{(\nu-1)})/(2a_{ij}^{(\nu-1)}), \qquad a_{ij}^{(\nu-1)} \neq 0, \tag{3}$$

then solving for t in

$$t^2 + 2t\tau = 1 \qquad \left(t = \frac{\text{sign}(\tau)}{|\tau| + \sqrt{1+\tau^2}}\right) \tag{4}$$

and substituting in

$$c_\nu = (1+t^2)^{-1/2}, \qquad s_\nu = c_\nu t. \tag{5}$$

U_ν is set to the identity matrix, except in rows and columns i_ν and j_ν, where it is zero everywhere but in the 2×2 principal submatrix; there it is $\begin{pmatrix} c_\nu & s_\nu \\ -s_\nu & c_\nu \end{pmatrix}$. If $a_{ij}^{(\nu-1)} = 0$ then c_ν is set to 1 and s_ν to 0, for θ_ν is obviously 0.

There are several methods for choosing the rotation index pair (i,j). The classical Jacobi method selects (i,j) at each step to locally minimize the resulting off-diagonal Frobenius norm by choosing (i,j) as the location of the largest off-diagonal element. However, the effort of determining the location of the maximum element ($O(n^2)$ operations) exceeds the work in calculating and applying the orthogonal rotation U_ν (approximately $18n$ operations neglecting symmetry). For this reason the method is rarely used on computers.

The cyclic-by-rows ordering of elements $((i,j) = (1,2), (1,3), \ldots, (1,n), (2,3), \ldots, (2,n), \ldots, (n-1,n))$ is more amenable to automatic computation. However, the successive index pairs are almost always dependent (sharing a row or column), and thus not suited for parallel computation. Parallel orderings have featured other index pair selections chosen for data locality and utility on a systolic processor. The Brent-Luk and Sameh orderings [4,13] have many desirable features. They preserve data locality and are amenable to systolic or other parallel implementations, they

converge faster than the cyclic-by-rows ordering, and they rotate each off-diagonal element exactly once in a "sweep." A particularly useful feature is that at each step, the $n/2$ independent rotations (operating on $n/2$ mutually distinct pairs of rows and columns) may be carried out simultaneously.

3 Algorithm BCJ

We now develop a blocked analogue of the classical Jacobi algorithm for the symmetric eigenproblem (1) that performs more work in selecting the index pairs yet requires less run-time than a blocked Brent-Luk ordering. BCJ also generalizes to computation of the singular value decomposition of a rectangular matrix. The new Blocked Classical Jacobi (BCJ) method selects the largest off-diagonal block(s) for rotation, in order to locally minimize the off-diagonal mass of A. Through a suitable choice of the block size, the extra computations to determine the off-diagonal block of maximum mass are offset by a reduced number of iterations; BCJ is more efficient on a serial computer than the other Jacobi methods tested.

BCJ is also highly parallel in nature. Where several processors are available to solve a single eigenproblem, the $K > 1$ largest independent off-diagonal blocks may be selected for simultaneous rotations, leading to a straightforward parallel implementation.

At each iteration, BCJ selects an off-diagonal block submatrix (i, j) for rotation and computes a block orthogonal rotation matrix $\hat{U}_\nu$, which it then applies to help reduce A to block diagonal form. The block orthogonal rotation can be chosen as a sequence of scalar Jacobi rotations or from the eigendecomposition of the small block matrix; we use a full scalar Sameh sweep on the small block matrix. (However, there is no restriction that the small block matrix must be diagonalized, only that its off-diagonal mass be reduced. Computations by Bischof [1] on the SVD indicate that the extra effort of completely diagonalizing the block matrix at each step may be wasted.) The method then proceeds by selecting another block element of A to rotate. A final processing step of Sameh sweeps forms the eigenvalues and eigenvectors from the block diagonal elements of A. On an $m \times m$ block matrix, a BCJ "sweep" is $(m^2 - m)/2$ two-block by two-block rotations.

The precise block algorithm for carrying out BCJ to compute the symmetric eigendecomposition (1), with D overwriting A, is as follows. Assume, for ease of exposition, that the block size b divides the matrix size n exactly, so that $n = mb$. $K \geq 1$ independent off-diagonal blocks may be selected for simultaneous rotation. The iterations continue until a tolerance criterion TOL is met. The method begins with $U = I$, $\nu = 0$, and continues

1. Compute the squared masses $\{M_{ij}\}_{i,j=1}^{m}$, with

$$M_{ij} = \sum_{r,s=1}^{b} \left(a^{(\nu)}_{(i-1)b+r,(j-1)b+s}\right)^2.$$

2. Select K independent rotation pairs $(i_k, j_k), 1 \leq k \leq K$ with

$$M_{i_k j_k} = \max(M_{ij} | i \notin \{i_1, \ldots, i_{k-1}\}, j \notin \{j_1, \ldots, j_{k-1}\}).$$

3. Compute K block rotations $\{\hat{U}(\theta_k, i_k, j_k, A^{(\nu)})\}_{k=1}^{K}$ to reduce the block off-diagonal mass of A (as indicated below).

4. Apply the block rotations of step 3 to U and to $A^{(\nu)}$, forming $A^{(\nu+1)}$.

5. If the block off-diagonal mass of $A^{(\nu+1)}$ is not less than TOL times the block diagonal mass, then set $\nu := \nu + 1$ and go to step 1.

6. Diagonalize the diagonal blocks of $A^{(\nu)}$ (until the off-diagonal mass is less than TOL times the diagonal mass) and update U.

Step 3 of our BCJ implementation uses a single scalar Sameh sweep to reduce the off-diagonal mass of the two-block by two-block submatrix. This sweep includes $b(2b-1)$ pointwise rotations performed sequentially. Step 6 uses successive Sameh sweeps to diagonalize the block diagonals of $A^{(\nu)}$.

The BCJ algorithm is to be compared against the "block Brent-Luk" algorithm, which omits step 1 and replaces step 2 by selecting $m/2$ block index pairs according to the Brent-Luk ordering. A block Brent-Luk sweep also involves $(m^2 - m)/2$ two-block by two block rotations. It is important to note that the two methods under comparison differ only in their index selection methods.

4 Experimental Results

Several numerical experiments were conducted to compare the efficiency of BCJ and blocked Brent-Luk symmetric eigensolvers on matrices of random data. The test matrices were generated as matrices of uniform random deviates from (0,1]; in each case 10 tests were run to give non-parametric error bounds to within 10%. All computations were carried out with a tolerance of $\texttt{TOL} = 10^{-8}$. Table 1 summarizes the run times of the two methods on problems with various values of n, m, b, and k. (Comparable average Eispack times from TRED2/TQL2 are 1.46, 0.26, and 0.05 seconds for $n = 64, 32, 16$, respectively.) Figures 1–4 display iteration counts and relative efficiencies of the two algorithms.

These experiments show that the extra work of finding the largest independent off-diagonal blocks is offset by faster algorithmic convergence of BCJ, which makes the present method competitive with other blocked Jacobi rotation techniques. While, the QR algorithm is obviously superior on these random, dense matrices, its advantage will be reduced on nearly diagonal or quite sparse problems.

		BCJ execution times			Blocked Brent-Luk times		
		BCJ	BCJ	BCJ	B-L	B-L	B-L
N	B	MIN	AVG	MAX	MIN	AVG	MAX
64	2	26.04	26.93	28.68	44.96	79.39	136.12
64	4	22.11	25.48	29.98	30.02	63.13	146.09
64	8	31.48	39.97	66.47	38.19	145.70	424.32
64	16	58.36	107.90	253.24	97.38	242.68	513.49
32	2	3.60	4.09	4.71	3.70	8.04	15.39
32	4	3.71	5.20	7.66	5.34	9.14	19.00
32	8	5.91	13.50	23.14	9.41	16.55	31.36
16	2	0.51	0.72	0.83	0.45	2.47	14.23
16	4	0.52	1.40	7.11	0.62	1.04	2.36

Table 1: Multiflow Trace/7 execution times (sec.) for BCJ, blocked Brent-Luk, $\texttt{TOL} = 10^{-8}$, 10 trials.

5 Algorithmic Analysis

An important factor in determining the efficiency of the algorithm is the block size. BCJ has the following computational work per iteration ($n = mb$):

Step	Computations
1	$2n^2$
2	$O(m^2 \log m)$
3	$6K(2b)(2b-1)/2$
4	$18nK(2b)(2b-1)/2$
5	$2n^2$
6	$2n^2 b + O(nb^2)$

The work for step 1 is actually completed in step 5, where the block masses are computed, so that after the first iteration step 1 contributes no work. Step 2 can be done in $O(Km^2)$ operations, which is an improvement if $K = o(\log m)$. Step 6 is performed once at the end of the calculation and has asymptotically negligible work if $b = o(n)$; for very large b step 6 dominates the total work.

A moderate value of b should be preferable in order to maximize the sum of off-diagonal block masses. Indeed, Figure 1 reflects this

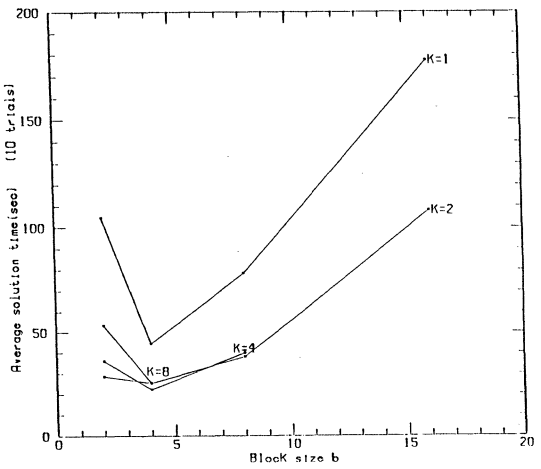


Figure 1. Average BCJ solution time for N=64, 10L = 100-8 on a Multiflow Trace/7 computer

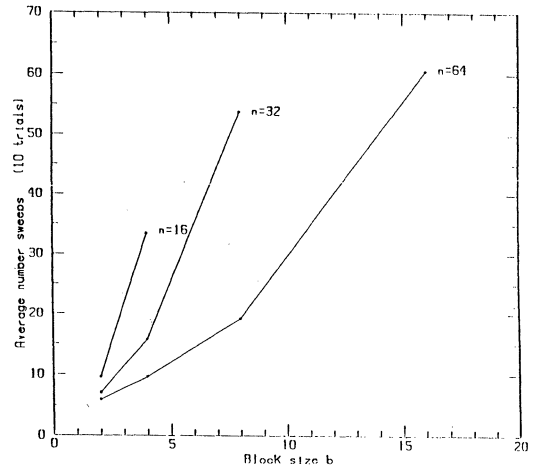


Figure 2. Average BCJ sweeps, 10L = 100-8, K=n/2b

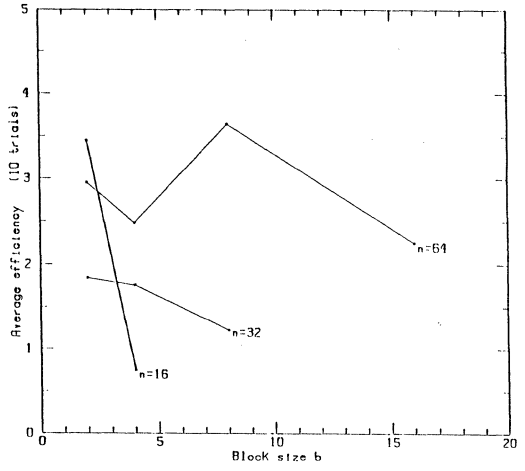


Figure 3. (Avg. Time Block Brent-Luk)/(Avg. Time BCJ) 10L = 100-8, K=n/2b

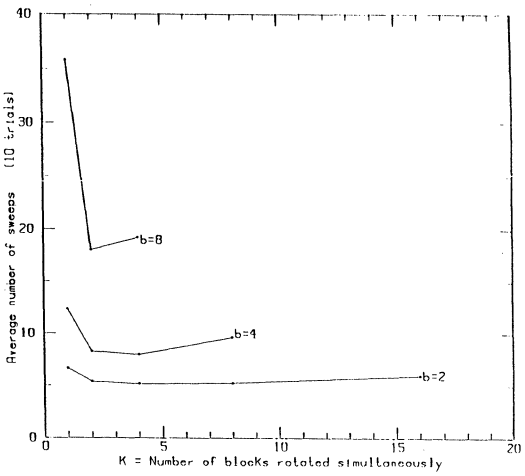


Figure 4. Average number of BCJ sweeps, n = 64, 10L = 100-8

behavior. For small b, the overhead of determining the largest block exceeds the work of diagonalizing A. As b increases, the maximal off-diagonal squared block mass will approach the average block mass, reducing the effect of each block rotation, and consequently lengthening BCJ computations. Figure 3 shows that for several matrix sizes n, increasing the block size b increases monotonically the number of sweeps of BCJ, as expected. Furthermore, the example of Figure 5 indicates that with relatively few blocks, two large off-diagonal masses are likely to be dependent.

We now examine BCJ's behavior with a brief review of relevant order statistics theory [6,11], which describes the behavior of sorted random variates in terms of the probability distributions of the individual elements. Given independent and identically distributed random variables X_1, X_2, ..., X_N, the N order statistics $X^*_{1,N}$, $X^*_{2,N}$, ..., $X^*_{N,N}$ are the random variables associated with the lowest ranked to highest ranked X_i.

A particular instance of the theory is instructive with regard to BCJ, which starts with the $(n^2-n)/2$-sized upper triangular array from an $n \times n$ symmetric matrix $A^{(0)}$ of uniform random variates. Let Y_i be one of $(n^2-n)/2$ *iid* uniform variates on the interval $(0, 1]$, and set $X_i = Y_i^2$. Then an average off-diagonal element of $A^{(0)}$ has squared mass $E[X_i] = 1/3$, while the maximum has mean squared mass $E[X^*_{(n^2-n)/2,(n^2-n)/2}] = 1 - 2/(n^2 - n + 2)$. Selecting the maximum off-diagonal element, rather than an average element, increases the reduction to diagonal form of an individual rotation.

Assuming further that each X_i, $1 \le i \le (m^2 - m)/2$, is distributed as the sum of b^2 squares of uniform random values from the interval (0, 1], as is M_{ij} in the first step of our blocked experiments, it is clear that the central limit theorem applies to the block mass distributions. For large b, one may represent the off-diagonal squared block mass as a normal random variable with mean $\mu = b^2/3$ and variance $\sigma^2 = 4b^2/45$ (corresponding to the sum of b^2 uniform random variables).

The maximal order statistic for these large blocks tends toward a standard limiting distribution, from which we may determine the moments. Although the example employs sums of uniform variates, the proposition holds for any blocks that have asymptotically normal squared mass.

Proposition 1. *Let $\{X_i\}_{i=1}^{(m^2-m)/2}$ be iid normal variates with mean μ and variance σ^2. In the limit as $m \to \infty$, the expected largest variate is*

$$E[X^*_{(m^2-m)/2,(m^2-m)/2}] = \mu + \sigma\sqrt{2\log((m^2-m)/2)} \qquad (6)$$

$$+ O\left(\log\log m/\sqrt{\log m}\right)$$

and the variance is

$$Var[X^*_{(m^2-m)/2,(m^2-m)/2}] = \sigma^2 \frac{\Gamma''(1) - \Gamma'(1)^2}{2\log((m^2-m)/2)} \qquad (7)$$

$$+ O\left(\log\log m/(\log m)^2\right),$$

where $\Gamma'(\cdot)$ and $\Gamma''(\cdot)$ are the first two derivatives of the gamma function, respectively. (Note that $\Gamma''(1) - \Gamma'(1)^2 \approx 1.64$.)

Thus the expected largest squared mass is about $2\sqrt{\log m}$ standard deviations above the mean, with asymptoticlly vanishing variance. Cohen [5] has derived similar results for generalized matrix products. The proof of Proposition 1 is left to section 7.

BCJ operates by maximizing the mass of the selected off-diagonal blocks. This works well when the ratio $(\mu + 2\sigma\sqrt{\log m})/\mu$ is large while the additional cost to determine the largest block is low. Both cost and benefit decrease with increasing blocksize.

For certain values of b, BCJ inherits the fast convergence of the classical Jacobi method without paying a large cost for maximal selection. If b is chosen approximately $b = (\log n)^{1/3}$ and $K = m/2$, the work of computing and selecting the independent maximal blocks is $O\left(n^2(\log n)^{1/3}\right)$ per iteration, as is the rotation work, so that the two are of comparable sizes. For larger block sizes b, the block selection cost is asymptotically negligible. If b grows as $\sqrt{\log n}$, then the largest squared block mass is a constant multiple of the average squared block mass, while the extra cost of determining the maximal off-diagonal blocks is of smaller order.

Figure 1 clearly indicates the benefits of choosing a moderate blocksize, as the average solution time initially decreases as b grows. However, the use of a large b produces longer runtimes.

The selection of $K > 1$ maximal independent off-diagonal blocks (step 2) forms a more complex sum of conditional order statistics, which we now examine. Let X_i, $1 \le i \le M_1$, be *iid* random variates with density $f(x)$ and distribution $F(x)$. Denote by $X^{*1}_{M_1}$ the maximal order statistic. Now fix a particular subset of size M_2 of the remaining variates (excluding the selected maximum and others), and let $X^{*2}_{M_1,M_2}$ be the maximal variate in the subset. It is clear that $X^{*2}_{M_1,M_2} \le X^{*1}_{M_1}$. Inductively define $X^{*k+1}_{M_1,\ldots,M_{k+1}}$ from $X^{*k}_{M_1,\ldots,M_k}$ as the maximal order statistic in a chosen subset of M_{k+1} variates selected from the previous subset of size M_k (excluding the previous maximum and others). We call $X^{*k}_{M_1,\ldots,M_k}$ the k^{th} *conditional maximal order statistic* of the $\{X_i\}_{i=1}^{M_1}$.

Proposition 2. *For $M_1 > M_2 > \cdots > M_k > 0$, the probability distribution of the k^{th} conditional maximal order statistic is*

$$\Pr\left\{X^{*k}_{M_1,\ldots,M_k} \le x\right\} = \sum_{i=1}^{k} F(x)^{M_i} \prod_{\substack{j=1 \\ j\neq i}}^{k} \left(\frac{M_j}{M_j - M_i}\right). \quad (8)$$

Letting $\mu_{M_i} = E[X^*_{M_i}]$ be the unconditional mean of the maximum on M_i observations, we have

$$E[X^{*K}_{M_1,\ldots,M_K}] = \sum_{i=1}^{K} \mu_{M_i} \prod_{\substack{j=1 \\ j\neq i}}^{K} \left(\frac{M_j}{M_j - M_i}\right). \quad (9)$$

We briefly indicate the formulation of the first step of BCJ in terms of Proposition 2. In BCJ, K maximal off-diagonal blocks are selected in K stages from an $m \times m$ upper triangular array of $(m^2 - m)/2$ *iid* random variates. Independence of the selected locations requires striking out the row and column of the maximum. At stage i, $1 \le i \le K$, the maximal variate will be drawn from a subset of $\binom{m+2-2i}{2}$ blocks in the strict upper triangle of the array and then two rows and columns of the array will be struck out, corresponding to the row and column indices of the selected maximal element.

For instance, in the 6 by 6 example of Figure 5, the first maximum is 10 (row 1, column 4). Thereafter rows and columns 1 and 4 are struck from the array (to preserve independence) and the second conditional maximum is selected; it is 5 (row 2, column 3). Note that larger elements that are dependent upon the first maximum may be ignored in the selection of the second maximum. Finally, rows and columns 2 and 3 are struck from the array and the final maximum of 4 (row 5, column 6) is selected.

The selection of the K maximal independent off-diagonal blocks (which forms the more complex sum of conditional order statistics discussed above), determines on average a smaller sum of off-diagonal masses than K successive iterations choosing the single largest block. However, it is observed in Figure 4 that the number

	2	4	10	6	7
		5	9	2	3
			3	4	4
				8	4
					4

Figure 5: Conditional maximum selection $X_{(1,4)} = X^{*1}_{15} = 10$, $X_{(2,3)} = X^{*2}_{15,6} = 5, X_{(5,6)} = X^{*3}_{15,6,1} = 4$.

of sweeps to convergence initially *declines* as K increases. This probably reflects the amortization of step 5 costs over additional blocks. As expected, BCJ requires slightly more iterations to converge as K reaches its upper limit of $m/2$ (*e.g.*, $b = 2, 4$).

Figure 2 presents in graphical form the ratios of the average BCJ and blocked Brent-Luk execution times on a Multiflow Trace/7 computer. The efficiency ratio shows the speedup of BCJ, with improvements up to a factor of 3.6 due entirely to improved index selection. Examination of Table 1 shows that, for almost all cases, BCJ runtimes have lower deviations from the mean.

Asymptotically, $b = \Omega\left((\log n)^{1/3}\right)$ guarantees that the work of selecting the maximal blocks will be at most comparable to the other arithmetic operations. However, assuming normality of initial data and intermediate results, the optimal b so that largest blocks are substantially larger than average ($\hat{b}^2 = O(\hat{b}\sqrt{\log m^2})$, from eq. (6)) is $b = O(\sqrt{\log n})$. For $b \approx (\log n)^\alpha$, $1/3 \le \alpha \le 1/2$, BCJ should be asymptotically faster than a blocked Brent-Luk method. The numerical experiments show speedups for problems of moderate size.

In general, the distribution of the elements of $A^{(\nu)}$ will be more complex than described here and the order statistic argument must be specialized to include the distributions of intermediate results, in order to rigorously prove rates of convergence. However, the improved performance of the new algorithm is consonant with the analysis performed here.

In cases where the matrix has few large elements or is close to diagonal, one expects BCJ to acheive shorter runtimes than indicated by these experiments on uniform random data. For instance, the method may prove useful in adaptive signal processing algorithms that rely on eigenvalue decompositions [14].

6 Conclusions

The improved index selection process of BCJ produces a substantial overall reduction in the program running time, compared to a blocked Brent-Luk algorithm. In particular, the extra work of determining the largest off-diagonal blocks is offset by fewer iterations needed for convergence. Furthermore, because the algorithm employs blocked data concepts, it is appropriate for computers with a hierarchical memory system. The concentration of work on the relatively small and numerous block elements is advantageous for parallelization of the algorithm.

The selection of parameters b and K is important to the efficiency of BCJ. A moderate value of b gives the lowest run times (though not the lowest number of block sweeps). The extra benefit of increasing K falls off rapidly for small n.

Nearly all stages of the algorithm are amenable to efficient parallel computation. Step 1 can be computed independently on m^2 processors; step 2 on various combinations of processors and interconnections; step 3 on K large-grained or Kb fine-grained processors, depending on whether the block rotation is parallelized or not; step 4 on up to bKn processors; step 5 on m^2 or more processors; and step 6 on K or more processors.

This investigation of BCJ was prompted by the use of a blocked Brent-Luk method in the Saxpy Computer Corp. mathematical subroutine library. It appears that a BCJ method could be more efficient than the current approach. Although Eispack routines are obviously quite fast for the dense examples used here, BCJ may improve upon the QR algorithm in cases of sufficiently sparse symmetric eigenvalue problems.

7 Calculation of Distributions of the Maximal and k^{th} Conditional Maximal Order Statistics

Proof of Proposition 1. David [6] presents the limiting distributional behavior of the maximal order statistic $X^*_{n,n}$, which depends upon the well-known distribution

$$\Lambda(x) = \exp(-e^{-x}) \qquad -\infty < x < \infty \tag{10}$$

in the case of *iid* 0-1 normal variates. We now carry through the analysis for general *iid* normal variates.

Let $\phi_{\mu\sigma^2}(x) = (\sigma\sqrt{2\pi})^{-1}\exp(-(x-\mu)^2/2\sigma^2)$ be the normal density function and let $\Phi_{\mu\sigma^2}(x) = \int_{-\infty}^{x}\phi_{\mu\sigma^2}(y)dy$ be the normal distribution function corresponding to mean μ and variance σ^2, respectively. For large x,

$$\frac{1-\Phi_{\mu\sigma^2}(x)}{\phi_{\mu\sigma^2}(x)} \approx \frac{\sigma^2}{x-\mu}, \tag{11}$$

based on a change of variables from the case $\mu = 0$ and $\sigma = 1$. Thus Theorem 9.3.5 [6] applies and

$$\lim_{n\to\infty}\Pr\left\{(X^*_{n,n} - l_n)n\phi_{\mu\sigma^2}(l_n) \le x\right\} = \Lambda(x) \tag{12}$$

holds uniformly for every $x \in (-\infty,\infty)$, where l_n is selected so that $\Phi_{\mu\sigma^2}(l_n) = 1 - 1/n$.

According to (11),

$$\frac{1}{n} = 1 - \Phi_{\mu\sigma^2}(l_n) \approx \frac{\sigma^2}{l_n-\mu}\frac{1}{\sigma\sqrt{2\pi}}\exp\left(\frac{-(l_n-\mu)^2}{2\sigma^2}\right). \tag{13}$$

The asymptotic form of l_n is then

$$l_n = \mu + \sigma\left(\sqrt{2\log n} - \frac{(\log\log n + \log 4\pi)}{\sqrt{8\log n}}\right) + O(1/\log n). \tag{14}$$

Using the relation $n\phi_{\mu\sigma^2}(l_n) = (l_n-\mu)/\sigma^2 + O(l_n^{-1})$, we see that

$$\lim_{n\to\infty}\Pr\left\{(X^*_n - l_n)(l_n-\mu)/\sigma^2 \le x\right\} = \Lambda(x). \tag{15}$$

It follows directly from (15) that

$$E[X^*_n] = l_n + \frac{\Gamma'(1)\sigma^2}{l_n-\mu} + o(l_n^{-1}) \tag{16}$$

$$= \mu + \sigma\sqrt{2\log n} + O\left(\log\log n/\sqrt{\log n}\right) \tag{17}$$

where $\Gamma'(1)$ (Euler's constant) is the mean associated with $\Lambda(x)$. The variance vanishes asymptotically according to

$$Var[X^*_n] = \sigma^4\frac{\Gamma''(1)-\Gamma'(1)^2}{(l_n-\mu)^2} \tag{18}$$

$$= \sigma^2\frac{\Gamma''(1)-\Gamma'(1)^2}{2\log n} + O\left(\log\log n/(\log n)^2\right), \tag{19}$$

where $\Gamma''(1) - \Gamma'(1)^2$ is the variance associated with $\Lambda(x)$. (Here we have used $\Gamma'(\cdot)$ and $\Gamma''(\cdot)$ to represent the first two derivatives of the gamma function, respectively.) □

Proof of Proposition 2. Let $\left\{X^*_{M_k,M_k} = x \,|\, X_i \le y\right\}$ denote the event that the maximal order statistic on M_k observations is x, conditioned on all observations X_i in the subset of size M_k being bounded above by y. Then the density of the k^{th} conditional maximal order statistic obeys the relation

$$\Pr\left\{X^{*k}_{M_1,\ldots,M_k} = x\right\} \tag{20}$$

$$= \int_x^\infty \Pr\left\{X^{*k-1}_{M_1,\ldots,M_{k-1}} = y\right\}\Pr\left\{X^*_{M_k,M_k} = x \,|\, X_i \le y\right\}dy,$$

where

$$\Pr\left\{X^*_{M_k,M_k} = x \,|\, X_i \le y\right\} = \begin{cases}\frac{d}{dx}\left(\frac{F(x)}{F(y)}\right)^{M_k}, & -\infty < x \le y < \infty\\ 0, & -\infty < y < x < \infty\end{cases} \tag{21}$$

is the probability distribution of the maximal order statistic on M_k bounded observations.

Define $V_k(x) = \Pr\left\{X^{*k}_{M_1,\ldots,M_k} \le x\right\}$. Then $V_1(x) = F(x)^{M_1}$. Inductively assuming that $V_k(x) = \sum_{i=1}^k a_{i,k}F(x)^{M_i}$ gives a recurrence relation on the a_{ik} of

$$a_{ik} = a_{i\,k-1}\frac{M_k}{M_k - M_i}, \qquad 1 \le i < k, \tag{22}$$

and

$$a_{kk} = \sum_{i=1}^{k} a_{i\,k-1}\frac{M_i}{M_i - M_k}, \tag{23}$$

where $a_{11} = 1$. Consideration of the $k-1$ degree Lagrangian polynomial interpolating the points $(M_i, 1)$, $1 \le i \le k$, establishes that

$$a_{ik} = \prod_{\substack{j=1\\ j\ne i}}^{k}\left(\frac{M_j}{M_j - M_i}\right). \tag{24}$$

The distribution is thus

$$\Pr\left\{X^{*k}_{M_1,\ldots,M_k} \le x\right\} = \sum_{i=1}^{k} F(x)^{M_i}\prod_{\substack{j=1\\ j\ne i}}^{k}\left(\frac{M_j}{M_j - M_i}\right). \quad \square \tag{25}$$

References

[1] C. Bischof. *Computing the Singular Value Decomposition on a Distributed System of Vector Processors.* Technical Report TR–87–869, Department of Computer Science, Cornell University, Ithaca, New York, September 1987.

[2] C. Bischof and C. Van Loan. Computing the singular value decomposition on a ring of array processors. In J. Cullum and R. Willoughby, editors, *Large Scale Eigenvalue Problems*, pages 51–66, Elsevier, 1986.

[3] K. A. Braun and Th. Lunde Johnsen. Hypermatrix generalization of the Jacobi and Eberlein method for computing eigenvalues and eigenvectors of Hermitian or non-Hermitian matrices. *Computer Methods in Applied Mechanics and Engineering*, 4:1–18, 1974.

[4] R.P. Brent and F.T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM J. Sci. Stat. Comput.*, 6(1):69–84, 1985.

[5] J. E. Cohen. Subadditivity, generalized products of random matrices and operations research. *SIAM Review*, 30(1):69–86, 1988.

[6] H.A. David. *Order Statistics.* J. Wiley and Sons, 2nd edition, 1981.

[7] P.J. Eberlein. On the diagonalization of complex symmetric matrices. *J. Inst. Math. Applic.*, 7:377–383, 1971.

[8] D.E. Foulser and R. Schreiber. The Saxpy Matrix-1: A general-purpose systolic computer. *IEEE Computer*, 20(7):35–43, 1987.

[9] G.H. Golub and C.F. Van Loan. *Matrix Computations.* Johns Hopkins Press, 1983.

[10] C.G.J. Jacobi. Uber ein leichtes verfahren die in der theorie der sacularstorungen vorkommendern gleichungen numerisch aufzulosen. *Crelle's Journal*, 30:51–94, 1846.

[11] S. Karlin and H. Taylor. *A Second Course in Stochastic Processes.* Academic Press, 1981.

[12] D.J. Kuck and A.H. Sameh. Parallel computation of eigenvalues of real matrices. In *Information Processing 1971*, pages 1266–1272, North-Holland, 1972.

[13] A.H. Sameh. On Jacobi and Jacobi-like algorithms for a parallel computer. *Math. Comp.*, 25:579–590, 1971.

[14] R.O. Schmidt. *A Signal Subspace Approach to Multiple Emitter Location and Spectral Estimation.* PhD thesis, Stanford University, November 1981.

[15] R. Schreiber and B. Parlett. Block reflector computation and applications. In R. Glowinski and J.L. Lions, editors, *Computing Methods in Applied Sciences and Engineering*, North Holland, 1986.

[16] C. Van Loan. The block Jacobi method for computing the singular value decomposition. In C. Byrnes and A. Lindquist, editors, *Computational and combinatorial methods in systems theory*, pages 245–256, North-Holland, 1986.

An Optimal Parallel Jacobi–Like Solution Method for the Singular Value Decomposition

G. R. Gao and S. J. Thomas *

January, 1988

Abstract

A new parallel Jacobi-like solution method for the singular value decomposition (SVD) is presented which is optimal in achieving both the maximum concurrency in computation and the minimum overhead in communication. Unlike previously published parallel SVD algorithms based on a nearest neighbour ring topology for communication, the new algorithm introduces a recursive divide–exchange communication pattern. As a result of the recursive nature of the algorithm, proofs are given to show that it achieves the lower bounds both in computation and communication costs. In general, the recursive pairwise exchange communication operations of the new algorithm can be efficiently supported by multiprocessors with interconnect patterns used in many networks that have been proposed to support large–scale parallelism. As an example, this paper illustrates that the new algorithm can be mapped efficiently and naturally onto hypercube architectures. Preliminary results with an implementation of the new algorithm are reported. Convergence aspects of the new algorithm are briefly discussed. A comparison with related work is outlined.

1 Introduction

Rapid technological advances in multiprocessor architectures have aroused much interest in parallel computation. Parallel methods to compute the singular value decomposition (SVD) have received attention due to its many important applications in science and engineering. A recent paper by Heath *et al* [8] includes a history of various Jacobi–like SVD algorithms.

An early investigation into parallel computation for the symmetric eigenvalue problem, on the SIMD Illiac IV is described by Sameh in [18]. Sameh outlines the criteria for maximal parallelism in a Jacobi–like algorithm. More recently, a number of authors including Berry *et al* [1] advocate the one–sided SVD of Hestenes [9], [8], [15] for parallel computation of the SVD. Luk and his co–workers have examined various systolic array configurations to compute the SVD [12], [3], [4]. Brent and Luk [4] have invented a linear array of $n/2$ processors which implements a one–sided Hestenes algorithm, that in real arithmetic, is an exact analogue of their Jacobi method applied to the eigenvalue problem. The array requires $O(mnS)$ time, where S is the number of sweeps (typically ≤ 10). Brent and Luk demonstrate that their algorithm is computationally optimal in the sense that it requires the minimum number of computational steps per sweep i.e. $n-1$, to ensure the execution of every possible pairwise column rotation. Maximum concurrency is maintained throughout the computation. Their systolic array is comparable to the architecture of a nearest–neighbour linear array of processors, where communication is based on a ring topology.

Brent and Luk's algorithm is not optimal in terms of communication overhead. Unnecessary costs are incurred by mapping the systolic array architecture onto a ring connected linear array due to the double sends and receives required between pairs of neighbouring processors. Eberlein [5], Bischof [2] and others have proposed various modifications for hypercube implementations, which require the embedding of rings via binary reflected *Gray codes*.

In this paper, we present a new parallel Jacobi–like solution method for the SVD which is optimal in achieving both the maximum concurrency in computation and the minimum overhead in communication. Unlike previously published parallel SVD algorithms based on a nearest neighbour ring topology for communication, the new algorithm proposed in this paper introduces a recursive divide–exchange communication pattern. As a result of the recursive nature of the algorithm, proofs are given to show that it achieves the lower bounds both in computation and communication costs. Convergence aspects of the new algorithm are briefly discussed. The paper illustrates that the new algorithm can be mapped efficiently and naturally onto hypercube architectures. We have implemented the new algorithm on the Intel hypercube through simulation and the preliminary results will be discussed. A comparison with related work is briefly outlined. We believe that the new algorithm can be efficiently mapped onto multiprocessors with interconnection patterns that have been proposed to support large–scale parallelism such as the many PM2I–based or cube–based networks [20].

2 Jacobi–like Algorithms

2.1 The Singular Value Decomposition

The singular value decomposition (SVD) of a general non–square matrix may be given as follows,

Theorem 2.1 *For a real matrix $A(m \times n)$ of rank r, there exists orthogonal matrices $U(m \times m)$ and $V(n \times n)$, such that*

$$U^T A V = \Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \cdots) \geq 0,$$

where the elements of $\Sigma(m \times n)$ may be ordered so that

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_q = 0, \quad q = \min\{m, n\}.$$

If $m = n$, Σ is a square diagonal $n \times n$ matrix [11].

In order to compute the SVD in an iterative fashion, a series of plane rotations may be applied to the matrix $A(m \times n)$ described in theorem 2.1 above. This approach is similar in nature to Jacobi's original method for computing the eigenvalues of a symmetric matrix where orthogonal matrices $J(i, j, \theta)$ are applied so as to annihilate a symmetrically placed pair of the $n(n-1)$ off–diagonal elements. These rotation matrices differ from the identity matrix of order n by the principal submatrix formed at the intersection of the row and column pairs corresponding to i and j. A 2×2 submatrix has the form

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

The cosine and sine of the rotation angle θ are the constants $c = \cos\theta$ and $s = \sin\theta$. Initially $A_1 = A$ and at the k–th iteration,

*School of Computer Science, McGill University, Montreal, Quebec, Canada, H3A 2K6. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada under Grant A9236.

$$A_{k+1} = J(i_k, j_k, \theta_k)^T A_k J(i_k, j_k, \theta_k).$$

Rotations are applied simultaneously, in a symmetric fashion from the left and right. Cyclic Jacobi methods refer to a sequence of rotations which update row and column pairs in some predetermined order. For a square matrix, a cyclic *sweep* refers to the updating of $n(n-1)/2$ elements. A number of sweeps are required in order to effectively reduce the off–diagonal mass of the matrix to a sufficiently small value, which eventually can be ignored. A diagonal containing the eigenvalues then remains.

Annihilation of 2 off-diagonal elements of a symmetric matrix takes the form,

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} \alpha_{ii}^{(k)} & \alpha_{ij}^{(k)} \\ \alpha_{ij}^{(k)} & \alpha_{jj}^{(k)} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} \alpha_{ii}^{(k+1)} & 0 \\ 0 & \alpha_{jj}^{(k+1)} \end{bmatrix}$$

Kogbetliantz appears to have been the first to apply this method to general nonsymmetric matrices [10] (see [8] and [7]). We can generalize the above equation to the computation of a 2×2 SVD, by using two different orthogonal rotation matrices [8]. A serial–cyclic sweep of a general $m \times n$ matrix A can be performed either by a *cyclic–by–row* or a *cyclic–by–column* scheme.

As noted by Brent *et al* [4] and others, serial cyclic–by–row and cyclic–by–column schemes are not suitable for parallel computation due to column and row conflicts throughout. In §2.2 we shall indicate that orderings suitable for parallel computation would apply $\lfloor n/2 \rfloor$ rotations simultaneously. In terms of convergence for algorithms which compute the SVD in a cyclic manner we may appeal to the results of Paige and Van Dooren [16].

2.2 Exploiting Parallelism

Sameh was one of the first researchers to observe that there is a bound on the number of rotations which may be applied in parallel [18], [1], [19]. Given a general $m \times n$ matrix, a Kogbetliantz cyclic sweep consists of a maximum of

$$N = \frac{\max\{m,n\}(\max\{m,n\} - 1)}{2}$$

pairs of rotations. Our goal is to complete a sweep in the minimum number of parallel steps each consisting of the maximum number of rotations applied in parallel. In addition the maximum number of processors should be kept busy at all times.

Criteria such as these were originally formulated by Sameh [18]. For square matricecs with $n(n-1)/2$ elements above the main diagonal, it is possible to update or annihilate $\lfloor n/2 \rfloor$ elements at a time. Defining $r = \lfloor (n+1)/2 \rfloor$, we can have $(2r-1)$ rotation sets applied per sweep. To summarize,

1. An orthogonal *rotation set* must annihilate or update $\lfloor n/2 \rfloor$ elements.
2. A sweep should annihilate each off–diagonal element only once. This implies each of $(2r-1)$ orthogonal rotation sets should annihilate or update $\lfloor n/2 \rfloor$ elements.

The size of a rotation set is simply determined by the maximum number of non–conflicting column pairings possible. For example, given an $n \times n$ square matrix with $n = 4$ we may simultaneously apply 2 rotations from the left or right. This is equivalent to multiplication by an orthogonal matrix V of the form,

$$V = \begin{bmatrix} c_1 & & s_1 & \\ & c_2 & & s_2 \\ -s_1 & & c_1 & \\ & -s_2 & & c_2 \end{bmatrix}$$

The number of parallel iterations in a computation is therefore bounded below by

$$\frac{n(n-1)}{2} \times \frac{1}{\lfloor n/2 \rfloor} \tag{2.1}$$

or equivalently,

$$2r - 1 = \begin{cases} n & n \text{ odd} \\ n-1 & n \text{ even} \end{cases}$$

Proposition 2.2 *For n a positive integer, if $r = \lfloor (n+1)/2 \rfloor$ then*

$$\begin{aligned} \frac{n(n-1)}{2} \times \frac{1}{\lfloor n/2 \rfloor} &= 2r - 1 \\ &= \begin{cases} n & n \text{ odd} \\ n-1 & n \text{ even} \end{cases} \end{aligned} \tag{2.2}$$

Proof. Consider two cases,

Case 1. When n is even, $\lfloor n/2 \rfloor = n/2$ so that,

$$\frac{n(n-1)}{2} \times \frac{1}{\lfloor n/2 \rfloor} = n - 1.$$

Furthermore since n is even, $n+1$ is odd, hence $\lfloor (n+1)/2 \rfloor = \lfloor n/2 \rfloor = n/2$ and

$$2r - 1 = 2\left(\frac{n}{2}\right) - 1 = n - 1.$$

Case 2. When n is odd, $\lfloor n/2 \rfloor = \lfloor (n-1)/2 \rfloor = (n-1)/2$ and

$$\frac{n(n-1)}{2} \times \frac{1}{\lfloor n/2 \rfloor} = n.$$

With n odd, $n+1$ is even, so that $\lfloor (n+1)/2 \rfloor = (n+1)/2$ and

$$2r - 1 = 2\left(\frac{n+1}{2}\right) - 1 = n.$$

If we assume that n is even, then not all algorithms described in the literature have achieved the $n-1$ lower bound. Sameh's implementation of Hestenes' one–sided computation on a linear array of processors requires $3n-2$ parallel iterations per sweep [19], whereas Brent and Luk report that they achieve the minimum with their systolic array [4].

2.3 A One–sided Computation

When we consider general non–square $m \times n$ matrices where $m \geq n$ there exists a convenient computation for the SVD which is appropriate for parallel implementation. This method is based on a one–sided computation originally due to Hestenes [9]. It is referred to as one–sided because orthogonal rotations are only applied from the right, updating columns. Brent and Luk's [4] systolic array implements Hestenes' algorithm. Basic operations in each processor of their array reflect a tournament ordering scheme for rotations performed in parallel. The performance of their scheme is analyzed in §3. Eberlein [5] has proposed a block variant of Hestenes' algorithm on a hypercube, suitable for computing either singular values or eigenvalues of symmetric matrices.

Hestenes' one–sided computation produces an orthogonal matrix V and a matrix Q with orthogonal columns such that

$$AV = Q = [q_1, q_2, \cdots, q_n]. \tag{2.3}$$

where A is $m \times n$, $m \geq n$. The Euclidean norms of the columns will be equated with the singular values of A.

$$q_i^T q_j = \sigma_i^2 \delta_{ij},\ i, j = 1, \ldots, n.$$

By normalizing the columns, we see that the SVD of theorem 2.1 is implicit in (2.3)

$$Q = U\Sigma, \quad A = U\Sigma V^T$$

A one–sided algorithm is somewhat different from its earlier counterparts, as rotations are applied from the right and therefore only columns are affected. Off–diagonal elements are no longer annihilated, instead rotations are designed in order to produce two orthogonal columns. As with similar Jacobi–like algorithms, the orthogonal matrix V may be accumulated from plane rotations $J(i,j,\theta)$ which differ from the unit matrix I_n in a 2×2 principal submatrix containing the cosines and sines of the rotation. Setting $A_1 = A$, the k–th iteration updates A_k

$$A_{k+1} = A_k J(i,j,\theta_k).$$

If the matrix sequence A_k converges, the result is Q in (2.3). A column update via a 2×2 submatrix takes the form

$$\begin{bmatrix} a_i^{(k)}, & a_j^{(k)} \end{bmatrix} \begin{bmatrix} c & -s \\ s & c \end{bmatrix} = \begin{bmatrix} a_i^{(k+1)}, & a_j^{(k+1)} \end{bmatrix},$$

The orthogonality condition determines the rotation angle θ.

$$\frac{2(a_i^{(k)})^T a_j^{(k)}}{(a_i^{(k)})^T a_i^{(k)} - (a_j^{(k)})^T a_j^{(k)}} = \tan 2\theta. \tag{2.4}$$

By avoiding a potential loss of significant digits the magnitude of the angle may be restricted to $|\theta| \leq \pi/4$ and provides formulae for the rotation (see Nash [15] and Rustishauser [17]).

As noted by Brent and Luk [4], if a cyclic–by–row rotation ordering is chosen to update the $n(n-1)/2$ column pairings determined by the off–diagonal elements above the main diagonal, convergence would follow. Hestenes' computation is mathematically equivalent to a Jacobi algorithm applied to A^TA, therefore we expect that the convergence analyses of Forsythe and Henrici [6] or Wilkinson [22] are applicable under these circumstances. Rather than testing for convergence, the threshold Jacobi method originally introduced in the symmetric eigenproblem is often employed [23, pp. 277–278], [17].

3 Parallel Computation

3.1 Maximizing Concurrency

In this paper the computation cost is measured by the number of parallel computation steps. The methods discussed process (i,j) pairings consisting of partitions containing at least 1 column or row. When n is even, if we assume one parallel computation step has unit cost, then of the algorithms presented the minimum cost achieved is $n-1$ per sweep. The systolic array and associated algorithm proposed by Brent and Luk were proven to achieve this lower bound in [4]. We have illustrated their basic scheme in figure 1 for the case $n = 8$, where a linear array of four processors $\{P_1, P_2, P_3, P_4\}$ is used.

3.2 Minimizing Communication Costs

Another important performance criteria for a parallel algorithm is the total communication cost. For our purposes the communication cost can be measured by the total number of interprocessor transactions (messages). A transaction consists of a column transmission between a pair of processors. The total communication cost of one sweep will be denoted C.

From the last section, we know that the minimum number of computation steps in a sweep is $K = n - 1$. The minimum number of interprocessor transactions is achieved when each processor retains one column from a pairing, and transmits the other to a destination processor. As a result, if there are p processors, p transmissions are performed between two consecutive steps. Hence the minimum total communication cost C_{min} is

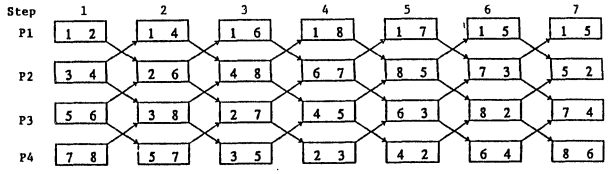


Figure 1: Brent and Luk's Systolic Array

defined by the following.

$$C_{min} = (K-1)p. \tag{3.1}$$

In the parallel one-sided SVD algorithm each processor is assigned one of $n/2$ column pairs at each step, assuming n is even. The total number of processors required is $p = n/2$ in (3.1) and the communication costs are $O(n^2)$.

$$\begin{aligned} C_{min} &= (K-1)p \\ &= \frac{n(n-2)}{2}. \end{aligned}$$

As a contrast, a global broadcasting strategy may request each processor to send both columns to all other $p-1$ processors between each step. The total cost for this case will be $O(n^3)$. Brent and Luk's algorithm has the following communication cost.

$$\begin{aligned} C_{BL} &= (K-1) \times 2p \\ &= (n-2) \times 2\left(\frac{n}{2}\right) \\ &= n(n-2). \end{aligned}$$

Therefore their algorithm is close to, but not quite optimal. In fact the inefficiency lies in the double sends and receives between processors in the systolic array which are dictated by the tournament ordering.

Several ways of modifying Brent and Luk's algorithm to avoid the double sends and receives have been proposed [13], [14], [5], [2]. These algorithms all represent a communication regimen based on a ring topology. A ring topology resembles the architecture of a linear array of processors. Embedding a ring within another topology, for example the binary n–cube, requires a special mapping scheme.

4 An Optimal Parallel SVD Algorithm

In this section we present a new parallel Jacobi–like algorithm which is optimal in terms of both achieving maximum concurrency and minimum communication overhead. The algorithm relies on a recursive divide–exchange of $n = 2^d$ columns.

Unlike several orderings cited earlier, the new algorithm maps naturally onto parallel architectures which support recursive pairwise exchanges. A mapping onto a hypercube is presented as an example in §5. Pairwise exchanges of columns here are specified by a *Perfect Shuffle* of processor addresses [21].

4.1 The Parallel Algorithm

Let us first illustrate the basic principle of the new algorithm through an example where $n = 8$ and $p = 4$. The computation steps K_l and communication steps X_l consisting of pairwise exchanges, are shown in figure 2.

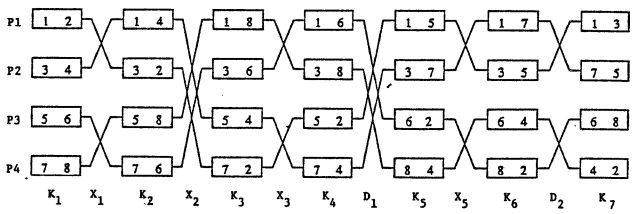


Figure 2: Recursive Divide–Exchange

Initially the 8 column indices are divided into two sets.

$$G_1 = \{1,3,5,7\}, \quad G_2 = \{2,4,6,8\}. \tag{4.1}$$

The pairs $\{(1,2), (3,4), (5,6), (7,8)\} \in G_1 \times G_2$ are assigned, in order, to processors in the set $\mathcal{P} = \{P_1, P_2, P_3, P_4\}$.

The algorithm for $n = 2^d = 2^3 = 8$ consists of three parts:

Part **1:** Compute–Exchange stage. The first stage consists of $n/2 = 4$ computation steps $\{K_1, K_2, K_3, K_4\}$ and $n/2 - 1 = 3$ communication (exchange) steps $\{X_1, X_2, X_3\}$. In one computation step, each processor performs a plane rotation on an (i,j) pairing. A communication step X_l exchanges columns with indices in G_2 between processor pairs.

Part **2:** Divide step. Processors are divided into two sets

$$\mathcal{P}_1 = \{P_1, P_2\}, \quad \mathcal{P}_2 = \{P_3, P_4\}.$$

The column indices in G_1 are divided into two subsets,

$$G_3 = \{1,3\}, \quad G_4 = \{5,7\},$$

and are assigned to $\mathcal{P}_1$. Similarly, G_2 is split into

$$G_5 = \{2,4\}, \quad G_6 = \{6,8\},$$

and assigned to $\mathcal{P}_2$, as indicated in figure 2 by step D_1.

Part **3:** Recursively solve the two subproblems using a scheme similar to parts 1 and 2. A subproblem consists of $n' = n/2 = 4$ column pairs and $n'/2 = 2$ processors.

In order to specify the *pairwise* exchange of columns between processors described in part 1 above we introduce the notion of *distance* between processors. Given an index set $S = \{1, 2, \ldots, N\}$ synonymous with processor addresses and a set of processors $\mathcal{P} = \{P_i : i \in S\}$ we have,

Definition 4.1 *The distance $s \in S$ between processors $P_{i_1} \in \mathcal{P}$ and $P_{i_2} \in \mathcal{P}$ is defined to be*

$$s = |i_1 - i_2|$$

The algorithm (for $n = 2^d$, $d = 3$) can be unwound into a sequence of $d = 3$ compute–exchange stages (with one divide step between each pair of successive compute–exchange stages) as shown in figure 2.

$$\{K_1, X_1, K_2, X_2, K_3, X_3, K_4, D_1, K_5, X_5, K_6, D_2, K_7\}.$$

Each exchange step X_l is a parallel *pairwise exchange* of column indices in G_2 between processor pairs (P_{i_1}, P_{i_2}), where P_{i_1} and P_{i_2} are at a distance 2^h, ($h \geq 0$, h an integer and $i_1 < i_2$). Furthermore, the binary representations of i_1 and i_2 may only differ in bit position h. For example, the three communication steps X_1, X_2 and X_3 result in the exchange pairings illustrated in figure 3.

X_l	(P_{i_1}, P_{i_2})	s
X_1	(P_1, P_2), (P_3, P_4)	2^0
X_2	(P_1, P_3), (P_2, P_4)	2^1
X_3	(P_1, P_2), (P_3, P_4)	2^0

Figure 3: Parallel Processor Pairings

In general, the algorithm (for $n = 2^d$) can be unwound into a sequence of d compute–exchange stages (with one divide step between each pair of successive compute–exchange stages). If we number the d compute–exchange stages by $k : k = 1, \ldots, d$, the k–th compute–exchange stage consists of $2^{d-k} = n/2^k$ computation steps K_l, $l = 1, \ldots, 2^{d-k}$ and $2^{d-k} - 1$ communication (exchange) steps X_l, $l = 1, \ldots, 2^{d-k} - 1$ forming

$$\{K_1, X_1, K_2, X_2, \ldots, X_{2^{d-k}-1}, K_{2^{d-k}}\}.$$

1. At each computation step K_l, processors concurrently compute rotations on their assigned column pairings.

2. At each communication step X_l a parallel pairwise exchange of columns with indices in G_2 is performed between processors pairs at a distance 2^h, where h is given by the function,

$$h = h(l) = \begin{cases} q & \text{if } l = 2^q, \\ h(l - 2^q) & \text{if } l > 2^q. \end{cases}$$

q is the largest integer which satisfies $2^q \leq l$.

4.2 Computation and Communication Costs

Let $n = 2^d$, and the total number of computation steps be $f(n)$. If $g(n)$ is the number of computation steps in stage 1 then a recurrence relation for $f(n)$ is

$$f(n) = g(n) + f(n/2)$$

From our description of the algorithm we have $g(n) = n/2$, hence

$$f(n) = \begin{cases} n/2 + f(n/2) & n > 2, \\ 1 & n = 2. \end{cases} \tag{4.2}$$

Solving the recurrence (4.2), we have $f(n) = n - 1$. Therefore, we have verified the fact that the new parallel algorithm has achieved the optimal computation cost. The reader should note that in solving the above recurrence, a geometric progression corresponding to the stage lengths results.

To establish that we have achieved the optimal communication cost consider a stage k consisting of 2^{d-k} computation steps and $2^{d-k} - 1$ communication steps. For $n - 1$ total computation steps, d stages are required. The inter–stage divide steps account for $d - 1$ of the total. The total number of communication steps $c(n)$ may be derived from a recurrence relation.

$$c(n) = \begin{cases} n/2 + c(n/2) & n > 2, \\ 0 & n = 2. \end{cases} \quad (4.3)$$

Solving (4.3), we obtain $c(n) = n - 2$. Multiplying by the number of processors $p = n/2$ gives the communication costs C_{DE} for our recursive divide–exchange algorithm. We have achieved the optimum since $C_{DE} = C_{min}$.

$$C_{DE} = (n-2)p = \frac{n(n-2)}{2}.$$

Referring to our example in figure 2, 4 column transactions have occurred at each communication step with a total cost of $6 \times 4 = 24$ transactions, which is optimal.

5 Mapping onto the Hypercube

In order to map the recursive divide–exchange algorithm of §4 onto a hypercube architecture we must first specify the operations performed by each processor in the cube. Given the two major components of our algorithm, namely a *compute–exchange* and a *divide*, deriving an algorithm for individual processors is straightforward. Due to the tail–recursion in the parallel SVD algorithm, it may be transformed into an iterative form.

Algorithm Divide–Exchange

for $k = 1$ **to** d **do**
  for $l = 1$ **to** $2^{d-k} - 1$ **do**
    Compute (i, j)
    $q = h(l)$
    Exchange 2^q
  end
  Compute (i, j)
  Divide 2^{d-k-1}
end

The step "Compute (i, j)" refers to a column update in the parallel version of Hestenes' one–sided computation. Using the terminology introduced in §4, each processor cycles through a Jacobi–sweep consisting of d stages. A divide step, exchanging at a distance of 2^{-1} would not be carried out. The function $h(l)$ computes the height of an exchange node X_l, where l is the label number derived by an inorder traversal of a complete binary tree.

Function $h(l)$

begin
  $q = \lfloor \log_2 l \rfloor$
  $t = l - 2^q$
  if $t = 0$ **then**
    return q
  else
    return $h(t)$
  end
end

The relative ease of mapping a recursive divide–exchange onto the hypercube is due to the recursive nature of the hypercube itself. The fact that a hypercube is recursively constructed out of lower dimensional subcubes may be exploited. A *divide* step in our algorithm corresponds to a subdivision of the problem, allowing computations to proceed on the subcubes. Exchanges will always consist of communication between pairs of nearest neighbours on the hypercube. A cube of dimension $d - 1$ is required for a problem with $n = 2^d$. The computation and communication steps are determined by the exchange sequence shown in figure 3.

5.2 Processor Pairings

Nearest neighbour processor pairings on the hypercube may be determined by a *Perfect Shuffle* of node addresses. Stone's original paper [21] details the generation of such pairings via a left cyclic shift of the bits in an address. A perfect shuffle of an N element vector is a permutation P of the indices or addresses a of the elements such that

$$P(a) = \begin{cases} 2a & 0 \le a \le N/2 - 1, \\ 2a + 1 - N & N/2 \le a \le N - 1. \end{cases} \quad (5.1)$$

Consider the binary representation of an integer address for which $N = 2^d$. Individual bits at position i are denoted a_i.

$$a = a_{d-1}2^{d-1} + a_{d-2}2^{d-2} + \cdots + a_1 2 + a_0 \quad (5.2)$$

A perfect shuffle (5.1) of an address a creating a new address a' corresponds to a left cyclic shift of all bits a_i to a_{i+1} with the leftmost bit a_{d-1} wrapped around to a_0 [21].

$$a' = a_{d-2}2^{d-1} + a_{d-3}2^{d-2} + \cdots + a_0 2 + a_{d-1}$$

Our earlier requirement for a pairwise exchange of columns at a distance 2^h is easily satisfied, due to the geometry of a hypercube. The implication is that for addresses of the form (5.2), a difference in a single bit a_i indicates a distance of 2^i. We also note that the addresses of neighbouring processors in the hypercube differ in only one bit position. Exchanges, therefore, will always be between directly connected neighbours.

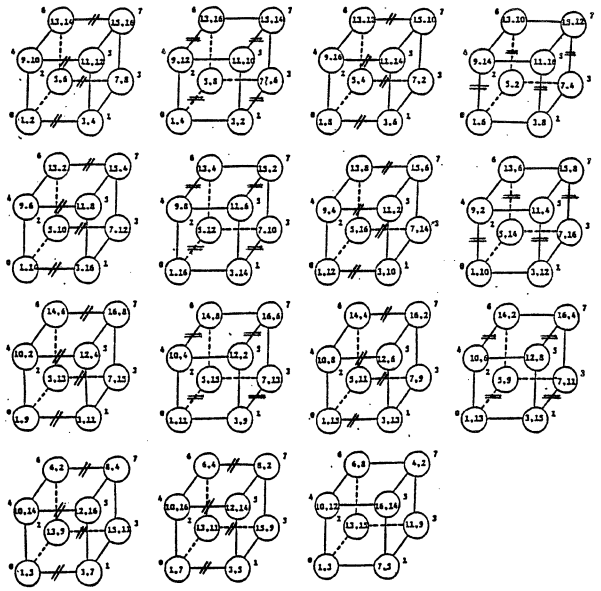
Processor nodes in a hypercube are labelled from 0 to $2^d - 1$, for example in a 3–dimensional cube there are 8 processors with addresses 0 to 7. We can use the perfect shuffle to generate processor pairings required for exchanges at a distance which is a power of 2. This may be illustrated by an example with $d = 3$. Initially processor pairings for exchanges are at a distance of

node	a	*node*	a'	*node*	a''
0	000	0	000	0	000
1	001	2	010	4	100
2	010	4	100	1	001
3	011	6	110	5	101
4	100	1	001	2	010
5	101	3	011	6	110
6	110	5	101	3	011
7	111	7	111	7	111

Figure 4: 3–Dimensional Processor Pairings

1. After a perfect shuffle from addresses a to a' exchanges may take place at a distance of 2, from a' to a'' at a distance of 4 and so on. Processor pairings before and after a perfect shuffle are given in figure 4.

The exchange and divide steps required to complete one sweep of a Jacobi–like algorithm, when $n = 2^4 = 16$ are illustrated in figure 5.



There are 15 computation steps. The column pairs (i,j) at each step are written in the processor nodes. The communication links used between the computation steps are marked.

Figure 5: Divide–Exchange on a 3–Cube

5.3 Computational Results

An implementation of Hestenes' one–sided SVD via the non–recursive version of our algorithm was written in 'C' for subsequent testing and analysis on the Intel iPSC hypercube. A simulator for the hypercube was provided by Intel Scientific Corp. to McGill University for a SUN 3/280 running the BSD 4.3 operating system. This SUN has an IEEE 754 standard co–processor with a floating point precision of $\varepsilon = 2.22 \times 10^{-16}$ in double precision arithmetic.

A threshold Jacobi method, as described in §2, was employed to insure proper termination of the algorithm. Following the methods introduced by Berry *et al* in [1] for computation on an array processor, each node processor in the hypercube maintains a counter *istop*. The counter is incremented by a processor when one of its assigned column pairs (i, j) is deemed to be orthogonal according to a threshold parameter τ. For the purposes of our tests we chose $\tau = \varepsilon \|A\|_F$.

The parallel computation terminates at the end of a sweep if each of the $n/2$ processors report *istop* counts of $n-1$. For a series of random 8×8 matrices generated using the interactive matrix software package *Matlab*, we typically observe convergence in the hypercube computation after 6 sweeps.

Finally we have observed a communication pattern for random 16×16 matrices matching exactly with that shown in figure 6.

6 Conclusions and Future Research

We have described a new optimal parallel Jacobi–like algorithm for the singular value decomposition (SVD). We have demonstrated that the new algorithm can be mapped naturally onto hypercube architectures, effectively utilizing the nearest neighbour communication capacity throughout the computation. In general, the recursive pairwise exchange communication operations of the new algorithm can be efficiently supported by multiprocessors with interconnect patterns used in many networks that have been proposed to support large–scale parallelism [20]. For example, we believe that the new algorithm can be mapped effectively onto SIMD or MIMD parallel computers with interconnection networks such as PM2I–based networks and cube–based networks These interconnection networks have the partitionability property: the ability to divide the network into independent subnetworks of different sizes [20], which match the recursive divide–exchange structure of the new parallel algorithm proposed in this paper.

We suggest the following future research directions: Study extensions of the new algorithm to various forms of the SVD, to the unsymmetric eigenvalue problem and to the generalized eigenvalue problem $Ax = \lambda Bx$. Furthermore, we would like to gather empirical information concerning convergence properties from numerical simulations.

7 Acknowledgements

Intel Scientific Corp. provided us with a hypercube simulator which allowed us to test the new algorithm. Martin Santavy gave many valuable comments concerning the details of software development for the hypercube, particularly in the area of synchronization problems. The figures were prepared with the help of Peggy Gao. We would especially like to thank Prof. Chris Paige for many helpful comments and corrections related to historical background and convergence results.

References

[1] M. Berry and A. H. Sameh, "Multiprocessor Jacobi algorithms for dense symmetric eigenvalue problems and singular value decompositions", *Proceedings of the International Conference on Parallel Processing*, 1986.

[2] C. Bischof, "The two–sided Jacobi method on a hypercube", *SIAM Proceedings of the Second Conference on Hypercube Multiprocessors*, 1987.

[3] R. P. Brent, F. T. Luk and C. F. Van Loan, "Computation of the singular value decomposition using mesh–connected processors", *J. VLSI Computer Systems*, 1, (1985), pp. 242–270.

[4] R. P. Brent and F. T. Luk, "The solution of singular–value and symmetric eigenvalue problems on multiprocessor arrays", *SIAM J. Sci. Stat. Comput*, 6 (1985), pp. 69–84.

[5] P. J. Eberlein, "On using the Jacobi method on the hypercube", *SIAM Proceedings of the Second Conference on Hypercube Multiprocessors*, 1987.

[6] G. E. Forsythe and P. Henrici, "The cyclic Jacobi method for computing the principal values of a complex matrix", *Trans. Amer. Math. Soc.*, 94, (1960), pp. 1–23.

[7] G. H. Golub and W. Kahan, "Calculating the singular values and pseudo–inverse of a matrix", *J. SIAM Ser. B: Numer. Anal.* 2, (1965), pp. 205–224.

[8] M. T. Heath, A. J. Laub, C. C. Paige and R. C. Ward, "Computing the singular value decomposition of a product of two matrices", *SIAM J. Sci. Stat. Comput.*, 7 (1986), pp. 1147–1159.

[9] M. R. Hestenes, "Inversion of matrices by biorthogonalization and related results", *J. Soc. Indust. Appl. Math.*, 6 (1958), pp. 51–90.

[10] E. G. Kogbetliantz, "Solution of linear equations by diagonalization of coefficients matrix", *Quart. Appl. Math.*, 13 (1955), pp. 123–132.

[11] C. Lawson and R. Hanson, *Solving Least Squares Problems*, Prentice–Hall, Englewood Cliffs, N.J., 1974.

[12] F. T. Luk, "A triangular processor array for computing singular values", *Linear Algebra Appl.*, 77 (1986), pp. 259–273.

[13] F. T. Luk and H. Park, "On parallel Jacobi orderings", Cornell University, School of Elec. Eng. Report, EE–CEG–86–5, 1986.

[14] J. J. Modi and J. D. Pryce, "Efficient implementation of Jacobi's diagonalization method on the DAP", *Numer. Math.*, 46 (1985), pp. 443–454.

[15] J. C. Nash, "A one–sided transformation method for the singular value decomposition and algebraic eigenproblem", *Comput. J.*, 18 (1975), pp. 74–76.

[16] C. C. Paige and P. Van Dooren, "On the quadratic convergence of Kogbetliantz's algorithm for computing the singular value decomposition", *Linear Algebra Appl.* 77 (1986), pp. 301–313.

[17] H. Rutishauser, "The Jacobi method for real symmetric matrices", *Numer. Math.*, 16, (1966), pp. 205–223.

[18] A. H. Sameh, "On Jacobi and Jacobi–like algorithms for a parallel computer", *Math. Comp.*, 25, (1971), pp. 579–590.

[19] A. H. Sameh, "Solving the linear least squares problem on a linear array of processors", *Algorithmically Specialized Parallel Computers*, Academic–Press, 1985, pp. 191–200.

[20] H. J. Siegel, *Interconnection Networks for Large–Scale Parallel Processing*, Lexington Books, D.C. Heath and Co., Mass., 1985.

[21] H. S. Stone, "Parallel processing with the perfect shuffle", *IEEE Trans. Comput.*, C–20 (1971), pp. 153–161.

[22] J. H. Wilkinson, "A note on the quadratic convergence of the cyclic Jacobi process", *Numer. Math.*, 4 (1962), pp. 296–300.

[23] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon–Press, Oxford, 1965.

Modeling and Optimal Scheduling of Parallel Sparse Gaussian Elimination

P. Sadayappan

Department of Computer and Information Science
The Ohio State University, Columbus, Ohio 43210

V. Visvanathan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Directed Acyclic Graphs (DAGs) have been extensively used to model parallel sparse Gaussian Elimination and its scheduling on multiprocessors, even though their use leads to sub-optimal schedules. In this paper, task graphs containing directed edges as well as undirected edges, called Minimally Constrained Task Graphs (MCTGs) are proposed to model parallel sparse Gaussian Elimination. An algorithm for scheduling MCTGs on multiprocessors is presented and the generated schedule is proven optimal. The scheme is evaluated using a number of practical matrices arising from circuit simulation and shown to be significantly better than scheduling using DAGs.

1. Introduction

The repeated solution of large sparse linear systems of equations using Gaussian Elimination (GE), or a variant thereof, is a computationally intensive component of many practical applications such as structural analysis, circuit simulation etc. Consequently there is considerable interest in parallelizing the solution of sparse matrices[1-9,11-15]. In order to identify the maximal degree of potential parallelism in sparse GE, it has been customary to view the computation at the level of elementary arithmetic operations using a directed acyclic graph (DAG). The vertices of such a DAG denote elementary arithmetic operations and edges between vertices represent execution dependencies between the operations. The dependence structure of the DAG is determined by a "symbolic" trace of the sequential form of the GE algorithm, creating edges from each given vertex to all vertices that use the value generated by it [2,5]. It has been recognized that the use of such a DAG for scheduling the operations of GE on a multiprocessor can result in sub-optimal schedules [5], but the resolution of this problem has not been previously pursued.

The problem with the use of a DAG based on symbolic unraveling of the sequential GE algorithm, for identifying dependence constraints on parallel execution of GE, is that the accumulative updates to any matrix element are unnecessarily constrained to take place in exactly the same order that they would be performed with sequential GE. In this paper, we propose the use of task graphs with directed edges as well as undirected edges to model parallel GE. Directed edges are used only to represent strict temporal dependencies, while undirected edges model constraints on the non-simultaneity of execution of multiple updates to a common matrix element. We refer to these task graphs as Minimally Constrained Task Graphs (MCTGs) and present an algorithm to schedule such graphs on a shared-memory multiprocessor. The optimality of scheduling parallel GE using the proposed scheduling algorithm is proved under the idealized model of a Concurrent Read Exclusive Write (CREW) multiprocessor with unbounded number of processors.

Optimal scheduling of DAGs on an idealized unbounded multiprocessor is very simply done using critical-path scheduling. Thus previous studies on parallel GE under the CREW model have typically focussed on heuristics for one of the following two (NP-complete) problems: 1) Given the zero-nonzero structure of a matrix, find a permutation for the rows/columns of the matrix so that the task graph (DAG derived from the dependence structure of the operations constituting sequential GE on the permuted matrix) has minimum depth (and hence minimal finishing time) [2,5,12,15]; and 2) Given a specific ordering (permutation) of rows/columns, find a schedule for a finite number of processors that minimizes finishing time [13,14]. In this paper, we do not focus on either of the above issues - matrix reordering for parallelism or scheduling on a limited number of processors. Rather, we focus on the fact that the underlying DAG-based model of parallel GE used by earlier studies is inherently overconstraining, and we provide an approach to avoid this problem using the notion of MCTGs. We present this framework under an idealized machine model; however the concept of MCTGs has wider ramifications and is more appropriate than a DAG-based model for the other problem formulations in this context.

The paper is organized as follows. In section 2, we use an example from [5] to explain the problem of suboptimality of scheduling with overconstrained DAGs. In section 3, we propose the use of MCTGs and provide an algorithm for scheduling MCTGs on an idealized multiprocessor. In section 4, we prove the optimality of the assignment generated by the scheduling algorithm for MCTGs arising with sparse GE. Section 5 is concerned with empirical evaluation of the algorithm. Various test matrices arising from the application domain of circuit simulation are used in the study. The scheduling algorithm proposed in this paper results in completion times that are up to forty percent less than that achieved by the DAG-based algorithm. Section 6 concludes the paper with a brief discussion.

2. DAG Based Scheduling of Parallel Sparse GE

We first outline the sparse GE algorithm for the solution of linear systems of equations. In solving the system $Ax = b$, where A is a sparse $N \times N$ matrix and b is an N-vector, values for the N-vector x are sought that satisfy the simultaneous equations. As is usual, for convenience of

representation of the algorithm, we represent the right hand side vector b as an additional $N+1$'st column of A.

In sparse GE, the order in which the variables are eliminated has a significant impact on the number of fill-ins (zero elements of A that become non-zero during the elimination process) created and hence the total number of arithmetic operations. Therefore, the actual elimination is preceded by an ordering phase, in which, based on the zero-nonzero structure of the matrix an elimination order which reduces the number of fill-ins is determined[10]. This ordering is then used for repeated solution of different sets of equations with the same zero-nonzero structure. Also, during the ordering phase the actual locations of the fill-ins are determined. Thus, in the following when we refer to a non-zero element of A it pertains to the filled-in matrix rather than the original one.

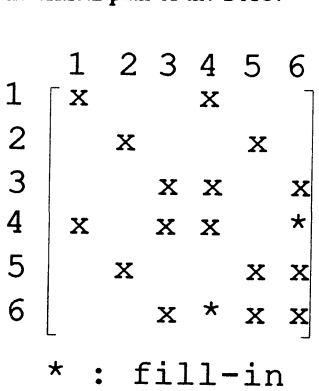
```
/* MATRIX TRIANGULATION */
M1  for k = 1,N
M2    for each j in [k+1,N+1] such that A_kj ≠ 0 do
        A_kj ← A_kj/A_kk
      endfor M2
M3    for each i in [k+1,N] such that A_ik ≠ 0 do
M4      for each j in [i+1,N+1] such that A_kj ≠ 0 do
          A_ij ← A_ij - A_ik*A_kj
        end for M4
      end for M3
    end for M1

/* BACK SUBSTITUTION */
B1  for k = N,1,-1
B2    for each j in [k+1 ,N] such that A_kj ≠ 0 do
        A_k,N+1 ← A_k,N+1 - A_kj*A_j,N+1
      end for B2
    end for B1
```

Figure 1. Sparse Gaussian Elimination Algorithm

GE (outlined in figure 1) consists of two steps - 1) matrix triangulation, and 2) back substitution. Matrix triangulation may be viewed as a sequence of elementary operations - either the division of a matrix element by the diagonal element in that row, or an incremental multiply-subtract (update) operation on a matrix element. The back-substitution phase of GE involves only update operations. We focus in our exposition only on the triangulation phase of GE, but all our observations relating to the update operations in the triangulation phase are directly applicable to the update operations of the back-substitution phase also. Thus, in the following, we sometimes refer to the triangulation phase of GE simply as GE. The scheduling issues for variants of Gaussian Elimination, such as LU factorization with forward/back substitution are also essentially the same.

Figure 2 shows an example of a sparse matrix (taken from [5]) and the sequence of elementary operations that constitute the triangulation phase of GE for this matrix.[1] This sequence of operations is obtained by symbolically tracing the above GE algorithm. A sequential execution of the GE algorithm involves stepping through this operation list in order. A parallel implementation of GE will require the same set of operations to be performed, each operation being executable as soon as data-dependence constraints are satisfied. The dependence constraints can be captured by a graph, as shown in figure 3a. Vertices of this graph represent the elementary operations of the operation list. A directed edge is drawn from a vertex to another if the value generated by the source vertex is utilized by the computation at the destination vertex. Such a DAG can be used to schedule the operations of GE on a parallel computer - a vertex (task) can be scheduled as soon as all the tasks that are represented as source vertices of its incoming edges have completed execution. Assuming an idealized shared-memory multiprocessor with arbitrary number of processors, where a divide operation and an update operation each take one unit of time, the parallel completion time is clearly the length of the critical path of the DAG.



1. $A_{14} \leftarrow A_{14}/A_{11}$
2. $A_{44} \leftarrow A_{44} - A_{41} \times A_{14}$
3. $A_{25} \leftarrow A_{25}/A_{22}$
4. $A_{55} \leftarrow A_{55} - A_{52} \times A_{25}$
5. $A_{34} \leftarrow A_{34}/A_{33}$
6. $A_{36} \leftarrow A_{36}/A_{33}$
7. $A_{44} \leftarrow A_{44} - A_{43} \times A_{34}$
8. $A_{46} \leftarrow A_{46} - A_{43} \times A_{36}$
9. $A_{64} \leftarrow A_{64} - A_{63} \times A_{34}$
10. $A_{66} \leftarrow A_{66} - A_{63} \times A_{36}$
11. $A_{46} \leftarrow A_{46}/A_{44}$
12. $A_{66} \leftarrow A_{66} - A_{64} \times A_{46}$
13. $A_{56} \leftarrow A_{56}/A_{55}$
14. $A_{66} \leftarrow A_{66} - A_{65} \times A_{56}$

Figure 2. A Sparse Matrix and its Triangulation [5]

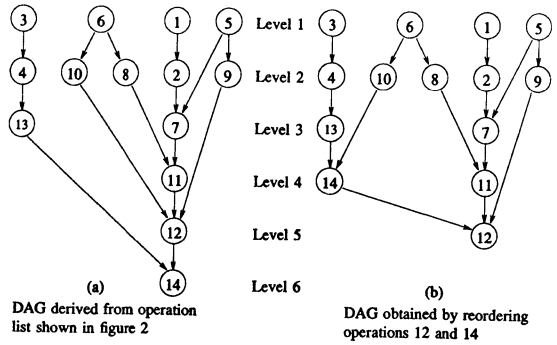


Figure 3. Non-Optimality of DAG-Based Scheduling

DAGs have formed the basis for prior studies relating to the parallel scheduling of GE [2,5,7,8,11-15]. They are however overly constraining because they require that multiple updates to a matrix element during a parallel execution occur in exactly the same order that they would have taken place if executed sequentially. The order in which multiple independent updates to a matrix element occur is clearly irrelevant as long as all of them are completed before the fully updated matrix element is used as an operand in some other operation. In the example used, as pointed out by Huang and Wing, operations 12 and 14 both represent (independent) update operations on A_{66}, and can therefore be

1. In order to remain consistent with the example used in [5] the operations on the r.h.s. vector are omitted. However, this does not affect our presentation.

executed in any order without affecting the final results computed. Figure 3b shows the DAG that results from interchanging these two operations; it has a shorter critical path compared to the original DAG.

3. Minimally Constrained Task Graphs and their Scheduling

The use of DAGs based on the dependence structure of operations of sequential GE is thus overly constraining with respect to the update operations in scheduling GE for parallel execution. All independent update operations to a matrix element should be independently schedulable except that no two of them can occur simultaneously. The use of directed edges between such update operations thus forces an unnecessary and arbitrary precedence constraint, whereas all that is really required is a weaker "non-simultaneity" constraint. We propose the use of task graphs, called Minimally Constrained Task Graphs (MCTGs), where a clear distinction is made between strict temporal ordering constraints and non-simultaneity constraints.

MCTGs use both directed edges and undirected edges. Directed edges are used as before to represent temporal dependence constraints. Non-simultaneity constraints are expressed using undirected edges. Two operations that are prohibited from occurring at exactly the same time, but are otherwise executable in either order are connected by an undirected edge. The set of update operations to a matrix element in GE can be performed in any order and will produce the same final result (except for round-off errors) due to the commutativity and associativity of the addition. Thus all update operations on any matrix element form a clique of vertices in the MCTG, connected among themselves by undirected edges, as shown in figure 4 for the same example as before. Directed edges are used between each update operation and succeeding operations that use the final updated value. As a matter of terminology, we refer to nodes that are connected by undirected edges as *sibling* nodes and the relationship represented by the edge as a sibling relationship. The notation (i,j) is used to denote the undirected edge between nodes i and j. A directed edge from node n to node m is denoted by $<n,m>$. The node n is referred to as *a parent* of the node m, while the latter is called *a child* of the former.

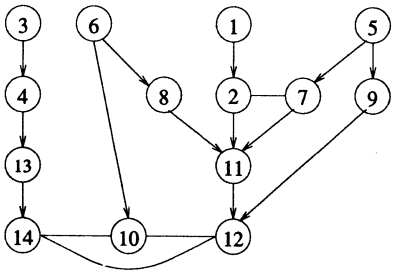


Figure 4. Minimally Constrained Task Graph for Example

We now present the formal definition of GE MCTGs. Given a sparse matrix and the sequence of operations that constitute its triangulation (see figure 2 for an example):

For each update operation i, d_i, f_i and s_i denote the destination, factor and source elements respectively, *i.e.*, $d_i \leftarrow d_i - f_i \times s_i$. By definition, d_i and f_i belong to the same row of the matrix. For each normalization operation i, d_i and f_i denote the destination and factor elements respectively, *i.e.*, $d_i \leftarrow d_i / f_i$. For each operation i, r_i denotes the position of i in the operation list. Further[2]

$$D(i) \equiv \underset{\{k \mid r_k < r_i \text{ and } d_i = d_k\}}{\arg\max} r_k$$

and,

$$F(i) \equiv \underset{\{k \mid r_k < r_i \text{ and } f_i = d_k\}}{\arg\max} r_k$$

In addition, for update operations we similarly define

$$S(i) \equiv \underset{\{k \mid r_k < r_i \text{ and } s_i = d_k\}}{\arg\max} r_k$$

We now define the GE MCTG as follows:

For any pair of update operations, i,j, the undirected edge (i,j) exists if and only if:

$d_i = d_j$

For each update operation i, the directed edge $<j,i>$ exists if and only if:

1. $j = S(i)$ or
2. $j = F(i)$ or
3. $(j,F(i))$ exists

For each normalization operation i, the directed edge $<j,i>$ exists if and only if:

1. $j = D(i)$ or
2. $(j,D(i))$ exists or
3. $j = F(i)$ or
4. $(j,F(i))$ exists

Note from the definition of undirected edges in the GE MCTG that they form cliques between tasks that constitute the updates of a common matrix element. We refer to this feature as *the clique property*. Regarding directed edges, first note that for any update operation i, $S(i)$ exists, and is a normalization operation. Hence, if a task has a sibling, it must have a parent. Further, $S(i)$ does not belong to a clique. Therefore, the child of any member of a clique is a child of every member of that clique. We refer to this last feature as *the common children property*.

For purposes of comparison, we now use the above terminology to define the GE DAG used by Huang and Wing[5] and subsequent researchers.

In a GE DAG:

For each update operation i, the directed edge $<j,i>$ exists if and only if:

1. $j = D(i)$ or
2. $j = F(i)$ or
3. $j = S(i)$ or

For each normalization operation i, the directed edge $<j,i>$ exists if and only if:

1. $j = D(i)$ or
2. $j = F(i)$ or

Thus, the interpretation of directed edges in MCTGs is slightly different from the interpretation when using

2. $\arg\max_{x \in X} f(x)$ is the value of x in X at which the maximum value of $f(x)$ is attained.

DAGs. With DAGs, an edge represented a temporal constraint in that a data result produced by a parent operation was needed and directly used by a child operation in the DAG. With MCTGs, a directed edge again represents a temporal execution constraint in that the parent operation necessarily has to be completed before the child operation can be executed. However, the value produced by the execution of the parent operation is not necessarily directly used as an operand for the child operation. This interpretation of a directed edge permits the necessary flexibility in scheduling the multiple updates of a matrix element to optimize GE completion time. Thus, unlike GE DAGs that are irredundant [14], an MCTG is not an irredundant graph; but as can be seen in the next section, this poses no problems in its optimal scheduling.

We now present an algorithm for scheduling MCTGs on an idealized CREW multiprocessor. As has typically been assumed in prior treatments on scheduling parallel GE [2,13,14], we consider an update operation and a divide operation to take the same (unit) amount of time. The algorithm however can be trivially generalized to handle non-uniform execution times for the various operations. The scheduling problem may be viewed as that of the assignment of positive integer level numbers to the nodes of the MCTG so that:

1) each node has a level number that is higher than that of any of its parent nodes (if any),
2) no two sibling nodes are assigned the same level, and
3) the highest assigned level number is as small as possible.

```
/* ALA : ALGORITHM for LEVEL ASSIGNMENT */

/* initialization */
Z←∅
for each root node m of G do
    L̄_m←1 ;
    Z←Z∪{m} ;
end for
for each non-root node n of G do
    P̄_n← Number of parent nodes of n ;
    Ē_n←1 ;
    F̄_n←false ;
end for

/* main */
while Z is not empty do
    Remove a node n from Z ;
    for each child m of n in G do
        Ē_m←max(Ē_m , L̄_n + 1) ;
        P̄_m←P̄_m−1 ;
        if P̄_m = 0 then
            L̄_m←Ē_m ;
            while any sibling i of m has
                (F̄_i = true and L̄_i = L̄_m) do
                    L̄_m←L̄_m + 1 ;
            F̄_m←true ;
            Z←Z∪{m} ;
        end if
    end for
end while
```

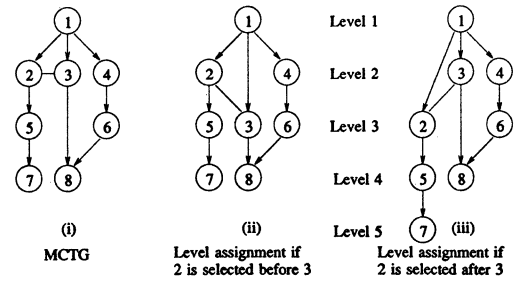
Figure 5. Level Assignment Algorithm

Assuming unit execution times for the operations of the MCTG, the level number assigned to a node corresponds to the earliest time at which that operation can be scheduled for execution on an idealized CREW multiprocessor. The level assignment algorithm shown in figure 5 associates a quadruple $(\bar{P},\bar{L},\bar{E},\bar{F})$ with each node of the MCTG G. $\bar{P}_n$ is a counter associated with node n that is initialized to the number of parent nodes of n in G. $\bar{L}_n$ is the level number assigned to n. $\bar{E}_n$ represents the earliest possible level assignable to n, based on directed-edge constraints; it is initialized to 1, and successively modified as the algorithm is executed. $\bar{F}_n$ is a flag associated with node n, to keep track of whether or not its level assignment has been finalized yet.

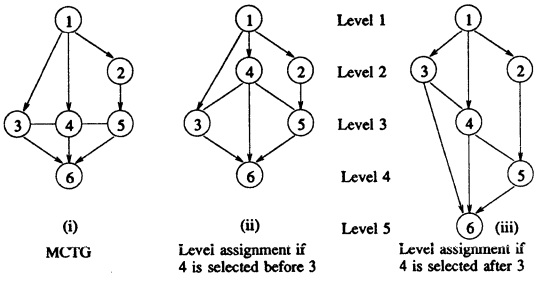
The algorithm essentially traverses the directed edges of G, ensuring that temporal dependence constraints are satisfied. Each directed edge $<n,m>$ is only traversed once, and only after the source node n has been assigned a level number by the algorithm. All root nodes (nodes without any incoming directed edges or undirected sibling edges) are initially assigned a level number of 1 and placed into an operating set Z. This set Z is used to temporarily maintain nodes whose level numbers have been finalized, until all outwardly directed edges from them have been traversed by the algorithm. As each edge $<n,m>$ is traversed, $\bar{E}_m$ is updated to be at least $\bar{L}_n + 1$, if it is not already so. $\bar{P}_m$, the counter associated with node m is decremented by one. If the currently traversed edge $<n,m>$ is the last incoming edge to m to be traversed, then $\bar{P}_m$ becomes zero, and node m is assigned its finalized level number $\bar{L}_m$. The earliest level it is schedulable at is its current value of $\bar{E}_m$, provided that none of its siblings (if any) has already been assigned at that level. If $\bar{E}_m$ is prohibited for node m due to sibling conflict, then $\bar{E}_m + 1$ is tried, and so on until the lowest conflict-free level is determined and assigned to $\bar{L}_m$. $\bar{F}_m$ is now set *true* to mark the assignment of a level to node m, and m is added to the operating set Z.

For graphs without undirected edges, the above algorithm reduces to the conventional DAG critical-path scheduling algorithm, and will clearly produce a unique, optimal levelization irrespective of the order of selection of nodes from the operating set Z for edge traversal. However, in general, when undirected edges are present, different orders of selection of nodes out of Z and of traversing the directed edges emanating from them can result in different schedules. This is illustrated by the examples in figure 6. In figure 6a, after node 1 (the only root-node) is assigned level number one, its outgoing edges could be traversed in any order. If <1,2> is traversed before <1,3>, the levelization in (ii) results, but if child node 3 is selected before node 2, then the different levelization shown in (iii) is the outcome.

Nevertheless, in the case of GE MCTGs, it can be shown that all possible schedules, produced by various selection orders, have the same (optimal) finishing time. This is a consequence of the Clique and Common Children properties of GE MCTGs that guarantee the optimality of the "greedy", "on-the-fly" approach to sibling conflict resolution adopted by ALA. Considering a set of nodes of G that form a clique, if these nodes have distinct earliest-schedulable-times, then they will each be so scheduled, leading to optimal scheduling. If some of these earliest-schedulable-times coincide at a value, say l, the fact that these nodes form a clique guarantees that no matter which node is visited first and assigned the level l, there will be a conflict of the



(a) Example where Common-Children property is not satisfied by MCTG



(b) Example where Clique property is not satisfied by MCTG

Figure 6. Selection-Order Dependence of Schedule

same number of clique sibling nodes at level l+1. As a result, the maximum of the levels assigned to the members of the clique will be the same (and optimal) independent of the specific level that is assigned to each of them. Due to the common-children property, any child of a clique node is also a child of all other nodes of that clique. Thus, the earliest-schedulable-time of the child of a clique is the same, independent of the order in which its parents in the clique were visited. These two properties therefore result in the fact that the ALA schedule for GE MCTGs is optimal.

The examples in figure 6 demonstrate the necessity of the clique property and the common-children property. The example of figure 6a violates the common-children property, and the two different selection orders shown result in different schedules, one of which is suboptimal. Figure 6b demonstrates the same point with respect to the clique property. In the following section, we first prove the optimality of the schedule generated by ALA for GE MCTGs. We then prove that any schedule that satisfies the constraints of the GE DAG for a matrix also satisfies the constraints of the corresponding GE MCTG. These two results imply that for any matrix, the ALA scheduling of the MCTG will result in a finishing time that is less than or equal to that produced by critical path scheduling of the corresponding DAG. The empirical results reported in section 5 show that the reduction in finishing time can be as much as forty percent for matrices arising in circuit simulation.

4. Optimality of the Algorithm for Level Assignment

As has been discussed in the previous section, GE MCTGs have the following three properties which will be used to prove the optimality of ALA for scheduling GE MCTGs.

Property 1: Sibling nodes form cliques. ■

Property 2: Any two sibling nodes have the same set of children nodes. ■

Property 3: If a node has a sibling, then it must have a parent. ■

4.1 Preliminaries

We begin by introducing some notation.

$V \equiv$ set of all nodes in the MCTG
$S \equiv \{n \in V \mid n \text{ has no sibling}\}$ (solitary nodes of V)
$R \equiv \{n \in V \mid n \text{ has no parent}\}$ (root nodes of V)

Property 3 can now be restated as:

$$R \subset S \tag{1}$$

For a node m,

$P_m \equiv \{n \in V \mid n \text{ is a parent of } m\}$ (parents of m)
$S_m \equiv P_m \cap S$ (solitary parents of m)

For GE MCTGs, since sibling nodes form cliques, we have the following additional notation:

$\mathbf{C} \equiv$ family of all cliques in the MCTG (of cardinality ge 2)

For a node m,

$\mathbf{Q}_m \equiv \{I \in \mathbf{C} \mid I \cap P_m \neq \emptyset\}$ (clique parents of m)

and,

$$Q_m \equiv \bigcup_{I \in \mathbf{Q}_m} I$$

Using the above notation, we can restate Property 2 as:

$$P_m = S_m \cup Q_m \qquad \forall\, m \in V \tag{2}$$

We now introduce some definitions and notation pertaining to level assignment algorithms. We use L_i to denote the level assigned to node i.

Definition 1: Given an MCTG G, a *valid assignment* is an assignment of levels (natural numbers) to the nodes of G such that:
1) If $<i,j>$ is a directed edge of G, $L_i < L_j$, and
2) If (i,j) is an undirected edge in G, $L_i \neq L_j$ ■

For a clique I, let

$$C_I \equiv \max_{i \in I} L_i$$

that is, C_I denotes the *completion time* of clique I. It follows from the above definitions that

$$\max_{I \in \mathbf{Q}_m} C_I = \max_{i \in Q_m} L_i \qquad \forall\, m \in V \tag{3}$$

The *minimum possible level* that can be assigned to the *solitary node m* by any valid assignment is denoted by $\hat{L}_m$, while the *minimum possible completion time* that can be assigned to the *clique I* by any valid assignment is denoted by $\hat{C}_I$.

Definition 2: A valid assignment (that assigns levels L_i) is said to be *optimal* if

$L_i = \hat{L}_i \qquad \forall\, i \in S$
$C_I = \hat{C}_I \qquad \forall\, I \in \mathbf{C}$ ■

We now prove a simple result pertaining to optimal assignments.

Lemma 1: For all $m \in S-R$,

$$\hat{L}_m = \max\left(\max_{i \in S_m} \hat{L}_i, \max_{I \in Q_m} \hat{C}_I\right) + 1$$

Proof: Consider any assignment (which assigns levels L_i) that is optimal. Due to its optimality,

$$L_m = \max_{i \in P_m} L_i + 1$$

It now follows from (2) and (3) that

$$L_m = \max\left(\max_{i \in S_m} L_i, \max_{I \in Q_m} C_I\right) + 1$$

The lemma now follows from Definition 2. ■

4.2 Clique Resolution Procedure

The key step in proving the optimality of ALA is showing the optimality of the resolution of sibling conflicts in the algorithm. We therefore consider an abstract clique resolution procedure in this section, prove its optimality and then use this result in the following subsection to prove the optimality of ALA.

Problem $R(I,E_i)$: Given a set I and $E_i \in$ N (the set of natural numbers), associated with each $i \in I$, assign $L_i \in$ N such that

1. $L_i \geq E_i \forall\ i \in I$
2. $L_i \neq L_k \forall\ i,k \in I,\ i \neq k$ ■

Any assignment of values for L_i that satisfies the above two requirements is called a *valid resolution* for $R(I,E_i)$, while one that minimizes $\max_{i \in I} L_i$ over all possible valid resolutions is called an *optimal resolution* for $R(I,E_i)$. As we shall prove in the following, the procedure outlined below results in an optimal resolution for $R(I,E_i)$.

```
/* CRP : Clique Resolution Procedure */
for all i ∈ I do
    F_i ← false ;
    L_i ← E_i ;
end for
while there exists i ∈ I with F_i = false do
    while there exists m ∈ I with F_m = true and L_m = L_i do
      L_i ← L_i + 1 ;
    F_i ← true ;
end while
```

Note that as stated, CRP does not specify a precise sequence in which the F_i's are set to *true*. We therefore have the following definition.

Definition 3: A *CRP Selection Order* is the sequence in which the F_i's are set to *true* by a particular instance of CRP. ■

For an assignment corresponding to a selection order, we define

$$C \equiv \max_{i \in I} L_i$$

$$T_j \equiv \begin{cases} 1 & \text{if } \exists\ k \in I \mid L_k = j \\ 0 & \text{otherwise} \end{cases}$$

It follows from the definition of T_j that

$$C = \max_{\{j \mid T_j = 1\}} j$$

Observe that for any selection order, when an element i is selected for the assignment of a value to L_i, the value that is assigned is the smallest available integer that is greater than or equal to E_i. We formalize this fact in the following lemma.

Lemma 2: For any CRP Selection Order, if $T_j = 0$, then

$$L_i > j \Rightarrow E_i > j \qquad \forall\ i \in I \blacksquare$$

We now present the main result of this subsection.

Theorem 1: Every CRP Selection Order results in an optimal resolution for $R(I,E_i)$.

Proof: Let $\tilde{O}$ (that assigns values $\tilde{L}_i$) denote the "worst" CRP Selection Order, that is,

$$\tilde{C} \equiv \max_{i \in I} \tilde{L}_i = \max_{\text{all CRP selection orders}} C$$

Let,

$$\tilde{T}_j \equiv \begin{cases} 1 & \text{if } \exists\ i \in I \mid \tilde{L}_i = j \\ 0 & \text{otherwise} \end{cases}$$

Suppose that the proposition of the theorem is false. Then there exists a valid resolution (called $\hat{O}$) which assigns levels $\hat{L}_i$ such that

$$\hat{C} \equiv \max_{i \in I} \hat{L}_i < \tilde{C} \tag{4}$$

Let,

$$\hat{T}_j \equiv \begin{cases} 1 & \text{if } \exists\ i \in I \mid \hat{L}_i = j \\ 0 & \text{otherwise} \end{cases}$$

Since any valid resolution assigns a unique value L_i for each $i \in I$ (requirement 2 of problem $R(I,E_i)$), it follows from (4) that there exists at least one level $l < \hat{C}$ at which $\hat{O}$ has assigned an element but $\tilde{O}$ has not, that is,

$$\tilde{T}_l = 0 \text{ and } \hat{T}_l = 1$$

Let l^* be the largest such l. For all $l > l^*$, one and only one of the following is true:

$$\tilde{T}_l = \hat{T}_l = 1 \tag{5a}$$
$$\tilde{T}_l = \hat{T}_l = 0 \tag{5b}$$
$$\tilde{T}_l = 1 \text{ and } \hat{T}_l = 0 \tag{5c}$$

Let,

$$\tilde{M} \equiv \mid \{i \in I \mid \tilde{L}_i > l^*\} \mid$$
$$\hat{M} \equiv \mid \{i \in I \mid \hat{L}_i > l^*\} \mid$$

From (4) and (5),

$$\tilde{M} > \hat{M} \tag{6}$$

Since $\tilde{T}_{l^*} = 0$, it follows from Lemma 2 and (6) that

$$\mid \{i \in I \mid E_i > l^*\} \mid \geq \tilde{M} > \hat{M}$$

which contradicts the assumption that $\hat{O}$ satisfies requirement 1 of problem $R(I,E_i)$. Hence the lemma. ■

4.3 Proof of Optimality

We return now to the proof of the assertion that ALA results in the minimum possible number of levels. Recall from Section 3 that the levels assigned by ALA are denoted by $\bar{L}_i$. Further,

$$\bar{L}_i \geq \bar{E}_i \qquad \forall\, i \in V\text{–}R \tag{7}$$

and

$$\bar{L}_i = \bar{E}_i \qquad \forall\, i \in S\text{–}R \tag{8}$$

For a clique I, let

$$\bar{C}_I \equiv \max_{i \in I} \bar{L}_i$$

that is, $\bar{C}_I$ is the completion time assigned to clique I by ALA.

Lemma 3: For all $m \in V\text{–}R$

$$\bar{L}_i = \hat{L}_i \forall\, i \in S_m \text{ and } \bar{C}_I = \hat{C}_I \forall\, I \in \mathbf{Q}_m$$

$$\Rightarrow\ \bar{E}_m = \max\left(\max_{i \in S_m} \hat{L}_i, \max_{I \in \mathbf{Q}_m} \hat{C}_I\right) + 1$$

Proof: Let $m \in V\text{–}R$ be arbitrary. By the construction of the algorithm,

$$\bar{E}_m = \max_{i \in P_m} \bar{L}_i + 1$$

But due to (2) and (3),

$$\bar{E}_m = \max\left(\max_{i \in S_m} \bar{L}_i, \max_{I \in \mathbf{Q}_m} \bar{C}_I\right) + 1$$

The lemma now follows. ■

Lemma 4: For all $I \in \mathbf{C}$

$$\bar{L}_j = \hat{L}_j \forall\, j \in \bigcup_{i \in I} S_i \text{ and } \bar{C}_J = \hat{C}_J \forall\, J \in \bigcup_{i \in I} \mathbf{Q}_i$$

$$\Rightarrow\ \bar{C}_I = \hat{C}_I$$

Proof: Let $\hat{A}$, which assigns levels L_i, be an optimal assignment. Let $I \in \mathbf{C}$ be arbitrary. Since $\hat{A}$ is optimal, for all $i \in I$

$$L_j = \hat{L}_j \qquad \forall\, j \in S_i$$
$$C_J = \hat{C}_J \qquad \forall\, J \in \mathbf{Q}_i$$

It now follows from the assumptions of the lemma, Lemma 3, and the fact that $\hat{A}$ is a valid assignment that

$$L_i \geq \bar{E}_i \qquad \forall\, j \in S_i$$
$$L_i \neq L_j \qquad \forall\, i,j \in I,\ i \neq j$$

In other words, the assignment produced by $\hat{A}$ for the nodes in I is a valid resolution for $R(I,\bar{E}_i)$. Now, in ALA, different selection orders for the nodes in the operating set Z and the directed edges to their children will result in different sequences in which the $\bar{F}_i$'s are set to *true*. However, each one of these sequences corresponds to a CRP Selection Order for $R(I,\bar{E}_i)$, which by Theorem 1 is an optimal resolution. But since we have already shown that the optimal assignment $\hat{A}$ is a valid resolution for $R(I.\bar{E}_i)$, it follows that $\bar{C}_I = C_I$. ■

Theorem 2: ALA results in the minimum possible number of levels.

Proof: Note first that due to (1) and the construction of the operating set Z in the initialization phase of ALA that

$$\bar{L}_i = 1 = \hat{L}_i \forall\, i \in R \tag{9}$$

From Lemmas 1 and 3 and (8)

$$\bar{L}_i = \hat{L}_i \forall\, i \in S_m \text{ and } \bar{C}_I = \hat{C}_I \forall\, I \in \mathbf{Q}_m \tag{10}$$

$$\Rightarrow\ \bar{L}_m = \hat{L}_m \ \forall\, m \in S\text{–}R$$

Now consider a graph derived from the MCTG in which each clique is collapsed into a single node. It follows from [14, Lemma 1] that this graph is a DAG. Hence from (9), (10) and Lemma 4, it follows via induction that ALA is an optimal assignment, which implies that it results in the minimum possible number of levels. ■

We conclude this section by proving that for any matrix, the number of levels generated by any level assignment procedure that satisfies the constraints of the corresponding GE DAG (defined in section 3) will be greater than or equal to the number resulting from an application of ALA to the corresponding MCTG. This is done by showing that any valid assignment for the DAG is a valid assignment for the MCTG.

Definition 4: A level assignment A (that assigns level L_k to node k) is said to be valid for a given DAG if:

$\langle i,j \rangle$ is a directed edge of the DAG $\Rightarrow$ $L_i < L_j$ ■

Theorem 3: A level assignment that is valid for the GE DAG corresponding to a matrix is a valid assignment for the corresponding GE MCTG.

Proof: Let A be a valid assignment for the DAG and (n,m) an arbitrary undirected edge in the MCTG. Since n and m represent updates to the same matrix element, in the DAG there exists a chain of edges from n to m or vice versa. In either case A cannot assign the same level to both n and m, *i.e.*, A satisfies all the constraints imposed by the undirected edges of the MCTG.

Let $\langle j,i \rangle$ be an arbitrary directed edge in the MCTG. If $\langle j,i \rangle$ exists in the DAG then clearly A satisfies the constraint imposed by it. If $\langle j,i \rangle$ does not exist in the DAG, it follows from the definition of the DAG and the MCTG (*c.f.* section 3) that j belongs to a sibling clique in the MCTG. Suppose that j belongs to the same clique as $F(i)$. Since in the DAG there is a chain of edges from j to i that goes through $F(i)$, A will assign a level to j that is less than the level it assigns to i. The same argument holds if i is a normalization operation and j belongs to the same clique as $D(i)$. Hence A satisfies all the constraints imposed by the directed edges of the MCTG. ■

5. Empirical Evaluation and Discussion

The MCTG scheduling algorithm ALA was evaluated empirically using matrices deriving from the application domain of circuit simulation. The three examples used arose in the simulation of portions of a Digital Signal Processor, a Digital-to-Analog Converter, and a Memory circuit respectively. The matrices were first reordered using the Markowitz ordering scheme [10]. The elementary arithmetic operations for GE under this ordering were generated and scheduled using a) the conventional DAG-based scheme, where the dependencies were determined using a symbolic trace of the sequential GE algorithm, and b) using the MCTG scheduling algorithm ALA presented in Sec. 3.

Table 1 lists some of the characteristics of the test matrices used and presents the results obtained for scheduling the triangulation phase of GE. We report the number of levels in the generated schedules and the average number of operations per level. The former represents the finishing time using an idealized CREW multiprocessor while the latter is representative of the average degree of parallelism

exploitable in GE triangulation. The MCTG-based schedule can be seen to provide 23% - 39% percent improvement over the DAG-based approach for the examples considered.

TABLE 1. Comparison of DAG-Based Levelization and ALA

Matrix	Size	No. of Ops.	No. of Levels	
			DAG	ALA
DSP	93	812	30	23
DAC	147	882	49	30
MEM	587	7048	109	77

The reduction in finishing time obtained with the MCTG-based scheduling scheme is comparable to that reported in the literature for matrix reordering schemes targeted at increasing parallelism in GE. Further, the decrease in finishing time obtained by the various proposed reordering heuristics is --unlike the MCTG-based approach-- typically at the expense of increased total number of arithmetic operations [2,3,12]. An interesting open question is whether the use of the MCTG-based scheduling scheme will provide comparable improvements in the number of levels with these matrix reordering schemes targeted at increasing parallelism, as it has for schedules based on the Markowitz ordering scheme. In any case, comparisons of different matrix reordering schemes with respect to the degree of parallelism exploitable, should be based on the less constrained MCTG-based schedule rather than the conventionally used DAG-based schedule.

The MCTG-based approach also has implications on the scheduling of practical finite-processor parallel systems. One approach to the parallel execution of GE on a multiprocessor is to use barrier synchronization between levels in the schedule. The greater average degree of parallelism obtainable with an MCTG-based schedule than the conventional DAG-based schedule implies better load-balancing on a multiprocessor with the former schedule. Further, since there are fewer levels with the MCTG-based schedule, fewer barrier synchronization points are required. Thus less overhead can be expected with an MCTG-based schedule, even though the actual performance improvement achieved will depend significantly on various machine characteristics and implementation dependent factors. It can be expected though, that the use of the inherently less constrained MCTG model, in conjunction with an appropriate characterization of the performance of a practical multiprocessor, can lead to more effective scheduling schemes than the use of the conventional DAG-based model.

6. Conclusions

In sum, we have presented a new approach to modeling task graphs for scheduling on a shared-memory multiprocessor. The key idea is the use of undirected edges to connect tasks (such as the additive updates of a matrix location) that cannot be done simultaneously but can be executed in either order. Such a task graph is inherently less constraining than one in which only directed edges are used.

We have developed a scheduling algorithm for these Minimally Constrained Task Graphs, and showed the algorithm to be optimal for Gaussian Elimination task graphs under an idealized Concurrent Read Exclusive Write multiprocessor model. We have empirically evaluated the proposed scheduling algorithm using sparse matrices derived from circuit simulation of sample electronic circuits, and showed it to provide up to forty percent improvement over the conventional approach.

Even though an idealized machine model has been used in this paper to present the Minimally Constrained Task Graph approach, the approach holds promise in the context of scheduling computations on real multiprocessors. The refinements required to accommodate characteristics of practical finite-processor systems for their effective scheduling are open questions for future research.

Acknowledgements

The authors would like to thank Eric Grosse and Sailesh Rao for their careful scrutiny of the manuscript and valuable comments.

References

[1] G. Alghband and H.F. Jordan, "Multiprocessor Sparse L/U Decomposition with Controlled Fill-in," *Tech. Rep. 85-48*, ICASE, NASA Langley Rsch. Center, Hampton, VA, 1985.

[2] R. Betancourt, "Efficient Parallel Processing Technique for Inverting Matrices with Random Sparsity," *IEE Proceedings*, Vol. 133, Pt. E, No. 4, pp. 235-240, July 1986.

[3] P. Cox, R. Burch and B. Epler, "Circuit Partitioning for Parallel Processing," *Proc. Intl. Conf. on Computer-Aided Design*, pp. 186-189, Santa Clara, CA, Nov. 1986.

[4] I. S. Duff, "Parallel Implementation of Multifrontal Schemes," *Parallel Computing*, vol. 3, pp. 193-204, 1986.

[5] J. W. Huang and O. Wing, "Optimal Parallel Triangulation of a Sparse Matrix," *IEEE Trans. on Circuits and Systems*, Vol. CAS-26, No. 9, pp.726-732, September 1979.

[6] G. K. Jacob, A. R. Newton and D. O. Pederson, "Parallel Linear-Equation Solution in Direct-Method Circuit Simulators," *Proc. Intl. Symposium on Circuits and Systems*, pp. 1056-1059, Philadelphia, PA, May 1987.

[7] J. A. G. Jess and H. G. M. Kees, "A Data Structure for Parallel L/U Decomposition," *IEEE Trans. on Computers*, Vol C-31, No. 3, pp. 231-238, March 1982.

[8] J. W. H. Liu, "Computational Models and Task Scheduling for Parallel Sparse Cholesky Factorization," *Parallel Computing*, vol. 3, pp. 327-342, 1986.

[9] R. Lucas, T. Blank and J. Tiemann, "A Parallel Solution Method for Large Sparse Systems of Equations," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 6, pp. 981-991, November 1987.

[10] H. M. Markowitz, "The Elimination Form of the Inverse and its Application to Linear Programming," *Management Science*, Vol. 3, pp. 255-269, April 1957.

[11] F. J. Peters, "Parallel Pivoting Algorithms for Sparse Symmetric Matrices," *Parallel Computing*, vol. 1, pp. 99-110, 1984.

[12] D. Smart and J. White, "Reordering Algorithms to Reduce Parallel Solution Time of Sparse Matrices Associated with Circuit Simulation," *Proc. Intl. Symposium on Circuits and Systems*, to appear, 1988.

[13] M. A. Srinivas, "Optimal Parallel Scheduling of Gaussian Elimination DAG's" *IEEE Trans. on Computers*, Vol. C-32, No. 12, pp. 1109-1117, December 1983.

[14] O. Wing and J. W. Huang, "A Computational Model of Parallel Solution of Linear Equations," *IEEE Trans. on Computers*, Vol. C-29, no. 7, pp. 632-638, July 1980.

[15] V. Zhou, "Optimal Parallel Triangulation of a Sparse Matrix - A Graphical Approach," *Proc. Intl. Symposium on Circuits and Systems*, 1981, pp. 624-627.

PERFORMANCE OF PARALLEL PARTITIONING ALGORITHMS †

Sridhar Madala
James B. Sinclair

Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77251-1892

Abstract

Task graphs of parallel algorithms which are based on the divide-and-conquer strategy often exhibit a characteristic structure known as the partitioning structure. We present some new methods for bounding and approximating the mean execution time of a partitioning structure when the execution times for the tasks are non-deterministic and compare them with previous approaches. Distribution-driven simulation results show that two of the methods, the iterative approximation and the independent paths approximation, provide accurate estimates, usually to within 10 percent. Results from program-driven simulation of a parallel quicksort algorithm running on the Rice Parallel Processing Testbed indicate that the methods give good estimates even when certain independence assumptions are violated. The independent paths approximation is used to derive an analytical expression for the mean execution time of a parallel mergesort algorithm.

1. Introduction

A common approach to solving problems is to partition the problem into smaller parts, find solutions for the parts, and then combine the solutions for the parts into a solution for the whole. This divide-and-conquer strategy, applied recursively, is the basis for several classes of parallel algorithms, including a number of sorting and searching algorithms. These algorithms typically consist of three phases: a divide phase during which work is partitioned, a work phase during which computation is performed on the partitions, and a merge phase during which results from the previous steps are combined. Task graphs of such algorithms have a characteristic structure known as the partitioning structure [1]. A classic example of such an algorithm is the quicksort algorithm which partitions an array of elements to be sorted into two subarrays, each of which is subdivided recursively until the number of elements in a subarray is below a threshold. The work phase consists of sorting the elements in the subarray. The merge phase is either non-existent (if the partitioning and sorting are done in place), or trivial (if the partitioning and sorting are done on copies). The mergesort is a similar algorithm with a non-existent or trivial divide phase and non-trivial work and merge phases.

Figure 1 shows a two-stage partitioning task graph structure. Each node in the graph represents a computational task and each edge represents a dependency between tasks. A task a is said to be a predecessor of a task b if there is a directed edge from a to b. Tasks without predecessors are called initial tasks and tasks that are not the predecessors of any task are called final tasks. A task cannot start until all its predecessor tasks have completed execution and once started a task runs to completion without interruption. The level of a task is the length of the longest path from an initial task to that task. The execution time for the graph is the time from the start of an initial task to the completion of all the tasks. The number of stages in a partitioning structure is the number of divide levels or the number of merge levels. The branching factor is the number of successors to each divide task or equivalently the number of predecessors to each merge task. Many algorithms have a small constant branching factor, usually two or three. The task graph in Fig. 1 has two stages, five levels and a constant branching factor of two.

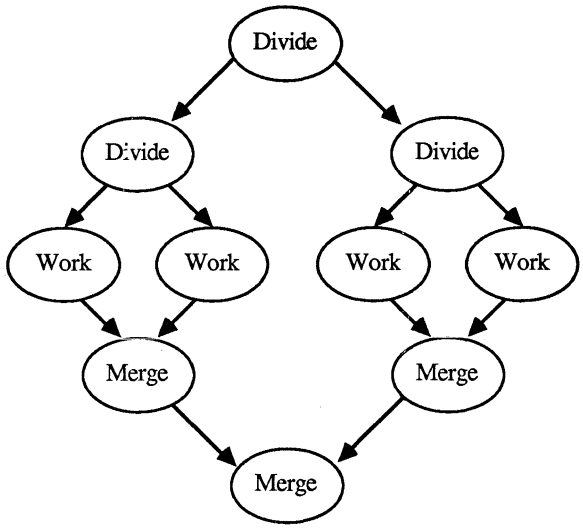


Figure 1: Task graph for a partitioning algorithm

If the execution times of each of the tasks in a partitioning structure are deterministic, the computation of the execution time for the entire graph is be trivial. However, the task execution times in real programs are often non-deterministic because of queueing delays due to contention for resources such as memory or communication channels, and because of data-dependent computation times.

Non-deterministic execution times generally result in synchronization delays where one task has to await the completion of other tasks. Synchronization delays and

† This research was supported by a grant from Texas Instruments Inc., and by a grant from the Office of Naval Research, under Research Contract No. N00014-K-0324.

communication costs are considered to be the most important factors effecting the performance of parallel algorithms [2]. Our goal is to determine the effect of non-deterministic task execution times on the total execution time of the algorithm. We will show that, given information about the nature of task execution times, it is possible to make accurate statements about the mean execution time of a parallel algorithm with the partitioning structure by drawing on results from extreme order statistics.

It is important to be able to determine the effects on performance of synchronization delays in parallel programs for several reasons. First, this gives a lower bound on execution time that is independent of the number of processors, the structure of the interconnection network, and the communication bandwidth. Also, in those cases in which a task that becomes ready to execute always finds an available processor, if the interprocessor communication times are negligible or deterministic, they can be included as part of the task execution times to obtain accurate execution time estimates for the entire program. Finally, we can compare the performances of algorithms based on their synchronization structures which may in turn lead us to better parallel algorithm design.

We make the following assumptions:

1) There are enough processors, *i.e.*, if a task is ready to execute it does not have to wait for a free processor.
2) Communication costs are either negligible or are incorporated into the task execution times.
3) The execution time for each task is a random variable, and either the probability distribution or at least the mean and the variance are known.
4) The execution times for tasks at a particular level are independent of each other and identically distributed (i.i.d.), and the execution time of a task is independent of the execution time of its predecessors.

Assumptions 1 and 2 are necessary to isolate the effect of non-deterministic execution times on synchronization delays. It may not be possible to have complete information about the probability distribution of a task execution time, but the mean and variance can often be experimentally estimated from performance measurements when dealing with real systems. The assumption that tasks at a level are identically distributed is usually justified since all tasks at a particular level in partitioning algorithms do an identical computation albeit on different data. However, more often than not tasks at a particular level are not independent of each other.

The rest of the paper is organized as follows. In the next section we review some previous work that is relevant in this area. We then present five methods for bounding and approximating the mean execution times for partitioning structures. This is followed by an evaluation of the accuracy and applicability of each of the various methods. The evaluation is based on distribution-driven simulations. We also present results for a parallel quicksort algorithm running on the Rice Parallel Processing Testbed (RPPT). Finally we derive an analytical expression for the mean execution time of a parallel mergesort algorithm and compare its predictions with results from the RPPT.

2. Previous Work

Kung [3] in an early work in the area classified parallel algorithms as synchronous or asynchronous algorithms and analyzed examples of both in detail. Briggs and Dubois [4] analyzed the performance of synchronized iterative algorithms on three different machine architectures. Models based on deterministic execution times are discussed by Vrsalovic *et al.* Weide [5] used order statistics to study the anomalous behavior of a specific algorithm structure.

Classification of parallel algorithms based on the structure of task graphs has been developed by Mohan [1]. He used a hybrid simulation tool, PEP, that accepts distribution information about tasks and determines the execution time for a task graph. PEP can be used to model resource contention by means of simple queuing models. Mohan studied the partitioning structure in particular using PEP but did not give any analytical results for the case when the task times are non-deterministic.

Robinson [6] gave an upper bound for the mean execution time of a general task graph under the assumption that execution times for tasks at the same level are i.i.d. His bound is applicable for any task graph provided the means and variances of the task execution times at each level are known. A well known result from order statistics (see pages 57-59 [7]) states that if i.i.d. random variables $X_1, X_2, ..., X_m$ have mean μ and variance σ^2 then

$$E\left[\max_{1\le i\le m} X_i\right] \le \mu + \frac{m-1}{(2m-1)^{1/2}}\sigma \qquad (1)$$

Robinson used this to derive

$$E(T_G) \le \sum_{j=1}^{L}\left[\mu_j + \frac{m_j-1}{(2m_j-1)^{1/2}}\sigma_j\right] \qquad (2)$$

where T_G is a random variable denoting the execution time for the general task graph G, m_j is the total number of tasks at level j, μ_j and σ_j are the mean and standard deviation, respectively, of the execution time of a task at level j, and L is the number of levels.

Eq. (2) can be interpreted as follows. In a general task graph, tasks at a particular level cannot start execution until their respective predecessors in the previous level have completed execution. With the restriction that the tasks at a level start execution only after *all* the tasks in the previous level have completed execution, an upper bound on total execution time for the task graph can be obtained. Loosely speaking, Robinson's upper bound is the mean execution time of a modified task graph where all tasks at a level synchronize at the end of execution.

Eq. (2) is a strict but loose upper bound. It can be improved if the nature of the distribution of execution time for tasks at each level is known. Results for the behavior of extremes for some common distributions such as exponential, normal, and uniform are applicable under these circumstances. We will use Robinson's bound for comparison in evaluating the accuracy of the bounds and approximations to be presented. We will also present and use an extension of Robinson's approach based on an expression analogous to (1)

but dealing with dependent random variables.

3. Analysis

We have developed five methods for bounding and approximating the mean execution time of a partitioning structure. The first uses Robinson's approach for specific distributions. A second bound is based on an expression analogous to (1) for dependent variables. We then provide two approximations for the mean execution time based on the number of paths from the initial task to the final task in the partitioning structure. Our last approximation is an iterative technique that takes advantage of the recursive nature of the partitioning structure. All the methods draw on results from extreme order statistics.

3.1. Bounds for Specific Distributions

Eq. (2) requires that only the mean and variance of task times are known. If information is available about the nature of the task time distributions, it is possible to improve upon (2). If $X_1, X_2, ..., X_m$ are i.i.d. random variables distributed exponentially with parameter λ, then from extreme value theory [8]

$$E\left[\max_{1\le i\le m} X_i\right] \approx \frac{\log m + \gamma}{\lambda} \tag{3}$$

where γ is Euler's constant (0.5772...). All log functions in this paper are natural logarithms. If the variables are uniformly distributed between a and b then

$$E\left[\max_{1\le i\le m} X_i\right] \approx b - \frac{(b-a)}{m} \tag{4}$$

If the variables are normally distributed with parameters μ and σ then

$$E\left[\max_{1\le i\le m} X_i\right] \approx \mu + \sigma\left[(2\log m)^{1/2} - \frac{\log\log m + \log 4\pi}{2(2\log m)^{1/2}} + \frac{\gamma}{(2\log m)^{1/2}}\right] \tag{5}$$

Eqs. (3), (4), or (5) can be used in place of (1) to obtain tighter upper bounds for the mean execution time of a partitioning structure. These will not be strict bounds since (3), (4), and (5) are approximations that become exact for large values of m. However, these can be used in deriving approximate upper bounds that are asymptotically correct.

This approach cannot be used in all cases since some distributions do not have tractable expressions for extreme values. An example is the beta distribution.

3.2. Bound for Dependent Task Times

Frequently, the assumption that the tasks at a level are independent is violated in real programs. For example, in a quicksort algorithm the execution times for the two successor work tasks of a divide task will be negatively correlated since more work for one task would result in less work (a smaller subarray to be sorted) for the other task. Under such circumstances we can use an expression analogous to (1) (see pages 78-79 [7]) that states

$$E\left[\max_{1\le i\le m} X_i\right] \le \mu + (m-1)^{1/2}\sigma \tag{6}$$

to give

$$E(T_G) \le \sum_{j=1}^{L}\left[\mu_j + (m_j-1)^{1/2}\sigma_j\right] \tag{7}$$

This bound gives a higher estimate for the mean execution time than would (2).

3.3. Independent Paths Bound

Both of the above methods, as well as Robinson's bound apply to general task graphs and do not take advantage of the regularity of the partitioning structure. A partitioning structure has a single initial task, a single final task, and b^s different paths from the initial to the final task, where b is the branching factor and s is the number of stages. Under the assumption that the execution times of tasks at a level are i.i.d., the execution times for all paths are identically distributed random variables with mean and variance given by

$$\mu_{path} = \sum_{i=1}^{i=2s+1} E(T_i) \tag{8}$$

$$\sigma^2_{path} = \sum_{i=1}^{i=2s+1} variance(T_i) \tag{9}$$

where T_i is the random variable denoting the execution time of a task at level i. If the paths are independent of each other, the execution time of the partitioning algorithm is the maximum of b^s random variables with the given mean and variance. Using (1) and (3) we get

$$E(T_G) \le \mu_{path} + \frac{k-1}{(2k-1)^{1/2}}\sigma_{path} \tag{10}$$

where $k = b^s$. The assumption that the paths are independent of each other is clearly false since each path shares two or more tasks with every other path. Paths which share a large number of tasks will have highly correlated execution times. Nevertheless, simulation results have shown that this is an improvement over Robinson's bound.

3.4. Independent Paths Normal Approximation

The independent paths bound does not make any assumptions about the nature of distribution of the path execution times. If we assume that the path execution time is normally distributed, the execution time for the entire task graph will be the maximum of b^s *i.i.d* random variables which are normally distributed. Using (5) the expected value of the partitioning algorithm can be approximated by

$$E(T_G) \approx \mu_{path} + \sigma_{path}\left[(2\log k)^{1/2} - \frac{\log\log k + \log 4\pi}{2(2\log k)^{1/2}} + \frac{\gamma}{(2\log k)^{1/2}}\right] \tag{11}$$

where k is the number of different paths.

This approximation will be poor if the number of tasks along a path is small or if the execution time for the path is dominated by a single task. In either case the normal distribution assumption will be invalidated. Nevertheless this approximation is quite accurate as is shown by comparison with simulation results in Section 4.

3.5. Type I Iterative Approximation

The recursive nature of the partitioning task graph suggests an iterative solution. The execution time for an i-stage algorithm can be written as follows:

$$T_i = T_{divide_i} + maximum\ (T_{i-1},...,T_{i-1}) + T_{merge_i}$$

where T_i is a random variable denoting the execution time for an i-stage structure and T_{divide_i} and T_{merge_i} are the execution times for the divide and merge tasks from the appropriate levels. T_0 will be the execution time for a work task. The number of terms in the maximum operation is the branching factor of G. If we further assume that the three terms in the expression are independent random variables we can iteratively sum the means and variances of the three variables provided we have information about the behavior of the maximum of several i.i.d. random variables.

Since determining the execution time for an s-stage algorithm using this iterative approach indirectly involves taking the maximum of b^s random variables we will assume that it will tend asymptotically to an extreme value distribution. If the maximum of several i.i.d random variables tends to a distribution asymptotically it has to be one of three types of distributions, usually referred to as Type I, Type II, or Type III extremal distributions [8]. Extreme values from distributions with an exponential tail behavior tend to the Type I or Gumbel distribution, those from distributions with a polynomial tail behavior tend to the Type II distribution, and those from bounded distributions tend to the Type III distribution. The exponential and normal distributions are examples of distributions whose maximum values tend to the Type I distribution.

The Type I distribution has the properties that the maximum of n i.i.d. variables from a Type I distribution will remain Type I, and further the distribution of the maximum has the same shape as the distribution of the i.i.d. random variables but is shifted to the right. In particular

$$\mu_n = \mu + \frac{\sigma \log n}{\alpha}$$

$$\sigma_n = \sigma$$

$$\alpha^2 = \frac{\pi^2}{6\,\sigma^2}$$

where μ and σ are the mean and variance, respectively, of the Type I distribution, μ_n and σ_n are the mean and variance of the extreme value distribution for n variables and α is a shape parameter. The mean is increased while the variance remains the same. Under the assumption that the execution time for an i-stage algorithm has a Type I distribution and that the distribution will remain Type I even with the addition of the divide and merge time distributions the mean execution time can be computed iteratively as follows:

begin

$E[T_0] = E[T_{work}]$
$\sigma_0 = \sigma_{work}$

for i = 1 until s do

begin

$$E[T_i] = E[T_{divide_i}] + E[T_{i-1}] + \frac{\sqrt{6}\log b\ \sigma_{i-1}}{\pi} + E[T_{merge_i}]$$

$$\sigma_i^2 = \sigma_{divide_i}^2 + \sigma_{i-1}^2 + \sigma_{merge_i}^2$$

end

end

The addition of variances follows from the assumption that task times are independent of their predecessor tasks.

The Type I Iterative method makes use of the fact that there is a simple relation between the means and variances of a Type I distribution and its extreme value distribution. Such convenient relations are not available for Type II and Type III distributions. In particular the variance of the extreme value of a Type II distribution increases as the number of terms in the maximum operation is increased.

4. Performance Comparison

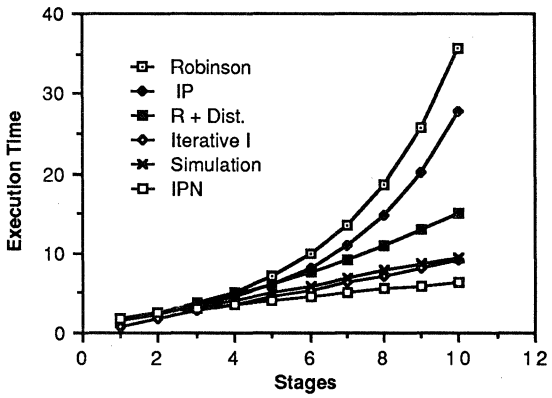
We evaluated the methods presented above by comparison with simulation results and with Robinson's bound. We first present distribution-driven simulation results for the exponential, uniform, and beta distributions. Results predicted by each of the methods are compared against simulation values to determine the accuracy of the methods. We then present a comparison of three of these methods with simulation results for a quicksort algorithm running on the Rice Parallel Processing Testbed (RPPT). The quicksort algorithm violates the independence assumptions on which the methods are based. Nevertheless, as will be seen, the methods predict the mean execution times with reasonable accuracy.

4.1. Stochastic, Independent Task Execution Times

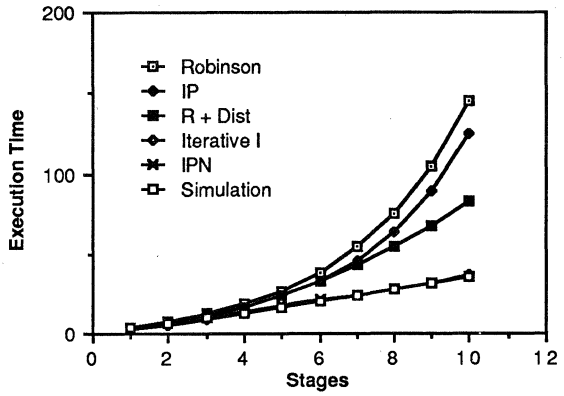
Results from simulation and analysis were obtained for three different distributions, namely, the exponential, uniform, and beta distributions. In each case the three task types, divide, work, and merge, were assumed to have the same type of distribution and the parameters were chosen such that the mean executions times would differ by an order of magnitude.

Fig. 2(a) and 2(b) are graphs of the mean execution time of as a function of the number of stages for the exponential case. Results obtained from Robinson's bound, the independent paths (IP) method, the independent paths normal approximation (IPN), the Type I iterative approximation, and the simulation are shown. The exponential distribution has a well known extreme value behavior, and the curve labeled R + Dist. in the graph is for results predicted by Robinson's bound when modified by extreme value formula for the exponential distribution. Fig. 2(a) gives results for the case where the divide and merge tasks are exponentially distributed with $\lambda = 10$ and the work task is exponential with $\lambda = 1$. Fig. 2(b) gives results for the case where all three task types are exponentially distributed with $\lambda = 1$.

The following observations can be made from the results. Most methods predict the mean execution time

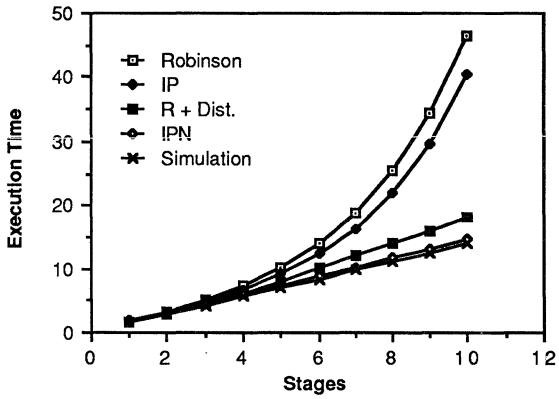


a) Divide = exp(10), work = exp(1), merge = exp(10)

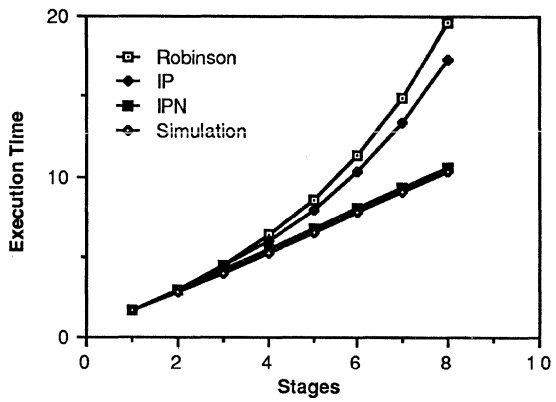


b) Divide = exp(1), work = exp(1), merge = exp(1)

Figure 2: Comparison of bounds and approximations for exponential distribution



a) Divide = unif(0,1), work = unif(0,1), merge = unif(0,1)



b) Divide = beta(0,1), work = beta(0,1), merge = beta(0,1)

Figure 3: Comparison of bounds and approximations for uniform and beta distributions

accurately for up to a four stage structure. Beyond four stages, Robinson's bound diverges rapidly from simulation values. The IP method is better than Robinson's bound but it also diverges. Additional information about the distribution improves Robinson's bound (R + Dist.) considerably. The independent paths normal approximation (IPN) is accurate to within 30 percent in Fig. 2(a) and is accurate to within 5 percent in Fig. 2(b). This is due to the fact the choice of parameters in Fig. 2(a) results in a poor normal approximation to the path execution time since the path time is dominated by a single task, the work task. The Type I iterative method is the most accurate of all and is within 5 percent of the simulation results in both figures.

Figs. 3(a) and 3(b) give similar results for the uniform and beta distributions, respectively. In both cases parameters for all task times were chosen to be identical. The Type I iterative method is not applicable for either distribution. No results are given for R+Dist. in Fig. 3(b) since formulae for extreme value behavior of the beta distribution are unavailable. Both these graphs reinforce our earlier observations. The IPN method proves to be highly accurate in both cases, predicting the simulation values to within 5 percent.

If a strict (asymptotic) upper bound is required and the task time distributions are known, the method of choice is Robinson's method as modified by distribution information. However the extreme value behavior is not analytically available for all distributions and one has to revert to Robinson's original bound in these cases. The IPN method requires only the means and variances of task execution times and provides a good approximation even when the normal distribution assumption for path execution times is not satisfied. The Type I iterative is highly accurate but can be justified only for distributions whose extreme values converge to the Gumbel or Type I asymptotic distribution.

4.2. Program Driven Task Execution Times

Program driven results were obtained for the quicksort algorithm running on the Rice Parallel Processing Testbed (RPPT) [9]. The RPPT is a software simulation tool that facilitates the performance evaluation of parallel programs on parallel architectures. Parallel programs are analyzed for

timing information at the assembly language level. The program then drives an architectural model to provide accurate statistics about resource usage and execution times.

The architecture used for the RPPT simulation was a single bus, shared memory multiprocessor with enough processors so that ready tasks did not have to wait. During the simulation, statistics on individual task execution times were collected for a task at each level. Communication times, which were negligible, were included in the task execution times.

The partitioning quicksort algorithm works as follows. The divide tasks partition the input array using a median element and start further divide or work tasks depending on the level. Each work task sorts its input array using quicksort. The merge tasks are trivial and simply terminate after informing the next level merge tasks. The number of stages of the algorithm was varied from 1 to 8 and the algorithm was run with random integer input.

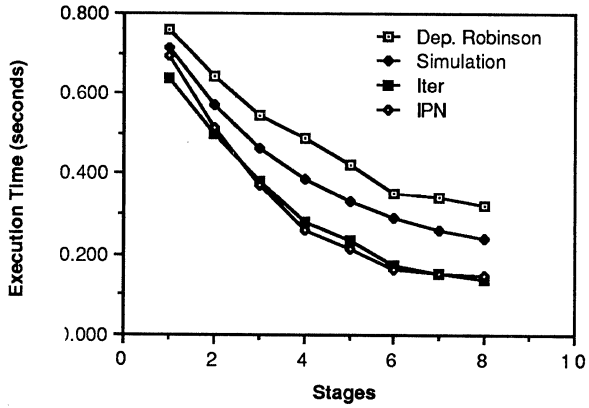


Figure 4: Quicksort of 8192 integers

Fig. 4 shows the execution times for the quicksort algorithm along with the predictions by Robinson's method, the IPN method, and the iterative method. The tasks at a particular level are not independent and tasks close together are highly correlated. Thus the independence assumptions on which all the methods are based fail. However, it is possible to correct for dependence among tasks at a level for the Robinson's method using eq. (6). The plot in Fig. 4 reflects this. Results for the iterative method are presented even though the distribution types are not known and the independence assumptions are violated. Nevertheless, the iterative method gives results that have accuracy comparable to the IPN method.

The results show that Robinson's method when corrected for dependencies still bounds the simulation values from above. The IPN and iterative methods are close together and consistently underestimate the mean execution time. This is to be expected since the quick sort algorithm violates the independence assumptions.

5. Analysis of a Parallel Mergesort Algorithm

The IPN approximation is a useful method for numerically estimating program execution times. It can also be used as a basis for finding an analytic expression for the expected execution times of parallel programs with a partitioning structure. We now derive and expression for the mean execution time of a mergesort algorithm using the independent paths normal approximation method. Predictions from the analysis are compared against simulation results from a mergesort algorithm running on RPPT.

The mergesort algorithm is an example of a partitioning algorithm which has no divide tasks. Each work task sorts a subarray of size $\frac{N}{k}$ where N is the number of elements to be sorted, and k is the number of work tasks. Each merge task accepts two sorted subarrays of equal size from its predecessor tasks, merges them into a single sorted array, and passes it to its successor. Task execution times at a particular level are i.i.d. since all tasks at a level do the same type of work but on different sections of data.

Let the number of work tasks be k, a power of two, and let N be the number of elements to be sorted. Assuming that the time to mergesort an array of size N has mean $a_s N\log(N)$ and standard deviation $b_s N^{1/2}$, and the time to merge two subarrays of size N has mean $2a_m N$ and standard deviation b_m [10] we get:

$$\mu_{path} = a_s\frac{N}{k}\log(\frac{N}{k}) + a_m N(1-\frac{1}{k})$$

and

$$\sigma^2_{path} = b_s^2\frac{N}{k} + b_m^2\log(k)$$

where μ_{path} and σ^2_{path} are the mean and variance respectively of the execution time for a path from a work task to the final merge task. Using the independent paths approximation we obtain

$$E[mergesort] = \mu_{path} + \sigma_{path}^{1/2}\left[(2\log k)^{1/2} - \frac{\log\log k + \log 4\pi}{2(2\log k)^{1/2}} + \frac{\gamma}{(2\log k)^{1/2}}\right]$$

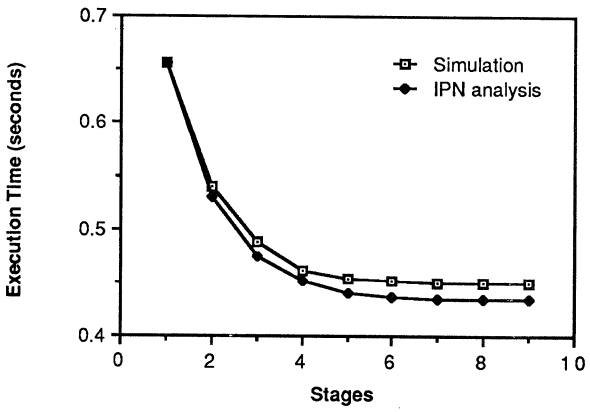


Figure 5: Mergesort of 8192 integers

Fig. 5 shows the results from the IPN analysis and from an RPPT simulation of the mergesort algorithm. The coefficients a_m, b_m, a_s, and b_s, were estimated from task time measurements made during the simulation. The analysis and simulation agree to within five percent.

6. Conclusion

Results from extreme value theory are applicable in predicting the execution times of certain parallel program structures. Two of the methods we have presented, which are based on Robinson's approach, can be used to bound the mean execution time of a general task graph. The three other methods, the independent paths approximation, the independent paths normal approximation, and the iterative approximation, can be used to approximate the mean execution of a parallel program with a partitioning structure. The IP approximation is empirically shown to be slightly better than Robinson's bound and of a general applicability since it does not require complete information about the probability distribution of task execution times. The IPN approximation gives excellent results and is again of a general applicability as has been shown by our analysis of a parallel mergesort algorithm. The iterative method is very accurate, but is based on task time distributions whose extreme values tend to the Type I asymptotic distribution. However, simulation results show that the iterative method gives good results even when the distributions are unknown and independence assumptions are violated.

Since bounded distributions are common in practice, an iterative method for the Type III asymptotic distribution would be of considerable interest and we are currently working on such a method. We are also studying the problem of predicting the execution times of other parallel program structures such as the multiphase algorithms and general pipeline algorithms. Another area of future study in partitioning algorithms is to consider the effects of non-negligible communication costs and resource contention, both for processors and communication bandwidth.

References

1. J. Mohan, "Performance of Parallel Programs: Model and Analyses," Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University (July 1984).
2. B. Lint and T. Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," *IEEE Trans. Software Engineering* **SE-7**(2) pp. 174-188 (March 1981).
3. H. T. Kung, "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors," pp. 153-200 in *Algorithms and Complexity: New Directions and Recent Results*, ed. J. F. Traub,Academic Press, New York (1976).
4. M. Dubois and F. A. Briggs, "Performance of Synchronized Iterative Processes in Multiprocessor Systems," *IEEE Trans. Software Engineering* **SE-8**(4) pp. 419-431 (July 1982).
5. B. W. Weide, "Analytical Models to Explain the Anomalous Behavior of Parallel Programs," *Proc. 1981 Int. Conf. on Parallel Processing*, pp. 183-187 (August 1981).
6. J. T. Robinson, "Some Analysis Techniques for Asynchronous Multiprocessor Algorithms," *IEEE Trans. Software Engineering* **SE-5** pp. 24-31 (January 1979).
7. H. A. David, *Order Statistics,* John Wiley & Sons Inc., New York (1981).
8. A. H-S. Ang and W. H. Tang, *Probability Concepts in Engineering Planning and Design Volume II,* Rainbow Bridge Book Co. (1984).
9. R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed," *1988 ACM Sigmetrics Conference*, (May 1988). to appear
10. D.E. Knuth, *The Art of Computer Programming Vol.3,* Addison-Wesley, Reading, MA (1973).

A RANDOMIZED PARALLEL BRANCH-AND-BOUND ALGORITHM

Virendra K. Janakiram
Dharma P. Agrawal
Ravi Mehrotra

Computer Systems Lab, ECE Department
North Carolina State University
Raleigh, NC 27695-7911

Abstract — A new technique for the parallel execution of branch-and-bound algorithms using "randomization" is proposed. The algorithm requires relatively little inter-processor communication, while achieving good speedups over the uniprocessor execution times; in precisely those cases where the problem size becomes very large, randomization is found to be extremely successful in achieving very good speedups. A probabilistic model has been devised to explore the effectiveness of this technique, and to estimate the expected speedups that could be obtained. The model has been validated by extensive simulation work on a multiprocessor simulator. Besides being very simple to implement, the technique also ensures high reliability, flexibility, and fault-tolerance.

I. INTRODUCTION

Branch and bound has often been the algorithm of choice for the solution of combinatorial search problems. Some of the classical combinatorial problems involving discrete optimization include the Travelling Salesman problem, the Knapsack problem, Job Scheduling, and Integer Linear Programming. These problems, which occur commonly in different forms in diverse areas belong to the class *NP-hard* [4]. We propose a "randomized" parallel branch-and-bound algorithm, which, by introducing an element of randomness into the conventional branch-and-bound algorithm, makes parallelization simple, yet effective, while, at the same time, ensuring that interprocessor communication does not get out of hand. Additionally, a system employing this randomization technique would have good fault tolerance and reliability, in the event of processor failures, and can be easily expanded by adding more processors.

In the next section, we describe the randomized algorithm *vis-a-vis* the conventional one. Section III establishes a simple model that is used effectively in Section IV to estimate the speedup performance of the randomized algorithm. Section V then presents some simulation results that were obtained from a multiprocessor simulator. Finally, conclusions and suggestions for future work is given in Section VI.

II. THE RANDOMIZED BRANCH-AND-BOUND ALGORITHM

The branch-and-bound algorithm has been exhaustively described and analyzed in the literature (See for example, [4]). A rigorous analysis of the branch-and-bound procedures in [8] shows that the expected time to solve some problems is a polynomial function of time. Stone and Sipala [9] present related results. Wah and Yu have given a stochastic model for the branch-and-bound algorithm in [10].

The branch-and-bound process may be visualized as searching a *branch-and-bound tree*, with each node representing a subproblem, each leaf representing either a feasible solution, or a sub-problem that will not yield a possible solution. Each node will have a value associated with it which is simply the value of the *bounding function*, g. From this point on, we will assume, without loss of generality, that a solution with the minimum cost is being sought. Further, we assume that the selection rule results in the search of the branch-and-bound tree in a *depth-first* fashion. Our method seems to be applicable to best-first searches also, as will be apparent, but we are yet to study its performance.

The success of the branch-and-bound algorithm stems primarily from the fact that the bounding function, g, can be used to remove from consideration, fairly early in the proceedings, subproblems that will not yield solutions better than the one at hand. As time progresses, successively better solutions will have been found, and with this "experience" gained, the branch-and-bound algorithm is able to eliminate progressively larger sub-trees. Thus, the branch-and-bound algorithm dynamically prunes the search tree. The branch-and-bound algorithm just described is called *lfbb*, (*l*eft *f*irst *b*ranch and *b*ound).

A parallel version of this conventional branch-and-bound algorithm has been considered by Wah and Ma [11]. MANIP is a special-purpose machine proposed by them solely for the solution of branch-and-bound problems. The selection rule used here is best-first, but depth-first search is used when secondary memory runs out. They have indicated that simulation studies have shown a speedup of k, for k processors. However, the speedup is calculated only with respect to the number of iterations; important factors such as (secondary) memory access times which could have a serious impact on the performance have been neglected.

El-Desouki and Huen [3] have considered a scheme somewhat similar to the one we are about to describe. Here, the workload of each processor is deterministically apportioned at the outset. Suppose that (as is very likely) one processor finishes examining its portion of the solution tree before the others. It then enters into a complicated and lengthy dialogue with each of the other processors to determine which particular portion of the solution tree is most suitable for exploration. Not only is there extra communication cost involved here, but also an unknown amount of extra computation overhead incurred by all the processors. Another serious drawback of this scheme is apparent when considering what would happen should a processor fail: In such a case, a portion of the of the tree will remain unexamined, thereby producing erroneous results. The method to be described avoids these problems.

Consider the conventional branch-and-bound algorithm, *lfbb*. Instead of choosing the next node to be evaluated in left-first fashion, we will "randomize" the search by making a random choice for the next node from among the unexamined children. The cost of the best feasible solution available currently is made available globally to all processors. Other details of the algorithm remain unchanged. This algorithm is called *rsbb* (for *r*andom *s*earch *b*ranch and *b*ound) and is given below:

This work has been supported by the U.S. Army Research Office under contract no. DAAG 29-85-K-0236

```
minfeas : real { Cost of the minimum feasible solution }
function rsbb (instance)
    cost ← g(instance) { bounding function }
    if (cost < minfeas) then
        if (not feasible(instance)) then
            repeat
                nextinstance ← { randomly chosen child of instance }
                rsbb(nextinstance)
                quit ← { all children of instance have been examined
                }
            until (quit)
        else
            minfeas ← cost
```

It can be shown that the expected solution time on a uni-processor using *rsbb* is the same as that of *lfbb* for a very general class of problems. However, the advantage of *rsbb* is that, under parallel execution, the decision as to the next node to evaluate can be made locally, without global information. If we use k processors, for example, each processor will, in the main, examine a different portion of the tree. Of course there is a finite probability that replication of work will occur, but by allowing a very small amount of simple and terse communication between processors, this probability can be kept relatively small, and good speedups can be achieved. Another advantage of the decentralized operation is that fault-tolerance and easy expandability are inherent in such a system.

Randomization techniques have been used as effective heuristics to obtain sub-optimal solutions for many combinatorial problems [7]. Stochastic annealing is one example. The parallelization of these kinds of searches would be an interesting research area. We, however, consider here search algorithms for *optimal* solutions. We use randomization not as a *search* heuristic, but as a *scheduling* technique.

Janakiram *et al.* have described a randomized parallel version of the backtracking algorithm in [5], where a similar technique has been employed. Three classes of backtracking problem-types have been identified, and for each class the expected speedup has been determined. The expected speedup has been shown to vary from $(k+1)/2$, to k (k is the number of processors), depending on the problem-type.

In the next section we discuss the randomized branch-and-bound algorithm in more detail, and attempt to derive a model that is used estimate the speedups obtainable by using parallel *rsbb*.

III. MODEL DESCRIPTION

We will first establish that the expected solution time for a branch-and-bound problem solved using *lfbb* is the same as when using *rsbb*.

A probabilistic model for the kind of solution trees generated by the branch-and-bound algorithm has been given by Smith [8]. A random branch-and-bound tree may be generated as follows:

(i) Let a root node exist, which is unsprouted (level=0).

(ii) Each unsprouted node, n, at level i, is sprouted as follows: Let n have S children, where S is a random integer whose probability mass function (p.m.f.) is given by $p_S(t;i)$, $(p_S(t;i=0)>0)$. Each node will be assigned a cost that is the sum of the node costs of its ancestors, together with a random number called Q, whose distribution is given by $p_Q(t)$. $(P_Q(t=0)=0)$.

(iii) Repeat (ii) until there are no unsprouted nodes.

The above procedure is slightly more general than that given by Smith. The node cost obtained in this way is the simulated equivalent of the value of the bounding function g, described in the last section. The cost of a particular node is at least equal to that of its parent. The increase in node cost (*viz.*, the r.v. Q) over its parent is termed the *incremental node cost*, and is a non-negative number.

Consider a random tree being searched by a processor using *rsbb*. Suppose that the *left* sub-tree is selected first, and it happens that the best solution lies in this subtree. Let $I(1)$ be the number of nodes examined by *rsbb* in this instance. Now consider the case when two processors are deployed to solve the same problem. It is possible that both processors choose the *right* sub-tree first, and the two processor system solves the problem in $I(2)$ steps, where $I(2) \geq I(1)$. This is the kind of anomalous behavior observed by Lai and Sahni [6], for the best-first branch-and-bound algorithm. For parallel *rsbb*, however, we prove the following:

Theorem 1: Let the expected speedup obtained by parallel *rsbb*, using k processors be S_k. Then $S_{k_1} \geq S_{k_2} \geq 1$, $(k_1 \geq k_2 \geq 1)$, even if there is no inter-communication between processors. □

Theorem 2: For all the trees generated by the above procedure, the probability distribution of the solution times using depth-first search, is independent of the order in which successive nodes are chosen. □

It is thus clear that *lfbb* and *rsbb* take the same time to solve a problem, on average. The next step is to estimate the expected solution time of *lfbb*. This has been determined by Smith [8], and by Wah and Yu [12]. The latter paper has given a conceptually satisfying model of the branch-and-bound process which is also very accurate. The extension of this model to the multiprocessor case, however, does not seem to be possible. We have, instead, devised a simpler model, which is adequate for our purpose, *viz.*, to estimate the speedup on a parallel system.

First, some assumptions are made regarding the structure of the branch-and-bound tree which are similar to those made by Wah and Yu:

(i) We assume that the solution tree is finite and of constant degree. As trees of degree two are the most commonly occurring ones, we have confined ourselves to binary trees; but this is not a restriction on the analysis.

(ii) We assume the tree is full, and of depth D.

(iii) The incremental node cost is assumed to be *exponentially* distributed. This assumption is made not only because it makes the mathematics tractable, but because this has been the distribution that has been observed in practice. Wah and Yu have shown that it occurs in the Knapsack problem [12], and in integer programming [10], while Smith [8] reports that for the Travelling Salesman problem, the distribution is geometric, which is the discrete analogue of the exponential distribution.

Each node of the branch-and-bound tree is identified by the pair (i,j), where i is the level and j the serial number of the node, beginning from the left with 0. As *lfbb* proceeds, it will follow the path (0,0),(1,0), . . . ,(D,0), where D is the depth of the tree. Node (D,0) is a feasible solution. Now *lfbb* will backtrack and look for other feasible solutions, each time updating the variable *minfeas*, which contains the cost of the best possible solution obtained thus far. When *lfbb* encounters a node whose cost hap-

pens to be greater† than, or equal to, *minfeas*, it will simply discard the children of this node.

Let $N_{i,j}$ be the number of nodes examined by *lfbb* in a tree rooted at (i,j). We want to find $N_{0,0}$, the number of nodes examined in the entire tree. $N_{i,j}$ can be expressed as a recurrence:

$$N_{i,j} = 1 + \rho_{i,j}(N_{i+1,2j} + N_{i+1,2j+1}) ;$$
$$\rho_{0,0} = 1 ;$$
$$N_{D,j} = 1 . \qquad ..(1)$$

where $\rho_{i,j}$ is the probability that the node cost of (i,j) is less than *minfeas*.

Eq. (1) states that *lfbb must* examine node (i,j), and then with probability $\rho_{i,j}$, examine the subtrees rooted at its children. The root (0,0) is examined with probability 1 ($\rho_{0,0}=1$) and a leaf node is terminal ($N_{D,j}=1$). The computation of $\rho_{i,j}$ is now described.

Consider the scenario in Fig. 1. Here, *lfbb* has encountered the l solutions in subtrees T_2, T_3, and T_4, and is now examining node a, deciding whether to select the subtree rooted at a for searching or to backtrack. The best feasible solution, will be the one with minimum cost among those obtained so far. Each solution is the sum of D (not necessarily independent) random variables, each random variable being the incremental node cost.

An exact analysis would proceed as follows: First, the probability distribution of the cost of the best solution in a given subtree must be found. This may be represented as a recurrence relation:

$$F_{LC(d)}(t) = 1 - [1-\int_0^t f_Q * f_{LC(d-1)}(x)dx]^2 \qquad ..(2)$$

where

$F_{LC(d)}(t)$ is the probability distribution of the best solution cost in a subtree of depth d.

$f_Q(t)$ is the probability density of the incremental node cost and,

$*$ stands for convolution.

The convolution in Eq. (2) represents the probability density function of the sum of the incremental cost of a node and the random variable that is the minimum cost solution in the sub-tree rooted at this node. The minimum solution cost in a tree is then the minimum of two random variables (for the right and left subtrees), each of which is distributed as the sum just described.

Next, using the tree in Fig. 1 as an example, the probability of the following two (in this case) events must be calculated (the q's denote the incremental node costs):

(i) {The minimum cost solution in tree $T_1 + q'_{l-1}$} $\leq$ $\{q_{l-1}+q_l\}$.

(ii) {Minimum cost solution in $T_4 + q'_l\} \leq \{q_l\}$.

Let p_c be the probability of both events (i) and (ii) occurring, i.e., the *cut-off* probability at node a. Then $\rho_{i,j}$ for a is simply $(1-p_c)$. Unfortunately, the analysis, if pursued as above, quickly becomes intractable. The treatment we follow makes certain simplifying assumptions. The results thus obtained are approximate, but are of sufficient accuracy for limited purpose for which they will be used.

†Recall that we are dealing with *minimization* problems.

Suppose that l solutions have been found by the time node a, at level m is encountered. (See Fig. 1). Now the cost of a particluar solution is the sum of D incremental costs. This sum may be considered in two parts: (a) The sum of the incremental cost of the node at level $m-1$, and the cost of its parent, and (b) the sum of the remaining incremental costs. Although, in general, node a and all the solution nodes before node a is encountered may not have all their ancestors in common, we assume that this is indeed the case, and thus ignore the contribution of these ancestors to the variation in node costs amongst the solution nodes, and also node a. The contribution to the solution cost under part (b) above will be due to the nodes at and below level $(m-1)$. Many of these ancestors are common to the l solution nodes. However, we will assume that these ancestors are not common, i.e., the events that lead to the generation of the solutions are *independent*. The approximations made in computing the sums (a) and (b) are "opposite", in a sense. In (a), we assume complete *dependence* when independent events exist, while in (b) we assume *independence* when dependence exists. Hence, the errors caused by these approximations could be expected to compensate each other, to some extent.

A chain of nodes, then, from level $m-1$ to level D will have on it some n, $(n=d-m+1)$, nodes. The contribution of these n nodes to the solution cost will be the sum of n random variables, each of which is the incremental cost of each node. This sum is termed the *length* of the chain. The situation is depicted in Fig. 2. There are l such chains, corresponding to the l solutions. By the assumptions made earlier, the length of each chain is an independent random variable. If the minimum length of these l chains is less than, or equal to, the incremental cost of node a, then we conclude that there is no solution in the subtree rooted at a that is any better than the best one found so far. Further exploration of this subtree is useless, and hence node a will be cut off.

Let p_c be the probability of cutoff, and Q a random number that is the incremental node cost. Then,

$$dp_c = Pr\{\min(\text{cost of } l \text{ chains}) \leq t\} \times Pr\{t \leq Q \leq t+dt\}$$
$$= \left\{1-[1-F_{Q,n}(t)]^l\right\} f_Q(t)\, dt \qquad ..(3)$$

$F_{Q,n}(t)$ is the probability distribution of length of a chain. A chain is made up of n nodes, each of which contributes a random amount Q to the length. Q has a density function $f_Q(t)$. So,

$$F_{Q,n}(t) = \int_0^t f_Q^{(n)}(x)\, dx \qquad ..(4)$$

where the power is in the sense of convolution. The cut-off probability can then be written:

$$p_c(n,l) = \int_0^\infty \{1-[1-\int_0^t f_Q^{(n)}(x)\, dx]^l\}\, f_Q(t)\, dt \qquad ..(5)$$

Because of the reasons adduced earlier, we set $f_Q(t)$ to be the exponential distribution:

$$f_Q(t) = \lambda e^{-\lambda t}$$

Then (5) becomes:

$$p_c(n,l) = 1 - \int_0^\infty [\sum_{k=0}^n \frac{t^k}{k!}]^l\, e^{-(l+1)t}\, dt \qquad ..(6)$$

Although the integral in (6) can be obtained for small values of l and n by applying the multinomial expansion, this technique is impractical for large l. (l may take values in the order of 10^5). An alternative is to use the Guass-Laguerre quadrature. But for large l, this method gives rise to significant inaccuracies. An asymptotic expression may, however, be derived for large l, using the Laplace method [2]:

$$p_c(n,l) \sim 1 - \frac{\mu}{n+1} \sum_{i=0}^{\infty} \frac{(-1)^i \mu^i}{i!} \cdot \frac{\Gamma\left(\frac{i+1}{n+1}\right)}{l^{\frac{i+1}{n+1}}} \qquad ..(7)$$

where

$$\mu = [(n+1)!]^{\frac{1}{n+1}}$$

The convergence of the infinite sum in (7) is quite rapid: when $l \approx 10n$, about four terms are required for a 1% error. For larger l, an even smaller number of terms is sufficient.

The quantity $\rho_{i,j}$ is related to $p_c(n,l)$ by:

$$\rho_{i,j} = 1 - p_c(d-i+1, j2^{d-i}) \qquad ..(8)$$

At this point all the elements are in place to solve the original problem given in (1), using (8), (7), and (6). The expected number of nodes examined by the branch-and-bound algorithm, for trees of various depths can be calculated. The results are shown in Table 1, which compares the model with a simulation using 1000-2000 trials on random trees. For trees of depth 10 and below, the discrepancy between the model and the simulation is about 5%, increasing to about 14% for a depth of 12.

Suppose there are two processors working on two different subtrees, each exchanging information as to the cost of the best solution. Then the decision whether to cut off at some node will have the benefit of both these searches: twice the number of independent chains have been generated. The effect of this mutual assistance can be represented in our model by simply replacing l by kl in (6), where k is the number of processors. The result is the speedup *given* that each processor has chosen a different subtree to work upon. In the next section we will derive an expression for the unconditional expected speedup for a k-processor system with global memory.

IV. SPEEDUP CALCULATIONS

We now proceed to use the results obtained from our model to estimate the expected speedups. In the derivation we have neglected to account for the queueing and communication delays. In the next section, we observe that these are quite small.

Suppose some k processors are deployed to search the tree using *rsbb*. At each level each processor makes a random decision regarding the next child to examine. Given a sufficiently large tree, there is but an infinitesimal probability that two processors follow the exact same path while searching the tree. Of more concern is the likelihood that a processor will take a path already trodden earlier by another processor, resulting in replication of work. To avoid this occurring, a global list is maintained that keeps the status of the subtrees at each level. It is easy to show by calculating the probability of the occupancy numbers, that the level to which this list must be maintained does not need to be large. In the next section it will be shown that a list maintained for the first five levels is adequate for up to 10 processors.

We define the following terms. Let S_k be the expected speed up using k processors; $T(j,n)$ be the expected time taken by j processors to search the left subtree, which has n nodes; and $T'(j,n)$ be the expected time to search the right subtree, *after* the left has been searched. The time to examine one node is taken as 1 unit. Define $\gamma \equiv T'(j,n)/T(j,n)$. It is clear that

$$\begin{aligned} T(j,2n) &= T(j,n) + T'(j,n) + 1 \\ &\approx T(j,n) + T'(j,n) \quad \text{(for large trees)} \end{aligned}$$

So, ignoring the error due this approximation,

$$\frac{T(j,2n)}{T(j,n)} = (1+\gamma) \qquad ..(9)$$

Eq. (9) expresses the ratio of the time that j processors take to search a tree of $2n$ nodes to the time taken for a tree of n nodes. Evidently, this ratio is independent of j. Eq. (9) may thus be used to estimate γ from Table 1, which gives an average value of $\gamma \approx 0.4$. Extending (9),

$$\frac{T(j,2^{i+1})}{T(j,1)} = (1+\gamma)^{i+1} \qquad ..(10)$$

Now, $2^{D+1} \approx N$ is the total number of nodes. Let $2^{i+1} \approx n$ be the size of the subtree searched at time t. Then,

$$T(j,n) = T(j,N)(1+\gamma)^{\log_2 n - \log_2 N} \qquad ..(11)$$

which gives:

$$\beta(t) \equiv \frac{n}{N} = \left(\frac{t}{T_N}\right)^{\alpha} \qquad ..(12)$$

where:

$\beta(t) \equiv n/N$ = fraction of nodes examined at time t,
T_N = time to search the entire tree of N nodes, and,
$\alpha = 1/\log_2(1+\gamma)$.

Suppose there are some k processors working on one part of the tree, with l processors working elsewhere in another subtree, assisting these k processors. Let $S_{k,l}$ be the expected speedup obtained in the first subtree. ($S_k \equiv S_{k,0}$). At the node rooted at this sub-tree, the k processors will randomly choose their next sub-tree to examine. Of the k, some k_1 will choose the left subtree and k_2, the right. Let p_i be the probability of this partition, $\{k_1,k_2\}$. Without loss of generality, we can assume that $k_1 \geq k_2$. Let σ_i be the speedup, given this partition.

$$S_{k,l} = \sum_i p_i \, \sigma_i \qquad ..(13)$$

where the sum is over all the possible partitions. Consider the partition $\{k_1,k_2\}$; $k_1 \geq k_2$, $k_2 \neq 0$. The k_1 processors — now assisted by $l+k_2$ processors — will finish searching the left subtree consisting of $N/2$ nodes in an expected time $t_1 = T(k_1,N/2)$.

$$\begin{aligned} t_1 = T(k_1,N/2) &= \frac{T(1,N/2)}{S_{k_1,l+k_2}} \\ &= \frac{1}{1+\gamma} \cdot \frac{T(1,N)}{S_{k_1,l+k_2}} \qquad ..(14) \end{aligned}$$

In time t_1, the k_2 processors have finished some fraction $\beta(t_1)$ of their work. From (11),

$$\beta(t_1) = \left[\frac{t_1}{T(k_2,N/2}\right]^{\alpha} \qquad ..(15)$$

Now, $T(k_2,N/2)$ can be written as:

$$T(k_2,N/2) = \frac{T(1,N/2)}{S_{k_2,l+k_1}}$$

$$= \frac{T(1,N)}{1+\gamma} \cdot \frac{1}{S_{k_2,l+k_1}} \quad ..(16)$$

whence,

$$\beta(t_1) = \left[\frac{S_{k_2,l+k_1}}{S_{k_1,l+k_2}} \right]^{\alpha} \quad ..(17)$$

After time t_1, all the processors will combine to work upon the unfinished portion of the tree. Let t_2 be the expected time to finish the remaining work. By properly juxtaposing the β-curves for k_2 and k_1+k_2 processors, t_2 can be calculated:

$$t_2 = T(k,N/2) - T(k,N/2)\beta^{\frac{1}{\alpha}}(t_1)$$

$$= \frac{T(1,N)}{S_{k,l}} \cdot \frac{1}{1+\gamma} \cdot [1-\beta^{\frac{1}{\alpha}}(t_1)] \quad ..(18)$$

The total time to search the tree, $T(k,N)$ is t_1+t_2 and is given by:

$$T(k,N) = \frac{T(1,N)}{1+\gamma} \left(\frac{1}{S_{k_1,l+k_2}} + \frac{1-\beta^{\frac{1}{\alpha}}(t_1)}{S_{k,l}} \right)$$

Transposing terms, the speedup σ_i, for a given partition can then be written:

$$\sigma_i = (1+\gamma) \left(\frac{1}{S_{k_1,l+k_2}} + \frac{1-\beta^{\frac{1}{\alpha}}(t_1)}{S_{k,l}} \right)^{-1} \quad ..(19)$$

Now, (13) can be expanded as:

$$S_{k,l} = p_0 \, S_{k,l} + \sum_i p_i \, \sigma_i \quad ..(20)$$

where the summation is over all partitions.

Eq. (20) expresses a recurrence relation for the expected speedup. The calculation of the probability p_i, of a particular partition $\{k_1,k_2\}$ is a straightforward matter of considering the probability of the occupancy numbers:

$$p_i = \begin{cases} \frac{k!}{k_1!k_2!} \cdot \frac{1}{2^{k-1}} ; & k_1 \neq k_2 \\ \frac{k!}{[(k/2)!]^2} \cdot \frac{1}{2^k} ; & k_1 = k_2 \end{cases} \quad ..(21)$$

The terminating condition for the recurrence, *viz.*, the set $S_{1,l}$ can be obtained from the model as explained in the previous section. Table 2 gives the speedup calculated in this fashion, and Fig. 3 shows a plot of these values. Setting $S_{1,l}=1$, (all l), gives the case when only local versions of the best solution are kept by each processor.

In the next section, we will describe the simulations and present results obtained.

V. SIMULATION RESULTS

The performance of the randomized branch-and-bound algorithm was obtained on a multiprocessor simulator MPSIM [1], (implementing the PRAM-CREW model), running under ULTRIX on a MICROVAX. The randomized algorithm *rsbb* is implemented in C. The software can be divided into three parts: (i) The multiprocessor simulator, (ii) the skeletal algorithm *rsbb*, and (iii) problem specific procedures and data-structures. Mutual exclusion is enforced by the use of a monitor.

The problem that was chosen was the 30-element 0/1 knapsack problem [4]. The knapsack problem was chosen because of its relative ease of formulation. In order to fully explore the possible range of the problem space, we have used a suggestion of Horowitz and Sahni [4]. Six sets of 50 problems (300 in all) were used, with the problems in each set being randomly generated. The sets are described in Table 3. To determine the effect of keeping a global list of the nodes visited, the same problems were solved using three different sizes of lists: (i) For the first 5 levels (31 nodes in all), (ii) The first 4 levels (15 nodes), and (iii) No global list. The results are shown in Table 4.

An important observation about the behavior of the algorithm is that large speedups occur in precisely those problem instances that take large times to solve. Fig. 4 shows a scatterplot of the speedup obtained versus the solution time for a single processor, where this trend is readily discernable.

A detailed analysis of the trace files produced by the simulator was done in order to obtain real time information. Table 5 shows the speedups obtained as ratios of actual times. Unfortunately, the size of the trace file thus obtained grows enormously as the solution time or the number of processors increases, and is beyond the capacity of our machine. We have, hence, been able to obtain results only for a few of the problems, and only for up to five processors. A comparison with Table 4 shows that the degradation because of global memory accesses is quite small. The analysis for these instances showed that, on average, processors spent about 20% of the total time in accessing the global location containing the cost of the best solution; and about 8% of the time accessing the global list of searched subtrees (for a list size of 15 nodes). It should be noted in this connection that for the Knapsack problem relatively simple bounding functions and branching rules can be formulated which incur little computation costs, as compared to, say, the Travelling Salesman problem. The global memory access percentages would be even better in the latter case.

VI. CONCLUSION

We have proposed a randomized version of the branch-and-bound algorithm suitable for multi-computing. With little communication overhead, we are able to obtain reasonable speedups for small numbers of processors. The proposed method has the following advantages:

(i) It is application independent, and transparent to the user.

(ii) Each processor is capable of solving the problem by itself. So if any processor were to fail, the other processors would still be able to perform unhindered. The overall performance would depend on these working processors. The same would apply if the communication links should fail.

(iii) The scheduler is very simple to implement, and is very flexible. There is no need to modify it when more processors are added. Changes due to modification of global memory size are trivial.

A system based on deterministic search would be hard pressed to provide these advantages.

A model was devised to estimate the speedups, and predicts the behavior of the system to a good degree. Finally detailed simulation results were presented, which show the actual performance of the new system.

Future research will be directed towards exploring the best-first branch-and-bound algorithm as a candidate for randomization. The present model neglects delays caused by queueing and communication times. A better model must be devised which will

be able to account for these delays.

VII. REFERENCES

[1] E. D. Brooks, "A multitasking kernal for the C and Fortran programming languages," Tech. Rep. UCID 20167, Lawerence Livermore National Laboratory, Livermore, CA, Sept. 1984.

[2] N. G. De Bruijn, *Asymptotic Methods in Analysis*. Amsterdam: North Holland Publishing Co., 1961.

[3] O. I. El-Dessouki and W. H. Huen, "Distributed enumeration on network computers," *IEEE Transactions on Computers*, vol. C-29, no. 9, pp. 818-825, Sept. 1980.

[4] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Maryland: CSC Press, 1984.

[5] V. K. Janakiram, D. P. Agrawal, and R. Mehrotra, "Randomized parallel algorithms for PROLOG programs and backtracking applications," in *Proc. 1987 Intl. Conf. on Parallel Processing*, Chicago, IL, pp. 278-282, 1987.

[6] T. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms," in *Proc. 1984 Intl. Conf. Parallel Processing*, Chicago, IL, pp. 183-190, 1983.

[7] F. Maffioli, "Randomized algorithms in combinatorial optimization: A survey," *Discrete Applied Mathematics*, vol. 14, pp. 157-170, 1986.

[8] D. R. Smith, "Random trees and the analysis of branch and bound procedures," *J. ACM*, vol. 31, no. 1, pp. 163-188, Jan. 1984.

[9] H. S. Stone and P. Sipala, "The average complexity of depth-first search with backtracking and cutoff," *IBM Journal of Research and Development*, vol. 30, no. 3, pp. 242-258, May 1986.

[10] B. W. Wah and C. F. Yu, "Probabilistic modelling of branch and bound algorithms," in *Proc. of the COMPSAC*, pp. 647-653, 1982.

[11] B. W. Wah and Y. W. Eva Ma, "MANIP - A Multicomputer architecture for solving combinatorial extremum-search problems," *IEEE Trans. on Computers*, vol. C-33, no. 5, pp. 377-390, May 1984.

[12] B. W. Wah and C. F. Yu, "Stochastic modeling of branch-and-bound algorithms with best-first search," *IEEE Trans. on Software Eng.*, vol. SE-11, no. 9, pp. 922-933, Sept. 1985.

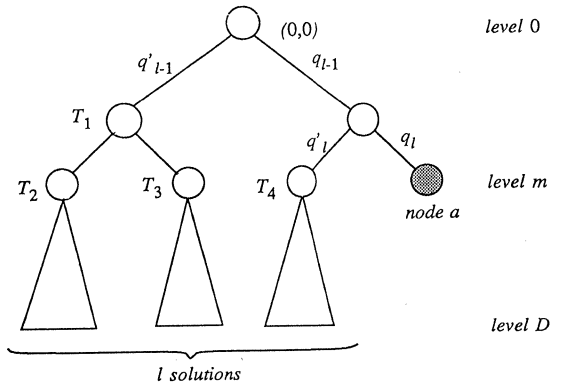


Fig. 1: Branch-and-bound Solution in Progress.

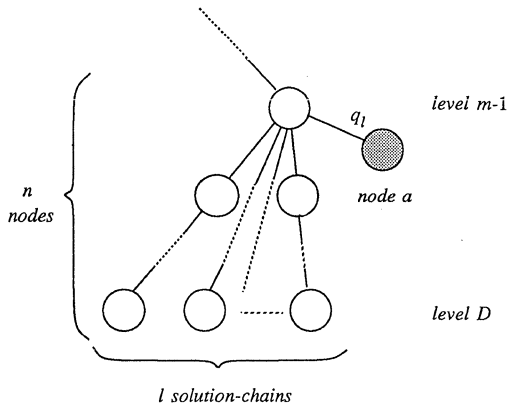


Fig. 2: Calculation of Cut-Off probabilities

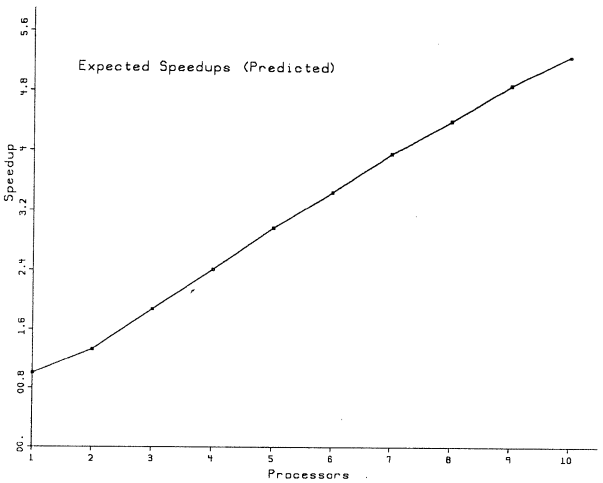


Fig. 3: Estimated Speedup.

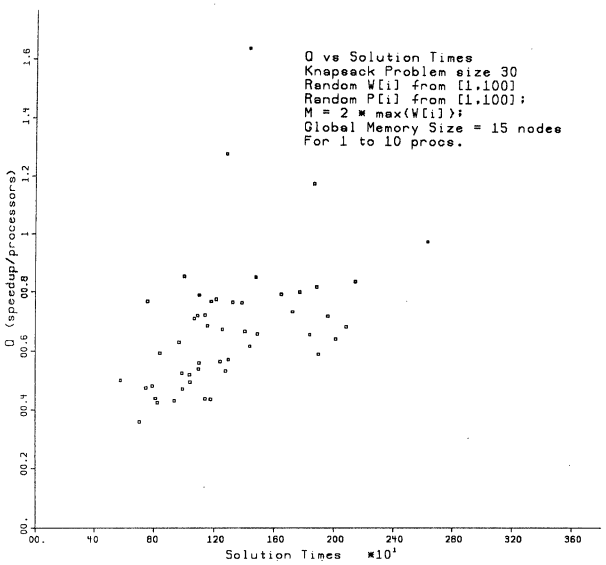


Fig. 4: Scatter-plot of Solution Times for Data-set 2.

TABLE 1: Expected number of nodes examined

Tree depth	Simulation	Model
2	6.34	6.58
3	11.50	12.59
4	19.83	21.85
5	32.36	34.43
6	49.32	50.45
7	76.76	73.57
8	110.72	107.21
9	160.19	156.64
10	229.83	230.03
11	321.37	340.06
12	443.40	506.45

TABLE 2: Expected Speedups (Predicted)

Procs.	2	3	4	5	6	7	8	9	10
Spdp.	1.33	1.87	2.41	2.97	3.45	3.96	4.40	4.88	5.27

TABLE 3: Details of Data-sets for the Knapsack Problem

Data-set No.	Profits (P_i)	Weights (W_i)	Capacity (M)
1	Random	Random	$\sum W_i/2$
2	Random	Random	$2\times\max\{W_i\}$
3	W_i+10	Random	$\sum W_i/2$
4	W_i+10	Random	$2\times\max\{W_i\}$
5	Random	P_i+10	$\sum W_i/2$
6	Random	P_i+10	$2\times\max\{W_i\}$

Note: Random ≡ random integer in the range [1,100].

TABLE 4: Average Speedups Obtained from Simulation

Data Set No.	Processors	Speedup for Global Memory Size 31	15	Nil	Range of Uniprocessor Solution Times
1	2	1.72	1.78	1.50	120—880
	4	2.21	2.00	1.77	
	6	4.23	4.09	2.26	
	8	4.37	4.53	2.83	
	10	4.79	4.39	2.90	
2	2	1.57	1.51	1.37	580—2600
	4	2.11	2.19	2.04	
	6	4.16	4.67	2.62	
	8	4.44	4.19	3.32	
	10	6.01	5.64	3.98	
3	2	1.67	1.64	1.54	280—3200
	4	2.51	2.34	1.97	
	6	3.67	3.92	2.69	
	8	5.90	6.08	3.33	
	10	5.47	6.21	3.23	
4	2	1.51	1.46	1.52	2500—24100
	4	2.23	2.12	2.25	
	6	4.15	4.84	2.68	
	8	3.79	3.43	3.79	
	10	4.91	5.29	4.14	
5	2	1.40	1.27	1.44	62—235
	4	1.65	1.70	1.64	
	6	1.74	1.89	1.70	
	8	1.73	1.90	1.78	
	10	1.86	1.74	1.70	
6	2	1.48	1.42	1.23	120—2240
	4	2.34	1.78	1.50	
	6	5.08	4.79	3.55	
	8	5.34	4.97	3.88	
	10	6.30	4.94	3.65	

Notes: (i) For a description of the data sets see Table 3.
(ii) Time to examine a node is taken as 1 time unit.

TABLE 5: Average Speedups Obtained from Simulation

Data Set No.	Procs.	Speedup (Global Mem. Size = 15 nodes)	Range of Uniprocessor Soln. Times
1	2	1.63	3.0×10^4—1.70×10^5
	3	2.13	
	4	1.88	
	5	2.73	
2	2	1.51	1.88×10^5—5.20×10^5
	3	2.26	
	4	2.16	
	5	2.91	
3	2	1.62	7.0×10^4—5.1×10^5
	3	2.80	
	4	2.26	
	5	2.55	

Notes: (i) For a description of the data sets see Table 3.
(ii) Speedups are real time ratios.
(iii) Unit of time is 1 VAX Instruction.

Generalized Parallel Processing Models for Database Systems

Pramanik, Sakti
&
Kim, Myoung Ho

Michigan State University
Computer Science Department,
East Lansing, MI 48824-1027

Abstract

In this paper we propose a two stage abstract parallel processing model to facilitate systematic design of parallel processing database systems. The objective of this model is to maximize throughput and minimize response time through concurrent I/O and processing of databases.

Based on the classification of database queries whose parallel processing characteristics are different, we present two specific parallel processing models which follow the abstract model. One is the FX model for partial match retrieval type applications, and the other is the Multi-directory Hashing model where database accesses are based on primary keys. We show that these proposed models perform better than those proposed earlier.

This two-stage modeling approach presents a new basis on which parallel processing systems for various database applications can be easily constructed.

1. Introduction

Parallel processing in database systems is important because it can maximize throughput and minimize response time by increasing concurrency in query processing. However, parallel processing by itself does not necessarily leads to high performance. Some of the reasons are attributed to overhead due to interprocessor communication, remote memory accesses and data access conflicts. For parallel database operations external I/O also causes serious bottleneck [3].

Most past research in this area have focused on machine architectures [23,24] which are specifically designed for database work. Our objective here is to investigate database processing model for general purpose parallel processing systems.

Stone [25] showed that parallel query algorithms in a multiprocessor system may perform poorly than efficient serial algorithms on single processor system. The advantage of indexing was also emphasized in that paper. Hillyer, et al. [10] and Hawthorn, et al. [9] investigated the performance of several database machines and the results show that the performance improvement depends on the query type as well as the architecture.

We observe that a parallel processing model which is appropriate for one database application may not be appropriate for another. For example, the granularity of parallel processing which is suitable for one application mat not be so for others. Thus, it may be necessary to develop different parallel processing algorithms for various types of applications. In the following paragraph we classify applications based on their parallel processing characteristics.

This research is supported in part by National Science Foundation Grant No. CCR-8706069, and Naval Research Laboratory Grant No. N00014-87-K-2022.

The types of queries whose parallel processing characteristics are different are classified, as follows.

(A1) Single query with multiple hits

(A2) Single query with a single hit

(A3) Single complex query

(A4) Multiple queries accessing the same relation

(A5) Multiple queries accessing different relations

Examples of queries of type A1 are partial match retrieval and range queries. Here, intra-query parallel processing is advantageous because a single query requiring many data records can be processed in parallel by multiple processors. Rosenau, et al. have also applied this type of parallel processing for projection operation on a relation [21].

It is rather difficult to exploit parallel processing of type A2 query because applying parallelism for this type of query may require finer granularity which may result in lower throughput of the system. On the other hand, parallel processing may achieve the lower bound on access time for these types of queries when appropriate software and hardware architecture is used. Achieving and guaranteeing this lower bound are important for many real-time critical applications. Parallel processing models of type A2 queries can be found in [17, 18].

Queries of type A3 include join functions, sorting of files, and complex qualifications. Several database machines which use functionally distributed architectures have been proposed for this type of query [11, 23].

For type A4 and A5 queries, transaction processing applications are good examples, where many independent queries can be processed in parallel. The throughput of the system for these applications can be improved by maximizing concurrency among the queries. Parallel processing models for these applications may also be developed based on the parallel processing models of type A1, A2 and A3 queries.

The remainder of this paper is organized as follows. In section 2 we propose an abstract parallel processing model for database systems. Section 3 describes optimal file distribution for A1 type queries. In section 4 we propose Multi-directory hashing model for A4 type queries. Section 5 contains concluding remarks.

2. Two Stage Abstract Parallel Processing Model for Database Systems

We propose an abstract database parallel processing model as given in Figure 1. The basic idea is to partition data mapping into two stages.

As shown in the Figure, the first stage, H1, is called Data Distribution algorithm and the second stage, H2, is called Data Construction algorithm. Q_i represents a parallel access node. This can be a memory module or a disk, depending on the parallel processing environment.

Data distribution algorithms determine how the data is appropriately distributed to the parallel access nodes so that maximum con-

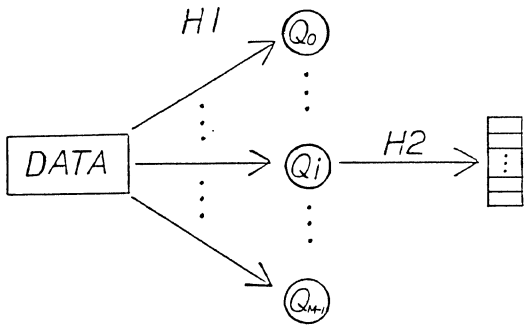


Figure 1. Abstract parallel processing model for database systems

currency is achieved between the access nodes. Data construction algorithms, on the other hand, determine the appropriate data structure to minimize the access time. It receives data from the data distribution algorithm and then create local access structures such as hashed or indexed files.

In general, the following strategies can be employed for data distribution :

(B1) Declustering based on query's data reference pattern

(B2) Random distribution

(B3) Objective specific declustering

(B4) Clustering based on data reference pattern

In method B1, the data distribution technique takes advantage of the data reference pattern of a query. For example, if a query references numerous records, the strategy may be to distribute the data so that these records are stored uniformly among the nodes. This approach may be useful for A1 type applications discussed in the previous section. In method B2, records are randomly distributed between the nodes. This method is simple, but may not guarantee a good distribution. In the objective specific method, records are allocated to optimize certain objective functions. For example, [17] proposes a data distribution technique to construct multiple directories for a single relation, where a record is allocated to the node which has the smallest directory size. It has been shown in that paper that this approach gives the minimum total directory size. However, declustering of data may not be always beneficial. For example, if the interconnection network topology is based on point-to-point connection and the communication cost is large, clustering may give better performance than declustering. So, B4 type strategies may depend on the interconnection network topology.

Data distribution algorithm can be a functional mapping which depends only on data values. It maps a set of data values into a set of nodes. For example, if node addresses are determined by hashed values of input data, distribution algorithm is a mapping which is independent of time or other system parameters. On the other hand, the distribution algorithm need not be functional. For example, random distribution may map the same record to different nodes at different time. Since data accesses are content-based for most database applications, it is advantageous to make a data distribution algorithm functional depending only on data values.

Let D be a set of data and $Z_M = \{0, 1, \ldots, M\text{-}1\}$ be a set of parallel access nodes. Let data distribution algorithm be a function from D to Z_M. Since actual data are quite unevenly distributed in the domain of data, data distribution algorithms are commonly designed based on the hashed values of data which are evenly distributed in hashed address space. Thus, we define data distribution algorithm H1 as a composition of two functions, $H1^{(1)}$ and $H1^{(2)}$, such that $H1^{(1)}$ is a mapping from D to T and $H1^{(2)}$ is a mapping from T to Z_M, where T is the set of hashed values. Figure 2 shows two level implementation of H1 for the abstract database parallel processing model. This model can be thought of as one class of database parallel processing model described in Figure 1.

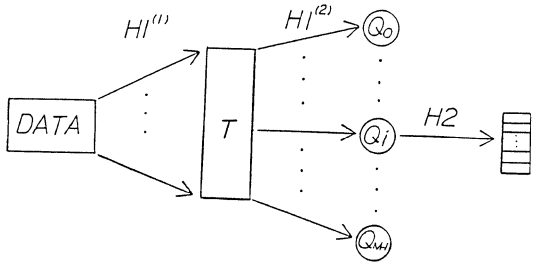


Figure 2. Two level implementation of H1 for the abstract model

Let H2 be a hash-based data construction algorithm and LD be a set of entries in all local directories generated by H2 for a given file system. If there exists one-to-one correspondence between T and LD, T is called a *real global directory*. Otherwise, it is called a *virtual global directory*. When T is a real global directory, the set of all the local directories can be thought of as a partition of T. $H1^{(1)}$ is usually static because the use of dynamic hashing for $H1^{(1)}$ will cause significant overhead due to internode data movement. However, static hashing scheme for $H1^{(1)}$ may result in very sparse local directories or long overflow chains. These problems can be avoided by using a virtual global directory, where the actual local directories are determined by H2.

When T is a real global directory, the ratio |T|/|D| directly affects the data retrieval time as well as storage utilization. On the other hand, we have more flexibility when T is a virtual global directory. The comparison of these two approaches will be described in more detail in section 3.2 and 4.2. Functional distribution, and real/virtual global directory concepts are used for FX model and Multi-directory hashing model presented in section 3 and 4.

3. Parallel Processing Model for Partial Match Retrieval Type Queries

In this section we present parallel processing model to process partial match retrieval type queries. Partial match queries are queries where some of the attributes are specified, hence a set of qualified records need to be retrieved. For example, q = [Age = *, Department = "mathematics", State = "Ohio"] is a partial match query, where * denotes a don't care condition.

It has been shown that multi-key hashing is effective for partial match retrieval type applications. Multi-key hash function, H, for a database consisting of n fields is a set of n functions $\{H_1, \cdots, H_n\}$ such that given a record $r = <r_1, \cdots, r_n>$, $H(r) = <H_1(r_1), \cdots, H_n(r_n)>$. $H(r)$ is usually called a bucket. Rothnie, et al. [22] and Rivest [20] have independently proposed the use of multi-key hashing, as an alternative to inverted files, to reduce the total search time for partial match retrieval type queries. The design of multi-key hash functions was considered in [4]. The determination of each field size for minimum search time based on query statistics was also investigated by [1, 2]. In [5] it has been shown that the problem of finding the optimal field sizes for multi-key hashing scheme is NP-hard.

The main objective of this section is to minimize the total number of bucket accesses for a partial match query by distributing buckets in multi-key hashing.

3.1. Data Distribution Algorithm For Partial Match Retrieval

The data distribution is said to be optimal for a partial match query, when no device has more than ⌈total number of qualified buckets / number of devices⌉ buckets. It has been shown in [26] that there does not exist an optimal data distribution method in certain types of file systems.

There are a few heuristic methods for distributing data in partial match retrieval type queries. Du, et al. have proposed data distribution method based on modulo allocation [6]. Modulo allocation is simple but does not work in many cases. For example, it may not give optimal distribution if some of the field sizes are less than the given number of devices. So, for a large number of parallel processing nodes such as Butterfly machines[27], Modulo distribution may not be appropriate. Generalized Disk Modulo (GDM) method has also been proposed in [6] to overcome this problem. This method gives a sufficient condition to achieve optimal distribution. However, no general method has been given to find the optimal distribution parameters. In fact, the problem of finding the optimal parameter values could be very complex [6]. Several useful properties of these modulo based distribution methods have also been given in [26]. Data distribution methods based on minimal spanning trees and short spanning paths have also been proposed in [8].

In this section we propose Fieldwise eXclusive-or (FX) distribution method which gives better performance for a wider range of parameter values than existing methods. The basic idea of the FX distribution method is the use of bitwise exclusive-or operation on the field values which are computed by multi-key hashing. Here, we show several useful characteristics of exclusive-or operation for optimal data distribution. Field transformation techniques have been used to extend the scope of optimality in FX distribution.

Before describing FX distribution method, it is necessary to introduce some notations as well as relevant definitions and assumptions for this section.

Definition :

- $f_i = \{0, 1, \ldots, F_i-1\}$, a set of hashed values of field i.
- F_i denotes $|f_i|$.
- M denotes the number of parallel devices.
- N is the set of all natural numbers including 0.
- Z_M is the set of all integers from 0 to M-1.
- $(a_{m-1} \ldots a_0)_B$ is a binary notation of an integer, where a_i is a binary digit.

$|f_i|$ is assumed to be a power of 2 which is common for hash directory files for partitioned [1] or dynamic hashing schemes [7, 13, 14]. The number of devices M is also assumed to be a power of 2.

Definition : Let R(q) be the set of buckets which satisfy qualifications for a partial match query q. The distribution method is called *strict optimal* for a partial match query in a given file system if each device has no more than $\lceil |R(q)|/M \rceil$ number of buckets. When the distribution method is strict optimal for all possible partial match queries in a given file system, it is called *perfect optimal* for that file system.

Definition : [+] denotes exclusive-or operation between two bits. We will use the same notation [+] to denote exclusive-or operation between integers and sets of integers as follows. When $X = (a_{m-1} \ldots a_0)_B$ and $Y = (b_{m-1} \ldots b_0)_B$ are two integers, $X [+] Y = (a_{m-1} [+] b_{m-1} \ldots a_0 [+] b_0)_B$. If X is an integer and $Y = \{y_1, \ldots, y_L\}$ is a set of integers, X [+] Y is defined as $\{X [+] y_i \mid y_i \in Y\}$. If both $X = \{x_1, \ldots, x_K\}$ and $Y = \{y_1, \ldots, y_L\}$ are sets of integers, X [+] Y is defined as $\{x_i [+] y_j \mid x_i \in X, y_j \in Y\}$

For example, if $X_1 = 2$ and $Y_1 = 3$ then $X_1 [+] Y_1 = 1$. If $X_2 = 2$ and $Y_2 = \{0, 1, 2, 3\}$ then $X_2 [+] Y_2 = \{0, 1, 2, 3\}$.

Definition : $\overset{n}{\underset{i=1}{[+]}}(Y_i) = Y_1 [+] Y_2 [+] Y_3 \cdots [+] Y_n$.

Note that $\overset{n}{\underset{i=1}{[+]}}$ is a shorthand notation for performing exclusive-or operation between sets of integers $Y_1, Y_2, \ldots, Y_n$.

Because of space limitation, the proofs of the lemmas and the theorems are not given. They can be found in [12, 19].

Compared to the abstract model of Figure 2, $H1^{(1)}$ is a multi-key hashing and FX distribution in this section corresponds to $H1^{(2)}$. In FX distribution model, T described in Figure 2 is a set of ordered n-tuple produced by multi-key hashing.

3.1.1. Basic FX Distribution

Let $f_1 \times f_2 \times \ldots \times f_n$ be a set of all buckets. Basic FX distribution method allocates bucket $\langle J_1, \ldots, J_n \rangle$ into device $T_M \left[\overset{n}{\underset{j=1}{[+]}}(J_j) \right]$, where $T_M : N \rightarrow Z_M$ is a function which returns only the rightmost $\log_2 M$ bits of domain values, and $J_j \in f_j$ for j = 1, ... , n.

Example 1. Table 1 shows the bucket distribution by Basic FX distribution method, where $f_1 = \{0, 1\}$, $f_2 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and M = 4. In this table, binary numbers are used for field values and decimal numbers are used for Device No. (This convention will be used in all examples of FX distribution). Here, Device No = $T_M \left[J_1 [+] J_2 \right]$, where $J_1 \in f_1$, $J_2 \in f_2$ and T_M returns the rightmost two bits of the result of $J_1 [+] J_2$.

f_1	f_2	Device No
000	000	0
000	001	1
000	010	2
000	011	3
000	100	0
000	101	1
000	110	2
000	111	3
001	000	1
001	001	0
001	010	3
001	011	2
001	100	1
001	101	0
001	110	3
001	111	2

Table 1. Basic FX distribution

As shown in the Table 1, Basic FX distribution is strict optimal for any partial match query in the file system of example 1. For example, when $(001)_B$ is specified for the first field and the second field is unspecified, we have to access eight buckets $\langle (001)_B, (000)_B \rangle, \ldots, \langle (001)_B, (111)_B \rangle$. Since each device has two qualified buckets for this partial match query, FX distribution is strict optimal for this query.

Lemma 1.1. Z_M is a set which contains M different nonnegative integers from 0 to M-1. Let k be some integer $0 \le k \le M-1$. Then $Z_M [+] k = Z_M$.

Example 2. Let $Z_8 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and k = 3. Then $Z_8 [+] k = \{3, 2, 1, 0, 7, 6, 5, 4\} = Z_8$.

Lemma 1.1 is a basic property which is used in the proofs of several

theorems.

Theorem 1. Basic FX distribution is strict optimal for any partial match query in which the number of unspecified fields is 0 or 1.

Theorem 2. For any partial match query which has two or more unspecified fields, Basic FX distribution is strict optimal, if there exists at least one unspecified field i such that $F_i \geq M$.

Note that Theorem 1 works for partial match queries with only one unspecified field while Theorem 2 applies to partial match queries with more than one unspecified fields.

Theorem 1 and 2 show general characteristics of exclusive-or operation for optimal file distribution. This is mainly due to the property described in Lemma 1.1. However, Basic FX distribution does not give optimal distribution for partial match queries with 2 or more unspecified fields, when the size of none of the unspecified fields is greater than or equal to M. For example, when M = 16 and all others are the same as in example 1, the distribution is not optimal. Since every element in f_1 and f_2 is much smaller than M, the reason of not being optimal is clear. Theorem 3 gives the sufficient conditions for optimal distribution for these cases.

Theorem 3. Let q(f) = $\{i_1, i_2, \ldots, i_k\}$ be the set of unspecified fields for partial match query q, where $F_j < M$, for all j ∈ q(f). Basic FX distribution is strict optimal for partial match query q, if there exist a set of fields $\{i_1, \ldots, i_j\} \subseteq$ q(f) such that $|f_{i_1} \times \cdots \times f_{i_j}| \geq M$ and $\#\{(J_{i_1}, \cdots J_{i_j}) \in f_{i_1} \times \cdots \times f_{i_j} \mid T_M\left[\overset{j}{\underset{p=1}{[+]}}(J_{i_p})\right] = z\} = |f_{i_1} \times \cdots \times f_{i_j}|/M$ for all $z \in Z_M$.

Theorem 3 says that we can guarantee optimal distribution, if (1) there exists a subset of the unspecified fields whose size of cartesian product is greater than or equal to M and (2) the records projected on these fields are distributed uniformly among the M devices.

However, when the size of none of the unspecified fields is greater than or equal to M, the conditions given in Theorem 3 is not satisfied in Basic FX distribution. In the next section we will introduce field transformation techniques. These field transformation techniques increase the scope of optimality by themselves, and also utilize Theorem 3.

The following paragraph exemplifies the idea of field transformation techniques. Let $f_1 = \{0, 1\}, f_2 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and M = 16. As we discussed, Basic FX distribution method does not give an optimal distribution for this file system. Let X be an one-to-one mapping such that $X(f_1) = \{0, 8\}$. When Basic FX distribution method is applied for $X(f_1) \times f_2$, the distribution is perfect optimal. (It can be easily verified by substituting $(1000)_B$ for $(001)_B$ in f_1 column of Table 1.) Now, the problem is to find a general one-to-one mapping, X, such that Basic FX distribution method for $X(f_1) \times f_2$ gives optimal distribution.

We will present several field transformation functions such as, X, described above. Even though the techniques developed in this paper may not achieve perfect optimal distribution in all the cases, this extended FX distribution method will give strict optimal distribution for a large class of partial match queries.

3.1.2. FX Distribution With Field Transformation Functions

In the previous section Basic FX distribution method was defined. In this section we extend Basic FX distribution method by using the field transformation techniques.

Let $f_1 \times f_2 \times \ldots \times f_n$ be a set of all buckets. Extended FX distribution method allocates bucket $\langle J_1, \ldots, J_n \rangle$, $J_j \in f_j$ for j = 1, ... , n, into device $T_M\left[\overset{n}{\underset{j=1}{[+]}}(X^{M,|f_j|}(J_j))\right]$, where

i) if $|f_j| \geq M$, $X^{M,|f_j|}$ is an identity function,

ii) if $|f_j| < M$, $X^{M,|f_j|}$ is an element of set of injective (one-to-one) functions whose domains are f_j and ranges are Z_M.

$X^{M,|f_j|}$ is called *field transformation function.*

When $X^{M,|f_j|}$ is identity function for all j=1, ... , n, Extended FX distribution method reduces to Basic FX distribution method. From now on, we will simply call FX distribution instead of Extended FX distribution.

It is easy to see that all lemmas and theorems that hold for Basic FX distribution also hold for FX distribution. Since the fields whose sizes are no less than the given number of devices M, do not cause any problem (whether it is specified or not), in this subsection we will focus only on the fields whose sizes are less than M.

Definition : Let M be a power of 2.

(1) I : N → N is an identity function.

(2) For each proper subset f_l of Z_M, where $|f_l|$ is some power of 2, $\mathbf{U}^{M,|f_l|} : f_l \rightarrow Z_M$ is a function such that $\mathbf{U}^{M,|f_l|}(l) = ld_l^{M,|f_l|}$, where $l \in f_l$, $d_l^{M,|f_l|} = \frac{M}{|f_l|}$.

(3) For each proper subset f_l of Z_M, where $|f_l|$ is some power of 2, $\mathbf{IU1}^{M,|f_l|} : f_l \rightarrow Z_M$ is a function such that $\mathbf{IU1}^{M,|f_l|}(l) = l\ [+]\ ld_l^{M,|f_l|}$, where $l \in f_l$, $d_l^{M,|f_l|} = \frac{M}{|f_l|}$.

(4) For each proper subset f_l of Z_M, where $|f_l|$ is some power of 2, $\mathbf{IU2}^{M,|f_l|} : f_l \rightarrow Z_M$ is a function such that $\mathbf{IU2}^{M,|f_l|}(l) = l\ [+]\ ld_{l1}^{M,|f_l|}\ [+]\ ld_{l2}^{M,|f_l|}$, where $l \in f_l$, $d_{l1}^{M,|f_l|} = \frac{M}{|f_l|}$, $d_{l2}^{M,|f_l|} = \begin{cases} d_{l1}^{M,|f_l|}/|f_l| & \text{if } |f_l|^2 < M \\ 0 & \text{otherwise} \end{cases}$

We have defined the four groups of basic functions, I, $U^{M,|f_l|}$, $IU1^{M,|f_l|}$ and $IU2^{M,|f_l|}$ which will be used in various combinations for optimal file distribution. For example, for any values of $|f_i|$, $|f_j|$ and M, FX distribution method distributes elements of $I(f_i) \times U^{M,|f_j|}(f_j)$ optimally.

It is not difficult to see that for any proper subset f_l of Z_M whose $|f_l|$ is some power of 2, all the functions defined above satisfy the requirements of field transformation functions described previously.

Because of notational complexity, when the context is clear, we will leave out the superscripts M, $|f_l|$ from transformation functions and their parameters.

Theorem 4. When there are only two fields i, j whose sizes are less than the given number of devices M, FX distribution with $I(f_i)$ and $U(f_j)$ is perfect optimal.

Example 3. Let $f_1 = \{0, 1, 2, 3\}, f_2 = \{0, 1, 2, 3\}$ and M = 16. Table 2 shows bucket distribution by FX and Modulo methods. Note that $I(f_1) = \{0, 1, 2, 3\}$ and $U(f_2) = \{0, 4, 8, 12\}$ are I and U transformed values of f_1, f_2 and denoted by binary numbers in the table. Here, Device No = $T_M(I(J_1)\ [+]\ U(J_2))$ for FX distribution, and Device No = $(J_1 + J_2)\ mod\ M$ for Modulo distribution, where $J_1 \in f_1, J_2 \in f_2$.

The FX distribution in Table 2 is optimal. But in Modulo distribution, it is skewed. GDM method can also give optimal distribution by multiplying 3 to the first field values and by 4 to the second field values. However, these parameters should be found by trial and error method.

f_1	f_2	$I(f_1)$	$U(f_2)$	Device No (FX)	Device No (Modulo)
0	0	0000	0000	0	0
0	1	0000	0100	4	1
0	2	0000	1000	8	2
0	3	0000	1100	12	3
1	0	0001	0000	1	1
1	1	0001	0100	5	2
1	2	0001	1000	9	3
1	3	0001	1100	13	4
2	0	0010	0000	2	2
2	1	0010	0100	6	3
2	2	0010	1000	10	4
2	3	0010	1100	14	5
3	0	0011	0000	3	3
3	1	0011	0100	7	4
3	2	0011	1000	11	5
3	3	0011	1100	15	6

Table 2. FX distribution with I and U transformation

On the other hand, FX distribution techniques give a specific method.

Theorem 5 When there are only two fields i, k whose F_i and F_k are less than the given number of devices M, FX distribution with $I(f_i)$ and $IU1(f_k)$ is perfect optimal.

Theorem 6. When there are only two fields j, k whose F_j, F_k are less than the given number of devices M, FX distribution with $U(f_j)$ and $IU1(f_k)$ is perfect optimal.

Example 4. Let $f_1 = \{0, 1, 2, 3\}, f_2 = \{0, 1, 2, 3\}$ and M = 16. Table 3 shows the FX distribution with $U(f_1)$, $IU1(f_2)$. Here, Device No = $T_M(U(J_1)$ [+] $IU1(J_2))$, where $J_1 \in f_1, J_2 \in f_2$.

$U(f_1)$	$IU1(f_2)$	Device No
0000	0000	0
0000	0101	5
0000	1010	10
0000	1111	15
0100	0000	4
0100	0101	1
0100	1010	14
0100	1111	11
1000	0000	8
1000	0101	13
1000	1010	2
1000	1111	7
1100	0000	12
1100	0101	9
1100	1010	6
1100	1111	3

Table 3. FX distribution with U and IU1 transformation

Theorem 7. When there are only two fields i and k whose F_i and F_k are less than the given number of devices M, FX distribution with $I(f_i)$ and $IU2(f_k)$ is perfect optimal.

Theorem 8. When there are only two fields j and k whose F_j and F_k are less than the given number of devices M, FX distribution with $U(f_j)$ and $IU2(f_k)$ is perfect optimal.

Lemma 9.1. When there are only three fields i, j and k whose sizes are less than the given number of devices M, FX distribution with $I(f_i)$, $U(f_j)$ and $IU2(f_k)$ is perfect optimal, if either

(1) there exist at least 2 fields p and q such that p,q $\in$ {i, j, k} and $F_pF_q \geq M$ or

(2) $F_k \geq F_j$ and $F_k^2 < M$

Theorem 9. Let L be the set of fields whose sizes are less than the given number of devices M in a given file system. FX distribution with I, U and IU2 transformation can be always perfect optimal, if $|L| \leq 3$.

Example 5. Let $f_1 = \{0, 1, 2, 3\}, f_2 = \{0, 1\}, f_3 = \{0, 1\}$ and M = 16. Table 6 shows the FX distribution with $I(f_1)$, $U(f_2)$ and $IU2(f_3)$.

$I(f_1)$	$U(f_2)$	$IU2(f_3)$	Device No
0000	0000	0000	0
0000	0000	1101	13
0000	1000	0000	8
0000	1000	1101	5
0001	0000	0000	1
0001	0000	1101	12
0001	1000	0000	9
0001	1000	1101	4
0010	0000	0000	2
0010	0000	1101	15
0010	1000	0000	10
0010	1000	1101	7
0011	0000	0000	3
0011	0000	1101	14
0011	1000	0000	11
0011	1000	1101	6

Table 4. FX distribution with I, U and IU2 transformation

We have determined, through theorems, the class of partial match queries whose qualified buckets are distributed optimally under FX distribution. Even though FX distribution does not always guarantee strict optimal distribution, FX distribution gives optimal distribution for a large class of partial match queries.

3.1.3. Performance Comparisons to Other Distribution Methods

In this section we compare FX distribution with Modulo and GDM method. The performance comparisons are based on the probability of strict optimality and response time for a given partial match query. In the following subsections, it is assumed that the probability of each field being specified is the same for all fields and some field being specified is independent of each other.

3.1.3.1. Probability of Strict Optimality

In this section we show that the probability of strict optimality for FX distribution is higher than Modulo distribution. Even for the worst case the decrease of probability of strict optimality for FX distribution is not much. On the other hand, in Modulo distribution the decrease is quite large. Since no general method has been given to determine the existence of parameter values for strict optimal distribution in GDM method, we compare FX distribution to only Modulo distribution in this section.

Figure 3 and 4 show the percentage of strict optimal distribution for all possible partial match queries in a given file system consisting of ten fields. In all these Figures MD denotes Modulo Distribution and FD denotes FX Distribution. Here, results are computed from sufficient conditions given for each method. Figure 3 shows the case where any two fields p and q satisfy the condition, $F_pF_q \geq M$. In this figure FX distribution used I, U and IU1 transformation methods.

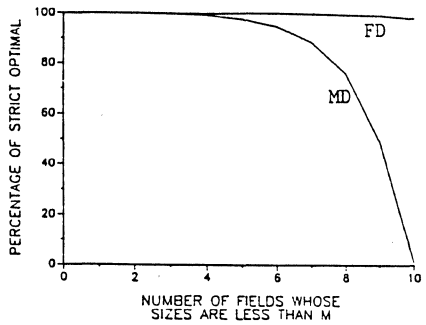


Figure 3. Probability of strict optimality

Figure 4 shows the case when for any two fields p,q, $F_pF_q < M$ but for any three fields p,q,r $F_pF_qF_r \geq M$. Here, in FX distribution I, U and IU2 transformation methods are used.

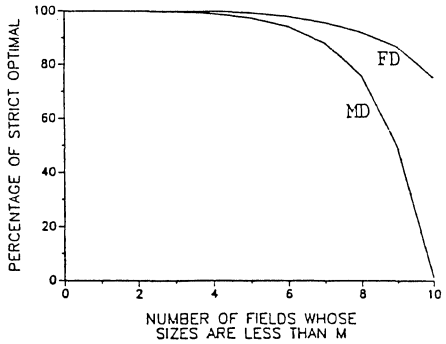


Figure 4. Probability of strict optimality

3.1.3.2. Average Response Time

Definition : For a given partial match query q, $r_i(q)$ is defined as the number of qualified buckets in device i for a partial match query q. We call this a *response size* for device *i*. Then, the *largest response size* for a partial match query q is defined as $MAX(r_1(q), r_2(q), \cdots r_{M-1}(q))$.

For the response time of a partial match query, we consider two factors, namely, largest response size and CPU computation time for bucket distribution and inverse distribution, where inverse distribution is a procedure used to find qualified buckets. In parallel disks environment, largest response size is the most important factor, while in main memory databases CPU computation time is more important.

(1) Largest Response Size

When systems are configured such that data retrieval time for any device is the same, the response time for a partial match query is determined by the device which has the largest number of qualified buckets.

Table 5 through 7 show the largest response size of Modulo, GDM and FX distribution for some typical file system environments. The number of fields is assumed to be 6 for all these experiments. The first column denotes the number of unspecified fields. For GDM method we used three different sets of multiplication parameters. These sets are GDM1 : 2, 3, 5, 7, 11, 13 and GDM2 : 2, 5, 11, 43, 51, 57 and GDM3 : 41, 43, 47, 51, 53, 57. The FX distribution of Table 5 and 6 used I transformation for fields 1 and 4, U transformation for fields 2 and 5, IU1 transformation for fields 3 and 6. The FX distribution of Table 7 used IU2 transformation instead of IU1 transformation and others are the same as in Table 5 and 6. In all Tables, each entry is computed as an average value of largest response sizes from all possible partial match queries for that entry.

The tables show that except for first row of Table 6 and 7, FX distribution gives smaller largest-response-size than the other methods. FX distribution is also very close to optimal. It should also be noted that there may be a set of multiplication parameters by which GDM method can give better performance than those of GDM1, GDM2 and GDM3. However, even though such a set of parameters may exist, it can only be found by trial and error method.

(2) CPU Computation Time

In GDM method we reduce computation time by changing Modulo function into AND operation. This can be done because the number of devices is assumed to be a power of 2. In FX distribution, since the multipliers for U, IU1 and IU2 transformation are always power of 2, we can substitute multiplication by shift operation. Note that we cannot do this in GDM method because multipliers in GDM method are usually chosen from prime or odd numbers. Function T_M is done by AND operation.

	Modulo	GDM1	GDM2	GDM3	FX	Optimal
2	8.0	3.3	3.6	3.7	3.2	2.0
3	48.0	18.1	18.9	18.9	16.0	16.0
4	344.0	130.5	132.7	132.5	128.0	128.0
5	2460.0	1026.3	1029.7	1031.7	1024.0	1024.0
6	18152.0	8196.0	8198.0	8202.0	8192.0	8192.0

Table 5. M = 32, $F_1 = \ldots = F_6 = 8$

	Modulo	GDM1	GDM2	GDM3	FX	Optimal
2	8.0	2.1	2.2	2.4	2.4	1.0
3	48.0	10.2	10.3	10.6	8.0	8.0
4	344.0	68.3	68.1	67.5	64.0	64.0
5	2460.0	520.5	517.0	517.3	512.0	512.0
6	18152.0	4114.0	4102.0	4102.0	4096.0	4096.0

Table 6. M = 64, $F_1 = \ldots = F_6 = 8$

	Modulo	GDM1	GDM2	GDM3	FX	Optimal
2	9.6	1.7	1.3	1.4	2.3	1.0
3	91.2	10.0	5.5	5.6	5.1	3.2
4	911.2	90.3	40.5	42.2	37.3	35.2
5	9076.0	909.5	397.3	408.67	384.0	384.0
6	90404.0	9176.0	4144.0	4313.0	4096.0	4096.0

Table 7. M = 512, $F_1 = ... = F_3 = 8$ and $F_4 = ... = F_6 = 16$

In MC68000 processor, computation time of FX method is much faster than GDM method (In MC68000, XOR takes 8 cpu clock cycles, ADD takes 4 clock cycles, AND takes 4 clock cycles, n bit shift takes 6 + 2n clock cycles. But multiplication takes about 70 clock cycles). In intel 80286/80386 processor the ratios of clock cycles between different operations are almost similar to those of MC68000.

For main memory database systems FX method is more efficient than GDM method. The computation time for Modulo distribution is shorter than FX distribution. However, as shown in Table 5 through 7, Modulo distribution is not suitable for a large number of parallel devices.

3.2. Data Construction Algorithms For Partial Match Queries

Multi-key hashing for a given file with n fields produce a subset of T, where $T = f_1 \times f_2 \times \ldots \times f_n$. As discussed in section 2, T can be used as either a real global directory or a virtual global directory.

3.2.1. Data Construction Using T As A Real Global Directory

Let $GD = [0..F_1-1, \ldots, 0..F_n-1]$ be a multi-dimensional array in which the range of i-th dimension is $0..F_i-1$. This is the same range of multi-key hashing for field i. Here, GD serves as a real global directory. Each element of GD contains an address of a bucket. FX distribution partition multi-dimensional array GD into M subsets. Note that a directory is also distributed among the nodes to achieve maximum concurrency. Then, we have to have efficient storage rule of this multi-dimensional array (e.g., the local address of array elements in each device). The storage rule for the distribution of this multi-dimensional array can be found in [19].

Using T as a real global directory is advantageous when for most $t \in T$, $\Theta 1^{-1}(t) \neq \phi$, where $\Theta 1$ is a given multi-key hash function. However, since T consists of cartesian product of all fields, many elements in GD may be empty. For example, when only $\frac{1}{10}$ of GD are actually used, the waste of storage to construct a whole directory is quite significant.

3.2.2. Data Construction Using T As A Virtual Global Directory

Let $< J_1, J_2, \ldots, J_n >$ be an ordered n-tuple produced by multi-key hashing $\Theta 1$. The local hash function $\Theta 2$ use this ordered n-tuple as an input key for its local directory when T is used as a virtual global directory. Note that the local directories physically exist and are considerably smaller than virtual global directory T which does not exist in this case. The local directories can dynamically grow and shrink while the virtual global directory is static.

This mapping scheme for local data construction is quite useful when for most $t \in T$, $\Theta 1^{-1}(t) = \phi$. The disadvantage of this approach is that it may cause more probings to find qualified records. This is because different buckets produced by $\Theta 1$ can be mapped into the same local directory entry by $\Theta 2$.

Let V1 be a set of buckets produced by $\Theta 1$ for a given file which are allocated into a same device. Let V2 be the range of local hash function $\Theta 2$. Let $\mu 1 = |V1|$ and $\mu 2 = |V2|$. Let τ be an average number of elements in V1 which are mapped into same $v \in V2$ by $\Theta 2$. Then, the probability P that $\Theta 2^{-1}(v) = \phi$ for some $v \in V2$, is given by $P = \left[1 - \frac{1}{\mu 2}\right]^{\mu 1}$. Let $c = \mu 2/\mu 1$. Then, $\tau \approx \frac{1}{c(1-e^{-1/c})}$. This can be derived by using $\lim_{\mu 1 \to \infty} (1 - \frac{1}{c\,\mu 1})^{\mu 1} = e^{-1/c}$, and $\tau = \frac{\mu 1}{(1-P)\mu 2}$. τ decreases with the increase of c and converges to 1.

Achieving efficient storage utilization, e.g., c = 1, causes more probings to find a bucket. To reduce the number of probings, larger local hashed directory is needed. Compared to the multi-dimensional array view of a real global directory, if the empty portion of GD is large, the mapping scheme of a virtual global directory can achieve data retrieval as fast as a real global directory scheme, while it requires less amount of memory than a real global directory. Otherwise, both methods have time/space trade-off.

4. Multi-Directory Hashing for Key Access Applications

In this section we present another application of the abstract parallel processing model. This application is for random access file system and is based on multi-directory hashing scheme [PrCh87]. Multi-directory hashing is a class of hashing schemes which use multiple directories to maximize concurrent accesses to a single relation. Each one of the directories can be of different size and it grows and shrinks dynamically.

It will be shown that multi-directory hashing provides better performance than single directory hashing schemes. The performance difference between single directory and multi-directory hashing schemes becomes significant in main memory databases. This is because short overflow chain (e.g., one record) is needed for main memory databases to reduce the main memory processing cost (e.g., comparing key values). In the following section we present one possible method for implementing multi-directory hashing.

4.1. Construction of A Multi-Directory Hashing

The algorithm for a multi-directory hashing is described below.

(1) The hashed address is partitioned into two parts. One part is used for directory number and the other part is used for locating the record within a local directory.

(2) Each directory of a multi-directory hashing is created based on a hashing method.

Figure 5 shows the address mapping scheme of a proposed multi-directory hashing. This figure also shows relationship between the functions of this model and those of an abstract model in Figure 2.

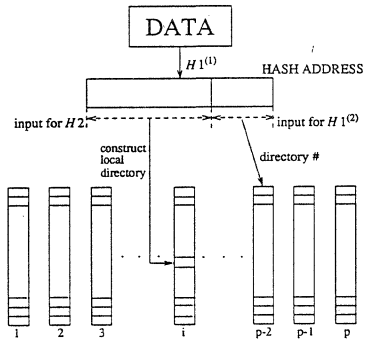


Figure 5. Hash address Mapping in Multi-Directory Hashing

In the next section the performance improvement of a multi-directory hashing over single directory hashing is described.

4.2. Performance Comparison for Multi-directory Hashing

Here, we show the reduction in main memory requirement while achieving near optimal response time (i.e., one access for a record and one key comparison). Extendible and linear hashing schemes are used for constructing the directories.

In the following figures the directory size for a multi-directory hashing corresponds to the total size of all the directories. Figure 6 gives the directory sizes for a multi-directory hashing when 5000 unique key values are inserted into the file. In this figure the total directory size decreases considerably with increasing number of directories.

Figure 7 shows the cases of various file sizes. We see that the reduction in directory sizes is significant for larger files when the number of directories increases.

The lower bound on response time can be achieved in multi-directory hashing at a much lower main memory requirement. On the other hand, the throughput increases considerably by concurrent processing of the multi-directory hashing. Here, we process data requests in parallel by accessing multiple directories concurrently.

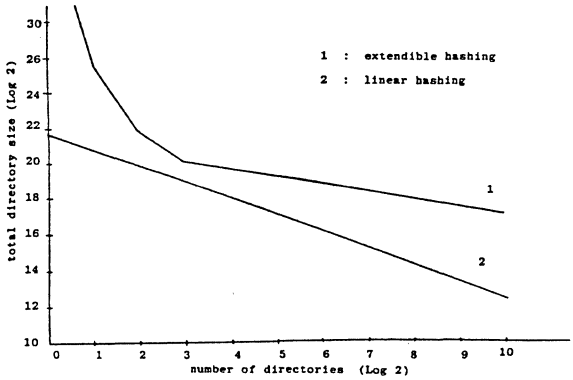


Figure 6. Total directory size in multi-directory hashing

5. Conclusion

In this paper we propose abstract parallel processing model for database systems which consists of data distribution stage and data construction stage. This two-stage model helps in systematically developing an efficient parallel processing database system.

We classify database queries whose parallel processing characteristics are different. Parallel processing models for two of these classes are then presented. Maximizing concurrency and minimizing response time are the most important objectives for these two models.

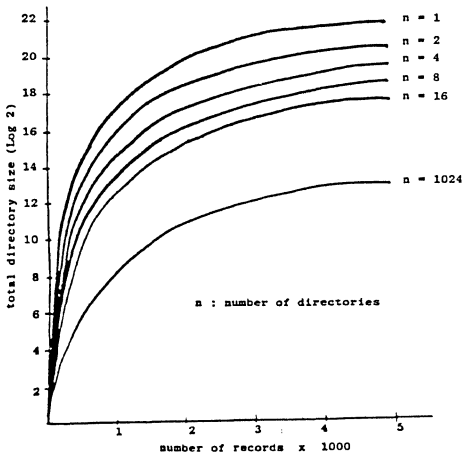


Figure 7. Multi-directory hashing for various file sizes

First, we propose the FX model for partial match retrieval type queries. In the distribution stage of FX model, several characteristics of exclusive-or operation are exploited to achieve optimal file distribution. The optimality conditions are derived through lemmas and theorems. We compare the FX distribution method with others and show the performance improvement of our methods. In the construction stage of FX model, two data construction methods based on the real and virtual global directory, are presented. Here, performance trade-off for these methods are investigated.

Second, we propose the multi-directory hashing scheme which is suitable for concurrent accesses to a single file. Our focus here is main memory database accesses on primary keys. We show that the proposed multi-directory hashing scheme gives improved performance over single directory hashing.

Acknowledgement

The authors would like to thank Hsiao-Yu Chou for the simulation results of Figure 6 and 7.

6. References

[1] Aho, A.V. and Ullman, J.D., "Optimal Partial-Match Retrieval When Fields Are Independently Specified," *ACM Trans. Database Systems*, vol. 4 no. 2 , June 1979, pp. 168-179.

[2] Bolour, A., "Optimality Properties of Multiple-key Hashing Functions," *JACM, vol. 26, no. 2, April 1979, pp. 196-210.*

[3] Boral, H. and Dewitt, D.J., "Database Machines : An Idea Whose Time Has Passed? A Critique of the Future of Database Machines," *Database machines*, Leilich, H.O. and Missikoff, M., eds., Springer-Verlag, 1983, pp. 166-187.

[4] Burkhard, W.A., "Hashing and Trie Algorithms for Partial Match Retrieval," *ACM Trans. Database Systems*, vol. 4, no. 2, June 1976, pp. 175-187.

[5] Du, H. C., "On the File Design Problem for Partial Match Retrieval," *IEEE Trans. Software Eng.* Vol. SE-11, No. 2, Feb. 1985, pp. 213-222.

[6] Du, H.C. and Sobolewski, J.S., "Disk Allocation for Cartesian Product Files on Multiple-Disk Systems," *ACM Trans. Database Systems*, vol. 7 no. 1, March 1982, pp.82-101.

[7] Fagin, R., "Extendible Hashing - A Fast Access Method For Dynamic Files,"*ACM Trans. Database Systems*, vol. 4 no. 3, Sept. 1979, pp.315-344.

[8] Fang, M.T., Lee, R.C.T. and Chang,C.C., "The idea of Declustering and Its Applications," *Proc. Conf. on Very Large Data Bases,"* Aug. 1986, pp. 181-188.

[9] Hawthorn, P.B. and Dewitt, D.J., "Performance Analysis of alternative Database Machine Architecture," *IEEE Trans. Software Eng.," vol. SE-8, Jan. 1982, pp. 61-75.*

[10] Hillyer, B., Shaw, D.E. and Nigam, A., "NON-VON's Performance on Certain Database Benchmarks," *IEEE Trans. Software Eng.* vol. SE-12, no. 4, April 1986, pp. 577-583.

[11] Kakuta,T., Miyazaki,N., Shibayama,S., Yokota,H. and Murakami,K.,"The Design and Implementation of Relational Database Machine Delta," *Database Machines, Fourth International Workshop*, March 1985, pp.

[12] Kim, M.H. and Pramanik, S., "Optimal File Distribution For Partial Match Retrieval," *Proc. ACM SIGMOD Conf.*, 1988.

[13] Larson, P., "Dynamic Hashing," *BIT*, 1978, pp. 184-201.

[14] Litwin, W., "Linear Hashing : A New Tool for File and Table Addressing," *Proc. 6th VLDB*, 1980, pp.212-223.

[15] Pramanik, S., "Performance analysis of a Database filter search hardware," *IEEE Trans. on Computers*, vol.C-35, no.12, Dec. 1986.

[16] Pramanik, S. and Chou, H., "Performance of Multi Directory Hashing," Technical Report, Computer Science Department, Michigan State University, Oct. 1987.

[17] Pramanik,S., Davis, H., "Multi Directory Hashing," Technical Report, Computer Science Department, Michigan State University, Oct. 1986.

[18] Pramanik, S. and Kim, M.H., "HCB_tree : A B_tree Structure for Parallel Processing," *Proc. Int'l Conf. on Parallel Processing*, Aug. 1987, pp. 140-146.

[19] Pramanik, S. and Kim, M.H., "Parallel Processing Models for Database Systems," *Techinical Report*, Computer Science Department, Michigan State University, 1987.

[20] Rivest,R.L., "Partial-Match Retrieval Algorithms," *SIAM J. Computing*, vol.5, No.1, March 1976, pp. 19-50.

[21] Rosenau,T. and Jajodia,S., "Parallel Relational Database Operations on the Butterfly Parallel Processor : Projection Results," Technical Report, Naval Research Laboratory, July 1987.

[22] Rothnie,J.B.Jr. and Lozano,T.,"Attribute based file organization in a paged memory environment," *Comm. ACM, vol.17, no.2, 1974, pp. 63-69.*

[23] Schweppe,H., Zeidler,H.Ch., Hell,W., Leilich,H.O., Stiege,G. and Teich,W.,"RDBM-A Dedicated Multiprocessor System for Database Management," *Advanced Database Machine Architecture*, Hsiao, D.K. ed., Prentice Hall, 1983, pp. 36-86.

[24] Stanley Y.W.Su, L.H. Nguyen, A. Eman and G. J. Lipovski, "The Architectural Features and Implementation Techniques of multicell CASSM," *IEEE Trans. on Computers*, June 1979, pp. 430-445.

[25] Stone, H., "Paralle Querying of Large Databases : A Case Study," *IEEE Computer*, Oct. 1987, pp. 11-21.

[26] Sung, Y. Y., "Performance Analysis of Disk Modulo Allocation Method for Cartesian Product Files," *IEEE Trans. on Software Eng.*, Vol. SE-13, No. 9, Sept. 1987, pp. 1018-1026 .

[27] *Butterfly Parallel Processor Overview*, BBN Report No. 6148 version 1, March 6, 1986.

IMAGE TEMPLATE MATCHING ON SIMD HYPERCUBE MULTICOMPUTERS [+]

Sanjay Ranka and Sartaj Sahni

Department of Computer Science

University of Minnesota

Abstract

Efficient algorithms for image template matching on fine grained SIMD hypercube multicomputers are developed. Our algorithms are asymptotically faster than previously known algorithms for this problem.

Keywords and Phrases

Hypercube multicomputer, image template matching, SIMD and MIMD multicomputers, one and two dimensional convolution

1. INTRODUCTION

The inputs to the image template matching problem are an $N \times N$ image matrix $I[0..N-1, 0..N-1]$ and an $M \times M$ template $T[0..M-1, 0..M-1]$. The output is an $N \times N$ matrix C2D where

$$C2D[i, j] = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} I[(i+u) \bmod N, (j+v) \bmod N] * T[u,v]$$

$C2D$ is called the two dimensional convolution of I and T. Template matching, i.e., computing $C2D$, is a fundamental operation in computer vision and image processing. It is often used for edge and object detection; filtering; and image registration [BALL85] [ROSE82]. Because of the fundamental nature of this problem and because of its high complexity ($O(M^2N^2)$ on a single processor computer), much attention has been devoted to the development of efficient fine grain multicomputer parallel algorithms. For example, Chang, Ibarra, Pong and Sohn [CHAN87] have studied this problem on an SIMD pyramid computers; Ranka and Sahni [RANK87a], Maresca and Li [MARE86] and Lee and Agarwal [LEE87] have considered mesh connected computers; and Fang, Li and Ni [FANG85], Fang and Ni [FANG86], and PrassanaKumar and Krishnan [PRAS86] have considered SIMD hypercube multicomputers; and Ranka and Sahni [RANK88] have considered MIMD hypercube multicomputers.

In this paper, we restrict our attention to SIMD hypercube multicomputers. We develop three asymptotically optimal algorithms that require N^2 processors. These require $O(M)$, $O(logM)$[!] and $O(1)$ memory per processor, respectively. The $O(M)$ memory algorithm is faster than the algorithms for $O(logM)$ and $O(1)$ memory by a constant factor. While the $O(logM)$ and O(1) memory algorithms are of comparable complexity, the former is conceptually simpler. [PRAS87] considers only the cases of $O(M)$ and $O(1)$ memory. The algorithms developed in [PRAS87] require a broadcast capability in the SIMD hypercube. Our algorithms do not require this. While the algorithm in [PRAS87] for the case of $O(M)$ memory is optimal, ours uses fewer interprocessor routes and so is faster even though no broadcasting is used. The algorithm of [PRAS87] for $O(1)$ memory is suboptimal by an $O(logM)$ factor. Our algorithm runs in the asymptotically optimal time of $O(M^2)$. Using the techniques of [PRAS87], each of our three algorithms may be generalized to obtain asymptotically optimal algorithms for SIMD hypercube computers with N^2K^2, $1 \leq K \leq M$ processors and $O(M/K)$, $O(log(M/K))$, and $O(1)$ memory per processor respectively.

Section 2 describes our hypercube model. In addition, notation and some fundamental data movement operations are developed in this section. In section 3, we develop fine grained algorithms for one dimension convolution. These form a basic component of our two dimensional convolution algorithms which are developed in Section 4.

2. PRELIMINARIES

2.1. Hypercube Multicomputer

The important features of an SIMD hypercube and the programming notation we use are:

1. There are $P = 2^p$ processing elements connected together via a hypercube interconnection network (to be described later). Each PE has a unique index in the range $[0, 2^p - 1]$. We shall use brackets([ ]) to index an array and parentheses('()') to index PEs. Thus A[i] refers to the i'th element of array A and A(i) refers to the A register of PE i. Also, A[j](i) refers to the j'th element of array A in PE i. The local memory in each PE holds data only (i.e., no executable instructions). Hence PEs need to be able to perform only the basic arithmetic operations (i.e., no instruction fetch or decode is needed).

2. There is a separate program memory and control unit. The control unit performs instruction sequencing, fetching, and decoding. In addition, instructions and masks are broadcast by the control unit to the PEs for execution. An *instruction mask* is a boolean function used to select certain PEs to execute an instruction. For example, in the instruction
$$A(i) := A(i) + 1, \quad (i_0 = 1)$$
$(i_0 = 1)$ is a mask that selects only those PEs whose index has bit 0 equal to 1. I.e., odd indexed PEs increment their A registers by 1. Sometimes, we shall omit the PE indexing of registers. So, the above statement is equivalent to the statement:
$$A := A + 1, \quad (i_0 = 1)$$

3. A p dimensional hypercube network connects 2^p PEs. Let $i_{p-1}i_{p-2}\ldots i_0$ be the binary representation of the PE index i. Let $\overline{i_k}$ be the complement of bit i_k. A hypercube network directly connects pairs of processors whose indices differ in exactly one bit. I.e., processor $i_{p-1}i_{p-2}\ldots i_0$ is connected to processors $i_{p-1}\cdots\overline{i_k}\ldots i_0$, $0 \leq k \leq p-1$. We use the notation $i^{(b)}$ to represent the number that differs from i in exactly bit b.

4. Interprocessor assignments are denoted using the symbol $\leftarrow$, while intraprocessor assignments are denoted using the symbol :=. Thus the assignment statement:
$$B(i^{(2)}) \leftarrow B(i), \quad (i_2 = 0)$$
is executed only by the processors with bit 2 equal to 0. These processors transmit their B register data to the corresponding processors with bit 2 equal to 1.

5. In a *unit route*, data may be transmitted from one processor to another if it is directly connected. We assume that the links in the interconnection network are unidirectional. Hence at any given time, data can be transferred either from PE i $(i_b = 0)$ to PE $i^{(b)}$ or from PE i $(i_b = 1)$ to PE $i^{(b)}$. Hence the instruction
$$B(i^{(2)}) \leftarrow B(i), (i_2 = 0)$$
takes one unit route, while the instruction:
$$B(i^{(2)}) \leftarrow B(i)$$
takes two unit routes.

[+] This research was supported in part by the National Science Foundation under grants DCR84-20935 and MIP 86-17374

[!] All logarithms are assumed to have base 2

6. Since the asymptotic complexity of all our algorithms is determined by the number of unit routes, our complexity analysis will count only these.

2.2. Hypercube Embedding of a Grid

Figure 1 gives a two dimensional grid interpretation of a 4

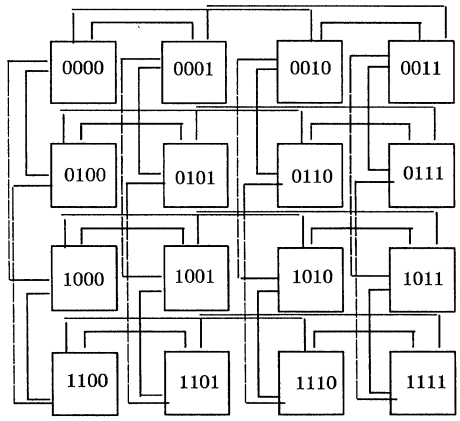


Figure 1: Embedding of a 4 × 4 mesh in a hypercube of dimension 4

dimensional hypercube. The index of the PE at position (i, j) of the grid is obtained using the standard row major mapping of a two dimensional array onto a one dimensional array [HORO85]. I.e, for an $N \times N$ grid, the PE at position (i, j) has index $iN + j$. Using this mapping, a two dimensional image grid $I(0..N-1, 0..N-1)$ is easily mapped onto an N^2 hypercube (provided N is a power of 2) with one element of I per PE. Notice that in this mapping, image elements that are neighbors in I (i.e., to the north, south, east, or west of one another) may not be neighbors (i.e., may not be directly connected) in the hypercube. This does not lead to any difficulties in the SIMD algorithms we develop.

2.3. Basic Data Manipulation Operations

2.3.1. Data Circulation

Consider a $P = 2^p$ processor hypercube. We are required to circulate the data in the A register of these PEs so that this data visits each of the P processors exactly once. A near optimal circulation for SIMD hypercubes results from the use of the exchange sequence X_p [DEKE81] defined as

$X_1 = 0, \quad X_q = X_{q-1}, q-1, X_{q-1} \ (q > 1)$

This sequence essentially treats a q dimensional hypercube as two $q-1$ dimensional hypercubes. Data circulation is done in each of these in parallel using X_{q-1}. Next an exchange is done along bit $q-1$. This causes the data in the two halves to be swapped. The swapped data is again circulated in the two half hypercubes using X_{q-1}. Let $f(q, i)$ be the i 'th number (left to right) in the sequence X_q, $1 \le i < 2^q$. The resulting SIMD data circulation algorithm is given in Figure 2. Because of our assumption of unidirectional

```
procedure CIRCULATE(A);
  {data circulation}
  for i = 1 to P - 1 do
      A(j^f(p, i)) ←A(j);
end
```

Figure 2: Data circulation in an SIMD hypercube

links, each iteration or the *for* loop of Figure 2 takes 2 unit routes. Hence Figure 2 takes 2(P-1) unit routes . The function f can be computed by the control processor in $O(P)$ time and saved in an array of size $P-1$ (actually it is convenient to compute f on the fly using a stack of height $\log P$). The following Lemma allows each processor to compute the origin of the current A value.

Lemma 1: Let $A_0, A_1, ..., A_{2^p-1}$ be the values in $A(0), A(1), ..., A(2^p-1)$ initially. Let $index(j, i)$ be such that $A[index(j, i)]$ is in $A(j)$ following the i'th iteration of the for loop of Figure 2. Initially, $index(j, 0) = j$. For every $i, i>0$, $index(j, i) = index(j, i-1) \ \theta \ 2^{f(p,i)}$ (θ is the exclusive or operator).

Proof: See [RANK87b].

Some of our algorithms will require the circulating data to return to the originating PEs. This can be accomplished by a final data exchange along the most significant bit. For convenience, we define $f(p, 2^p) = p - 1$.

2.3.2. Data Broadcast

In a data broadcast, data originates at one PE and is to be transmitted to the remaining $P-1$ PEs. This can be done using $\log P$ unit routes [DEKE81].

2.3.3. Window Broadcast

Assume that W is a power of 2 and that a P processor hypercube is tiled by windows of size $1 \times W$ such that each window forms a subhypercube with W PEs. In a window broadcast, data originates in one of these windows (different data in different PEs of the window). The data in this window is to be copied to the remaining $(P/W) - 1$ windows. This copying can be done using $\log(P/W)$ unit routes [DEKE81].

2.3.4. Data Sum

Assume the window tiling of Section 2.3.3.. The data in each of the windows is to be summed and the sum left in a prespecified PE (same relative PE for each window). For example, if we are summing the A register data, we may be required to compute:

$$Sum(iW) = \sum_{j=0}^{W-1} A(iW+j), \quad 0 \le i < (P/W)$$

Here, the sum is left in the first PE of each window. Data sum can be done in log W unit routes [DEKE81].

2.3.5. Shift

$SHIFT(A, i, W)$ shifts the A register data circularly counterclockwise by i in windows of size W. I.e, $A(qW+j)$ is replaced by $A(qW + (j-i) \ mod \ W)$, $0 \le q < (P/W)$. $SHIFT(A, i, W)$ on an SIMD computer can be performed in $2 \log W$ unit routes [PRAS87]. A minor modification of the algorithm given in [PRAS87] performs $i = 2^m$ shifts in $2 \log(W/i)$ unit routes ([RANK87b]).

2.3.6. Two Dimensional shift

$SHIFT2D(A, i, j, W, L)$ is used in conjunction with a two dimensional interpretation of a hypercube (or a grid mapping); $A(a, b)$ is shifted to $A((a-i) \ mod \ W, (b-j) \ mod \ L)$ in each $W \times L$ window. This can be done by first using SHIFT along one dimension and then along the other.

In a SHIFT2P, a $W \times W$ window is assumed and the amount of shift in each window can be different. A SHIFT2P takes at most $4 \log W$ unit routes on an SIMD hypercube.

2.3.7. Data Accumulation

For this operation, PE j has an array $A[0..M-1]$ of size M. The notation $A[i](j)$ refers to the element $A[i]$ in PE j. In addition, each PE has a value in its I register. After the data accumulation, the M elements of A in each PE j are such that:

$$A[i](j) = I((j + i) \ mod \ P), \ 0 \le i < M, \ 0 \le j < P$$

Data accumulation may be done efficiently by adapting the data circulation algorithm of Figure 2. Procedure ACCUM(A, I, M) can be completed in $2(M-1)+\log(N/M)$ unit routes [RANK87b].

2.3.8. Adjacent Sum

This operation is defined in [PRAS87]. For each PE, p, $0 \leq p < P$, the sum

$$T(p) = \sum_{i=0}^{M-1} A[i]((p+i) \bmod M))$$

is to be computed. Data accumulation may be done efficiently by adapting the data circulation algorithm of Figure 2. The number of unit routes required to complete AdjacentSum(A, M) is $4M-4+2log(P/M)$ [RANK87b].

3. ONE DIMENSIONAL CONVOLUTION

The inputs to the one dimensional convolution problem are vectors $I[0..N-1]$ and $T[0..M-1]$. The output is the vector C1D where:

$$C1D[i] = \sum_{v=0}^{M-1} I[(i+v) \bmod N] * T[v] \quad , 0 \leq v < N$$

We use the computation of C1D as a basic step in our algorithms to compute C2D. In this section, we develop algorithms for C1D. We consider three cases:

(i) Each PE has $O(M)$ memory

(ii) Each PE has $O(log\ M)$ memory

(iii) Each PE has $O(1)$ memory

Our algorithms assume that there are $P = N$ processors and that the vector I is mapped onto the hypercube using the identity mapping (i.e., $I(i)$ on PE i) . Further, we assume that there are (N/M) copies of T in the hypercube with one copy in each block of M processors. Within a block, the mapping of T is this same as that of I.

3.1. O(M) Memory

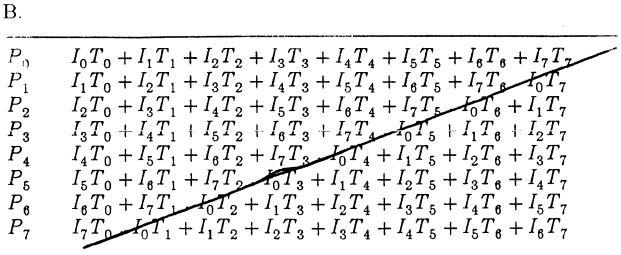
When each processor has $O(M)$ memory, the most effective way to compute C1D is to first perform a data accumulation on I. Following this, each processor has all the I values needed to compute the corresponding entry of C1D. Next, the T values are circulated through each block of M processors. During this circulation, the T values are multiplied by I values and the C1D values computed. Procedure C1D_M (Figure 3) provides the details. The data accumulation takes $2(M-1)+\log(N/M)$ unit routes and the *for* loop requires another $2M$. The total number of unit routes is therefore $4M+\log(N/M)-2$. Note that while the final shift on T is not necessary for the computation of C1D, our algorithms for C2D assume that T is unchanged by the C1D algorithms. This final shift restores the original T values.

```
procedure C1D_M(M)
{ O(M) memory one dimensional convolution}
begin
   ACCUM(A, I, M);
   C1D := 0;
   in := p mod M { in = index of T in processor p}
   for j := 1 to M do
   begin
      C1D := CID + A[in] * T;
      l := f(logM, j);
      T ←T^(l);
      in := in θ 2^l;
   end
end; { of C1D_M}
```

Figure 3: $O(M)$ memory computation of C1D

3.2. O(log M) Memory

Since O(log M) memory is not sufficient to perform a data accumulation, we need to devise another strategy to compute C1D. Following the strategy in procedure AdjacentSum, each PE computes two sums A and B. A is the sum of all terms in the C1D for that processor for which the I values are in the M block containing the processor. B is the sum of all terms in the C1D for the corresponding processor in the previous M block for which the I values are in this M block. Figure 4 shows the components of the A and B sums for each processor in an M block of processors (in the figure, M=8). The processor and I value indexing is relative to the block. The absolute index is obtained by adding Mk, where k is the block index, to the relative index. Values above and including the off diagonal correspond to A while those below correspond to B.



Sums above and including the off diagonal are A
Sums below the off diagonal are B
Figure 4: A and B values to be completed by each PE

A and B can be computed recursively by decomposing a problem of size M into four problems of size $(M/2)$ each as shown in Figure 5. Problems (a) and (c) can be solved in parallel and so also can problems (b) and (d). The algorithm is given in Figure 6.

The number of unit routes required by Recursive C1D is given by the recurrence:

$$routes(M) = \begin{cases} 2routes(M/2)+8 & M>1 \\ 0 & M=1 \end{cases}$$
$$= 8M-8$$

Adding to this the number of unit routes required by the SHIFT(-M, B, p), we get $8M+\log(P/M)+O(1)$ as the number of unit routes required to compute C1D using $O(logM)$ memory,

Note that M invocations of the above algorithm will require $8M^2 + O(MlogN)$ unit routes. In case the image values remain the same for all invocations the $O(MlogN)$ factor can be reduced to $O(logN)$. This is achieved by making each block of size M calculate all the result values itself. A SHIFT(-M, A, P) is performed before invoking C1D_logM. Now each block has all the I values it requires for its convolution values. For every invocation of one dimension convolution two C1D_logM (without *line* 4 and 5) procedures are invoked. The first with the original I values and the second with the new I values recieved (i.e the one from the next block). By adding the A values of the first call with the B values of the second, we get the desired convolution. Thus M invocations will require $16M^2 + O(logP) + O(M)$ unit routes. Further optimization is possible. Notice that if all the I_i terms below the off diagonal of the matrix are replaced by $I_{(i+M) \bmod P}$, then the sum B will represent the values corresponding to its own block. Moreover the I values are not moved in the algorithm. Thus by passing M values along with their index values and by modifying *line* 10

P_0 $I_0T_0 + I_1T_1 + I_2T_2 + I_3T_3$
P_1 $I_1T_0 + I_2T_1 + I_3T_2 \cdot I_0T_3$
P_2 $I_2T_0 + I_3T_1 \cdot I_0T_2 + I_1T_3$
P_3 $I_3T_0 \cdot I_0T_1 + I_1T_2 + I_2T_3$

(a)

P_0 $I_0T_4 + I_1T_5 + I_2T_6 + I_3T_7$
P_1 $I_1T_4 + I_2T_5 + I_3T_6 \cdot I_0T_7$
P_2 $I_2T_4 + I_3T_5 \cdot I_0T_6 + I_1T_7$
P_3 $I_3T_4 \cdot I_0T_5 + I_1T_6 + I_2T_7$

(b)

P_4 $I_4T_4 + I_5T_5 + I_6T_6 + I_7T_7$
P_5 $I_5T_4 + I_6T_5 + I_7T_6 \cdot I_4T_7$
P_6 $I_6T_4 + I_7T_5 \cdot I_4T_6 + I_5T_7$
P_7 $I_7T_4 \cdot I_4T_5 + I_5T_6 + I_6T_7$

(c)

P_4 $I_4T_0 + I_5T_1 + I_6T_2 + I_7T_3$
P_5 $I_5T_0 + I_6T_1 + I_7T_2 \cdot I_4T_3$
P_6 $I_6T_0 + I_7T_1 \cdot I_4T_2 + I_5T_3$
P_7 $I_7T_0 \cdot I_4T_1 + I_5T_2 + I_6T_3$

(d)

Figure 5: 4 decomposed problems with M = 4

appropriately we can make sure that each term is calculated by using either I_i or $I_{(i+M) \bmod P}$. It can be easily shown that $12M^2 + O(logN) + O(M)$ unit routes are required.

3.3. O(1) Memory

First, we develop two data rotation patterns that are needed by our $O(1)$ memory algorithm. The first pattern obtains all circular shifts of even length in the interval $[1, M-1]$. There are exactly $(M/2)-1$ such shifts (recall that $M = 2^m$ is a power of 2). A *shift distance sequence*, E_k, is a sequence $d_1 d_2 \cdots d_{2^{k-1}-1}$ of positive integers such that a clockwise shift of d_1, followed by one of d_2, followed by one of d_3, etc. covers all even length shifts.

Note that $E_0 = E_1 = null$ as there are no even length shifts in the range $[1, 2^m-1]$ when $m = 0$ and 1. $E_2 = 2$. This transforms the length $M = 2^2$ sequence *abcd* into the sequence *cdab*. In general, the choice $E_k = 2, 2, 2, \cdots$ will serve to obtain all even length shifts. From the complexity standpoint this choice is poor as each shift requires $2\log(M/2)$ unit routes. Better performance is obtained by defining

$$E_0 = E_1 = null,\ E_2 = 2$$
$$E_k = InterLeave(E_{k-1},\ 2^{k-1}),\ k \geq 2$$

where InterLeave is an operation that inserts a 2^{k-1} in front of E_{k-1}, at the end of E_{k-1}, and between every pair of adjacent distances in E_{k-1}. Thus,

$$E_3 = Interleave(E_2,\ 4)$$
$$= 4\ 2\ 4$$
$$E_4 = Interleave(E_3,\ 8)$$
$$= 8\ 4\ 8\ 2\ 8\ 4\ 8$$

When a shift sequence E_k is used, the effective shift following d_i is $(\sum_{j=1}^{i} d_j) \bmod 2^k$ Thus when E_3 is used on the sequence

```
line  procedure C1D_logM(M)
1     { O(log M) memory algorithm for C1D}
2     begin
3       RecursiveC1D(M, A, B);
4       SHIFT(B, -M, P);
5       C1D := A + B;
6     end; { of C1D_log M}
7     procedure RecursiveC1D(M, A, B)
8     { compute A and B in M blocks}
9     begin
10      if M=1 then [ A := I*T; B :=0; return ]
11      RecursiveC1D(M/2, A1, B1); {problems (a) and (c)}
12      b := log2M - 1;
13      T(p) ←T(p^(b)); {P0– P3 and P4– P7 exchange T values}
14      RecursiveC1D(M/2, A2, B2); {problems (b) and (d)}
15      T(p) ←T(p^(b)); { restore T values }
16      X := B2; (p_b = 1)
17      X := B1; (p_b = 0)
18      Y := A1; (p_b = 1)
19      Y := A2; (p_b = 0)
20      X ←X^(b); {move partial sums to correct PEs}
21      Y ←Y^(b); {move partial sums to correct PEs}
22      A := A1 + X + Y; (p_b = 0)
23      A := A2 + X + Y; (p_b = 1)
24      B := B2; (p_b = 0)
25      B := B1; (p_b = 1)
26    end; { of Recursive C1D}
```

Figure 6: $O(log\ M)$ memory computation of C1D

$abcdefgh$, we get

d	sequence	effective shift
4	$efghabcd$	4
2	$ghabcdef$	6
4	$cdefghab$	$2 = 10 \bmod 8$

Theorem 1: Let $E[k, i]$ be d_i in the sequence E_k, $k \geq 2$. Let $ESUM[k, i] = (\sum_{j=1}^{i} E[k, j]) \bmod 2^k$. Then $\{ESUM[k, i] \mid 1 \leq i \leq 2^{k-1}-1\} = \{2, 4, 6, 8, ..., 2^k - 2\}$.

Proof: See [RANK87b].

Theorem 2: The shift sequence E_k can be done in $2(2^k - k - 1)$ unit routes, $k \geq 2$.

Proof: See [RANK87b].

The result of the preceding theorem is important as it says that the average cost of rotation in E_k is $\frac{2(2^k - k - 1)}{2^{k-1}-1} < 4$. So, we can perform even length rotations with $O(1)$ average cost. This is crucial to our algorithm.

Let F_k be the sequence obtained by dividing each distance in E_k by 2. So, $F_0 = F_1 = null$, $F_2 = 1, F_3 = 2, 1, 2$, etc.

Theorem 3: Let $FSUM[k, i] = (\sum_{j=1}^{i} F[k, j]) \bmod 2^{k-1}$ where $F[k, j]$ is the j'th distance in F_k

(a) $\{FSUM[k, i] \mid 1 \leq i \leq 2^{k-1}-1\} = \{1, 2, 3, ..., 2^{k-1}-1\}$

(b) All the shifts in F_k can be done in a window of size 2^{k-1} in $2(2^k - k - 1)$ unit routes.

Proof: Similar to the proof of Theorems 1 and 2.

As in our earlier algorithms each PE will compute two quantities A and B. For any PE, A is the sum of all the C1D terms that are in the M block containing the PE. B is the sum of all C1D terms that are needed by the corresponding PE in the previous M block. The terms contributing to A and B are shown in Figure 4. The AB values are computed in two stages. In the first, we compute the contribution to A and B by all I terms I_j for j even. In the next stage, we do this for the case j odd.

Consider the case M=8. If we begin by computing the terms on the major diagonal of Figure 4 , then PEs (0, 1, 2, ,,, 7) compute (I_0T_0, I_2T_1, I_4T_2, I_6T_3, I_0T_4, I_2T_5,I_4T_6, I_6T_7). The I and T values required by each of the 8 PEs are shown in the first two rows of Figure 7. Notice that if we rotate the I values in windows of size 4 by some amount j, then the T values need to be rotated by $2j$ so that each PE has a pair (I, T) whose product is needed in the computating of its A or B value. For this rotation we use the sequences F_3 and E_3. Rotating I by $F[3, 0]$ in size 4 windows and T by $E[3,0]$ in a size 8 window gives the next two rows of Figure 7. The result of performing the remaining rotations is also given in Figure 7. Figure 8 gives the computation of the odd terms.

PE	0	1	2	3	4	5	6	7
I	I_0	I_2	I_4	I_6	I_0	I_2	I_4	I_6
T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7
I	I_4	I_6	I_0	I_2	I_4	I_6	I_0	I_2
T	T_4	T_5	T_6	T_7	T_0	T_1	T_2	T_3
I	I_6	I_0	I_2	I_4	I_6	I_0	I_2	I_4
T	T_6	T_7	T_0	T_1	T_2	T_3	T_4	T_5
I	I_2	I_4	I_6	I_0	I_2	I_4	I_6	I_0
T	T_2	T_3	T_4	T_5	T_6	T_7	T_0	T_1

Figure 7: Computing the even terms

PE	0	1	2	3	4	5	6	7
I	I_1	I_3	I_5	I_7	I_1	I_3	I_5	I_7
T	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_0
I	I_5	I_7	I_1	I_3	I_5	I_7	I_1	I_3
T	T_5	T_6	T_7	T_0	T_1	T_2	T_3	T_4
I	I_7	I_1	I_3	I_5	I_7	I_1	I_3	I_5
T	T_7	T_0	T_1	T_2	T_3	T_4	T_5	T_6
I	I_3	I_5	I_7	I_1	I_3	I_5	I_7	I_1
T	T_3	T_4	T_5	T_6	T_7	T_0	T_1	T_2

Figure 8: Computing the odd terms

The initial configuration for the I's can be obtained by concentrating the even I's using the strategy described in Figure 9 for the case of M = 16. This requires log$M$ unit routes. Let CONCENTRATE(I, M) be the algorithm that does this. The algorithm for one dimensional convolution now takes the form given in Figure 10. Note that the E's and F's are known only to the control unit. These may be computed, on the fly, in linear time using a stack of height $m = logM$. The memory required in each hypercube PE is only $O(1)$. Lines 5 through 15 handle the even terms . Notice that $(CShift + 2p) \bmod M$ gives the index of the I value currently in $C(p)$. So, if this index is less than p the term CD corresponds to the previous block. Otherwise the term CD is for this PE. The fact that each PE always has a C and a D whose product contributes to either A or B follows from the observations that this is so initially and on each iteration, D rotates twice as much as C. The total number of unit routes is $8M + O(logP) + O(logM)$.

Let us consider M invocations of one dimensional convolution with the same I values. By an argument similiar to the one presented in the previous sub-section, M invocations can be completed in $16M^2 + O(logN) + O(MlogM)$. We perform a SHIFT(-M, I, P), followed by *Line* 7, 18, 19 on the old and new I values and store these results for the later invocations. Now by defining C = $\{I_{old}, I_{new}\}$ and modifying steps 11 and 23 so that they calculate terms for this block, we can show that M invocations can be completed in $12M^2 + O(logN) + O(MlogM)$.

4. TWO DIMENSIONAL CONVOLUTION

Assume that $P = N^2$ PEs are available. These may be viewed as an N × N array as described in Section 2. We assume that $I(i, j)$ is initially in the I register of PE(i, j). Further since N and M are assumed to be powers of 2, the N × N array may further be viewed as composed of (N^2/M^2) arrays of size M × M. We assume that T is initially in the top left such array.

4.1. O(M) Memory

When O(M) memory is available, PE(i,j), $1 \leq i \leq N$, $1 \leq j \leq N$ computes M one dimensional convolutions $S[q]$, $0 \leq q < M$ defines as below

$$S[q] = \sum_{r=0}^{M-1} I[(i, (j + r) \bmod N] * T[q, r]$$

Next, C2D is obtained by performing an adjacent sum operation along the columns of the N × N PE array. A high level description of the algorithm is given in Figure 11.

The total number of unit routes is $2M^2 + O(M + \log M)$. The number of unit routes for Steps 1-4 are $\log(N^2/M^2)$, $2M + log(N/M)$, $M(2M + log(N/M)$ and $4M - 4 + 2log(N/M)$ respectively.

Route																
0	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
1	I_0	-	-	I_2	I_4	-	-	I_6	I_8	-	-	I_{10}	I_{12}	-	-	I_{14}
2	I_0	I_2	-	-	-	-	I_4	I_6	I_8	I_{10}	-	-	-	-	I_{12}	I_{14}
3	I_0	I_2	I_4	I_6	-	-	-	-	-	-	-	-	I_8	I_{10}	I_{12}	I_{14}
4	I_0	I_2	I_4	I_6	I_8	I_{10}	I_{12}	I_{14}	I_0	I_2	I_4	I_6	I_8	I_{10}	I_{12}	I_{14}

Figure 9: Initial configuration for even terms

```
line  procedure C1D_1 (M)
1     {O(1) Memory C1D algorithm}
2     begin
3           A := 0; B := 0; m = log M;
4           {even terms}
5           C := I; D:= T;
6           Cshift := 0;
7           CONCENTRATE (C, M);
8           for j := 1 to M/2 do
9           begin
10                A := A + C * D; ((CShift + 2p) mod M ≥ p)
11                B := B + C * D; ((CShift + 2p) mod M <p)
12                SHIFT(C, F[m, j - 1], M/2);
13                CShift := (CShift + F[m, j - 1]) mod (M/2);
14                SHIFT(D, E[m, j - 1], M);
15          end
16          {odd terms}
17          C := I; D := T;
18          SHIFT(C, -1, P); CShift := 1; SHIFT(D, -1, M);
19          CONCENTRATE(C, M);
20          for j := 1 to M/2 do
21          begin
22                A := A + C * D; ((CShift + 2p) mod M ≥ p)
23                B := B + C * D; ((CShift + 2p) mod M <p)
24                SHIFT(C, F[m, j - 1], M/2);
25                CShift := (CShift + F[m, j - 1]) mod (M/2);
26                SHIFT(D, E[m, j - 1], M);
27          end
28          SHIFT(B, -M, P);
29          C1D := A + B;
30    end; { of C1D_1}
```

Figure 10: O(1) memory SIMD C1D algorithm

procedure C2D_M(N, M)
{assumes O(M) memory per PE}

Step1: Broadcast T to all M × M blocks in the N × N PE array

Step2: Perform a data acumulation on I. Now each PE contains the M I values it needs to compute its S(q)'s.

Step3: Compute the S(q)'s. Each S(q) is a one dimensional convolution. However, the data accumulation step of the algorithms of Figure 10 may be omitted as the I values have already been accumulated in Step 2. To go from one S to another, the T values need to be circulated along the columns of each M × M block. This can be done using the data circulation algorithm of Section 2.

Step4: Compute C2D[i, j] = $\sum_{r=0}^{M-1} S[r]((i + r)\ mod\ N, j)$. This is done using the adjacent sum algorithm of Section 2 on the columns of the N × N PE array

end

Figure 11: High level description of two dimensional convolution with each PE having O(M) memory

Now, it is not possible for each PE to accumulate the M values of I it needs from its row. Nor is it possible for a PE to compute the values $S[q]$, $0 \leq q < M$. The new strategy is similar to that used in computing C1D when only O(1) memory is available. We may rewrite the definition of C2D as

$$C2D[i, j] = \sum_{r=0}^{M-1} CXD[i, r, j]$$

where

$$CXD[i, r, j] = \sum_{a=0}^{M-1} I[(i + r)\ mod\ N, (j + a)\ mod\ N]*T[r, a]$$

Since each CXD is a one dimensional convolution, it can be computed using algorithm C1D_1. PE(i, j) computes $E = \sum_{r=0}^{M-1\ mod\ M-1} CXD[i, r\ j]$ and

$F = \sum_{r=M-i \bmod M}^{M-1} CXD[(i-M) \bmod N, r, j]$. Thus each PE computes a value for itself (i.e., E) and a value for the corresponding PE in the adjacent upper M × M block (i.e., F). The F values are then shifted M units along the columns and added to the E values to get the C2D values. A high level description of the algorithm is provided in Figure 12. In iteration k of *Step* 3, the PEs in column j of an M × M PE block compute the CXD terms needed for the E and F of PE($\lfloor i/M \rfloor M + k, j$). Then in *Step* 4, these terms are added together to get the E and F for this PE. The time complexity of Steps 1, 3, 4 and 5 is $log(N^2/M^2)$, $12M + O(logM)$, $3logM$ and $2log(N/M)$ respectively. The total number of unit routes taken is $12M^2 + O(M \log M) + O(logN)$. A slightly more efficient algorithm results if we interpret C2D as:

$$C2D[i, j] = \sum_{r=0}^{M-1} X[i, j, r] * Y[r]$$

where $X[i, j, r]$ is the 1 × M vector $I[(i+r) \bmod N, j .. (j+M-1) \bmod N]$ and $Y[r]$ is the 1 × M vector $T[r, 0 .. M-1]$. Thus C2D is viewed as the one dimensional convolution of X and Y where X and Y are vectors. We can extend algorithm C1D_1 to obtain an algorithm that requires $12M^2 + O(M) + O(logN)$ unit routes and computes this one dimensional convolution. This algorithm is quite a bit more complex than Figure 12 and is omitted.

---

procedure C2D_1(N, M)
{assumes O(1) memory per PE}

Step1: Broadcast T to all M × M blocks in the N × N PE array

Step2: Repeat Steps 3 and 4 for k := 0 to M-1

Step3: Compute $CXD[(\left\lfloor \frac{i}{M} \right\rfloor M + k) \bmod N, i \bmod M - k, j]$ if $i \bmod M \geq k$ using C1D_1(M) and put the result in A, otherwise A = 0;
Compute
$CXD[(\left\lfloor \frac{i}{M} \right\rfloor M + k - M) \bmod N, i \bmod M - k + M, j]$
if $i \bmod M < k$ using C1D_1(M) and put the result in B, otherwise B = 0;

Step4: Use the data sum operation, described in Section 2, to sum the results for the adjacent upper block and itself, by summing up B's and A's in $PE(\left\lfloor \frac{i}{M} \right\rfloor M + k, j)$ in F and E respectively. Shift the T values along columns by 1, using the algorithm of Section 2.

Step5: Shift(F, -M, N) along columns . E := E + F .

Figure 12: High level description of two dimensional convolution with each PE having O(1) memory

---

4.2. O(log M) Memory

The algorithm for two dimensional convolution for Each PE having $O(logM)$ memory is the same as that of Figure 12. The only difference is that the one dimensional convolution used in Step 2 refers to one dimensional convolution with $O(logM)$ memory. The number of unit routes required is $12M^2 + O(M\log M)+O(logN)$.

5. EXTENSIONS

It is easy to see that the algorithms developed for two dimensional convolutions in the previous section can be extended to windows of size $M_1 \times M_2$ (where M_1 and M_2 are powers of 2). Let us consider the case where we have an M × M window and M is not a power of 2. In this case, let $m_l m_{l-1} ..... m_0$ be the binary representaion of M. The M × M T matrix may be viewed as several $2^l \times 2^k$ matrices ($m_l = m_k = 1$). A two dimensional convolution is performed for each such matrix and the results added.

When the number, P, of processors is N^2K^2, $1 \leq K \leq M$ the techniques of [PRAS87] may be used to extend our algorithms to obtain optimal N^2K^2 PE algorithms.

6. CONCLUSION

In this paper, we have presented optimal algorithms for 1-D convolution and Image Template Matching (2-D Convolution). These algorithms use novel strategies to achieve optimal speed-up using $O(M)$, $O(logM)$ and $O(1)$ memory per PE for an M × M Template. Our algorithm for $O(1)$ memory is asymptotically faster then previously known algorithms. Unlike previous algorithms for this problem, our algorithms do not use data broadcasting.

7. REFERENCES

[BALL85] D. H. Ballard and C. M. Brown, "*Computer Vision*", **1985**, Prentice Hall, New Jersey.

[CHAN87] J. H. Chang, O. Ibarra, T. C. Pong, and S. Sohn, "Convolution on a Pyramid Computer", *International Conference on Parallel Processing*, **1987**, pp 780-782.

[DEKE86] E. Dekel, D. Nassimi and S. Sahni, " Parallel matrix and graph algorithms", *SIAM Journal on computing*, **1981**, pp. 657-675.

[FANG85] Z. Fang, X. Li and L. M. Ni, "Parallel Algorithms for Image Template Matching on Hypercube SIMD Computers", *IEEE CAPAMI workshop*, **1985**, pp 33-40.

[FANG86] Z. Fang and L. M. Ni, "Parallel Algorithms for 2-D convolution", *International Conference on Parallel Processing*, **1986**, pp 262-269.

[HORO85] E. Horowitz and S. Sahni, "*Fundamentals of Data Structures in Pascal*", Computer Science Press, **1985**.

[KUNG82] H. T. Kung and S. W. Song, "A Systolic 2-D Convolution Chip", *Multicomputers and Image Processing: Algorithms and Programs, editors: Preston and Uhr (Academic Press, New York)*, **1982**, pp 373-384.

[LEE87] S. Y. Lee and J. K. Aggarwal, "Parallel 2-D convolution on a mesh connected array processor", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **July 1987**, pp 590-594.

[MARE86] M. Maresca and H.Li, "Morphological Operations on Mesh-connected Architecture : A generalized convolution Algorithm", *Proceedings of 1986 IEEE Computer Society Workshop on Computer Vision and Pattern Recognition* ,**1986**, pp 299-304.

[PRAS87] V. K. Prasanna Kumar and V. Krishnan, "Efficient Image Template Matching on SIMD Hypercube Machines", *International Conference on Parallel Processing*, **1987**, pp 765-771.

[RANK87a] S. Ranka and S. Sahni, "Convolution on an SIMD mesh-connected computer", *University of Minnesota Tech. Report*, **1987**.

[RANK87b] S. Ranka and S. Sahni, "Image Template Matching on SIMD hypercube multicomputers", *University of Minnesota Tech. Report,* **1987**.

[RANK88] S. Ranka and S. Sahni, "Image Template Matching on MIMD hypercube multicomputers", *University of Minnesota Tech. Report,* **1987**.

[ROSE82] A. Rosenfeld and A. C. Kak, "*Digital Picture Processing*", Academic Press, **1982**

IMAGE TEMPLATE MATCHING ON MIMD HYPERCUBE MULTICOMPUTERS [+]

Sanjay Ranka and Sartaj Sahni

University of Minnesota

Abstract

Efficient algorithms for image template matching on fine grained as well as medium grained MIMD hypercube multicomputers are developed. Template matching algorithms for MIMD hypercube multicomputers have not, to our knowledge, been previously developed. The medium grained MIMD algorithm is developed specifically for the NCUBE multicomputer. This algorithm is compared experimentally with an algorithm that is optimized for the CRAY2 supercomputer. In addition, customized algorithms are developed for Kirsch templates.

1. INTRODUCTION

The inputs to the image template matching problem are an $N\times N$ image matrix $I[0..N-1, 0..N-1]$ and an $M\times M$ template $T[0..M-1, 0..M-1]$. The output is an $N\times N$ matrix C2D where

$$C2D[i,j] = \sum_{u=0}^{M-1}\sum_{v=0}^{M-1} I[(i+u)\ mod\ N, (j+v)\ mod\ N] * T[u,v],$$
$$0 \le i, j < N$$

$C2D$ is called the two dimensional convolution of I and T. Template matching, i.e., computing $C2D$, is a fundamental operation in computer vision and image processing. It is often used for edge and object detection; filtering; and image registration [ROSE82, BALL85]. Because of the fundamental nature of this problem and because of its high complexity ($O(M^2N^2)$ on a single processor computer), much attention has been devoted to the development of efficient fine grain multicomputer parallel algorithms. For example, Chang, Ibarra, Pong and Sohn [CHAN87] have studied this problem on an SIMD pyramid computer; Ranka and Sahni [RANK87a], Maresca and Li [MARE86], and Lee and Agarwal [LEE87] have considered mesh connected computers; and Fang, Li and Ni [FANG85], Fang and Ni [FANG86], Prasanna Kumar and Krishnan [PRAS86] and Ranka and Sahni [RANK87b] have considered SIMD hypercube multicomputers.

In this paper, we restrict our attention to MIMD hypercube multicomputers. We assume that M and N are powers of 2 and develop two asymptotically optimal algorithms that require N^2 processors. These require $O(M)$ and $O(1)$ memory per processor, respectively. The algorithms developed in [PRAS87], [LEE87], [FANG85], [FANG86], and [MARE86] require a broadcast capability. Our algorithms do not require this. Using the techniques of [PRAS87], both our algorithms may be generalized to obtain asymptotically optimal algorithms for MIMD hypercube computers with N^2K^2, $1\le K\le M$ processors and $O(M/K)$ and $O(1)$ memory per processor respectively. Our medium grain MIMD algorithm is developed for the NCUBE hypercube. This algorithm is evaluated experimentally and a comparison with a single processor CRAY2 is made.

Section 2 describes our hypercube model. In addition, notation and some fundamental data movement operations are developed in this section. In Section 3, we develop fine grained algorithms for one dimensional convolution. These form a basic component of our two dimensional convolution algorithms which are developed in Section 4. Section 5 considers Kirsch templates and in Section 6, the medium grain MIMD algorithm for the NCUBE together with one for the CRAY2 supercomputer are developed. Experimental results are also presented in this section.

[+] This research was supported in part by the National Science Foundation under grants DCR84-20935 and MIP 86-17374

2. PRELIMINARIES

2.1. Hypercube Multicomputer

The important features of an MIMD hypercube and the programming notation we use are:

1. There are $P=2^p$ processing elements connected together via a hypercube interconnection network (to be described later). Each PE has a unique index in the range $[0, 2^p-1]$. The local memory of each PE holds both the data and the program that the PE is to execute. Throughout this paper, we shall uses brackets([ ]) to index an array and parentheses('()') to index the PEs. Thus A[i] refers to i'th element of the array A while A(i) refers to the A register of PE i. Likewise $A[i](j)$ refers to the i'th element of array A of PE j.
2. A p dimensional hypercube network connects 2^p PEs. Let $i_{p-1}i_{p-2}\ldots i_0$ be the binary representation of the PE index i. Let $\overline{i_k}$ be the complement of bit i_k. A hypercube network directly connects pairs of processors whose indices differ in exactly one bit. I.e., processor $i_{p-1}i_{p-2}\ldots i_0$ is connected to processors $i_{p-1}\cdots\overline{i_k}\ldots i_0$, $0\le k\le p-1$. We use the notation $i^{(b)}$ to represent the number that differs from i in exactly bit b.
3. At any given instance, different PEs may execute different instructions. In particular, PE i may transfer data to PE $i^{(b)}$, while PE j simultaneously transfers data to PE $j^{(a)}$, $a\ne b$.
4. An *instruction mask* is a boolean function used to describe PEs which will remain active during an instruction. For example, in the instruction
   $A(i) := A(i)+1, \quad (i_0=1)$
   $(i_0=1)$ is a mask, which states that only PEs with index bit 0 equal to 1 remain active during the instruction. I.e., odd indexed PEs increment their A register value by 1. We shall often omit the PE index from our instructions. Thus, the above statement can also be written as
   $A := A+1, \quad (i_0=1)$
5. Interprocessor assignments are denoted using the symbol $\leftarrow$, while intraprocessor assignments are denoted using the symbol :=. Thus the assignment statement:
   $B(i^{(2)}) \leftarrow B(i), \quad (i_2=0)$
   is executed only by the processors with bit 2 equal to 0. These processors transmit their B register data to the corresponding processors with bit 2 equal to 1.
6. In a *unit route*, data may be transmitted from one processor to another to which it is directly connected. We assume that the links in the interconnection network are unidirectional. Hence at any given time, data can be transferred either from PE i $(i_b=0)$ to PE $i^{(b)}$ or from PE i $(i_b=1)$ to PE $i^{(b)}$. Hence the instruction.
   $B(i^{(2)}) \leftarrow B(i), (i_2=0)$
   takes one unit route, while the instruction:
   $B(i^{(2)}) \leftarrow B(i)$
   takes two unit routes.
7. Since the asymptotic complexity of all our algorithms is determined by the number of unit routes, our complexity analysis will count only these.

2.2. Hypercube Embedding of a Grid

Figure 1 shows an embedding of a 4×4 image grid into a hypercube of dimension 4. The number inside a box is the binary representation of the index of the PE to which that element is mapped. The embedding of Figure 1 uses the binary reflected gray code mapping of [CHAN86]. An i bit binary gray code S_i is defined recursively as below:

$S_1 = 0, 1; \quad S_k = 0[S_{k-1}], 1[S_{k-1}]^R$

where $[S_{k-1}]^R$ is the reverse of the $k-1$ bit code S_{k-1} and $b[S]$ is obtained from S by prefixing b to each entry of S. So, $S_2 = 00, 01, 11, 10$ and $S_3 = 000, 001, 011, 010, 110, 111, 101, 100$.

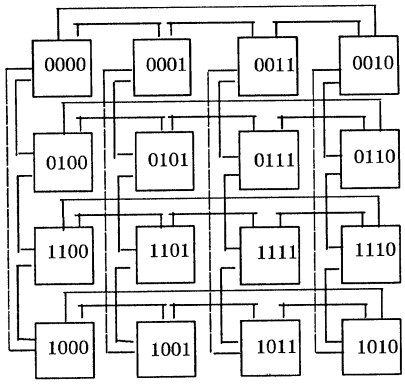


Figure 1: Embedding of a 4 × 4 mesh in a hypercube of dimension 4

If $N = 2^n$, then S_{2n} is used to map an N × N grid into a $P = N^2$ hypercube. The elements of S_{2n} are assigned to the elements of the $N \times N$ grid in a snake like row major order [THOM77]. This mapping has the property that grid elements that are neighbors are assigned to neighboring hypercube nodes. Another interesting property is evident from the definition of S_k and the linear drawing of Figure 2. In this figure, PEs appear in the order given by S_k. A hypercube has circular lists of length 2^i for all i embedded in it. Furthermore, these circular lists of length 2^i are present in every row and column of the grid embedding. Also, the PEs in each circular list of length 2^i form a 2^i PE subhypercube, $i > 1$.

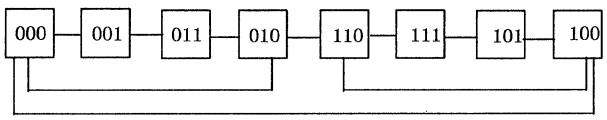


Figure 2: Rings of size 2, 4 and 8 in an 8 PE hypercube

For a hypercube with $P = 2^p$ PEs, we define the function $gray(i)$ such that $gray(0) = 0$ and $gray(i)$ is the index of the PE that immediately follows the PE $gray(i-1)$ in the circular list of size 2^p obtained from the above gray code embedding. For the example of Figure 2, $P = 2^3 = 8$, $gray(0..7) = (0, 1, 3, 2, 6, 7, 5, 4)$. The function $igray$ is the inverse of $gray$. So, $igray(0 .. 7) = (0, 1, 3, 2, 7, 6, 4, 5)$. pp

2.3. Basic Data Manipulation Operations

2.3.1. Data Circulation

Consider a $P = 2^p$ processor hypercube. We are required to circulate the data in the A register of these PEs so that this data visits each of the P processors exactly once. This operation is easy to accomplish on an MIMD hypercube using the binary gray code mapping and the observation that we have a P processor circular list (Figure 2). We simply shift the contents of the A registers circularly clockwise by 1 each time (Figure 3). Procedure CIRCULATE circulates the A register data through the P processors in $O(P)$ time. This is trivially optimal.

```
procedure CIRCULATE(A);
  {data circulation}
  for i = 1 to P - 1 do
      A(gray(j)) ←A(gray((j + 1) mod P));
end
```

Figure 3: Data circulation in an MIMD hypercube

2.3.2. Data Broadcast

In a data broadcast, data originates at one PE and is to be transmitted to the remaining $P-1$ PEs. This can be done in $\log P$ unit routes using a tree broadcast scheme [DEKE81].

2.3.3. Window Broadcast

Assume that W is a power of 2 and that a P processor hypercube is tiled by windows of this size such that each window forms a subhypercube with W PEs. In a window broadcast, data originates in one of these windows (different data in different PEs of the window). The data in this window is to be copied to the remaining $P/W - 1$ windows. This copying can be done using $\log(P/W)+2$ unit routes [DEKE81].

2.3.4. Data Sum

Assume the window tiling of Section 2.3.3.. The data in each of the windows is to be summed and the sum left in a prespecified PE (same relative PE for each window). For example, if we are summing the A register data, we may be required to compute:

$$Sum(index(iW)) = \sum_{j=0}^{W-1} A(index(iW+j)), \qquad 0 \le i < (P/W)$$

Here, $index(q)$ gives the physical index of the q'th PE in the tiling scheme. We assume that the P PEs are first ordered (for example using an S_r) and then tiled using 1 × W tiles. Thus, $iW + j$ is the j'th PE in the i'th tile and iW is the 0'th PE in the i'th tile. Data sum can be done in log W unit routes [DEKE81].

2.3.5. Shift

$SHIFT(A, i, W)$ shifts the A register data circularly counterclockwise by i in windows of size W. I.e., $A(gray(qW+j))$ is replaced by $A(gray(qW + (j-i) \bmod W))$, $0 \le q < (P/W)$, $0 \le j < W$. In a gray code indexing, the indexing within each size 2^j window also corresponds to a gray code (consider the least significant j bits). Hence each pair of adjacent size 2^j windows differs in exactly one bit. Now suppose the shift amount i is a power of 2. We can get data to the correct size i window by routing along the single bit in which adjacent size i windows differ

Following this, the data in each size i window needs to be reversed (unless $i = 1$). This reversal may be accomplished by exchanging data in the two size $i/2$ windows that make up a size i window. The total number of unit routes required when i is a power of 2 is therefore at most 2 to get the data to the correct size i window (note that 2 routes are needed when $i = W/2$ and one otherwise) plus at most 2 to reverse within the size i window. Hence at most 4 unit routes are needed to perform a shift of size i. When i is not a power of 2, i can be written as the sum of powers of 2 and the shift obtained by performing successive power of 2 shifts. Since only one of these can be a $W/2$ shift, the number of unit routes is at most $3\#1(i) + 1$, where $\#1(i)$ is the number of ones in the binary representation of i. The worst case performance can be kept at $3(logW)/2 + 1$ by noting that if there are more than $(logW)/2$ one bits, we can do a $W-1-i$ clockwise shift followed by a unit clockwise shift. Also note that the special cases of $i = 1$, 2, and 3 are easily done in i unit routes unless $W = 1$ (in this case, a shift of 1 takes 2 unit routes).

2.3.6. Data Accumulation

For this operation, PE j has an array $A[0..M-1]$ of size M. In addition, each PE has a value in its I register. After the data accumulation, the M elements of A in each PE j are such that:

$$A[i](gray(j)) = I(gray((j+i) \bmod P)),\ 0 \le i < M,\ 0 \le j < P$$

This can be accomplished in M-1 unit routes (for P > 2) by repeatedly shifting by -1 in windows of size P. The algorithm is given in Figure 4.

```
procedure ACCUM(A, I, M)
{each PE accumulates in A, the I values of the next M PEs
 including itself}
begin
    A[0] := I;
    for i := 1 to M-1 do
    begin
      SHIFT(I, -1, P);
      A[i] := I;
    end
end {ACCUM}
```

Figure 4: Data accumulation

2.3.7. Adjacent Sum

This operation is defined in [PRAS87]. For each PE, p, $0 \le p < P$, the sum

$$T(gray(p)) = \sum_{i=0}^{M-1} A[i](gray(p+i) \bmod M))$$

is to be computed.

As mentioned earlier, every hypercube of size P can be viewed as consisting of P/M subhypercubes (blocks) each of size M. For every PE p, some (or all) of the A's needed to compute $T(gray(p))$ are in the block containing PE p.The remainder are in the next block of PEs. The strategy to compute T is as follows:

1) Each PE, p, begins with two variables S and T (initially 0). These values circulate through the M PEs in the block. T accumulates the A values in the block needed in the sum for $T(gray(p))$. S accumulates the A values needed for $T(gray((p-M) \bmod P))$.

2) The S values are shifted clockwise by M positions and added to the T values.

The formal algorithm is given in Figure 5. The number of unit routes is $2M + 4$ (recall that M is a power of 2 and a power 2 shift takes at most 4 unit routes). This can be reduced to $M + 4$ by shifting S and T as a single packet.

```
procedure AdjacentSum(A, M)
begin
    S:= 0; T:=0;
    for i := 0 to M-1 do
    begin
       T(p) := T(p) + A[i](p); (igray(p) mod M ≥ i)
       S(p): = S(p) + A[i](p); (igray(p) mod M <i)
       SHIFT(T, 1, M);
       SHIFT(S, 1, M);
    end
    SHIFT(S, -M, P);
    T := T + S;
end {of AdjacentSum}
```

Figure 5: Adjacent Sum

3. ONE DIMENSIONAL CONVOLUTION

The inputs to the one dimensional convolution problem are vectors $I[0..N-1]$ and $T[0..M-1]$. The output is the vector C1D where:

$$C1D[i] = \sum_{v=0}^{M-1} I[(i+v) \bmod N] * T[v] \quad ,\ 0 \le i < N$$

We use the computation of C1D as a basic step in our algorithms to compute C2D. In this section, we develop algorithms for C1D. We consider two cases:

(i) Each PE has $O(M)$ memory

(ii) Each PE has $O(1)$ memory

Our algorithms assume that there are $P = N$ processors and that the vector I is mapped onto the hypercube using the gray code mapping (i.e., $I[i]$ on PE $gray(i)$) . Further, we assume that there are N/M copies of T in the hypercube with one copy in each block of M processors. Within a block, the mapping of T is this same as that of I.

3.1. O(M) Memory

When each processor has $O(M)$ memory, the most effective way to compute C1D is to first perform a data accumulation on I. Following this, each processor has all the I values needed to compute the corresponding entry of C1D. Next, the T values are circulated through each block of M processors. During this circulation, the T values are multiplied by the I values and the C1D values computed. Procedure C1D_M (Figure 6) provides the details. Note that while the final shift on T is not necessary for the computation of C1D, our algorithms for C2D assume that T is unchanged by the C1D algorithms. This final shift restores the original T values. The number of unit routes taken is 2M.

```
procedure C1D_M
{O(M) memory algorithm for one dimensional convolution}
begin
        ACCUM(A, I, M);
        b := igray(p) mod M; {relative index of PE in M block}
        C1D := 0;
        for j := 1 to M do
        begin
                C1D := C1D + A[b] * T;
                b := (b+1) mod M;
                SHIFT(T, -1, M);
        end
end; {of C1D_M}
```

Figure 6: $O(M)$ memory computation of C1D

3.2. O(1) Memory

When only O(1) memory per PE is available, we begin by first pairing I values in the processors. The pair in processor p is (A(p), B(p)) = (I $[(jM + 2k)\ mod\ N]$, I$[(jM+2k+1)\ mod\ N]$) where $i = igray(p)$, $j = \lfloor i/M \rfloor$, and $k = i\ mod\ M$. Figure 7 gives the initial AB pairs in each PE for the case $N = 16$, $M = 4$. The algorithm to obtain this is given in Figure 8.

p	i=$igray(p)$	j	k	I	AB
0	0	0	0	I_0	I_0I_1
1	1	0	1	I_1	I_2I_3
3	2	0	2	I_2	I_4I_5
2	3	0	3	I_3	I_6I_7
6	4	1	0	I_4	I_4I_5
7	5	1	1	I_5	I_6I_7
5	6	1	2	I_6	I_8I_9
4	7	1	3	I_7	$I_{10}I_{11}$
12	8	2	0	I_8	I_8I_9
13	9	2	1	I_9	$I_{10}I_{11}$
15	10	2	2	I_{10}	$I_{12}I_{13}$
14	11	2	3	I_{11}	$I_{14}I_{15}$
10	12	3	0	I_{12}	$I_{12}I_{13}$
11	13	3	1	I_{13}	$I_{14}I_{15}$
9	14	3	2	I_{14}	I_0I_1
8	15	3	3	I_{15}	I_2I_3

$I_q = I[q]$

Figure 7: Initial AB pairs for N = 16, M = 4

```
procedure PAIRING(M)
{pairing I values in AB registers}
begin
    i := igray(p); {p is processor index}
    B := I;
    SHIFT(B, -1, P);
    A := I;
    for j := 1 to logM-1 do
    begin
        C := B; SHIFT(B, -2^(j-1), M);
        B := C; (i_j = 0)
        C := A; SHIFT(A, -2^(j-1), M);
        A := C; (i_j = 0)
    end
    C := B; SHIFT(B, -(M/2), P);
    B := C; (i_(logM-1) = 0)
    C := A; SHIFT(A, -(M/2), P);
    A := C; (i_(logM-1) = 0)
end; {of PAIRING}
```

Figure 8: Pairing of the I's

The number of unit routes is at most 8 log M. This can be reduced to 4 logM by routing (A, B) pairs as single packets.

Once the AB pairing has been done C1D may be computed by rotating the AB values clockwise in a window of size P (in a single rotation, B's move to A's in the same PE and A's move to B's of the next PE) and rotating the T values clockwise in a window of size M. Figure 9 shows the initial AB pairs and T values for the case N = 16 and M = 4. Throughout the algorithm, the product of A(p) and T(p) will give one of the terms needed to compute C1D($igray$(p)) for every PE p. B(p) will be the next I value needed. Initially, this is true for all processes except those with $igray(p)\ mod\ M = M - 1$. This situation is remedied by replacing B with I in these processors to get the first row labeled AB'. Following a rotation of AB, we get the second row labeled AB. Now, the B value in processors with $igray(p)\ mod\ M = M - 2$ needs to be changed to $I(p)$. With this insight, one arrives at the algorithm of Figure 10. Its correctness is easily established. The number of unit routes (including those for pairing) is at most $M + 8logM$.

4. TWO DIMENSIONAL CONVOLUTION

Assume that $P = N^2$ PEs are available. These may be viewed as an N × N array as described in Section 2. We use (i, j) to refer to the PE in position(i, j) of the N × N array. Thus, for the example of Figure 2, PE(0, 0) is PE 0, PE(2, 3) is PE 7, and PE(3, 3) is PE 6. The index of PE(i, j) is $gray(iN + j)$ if i is even and $gray(iN + N - 1 - j)$ if i is odd. This corresponds to the snake like row major interpretation. We assume that $I[i, j]$ is initially in the I register of PE(i, j). Further since N and M are assumed to be powers of 2, the N × N array may further be viewed as composed of N^2/M^2 arrays of size M × M . We assume that T is initially in the top left such array.

4.1. O(M) Memory

When O(M) memory is available, PE(i,j), $0 \le i < N$, $0 \le j < N$ computes M one dimensional convolutions $S(q)$, $0 \le q < M$ defined as below

$$S(q) = \sum_{r=0}^{M-1} I((i, (j + r)\ mod\ N) * T(q, r)$$

Next, C2D is obtained by performing an adjacent sum operation along the columns of the N × N PE array. A high level description of the algorithm is given in Figure 11. The total number of unit routes is $M^2 + O(M\log M)$. The number of unit routes for each of the steps of Figure 11 is $\log(N/M) + 4$, $M-1$, $M(M - 1)$ and $2M + 4$ respectively.

4.2. O(1) Memory

Now, it is not possible for each PE to accumulate the M values of I it needs from its row. Nor is it possible for a PE to compute the values $S(q)$, $0 \le q < M$. We may rewrite the definition of C2D as

$$C2D[i, j] = \sum_{r=0}^{M-1} CXD[i, r, j]$$

where

$$CXD[i, r, j] = \sum_{a=0}^{M-1} I[(i + r)\ mod\ N, (j + a)\ mod\ N] * T[r, a]$$

Some of the CXD terms needed for the computation of $C2D(i, j)$ can be computed within the M × M PE block that contains $PE(i, j)$ as all the needed I and T values are in the block. The remaining terms can be computed by the corresponding PE in the next lower M × M block as this block contains the needed I values. Thus each PE computes an E value (for itself) and an F value (for the corresponding PE in the adjacent upper M × M block).

The E and F values are computed in k iterations. During iteration k, the PEs in the k'th row of each M × M PE block compute their E and F values. These rows have index k, $M+k$, $2M+k$, · · · . Also

$$E(aM + k, j) = \sum_{r=0}^{M-1-k} CXD[aM + k, r, j] \text{ and}$$

$$F(aM + k, j) = \sum_{r=M-k}^{M-1} CXD[((a-1)M + k)\ mod\ N, r, j].$$

For this, we note that PE(i, j) is in the $i\ mod\ M$ row of the $\lfloor i/M \rfloor$'th M × M block. So, each PE needs to compute
$A = CXD[\lfloor i/M \rfloor M + k, i\ mod\ M - k, j]$ if $i\ mod\ M \ge k$ and
$B = CXD[\lfloor i/M \rfloor M + k - M, i\ mod\ M - k + M, j]$ if $i\ mod\ M < k$

i	I	AB	T	AB'	AB	T	AB'	AB	T	AB'	AB	T	AB'
0	I_0	I_0I_1	T_0	I_0I_1	I_1I_2	T_1	I_1I_2	I_2I_3	T_2	I_2I_3	I_3I_4	T_3	I_3I_0
1	I_1	I_2I_3	T_1	I_2I_3	I_3I_4	T_2	I_3I_4	I_4I_5	T_3	I_4I_1	I_1I_2	T_0	I_1I_2
2	I_2	I_4I_5	T_2	I_4I_5	I_5I_6	T_3	I_5I_2	I_2I_3	T_0	I_2I_3	I_3I_4	T_1	I_3I_4
3	I_3	I_6I_7	T_3	I_6I_3	I_3I_4	T_0	I_3I_4	I_4I_5	T_1	I_4I_5	I_5I_6	T_2	I_5I_6
4	I_4	I_4I_5	T_0	I_4I_5	I_5I_6	T_1	I_5I_6	I_6I_7	T_2	I_6I_7	I_7I_8	T_3	I_7I_4
5	I_5	I_6I_7	T_1	I_6I_7	I_7I_8	T_2	I_7I_8	I_8I_9	T_3	I_8I_5	I_5I_6	T_0	I_5I_6
6	I_6	I_8I_9	T_2	I_8I_9	I_9I_{10}	T_3	I_9I_6	I_6I_7	T_0	I_6I_7	I_7I_8	T_1	I_7I_8
7	I_7	$I_{10}I_{11}$	T_3	$I_{10}I_7$	I_7I_8	T_0	I_7I_8	I_8I_9	T_1	I_8I_9	I_9I_{10}	T_2	I_9I_{10}
8	I_8	I_8I_9	T_0	I_8I_9	I_9I_{10}	T_1	I_9I_{10}	$I_{10}I_{11}$	T_2	$I_{10}I_{11}$	$I_{11}I_{12}$	T_3	$I_{11}I_8$
9	I_9	$I_{10}I_{11}$	T_1	$I_{10}I_{11}$	$I_{11}I_{12}$	T_2	$I_{11}I_{12}$	$I_{12}I_{13}$	T_3	$I_{12}I_9$	I_9I_{10}	T_0	I_9I_{10}
10	I_{10}	$I_{12}I_{13}$	T_2	$I_{12}I_{13}$	$I_{13}I_{14}$	T_3	$I_{13}I_{10}$	$I_{10}I_{11}$	T_0	$I_{10}I_{11}$	$I_{11}I_{12}$	T_1	$I_{11}I_{12}$
11	I_{11}	$I_{14}I_{15}$	T_3	$I_{14}I_{11}$	$I_{11}I_{12}$	T_0	$I_{11}I_{12}$	$I_{12}I_{13}$	T_1	$I_{12}I_{13}$	$I_{13}I_{14}$	T_2	$I_{13}I_{14}$
12	I_{12}	$I_{12}I_{13}$	T_0	$I_{12}I_{13}$	$I_{13}I_{14}$	T_1	$I_{13}I_{14}$	$I_{14}I_{15}$	T_2	$I_{14}I_{15}$	$I_{15}I_0$	T_3	$I_{15}I_{12}$
13	I_{13}	$I_{14}I_{15}$	T_1	$I_{14}I_{15}$	$I_{15}I_0$	T_2	$I_{15}I_0$	I_0I_1	T_3	I_0I_{13}	$I_{13}I_{14}$	T_0	$I_{13}I_{14}$
14	I_{14}	I_0I_1	T_2	I_0I_1	I_1I_2	T_3	I_1I_{14}	$I_{14}I_{15}$	T_0	$I_{14}I_{15}$	$I_{15}I_0$	T_1	$I_{15}I_0$
15	I_{15}	I_2I_3	T_3	I_2I_{15}	$I_{15}I_0$	T_0	$I_{15}I_0$	I_0I_1	T_1	I_0I_1	I_1I_2	T_2	I_1I_2

Figure 9: Execution Trace N = 16, M = 4

procedure C1D_1(M)
{O(1) memory one dimensional convolution}
begin
    PAIRING(M);
    C1D := 0;
    for j :=0 *to* M-1 *do*
    begin
        B(p) := I(p); (*igray*(p) *mod* M = M -1 -j)
        C1D := C1D + A * T;
        SHIFT(A, -1, P);
        C := B; B := A; A := C; { interchange A and B}
        SHIFT(T, -1, M);
    end
end {of C1D_1}

Figure 10: $O(1)$ memory computation of C1D

procedure C2D_M(N, M)

{assumes O(M) memory per PE}

Step1: Broadcast T to all M × M blocks in the N × N PE array

Step2: Perform a data accumulation on I. For this operation, the N × N PE array is viewed as N independent hypercubes with each row forming one such hypercube. Following the operation, each PE contains the M I values it needs to compute its S(q)'s.

Step3: Compute the S(q)'s. Each S(q) is a one dimensional convolution. However, the data accumulation step of the algorithm of Figure 9 may be omitted as the I values have already been accumulated in Step 2. To go from one S to another, the T values need to be circulated along the columns of each M × M block. This can be done using the data circulation algorithm of Section 2.

Step4: Compute $C2D(i, j) = \sum_{r=0}^{M-1} S[r]((i + r) \ mod \ N, j)$. This is done using the adjacent sum algorithm of Section 2 on the columns of the N × N PE array

end

Figure 11: High level description of two dimensional convolution with each PE having O(M) Memory

Then, the PEs in rows $aM + k$, $0 \leq a < N/M$ can compute E and F by summing the As and Bs in their column and in their M × M block. Once this has been done, C2D is computed by shifting the F's up the columns by M units and adding to the E's. A high level description of the algorithm is provided in Figure 12. The number of unit routes for Steps 1, 3, 4 and 5 of Figure 12 is $3log(N/M) + 2$, $2M + O(logM)$, $logM + O(1)$ and 4 respectively. The total number of unit routes is $2M^2 + O(M \log M)$.

procedure C2D_1(N, M)

{assumes O(1) memory per PE}

Step1: Broadcast T to all M × M blocks in the N × N PE array

Step2: Repeat Steps 3 and 4 for k := 0 to M-1

Step3: PE(i, j) computes $CXD[\left\lfloor \frac{i}{M} \right\rfloor M + k, \ i \ mod \ M - k, \ j]$ if $i \ mod \ M \geq k$ using C1D_1(M) and puts the result in A, otherwise A = 0;
PE(i, j) computes $CXD[(\left\lfloor \frac{i}{M} \right\rfloor M + k - M) \ mod \ N, \ i \ mod \ M - k + M, \ j]$ if $i \ mod \ M < k$ using C1D_1(M) and puts the result in B, otherwise B = 0;

Step4: Use the data sum operation, described in Section 2, to sum the B's and A's in $PE(\left\lfloor \frac{i}{M} \right\rfloor M + k, \ j)$ in F and E respectively. Shift the T values up the columns by 1.

Step5: SHIFT(F, -M, N) along columns . C2D := E + F .

Figure 12: High level description of two dimensional convolution with each PE having O(M) Memory

5. KIRSCH MOTIVATED TEMPLATES

Kirsch templates, [BALL85] are commonly used in image processing. Kirsch templates of size 1 (M = 3) and 2 (M = 5) are shown in Figure 13.

By exploiting the special structure of these templates, template matching can be done more efficiently. A high level description of the algorithm is given in Figure 14. Its complexity is O(M). The amount of memory required per PE is O(M). While efficient $O(1)$ memory algorithms can also be developed, we shall not do this here as Kirsch templates usually have small M and it is reasonable to assume this much memory is available.

Steps 3, 4, 5, 6 can be done efficiently by a simple adaptation of procedure AdjacentSum of Section 2.

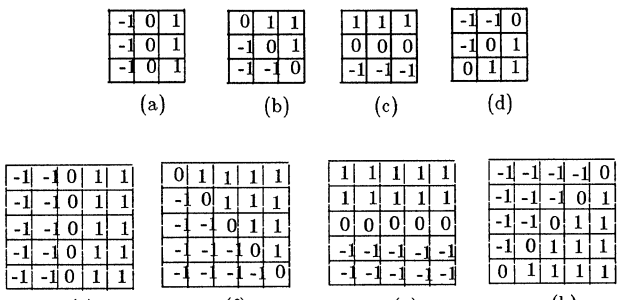


Figure 13: Kirsch templates of size 1 (M = 3) and 2 (M = 5)

Step1: ACCUM(A, I, M);
Step2: B[-1] := 0; C[-1] := 0;
for i := 0 *to* M-1 *do*
begin
B[i] := A[i] + B[i-1];
C[i] := A[M-1-i] + C[i-1]
end;
Do exactly one of the following steps depending on the template type.
Step3: { Templates of types (a) and (e)}

$$C2D(i,j) = \sum_{a=0}^{M-1} (C[(M-3)/2] - B[(M-3)/2])((i+a) \bmod N, j)$$

Step4: { Templates of types (b) and (f)}

$$C2D(i,j) = \sum_{a=0}^{M-1} (C[M-2-a] - B[a-1])((i+a) \bmod N, j)$$

Step5: { Templates of types (c) and (g)}

$$C2D(i,j) = \sum_{a=0}^{(M-3)/2} C[M-1]((i+a) \bmod N, j) - \sum_{(M+1)/2}^{(M-1)} C[M-1]((i+a) \bmod N, j)$$

Step6: { Templates of types (d) and (h)}

$$C2D(i,j) = \sum_{a=0}^{M-1} (C[a-1] - B[M-2-a])((i+a) \bmod N, j)$$

Figure 14: Algorithm for Kirsch templates of Figure 13

6. MEDIUM GRAIN TEMPLATE MATCHING

In the previous sections we have developed algorithms to perform template matching on a fine grain hypercube. Such a computer has the property that the cost of interprocessor communication is comparable to that of a basic arithmetic operation. In this section, we shall consider the template matching problem on a a hypercube in which interprocessor communication is relatively expensive and the number of processors is small relative to the image size n. In particular we shall experiment with an NCUBE/7 hypercube which is capable of having up to 128 processors. The NCUBE/7 available to us, however, has only 64 processors. The time to perform a two byte integer addition on each hypercube processor is 4.3 microseconds whereas the time to communicate b bytes to a neighbor processor is approximately $447 + 2.4b$ microseconds.

Several cases of the template matching problem can be studied. These vary in the initial location of the image and the template and the final location of the convolution (result matrix). We consider the following cases . In all of these, the template is initially in the host.

1. Host-to-host: The image is in the host initially and the result is to be left in the host also.
2. Hypercube-to-host: The image is initially in the host but the result is left in the hypercube.
3. Hypercube-to-hypercube: The image is initially in the hypercube and the convolution is to be left there too.

Let p be the number of hypercube processors. We assume that p is a perfect square and that $\sqrt{p}$ divides n. Hence, the hypercube may be visualized as a $\sqrt{p} \times \sqrt{p}$ mesh and the $n \times n$ convolution matrix can be mapped onto this with each processor getting an $n/\sqrt{p} \times n/\sqrt{p}$ block. We assume that each processor has enough memory to hold one copy of the $m \times m$ template. As far as mapping the $n \times n$ image is concerned, we consider the two possibilities:

(1) Overlap Mapping: In this, each processor gets enough of the image to compute all its convolution values. Hence, the processor in position $(0, 0)$ of the mesh gets $I[0 .. n/\sqrt{p} + m - 2, 0 .. n/\sqrt{p} + m - 2]$.

(2) Nonoverlap Mapping: The image is decomposed into $n/\sqrt{p} \times n/\sqrt{p}$ blocks. This is done in the same way as the convolution decomposition. Each processor gets the image block that corresponds to its convolution block.

Notice that if overlap mapping is used, then the host must transfer more data to each hypercube processor than when the nonoverlap mapping is used. However, no interprocessor communication is needed when the overlap mapping is used. Interprocessor communication is, however, needed when the nonoverlap mapping is used. This can take the form of each processor communicating to its north, east, and northeast neighbor processors the image values they need to compute their convolution. Alternatively, each processor can compute the partial convolution values for its north, northeast, and east neighbors and then communicate these values. In either case, the communication overhead is the same. In our programs, we adopt the latter strategy.

It is also important to note that the communication overhead in the template matching problem is small relative to the computing cost. When the overlap mapping is used, $O(nm\sqrt{p} + pm^2)$ additional data is transmitted from the host to the hypercube nodes (i.e., in addition to the transfer of n^2 image values). However since the host can send data to several nodes in parallel, the overhead penalty is not as severe. While the same amount of data has to be transferred between processors when the nonoverlap mapping is used, the p processors can work in parallel so that the transfer time is approximately that for the transfer of $O(nm/\sqrt{p} + m^2)$ data. In either case, this overhead is expected to be small compared to the time required for the $O(n^2m^2/p)$ computing to be done by each node.

In each of the three cases listed above, we have assumed that the host broadcasts the template to the hypercube processors using a tree expansion scheme.

The NCUBE/7 run times for $p = 1, 4, 16,$ and 64; $n = 32, 64, 128, 252,$ and 512 and $m = 4, 8, 16,$ and 32 for the overlap memory mapping are given in Figures 15 through 17. For smaller values of p, the template matching can be done only for small n as there isn't enough memory on a hypercube processor to hold the convolution and the image subblocks assigned to it. The figures show that for the case $n = 512$, $m = 32$, and $p = 64$, the run times for the host-to-host case are approximately 2.6% higher than that for the hypercube-to-host case and approximately 13.0% higher than the hypercube-to-hypercube case. This reflects the cost of transmitting the image and the convolution between the host and the hypercube. The observed speed up is almost equal to the theoretical maximum of p. The speedup and efficiency ($speedup/p$) for $n = 64$ and $m = 8$ are shown in Figure 18.

		m			
p	n	4	8	16	32
1	32	0.456	1.479	5.391	20.439
	64	1.832	5.867	21.169	81.485
4	32	0.142	0.383	1.366	5.223
	64	0.524	1.480	5.392	20.440
	128	2.022	5.869	21.170	81.487
16	32	0.104	0.176	0.478	1.596
	64	0.238	0.507	1.477	5.225
	128	0.790	1.754	5.394	20.442
	256	2.925	6.592	21.173	81.491
64	32	0.270	0.421	0.910	2.590
	64	0.428	0.643	1.246	3.172
	128	0.933	1.273	2.349	7.029
	256	2.724	3.293	7.205	22.069
	512	9.365	10.597	25.243	81.491

Times are in seconds
m = template size
n = image size
p = number of processors
Figure 15: Overlap Mapping: Host-to-Host

		m			
p	n	4	8	16	32
1	32	0.407	1.308	4.773	18.200
	64	1.600	5.211	18.891	72.268
4	32	0.126	0.355	1.233	4.666
	64	0.462	1.364	4.860	18.226
	128	1.810	5.367	18.974	72.391
16	32	0.069	0.146	0.426	1.483
	64	0.198	0.456	1.402	5.022
	128	0.695	1.643	5.199	18.830
	256	2.620	6.279	19.875	73.350
64	32	0.108	0.190	0.459	1.424
	64	0.200	0.350	0.832	2.533
	128	0.511	0.880	2.111	6.539
	256	1.645	2.786	6.788	21.405
	512	5.968	9.831	24.341	79.440

Times are in seconds
Figure 16: Overlap Mapping: Hypercube-to-Host

		m			
p	n	4	8	16	32
1	32	0.376	1.274	4.727	18.134
	64	1.504	5.094	18.763	72.105
4	32	0.096	0.320	1.184	4.572
	64	0.378	1.275	4.729	18.136
	128	1.506	5.096	18.764	72.107
16	32	0.028	0.084	0.299	1.146
	64	0.098	0.322	1.185	4.573
	128	0.380	1.277	4.731	18.138
	256	1.508	5.097	18.767	72.109
64	32	0.013	0.027	0.086	0.291
	64	0.030	0.086	0.301	1.148
	128	0.100	0.324	1.187	4.575
	256	0.381	1.279	4.733	18.139
	512	1.510	5.099	18.768	72.110

Times are in seconds
Figure 17: Overlap Mapping: Hypercube-to-Hypercube

	p	1	4	16	64
Host-to-host	Speedup	1.00	3.96	11.57	9.12
	Efficiency	1.00	0.99	0.72	0.14
Hypercube-to-host	Speedup	1.00	3.82	11.43	14.89
	Efficiency	1.00	0.95	0.71	0.23
Hypercube-to-hypercube	Speedup	1.00	3.99	15.82	59.23
	Efficiency	1.00	0.998	0.99	0.93

Times are in seconds
Figure 18: Overlap Mapping: Speedup and Efficiency for $n = 64$ and $m = 8$

The run times for the nonoverlap mapping are presented only for the hypercube-to-hypercube case. In this case, there are two possibilities:

1. Overlap of computation and communication between nodes
2. No overlap of computation and communication between nodes

Our experiments indicate that there is no substantial difference in the run times in the above two cases. This is because the amount of computation is much larger than the amount of communication between nodes. The run times for the nonoverlap mapping are given in Figure 19. For small template sizes the nonoverlap method is significantly slower than the overlap method. For larger template sizes the difference in run time is not so significant. Much of the difference in the run time is attributable to the following observations:

1. The program for the nonoverlap case is considerably more complex and so has greater overhead than that for the overlap case.
2. The data transfer rate from the host to the nodes is much higher than that between nodes.

Figure 20 shows the time required by a CRAY-2 supercomputer to perform template matching. These are approximately one fifth of the hypercube-to-hypercube times on the NCUBE/7 with 64 processors.

		m			
p	n	4	8	16	32
1	32	0.505	1.857	7.000	20.450
4	32	0.139	0.482	1.417	
	64	0.514	1.872	7.026	20.497
16	32	0.045	0.115		
	64	0.142	0.484	1.422	
	128	0.516	1.874	7.031	20.510
64	32	0.021			
	64	0.047	0.118		
	128	0.144	0.487	1.426	
	256	0.519	1.878	7.036	20.520

Times are in seconds
Figure 19: Nonoverlap Mapping: Hypercube-to-Hypercube

	m			
n	4	8	16	32
64	0.007	0.023	0.086	0.345
128	0.022	0.080	0.300	1.205
256	0.073	0.283	1.118	4.485
512	0.273	1.082	4.273	17.350

Times are in seconds
Figure 20: Template Matching on CRAY-2

7. CONCLUSIONS

In this paper, we have presented optimal algorithms for 1-D convolution and image template matching (2-D Convolution) on an MIMD hypercube multicomputer. In addition, efficient algorithms for Kirsch templates were developed. Also, we have experimented with a 64 processor NCUBE hypercube and found that this computer can perform template matchings for large images and templates in about five times the time needed by the CRAY-2 supercomputer. Thus, the NCUBE has a very good cost-performance ratio for this problem.

8. REFERENCES

[BALL85] D. H. Ballard and C. M. Brown, "*Computer Vision*", **1985**, Prentice Hall, New Jersey.

[CHAN86] T. E. Chan and Y. Saad, "Multigrid algorithms on hypercube multiprocessor", *IEEE Transactions on Computers*", **Nov. 86**, pp 969-977.

[CHAN87] J. H. Chang, O. Ibarra, T. C. Pong, and S. Sohn, "Convolution on a Pyramid Computer", *International Conference on Parallel Processing,* **1987**, pp 780-782.

[DEKE81] E. Dekel, D. Nassimi and S. Sahni, " Parallel matrix and graph algorithms", *SIAM Journal on computing,* **1981**, pp. 657-675.

[FANG85] Z. Fang, X. Li and L. M. Ni, "Parallel Algorithms for Image Template Matching on Hypercube SIMD Computers", *IEEE CAPAMI workshop,* **1985**, pp 33-40.

[FANG86] Z. Fang and L. M. Ni, "Parallel Algorithms for 2-D convolution", *International Conference on Parallel Processing,* **1986**, pp 262-269.

[HORO85] E. Horowitz and S. Sahni, "*Fundamentals of Data Structures in Pascal*", Computer Science Press, **1985**.

[KUNG82] H. T. Kung and S. W. Song, "A Systolic 2-D Convolution Chip", *Multicomputers and Image Processing: Algorithms and Programs, editors: Preston and Uhr (Academic Press, New York),* **1982**, pp 373-384.

[LEE87] S. Y. Lee and J. K. Aggarwal, "Parallel 2-D convolution on a mesh connected array processor", *IEEE Transactions on Pattern Analysis and Machine Intelligence,* **July 1987**, pp 590-594.

[MARE86] M. Maresca and H.Li, "Morphological Operations on Mesh-connected Architecture : A generalized convolution Algorithm", *Proceedings of 1986 IEEE Computer Society Workshop on Computer Vision and Pattern Recognition* ,**1986**, pp 299-304.

[PRAS87] V. K. Prasanna Kumar and V. Krishnan, "Efficient Image Template Matching on SIMD Hypercube Machines", *International Conference on Parallel Processing,* **1987**, pp 765-771.

[RANK87a] S. Ranka and S. Sahni, "Convolution on an SIMD mesh-connected computer", *University of Minnesota Tech. Report,* **1987**.

[RANK87b] S. Ranka and S. Sahni, "Image Template Matching on an SIMD hypercube multicomputers", *University of Minnesota Tech. Report,* **1987**.

[ROSE82] A. Rosenfeld and A. C. Kak, "*Digital Picture Processing*", Academic Press, **1982**

[THOM77] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer", *Communications of the ACM,* **1977**, pp 263-271.

COMPUTATIONAL GEOMETRY ON A HYPERCUBE

Ivan Stojmenović
Department of Mathematics and Computer Science
University of Miami, P.O.Box 249085
Coral Gables, FL 33124, USA

Abstract. This research focuses on implementing algorithms to solve basic geometric problems on hypercube computers which are recently introduced on the commercial market. Two solutions for the planar convex hull problem are presented, both in $O(\log^2 n)$ time which is the best one can expect with existing sorting algorithms. An $O(\log^3 n)$ Voronoi diagram algorithm is given. $O(\log n)$ solutions for detecting and finding intersection of two convex polygons, computing minimal distance between two convex polygons and finding critical support lines of two convex polygons and $O(\log^2 n)$ solutions to the diameter, smallest enclosing box, width, minimax linear fit, vector sum of two convex polygons, ECDF searching, 2-D and 3-D maximal elements, 2-set dominance counting and closest points problems are described. Several data communications techniques used to solve geometric problems are also presented.

Introduction

A d-dimensional hypercube computer consists of $n = 2^d$ synchronized processing elements (or nodes), linked together in a d-dimensional binary cube network. Each node has associated a constant size memory. Each node m is given a unique d-bit identification number $(m_{d-1}, ..., m_1, m_0)$ (henceforth referred as the node i.d.). In the lexicographic order of nodes, node and its i.d. number are related by $m = 2^{d-1}m_{d-1} + ... + 2m_1 + m_0$. Two nodes in a hypercube are said to be neighboring if they share a communication link, i.e. iff their corresponding i.d.'s differ in exactly one bit position. The notation $\otimes_k(m)$ will be used to denote the node m with k-th bit flipped (for example, $\otimes_3(01001) = 00001$). The neighbors of a node m are exactly $\otimes_0(m), \otimes_1(m), ..., \otimes_{d-1}(m)$. The communication diameter of hypercube networks is logarithmic.

We use a model of hypercubes in which communication time is assumed to predominate. We ignore the time for start-up and termination and the transfer rate when sending messages. Thus, we assume that, in unit time, each processor may send at most one message to one of its neighbors or perform at most one operation (processor-bound model). Using the model described, we solve some geometric problems, assuming that we are given one element (point, edge,...) per processor (we suppose that input/output procedures are done in constant time via a processor with large memory connected with all other processors). The next section describes data communication techniques used in our solutions.

Data communication on hypercubes

Broadcasting. One node has to send the same message to all the other nodes in the hypercube. A $O(\log n)$ solution is presented in [20].

Parallel prefix. Given an array $b_0, ..., b_{n-1}$, one element per processor, compute $b_0 * b_1 * ... * b_i$ for $1 \le i \le n-1$, where '$*$' is arbitrary binary associative operation. We implement the standard parallel prefix algorithm (cf. [7]) on a hypercube, to run in $O(\log n)$ time. Each node m of hypercube stores three data: t, r and c, where t is initially equal to b_m and finally to $b_0 * b_1 * ... * b_m$. We use '$a \leftarrow_i b$'('$b \rightarrow_i a$') to denote sending data b from $\otimes_i(m)$(m, resp.) to node $m(\otimes_i(m)$, resp.) which receives it and stores as data a. '$:=_i$' is used for assignements made in node $\otimes_i(m)$. The algorithm runs for each node x in parallel.

FOR $i = 0$ TO $d-1$ DO IF $x + 1 = 0 (\text{mod } 2^{i+1})$ THEN
BEGIN $r \leftarrow_i t; t := r * t$ END;
FOR $i = d-2$ DOWNTO 0 DO
IF $x + 1 = 0 (\text{mod } 2^{i+1})$ and $x \neq 2^{i+1} - 1$ THEN
BEGIN $c \leftarrow_i t; r \rightarrow_i r; t :=_i r * t; r := r * c$ END

Maximum. For '$*$' being max the first step of our algorithm will report (in node $n-1$) the maximal element.

Ranking. Some nodes are selected. The rank of a node is the number of selected nodes with a smaller index. A $O(\log n)$ ranking algorithm has been presented in [15]. Our parallel prefix algorithm solves also the ranking problem (for '$*$' being '$+$' and b_m being 0 or 1) with less number of data movenment operations than in [15].

Sorting. Given an element per processor, the sorting can be done in $O(\log^2 n)$ time [4,22]. After sorting the elements are kept in nodes in the lexicographic order.

Merging. Given two sorted arrays A and B each stored in a hypercube of size $n/2$, their merging can be done in $O(\log n)$ time [22]. We present an iterative and simple code of merging procedure from [22].

FOR $i = 0$ TO $d-2$ DO IF $x < n/2$ THEN $x \rightarrow \otimes_i(x)$)
FOR $i = d-1$ DOWNTO 0 DO
IF $x_i = 0$ THEN order$(x, \otimes_i(x))$

If the data in x is less than the data in $\otimes_i(x)$ then x and $\otimes_i(x)$ will exchange data as the effect of function order$(x, \otimes_i(x))$. The symbol '$\rightarrow$' is used for passing data from one node to the other.

The cousins of $a \in A$ in B are two consecutive elements in B so that a is between them in sorted list $A \cup B$. The cousins in B of each element in A can be determined in $O(\log n)$ time on a hypercube by merging and interval broadcasting operations. The ranks of two cousins $b1$ and $b2$ from B for an element $a \in A$ are determined by $r(b1, B) = r(a, A \cup B) - r(a, A)$ and $r(b2, B) = r(b1, B) - 1$, where $r(e, X)$ denote the rank of an element e in the sorted set X (the rank of the first element being 0).

Reversing. The effect of the first step in the merging procedure is to reverse data in nodes 0,...,$n/2-1$. It means that a list of data can be reversed in $O(\log n)$ time.

Distribution. We assume that some nodes m of the hypercube store a record r_m and a node destination address h_m such that if $i < j$ then $h_i < h_j$. The distribution operation consists of routing, for each m the record r_m to the node h_m. It can be performed in $O(\log n)$ time [15].

Translation. Node x has to send a message to the node $x + s \pmod n$ concurrently for several nodes x. This can be done in $O(\log n)$ time by two distributions (ones for nodes with $x + s < n$ and ones for the remaining nodes). For $s = 1$ it gives an access for each node to the data in its succesor in the lexicographic order.

Compression. Some nodes of hypercube contain "active" elements while others do not. Compress the active elements, i.e. store them in nodes 0,1,2,...,s-1 where s is

the number of active elements. After ranking active elements compression became inverse distribution operation and a solution is presented in [15].

Unmerging. Given a sorted list of elements so that half of elements belong to a set A (thus the remaining belong to $\bar{A}$, the complement of A) and each element knows the corresponding rank in A or $\bar{A}$, permute the list to return each A and $\bar{A}$ to a hypercube of size $n/2$. The problem can be solved by running the merging algorithm in reverse order, or by two compressions and a translation.

Interval broadcasting. Certain of nodes 0,1,...,$n-1$ are leaders; they possess data that they must share with all the higher numbered nodes, up to but not including the next leader (the interval of nodes between two leaders). Interval broadcasting can be done in $O(\log n)$ time on a hypercube ([22, Theorem 6.9], and [15, Theorem 1]).

Many – one routing. Both origin and destination nodes have keys, with keys of origin nodes being different between each other. Each destination node should receive data from the origin with the same key. The problem can be solved in $O(\log^2 n)$ time on a hypercube[22], by applying sorting and interval broadcasting techniques.

Pairing elements. Given two sets A and B each containing $\sqrt{n}$ data distributed one per node of a hypercube of size n, broadcast these data in such a way that each node of the hypercube contains exactly one pair of data (taken one from each A and B) and all pairs are distributed. First we compress data from A. These data will be stored in a sub-hypercube A' having nodes $(0,...,0,x_{d/2-1},...,x_0)$. Also we compress data from B and translate them to the sub-hypercube B' having nodes $(x_{d-1},...,x_{d/2},0,...,0)$. Now, we broadcast data from each node of A' and B' to all nodes of a hypercube of size $\sqrt{n}$. As a result, each node $(x_{d-1},...,x_0)$ of hypercube receives a pair of data by broadcasting from nodes $(0,...,0,x_{d/2-1},...,x_0)$ and $(x_{d-1},...,x_{d/2},0,...,0)$.

Planar convex hull algorithms

We present two $O(\log^2 n)$ solutions for planar convex hull problem on a hypercube model of computation. One uses merging slopes technique (indenpendently used in [10,14,18] for solving several problems on mesh computers and in [18] for solving all problems mentioned in the section on CREW PRAM; the corresponding sequential technique is presented in [21,5]) while the other is based on CREW PRAM algorithm of [2].

Divide-and-conquer is a common strategy to find the convex hull $H(S)$ of a set of points S sorted by x-coordinate: Partition the points of S into two separated sets P and Q of half the size, each stored in a hypercube of size $n/2$, recursively compute $H(P)$ and $H(Q)$ and merge $H(P)$ and $H(Q)$ to form $H(S)$ by computing common tangents of $H(P)$ and $H(Q)$. Two proposed solutions differ in the way to merge $H(P)$ and $H(Q)$.

Merging slopes technique. α-distance of a point to an oriented edge p is its distance to the an edge p' obtained by rotating p for the angle α (with distances of points to the left (right) of p' being positive (negative, resp.)).

Let A and B be two convex polygons in the plane, each containing $O(n)$ edges given in counterclockwise order. Given an angle α, consider the following problem (we call it the extremal search problem $ES(A,B,\alpha)$): For each edge $p \in A$ find a vertex $P \in B$ with the smallest α-distance to p among vertices from B (P is associated point of p in direction α). It is easy to see that for α=0 ($\alpha = \pi$) P is the vertex with the smallest (greatest, resp.) distance from p among vertices of B. For $\alpha = \pi/2(\alpha = 3\pi/2)$ P is the easternmost (westernmost, resp.) point of p.

To describe the procedure $ES(A,B,\alpha)$, we first increase slopes of edges of A by α. The edges with minimal slopes in A and B are recognized and by some translations they are moved to first nodes of corresponding hypercubes. Since slopes of edges of both polygons are then given in increasing order, the sets A and B can be merged (by their slopes) in $O(\log n)$ time. Now sets A, B and $A \cup B$ are sorted and each edge e of A can find its cousins in B, the common elements of which is associated point of e. We use unmerge technique to return all edges to initial positions.

In order to to merge $H(P)$ and $H(Q)$, we decide for each their edge whether it is an external or internal edge, i.e. if it is convex hull edge of $H(S)$ as well. To judge if an edge is external, we need to test if $H(P)$ and $H(Q)$ are in the same half-plane bounded by the edge. However, instead of testing all the vertices of $H(Q)$ with an edge e of $H(P)$, we only test two representatives (associated points of e) such that if they are in the same half-plane bounded by e as $H(P)$, every point in $H(Q)$ is. These two representatives for e in $H(P)(H(Q))$ are nearest and furthest extreme points from $H(Q)(H(P)$, resp.) and are obtained by calling procedures $ES(H(P),H(Q),0)$, $ES(H(P),H(Q),\pi)$, $ES(H(Q),H(P),0)$ and $ES(H(Q),H(P),\pi)$. Now each edge can decide in constant time if it is external or not. Then each extreme point of $H(P)$ or $H(Q)$ can learn if it is an extreme point of $H(S)$ (translation by 1 can be used to find the neccesary data). Two of them in both $H(P)$ and $H(Q)$ share an external and an internal edge. These four points determine two common tangents of $H(P)$ and $H(Q)$. Then the computation of the circular edge list of $H(S)$ can be done in $O(\log n)$ time by some translations.

The time complexity of all procedures in merge step is $O(\log n)$. Because of $O(\log n)$ recursive calls, the overall time complexity of presented algorithm is $O(\log^2 n)$.

Using the merging slopes technique the diameter, smallest enclosing box, width and minimax linear fit of a set of n points and vector sum and critical support lines of two convex polygons (see [21,5,16,18,19] for definitions) can be found on a hypercube. The details of these solutions are presented in a full version of the article [19].

Another convex hull algorithm on a hypercube can be derived by using Atallah and Goodrich [2] CREW PRAM solution. The main point in the algorithm [2] is to divide P and Q into $\sqrt{n}$ equal portions by considering $\sqrt{n}$ vertices and, by examining each pair of considered vertices, to find common tangent of polygons of size $\sqrt{n}$ obtained in this way. Then one of polygons P or Q can be reduced to size $\sqrt{n}$ and, in one more iteration, the common tangent of P and Q will be constructed [2]. The pairing elements and the broadcasting (to construct the tangent passing through a point) techniques are applied. A similar approach was used in [6] to solve the convex hull problem in $O(\log^2 n)$ time on CREW PRAM and cybe connected cycle models. The later one is directly implementable on hypercube in $O(\log^2 n)$ time.

The problem of computing minimal distance between two convex polygons involves similar techniques, and is solved in [2] for CREW PRAM model of computation. Using pairing elements and other techniques a $O(\log n)$ hypercube solution can be obtained.

Planar point location and Voronoi diagram

In order to locate $O(n)$ points into the planar subdivision defined by $O(n)$ edges we use the chain method described by Lee and Preparata [11], a parallelization of which for mesh-connected computers is given in [12]. We slightly modify both methods in order to get an $O(\log^2 n)$ planar point location algorithm on a hypercube.

First we sort regions by x-coordinate of selected interior points (called centers). Then a monotone complete set of chains is defined as in [11,16,12]. These chains are nodes of a balanced binary tree the leaves of which correspond to regions of subdivision. Each chain has its level and index (the rank of the chain in the chains of given level). Chains may share common edges. If an edge e belongs to more than one chain it belongs to all members of a set (an interval) of consecutive chains. We assign e to hierarchically the highest chain to which e belongs. The level and index of the chain is determined in constant time by the rule described in [12]. Now we sort all edges by their level as the primary key, their index as the secondary and the y-coordinate of the endpoint of edge as the ternary key (endpoint with less y-coordinate among two endpoints of an edge is chosen). Also, we sort all query points by their y-coordinates. Initially, all query points are assigned highest level $\lceil \log n \rceil$ and index 0. Then, for each level i, from $i = \lceil \log n \rceil$ to $i = 0$ do the following:

(i) Merge the set of edges and query points (note that all query points have the same level, equal to i),

(ii) Perform interval broadcasting to find, for each query point Z, the corresponding edge e the query point should be discriminated against. If the y-coordinate of Z is not between y-coordinates of endpoints of e then Z has been discriminated at level before. Depending on which side of e the query point Z is, Z calculates the index of chain at the next level it should be discriminated,

(iii) Unmerge edges and query points (using former indices of query points),

(iv) Re-sort query points by new indices, by compressing query points with answer "left" of corresponding edge in Step (ii) (query points with answer "right" will be also compressed) and (since both subsets of query points are sorted by new indices after compressing) merging "left" and "right" query points by their new indices. Give next level to all query points.

All query points will be located in the Step (ii) when $i = 0$. Since all steps (i)-(iv) take $O(\log n)$ time, the time complexity of planar point location algorithm is $O(\log^2 n)$.

An $O(\log^3 n)$ algorithm to construct Voronoi diagram of a set S of n planar points on a hypercube with n processors can be obtained by using Jeong and Lee [10] algorithm to solve the problem on mesh-connected computers, planar point location technique and presented data communication techniques. In [19] it is shown that all operations in the merge step of the algorithm can be implemented in $O(\log^2 n)$ time.

Finding intersection of two convex polygons

We give a parallel algorithm for finding the intersection of two convex polygons P and Q with $O(n)$ vertices all together by modifying the sequential method of [17].

By drawing a vertical line through each vertex of P and Q we divide P and Q into slabs. The leftmost and the rightmost vertices of P and Q (they can be found in $O(\log n)$ time) divide both P and Q into two chains: the upper and the lower chain of vertices (denote them u_P, l_P, u_Q and l_Q respectively). The intersections $A1$, $A2$ and $A3$ of a vertical line passing through a vertex A of a chain with remaining three chains can be obtained in parallel (one processor per each vertex A) by computing nearest points A' and A'' of A to the left and to the right respectively in the considered chain and finding the intersection of $A'A''$ with vertical line through A. Clearly A' and A'' are the cousins of A in the chain. Upper and lower chains of P and Q can be formed by some translations and reversions from P and Q. Then desired intersections can be obtained by merging $u_P \cup u_Q, u_P \cup l_Q, u_P \cup l_P, l_P \cup u_Q, l_P \cup l_Q$, and $u_Q \cup l_Q$ and unmerging between two steps. Let $P \cup Q$ denote the list of vertices of P and Q sorted together by x-coordinate (it can be constructed in $O(\log n)$ time by merging upper and lower chains of P and Q). Consider each slab defined by two neighboring points A and B. On the basis of the coordinates of points A, $A1$, $A2$, $A3$, B, $B1$, $B2$ and $B3$ (translation by 1 can be used to exchange the data) one can decide in $O(1)$ time in parallel whether P and Q intersect within the slab and determine (at most two) points of $P \cap Q$ which are located in the slab (these are either intersections of edges of P and Q or vertices of P or Q which are located inside the other polygon). So far we have detected all vertices of intersection of P and Q in $O(\log n)$ time. However, we should order them to obtain their convex hull. For each vertex of intersection we decide whether it is an vertex of upper or lower chain of $P \cap Q$ (this can be done in constant time). Also, we assign the vertex to left vertex of corresponding slab defined by points of $P \cup Q$. Thus each vertex of P or Q will have assigned zero or one vertex of $P \cap Q$ from upper convex hull chain (and similarly for lower convex hull chain). Now, upper chain of $P \cap Q$ can be obtained by simply compressing points of $P \cup Q$, assuming that active points of $P \cup Q$ are those having assigned a vertex of $P \cap Q$. Similarly we find lower convex hull chain and finally order vertices of $P \cap Q$ by some translations and reverse steps.

This algorithm solves also the problem of detecting intersection of two convex polygons in parallel (linear separability). Clearly they intersect if at least one vertex of $P \cap Q$ is found. The time complexity is still $O(\log n)$.

The described algorithm can be also implemented in optimal time on a mesh computer and in $O(\log n)$ time on a CREW PRAM.

ECDF searching problem

Given a set $S = \{p_1, \ldots, p_n\}$ of n points in 2-dimensional space. A point p_i dominates a point $p_j (p_i > p_j)$ iff $p_i[k] > p_j[k]$ for $k = 1, 2$, where $p[k]$ denotes the k-th coordinate of a point p. The 2-dimensional ECDF searching problem consists of computing for each $p \in S$ the number $D(p, S)$ of points of S dominated by p.

Let the rank $B(p, S)$ of a point p in the set S containing n points be the position of p in the set S sorted according to the y-coordinate of points, the rank of bottommost and uppermost points being 0 and $n - 1$, resp.

As a preprocessing step of ECDF searching algorithm, we sort points by x-coordinate. The rest of algorithm is best described recursively. Suppose S is divided into two subsets L and R of equal size with $l[1] \leq r[1]$ for all $l \in L$ and $r \in R$, both sorted by y-coordinate. After the recursive calls for L and R in parallel we will have $D(l, L), D(r, R), B(l, L)$ and $B(r, R)$ for all $l \in L$ and $r \in R$. The main point is that the number of points from L which are below $r \in R$ is $\max\{B(l, L) | l[2] \leq r[2]\} = B(r, S) - B(r, R)$. Therefore the final result will be obtained directly from the relations:

$D(l, S) := D(l, L)$ for all $l \in L$,

$D(r, S) := D(r, R) + B(r, S) - B(r, R)$ for all $r \in R$.

Unfolding resursion yields the iterative solution. Initially $B = D = P = 0$ for each node x of hypercube.
FOR $i = 0$ TO $d - 2$ DO
BEGIN
MERGE consecutive blocks of size 2^i in pairs;
IF $x_i = 1$ THEN $D := D + P - B$;
$B := P$
END

In the merge procedure values B, D and P are exchanged whenever data are exchanged between nodes. The ranks of elements after merging are denoted by P and are easily obtained as the relative node's i.d. in the corresponding block of size 2^{i+1}.

The running time of merging step is $O(\log n)$ which give a total $O(\log^2 n)$ time for ECDF searching problem.

The same algorithm solves also the maximal elements problem, i.e. the problem of determining points which are dominated by no other point. We replace the sign $>$ by $<$ in the definition of domination and look for points p with $D(p, S) = 0$. Maximal elements can also be determined directly by sorting and parallel prefix (with $* = max$) operation, as suggested in [3] for CREW PRAM model.

The 2-set dominance counting problem (computing for each point from A the number of points from B dominated by the point) and the maximal element problem for point sets in three-dimensional space can be solved on hypercube in $O(\log^2 n)$ time by a similar iterative algorithm, using labeled functions from [3].

Closest points problem

A $O(\log^2 n)$ hypercube solution to the problem of computing two points with the smallest distance among n given points based on a sequential method presented in [16] will be described. We again apply iterative approach rather than recursive one.

First we sort n points from given set S of points by x-coordinate. At a stage i (where i ranges from 0 to $d-1$), let L and R be left and right halves of points in a given block of size 2^i, respectively. Suppose L and R are both sorted by y-coordinate, and $\delta_1(\delta_2)$ (the smallest distance between points in $L(R$, resp.)) are found. Let active elements of S be those with distance from a line separating L and R less than δ=min(δ_1, δ_2). Compress a copy of active elements in both L and R and merge them to form list S' of active elements. Each active element from $L(R)$ should calculate its distance to constant number of active elements in $R(L)$ (at most six, as shown in [16]). By repeating interval broadcasting technique constant number of times (six) we inform each active element about neighboring elements in other set. Then active elements choose the nearest element from other set and minimum over obtained distances is found and compared to δ. Now broadcast new value of δ and merge L and R for the next stage.

Acknowledgements

The author would like to thank Frank Dehne (Carleton University) and the referees for valuable comments.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing and C. Yap, "Parallel computational geometry," IEEE Symp. Found. Comp. Sci. (1985), pp. 468-477.

[2] M.J. Atallah and M.T. Goodrich, "Parallel algorithms for some functions of two convex polygons," Algorithmica, to appear.

[3] M.J. Atallah and M.T. Goodrich, "Efficient plane sweeping in parallel," ACM Symp. Comp. Geom. (1986), pp. 216-225.

[4] K.E. Batcher, "Sorting networks and their applications," Proc. Spring Joint Computer Conf. (1968), pp. 307-314.

[5] K.Q. Brown, Geometric transforms for fast geometric algorithms, Dept. of Computer Science, Carnegie-Mellon Univ. (1979).

[6] A.L. Chow, "A parallel algorithm for determining convex hulls of sets of points in two dimensions," Proc. Allerton Conf. Comm. Control and Comp. (1981), pp. 214-223.

[7] R. Cole and U. Vishkin, Faster optimal parallel prefix sums and list ranking, Ultracomputer Note 117, Comp. Sci. TR 277, (Feb. 1987).

[8] F. Dehne, "$O(\sqrt{n})$ algorithms for the maximal elements and ECDF searching problem on a mesh-connected parallel computer," Inform. Process. Lett., 22(1986), pp. 303-306.

[9] F. Dehne and I. Stojmenović, "An $O(\sqrt{n})$ algorithm for the ECDF searching problem for arbitrary dimensions on a mesh of processors," Inform. Process. Lett., to appear.

[10] C.S. Jeong and D.T. Lee, Parallel geometric algorithms on mesh – connected computers, Dept. of Electr. Eng. and Comp. Sci., Northwestern Univ., TR 87-02-FC-01, (1987).

[11] D.T. Lee and F.P. Preparata, "Location of points in a planar subdivision and its applications," SIAM J. Comp., 6,3 (1977), pp. 594-606.

[12] M. Lu, "Constructing the Voronoi diagram on a mesh-connected computer," IEEE Conf. Par. Proc. (1986), pp. 806-811.

[13] R. Miller and S.E. Miller, "Using hypercube multiprocessors to determine geometric properties of digitized pictures," Proc. IEEE Conf. Par. Proc. (1987), pp. 638–640.

[14] R. Miller and Q.F. Stout, Mesh computer algorithms for computational geometry (revised), Dept. Comp. Sci., Univ. Buffalo, State Univ. of New York, TR 86-18, (1987).

[15] D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers," IEEE Trans. on Comp. C-30, 2 (Feb. 1981), pp. 101-106.

[16] F.P. Preparata and M.I. Shamos, Computational Geometry, An Introduction, Springer-Verlag, N.Y., (1985).

[17] M.I. Shamos and D. Hoey, "Geometric intersection problems," IEEE Symp. Found. Comp. Sci. (1976), pp. 208-215.

[18] I. Stojmenović, Parallel computational geometry, Washington State Univ., Pullman, CS-87-176, (1987).

[19] I. Stojmenović, Computational geometry on a hypercube, Washington State Univ., Pullman, CS-87-180, (1987).

[20] Q.F. Stout and B. Wager, Intensive hypercube communication I : Prearranged communication in link – bound machines, Comp. Res. Lab. Univ. Michigan, CRL-TR-9-87, (1987).

[21] G.T. Toussaint, "Solving geometric problems with the rotating calipers," Proc. IEEE MELECON '83, Athens, Greece, (1983).

[22] J.D. Ullman, Computational aspects of VLSI, Comp. Sci. Press, Potomac, MD (1984).

CONSTANT-TIME GEOMETRY ON PRAMS

Preliminary Version

Quentin F. Stout
Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122 USA

Abstract

Given n points chosen uniformly and independently from the unit square, it is shown that a parallel random access machine (PRAM) with n processors can solve several geometric problems in constant expected time, achieving linear speedup. The PRAM is assumed to be synchronous, with concurrent read and "collision detecting" write, where if two or more processors write to the same memory location simultaneously then the memory value becomes "collision". Problems solvable in constant expected time include determining for each point whether it is an extreme point of the convex hull, determining for each point if it is dominated by any other points, determining for each dominated point a maximal point that dominates it, finding the closest pair of points, and finding the furthest pair of points. These results extend to points chosen uniformly from the unit cube in d-dimensional space, and to many nonuniform distributions.

1. Introduction

It has well-known that synchronous concurrent read, concurrent write parallel random access machines (CRCW PRAMs) are strictly more powerful than most other parallel computers. For example, an n processor CRCW PRAM can determine the minimum of n numbers in $\Theta(\log \log n)$ time [Val], while it is easy to show that on a PRAM with either an exclusive read (ER) or an exclusive write (EW), or on a distributed memory machine, at least $\Omega(\log n)$ time is needed. However, Valiant's results do not really need the full power of concurrent writes, and we show that a weaker property, here called *detecting write (DW)*, can solve many problems equally rapidly. DW is intermediate between CW and EW, in that if two or more processors write to the same memory location at the same time, then the value becomes "collision", no matter what values were being written. Valiant's approach can be utilized on a CRDW PRAM and still finish in only $\Theta(\log \log n)$ time.

It is also well-known that some geometric problems involving data known to be chosen from a uniform distribution can be solved faster, in the expected case, then the same problems for arbitrary data. For example, n points chosen uniformly and independently from the real interval [0,1] can be serially sorted in $\Theta(n)$ expected time, as opposed to the $\Omega(n \log n)$ expected time for comparison-based sorts. Given n points chosen uniformly and independently from the unit square in 2-space, the convex hull and nearest neighbor of each can be determined by a serial computer in $\Theta(n)$ expected time [BeSh, BWY], as opposed to the $\Omega(n \log n)$ time needed for arbitrary planar data sets [Yao].

This paper shows that by combining the synchronous CRDW PRAM model with use of randomization in the algorithms and data sets generated randomly via uniform distributions, several geometric problems can be solved in constant expected time. Note that the best n processor CRCW PRAM algorithm known for sorting n points chosen uniformly from [0,1] takes $\Theta(\log n)$ time, i.e., it is not known how to sort such data sets any faster than arbitrary data sets. Since determining the extreme points of the convex hull of a set of points in the plane requires as much time as sorting [Yao], this would seem to imply that it takes $\Omega(\log n)$ time for a CRCW PRAM to determine which points are extreme points, even when the data is generated from a uniform distribution. However, the proof that determining extreme points is as hard as sorting holds only for a worst-case analysis, and we will show that the extreme points can be determined in constant expected time in a CRDW PRAM with a linear number of processors. To the best of our knowledge, these are the first constant expected-time algorithms for these problems on any parallel machine with only a linear number of processors.

Because of length limitations, results will be given with only sketches of their proof. The algorithms involve doing preliminary work which, with high probability, reduces the data set to a small number of points remaining to be considered. For these points, the processor to point ratio is very high and significantly different techniques can be utilized. However, the remaining points must be first moved together into a small array so that all the processors can locate them and help in the processing. Usually the points would be packed into the initial positions of the array, but this would take more than constant time. Therefore the array is made larger than the expected number of points remaining, and the points are mapped to random locations. The array must be large enough so that with high probability no two points are mapped to the same location, but it must also be small enough so that the processor to array size ratio remains sufficiently high.

The algorithms also have the property that if a step is reached where something undesired happens, then they resort to a standard worst-case polylogarithmic time CREW PRAM algorithm. Since this happens with very small probability, the expected time remains $\Theta(1)$.

Throughout, no attempt has been made to optimize constants.

2. Results

The phrase *randomly chosen points* will mean points chosen independently and uniformly on the unit square $[0,1] \times [0,1]$ in Euclidean 2-space. Algorithms will also require that processors *(PEs)* generate pseudo-random integers in given intervals. It is assumed that these can be computed in constant time, and that they are uniformly and independently distributed on the interval. *Distance* will be measured with the Euclidean metric, though any other L_p-metric could be used. A point in a finite set S is an *extreme point of* S if it is one of the corners of the smallest convex polygon containing S. The point (x_1,y_1) *dominates* the point (x_2,y_2) if $x_1 \geq x_2$ and $y_1 \geq y_2$. A point is *maximal* in a set if it is not dominated by any other point in the set.

The following lemma is used whenever many points have been eliminated from further consideration, and those remaining must be compressed into a small array so that processors can find them. If the expected number of points remaining is k, then the array will be at least of size k^2. Each processor holding such a point must find a place in the array to put the point.

2.0.1 Lemma On a CRDW PRAM of k processors, in $\Theta(1)$ time each can probably be allocated a unique position in an initialized integer array of k^2 positions, with probability of failure $o(k^{-0.5})$.
Sketch: Suppose each position of the integer array is initialized to -1. Each PE writes its ID (a unique positive integer) to a random array position, and then reads that position. If it reads its ID then that is its allocated position, while otherwise a conflict occured. To determine if any processors experienced conflicts, PE 0 writes "false" to a boolean variable `problems`. Next, if any PE experienced a conflict it writes "true" to `problems`, while otherwise it pauses. Now all PEs read `problems`, and are finished if and only if it is false. Otherwise (i.e., if it is true or conflict), another round is repeated by those PEs without allocated positions, where now each such PE first reads the location it picked and does not write to it if it is already allocated. The process is repeated 3 times. One can show that the probability that some PE still has not been allocated a position is $o(k^{-0.5})$.

The proof of the lemma can be extended to show that for any $a > 1$ and $b > 0$, there is a constant $C(a,b)$ such that k processors can be allocated a position in an array of k^a positions in $C(a,b)$ iterations, with probability of failure $o(k^{-b})$.

2.1 Maximal Points and Extreme Points

The following lemma is based on a modification of Valiant's observation that a synchronous CRDW PRAM of k^2 processors can determine the maximum of k values in constant (worst-case) time. The reason for including "Not a Point" as a value is because later algorithms will place a few points into a large array, and hence many entries will not correspond to points.

2.1.1 Lemma In a CRDW PRAM of n processors, suppose each entry of a global array $p[0..n^{1/3}-1]$ contains a point or the value NAP (Not A Point). Then in constant worst-case time the maximal points can be determined, and for each nonmaximal point a maximal dominating point can be determined.
Proof: Let $s=n^{1/3}$, and assume that arrays maximal:[0..s-1] of boolean and dominator:[0..s-1] of point are initialized to true and NAP, respectively. When finished, maximal[i] is true if and only if p[i] is a maximal point, and if p[i] is a nonmaximal point then dominator[i] is one of its maximal dominators. Each PE executes the following algorithm, where i represents the index of the PE ($0 \leq i \leq n-1$), a local variable which is already initialized. Other local variables in each PE are i1, i2, and i3. A temporary global boolean array T:[0..s-1, 0..s-1] is also used. Throughout, whenever conditional instructions occur where some PEs may take one branch and others take the other branch, it is implied that pauses are inserted so that all PEs complete each branch in the same time.

1. Read p[i], and if it is NAP then write false to maximal[i].
2. Let i1=i div s^2, i2=(i div s) mod s, and i3=i mod s.
   {Notice that for each i1,i2,i3 triple with $0 \leq i1,i2,i3 \leq s-1$, there is exactly one PE with that triple}.
3. If i3=0 then write true to T[i1,i2]. {At the end of step 5, T[i1,i2] will still be true only if p[i2] dominates p[i1].}
4. Read p[i1], p[i2], and p[i3]. If p[i1] or p[i2] are not points, or if p[i2] does not dominate p[i1], then write false to T[i1,i2] and go to 6.
5. Otherwise, if p[i3] is a point and p[i3] dominates p[i1] and i3<i2 then write false to T[i1,i2]. {This signals that, even though p[i2] could be used to show that p[i1] is not maximal, there is a dominating point of smaller index and p[i2] should not be used. It doesn't matter whether T[i1,i2] ends up with the value false or "collision"}.
6. If i3=0 then read T[i1,i2], and if it is true then write false to maximal[i1].
7. {At this point, maximal is correctly determined for all positions. Now for each nonmaximal point we locate the maximal dominator of minimal index. These steps are similar to steps 3-6}
   If i3=0 then write true to T[i1,i2].
8. Read maximal[i2] and maximal[i3]. If p[i1] or p[i2] is not a point, or if p[i2] does not dominate p[i1], or if p[i2] is not maximal, then write false to T[i1,i2] and go to 10.
9. Otherwise, if p[i3] dominates p[i1], p[i3] is maximal, and i3<i2 then write false to T[i1,i2].
10. If i3=0 then read T[i1,i2], and if it is true then write p[i2] to dominator[i1].

Since each step takes constant time, the algorithm finishes in constant time.

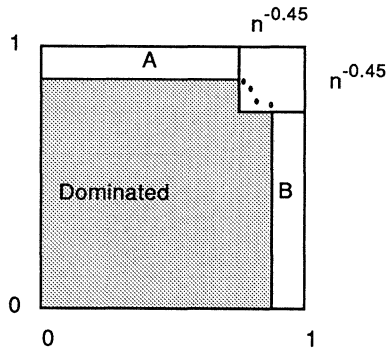


Figure 1

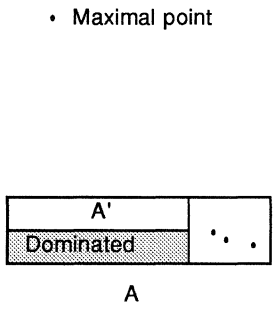


Figure 2

2.1.2 Theorem On a CRDW PRAM with n processors, given a set of n randomly chosen points, in constant expected time it can be determined which points are maximal. Further, in constant expected time, for each dominated point one of the maximal points dominating it can be determined.

Sketch: A sequence of steps is used to continually reduce the number of candidate extreme points for the next step. First it is determined if there are any points in the "corner" $[1\text{-}n^{-0.45},1] \times [1\text{-}n^{-0.45},1]$. With probability close to 1 there are some, but not more than $n^{0.11}$. If there are points in the corner, then they are moved to an array of size $n^{1/3}$, and the maximal points are determined. In this set, the points with greatest x-coordinate and greatest y-coordinate are put in prespecified locations. Every PE i reads them and determines if the i^{th} point is dominated by them. If so the point is marked as not maximal, and dominator is set.

For those remaining there are two groups: those in section A and those in section B of Figure 1. These are treated similarly, so only A will be discussed. Now it is determined if there are any points in A with x-coordinates in $[1\text{-}n^{-0.5},1\text{-}n^{-0.45}]$. With high probability there are some, but not more than $n^{0.11}$. These are also moved to an array of size $n^{1/3}$ and the maximal points determined. Then the point with the largest y-coordinate of this set is read by all PEs corresponding to points not yet dominated, and if they are dominated by this point they set maximal and dominator to appropriate values. This leaves a set A' as in Figure 2.

The process is repeated using regions with x-coordinates in the ranges $[1\text{-}n^{-0.55},1\text{-}n^{-0.5}]$, $[1\text{-}n^{-0.6},1\text{-}n^{-0.55}]$, ..., $[0,1\text{-}n^{-0.95}]$. With probability $o(n^{-0.01})$ each step is completed successfully. If any step does not complete successfully (i.e., either there are no points in the region, or else the PEs corresponding to the points cannot be allocated a position in the array of size $n^{1/3}$ in constant time) then all PEs revert to using a deterministic CREW PRAM algorithm taking $\Theta(\log n)$ worst-case time [AtGo]. The total expected time is $\Theta(1)$.

A similar approach can be used to determine extreme points of the convex hull of the points, starting at the corners and working inwards. In case of failure at some step, CREW PRAM algorithms which finish in polylogarithmic worst-case time [ACGOY, AtGo, MiSt] are used.

2.1.3 Theorem On a CRDW PRAM with n processors, given a set of n randomly chosen points, in constant expected time it can be determined which points are extreme points. Further, in constant expected time, for each point which is not extreme, three extreme points which contain the point in the triangle they form can be determined (or two can be determined, if the point is on the boundary of the convex hull).

Let E denote the number of extreme points. One can show that the expected value of E is $\Theta(\log n)$ [ReSu], which implies that, for any integer k, with probability close to 1 the ratio n/E is $\Omega(E^k)$. Using this, in constant expected time one can apply algorithms which examine all possible combinations of k of the extreme points. This easily yields the following.

2.1.4 Corollary On a CRDW PRAM with n PEs, given a set of n randomly chosen points, in constant expected time the maximal distance between any pair of points can be determined, and enclosing rectangles and circles of minimal area can be determined.

2.2 Closest Pair

A significantly different problem is to determine the closest pair of points. For this problem there is no immediate technique to eliminate points, since even if n-1 points are known it is not possible to determine the closest pair without knowing the last point, and the closest pair might consist of the last point and any of the previously examined points. However, one can reduce the expected number of pairs for which the distance must be determined, by partitioning the unit square into subsquares of edgelength L. If it is known that at least one square has two points in it, then the answer is known to be no more than $\sqrt{2}$ L. In this situation, if a point is in square S in Figure 3, then it may be part of the closest pair only if there are points in S or the 20 nearby squares. By choosing L to be $n^{-0.99}$, then with probability close to 1 there is a square with at least 2 points, no square has 4 or more points, and the number of squares with 2 or more points is $o(n^{0.1})$.

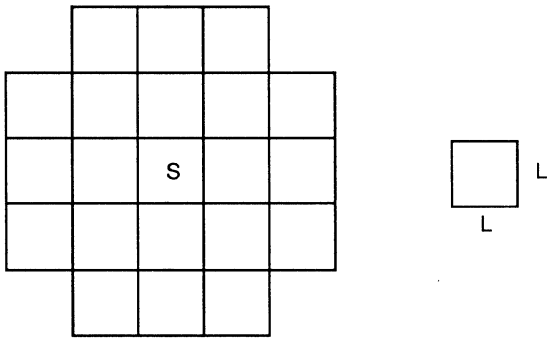


Figure 3. The 20 neighbors with points within $\sqrt{2}$ L of S.

Using this fact, first points are written to their squares. All points in squares with collisions, or which are in one of the 20 squares near a square with collisions, are written to a new array of size $n^{0.2}$. The closest pair within this new array is determined, and also the closest pair involving a square and one of its 20 nearby squares, where all 21 have at most one point in them. The closest pair for the original set is the closer of these two pairs. As before, if any steps cannot be completed properly, then the PEs resort to using a polylogarithmic CREW PRAM algorithm [AtGo].

2.2.1 Theorem On a CRDW PRAM with n processors, given a set of n randomly chosen points, in constant expected time a closest pair can be determined.

3. Final Remarks

This short preliminary version of the paper sketches some ways to accomplish constant time algorithms using random data on a synchronous CRDW PRAM. To the best of my knowledge, these are the first constant expected time algorithms for these problems on any parallel model using a linear number of processors. It seems that the CRDW PRAM is the "weakest" parallel computer which can solve these problems in constant time using only a linear number of processors, but it is not clear how to formalize this properly, let alone prove it.

Many existing algorithms for CRCW PRAMs can be modified to work in the same time on CRDW PRAMs. This occurs because the only concurrent writes they utilize have the property that whenever two PEs are writing to the same memory location at the same time, then they are writing the same value. Such algorithms can often be converted to a scheme as in Lemma 2.0.1, where "conflict" is as useful as the value that was being written. However, not all CRCW PRAM algorithms are of this form, and more work needs to be done to understand which problems can be solved faster on a CRCW PRAM than on a CRDW PRAM.

The algorithms given herein extend quite naturally higher dimensional data. For any fixed dimension d, given n random points chosen from the d-dimensional unit cube, an n processor CRDW PRAM can solve the d-dimensional domination, extreme points, furthest pair, smallest enclosing box, smallest enclosing sphere, and closest pair problems in constant expected time.

The algorithms can also be easily extended to many nonuniform distributions, such as the d-dimensional normal distribution. However, some distributions will cause difficulties because the number of maximal or extreme points may be too large to permit the use of techniques which assume a high processor/point ratio. For example, using the uniform distribution on the unit sphere will result in $\Theta(n^{1/2})$ extreme points, on average [Rayn], rendering the approach of Corollary 2.1.4 invalid.

Acknowledgements

This research was partially supported by Incentive for Excellence Awards from Digital Equipment Corporation, and by National Science Foundation grant DCR-85-07851.

References

[ACGOY] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap, "Parallel computational geometry", *Proc. IEEE Symp. on Found. Comp. Sci.* 26 (1985), pp. 468-477.

[AtGo] M.J. Atallah and M.T. Goodrich, "Efficient parallel solutions to geometric problems", CSD-TR-504, Dept. of Comp. Sci., Purdue Univ., 1985.

[BeSh] J. Bentley and M.I. Shamos, "Divide and conquer for linear expected time", *Info. Proc. Let.* 7 (1978), pp. 87-91.

[BWY] J. Bentley, B.W. Weide, and A.C. Yao, "Optimal expected time algorithms for closest point problems", *ACM Trans. Math. Software* 6 (1980), pp. 563-580.

[MiSt] R. Miller and Q.F. Stout, "Parallel algorithms for convex hulls", *Proc. Comp. Vision and Pat. Recogn. 1988*, to appear.

[Rayn] H. Raynaud, "Sur l'enveloppe convexe des nuages de points aleatoires dans R^n. I", *J. Appl. Prob.* 7 (1970), pp. 35-48.

[ReSu] A. Renyi and R. Sulanke, "Uber die konvexe Hulle von n zufallig gewahlten Punkten, I", *Z. Wahrschein.* 2 (1963), pp. 75-84.

[Val] L. Valiant, "Parallelism in comparison problems", *SIAM J. Computing* 4 (1975), pp. 348-355.

[Yao] A.C. Yao, "A lower bound to finding convex hulls", *J. ACM* 28 (1981), pp. 780-789.

Parallel Implementation of a Kalman Filter on the Warp Computer

David R. O'Hallaron and Radhakisan S. Baheti
GE Research and Development Center, Schenectady, NY 12301

Abstract

A parallel implementation of a 9-state square-root extended Kalman filter for target tracking applications on the Warp computer is described. The Warp computer is a linear array of ten powerful and programmable cells with a maximum performance of 100 million floating-point operations per second (100 MFLOPS). The Kalman filter uses numerically reliable square-root filtering algorithms to estimate the position, velocity and acceleration of a maneuvering target from noisy radar measurements at high data rates. The computations include matrix multiplication, matrix triangularization, coordinate transformation, and Jacobian transformation. We describe the current implementation of the Kalman filter and compare its performance on Warp to a variety of machines.

I. Introduction

Kalman filtering is a general technique for recursively estimating the state variables of a dynamic system from noisy measurements. Applications of the Kalman filter include spacecraft orbit determination, target tracking, image processing, economic forecasting and industrial process control.

This paper describes a parallel implementation on the Warp computer of a 9-state square-root extended Kalman filter for subsonic aerial target tracking applications. The filter estimates, from noisy measurements, the state (position, velocity, and acceleration in each of the x, y, and z coordinates) of a maneuvering aircraft.

We chose Warp [1] as the target machine for a number of reasons. First, we had access to a working machine. Second, linear arrays such as Warp can be highly scalable. Third, Warp is specifically designed for applications (such as Kalman filters) where the ratio of floating-point computations to inputs and outputs is large. Fourth, Warp is programmed in a high-level language and has a rich programming environment. Finally, Warp will be introduced as an Intel product in 1990; the programs we write and the lessons we learn today using the current Warp will be directly applicable to this new, smaller, more powerful, and less expensive machine.

Sections 2 and 3 give background information on Kalman filtering and the Warp computer. Section 4 characterizes the computations of the Kalman filter as a directed graph, describes the mapping of the nodes of this graph onto the Warp cells, and compares the performance of the Warp filter to the performance of an identical filter running on a Vax 11/780, a Sun-3, and a Cray-2.

II. Kalman Filter

A discrete-time equation of target motion can be expressed by the following state variable model [2]:

$$\begin{aligned} \mathbf{x}(t+1) &= \Phi(\Delta t, \alpha)\mathbf{x}(t) + \mathbf{w}(t) \\ \mathbf{y}(t) &= \mathbf{h}(\mathbf{x}(t)) + \mathbf{v}(t) \end{aligned} \qquad (1)$$

$$t = 0, 1, 2, \ldots$$

where t is the normalized discrete-time; Δt denotes the sampling; $\mathbf{x}(t)$ is a nine-dimensional state vector with position, velocity, and acceleration in each of the Cartesian coordinate axes x, y, z. The first-order Markov parameters of stochastic acceleration models in x, y and z axes are denoted by a (3×3) matrix $\alpha = \mathrm{Diag}[\alpha_x, \alpha_y, \alpha_z]$. The (9×9) state transition matrix $\Phi(\Delta t, \alpha)$ is given by

$$\Phi(\Delta t, \alpha) = \begin{bmatrix} I & (\Delta t)I & 1/2(\Delta t)^2 I \\ 0 & I & (\Delta t)I \\ 0 & 0 & \alpha \end{bmatrix} \qquad (2)$$

where I denotes the (3×3) identity matrix. The state vector $\mathbf{x}(t)$ is represented by

$$\mathbf{x}(t) = [x, y, z, \dot{x}, \dot{y}, \dot{z}, \ddot{x}, \ddot{y}, \ddot{z}]^T \qquad (3)$$

where T is the transpose operation. The (3×1) vector $\mathbf{h}(\mathbf{x}(t))$ is the transformation from the Cartesian coordinates to the polar coordinates defined by

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} \\ \theta_T &= \tan^{-1}(\frac{y}{x}) \\ \theta_E &= \sin^{-1}(\frac{z}{r}) \end{aligned} \qquad (4)$$

where r, θ_T, and θ_E denote the target range, azimuth angle, and elevation angle, respectively. The measurement vector

$$\mathbf{y}(t) = [r, \theta_T, \theta_E]^T \tag{5}$$

contains the noisy radar measurements of range, azimuth, and elevation angles.

It is assumed that the state and the measurement noise sequences $\{\mathbf{w}(t)\}$ and $\{\mathbf{v}(t)\}$ have the properties:

$$\begin{aligned} E\{\mathbf{w}(t)\} &= 0; E\{\mathbf{v}(t)\} = 0 \\ E\{\mathbf{w}(t)\mathbf{w}^T(\tau)\} &= R_1(t)\delta_{t\tau} \\ E\{\mathbf{v}(t)\mathbf{v}^T(\tau)\} &= R_2(t)\delta_{t\tau} \\ E\{\mathbf{w}(t)\mathbf{v}^T(\tau)\} &= 0 \\ & t, \tau = 0, 1, 2, ... \end{aligned} \tag{6}$$

where $\delta_{t\tau}$ denotes the Kronecker delta function and E denotes the expectation operator. The elements of the state noise $\{\mathbf{w}(t)\}$ are assumed uncorrelated in x, y, z axes. The radar measurement errors in the target range, azimuth angle, and elevation angle are assumed uncorrelated and may depend on the target range. Let σ_R^2, $\sigma_{\theta T}^2$, and $\sigma_{\theta E}^2$ denote the variance of the measurement noise in range, azimuth angle, and elevation angle, respectively. The (3×3) measurement noise covariance matrix $R_2(t)$ is given by

$$R_2(t) = \text{Diag}[\sigma_R^2, \sigma_{\theta T}^2, \sigma_{\theta E}^2]. \tag{7}$$

The dynamics and the measurement models presented above are commonly used in many tracking and navigation applications. The extended Kalman filter recursively estimates the (9×1) state vector $\mathbf{x}(t+1)$ using the following equation:

$$\hat{\mathbf{x}}(t+1) = \Phi\hat{\mathbf{x}}(t) + \mathbf{K}(t)\{\mathbf{y}(t+1) - \mathbf{h}[\Phi\hat{\mathbf{x}}(t)]\} \tag{8}$$

where $\hat{\mathbf{x}}(t)$ denotes the state estimate based on the measurements $\mathbf{y}(t)$. The (9×3) gain matrix $\mathbf{K}(t)$ is given by

$$\mathbf{K}(t) = \mathbf{P}(t+1|t)\mathbf{H}_x^T\{\mathbf{H}_x\mathbf{P}(t+1|t)\mathbf{H}_x^T + \mathbf{R}_2(t)\}^{-1} \tag{9}$$

where $\mathbf{P}(t+1|t)$ denotes the (9×9) covariance matrix of the filtered error at time $t+1$ before processing measurements $\mathbf{y}(t+1)$. The extended Kalman filter recursively computes the (9×9) estimation error covariance matrix using the following equations:

$$\mathbf{P}(t+1|t) = \Phi\mathbf{P}(t|t)\Phi^T + \mathbf{R}_1(t) \tag{10}$$

$$\mathbf{P}(t+1|t+1) = (\mathbf{I} - \mathbf{K}(t)\mathbf{H}_x)\mathbf{P}(t+1|t) \tag{11}$$

The (3×9) matrix $\mathbf{H}_x$ represents the Jacobian matrix of $\mathbf{h}(\mathbf{x})$ evaluated at $\hat{\mathbf{x}}(t)$.

$$\mathbf{H}_x = \frac{\partial \mathbf{h}(\mathbf{x})}{\partial \mathbf{x}} | \mathbf{x} = \hat{\mathbf{x}}(t) \tag{12}$$

The square-root extended Kalman filter implemented on Warp is a variant of Equations (8)-(12) that manipulates a factored form of the covariance matrix $\mathbf{P}$ [3]. The motivation for using a square-root filter is to improve numerical accuracy by reducing the dynamic range of the numbers in the covariance matrix. Loosely speaking, a square-root filter that uses 32-bit floating-point arithmetic provides the same accuracy as a conventional filter that uses 64-bit arithmetic. This was important for our application because the Warp performs only 32-bit floating-point arithmetic.

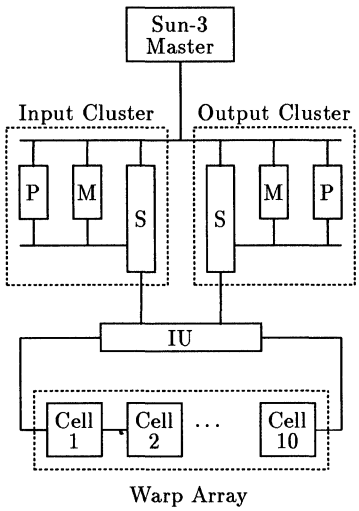


Figure 1: Architecture of Warp

Measurement samples for the Warp Kalman filter were obtained from a simulated benchmark trajectory [2] where a maneuvering target makes a high-g turn past a stationary radar. The target starts at an initial range of 2000 meters, and approaches at a velocity of 400 knots at an altitude of 50 meters. Samples of the simulated trajectory were taken every 20 ms. and noise samples with respective standard deviations of 15 meters, 2 milliradians, and 3 milliradians were added to the sampled range, azimuth angle, and elevation angle to generate the noisy measurement samples.

III. The Warp Computer

The Warp computer [1] (or simply Warp) is a linear array of 10 or more identical and programmable cells connected to a general-purpose host. Warp is designed for computationally intensive applications such as image processing [5], and scientific computing [6].

Figure 1 shows the architecture of the current 10-cell Warp. A Sun-3 workstation called the *master* is connected via a VME repeater to a pair of *clusters*. Each cluster consists of an MC68020 processor(P) with an MC68881 floating point coprocessor, 3 Mbytes of data memory(M), 1 Mbyte of program memory, and a switch(S), all connected by a local VSB bus. The clusters are known collectively as the external host, so called because they are external to the Master.

The Warp array is a linear array of 10 programmable

cells, each containing a pipelined floating-point adder and multiplier, 8K instruction words, and 32K data words. The clock cycle of each cell is 200ns. Each functional unit can emit one result per cycle, for a maximum performance of 5 MFLOPS per functional unit, 10 MFLOPS per cell, and 100 MFLOPS for the 10-cell Warp array. Each cell is connected to its nearest neighbor by two 32-bit data channels (X and Y). Data flows from left to right along the X channel; the Y channel can be statically reconfigured to pass data in either direction. Each cell can transfer up to 20 million 32-bit data words to its neighboring cells each second.

W2 is an Algol-like language with *send* and *receive* statements for cell/cell and host/array data transfers. The W2 compiler inputs a W2 source file, generates microcode for the Warp array and IU, and generates C code for the clusters. WPE is a programming environment that provides a programmable shell for interactively running and debugging W2 programs, as well as a set of lower-level routines that allow user-written C and Lisp programs to access Warp.

IV. Implementation

In this section, we characterize the computations performed by the Warp Kalman filter, describe the implementation of these computations on Warp, and compare the performance of Warp to other machines.

Kalman Filter Computations

For our purposes, a computation graph is a directed acyclic graph consisting of a starting node, s, which has no input edges, an ending node, e, which has no output edges, and m computation nodes, $\{v_1, \ldots, v_m\}$, which represent a partial ordering of computations. Edges are labeled with an integer number of data items. An edge (v_i, v_j) labeled with d indicates that node v_i produces d data items which are consumed by node v_j. Edge (v_i, v_j) also represents a weak precedence relation between nodes v_i and v_j in that node v_j is guaranteed not to complete its computation until node v_i has produced all d data items.

The computations required to process one sample can be characterized at a coarse level by the computation graph, G, in Figure 2. Node v_1 computes state propagation $\Phi\hat{\mathbf{x}}(t)$ in Equation (8). Node v_2 computes the Jacobian transformation of Equation (12). Node v_3 computes a factored form of the right hand side of Equation (10), as described in [3]. Node v_4 computes the Cartesian to polar coordinate transformation, $\mathbf{h}[\Phi\hat{\mathbf{x}}(t)]$, in Equation (8). Node v_5 computes a factored form of the covariance matrix $\mathbf{P}(t+1|t)$ in Equation (10) by triangularizing the matrix produced by node v_3 using a modified weighted Gram-Schmidt orthogonalization technique described in [3]. Node v_6 computes a factored form of the covariance matrix $\mathbf{P}(t+1|t+1)$ in Equation (11) and the updated state vector $\hat{\mathbf{x}}(t+1)$ in Equation (8) using the scalar measurement update technique described in [3].

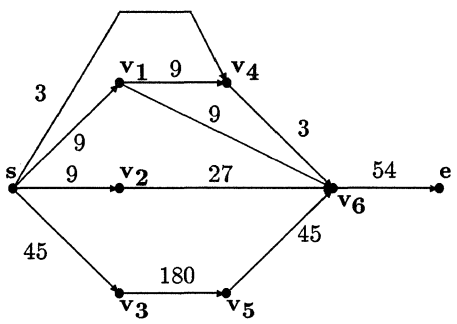


Figure 2: Kalman Filter Computation Graph

Node	Adds	Mults	In	Out	Computation
v_3	324	324	45	180	Matrix multiply
v_5	1458	1602	180	55	Matrix triangularize
v_1	9	12	9	9	Matrix times a vector
v_4	47	73	12	3	Coordinate transform
v_2	14	55	9	27	Jacobian transform
v_6	708	810	84	54	Measurement update
total	2560	2876	57	54	Kalman filter

Figure 3: Warp Kalman Filter Operation Counts

Figure 3 lists the number of floating-point additions, floating-point multiplications, inputs, and outputs for each node in G. Note that the bulk of the computation occurs in nodes v_3, v_5, and v_6 and that these nodes are totally ordered.

Mapping on Warp

The current implementation of the Warp Kalman filter is 700 lines of W2 code. It uses the extremely simple mapping shown in Figure 4. We start with a topological ordering, $T =< v_3, v_5, v_1, v_4, v_2, v_6 >$, of the computation nodes in G. (Recall that a topological ordering of a directed acyclic graph is a total ordering, $< s_1, s_2, \ldots, s_m >$ such that an edge from s_i to s_j implies that $i < j$. Further, such an ordering is guaranteed to exist[4].) Using T as a guide, we then assign node v_3 to cell 1, node v_5 to cell 2, node v_1 to cell 3, and so on. The assignment of nodes to cells in topological order guarantees that data produced by a cell is not needed by cells to its left.

Data flows through the Warp array from left to right along the X channel. For each sample, the measurement vector, state vector, and factored covariance matrix from the previous iteration are received by the leftmost cell from the input cluster. The updated state vector and covariance matrix are sent by the rightmost cell to the output cluster for use with the next sample.

Each cell has the same simple behavior. A cell receives all of its inputs from its left neighbor. Data that is needed for the computation is stored in local memory and data that is required by cells to the right is sent to its right neighbor. The cell then performs its computation and sends the results to its right neighbor. This is repeated

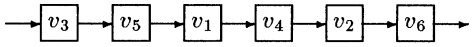


Figure 4: Mapping on Warp

Machine	Sample Time
Vax 11/780	128 ms.
Sun-3/75	96 ms.
Warp	12 ms.
Cray-2	3 ms.

Figure 5: Performance

for each sample.

The mapping we chose for our first implementation is unbalanced, does not scale to linear arrays of arbitrary size, and does not "exploit all of the potential parallelism". The simple mapping did, however, enable us to quickly implement and debug the filter on Warp, to test it using realistic data, to alleviate our concerns about software divides and square-roots, and to gather some baseline performance data.

Performance

The key performance measure for the Kalman filter is *sample time*, that is, the real time required to process one measurement sample. To compare the sample time of the Warp with other machines, we used the first 50 samples from the benchmark trajectory. We ran a C version of the Kalman filter, with compiler optimization enabled, on a Sun-3/75 with 68881 floating-point coprocessor, a Vax 11/780 with floating-point accelerator, and a Cray-2. The sample times for Warp and these machines are listed in Figure 5.

The C programs do not access disk; all data are loaded with the programs and are available when the programs begin executing. The C program for the Cray was compiled using Cray's new vectorizing C compiler. Since the benchmark was not fine-tuned for the Cray-2, we are probably not getting the full benefit of the Cray-2 vector units. However, with vectorization turned off, the sample time for the Cray-2 was identical to the sample time for Warp.

The sample times in Figure 5 for the Sun-3/75, Vax 11/780, and Cray-2 were obtained using the *time* command of the Unix shell. Total user CPU seconds was divided by a factor of 50 (the number of samples) to arrive at the sample time. We also ran the filters using thousands of measurements, with no significant effect on sample time.

The sample time for Warp was obtained using a built-in timing facility on the external host. We noted the time that the input cluster started executing before processing the first sample and we noted the time that the output cluster finished executing after processing the last sample. We divided the difference by 50 to arrive at the elapsed time per sample. The Warp sample time includes the time required to transfer data between the external host and the Warp array, as well as startup times for the external host and Warp array.

We are encouraged by the Warp numbers, especially in view of the naive mapping. Only six of the ten available cells are used and the mapping is unbalanced. Further, in order to simplify the implementation, we restarted the external host and Warp array from the Sun master for *each* new sample. This (completely avoidable) restart accounted for 1/2 of the 12 ms. sample time for the Warp. Our current goal is to reduce the sample time to 1 ms. by developing a more balanced implementation that uses all 10 cells, by eliminating the restart for each sample, and by passing the updated state vector and estimation error covariance matrix from the rightmost cell to the leftmost cell along the Y channel rather than going back through the clusters.

V. Conclusions

We have described the implementation of a 9-state extended Kalman filter for target tracking applications on the Warp computer. It was shown that Warp, using a very simple mapping of the Kalman filter, achieved speeds within a factor of four of a Cray-2. The work has expanded the application domain of the Warp computer to include Kalman filtering, a general technique with many applications.

VI. Acknowledgements

Many thanks to the GE Aerospace Business Group for supporting the work, and to Prof. H. T. Kung, Ed Clune, Etta LeBlanc, Phil Shaffer, and Ko-Haw Nieh for their help.

References

[1] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. Webb. The warp computer: architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, December 1987.

[2] R. S. Baheti. Efficient approximation of kalman filter for target tracking. *IEEE Transactions on Aerospace and Electronic Systems*, AES-22(1):8–14, January 1986.

[3] G. Bierman. *Factorization Methods for Discrete Sequential Estimation.* Academic Press, New York, 1977.

[4] D. Knuth. *The Art of Computer Programming - Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1973.

[5] H. T. Kung and J. Webb. Mapping image processing operations onto a linear systolic machine. *Distributed Computing*, 1:246–257, 1986.

[6] D. R. O'Hallaron. Computing the cholesky decomposition on the warp computer. In *3rd International Conference on Supercomputing*, May 1988.

SOLVING LINEAR PROGRAMMING ON FIXED-SIZE HYPERCUBES

H. F. Ho
Dept. of Elec. Engr.,
National Taiwan Univ.,
Taipei, Taiwan, R.O.C.

G. H. Chen, S. H. Lin
Dept. of Comp. Sci. & Info. Engr.,
National Taiwan Univ.,
Taipei, Taiwan, R.O.C.

J. P. Sheu
Dept. of Elec. Engr.,
National Central Univ.,
Chung-Li, Taiwan, R.O.C.

Abstract -- Although many solution methods are available for the linear programming problem, the simplex method is undoubtedly the most widely used one for its simplicity. In this paper, we shall propose an implementation of the simplex method on fixed-size hypercubes. A partitioning technique and a mapping technique are also presented to fit large-size problem instances into relatively small-size hypercubes. Two cases, pipelined broadcastings allowed and pipelined broadcastings not allowed, are considered. We have shown that the proposed implementation achieves the optimal speedup asymptotically for the both cases. In addition, we have derived sufficient conditions for optimal partitionings when the problem instance sizes are considered finite. These sufficient conditions will be useful to obtain better partitionings. Further, optimal partitionings are found for some special cases.

1. Introduction

Linear programming is a fundamental problem in operations research, and has received much attention for its importance. Mathematically, this problem can be formulated in *standard form* [11] as follows.

$$\begin{array}{ll} \text{minimize} & z=\mathbf{cx} \\ \text{subject to} & \mathbf{A}\,\mathbf{x}=\mathbf{d} \\ & \mathbf{x}\geq 0 \end{array} \qquad (1)$$

where $\mathbf{A}=[a_{ij}]$ is an MxN constraint matrix, $0\leq i\leq M-1$, $0\leq j\leq N-1$, $\mathbf{d}=[d_0,d_1,...,d_{M-1}]^T$ is a positive column vector of length M, $\mathbf{c}=[c_0,c_1,...,c_{N-1}]$ is a row vector of length N, and $\mathbf{x}=[x_0,x_1,...,x_{N-1}]^T$ is a column vector of length N. The linear programming problem is to find the minimum of z. Dantzig [5] has proposed a well-known solution, the *simplex method*, for this problem. The simplex method starts from an initial feasible solution, and then moves continuously from one feasible solution to another, if improvement is obtained. Since the feasible solution space is a convex set, the optimum will be reached eventually after a finite number of iterations. The sequential time for each iteration is $O(MN)$.

The simplex method have been implemented on several parallel machines. For example, a VLSI wavefront array processor implementation was proposed by Onaga and Nagayasu[10], and a VLSI mesh of trees implementation was proposed by Bertossi and Bonuccelli[2]. Onaga and Nagayasu's approach uses MxN processors and requires $O(N+M)$ time for each iteration. Bertossi and Bonuccelli's approach uses MxN mesh of trees with $O(MN\log M\log^3 N)$ area and requires $O(\log N)$ time for each iteration. Both approaches need extra hardware and more complicated control when no sufficient processors are available. In Onaga and Nagayasu's approach, some fixed-size chips are connected together to fit problem instance sizes. The extra hardware connections incur a lot of cost, and the intra-chip controls and communications are not trivial. Moreover, they proposed an alternative that the constraint matrix is folded into a linear array. So, each processor holds a submatrix. However, this still complicates the necessary controls and communications. Bertossi and Bonuccelli proposed another partitioning approach in which the constraint matrix is partitioned into submatrices, each with the same size as the mesh of trees. Their approach needs to perform several successive inputs and outputs during each iteration, which may slow down the execution. Finally, both of them can not achieve the optimal speedup.

The *hypercube* is a high-connectivity and regular parallel architecture. These two properties favor communications among processors. In practice, any two processors can communicate each other within $\log_2 p$ steps, where p is the number of processors in the hypercube. Further, the ability of fault tolerance enhances its reliability. To fully utilize these advantages, several practical hypercube machines have been built, such as the Caltech's Cosmic Cube[13], the NCUBE computer [6], the Intel iPSC[8], the Connection Machine[7], and the Butterfly Machine[3]. In this paper, we shall propose an implementation of the simplex method on fixed-size hypercubes. We consider two cases: *pipelined broadcastings* allowed and pipelined broadcastings not allowed. We have shown that both cases can achieve the optimal speedup asymptotically. In addition, we focus our attention on minimizing communication overheads when the problem instance sizes are considered finite. Sufficient conditions for optimal partitionings (here, optimal partitionings mean those that optimize communication given the mapping method) are derived which will be useful to obtain better partitionings. Also, optimal partitionings are found for some special cases.

The remainder of this paper is organized as follows. The next section briefly reviews the simplex method. Section 3 describes the hyprecube and the *embedded hypercube*. In section 4, we present an implementation of the simplex method on fixed-size hypercubes. A partitioning technique and a mapping technique are presented to fit large-size problem instances into relatively small-size hypercubes. Also, we have a discussion on optimal partitionings. Finally, concluding remarks are given in section 5.

2. The Simplex Method

In this section we shall briefly review the simplex method. For more detailed description, the interested reader may consult [14]. Given a linear programming problem in standard form of (1), the simplex method starts from an initial feasible solution and moves toward the optimal solution. The execution is performed by an iterative procedure. In each iteration, a *basic solution* will be obtained by setting N-M variables equal to zero. These variables are called *nonbasic variables*; the remaining ones are called *basic variables*. With respect to the simplex method, the basic variables in adjacent iterations are only

different by one. Therefore, the basic solution for the next iteration can be obtained from the current basic solution by exchanging one nonbasic variable for one basic variable. The nonbasic variable that "enters" the basic solution is called the *entering variable*, and the basic variable that "leaves" the basic solution is called the *leaving variable*. These two variables can be determined according to the *optimality condition* and the *feasibility condition* as stated below.

Optimality condition. The entering variable is the nonbasic variable with the most negative coefficient c_i. A tie is broken arbitrarily. When all the nonbasic variables have nonnegative c_i's, the current value of z is optimal.

Feasibility condition. Assume that x_u is selected as the entering variable. Then, the leaving variable is the basic variable x_v satisfying $a_{tv}=1$ and $d_t/a_{tu}=min\{d_i/a_{iu} \mid 0\le i\le M-1 \text{ and } a_{iu}>0\}$. Further, a_{tu} is called the *pivot element*, and the $(t+1)$-th row (the $(u+1)$-th column) of **A** is called the *pivot row* (*the pivot column*).

After determining the entering variable and the leaving variable, the constraint matrix **A** and the vectors **c**,**d** ,and z are updated as follows.

$$a_{tj}' \leftarrow a_{tj}/a_{tu} \quad (2)$$

$$a_{ij}' \leftarrow a_{ij} - a_{iu}a_{tj}', \quad i\neq t \quad (3)$$

$$d_t' \leftarrow d_t/a_{tu} \quad (4)$$

$$d_i' \leftarrow d_i - a_{iu}d_t', \quad i\neq t \quad (5)$$

$$c_j' \leftarrow c_j - c_u a_{tj}' \quad (6)$$

$$z' \leftarrow z + c_u d_t', \quad (7)$$

where $j=0,1,...,N-1$, and a_{ij}, d_i, c_j, and z (a_{ij}', d_i', c_j', and z') denote the old (new) values. The use of both the optimality condition and the feasibility condition will result in a better feasible basic solution in each iteration. Since all the feasible basic solutions correspond to the vertices of a convex polytope in N dimensions, the optimum will be reached eventually after a finite number of iterations. Let us assume that performing each binary operation requires the same time and therefore counts one computation step. Then, the total number of (sequential) computation steps required for each iteration is $2MN+2N+3M-1$.

3. The Hypercube and the Embedded Hypercube

In general, there are two classifications of parallel processing models [9]. The shared memory model characterizes the tightly coupled processing, and the distributed memory model characterizes the loosely coupled processing. In the latter, it is assumed that each processor has its own local memory and communicates with others through the interconnection mechanisms. The hypercube belongs to the distributed memory model.

An *h-dimensional hypercube* contains $p=2^h$ (h is a positive integer) identical processors. Each processor is given an h-bit address $(b_h,b_{h-1},...,b_1)$, $b_i=0,1$, $1\le i\le h$, and there exists a link between two processors if and only if their addresses differ in exactly one bit position. The hypercube has a recursive structure as explained below. An h-dimensional hypercube can be regarded as composed of two $(h-1)$-dimensional hypercubes, one with $b_h=0$ and the other with $b_h=1$. Similarly, each of the two $(h-1)$-dimensional hypercubes can be further regarded as composed of two $(h-2)$-dimensional hypercubes, one with $b_{h-1}=0$ and the other with $b_{h-1}=1$, and so on. Generally, each of the sets of processors whose addresses differ in k ($1\le k\le h$) specified bit positions $b_{i1},b_{i2},...,b_{ik}$ and are the same in remaining $(h-k)$ bit positions forms a k-dimensional hypercube. Such a hypercube is called a *k-dimensional embedded hypercube* on $(b_{i1},b_{i2},...,b_{ik})$.

One important issue to implement the simplex method on the hypercube is data broadcasting on some embedded hypercubes. That is, some designated processors are necessary to transmit data to all the other processors belonging to the same embedded hypercubes. These data include the pivot element, the pivot row, the pivot column, etc. To broadcast on a k-dimensional embedded hypercube (on $(b_{i1},b_{i2},...,b_{ik})$), k communication steps are necessary. Initially, the data to be broadcast are located in the designated processor. In the r-th step ($1\le r\le k$), each of the processors owning the data sends a duplicate to the processor whose address differs from its address in the bit position b_{ir}. Thus, k communication steps are sufficient to complete the broadcasting. By taking advantage of the property of fast broadcasting, we can implement the simplex method efficiently on the hypercube.

From the above discussion, two properties of the hypercubes immediately follow.

Property 1. Broadcasting on a k-dimensional embedded hypercube requires k communication steps.

An operation is called a semigroup operation if it is associative. Some well-known semigroup operations are addition, multiplication, finding maximum, and finding minimum. The following property can be obtained from Property 1.

Property 2. Performing semigroup operations on a k-dimensional embedded hypercube (one operand in each processor) requires k communication steps and k computation steps. Additional k communication steps are necessary if the computation result is required by every processor.

4. The Implementation of the Simplex Method on Fixed-Size Hypercubes

In this section, we shall propose an approach to implement the simplex method on fixed-size hypercubes. Suppose that the hypercube is h-dimensional and therefore contains $p=2^h$ processors, where $MN\ge p$ is assumed. The case of $MN<p$ is trivial. Further, we assume that each processor can simultaneously communicate (send data or receive data) with its neighbor processors within a communication step. Therefore, if a processor wants to broadcast w data on a k-dimensional embedded hypercube, it can send out these data in consecutive w communication steps. After $k+w-1$ communication steps, the broadcasting can be completed[4]. Broadcasting in such a way is called *pipelined broadcasting*.

Before presenting our approach, let us discuss the partitionings of data and then the mappings of data onto the hypercube. First, we consider the simple case of $MN=p$. In this case, the data partitionings are not necessary since the sizes of **A**, **c**, and **d** are not larger than the size of the hypercube. Accordingly, **A**, **c**, and **d** are mapped directly onto the hypercube as follows (let $q=\log_2 N$).

1. a_{ij} is placed into the processor with address $(b_h,b_{h-1},...,b_{q+1},b_q,...,b_1)$, where $b_h b_{h-1}...b_{q+1}$ is the binary representation of i and $b_q...b_1$ is the binary representation of j.

2. c_j is placed into the processor with address

$(0,0,...,0,b_q,...,b_1)$, where $b_q...b_1$ is the binary representation of j.

3. d_i is placed into the processor with address $(b_h,b_{h-1},...,b_{q+1},1,1,...,1)$, where $b_hb_{h-1}...b_{q+1}$ is the binary representation of i.

In the case of $MN>p$, the size of **A** is larger than the size of the hypercube. Therefore, **A** must be partitioned into p equal-size submatrices before it can be mapped onto the hypercube. Since **A** is an MxN matrix, each of the submatrices is of size k_1xk_2, where $k_1k_2=MN/p$. Without loss of generality, let us assume that $M=k_12^m$ and $N=k_22^n$ $(m+n=h)$ (if not so, we can extend **A** appropriately by adding dummy rows and dummy columns to **A**) in the following discussion. The partitioning of **A** is shown in Figure 1(a), where $\mathbf{A}_{ij}$'s ($0\le i\le 2^m-1$ and $0\le j\le 2^n-1$) represent the submatrices. As for **c** and **d**, they must be partitioned as well. The partitionings of **c** and **d** are shown in Figure 1(b) and Figure 1(c) respectively, where $\mathbf{c}_j$'s ($0\le j\le 2^n-1$) and $\mathbf{d}_i$'s ($0\le i\le 2^m-1$) represent row subvectors of length k_2 and column subvectors of length k_1 respectively. The mappings of **A**, **c**, and **d** onto the hypercube are as follows.

1. $\mathbf{A}_{ij}$ is placed into the processor with address $(b_h,b_{h-1},...,b_{n+1},b_n,...,b_1)$, where $b_hb_{h-1}...b_{n+1}$ is the binary representation of i and $b_n...b_1$ is the binary representation of j.

2. $\mathbf{c}_j$ is placed into the processor with address $(0,0,...,0,b_n,...,b_1)$, where $b_n...b_1$ is the binary representation of j.

3. $\mathbf{d}_i$ is placed into the processor with address $(b_h,b_{h-1},...,b_{n+1},1,1,...,1)$, where $b_hb_{h-1}...b_{n+1}$ is the binary representation of i.

In both cases, z is placed into the processor with address $(0,0,...,0,1,...,1)$ (that is, $b_h=b_{h-1}=...=b_{n+1}=0$ and $b_n=...=b_1=1$).

According to the above mappings, four facts are as follows.

Fact 1. For any fixed i, the submatrices $\mathbf{A}_{ij}$'s, $0\le j\le 2^n-1$, are mapped onto an n-dimensional embedded hypercube on $(b_n,...,b_1)$. Moreover, each of the processors in the embedded hypercube has the most significant m bits equal to the binary representation of i. For easy description, we shall refer to this embedded hypercube as *row-i embedded hypercube.*

Fact 2. For any fixed j, the submatrices $\mathbf{A}_{ij}$'s, $0\le i\le 2^m-1$, are mapped onto an m-dimensional embedded hypercube on $(b_h,b_{h-1},...,b_{n+1})$. Moreover, each of the processors in the embedded hypercube has the least significant n bits equal to the binary representation of j. In the following discussion, we shall refer to this embedded hypercube as *column-j embedded hypercube.*

Fact 3. The row vector **c** is mapped onto the row-0 embedded hypercube.

Fact 4. The column vector **d** is mapped onto the column-(2^n-1) embedded hypercube.

Now, it is time to describe the implementation of the simplex method on the hypercube. Since the simplex method consists of a finite number of iterations, we shall concentrate our efforts on the necessary operations of each iteration. These operations include, determining the pivot row, the pivot column, and the pivot element, and updating **A**, **d**, **c**, and z as stated by (2), (3), (4), and (5) respectively. In the following, we describe the implementation of these operations.

Determine the Pivot Column

The operands required for this operation are **c**. According to *Fact* 3, we know that k_2 elements of **c** are distributed in each processor of the row-0 embedded hypercube. Thus, the most negative element in each processor is determined first. This takes k_2-1 computation steps. Then, according to Property 2, the most negative element of **c** can be determined taking $2n$ communication steps and n computation steps. The pivot column is the $(u+1)$-th column of **A**, if c_u, $0\le u\le N-1$, is the most negative element. In case of no negative elements, the current value of z is optimal.

Determine the Pivot Row and the Pivot Element

The operands required for this operation are **d** and the pivot column. First, each subvector $\mathbf{d}_i$, $0\le i\le 2^m-1$, is sent to the processor that holds $\mathbf{A}_{iy}$, where $y=u$ DIV k_2 and u is the index of the pivot column. Since $\mathbf{d}_i$ and $\mathbf{A}_{iy}$ are in the same embedded hypercube (the row-i embedded hypercube), the transmissions can be pipelined and performed in parallel. Therefore, k_1+n-1 communication steps are required. Then, the minimum of d_i/a_{iu}'s with $a_{iu}>0$ is determined in each processor. This takes at most $3k_1-1$ computation steps. Finally, the minimum of these 2^m minima is determined. Since these minima are in the column-u embedded hypercube, $2m$ communication steps and m computation steps are required. The pivot row is the $(t+1)$-th row of **A**, if d_t/a_{tu} is the minimum. Besides, a_{tu} is the pivot element.

Update **A**, **d**, **c**, *and* z

(a) The pivot element is broadcast on the row-(t DIV k_1) embedded hypercube, and then the pivot row is updated as stated by (2). This takes n communication steps and one computation step.

(b) The elements of the pivot column belonging to $\mathbf{A}_{iy}$, where $0\le i\le 2^m-1$ and $y=u$ DIV k_2, are broadcast on the row-i embedded hypercube. Since these broadcastings can be pipelined and performed in parallel, k_1+n-1 communication steps are required.

(c) c_u is broadcast on the row-0 embedded hypercube. This takes n communication steps.

(d) The elements of the pivot row belonging to $\mathbf{A}_{xj}$, where $0\le j\le 2^n-1$ and $x=t$ DIV k_1, are broadcast on the column-j embedded hypercube. This takes k_2+m-1 communication steps.

(e) d_t is updated as stated by (4). This takes one computation step.

(f) d_t is broadcast on the column-(2^n-1) embedded hypercube. This takes m communication steps.

(g) $\tilde{\mathbf{A}}$, **c**, **d**, and z are updated as stated by (3), (5), (6), and (7) respectively. This takes totally $2(k_1k_2+k_1+k_2+1)$ computation steps.

Thus, the total numbers of (parallel) computation steps and (parallel) communication steps required for each iteration are

$$2k_1k_2+5k_1+3k_2+h+2 \quad (8)$$

and

$$2k_1+k_2+2n+4h-3 \quad (9)$$

respectively. Since k_1k_2 is equal to MN/p, the optimal speedup is achieved asymptotically. On the other hand, if MN is considered finite, then we choose k_1 and k_2 to minimize (9) (It is quite necessary as the communication steps are much costly, compared with the computation steps). By substituting MN/pk_1 for k_2 and $\log_2(pk_1/M)$ for n, (9) becomes

$$2k_1+MN/pk_1+2\log_2(pk_1/M)+4h-3. \quad (10)$$

Then, differentiating (10) at k_1 and equalizing it to 0, we have

$$2-MN/pk_1^2+(2/k_1)\log_2 e = 0, \quad (11)$$

where e denotes the base of the natural logarithm. It is impossible to solve (11) for integer k_1. However, we think that it is useful to obtain a better k_1.

In the above discussion, we assume that pipelined broadcastings are allowed. In case pipelined broadcastings are not allowed, broadcasting k_1 (k_2) data on an $n(m)$-dimensional embedded hypercube requires k_1n (k_2m) communication steps. In this case, the total number of communication steps required for each iteration is equal to

$$(2k_1+1)n+k_2m+3h. \quad (12)$$

It is clear that the optimal speedup can still be achieved asymptotically when pipelined broadcastings are not allowed. On the other hand, if MN is considered finite, then k_1 and k_2 must be chosen carefully to minimize (12). By substituting MN/pk_1 for k_2 and $\log_2(pk_1/M)$ for n, (12) becomes

$$(2k_1+1)\log_2(pk_1/M) + (MN/pk_1)\log_2(M/k_1)+3h. \quad (13)$$

Then, differentiating (13) at k_1 and equalizing it to 0, we have

$$2\log_2(pk_1/M)-(MN/pk_1^2)\log_2(M/k_1)+(2+(1/k_1) -(MN/pk_1^2))\log_2 e = 0. \quad (14)$$

It is very difficult (even impossible) to solve (14) for integer k_1. However, the constant 1 in (13) is negligible when compared with $2k_1$, and (13) can therefore be simplified to

$$2k_1\log_2(pk_1/M) + (MN/pk_1)\log_2(M/k_1)+3h. \quad (15)$$

In the following, we show that $2k_1=k_2$ will minimize (15) when $2M=N$.

Lemma 1. If the number of communication steps required can be expressed in the form of

$$k_1n+k_2m+c, \quad (16)$$

where c is a constant, then $k_1=k_2=(MN/p)^{1/2}$ *minimizes* (16) *when* $M=N$.

Proof. Substituting MN/pk_1 for k_2 and $\log_2(pk_1/M)$ for n, (16) becomes

$$k_1\log_2(pk_1/M) + (MN/pk_1)\log_2(M/k_1)+c. \quad (17)$$

By differentiating (17) at k_1 and equalizing it to 0, we have

$$\log_2(pk_1/M)-(MN/pk_1^2)\log_2(M/k_1) +(1-(MN/pk_1^2))\log_2 e = 0. \quad (18)$$

It is not difficult to check that $k_1=k_2=(MN/p)^{1/2}$ is a solution of (18) when $M=N$. Thus, this lemma follows. Q.E.D.

Lemma 2. If the number of communication steps required can be expressed in the form of

$$ak_1n+bk_2m+c, \quad (19)$$

where a,b and c are constants, then $ak_1=bk_2$ *minimizes* (19) *when* $aM=bN$.

Proof. This lemma is a generalization of *Lemma* 1. Substituting MN/pk_1 for k_2 and $\log_2(pk_1/M)$ for n, (19) becomes

$$ak_1\log_2(pk_1/M) + b(MN/pk_1)\log_2(M/k_1)+c. \quad (20)$$

Let $k_1'=ak_1$, $M'=aM$, and $N'=bN$. Then, $k_2'=M'N'/pk_1'=bk_2$. By introducing k_1', M' and N' into (20), we have

$$k_1'\log_2(pk_1'/M') + (M'N'/pk_1')\log_2(M'/k_1')+c, \quad (21)$$

which is in the same form as (17). Thus, according to *Lemma* 1, we know that $k_1'=k_2'$ minimizes (21) when $M'=N'$. Thus the lemma follows. Q.E.D.

Since (20) is a general form of (15), it is proved that $2k_1=k_2$ minimize (15) when $2M=N$.

5. Concluding Remarks

The simplex method is a well-known solution method for the linear programming problem. In this paper, we have proposed an implementation of the simplex method on fixed-size hypercubes. Two cases, pipelined broadcastings allowed and pipelined broadcastings not allowed, were considered. For both cases, the optimal speedup can be achieved asymptotically. When the problem instance sizes are considered finite, we derived two sufficient conditions for optimal partitionings. It is difficult to obtain optimal partitionings from these two conditions. However, we think that suboptimal partitionings can be obtained with the aid of them. Besides, we have obtained optimal partitionings for some special cases. Although the simplex method can also be implemented on other parallel architectures [2],[10], they can not achieve the asymptotically optimal speedup. Besides, the numbers of processors they used are dependent upon the problem instance sizes.

Since the linear programming problem we considered in this paper is in standard form, a basic feasible solution must be provided initially. This initial feasible solution can be obtained by using the two-phase technique [11]. In the proposed implementation, the processor with address (0,0,...,0,1,...,1) ($b_h=b_{h-1}=\ldots=b_{n+1}=0$ and $b_n=\ldots=b_1=1$) have more computation loads than others. If its computation loads can be shared by the other processors, then the number of computation steps required for each iteration can be further reduced. To do this is not difficult. We can simply let $M=k_1 2^m-1$ and $N=k_2 2^n-1$ and reduce the sizes of $\mathbf{A}_{0j}$'s and $\mathbf{A}_{i(2^n-1)}$'s, where $0\le j\le 2^n-1$ and $0\le i\le 2^m-1$, to $(k_1-1)\times(k_2-1)$.

References

[1] Ametek Corporation, Hypernet System 14/n, 1985.

[2] A. A. Bertossi and M. A. Bonuccelli, "A VLSI Implementation of the Simplex Algorithm," IEEE Trans. on Comput., Vol. C-36, No. 2, Feb. 1987, pp.241-247.

[3] C. M. Brown, C. S. Ellis, J. A. Feldman, T. J. LeBlanc, and G. L. Peterson, "Research with the Butterfly Multicomputer," Comput. Sci. and Comput. Eng. Res. Rev., Univ. Rochester, Rochester, N.Y., 1985.

[4] P. R. Cappello, "Gaussian Elimination on a Hypercube Automaton," Journal of Parallel and Distributed computing, Vol. 4, 1987, pp .288-308.

[5] G. B. Dantzig, Linear Programming and Extensions, Princeton University Press, Princeton, N.J., 1963.

[6] J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "Architecture of a Hypercube Supercomputer," Proc. Int'l Conf. on Parallel Processing, Aug. 1986, pp. 653-660.

[7] D. Hillis, The Connection Machine, MIT Press, 1985.

[8] Intel Corporation, The iPSC Data Sheet, Beaverton, OR, 1985.

[9] T. N. Mudge and T. S. Abdel-Rahman, "Vision Algorithms for Hypercube Machines," Journal of Parallel and Distributed Computing, Vol. 4, 1987, pp. 79-94.

[10] K. Onaga and H. Nagayasu, "A Wavefront-Driven Algorithm for Linear Programming on Dataflow Processor-Arrays," Proc. of Int'l Computer Symp., 1984, pp. 739-746.

[11] C. H. Papadimitriou and K. S. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Englewood Cliffs, NJ., Prentice-Hall, 1983.

[12] F. P. Preparata and J. Vuillemin, "The Cube Connected Cycles: A Versatile Network for Parallel Computation," Commun. Ass. Comput. Mach., Vol. 24, 1981, pp. 300-309.

[13] C. L. Seitz, The Cosmic Cube, Jan. 1985, pp. 22-23.

[14] A. H. Taha, Operations Research: An Introduction, Macmillan, Inc., N.Y., 1971.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{0\,0} & \mathbf{A}_{0\,1} & \mathbf{A}_{0\,2} & \ldots & \mathbf{A}_{0\,2^n-1} \\ \mathbf{A}_{1\,0} & \mathbf{A}_{1\,1} & \mathbf{A}_{1\,2} & \ldots & \mathbf{A}_{1\,2^n-1} \\ \mathbf{A}_{2\,0} & \mathbf{A}_{2\,1} & \mathbf{A}_{2\,2} & \ldots & \mathbf{A}_{2\,2^n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ \mathbf{A}_{2^m-2\,0} & \mathbf{A}_{2^m-2\,1} & \mathbf{A}_{2^m-2\,2} & \ldots & \mathbf{A}_{2^m-2\,2^n-1} \\ \mathbf{A}_{2^m-1\,0} & \mathbf{A}_{2^m-1\,1} & \mathbf{A}_{2^m-1\,2} & \ldots & \mathbf{A}_{2^m-1\,2^n-1} \end{bmatrix}$$

(a)

$$\mathbf{d} = [\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2, \ \ldots\ , \mathbf{d}_{2^m-2}, \mathbf{d}_{2^m-1}]^T$$

(b)

$$\mathbf{C} = [\mathbf{C}_0, \mathbf{C}_1, \mathbf{C}_2, \ \ldots\ , \mathbf{C}_{2^n-2}, \mathbf{C}_{2^n-1}]$$

(c)

Fig. 1. The partitionings of (a) **A**, (b) **d**, and (c) **c**.

PARALLEL COMPUTATION FOR STOCHASTIC DYNAMIC PROGRAMMING: ROW VERSUS COLUMN CODE ORIENTATION [a]

Floyd B. Hanson
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439-4844
and
Department of Mathematics, Statistics, and Computer Science[b]
University of Illinois at Chicago
Chicago, IL 60680, Box 4348

Abstract -- Parallel computations are presented for stochastic dynamic programming problems arising from the optimal control of nonlinear, continuous time dynamical systems, perturbed by Poisson as well as Gaussian random white noise. The numerical formulation is highly suitable for a vector multiprocessor. Column-oriented code can run more than twice as fast as row-oriented code for highly refined meshes, but row-oriented code is more efficient and practical for coarser meshes. Advanced computing techniques and hardware help alleviate Bellman's *curse of dimensionality* in dynamic programming computations.

1. Problem Summary

The governing Markov dynamical system is the stochastic differential equation (SDE):

$$\begin{aligned} &d\mathbf{y}(s) = \mathbf{F}(\mathbf{y},s,\mathbf{u})ds + G(\mathbf{y},s)d\mathbf{W}(s) + H(\mathbf{y},s)d\mathbf{P}(s) , \\ &\mathbf{y}(t) = x \; ; \; 0 < t < s < t_f \; ; \; \mathbf{y}(s) \in D_y \; ; \; \mathbf{u} \in D_u , \end{aligned} \tag{1.1}$$

where $\mathbf{y}(s)$ is the $m\times1$ state vector at time s starting at time t, $\mathbf{u} = \mathbf{u}(\mathbf{y},s)$ is the $n\times1$ feedback control vector, $\mathbf{W}$ is the r-dimensional normalized Gaussian white noise vector, $\mathbf{P}$ is the independent q-dimensional Poisson white noise vector with jump rate vector λ, $\mathbf{F}$ is the $n\times1$ deterministic nonlinearity vector, G is an $n\times r$ diffusion coefficient array, and H is an $n\times q$ Poisson amplitude coefficient array. Some applications are models of resources in an uncertain environment [8], [10], [7], and flight dynamics under random wind conditions [2].

The control criterion is the optimal expected cost performance,

$$\begin{aligned} &V^*(\mathbf{x},t) = \underset{\mathbf{u}}{MIN}\,[\underset{\mathbf{P},\mathbf{W}}{MEAN}\,[V[\mathbf{y},s,\mathbf{u},\mathbf{P},\mathbf{W}] \mid \mathbf{y}(t) = \mathbf{x}]] , \\ &V[\mathbf{y},t,\mathbf{u},\mathbf{P},\mathbf{W}] = \int_t^{t_f} ds\; C(\mathbf{y}(s),s,\mathbf{u}(\mathbf{y}(s),s)) , \end{aligned} \tag{1.2}$$

on the time horizon (t, t_f), where the instantaneous cost function $C = C(\mathbf{x},t,\mathbf{u})$ is assumed to be a quadratic function of the control,

$$C(\mathbf{x},t,\mathbf{u}) = C_0(\mathbf{x},t) + \mathbf{C}_1^T(\mathbf{x},t)\mathbf{u} + \tfrac{1}{2}\mathbf{u}^T C_2(\mathbf{x},t)\mathbf{u} . \tag{1.3}$$

The unit cost of the control increases with $\mathbf{u}$ when C_2 is positive definite. In addition, the dynamics in (1.1) are assumed to be linear in the controls,

$$\mathbf{F}(\mathbf{x},t,\mathbf{u}) = \mathbf{F}_0(\mathbf{x},t) + F_1(\mathbf{x},t)\mathbf{u} , \tag{1.4}$$

remaining nonlinear in the state variable $\mathbf{x}$.

The Bellman functional PDE of dynamic programming,

$$\begin{aligned} 0 &= V_t^* + L[V^*] \equiv V_t^* + \mathbf{F}_0^T\nabla V^* + \tfrac{1}{2}GG^T(\mathbf{x},t):\nabla\nabla^T V^* \\ &+ \sum_{l=1}^{q} \lambda_l[\, V^*(\mathbf{x}+\mathbf{H}_l(\mathbf{x},t),t) - V^*(\mathbf{x},t)\,] + C_0 + (\tfrac{1}{2}\mathbf{U}^* - \mathbf{U}_R)^T C_2 \mathbf{U}^* , \end{aligned} \tag{1.5}$$

follows from the generalized *Itô* chain rule for Markov SDEs as in [6] and [10], where $\mathbf{U}^*$ is the optimal feedback control computed by constraining the unconstrained or regular control,

$$\mathbf{U}_R(\mathbf{x},t) = -C_2^{-1}(\mathbf{C}_1 + F_1\nabla V^*) , \tag{1.6}$$

to the control set D_u. In general, the PDE (1.5) is nonlinear with discontinuous coefficients.

As the number of state variables, m, increases, the spatial dimension rises, and computational difficulties are present that can compare to those of three-dimensional fluid dynamics computations. This is the famous Bellman's *curse of dimensionality* [3]. Thus there is a great need to make use of advanced-architecture computers, to use parallelization as well as vectorization.

2. Numerical Summary

The integration of the PDE in (1.5) is backward in time, because V^* is specified finally at the final time $t = t_f$, rather than at the initial time. A summary of the discretization in state and backward time is given below:

$$\begin{aligned} &\mathbf{x} \to \mathbf{X_j} = [X_{ij_i}]_{m\times1} = [X_{i1} + (j_i - 1)\cdot DX_i]_{m\times1} , \\ &\mathbf{j} = [j_i]_{m\times1} \; , \; \textit{where } j_i = 1 \textit{ to } M_i \; , \; \text{for } i = 1 \textit{ to } m \; ; \\ &s \to T_k = t_f - (k-1)\cdot DT \; , \; \text{for } k = 1 \textit{ to } K \; ; \\ &V^*(\mathbf{X_j},T_k) \to V_{\mathbf{j}k} \; ; \quad L[V^*](\mathbf{X_j},T_{k+\frac{1}{2}}) \to L_{\mathbf{j},k+\frac{1}{2}} \; ; \end{aligned} \tag{2.1}$$

where DX_i is the mesh size for state i and DT is the step size in backward time.

The numerical algorithm is a modification of the predictor-corrector Crank Nicolson methods for nonlinear parabolic PDEs in [5]. Modifications are made for the switch term and delay term calculations. Derivatives were approximated with an accuracy that was second order in the local truncation error. Variations of this algorithm have been successfully utilized in [10] and [7].

Prior to calculating the values, $V_{\mathbf{j},k+1}$, at the new $(k+1)^{st}$ time step for $k = 1$ *to* $K-1$, the old values, $V_{\mathbf{j}k}$ and $V_{\mathbf{j},k-1}$, are assumed to be known, with $V_{\mathbf{j}0} \equiv V_{\mathbf{j}1}$. The algorithm begins with an *extrapolator (x) start*:

$$V_{\mathbf{j},k+\frac{1}{2}}^{(x)} = \tfrac{1}{2}(3\cdot V_{\mathbf{j}k} - V_{\mathbf{j},k-1}) . \tag{2.2}$$

These evaluations are used in the *extrapolated predictor (xp) step*:

$$V_{\mathbf{j},k+1}^{(xp)} = V_{\mathbf{j}k} + DT\cdot\tfrac{1}{2}L_{\mathbf{j},k+\frac{1}{2}}^{(x)} . \tag{2.3}$$

which are then used in the *predictor evaluation (xpe) step*:

$$V_{\mathbf{j},k+\frac{1}{2}}^{(xpe)} = \tfrac{1}{2}(V_{\mathbf{j},k+1}^{(xp)} + V_{\mathbf{j}k}) . \tag{2.4}$$

The evaluated predictions are used in the *corrector (xpec) step*:

$$V_{\mathbf{j},k+1}^{(xpec,\gamma+1)} = V_{\mathbf{j}k} + DT\cdot L_{\mathbf{j},k+\frac{1}{2}}^{(xpece,\gamma)} \tag{2.5}$$

for $\gamma = 0$ to γ_{max} while stopping criterion unmet, with *corrector evaluation (xpece) step*:

$$V_{\mathbf{j},k+\frac{1}{2}}^{(xpece,\gamma+1)} = \tfrac{1}{2}(V_{\mathbf{j},k+1}^{(xpec,\gamma+1)} + V_{\mathbf{j}k}) . \tag{2.6}$$

The stopping criterion for the corrections is a heuristically motivated comparison to a predictor-corrector convergence criterion for a linearized, constant coefficient PDE [9].

Parallelization and vectorization of the algorithm was done by what might be called the "Machine Computational Model Method", i.e., tuning the code to optimizable constructs that are automatically recognized by the compiler, with the Alliant FX/8 vector multiprocesor [1] in mind. All inner double loops were reordered to

[a] This research was supported in part by Faculty Research Participantships, a Faculty Research Leave at Argonne National Laboratory Advanced Computing Research Facility, and by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under Contract W-31-109-Eng-38.

[b] Permanent address.

fit the Alliant *concurrent outer - vector inner* (*COVI*) model. All non-short single loops were made *vector-concurrent* Short loops became *scalar-concurrent* only. Multiple nested loops were reordered with the two largest loops innermost. A total of 37 out of 39 loops was optimized.

Dongarra, Gustavson, and Karp [4] have demonstrated that loop reordering gives vector or supervector performance for linear algebra loops on a CRAY-1 type column-oriented FORTRAN environment with vector registers. Here performance measurements have been made with both column-oriented loop code and row-oriented loop code, in order to make a comparison for our particular problem and machine environment. In the *column-oriented* loop, the most inner loop iterates on the first subscript, $\cdots$, and the most outer loop iterates on the last subscript, as in the following code fragment that is the main *not inner-COVI* loop from the corrector step (2.5):

```
      do 21 l=1,m
      do 21 j2=1,M2
      do 21 j1=1,M1
      ss(j1,j2,l)=F1(j1,j2,l)*DV(j1,j2,l)
     &  + GSQ(j1,j2,l)*DDV(j1,j2,l)
     &  + lambda(l)*(HV(j1,j2,l) - VM(j1,j2))
21    continue
```

where $m = 2 = n$ with the array subscripts and loop nesting ordered so that the major part of the finite difference work is done in the two most inner loops. In the *row-oriented* loop the inner two subscripts are iterated in reverse order as follows:

```
      do 22 l=1,m
      do 22 j1=1,M1
      do 22 j2=1,M2
      ss(l,j1,j2)=F1(l,j1,j2)*DV(l,j1,j2)
     &  + GSQ(l,j1,j2)*DDV(l,j1,j2)
     &  + lambda(l)*(HV(l,j1,j2) - VM(j1,j2))
22    continue
```

The row orientation in this *do* 22 *loop* refers to an array slice with the state counter l fixed. In the two above code fragments, $GSQ = \frac{1}{2}G^TG$ and $VM = V$ at $T_{k+\frac{1}{2}}$.

The advantages of the algorithm is that it 1) permits the treatment of general continuous time Markov noise or deterministic problems without noise in the same code, 2) permits the cheap control limit to linear singular control to be found from the same quadratic cost code, and 3) produces very vectorizable and parallelizable code whose performance is described in the next section.

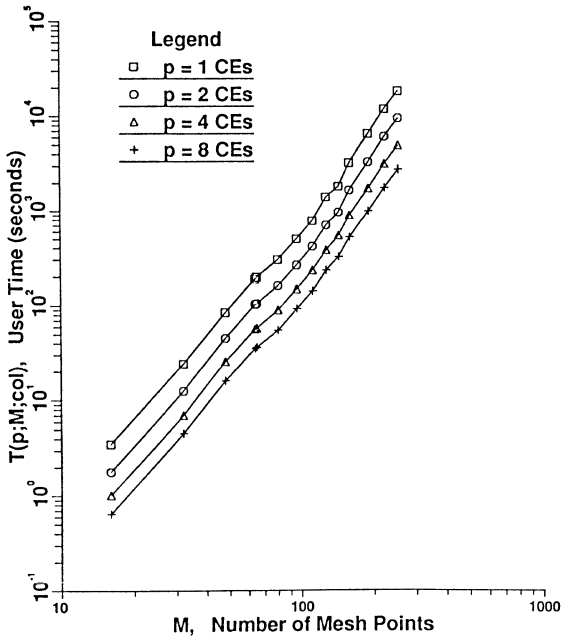
3. Results Summary

The vector multiprocessor used for our performance measurements was the Alliant FX/8 in the Advanced Computing Research Facility (ACRF) at Argonne National Laboratory. This Alliant FX/8 has 8 vector Computing Elements (*CEs*). Each of the *CEs* has eight vector registers whose length is 32 eight-byte elements, and the *CEs* are connected to a 128 KB cache.

The two state and two-control resource model in [7] was used as the test example. The two controls represent removals from the system by respective commercial and recreational users of the system. In this test example, only Poisson noise was used for both state populations.

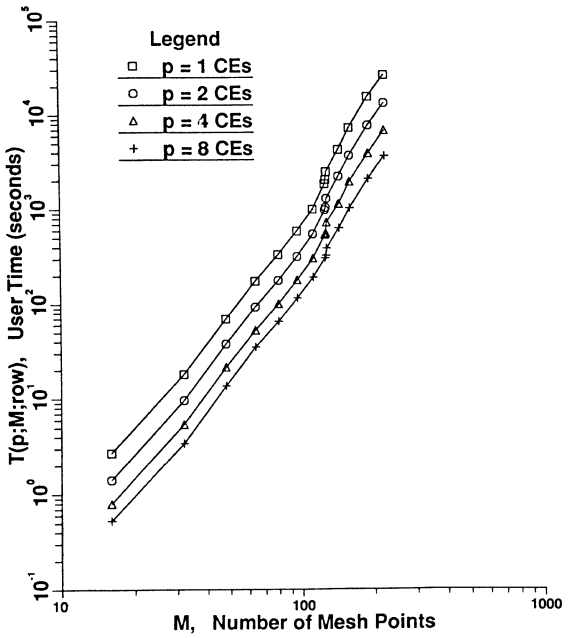
Figure 1 shows the dependence of the user or *CPU* time on the common number of mesh points, $M_1 = M = M_2$, in a log-log plot, for the column-oriented 2-state and 2-control code. $K = 4\cdot(M-1) + 1$ to maintain stability properties. When M is between 16 and 64, the user time $T(p;M;col) = O(M^3)$, because the slope is quite close to three. This corresponds to a *COVI loop dominated limit* with $m = 2$ states and $K = O(M)$ time steps. However, when M is 96 or greater, $T(p;M) = O(M^4)$, implying that additional overhead from the memory hierarchy is present that is equivalent to an extra state dimension ($m \rightarrow m+1$). This change is reflected in the assembler code for the *do 21* loop for $M \geq 65$.

Figure 1. Column Timings vs. M
Stochastic Dynamic Programming
ANL ACRF Alliant FX/8



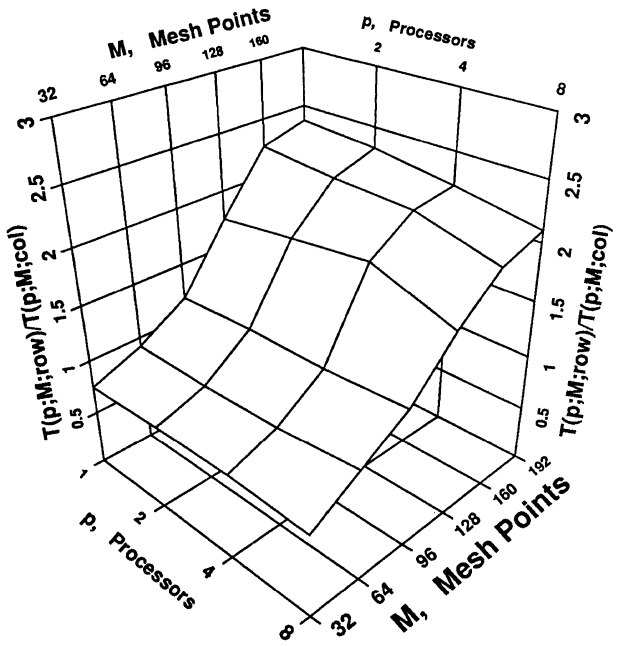
In Figure 2, the dependence of the user execution time for row-oriented code is shown versus the common mesh size M. The transition between $T(p;M;row) = O(M^3)$ to $O(M^4)$ overhead occurs between $M = 126$ and $M = 127$, but is much sharper than in the column-oriented Figure 1. The assembler encoding of the *do 22* loop exactly reflects this transition, even for pure double loops. The extra overhead in the $O(M^4)$ region is nominally due to the fact that multiples of 32 beyond 64 are not treated as multiples by the Alliant compiler.

Figure 2. Row Timings vs. M
Stochastic Dynamic Programming
ANL ACRF Alliant FX/8



In Figure 3, the ratio of the row-oriented to column-oriented user execution time is displayed in a 3-D representation versus both M and the number of processors p. The effect of the number of processors p is much weaker on the ratio of the timings than that of the mesh size M is on the timings, because the array orientation is primarily a memory problem. While the row-oriented version is slightly faster for $M \leq 64$, the column-oriented version is up to two times faster for finer meshes. Hence the column advantage in this FORTRAN environment is present only for the finer meshes with this stochastic dynamic programming code and the Alliant FX/8. Unfortunately, this advantage occurs in a region of extra overhead, where the timings are $O(M^4)$.

Figure 3. Row to Column Ratio vs. p & M
Stochastic Dynamic Programming
ANL ACRF Alliant FX/8



4. Conclusions

Stochastic dynamic programming is practical for several to a moderate number of state variables using a vector multiprocessor. In order to handle a large number of state variables, a large number of parallel processors would be desirable, but Bellman's *curse of dimensionality* appears to very much weakened. These techniques are generally applicable to other vector and parallel computers, e.g., CRAY X-MP as applied to the most inner loops. Row-oriented codes can be very competitive with column-oriented codes for small to moderate mesh sizes ($M \leq 64$), but are twice as slow for finer meshes.

References

[1] Alliant, **FX/FORTRAN Programmer's Handbook**, Alliant Computer Systems Corporation, Acton, Mass., (1985).

[2] M. Athans, D. Castanon, K. P. Dunn, C. S. Greene, W. H. Lee, N. R. Sandell, Jr., and A. S. Willsky, *The stochastic control of the F-8C aircraft using a multiple model adaptive control (MMAC) method - Part I: Equilibrium flight*, **IEEE Trans. Autom. Control, AC-22**, (1977), pp. 768-780.

[3] R. E. Bellman, **Adaptive Control Processes: A Guided Tour**, Princeton University Press, Princeton, (1961).

[4] J. J. Dongarra, F. G. Gustavson, and A. Karp, *Implementation of linear algebra algorithms of dense matrices on a vector pipeline machine*, **SIAM Rev., 26**, (1984), pp. 91-112.

[5] J. Douglas, Jr., and T. DuPont, *Galerkin methods for parabolic equations*, **SIAM J. Num. Anal., 7**, (1970), pp. 575-626.

[6] I. I. Gihman and A. V. Skorohod, **Controlled Stochastic Processes**, Springer-Verlag, New York, (1979).

[7] F. B. Hanson, *Bioeconomic model of the Lake Michigan alewife fishery*, **Can. J. Fish. Aquat. Sci., 44 Suppl. 2**, (1987), pp. 1-29.

[8] D. Ludwig, *Optimal harvesting of a randomly fluctuating resource I: Application of perturbation methods*, **SIAM J. Appl. Math., 37**, (1979), pp. 166-184.

[9] K. Naimipour and F. B. Hanson, *Convergence of a numerical method for the Bellman equation of stochastic optimal control with quadratic costs*, In Preparation, (1988).

[10] D. Ryan and F. B. Hanson, *Optimal harvesting of a logistic population in an environment with stochastic jumps*, **J. Math. Biol., 24**, (1986), pp. 259-277.

PARALLEL LAGRANGIAN INTERPOLATION

G. H. Atwood
Department of Computing Science
The University of Alberta
Edmonton, Canada, T6G 2H1

Abstract

Fast parallel algorithms for the computation and evaluation of interpolating polynomials in the Lagrangian basis are presented. On $n+1$ data points, the interpolation algorithm requires $\lceil \log n \rceil + 2$ parallel arithmetic steps and $n(n+1)$ processors. The results are compared with Egecioglu, Gallopoulos and Koc's [2] parallel Newtonian algorithm, and an extension to their work is presented. The algorithms discussed are suitable for a shared memory PRAM architecture.

Introduction

Given a set of $n+1$ pairs of real values, (x_i, f_i) for $i = 0,1,...,n$ with distinct x_i's, there exists a unique polynomial $p_n(x)$ of degree n such that $p_n(x_i) = f_i$ for $i = 0,1,...,n$. This interpolating polynomial $p_n(x)$ can be written in the Newtonian form

$$p_n(x) = \sum_{i=0}^{n} f_{0,i}\,(x-x_0)(x-x_1)\cdots(x-x_{i-1}) \qquad (1)$$

in which the coefficients $f_{0,i}$ are the *divided differences* of f.

Alternatively, $p_n(x)$ can be written in the Lagrangian form

$$p_n(x) = \sum_{i=0}^{n} l_i\, f_i\,(x-x_0)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n) \qquad (2)$$

where l_i is the real factor of the *Lagrange polynomial* $l_i(x)$.

Various forms of polynomial interpolation have been studied in both the context of numerical analysis and in computational complexity. However, Egecioglu, Gallopoulos and Koc [2] seem to be the first to study the problem in terms of the Newtonian form with respect to parallel solution. In fact, the typical form discussed is the classical form

$$p_n(x) = \sum_{i=0}^{n} p_i\, x^i \ . \qquad (3)$$

Equations (1), (2) and (3) represent the same polynomial function under different bases. The usefulness of a particular basis depends on the application. Divided differences in particular are useful for the evaluation of derivatives and integrals. The Lagrangian form has its own benefits including ease of derivation as will be shown, without adversely affecting such properties as permanence and evaluation.

Egecioglu et. al. remark that fast interpolation algorithms (those with time complexity less than $\mathbf{O}(n^2)$) are still based on slow serial schemes which require $\mathbf{O}(n^2)$ operations (in the sequential machine). Because of the increasing availability of parallel systems, they suggest that existing methods are impractical because: 1) the constant multiplier of the order tends to be relatively large and 2) the current fast interpolation algorithms are subject to significant roundoff errors when implemented in finite precision arithmetic.

The first reason suggests that the size of the problem must be sufficiently large to make these algorithms competitive. However, polynomial interpolation is not usually recommended on a large number of data points. The second reason is self-explanatory, why use fast algorithms if their numerical accuracy is unstable? Factors which affect both these areas are requirements such as equidistant points or the use of convolution techniques (e.g. fast Fourier transforms).

The reader should note that over some fields, polynomial interpolation of a large number of data points is desirable. Modular techniques (c.f. McClellan [5]) can be used to solve large problems efficiently. In this respect the Chinese Remainder Theorem often occurs, and Zhang, Shirazi and Yun [8] have recently shown that the CRT problem can be solved in[a] $\mathbf{O}(\log n)$ parallel steps. This result is applicable to modular techniques and may affect the complexity of parallel algorithms for large polynomial interpolation problems.

In an attempt to address the concerns enumerated above, Egecioglu et. al. present a fast ($\mathbf{O}(\log n)$) and practical parallel algorithm for the computation of interpolating polynomials (in Newtonian form). By *practical* they intend that the proposed algorithm be numerically stable and can be implemented in floating-point with resulting error accumulation similar to stable serial algorithms. Recently, Reif [6] showed that polynomial interpolation of n points in the classical form (3) can be performed in $\mathbf{O}(\log n)$ time complexity, with a circuit of size[b] of $\mathbf{O}(n^2 \log n)$. As the method makes use of discrete Fourier transforms the constant multiplier in the time complexity for this algorithm is greater than 4. Thus, both Egecioglu et. al.'s results and those presented in this paper can be implemented with a circuit of smaller size and depth and without the use of convolution techniques.

In this paper two separate problems are investigated. The first involves extending Egecioglu et. al.'s results to include the permanence property of the Newtonian form. The second problem is to show that the Lagrangian form can be interpolated and evaluated in a manner competitive with Egecioglu et. al.'s results.

The algorithms discussed may be implemented on a shared memory PRAM architecture. Borrowing the description from Kruskal, Rudolph and Snir [3] the PRAM computational model consists of p autonomous processors, executing synchronously, all having access to a common shared memory. The processors execute in SIMD fashion, and memory access is assumed to be accomplished in one cycle.

Parallel Newtonian Interpolation

The success of Egecioglu et. al.'s algorithm depends on two key observations: 1) Computing the divided differences in an alternative form and 2) Using the *parallel prefix* computation (see Kruskal, Rudolph and Snir [3] and Ladner and Fischer [4]) for the evaluation of the divided differences.

Once the divided differences are computed the Newtonian representation of $p_n(x)$ is completely defined. Typically, the divided differences are computed by

$$f_{i,j} = \frac{f_{i,j-1} - f_{i+1,j}}{x_i - x_j} \qquad (4)$$

where $f_{i,i} = f_i$ and $0 \le i < j \le n$. $f_0, f_{0,1}, \ldots, f_{0,n}$ are the divided differences required by the Newtonian form. Note

(a) All logarithms are base 2.

(b) *Size* refers to the number of productive nodes in a circuit.

that from (4) the $f_{i,j}$'s for a particular $j-i$ value can be calculated independently of each other, and depend only on the divided differences calculated for the $j-i-1$ value and the x_i's. This yields a straightforward parallel algorithm where the divided differences for a particular $j-i$ value are computed in parallel. Thus $\mathbf{O}(n)$ operations using $\mathbf{O}(n)$ processors are required to compute all the divided differences.

Egecioglu et. al. present an alternative method that requires $\mathbf{O}(n^2)$ processors and solves the problem in $\mathbf{O}(\log n)$ steps. Let $y_{i,j} = x_i - x_j$ for $i \neq j$ and $y_{i,i} = 1$, $i, j = 0,1,...,n$. Then the kth $(n \geq k \geq 0)$ divided difference of f $(f_{0,k})$ can be expressed as a linear combination of the f_i's and the products of the $y_{i,j}$'s as follows

$$f_{0,k} = \sum_{j=0}^{k} \frac{f_j}{y_{j,0} y_{j,1} \cdots y_{j,k}} \quad . \tag{5}$$

Consider the reciprocals of the coefficients of f_i in $f_{0,i}, f_{0,i+1}, \ldots, f_{0,n}$ (the coefficient of f_i in $f_{0,j}$ is always zero for $j<i$). These are of the form $y_{i,0} y_{i,1} \cdots y_{i,i}$, $y_{i,0} y_{i,1} \cdots y_{i,i+1}$, ..., $y_{i,0} y_{i,1} \cdots y_{i,n}$. But these are just the prefixes of $y_{i,0} y_{i,1} \cdots y_{i,n}$ and the parallel prefix algorithm can generate all of them in $\lceil \log n \rceil$ using only n processors (c.f. Kruskal, Rudolph and Snir [3] and Ladner and Fischer [4]). Ladner and Fischer also show that such a parallel prefix algorithm may be constructed with a circuit of size $\mathbf{O}(n)$. There are $n+1$ data points, thus $n+1$ concurrent instances of the parallel prefix algorithm are needed to compute the prefixes of the term $y_{i,0} y_{i,1} \cdots y_{i,n}$ for $0 \leq i \leq n$. Hence $\mathbf{O}(n^2)$ processors are needed to perform this calculation. Note, $y_{i,i} = 1$ for all i, thus it can be omitted from the prefix calculations without affecting the results.

Egecioglu et. al. show that the Newtonian interpolation polynomial for $n+1$ points can be computed in less than $2\lceil \log (n+1) \rceil + 2$ parallel arithmetic steps using $n(n+1)$ processors and can be implemented as an arithmetic circuit of size $\mathbf{O}(n^2)$. Their assumptions require that all processors be able to perform any of the four arithmetic operations in one unit step and that the processors may be reused.

Improving the Parallel Newtonian Algorithm

The *permanence* property, or the ability to add a new data point and obtain the new interpolating polynomial $p_{n+1}(x)$ on $n+2$ data points without recalculating all the divided differences is an aspect of the Newtonian form that Egecioglu et. al. leave open. From (4) it is clear that the divided differences $f_{i,j}$ for $0 \leq i \leq n$ would remain the same (provided the first $n+1$ data points are kept in the same sequence), and that $n+2$ additional divided differences $f_{n+1,j}$ would have to be computed for $0 \leq j \leq n+1$. This assumes that some of the divided differences are not discarded once $p_n(x)$ is calculated. As stated, the parallel Newtonian algorithm does not retain any information once it has calculated the coefficients of $p_n(x)$, thus, adding one data point requires that the algorithm execute again with $(n+1)(n+2)$ processors. There is no problem if that many processors are available but if not, it would be desirable to retain the capability of computing $p_{n+1}(x)$ in a reasonable amount of time. By sacrificing some storage the number of processors needed for the computation can be reduced significantly without increasing the computation time by more than a few arithmetic steps.

Suppose the parallel Newtonian algorithm is executed again with $n+2$ data points (in the same sequence as in the initial run that generated $p_n(x)$, adding the new data point to the end). Then, from (1) it follows that

$$p_{n+1}(x) = p_n(x) + f_{0,n+1}(x-x_0)(x-x_1)\cdots(x-x_n) \quad .$$

The only difference between $p_{n+1}(x)$ and $p_n(x)$ is in the last coefficient that is calculated. Now from (5) and with $k = n+1$ it follows that

$$f_{0,n+1} = \sum_{j=0}^{n+1} \frac{f_j}{y_{j,0} y_{j,1} \cdots y_{j,n+1}} \quad .$$

Provided that the denominators of the coefficients of $f_{0,n}$ have been previously stored, it is easy to calculate $f_{0,n+1}$.

Theorem 1: The Newtonian interpolating polynomial for $n+2$ points can be computed from the Newtonian interpolating polynomial for $n+1$ points (where the first $n+1$ data points are in the same sequence) in at most $2\lceil \log (n+2) \rceil + 4$ parallel arithmetic steps using $n+2$ processors.

Proof: First, calculate $y_{i,n+1}$ and $y_{n+1,i}$ for $0 \leq i \leq n+1$ followed by $(y_{i,0} y_{i,1} \cdots y_{i,n}) y_{i,n+1}$ for $0 \leq i \leq n+1$ (recall that the left-hand factor is available from the previous computation of $p_n(x)$). This requires two parallel steps and one parallel step, respectively, using $n+2$ processors. Now perform a binary tree multiplication to compute $y_{n+1,0} y_{n+1,1} \cdots y_{n+1,n} y_{n+1,n+1}$, followed by $n+2$ parallel divisions. This requires $n+2$ processors and $\lceil \log (n+2) \rceil + 1$ parallel steps. Finally, use a binary tree addition algorithm to sum the $n+2$ elements and obtain $f_{0,n+2}$. This requires $\lceil \log (n+2) \rceil$ further parallel steps. □

Parallel Lagrangian Interpolation

Egecioglu et. al.'s parallel computation of certain features in the Newtonian basis (1) suggests that similar features in the Lagrangian form (2) might also be computed in parallel. The typical formulation of the Lagrangian representation (c.f. Abramowitz and Stegun [1]) is $p_n(x) = \sum_{i=0}^{n} l_i(x) f_i$ where

$$l_i(x) = \frac{(x-x_0)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n)}{(x_i-x_0)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_n)} \quad \text{for} \quad 0 \leq i \leq n.$$

Using the notation developed earlier it follows from (2) that

$$l_i = \frac{1}{y_{i,0} y_{i,1} \cdots y_{i,n}} \tag{6}$$

for $0 \leq i \leq n$ (recall that $y_{i,i} = 1$). This product is easily computed in $\lceil \log n \rceil$ parallel steps using n processors. $n+1$ such products are required, thus $n+1$ parallel instances of (6) are sufficient to calculate all the l_i's. Thus, $n(n+1)$ processors are needed to compute $p_n(x)$ in the Lagrangian form. The algorithm is given by the following steps, assuming the data point pairs are stored in memory, and $n(n+1)$ processors are available:

Parallel Lagrangian Interpolation

1) Compute $y_{i,j} = x_i - x_j$ for $i,j = 0,1,...,n$, $i \neq j$. $y_{i,i} = 1$ already set.
2) Compute $\frac{1}{l_i} = \prod_{j=0}^{n} y_{i,j}$.
3) Compute and store l_i for $0 \leq i \leq n$.

Theorem 2: The Lagrangian interpolating polynomial for $n+1$ points can be computed in at most $\lceil \log n \rceil + 2$ parallel arithmetic steps using $n(n+1)$ processors and can be implemented as an arithmetic circuit of size $\mathbf{O}(n^2)$.

Proof: Clearly, Step 1 of the above algorithm requires one parallel step while Step 2 takes $\lceil \log n \rceil$ parallel steps using a

binary tree multiplication algorithm. Finally, Step 3 takes one step to perform in parallel a division over $n+1$ processors. The processors involved are assumed to be reusable and able to perform the four basic arithmetic operations. In the case of an arithmetic circuit a binary tree multiplication of n inputs requires a circuit of depth $\lceil \log n \rceil$ and size n. The division step requires one operational node and there are $n+1$ such divisions. The computation of the $y_{i,j}$'s requires $n(n+1)$ operational nodes. Thus, the parallel Lagrangian algorithm for $n+1$ inputs can be performed with a circuit of depth $\lceil \log n \rceil + 2$ and size $\mathbf{O}(n^2)$. □

Why should any of the bases (1), (2) or (3) be used? Why not go directly to the evaluation step with the data point pairs (x_i, f_i) using the basic Lagrange formula $p_n(x) = \sum_{i=0}^{n} f_i \prod_{j=0, j\neq i}^{n} \frac{(x - x_j)}{(x_i - x_j)}$? While this computation could be done in about $3\lceil \log n \rceil + 3$ parallel arithmetic steps with n $(n+1)$ processors, it it not competitive since, as is shown later, $n+1$ such evaluations could be performed by either the parallel Newtonian or parallel Lagrangian algorithm in about the same amount of time.

Still another possibility would be to store the product $l_i f_i$. This increases the interpolation algorithm by one step and decreases the evaluation by one step.

The Parallel Lagrangian Algorithm with Fewer Processors

It has been assumed that there are sufficient processors to handle the entire problem simultaneously. Suppose there are a limited number of processors p, where $p = m(n+1)$ and $1 \leq m < n$ is an integer. All those operations which can be done in one step using n concurrent instances may also be done in at most $\lceil n / m \rceil$ steps.

Theorem 3: The parallel Lagrangian algorithm for $n+1$ data points can be performed in at most $2\lceil (n+1)/m \rceil + \lceil \log m \rceil$ arithmetic steps when $p = m(n+1)$ processors are available.

Proof: First, the $y_{i,j}$'s can be calculated in $\lceil n / m \rceil$ steps using p processors for all $i, j = 0,1,...,n$, $i \neq j$. Now, using $n+1$ concurrent instances of a binary tree multiplication algorithm $n+1$ elements can be multiplied together to compute each $1/l_i$ using m processors in each instance. This is done by breaking the $n+1$ elements into m blocks each with at most $\lceil (n+1)/m \rceil$ elements, and allowing each processor to multiply these elements together. This requires $\lceil (n+1)/m \rceil - 1$ steps. Now the m blocks can be multiplied together in $\lceil \log m \rceil$ steps. Finally, $n+1$ concurrent parallel divisions yields l_i for $0 \leq i \leq n$. □

If the number of available processors p is less than $n+1$ then pieces of each instance may be performed in parallel. However, even the first step of the parallel Lagrangian algorithm, calculating all the $y_{i,j}$'s requires $\lceil n(n+1)/p \rceil$ parallel computations. This is likely be uncompetitive except when p is fairly large since the serial equivalent to this algorithm is of complexity $\mathbf{O}(n^2)$.

Similar results for the parallel Newtonian algorithm are shown by Egecioglu et. al. and are summarized in Table 1.

Permanence Property for Parallel Lagrangian Interpolation

As with the Newtonian form, it is possible to calculate the interpolating polynomial $p_{n+1}(x)$ using $p_n(x)$ in the Lagrangian form. However, the Lagrangian form requires no additional storage to perform the computation.

Theorem 4: The parallel Lagrangian interpolating polynomial for $n+2$ points can be computed from the Lagrangian interpolating polynomial for $n+1$ points (where the first $n+1$ data points are in the same sequence) in at most $\lceil \log (n+1) \rceil + 4$ parallel arithmetic steps using $n+1$ processors.

Proof: The original $n+1$ l_i's need only be divided by $x_i - x_{n+1}$, which takes two parallel arithmetic steps using $n+1$ processors. l_{n+1} can be computed in $\lceil \log (n+1) \rceil + 2$ arithmetic steps, one to calculate in parallel all the $x_{n+1} - x_i$'s for $0 \leq i \leq n$, and the remainder to apply a binary tree multiplication algorithm to obtain $1 / l_{n+1}$ followed by a division to compute l_{n+1}. □

Evaluation of the Lagrangian Interpolating Polynomial

Egecioglu et. al. show that the Newtonian interpolating polynomial of degree n can be evaluated in $2\lceil \log (n+1) \rceil + 2$ parallel arithmetic steps using n processors and can be implemented as an arithmetic circuit of size $\mathbf{O}(n)$. A similar result for the Lagrangian interpolating polynomial can be shown and the algorithm is given by the following steps, assuming the Lagrangian form is stored in memory, and n processors are available:

Parallel Lagrangian Evaluation

1) Compute the prefixes of $\prod_{j=0}^{n-1}(x - x_j)$ and let $z_{i+1} = \prod_{j=0}^{i}(x - x_j)$ for $0 \leq i < n$ and $z_0 = 1$.
2) Compute the prefixes of $\prod_{j=0}^{n-1}(x - x_{n-j})$ and let $\bar{z}_{i-1} = \prod_{j=0}^{n-i}(x - x_{n-j})$ for $1 \leq i \leq n$ and $z_n = 1$.
3) Compute $l_i f_i z_i \bar{z}_i$ for $0 \leq i \leq n$.
4) Sum the $n+1$ terms computed in steps 3) and store the result.

Theorem 5: The Lagrangian interpolating polynomial of degree n can be evaluated in at most $3\lceil \log n \rceil + 9$ parallel arithmetic steps using n processors and can be implemented as an arithmetic circuit of size $\mathbf{O}(n)$.

Proof: Equation (2) shows that for $0 \leq i \leq n$ $(x-x_0) \cdots (x-x_{i-1})(x-x_{i+1}) \cdots (x-x_n)$ must be computed. Naively, the entire product (without omitting the term $x-x_i$) can be computed, and then the required term can be divided out in $n+1$ separate copies. Unfortunately, this introduces a slightly larger error, as redundant floating point operations are performed for each i. With a slight increase in complexity the precision of the data can be preserved.

First, compute the parallel prefixes of the product $(x-x_0) \cdots (x-x_{n-1})$ and let $z_{i+1} = (x-x_0) \cdots (x-x_i)$ for $0 \leq i < n$ with $z_0 = 1$. Then compute the parallel prefixes of the product $(x-x_n) \cdots (x-x_1)$ and let $\bar{z}_{i-1} = (x-x_n) \cdots (x-x_i)$ for $1 \leq i \leq n$ with $\bar{z}_n = 1$. Each parallel prefix calculation requires $\lceil \log n \rceil$ steps, and it is easily seen that $(x-x_0) \cdots (x-x_{i-1})(x-x_{i+1}) \cdots (x-x_n) = z_i \bar{z}_i$ for $0 \leq i \leq n$.

All the terms $x-x_i$ for $0 \leq i \leq n$ can be computed in two parallel steps. Thus, the entire sequence requires n processors to perform two parallel prefix operations on n inputs followed by n processors performing three multiplications in parallel (to compute $l_i f_i z_i \bar{z}_i$ for $0 \leq i < n$) and another three for the $n+1$th term. Finally, a binary addition algorithm can be used to sum the $n+1$ terms in $\lceil \log n \rceil + 1$ parallel steps with n processors.

To prove the result for an arithmetic circuit, observe that both the parallel prefix algorithm, and a binary addition (multiplication) algorithm may be implemented in a circuit with depth at most $\lceil \log n \rceil$ and size $\mathbf{O}(n)$. □

It is left to the reader to show that the evaluation can be done in $2\lceil \log n \rceil + 4$ parallel arithmetic steps if $2(n+1)$ processors are available.

Conclusions and Comments

A fast parallel algorithm for computing the Lagrangian interpolating polynomial similar to that for the Newtonian interpolating polynomial as shown by Egecioglu et. al. has been developed. It remains to be shown that this algorithm is practical (numerically stable). This, however, is left for future investigation. Both methods are competitive with each other since their input constraints and required number of processors are the same, and their time complexities differ only by very small constants for both interpolation and evaluation.

With regard to the parallel Newtonian algorithm a small, but useful change is proposed that allows the use of the permanence property of the Newtonian form. This change only increases the constant term in the time complexity of the algorithm and doubles the storage requirements from $n+1$ real values to $2(n+1)$. However, it decreases the required number of processors from $(n+1)(n+2)$ to $n+2$. The permanence property is also demonstrated for the Lagrange form and is shown to require about half the time that the Newtonian form does, with only $n+1$ processors.

The Lagrangian representation also has a property that may be valuable in certain applications. Once the l_i's have been calculated, the Lagrangian interpolating polynomial can be evaluated for different sets of data points f_i without recalculating the l_i's. While the same result can be obtained from the parallel Newtonian algorithm by modifying it so that with a slight increase in its storage requirements it can compute new divided differences in only $\lceil \log (n+1) \rceil + 1$, it requires the use of the full $n(n+1)$ processors. Once computed the evaluation would only take another $2\lceil \log (n+1) \rceil + 2$. Thus, the whole process would be six arithmetic steps faster than simply evaluating the Lagrangian interpolating polynomial at new values f_i with only n processors.

Any problem which requires the computation of a large number of interpolating polynomials on a small number of points, and then requires the repeated evaluation of these polynomials at a large number of test points, without requiring these polynomials in classical form should benefit greatly from either of these algorithms. The nature of the application will determine which of the two methods is most suitable.

While useful as a model, the shared memory PRAM architecture is not realistic as far as interprocessor communication is concerned. Egecioglu et. al. provide an example implementation of their algorithm on a cube-connected computer. Because of the similarities between the two methods it should be possible to implement the Lagrangian algorithm in much the same way. A shuffle-exchange architecture might also be suitable since Schwartz [7] has shown how to perform the parallel prefix algorithm in $\mathbf{O}(log\ n)$ time.

In terms of circuits the binary addition (or multiplication) algorithm (circuit) can also be implemented using the parallel prefix algorithm (circuit). Thus all the circuits described may be uniformly implemented using just a parallel prefix circuit. The depth of these circuits remains the same, but the size of these circuits increases by a constant (multiplied) amount.

Table 1 summarizes the results and compares them with the corresponding results of Egecioglu. et. al.

Problem	Number of Processors	Parallel Newtonian	Parallel Lagrangian
Interpolating $n+1$ data points	$n(n+1)$	$2\lceil \log (n+1) \rceil + 2$	$\lceil \log n \rceil + 2$
	$m(n+1)$ $1 \le m < n$	$6\left\lceil \frac{n+1}{m} \right\rceil + 2\lceil \log m \rceil - 4$	$2\left\lceil \frac{n+1}{m} \right\rceil + \lceil \log m \rceil$
	$p < n+1$	$\geq \mathbf{O}(\frac{n(n+1)}{p})$	$\geq \mathbf{O}(\frac{n(n+1)}{p})$
Permanence Property	$n+2$	$2\lceil \log (n+2) \rceil + 4$	$\lceil \log (n+1) \rceil + 4$
Evaluation	n	$2\lceil \log (n+1) \rceil + 2$	$3\lceil \log n \rceil + 9$
	$2(n+1)$	$2\lceil \log (n+1) \rceil + 2$	$2\lceil \log n \rceil + 4$

Table 1
Summary of Results

Acknowledgement

Thanks to Dr. X. Li and Dr. S. Cabay for their comments, and to C. Smith for text processing and system help.

References

[1] M. Abramowitz and I. A. Stegun, eds., Handbook of Mathematical Functions , Dover, (1972), pp. 878.

[2] O. Egecioglu, E. Gallopoulos and C. K. Koc, Fast and Practical Parallel Polynomial Interpolation , Center for Supercomputing Research & Development, The University of Illinois, Report No. 646, (Jan, 1987).

[3] C. P. Kruskal, L. Rudolph and M. Snir, "The Power of Parallel Prefix," IEEE Trans. on Comp. , (Oct, 1985), pp. 965-968.

[4] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," J. ACM , (Oct, 1980), pp. 831-838.

[5] M. T. McClellan, "The Exact Solution of Systems of Linear Equations with Polynomial Coefficients," J. ACM , (Oct, 1973), pp. 563-588.

[6] J. H. Reif, "Logarithmic Depth Circuits for Algebraic Functions," SIAM J. Comput. , (Feb, 1986), pp. 231-242.

[7] J. T. Schwartz, "Ultracomputers," ACM Trans. on Program. Lang. and Syst. , (Oct, 1980), pp. 484-521.

[8] C. N. Zhang, B. Shirazi and D. Y. Y. Yun, "Parallel Designs for Chinese Remainder Conversion," Proc. of the 1987 Inter. Conf. on Parallel Processing , (Aug, 1987), pp. 557-559.

A NEW APPROACH TO THE IMPLEMENTATION OF MULTIDIMENSIONAL SIGNAL PROCESSING ALGORITHMS

Winser E. Alexander and Seong-Mo Park
Department of ECE, Box 7911
North Carolina State University
Raleigh, N. C. 27695
Tel. (919) 737 -2336

Jung H. Kim
Department of EE
North Carolina A&T State University
Greensboro, N.C. 27411
Tel. (919) 334 - 7760

Abstract -- This paper presents a new approach to the implementation of multidimensional (N-D) digital signal processing algorithms in a multiprocessor environment. We develop a computational primitive for N-D signal processing algorithms using a state space model. We then map the state space model onto a linear finite state machine and implement the computational primitive in the combinational logic block of the linear finite state machine. The state variables are stored in the delay elements of the machine. With our approach data communications requirements among processors have been minimized without increasing the computational requirements for the given algorithms. We present an efficient multiprocessor system using our approach for implementing N-D signal processing algorithms.

Introduction

Extensive research and development have been devoted to N-D digital signal processing over the last decade [1,2]. Practical applications include military intelligence, remote sensing, industrial inspection, robot vision, data compression for communications, processing biomedical images for diagnosis, character recognition, finger prints, weather forecasting, etc. These applications are computationally intensive and require substantial data communications. In many cases, the reduction of computer hardware cost makes the design of special purpose computer systems tailored to the specific requirements of a given class of algorithms practical [3]. However, the complexity of most digital signal processing tasks is such that real-time implementation using single processor system will not be feasible in the near future [4]. In this paper, we have concentrated on the development of algorithms which can be effectively used for high speed N-D digital signal processing in a multiprocessor or multicomputer environment. The data communication requirements for many digital signal processing algorithms are of the same order of magnitude as the computational requirements. In particular, the transfer of a data word between chips in a multiple chip system can require as much time as for a 16 by 16 integer multiply (typically on the order of 50 to 100 nanoseconds). Thus, data communication requirements should be given at least equal consideration in developing algorithms for multiprocessor systems.

Algorithm Decomposition

A discrete, linear, shift-invariant (DLSI) system is a discrete system for which the system parameters do not vary with changes in the independent variables (time, space, distance, range, etc.). Many practical digital signal processing and digital control problems can be represented as DLSI systems. Our approach is to design computationally efficient algorithms for N-D DLSI systems which can be implemented on a multiprocessor system with localized data communication requirements. We also emphasize the preference for on chip data communications compared to data communications between chips.

In the 1-D case, partial fraction expansion or factorization can be used to partition the system into an efficient computational structure for a multiprocessor system. However, N-D systems can not be partitioned in this way, except for the special cases of product or sum separable systems [5]. An arbitrary multivariate transfer function can not be factored into distinct poles and zeros and can not be expanded into partial fractions. These approaches to developing a parallel or cascade implementation of transfer functions can not be extended to N-D systems. Thus, we must explore alternate means of partitioning N-D DLSI systems.

We use a state space representation as a vehicle to aid in the decomposition of N-D DLSI systems for implementation on a multiprocessor system. The state space representation provides the potential for minimizing the data communications requirements for a given algorithm without increasing computational complexity. Other advantages of the state space implementation over direct implementation include decreased sensitivity to parameter variations and improved performance when finite arithmetic is used. In order to clearly explain the concepts involved in this approach, we first discuss the state space implementation of 2-D DLSI systems. We then show that the concepts used in this special case can be extended to the N-D case (N > 2).

State Space Implementation of 2-D DLSI Systems

A set of finite difference equations is one of the forms commonly used for representing DLSI systems. A general order, causal 2-D DLSI system with quarter plane support can be represented by finite difference equations [1] as given by equation (1)

$$g(m,n)=\sum_{j=0}^{L}\sum_{k=0}^{L} a(j,k)f(m-j,n-k)-\sum_{\substack{j=0\\ j+k>0}}^{L}\sum_{k=0}^{L} b(j,k)g(m-j,n-k). \quad (1)$$

An input-output relationship between the transform of the input sequence $F(z_1,z_2)$ and the transform of the output sequence $G(z_1,z_2)$ can be written as equation (2)

This work was supported by the National Security Agency under contract number MDA904-86-H-003.

$$G(z_1,z_2) = a(0,0)F(z_1,z_2) \qquad (2)$$

$$+ \sum_{\substack{j=0 \\ j+k>0}}^{L} \sum_{k=0}^{L} \Big[a(j,k)F(z_1,z_2) - b(j,k)G(z_1,z_2)\Big] z_1^{-j} z_2^{-k}.$$

Figure 1 gives a block diagram representation of the 2-D DLSI system partitioned as specified by equation (2). Note that the number of vertical delays is the same as the order of the filter in the z_2 variable which is the minimum possible number. We can obtain the state space representation by assigning a horizontal state variable to the input of each of the horizontal delay blocks (associated with z_1 variable) and by assigning a vertical state variable to each of the vertical delay blocks (associated with the z_2 variable). For convenience, we define

$$y(n_1,n_2) = q_L(n_1-1,n_2) + r_L(n_1,n_2-1) \qquad (3)$$

$$c_{1I} = a(j,k) - a(0,0)b(j,k); \quad c_{2I} = - b(j,k).$$

Then, the typical vertical state equation for the 2-D DLSI system can be represented by

$$r_I(n_1,n_2) = c_{1I}f(n_1,n_2) + c_{2I}y(n_1,n_2) + q_J(n_1-1,n_2) \qquad (4)$$

$$+ r_{I-1}(n_1,n_2-1).$$

In a similar way, the typical horizontal state variable is given by

$$q_I(n_1,n_2) = c_{1I}f(n_1,n_2) + c_{2I}y(n_1,n_2) + q_J(n_1-1,n_2). \qquad (5)$$

The output equation is given by

$$g(n_1,n_2) = a(0,0)f(n_1,n_2) + y(n_1,n_2). \qquad (6)$$

The equation for the vertical state variables as given in equation (4) is a computational primitive for the 2-D DLSI system since the vertical state variables, the horizontal state variables and the output can be mapped into this equation with a suitable interchange of variables.

State Space Implementation of N-D DLSI Systems

The general multivariable difference equation for the causal, discrete, linear, shift invariant (DLSI) system with first section support (the N-D equivalent of quarter plane support) is given by [1]

$$g(n_1,...,n_N) = \sum_{j_1=0}^{L_1} \cdots \sum_{j_N=0}^{L_N} a(j_1,...,j_N) f(n_1-j_1, \ldots, n_N-j_N)$$

$$- \sum_{\substack{j_1=0 \\ j_1+\cdots+j_N>0}}^{L_1} \cdots \sum_{j_N=0}^{L_N} b(j_1,...,j_N) g(n_1-j_1, \ldots, n_N-j_N). \qquad (7)$$

The input $f(n_1,...,n_N)$ is assumed to be sampled at uniform intervals in each of the independent variables and $g(n_1,...,n_N)$ is the corresponding output. The parameters $a(j_1,...,j_N)$ and $b(j_1,...,j_N)$ are coefficients which determine the characteristics of the algorithm. Since the coefficients can take on arbitrary values as appropriate, this equation can represent many common N-D problems.

We can extend the approach used for the 2-D DLSI system to obtain a state space representation of the N-D DLSI system. Using the transform of the input sequence $F(\mathbf{Z})$ and the transform of the output sequence $G(\mathbf{Z})$, we can show the input and output relationship as follows:

$$G(\mathbf{Z}) = a(\mathbf{O})F(\mathbf{Z})$$

$$+ \sum_{\substack{k_1=0 \\ k_1+\cdots+k_N>0}}^{L_1} \cdots \sum_{k_N=0}^{L_N} \Big[a(\mathbf{K})F(\mathbf{Z}) - b(\mathbf{K})G(\mathbf{Z})\Big] \mathbf{Z}^{-\mathbf{K}} \qquad (8)$$

$$\mathbf{Z} = (z_1,...,z_N); \quad \mathbf{K} = (k_1,...,k_N); \quad \mathbf{Z}^{-\mathbf{K}} = (z_1^{-k_1} \cdots z_N^{-k_N}).$$

We can extend the block diagram structure presented above for the 2-D DLSI system to N-D systems and obtain the desired state space representation by assigning a state variable to the input of each delay block in each tuple. After doing this the most complicated section will have a single delay element in each tuple. Other subsections may be equivalent or they may be simpler in that one or more of the delays may be missing. For convenience, we define

$$y(n_1,...,n_N) = \sum_{k=1}^{N} q_{k,L_k}(...,n_k-1,...) \qquad (9)$$

$$c_{1I} = a(\mathbf{K}) - a(\mathbf{O})b(\mathbf{K}); \quad c_{2I} = - b(\mathbf{K}).$$

Then, the typical state equation for the N-D DLSI system is given by

$$q_{i,I}(n_1,...,n_N) = c_{1I}f(n_1,...,n_N) + c_{2I}y(n_1,...,n_N) \qquad (10)$$

$$+ q_{i,I-1}(...,n_i-1,...) + \sum_{\substack{k=1 \\ k\neq i}}^{N} q_{k,J}(...,n_k-1,...).$$

The output equation is given by

$$g(n_1,...,n_N) = a(0,...,0)f(n_1,...,n_N) + y(n_1,...,n_N). \qquad (11)$$

Equation (10) can be considered to be a computational primitive for the N-D DLSI system since the state variables for each tuple and the output can be mapped into it with a suitable interchange of the variables. Thus, N-D DLSI systems can be implemented by solving equation (10) repeatedly with proper parameter substitutions. Note that equation (10) is a generalization of the 2-D computational primitive as given in equation (4).

Multiprocessor System for N-D Signal Processing

We now consider the implementation of DLSI systems in a multiprocessor environment. We can derive the computational primitives for N-D signal processing from equation (10). These computational primitives then form the basis for the design of special purpose processors for the implementation of the overall system. These computational primitives require two multiplications and N+1 additions to calculate a state variable or an output for a N-D DLSI system. We can use a tree structure to implement them with two multipliers and N+1 adders arranged in a pipeline and parallel fashion as shown in Figure 2. All inputs in Figure 2 use shift registers or queues to hold appropriate data and

system coefficients for the computation. The current N-D input data $f(n_1,...,n_N)$ is stored in F register and the temporary value $y(n_1,...,n_N)$ is stored in Y register for the current computation and replaced by new data for the next computation. The system coefficients c_{1I} and c_{2I} are stored in C_1 and C_2, respectively. The coefficients are arbitrary and can be changed by software. Therefore an arbitrary N-D DLSI system can be implemented using this structure. The Q_k's are the queues holding the previous state variables for the current computation and are updated for each input. The pipeline will have $\lceil \log_2(N+4) \rceil$ stages to calculate a state variable or an output and will produce a result at every cycle once all the pipeline stages have been filled. Since the computational primitives depend only upon the dimensions of the system and not upon the order of the system nor upon the size of the input data, we can design programmable special purpose processors to implement the computational primitives.

There are $(L+1)^N$ -1 state variables to be calculated for each input data value for an N-D Lth order system. Thus, the computational requirements are too high for a single processor system or for a general purpose multiprocessor system to implement the DLSI systems in real-time at typical sampling rates. In this paper, we consider a multiprocessor system for the 2-D and 3-D cases. A spatial domain digital filter system for image processing will be presented as an example of 2-D signal processing and a motion picture analysis system will be presented as an example of a 3-D system. Higher dimensional systems can be developed using the same concept and the general computational primitive.

A Multiprocessor System for 2-D signal Processing

A multiprocessor architecture to implement the spatial domain digital filter in real-time has been proposed by Kim and Alexander [6]. The proposed system requires ten processors for a second order filter, each of which implements the computational primitive as given in equation (4). Each processor has two multipliers and three adders arranged in a pipeline and parallel fashion and can generate a state variable or an output at every cycle. These processors are connected in a linear array, as shown in Figure 3, where each processor is assigned to a separate row of data and the vertical state variables are passed to the next processor in the array for use in computing the output for the subsequent row of data. In this scheme, the processor assigned to the first row is available to be assigned the eleventh row because it has completed the processing for the first row when the eleventh row becomes available. Thus the vertical state variables which are output from the tenth processor can be inputs to the first processor for use in computing the outputs for the eleventh row.

This scheme can be extended to higher order spatial domain filters by increasing the number of processor in the linear array. For example, seventeen processors are required for a third order filter and twenty six processors are required for a fourth order filter, etc. This multiprocessor system is practical for implementing spatial domain filters in real-time for images with 512 rows by 512 pixels per row at 30 frames per second. The time interval between pixels for such an image is 127 nanoseconds if no allowance is made for latency periods. Thus, a practical computation cycle time for a system design is 100 nanoseconds. It is currently possible to design 16 by 16 integer multipliers and adders with cycle times significantly less than 100 nanoseconds [7].

A Multiprocessor System for 3-D Signal Processing

The above multiprocessor system for 2-D can be extended to 3-D signal processing. To implement 3-D DLSI systems each processor would be required to implement equations with two multiplications and four additions in order to update any state variable or to compute the output. There are eighteen q1 horizontal state variables, six q2 vertical state variables, two q3 frame state variables and an output to be computed for each input for a second order 3-D system. Considering the pipeline fill up time, twenty nine cycles will be required to calculate and update the output and all the state variables for a single input data for a second order 3-D system. Since outputs are generated at every 29th cycle, twenty nine processors can be arranged to achieve the throughput of one output per cycle for real-time signal processing.

A multiprocessor system for second order 3-D DLSI system is given in Figure 4. Each processor is assigned to a separate row of data and the q2 state variables are passed to the next processor through Q2BUF. In this multiprocessor system, the q3 state variables are transferred between processors through Q3BUFFER with one frame delay. Q3BUFFER is a first-in-first-out type memory buffer which could be partitioned by row for the row by row data processing operation. This multiprocessor system is practical for implementing 3-D digital filters in real-time for moving images with 512 rows by 512 pixels per row changing 30 frames per second. A higher order system can be built by adding processors and buffer memories.

Conclusion

We proposed a new approach to the design of a multiprocessor system which can implement multidimensional DLSI systems in real-time. This approach also has the advantage that the complexity and number of computations per input does not increase as the size of the input data is increased. Thus, very large multidimensional input data can be processed in near real-time with such a system. We are currently developing a special purpose digital signal processor and an associated multiprocessor system for the real-time implementation of 2-D DLSI systems using the approach proposed in this paper.

References

[1] D. E. Durgeon and R. M. Mersereau, Multidimensional Digital Signal Processing, Prentice-Hall, Englewood Cliffs, N.J., 1984.

[2] S. G. Tzafestas, Ed., Multidimensional Systems, Marcel Dekker, Inc., N.Y., 1986.

[3] S. Y. Kung, H. J. Whitehouse, and T. Kailath, Eds., VLSI and Modern Signal Processing, Prentice-Hall, Englewood Cliffs, N.J., 1985.

[4] K-S. Lin, G. A. Frantz, and R. Simar,Jr., "The TMS320 family of digital signal processors," Proc. IEEE, vol.75, no.9, pp.1143-1159, 1987.

[5] M. Morf, B. C. Levy, and S. Y. Kung, "New results in 2-D systems theory, Part I : 2-D polynomial matrices, factorization, and coprimeness, Proc. IEEE, vol.65, no.6, pp.861-872, 1977.

[6] J. H. Kim and W. E. Alexander, "A multiprocessor architecture for two-dimensional digital filters," IEEE Trans. Comp., vol.C-36, no.7, pp.876-884, 1987.

[7] D. A. Henlin, M. T. Fertsch, M. Mazin, and E. T. Lewis," A 16 bit X 16 bit pipelined multiplier macrocell," IEEE Journal of Solid-State Circuits, vol.SC-20, no.2, pp.542-547, 1985.

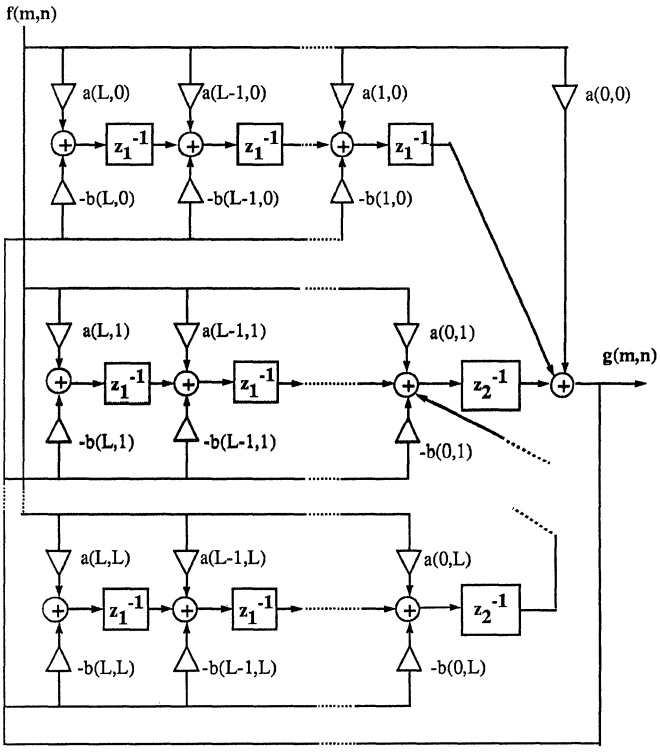


Figure 1. A block diagram of general order 2-D DLSI system.

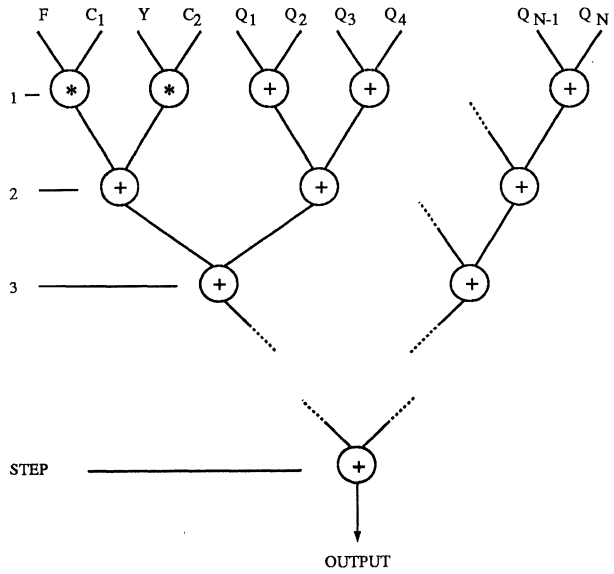


Figure 2. Computational primitive for N-D DLSI systems.

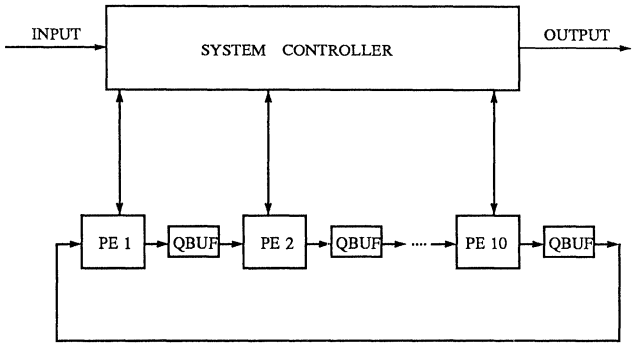


Figure 3. A multiprocessor system for a second order 2-D DLSI system.

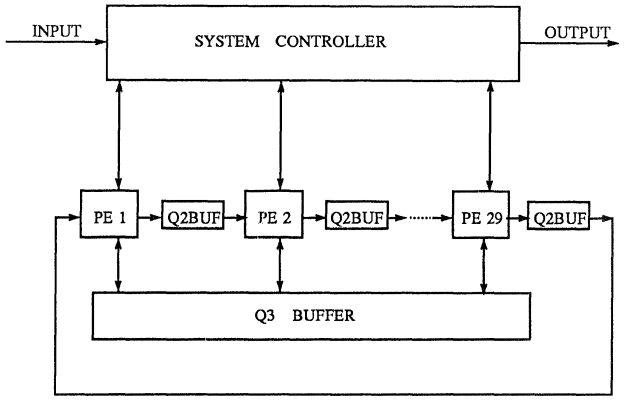


Figure 4. A multiprocessor system for a second order 3-D DLSI system.

Parallel Depth First Search on the Ring Architecture*

Vipin Kumar†, V. Nageshwara Rao and K. Ramesh
Department of Computer Sciences,
University of Texas at Austin,
Austin, Texas 78712

Abstract

This paper presents the implementation and analysis of parallel depth-first search on the ring architecture. At the heart of the parallel formulation of depth-first search is a dynamic work distribution scheme that divides the work between different processors. The effectiveness of the parallel formulation is strongly influenced by the choice of the work distribution scheme. In particular, a commonly used work distribution scheme is found to give very poor performance on large rings(> 32 processors). We present a new work distribution scheme that is better than the work distribution scheme used by other researchers, and gives good performance even on large rings (128 processors). We introduce the concept of iso-efficiency function to characterize the effectiveness of different work distribution schemes.

1 Introduction

Depth-First Search(DFS) is a general technique used in Artificial Intelligence for solving a variety of problems in planning, decision making, theorem proving, expert systems, etc. [11,5]. It is also used under the name of backtracking to solve various combinatorial optimization problems and constraint satisfaction problems. Execution of a Prolog program can be viewed as depth-first search of a proof tree. Iterative-Deepening algorithms perform cost-bounded DFS in successive iterations to solve discrete optimization problems[4] and theorem proving[12]. A major advantage of depth-first search strategy is that it requires very little memory. Since many of the problems solved by DFS are highly computation intensive, there has been a great interest in developing parallel versions of depth-first search [3,14,6,2].

This paper presents the implementation and analysis of parallel depth-first search on the ring architecture. At the heart of the parallel formulation of depth-first search is a dynamic work distribution scheme that divides the work between different processors. The effectiveness of the parallel formulation is strongly influenced by the choice of the work distribution scheme. In particular, a most commonly used work distribution scheme is found to give very poor performance on large rings(> 32 processors). We present a new work distribution scheme that is better than the work distribution scheme used by other researchers [2,13,14], and gives good performance even on large rings (128 processors). The performance is tested by solving the 15-puzzle problem[11]. The ring architecture is embedded on an 128-node Intel Hypercube. We introduce the concept of iso-efficiency function (representing the required growth in problem size with respect to number of processors to maintain the efficiency) to characterize the effectiveness of different work distribution schemes. We have also implemented parallel depth first search on Hypercube and shared-memory architectures[9,10]. The ring architecture is important because it is simple to construct and is highly scalable.

A detailed treatment of depth-first search is given in [11,5]. The unit of computation in a search algorithm is the time taken for one node expansion. The total time taken by a sequential search algorithm is roughly proportional to the total number of nodes it expands. Total number of nodes expanded by a search algorithm for a particular instance is called the **problem size** W of the instance. If the depth of the search tree is d, then the **effective-branching factor** b is defined as $log_d W$.

2 A Parallel of Depth-First Search

We parallelize depth-first search by sharing the work done among a number of processors. Each processor searches a disjoint part of the search space in a depth-first fashion. When a processor has finished searching its part of the search space, it tries to get an unsearched part of the search space from other processors. When a solution is found, all of them quit. If the search space is finite and has no solutions, then eventually all the processors would run out of work, and the (parallel) search will terminate.

Since each processor searches the space in a depth-first manner, the (part of) state-space to be searched is easily represented by a stack. The depth of the stack is the depth of the currently explored node, each level of the stack keeps track of untried alternatives. Each processor maintains its own local stack on which it executes DFS. When the local stack is empty, it sends a request for work to another processor. Each processor periodically checks for incoming work requests. If it has untried alternatives in the stack, then it sends some of them to the requesting processor;[1] otherwise it sends a null message back. In our implementation, at the start of each iteration, all the search space is given to one processor, and other processors are given null space (i.e., null stacks). From then on, the state-space is divided and distributed among various processors. A detailed treatment of our parallel formulation can be found in [9,10].

In the first formulation we implemented, an idle processor requests for work only from an immediate neighbor. This is

*This work was supported by Army Research Office grant # DAAG29-84-K-0060 to the Artificial Intelligence Laboratory, and Office of Naval Research Grant N00014-86-K-0763 to the computer science department at the University of Texas at Austin.

†Tel:512-471-9571,Arpanet: Kumar@sally.utexas.edu

[1] It is important to make sure that the work given out is not too small (otherwise the requesting processor will be out of work soon again) or too large (otherwise the donor processor will be out of work soon). The best strategy is to try to give nearly half of the local work.

a simple and natural scheme and has been used by many researchers (see Section 6). Other work distribution schemes are possible, and will be considered later.

3 Performance of Parallel DFS

To test the effectiveness of Parallel DFS, we have used it to solve the 15-puzzle problem [11]. The 15-puzzle is a 4x4square tray in which are placed 15 square tiles. The remaining sixteenth square is uncovered. Each tile has a number on it. A tile that is adjacent to the blank space can be slid into that space. A game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around. The 15-puzzle problem is particularly suited for testing the effectiveness of parallel DFS, as it is possible to create search spaces of different sizes (W) by choosing appropriate starting configurations. IDA* is the best known sequential algorithm to find optimal solutions for the 15-puzzle problem[4]. It is much faster than simple depth-first search, as it is able to use the Manhattan distance heuristic [11] to focus the search. Since each iteration of IDA* is a cost-bounded depth-first search, a parallel formulation of IDA* is easily obtained.

We implemented Parallel cost-bounded depth-first search (i.e., the last iteration of IDA*) to find all optimal solutions of the 15-puzzle problem on 1-ring and 2- ring embedded on an Intel Hypercube. On 1-ring (i.e., the unidirectional ring), a processor could ask for work from only one neighbor. On a 2-ring, a processor could ask for work from both of its neighbors. We ran our algorithm on a number of problem instances given in Korf's paper [4]. As shown in Fig 1, we are able to get linear speedup up to 16 processors, but for more processors, the performance is not very good. The performance of a 2-ring is better than a 1-ring, but the maximum speedup obtained is only 25 even on 128 processors. In general, for a given number of processors, we get more speedup for bigger problems and less speedup for smaller problems. The size of a problem is determined by it's sequential execution time. The average execution time of the problems for which the speedups of Fig. 3 were obtained is roughly 200 minutes. On smaller problems (sequential execution time 16 minutes), the maximum speedup for 128 processors for 2-ring is approximately 10. But even for very large problems, we were not able to get speedups significantly higher than 25. It seems that parallel depth-first search with the simple work distribution scheme is incapable of effectively utilizing larger rings. The next section presents an analysis of this scheme which explains this poor performance.

4 Analysis of Performance

In this section we analyze the performance of parallel cost-bounded DFS. We assume that the effective branching factor of the cost-bounded search space is greater then $1 + e$ (where e is a positive constant). To avoid speedup anomalies, we assume that both sequential and parallel DFS search the whole cost bounded space for all solutions. All these conditions are met by the parallel formulation presented in Section 3.

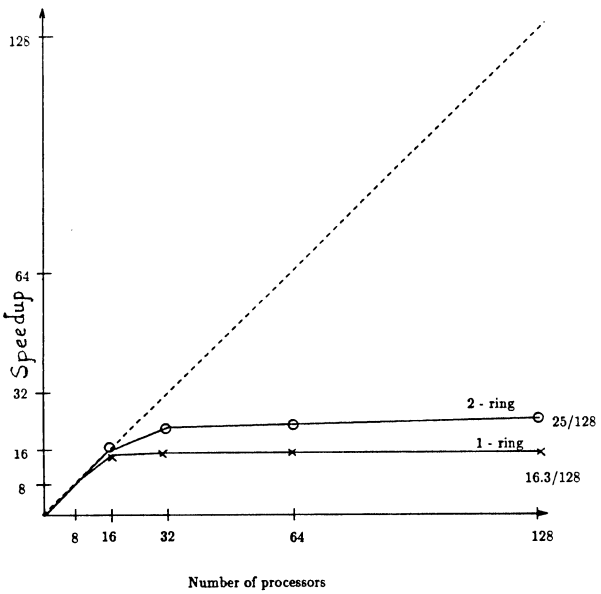


Figure 1: Average speedup vs Number of processors for parallel cost-bounded depth-first search on a ring embedded in Intel Hypercube. Sequential Exec. time $\simeq$ 10500 secs, problem size $\simeq$ 9 M nodes

4.1 Definitions and Terminology

1. Running time T_N: is the execution time on N processors. T_1 is the sequential execution time.

2. Computation time T_{calc}: is the sum of the time spent by all the processors in useful computation. Since, both sequential and parallel versions search exactly the same cost-bounded space (to find all optimal solutions),

$$T_{calc} \text{ on } N \text{ processors } = T_{calc} \text{ on } 1 \text{ processor } = T_1$$

3. Communication time T_{comm}: is the sum of the time spent by all processors in communicating with neighboring processors, waiting for arrival messages, time in starvation, etc. For single processor execution, $T_{comm} = 0$. Since, at any time, a processor is either communicating or computing,

$$T_{comm} + T_{calc} = N * T_N$$

4. Speedup S: is the ratio $\frac{T_1}{T_N}$.

5. Efficiency E: is the speedup divided by N. E denotes the effective utilization of computing resources.

$$E = \frac{S}{N} = \frac{1}{1 + \frac{T_{comm}}{T_{calc}}}$$

6. Unit Computation time U_{calc}: is the mean time taken for 1 node expansion.

7. Unit Communication time U_{comm}: is the mean time taken for sending a work request to a processor and receiving a response (work or a null message). In the work distribution scheme of section 3, U_{comm} is a fixed constant (determined by the speed of the communication).

Figure 2: Linear Chain of processors

4.2 Iso-efficiency Functions

As discussed in the previous section, the efficiency obtained in parallel DFS is determined by the number of processors and the problem size.[2] For a given problem size W, increasing the number of processors N causes the efficiency to decrease because T_{comm} increases while T_{calc} remains the same. For a fixed N, increasing W improves efficiency because T_{calc} increases and the work distribution scheme with α–splitting does not cause a proportionate increase in T_{comm} . If N is increased, then we can keep the efficiency fixed by increasing W. The rate of increase of W with respect to N is dependent upon the architecture and the work distribution algorithm. The required rate of growth of W w.r.t N (to keep efficiency fixed) essentially determines the scalability of the architecture for the work distribution algorithm. For example, if W is required to grow exponentially w.r.t. N, then it would be difficult to utilize the architecture for a large number of processors. On the other hand, if W needs to grow only linearly w.r.t N, then the work distribution scheme is highly suited for the architecture. If W needs to grow as f(N) to maintain an efficiency E, then f(N) is the **iso-efficiency function** and the plot of f(N) w.r.t N is the **iso-efficiency curve.**

Next we derive the iso-efficiency function of parallel cost-bounded DFS for 1-ring. The analysis for 2-ring is similar and is left out. We present a theoretical model that give us bounds on total communication time T_{comm} in terms of problem size W and number of processors N for different work distribution schemes. Predictions from our model seem to closely agree with experimental data, hence we feel that the model is reliable.

4.3 Iso-efficiency Analysis of the simple work distribution scheme

Consider a linear chain of N processors of Fig. 2. A 1–ring is a linear chain with a fold back from processor N-1 to 0. In a 1–ring a processor can get work from its left neighbor and send work to its right neighbor. Initially W work is available in processor 0. In order to achieve good work distribution every processor needs to get roughly $\frac{W}{N}$ for itself[3]. Suppose that whenever work W is split between a donor and a requester, then the requester gets at most αW (for some constant α such that $0 < \alpha < 1.0$). Then

Maximum piece of work coming into processor 0 is W
Maximum piece of work coming into processor 1 is αW
Maximum piece of work coming into processor i is $\alpha^i W$
From the above, we can see that in order to get $\frac{W}{N}$ work

Processor i has to get at least $\frac{\frac{W}{N}}{\alpha^i W}$ transfers ($i \geq 0$).

$$Hence\ the\ total\ number\ of\ stack\ transfers \geq \sum_{i=0}^{N-1} \frac{1}{N\alpha^i}$$

$$= \frac{1}{N}\sum_{i=0}^{N-1} \beta^i\ (where\ \beta = \frac{1}{\alpha}) = \frac{\beta^N - 1}{\beta - 1} * \frac{1}{N}$$

$$T_{comm} = U_{comm} * \frac{\beta^N - 1}{\beta - 1} * \frac{1}{N}\ (lower\ bound)$$

$$T_{calc} = U_{calc} W$$

$$Efficiency = \frac{1}{1 + \frac{T_{comm}}{T_{calc}}} = \frac{1}{1 + \frac{U_{comm}}{U_{calc} NW} * \frac{\beta^N - 1}{\beta - 1}}$$

For constant efficiency,

$$U_{calc} NW = U_{comm} \frac{\beta^N - 1}{\beta - 1}\ i.e.,\ W = \Omega(\frac{\beta^N}{N})$$

(since U_{comm} and U_{calc} are constant)

Thus the iso-efficiency function is exponential.[4] The Iso-efficiency function for 2-ring can be obtained similarly, and is also exponential. This explains the poor performance of 1-ring and 2-ring.

4.4 An Improved Strategy

In the previous work distribution scheme, we restricted communication to occur only between immediate neighbors of the ring. The analysis in the previous section clearly indicates a weakness due to this: the total count of stack transfers grows exponentially in a ring of processors because the size of the work pieces coming into successive processors decreases geometrically (in the ratio $1, \alpha, \alpha^2, ...$). To solve this problem, we designed the following work distribution scheme.

In this scheme, we designate a special processor that selects the target for each requesting processor. This special processor maintains a variable I whose value denotes the next donor processor. Whenever a processor needs work, it sends a message to the special processor, which returns the current value of I and also increments it. The requesting processor now sends a request for work to processor I.

This work distribution scheme appears to have lots of overhead, as even to decide the identity of the next donor, a processor needs to wait for O(N) time. Actual request for work and receiving a response again takes O(N) time. On the other hand, requesting work and receiving a response takes only a constant time in the simple scheme. But, as the analysis of the next section shows, the new scheme needs to make far fewer requests for work. Therefore it has a substantially better iso-efficiency function and speedup performance.

4.5 Iso-efficiency analysis of the Improved Scheme

Let ϵ be the minimum amount of work transferable. (The absolute minimum amount of work transferable is one node expansion. If we give out work only from levels that are above a CUTOFF depth, then ϵ can be increased by increasing the

[2]It is also determined by the architecture; eg., hypercube and shared memory architectures provide better efficiencies for parallel DFS [10]. In this paper we restrict our discussion to the ring architecture only.

[3]This is true only if the efficiency is high. Hence the analysis given here is not valid for "low-efficiency" iso-efficiency curves

[4]Since the value of T_{comm} used in the analysis is only a lower bound, the actual iso-efficiency function can be worse than exponential.

CUTOFF.) We now present an upper bound on the number of stack transfers.

Let us assume that in every V(N) requests made for work, every processor in the system is requested at least once. Clearly, $V(N) \geq N$. In general, V(N) depends on the load balancing algorithm. Recall that in a transfer, work (w) available in a processor is split into two parts (αw and $(1-\alpha)$w), and one part is taken away by the requesting processor. Hence after a transfer, neither of the two processors (donor and requester) has more than $(1-\alpha)$w work (assuming without loss of generality that $\alpha \leq 0.5$). The process of work transfer continues until work available in every processor is less than ϵ. Initially processor 0 has W units of work, and all other processors have no work.

It is easy to see that after $(log_{\frac{1}{1-\alpha}} \frac{W}{\epsilon})V(N)$ requests, maximum work available in any processor is less than ϵ.

Hence, the total number of transfers $\leq V(N) log_{\frac{1}{1-\alpha}} \frac{W}{\epsilon}$

$$T_{comm} \simeq U_{comm} * V(N) log_{\frac{1}{1-\alpha}} \frac{W}{\epsilon} \ (upper\ bound)$$

$$T_{calc} = U_{calc} W$$

$$Efficiency = \frac{1}{1 + \frac{T_{comm}}{T_{calc}}}$$

$$= \frac{1}{1 + \frac{U_{comm} * V(N) log_{\frac{1}{1-\alpha}} \frac{W}{\epsilon}}{U_{calc} * W}}$$

For the improved work distribution scheme, V(N) = N, and $U_{comm} = O(N)$. Hence for iso-efficiency,

$$W \sim O(N^2) log W \ or \ W \sim O(N^2 log N)$$

This iso-efficiency function is much better than β^N. We implemented the scheme and tested it's performance. As shown, in the Fig. 3, the speedups are substantially higher than the previous scheme.

4.6 Finkel and Manber's Scheme

Finkel and Manber used a different work distribution scheme in their implementation of parallel depth-first search [2]. In their scheme, each processor maintains a local variable, target, to point to a donor processor. Target is incremented (modulo N) every time the processor seeks work. We can compute the iso-efficiency function of this scheme by following the method in section 4.5. For this scheme, V(N) = N^2 in the worst case.[5] But U_{comm} is still O(N). Hence the iso-efficiency function can be as bad as $O(N^3 logN)$.

The superiority of our improved work-distribution scheme over this and the first scheme is clearly seen in the speedup curves of Fig 3. Initially our second scheme is slightly worse than the other two schemes due to the extra overhead of requesting the value of target before requesting for work. But, for larger number of processors, our second scheme makes sufficiently fewer requests than the other schemes, and hence gives higher speedups.

5 Related Research.

Many researchers have implemented parallel DFS on the ring architecture and studied its performance for around 16–20 processors. Monien[13] and Wah[14] present parallel depth-first search procedures on a ring network. The work distribution schemes in these formulations is very similar to the first scheme presented in this paper. From the analysis of Section 4.3 (and our experiments) it is clear that this work distribution scheme is not able to provide good speedup on large rings.

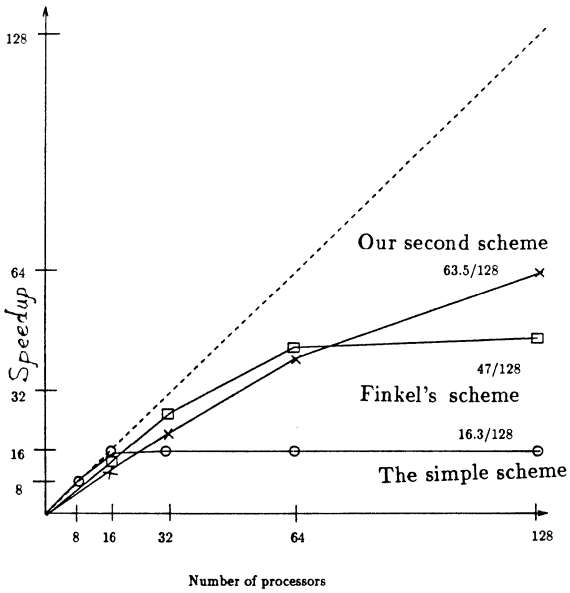


Figure 3: Average speedup vs Number of processors for parallel cost-bounded depth-first search on a ring embedded in Intel Hypercube. Sequential Exec. time $\simeq$ 10500 secs, problem size $\simeq$ 9 M nodes

Manber presents an abstract model in [8] that captures the distribution of work being done in parallel depth-first search. For this model, Manber presents different work distribution schemes and computes lower bounds on the amount of interference in a shared-memory system. (Part of the analysis presented in Section 4.5 uses the same technique that Manber used for the analysis of interference). Manber's analysis served as a basis for the design of parallel depth-first search scheme presented by Finkel and Manber in [2]. This scheme has a better iso-efficiency function ($O(N^3 \log W)$ worstcase) for the ring than the simple work distribution scheme (see section 4.6). But this function is worse than the iso-efficiency function ($N^2 logW$) of the improved scheme in Section 4.4. Superiority of our improved scheme is clearly seen in the speedup curves of Fig 3.

6 Conclusions.

We have presented experimental and analytical evaluation of a number of work distribution schemes used in parallel depth-first search on the ring architecture. We found that the choice of the work distribution algorithm has a significant impact on the performance of the parallel depth-first search algorithm. We have introduced the concept of iso-efficiency function to characterize the effectiveness of different work distribution schemes. Table 1 shows iso-efficiency functions of parallel depth-first search for different work distribution schemes. The development of the new work distribution scheme for the ring was motivated by the iso-efficiency analysis of the other two work distribution schemes. Even though, the new scheme appeared to have a lot

[5] This result was proved by Manber[8] while analyzing the memory interference in shared memory architectures.

Iso-efficiency Function	*Load balancing scheme*
β^N	Section2,Wah[14],Monien[13]
$N^3 logN$	Finkel and Manber[2]
$N^2 logN$	The improved scheme (Section4.4)

Table 1: Iso-efficiency functions of different work-distribution schemes.

of overhead, it had a better iso-efficiency function, and was expected to perform better on larger rings. We were pleased to find that the experimental results were in close agreement with our theoretical results.

The performance of parallel depth-first search is also greatly dependent upon the architecture. Our experimental results and iso-efficiency analysis shows that the hypercube and shared-memory architectures are significantly better than the ring. In[10] we also present a work distribution scheme that has almost optimal performance on shared-memory/ω-network-with-message-combining architectures (such as RP3[1]). The iso-efficiency function has been found useful in characterizing the scalability of many other parallel algorithms as well.

Acknowledgements: We would like to thank Ralph Brickner and Randy Michelson of Los Alamos National Lab for providing access to a 128-processor Intel Hypercube.

References

[1] G. F. Pfister et al. The ibm research parallel processor prototype (rp3). In *Proceedings of International conference on Parallel Processing*, pages 764–797, 1985.

[2] Raphael A. Finkel and Udi Manber. Dib - a distributed implementation of backtracking. *ACM Trans. of Progr. Lang. and Systems*, 9 No. 2:235–256, April 1987.

[3] M. Imai, Y. Yoshida, and T. Fukumura. A parallel searching scheme for multiprocessor systems and its application to combinatorial problems. In *IJCAI*, pages 416–418, 1979.

[4] R.E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985. Also a chapter in 'Search and Artificial Intelligence',Vipin Kumar and Laveen Kanal Eds, Springer-Verlag,1987(to appear).

[5] Vipin Kumar. Depth-first search. In Stuart C. Shapiro, editor, *Encyclopaedia of Artificial Intelligence: Vol 2*, pages 1004–1005, John Wiley and Sons, Inc., New York, 1987.

[6] Vipin Kumar and Laveen N. Kanal. Parallel branch-and-bound formulations for and/or tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6, November 84.

[7] T. H. Lai and Sartaj Sahni. Anomalies in parallel branch and bound algorithms. In *Proceedings of International conference on Parallel Processing*, pages 183–190, 1983.

[8] Udi Manber. On maintaining dynamic information in a concurrent environment. *SIAM J. of Computing*, 15 No. 4:1130–1142, 1986.

[9] V. Nageshwara Rao, and Vipin Kumar Parallel Depth-First Search on Multiprocessors, Part I: Implementation. To appear in **International Journal of Parallel Programming**, 1988.

[10] Vipin Kumar and V. Nageshwara Rao Parallel Depth-First Search on Multiprocessors, Part II: Analysis. To appear in **International Journal of Parallel Programming**, 1988.

[11] Nils J. Nilsson. *Principles of Artificial Intelligence*, Tioga Press, 1980.

[12] M.E. Stickel and W.M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *IJCAI*, pages 1073–1075, 1985.

[13] Monien B. and Vornberger O. The Ring Machine. Technical Report No. 27, Dept. of Math./Computer Science, University of Paderborn, Dec. 1985, Also in Computers and Artificial Intelligence, 3(1987).

[14] Benjamin W. Wah and Y. W. Eva Ma. Manip - a multicomputer architecture for solving combinatorial extremum-search problems. *IEEE Transactions on Computers*, c–33, May 1984.

[15] V. Nageshwara Rao and V. Kumar. Superlinear speedup in depth-first search. In *Submitted for publication*, 1988. Also AI Lab TR, University of Texas at Austin, March 88.

PARALLEL ALGORITHMS FOR ANSWERING THE TAUTOLOGY QUESTION

Professor Gary D. Hachtel and Peter H. Moceyunas
Department of Electrical and Computer Engineering
University of Colorado, Boulder, CO 80309

Abstract - This paper presents parallel multilevel and 1-level algorithms for answering the tautology question for Boolean Networks. These parallel algorithms recursively bipartition and assign work to available processors. The results from the implementation on a network of Sun 3 workstations, on an Encore Multi-Max multiprocessor and on an Intel hypercube computer are presented. A model for predicting speedups in a general divide and conquer setting is developed which includes communication delays and start up costs of the host multiprocessor system. This model accurately predicts speedups for problems with balanced recursion trees and can be used to characterize the parallelization potential of the host systems. For the specific context of Boolean tautology checking, this model can be used to show that unit parallel efficiency (i.e., linear speedup) is indeed possible, but that parallel efficiency is limited in the 1-level algorithm by tree imbalance and by size. The addition of dynamic processor scheduling to overcome tree imbalance is presented, along with preliminary results.

Introduction

In the computer aided design of VLSI circuits, the designer may describe pieces of the digital system as a Boolean function (1-level function) or as a set of interconnected Boolean functions (Multi-Level function). Applications like test generation, logic minimization and equivalence checking, which use as input these Boolean functions, require an algorithm to check if these functions are tautologous. Note:

> A Boolean function or network of interconnected Boolean functions is tautologous if and only if all its outputs are equal to one for all possible combinations of its inputs.

We will call an algorithm which determines whether or not a Boolean function is tautologous a **tautology algorithm**. Single level and multilevel tautology algorithms have been previously studied by [4] , [6], [9] , [7], [5] and [8]. These algorithms utilize Shannon cofactoring with recursive divide and conquer strategies. Due to the nature of the problem run times are $O\left[q\,2^q\right]$, where q is the number of inputs to the Boolean function, although heuristics are often effective in reducing run times for certain problem classes.

With the exception of [9], these previous tautology algorithms were implemented for serial processing. We have developed from a 1-level tautology algorithm [4], and a multilevel algorithm [7], corresponding parallel algorithms that use a recursive bipartitioning scheme to divide the work among processors. These new parallel tautology algorithms have been implemented on a network of SUN 3 workstations, on an Encore shared memory multiprocessor and on an Intel hypercube computer. With regard to purpose they are similar to that of [9], but differ with regard to domain of applicability and in method employed (cf., the conclusions section presented below).

The paper begins by briefly presenting how parallel computations can occur in divide and conquer algorithms and then develops a model for expected speedups of the parallel algorithm for a class of divide and conquer algorithms. The serial and parallel tautology algorithms using a static processor assignment are presented with further development of the speedup model specifically for tautology. This parallel speedup model is designed for the problem of Boolean Tautology checking. It is, however, generalizable with minor variations, to any of the following divide and conquer applications: nested dissection for linear algebraic equations, binary sorting, shortest path calculations, and VLSI placement. The results of the experiments run on the three computer systems are then given along with comparisons to the speedup models developed for both multilevel and 1-level tautology algorithms. Further improvements in run time for the parallel algorithms through dynamic processor scheduling are discussed followed by some conclusions and further work.

Divide and Conquer Algorithms

Given a particular problem, a recursive divide and conquer algorithm will generate a set of recursive calls. This set of recursive calls can be represented in a tree data structure, where a node in the tree represents a recursive call to the routine. We will assume that the divide and conquer algorithm is a binary recursive algorithm.

The manner in which a problem is divided and the size of the resulting smaller problems depends on the nature of the particular divide and conquer algorithm. Algorithms which can divide problems into equal pieces will always produce a full recursion tree. Other algorithms, like tautology, attempt to create two equally sized problems but cannot guarantee it and thus unbalanced recursion trees may occur.

Concurrency in Divide and Conquer

An underlying implication in a divide and conquer algorithm is that the computational problem at each node of the recursion tree is independent of the computations at all other nodes on the same level in the tree, thus the work done in each node could be performed in parallel. If an unlimited number of processors were available, all nodes on each level of recursion could be processed concurrently. It is clear that in the worst case an exponential number of processors is required.

Parallel processing is still possible with a limited number of processors. Let N, the number of processors, be a power of 2. For each of the first $\log_2 N$ levels of recursion there are enough processors to process in parallel all nodes of a level. For each level 0 through $\log_2 N$, assign a processor to each node on the level. Each processor assigned to perform the work of a node at level $\log_2 N$ is also assigned the work of all the node's progeny. Figure 1 shows a recursion tree and the processor assignment for a four processor system. Here, we assign a processor to each subproblem generated at level $\log_2 N$ in the recursion tree and let each processor find its solution through serial computation. Note that processor assignment is fixed relative to the recursion tree. Later, in the results section, we will discuss the use of dynamic processor scheduling.

Clearly this strategy will have low parallel efficiency if the maximum level of recursion is not greater than $\log_2 N$. It is also evident that as the maximum level of recursion (problem size) increases, the parallel efficiency approaches 100 percent for problems with balanced recursion trees. The details of how the parallel efficiency depends on the problem size are not immediately evident. This is the motivation for developing a model to predict speedup as a function of maximum recursion depth. With such a model, we can get some insight into how speedups will increase with problem size and determine the potential of this algorithm. Furthermore, parallel processing overheads, such as communications and startup costs, can be included in the model which can characterize the different computer environments.

Speedup Model for Divide and Conquer Algorithms

Our development of the speedup model begins with the following observation: Not all divide and conquer algorithms produce a recursion tree with a predictable structure. This unpredictable tree structure inhibits the development of accurate timing models for all possible data inputs to such an algorithm [3]. However, an accurate model can be made if we restrict ourselves to those algorithms or sets of inputs to an algorithm which produce recursion trees with some predictable structure, i.e., which satisfy suitable constraints. In order to develop our speedup model, the following assumptions are made about the divide and conquer algorithm and the structure of its recursion tree:

1) The algorithm produces a full binary recursion tree for any input data.

2) All nodes on a particular level of the tree will perform the same amount of work.

3) The amount of work performed by a node in the recursion tree is $\eta + \gamma$, where η is equal to $\frac{k}{2}$ times the η work done by its parent node ($0<k<2$) and γ is some constant for the particular problem.

In order to develop an accurate timing model, we assume communications and process start up costs (i.e. delays) to be non-zero. Clearly, communications and start up costs are a function of the type of computer system being used. Let α be a parameter representing the way in which the characteristics of a computer system influence these costs. In general both costs depend on the characteristics of the particular divide and conquer algorithm employed. They also depend on the level of recursion. Let A be a parameter embodying the properties of the particular algorithm and denote the particular recursion level in the algorithm by i. Then the communications and start up costs for a level i node in the recursion tree can be modeled by the functions $C(\alpha, A, i)$ and $I(\alpha, A, i)$ respectively. For any given computer system, the communications delay between different pairs of processors can be different. Therefore the communications costs are also a function of the different processor pairs. It may also be possible that start up cost is influenced by the identity of the processor starting the process and the processor which the new process is being started on. Therefore let $C_{m,n}(\alpha, A, i)$ represent the communications costs for processor m to transmit data to processor n on level $i+1$ in the recursion tree. Similarly, let $I_{m,n}(\alpha, A, i)$ represents the start up cost for processor m to start a new process on processor n on level $i+1$ in the recursion tree.

We also define a quantity $\Delta t(\alpha,A)$ which is an initialization cost. In the general case this parameter can be both a function of the particular algorithm and of the computer system. This cost represents the serial work a program may have to perform prior to the execution of the recursive divide and conquer part of the program.

Before developing the timing models, we formally define speedup. The speedup due to N processors, S_N, is

$$S_N = \frac{t_1}{t_N} , \tag{1}$$

where t_1 is the run time for a serial divide and conquer algorithm and t_N is the run time for an N processor parallel divide and conquer algorithm. In order to obtain a speedup model, models for the serial and parallel run times must be developed.

Serial Run Time Model

Let l be the maximum number of levels of recursion in the tree. We define $\eta_0 + \gamma$ to be the amount of work performed at the level 0 node in the recursion tree (assumption 3). Following the algorithmic constraints given above, the amount of work done by any node on a level can be found in terms of η_0 and γ by recursively applying assumption 3. The amount of work at any node on level i is $\eta_0\left[\frac{k}{2}\right]^i + \gamma$. There are 2^i nodes at a level i and l levels in the tree, thus the total amount of work done or the serial run time is

$$t_1 = \sum_{i=0}^{l}\left[\eta_0\left(\frac{k}{2}\right)^i + \gamma\right] 2^i .$$

This can be further simplified to

$$t_1 = \eta_0 \sum_{i=0}^{l} k^i + \gamma\left[2^{l+1} - 1\right] .$$

Finally, if there is some initial cost associated with starting the serial program on this computer system, then an additional cost $\Delta t(\alpha,A_1)$ must be added to the serial run time, where A_1 represents the parameter embodying the properties of the serial algorithm. The serial run time becomes

$$t_1 = \eta_0 \sum_{i=0}^{l} k^i + \gamma\left[2^{l+1} - 1\right] + \Delta t(\alpha,A_1) . \tag{2}$$

Parallel Run Time Model

The parallel run time model is developed as follows. Assume, as in the serial run time model, l is the maximum level of recursion and $\eta_0 + \gamma$ is the amount of work done at the level 0 node in the recursion tree. Let the number of processors be N, such that $N = 2^j$ for some $j \in \{ 1,2,... \}$. The N processor run time can be determined by first calculating the run time for those levels in which the nodes are outnumbered by the processors, and then determining the run time while all N processors are being used.

Let $\mu=\min(l,\log_2 N)$. During the first μ-1 levels in the recursion tree the work to be counted to obtain the run time for the first μ-1 levels is the sum of the work done by nodes along any path from the root to a node on level μ-1 , including the last node. This sum is explicitly

$$\sum_{i=0}^{\mu-1}\left[\eta_0\left(\frac{k}{2}\right)^i + \gamma\right]$$

This simplifies to

$$\eta_0 \sum_{i=0}^{\mu-1}\left(\frac{k}{2}\right)^i + \mu\gamma . \tag{3}$$

The parallel work done in the remaining levels of the recursion tree, μ through l, is the sum of the work done to finish a μ level node and all its progeny. This is similar to the sum found for the serial algorithm.

$$\sum_{i=\mu}^{l}\left[\eta_0\left(\frac{k}{2}\right)^i + \gamma\right] 2^{i-\mu}$$

simplifies to

$$\frac{\eta_0}{2^\mu} \sum_{i=\mu}^{l} k^i + \gamma\left[2^{l+1-\mu} - 1\right] . \tag{4}$$

As in the computation of the serial time, a $\Delta t(\alpha,A_N)$ term is required to include any initial start up costs for the N processor program. Combining equations (3) and (4) and $\Delta t(\alpha,A_N)$ produces the concurrent run time t_N, which is

$$t_N = \eta_0 \sum_{i=0}^{\mu-1}\left[\frac{k}{2}\right]^i + \mu\gamma + \frac{\eta_0}{2^\mu}\sum_{i=\mu}^{l} k^i + \gamma\left[2^{l+1-\mu} - 1\right] + \Delta t(A_N,\alpha) . \tag{5}$$

This model for the parallel algorithm does not include communications and process start up costs. To include these costs we observe that each node in the recursion tree from the root to level $\mu - 1$, where $\mu=\min(l,\log_2 N)$, will have a start up and communications cost associated with it. In the model for t_N developed above (5), it was possible to assume that processing at each level of the recursion tree finished at the same time because of the assumptions 1 and 2. If either the communications or start up costs are not equivalent for different pairs of processors, then this assumption becomes invalid. After level μ-1, no more new processes are started and the time for each processor to complete its work is equivalent. However, processors may not begin processing the level μ nodes at the same time. The last processor to reach a level μ node will be the last processor to finish processing. Therefore the overall run time is governed by this last processor. The sum of the communications and start up costs along a path from the root to a level μ node is the extra time required to reach that level μ node. The path with the largest sum will be the last node to begin processing at the μ level. Given a node z on level μ-1, the binary recursion tree follows a unique path L from the root node to z . Let Z be the set of all such paths for a particular set of μ-1 level nodes. Each node in a particular path L is on a recursion level i and has a processor m assigned to it. A new process on some processor n will be started by processor m. Let $p(L)$ be the set of all triples (m,n,i) corresponding to nodes on the path L. Therefore the delay due to communications and start up for an N processor algorithm is

$$\max_{L\in Z}\left[\sum_{(m,n,i)\in p(L)}\left[C_{m,n}(\alpha,i,A_N)+I_{m,n}(\alpha,i,A_N)\right]\right] \tag{6}$$

Combining equations (5) and (6) the general model for the concurrent run time now becomes

$$t_N = \eta_0 \sum_{i=0}^{\mu-1}\left[\frac{k}{2}\right]^i + \mu\gamma + \frac{\eta_0}{2^\mu}\sum_{i=\mu}^{l} k^i + \gamma\left[2^{l+1-\mu} - 1\right] + \Delta t(A_N,\alpha)$$

$$+ \max_{L\in Z}\left[\sum_{(m,n,i)\in p(L)}\left[C_{m,n}(\alpha,i,A_N)+I_{m,n}(\alpha,i,A_N)\right]\right] \tag{7}$$

Model for Speedup

The speedup now can be formulated by substituting the equations (2) and (7) for t_1 and t_N in equation (1). This yields

$$S_N = \frac{\eta_0 \sum_{i=0}^{l} k^i + \gamma\left[2^{l+1}-1\right] + \Delta t(\alpha, A_1)}{\eta_0 \sum_{i=0}^{\mu-1}\left[\frac{k}{2}\right]^i + \mu\gamma + \frac{\eta_0}{2^\mu}\sum_{i=\mu}^{l} k^i + \gamma\left[2^{l+1-\mu}-1\right] + \Delta t(A_N, \alpha)}$$

$$+ \max_{L \in Z}\left[\sum_{(m,n,j)\in p(L)}\left[C_{m,n}(\alpha, i, A_N) + I_{m,n}(\alpha, i, A_N)\right]\right]. \quad (8)$$

Thus a general model for speedup has been developed. It can be easily specialized to a specific algorithm and computer system. This model is a function of $A_1, A_N, k, N, \eta_0, \gamma, \alpha, l$ and initialization, startup and communications costs. After the tautology algorithm has been presented, further simplifications will be made to the model which are due to specific properties of the algorithm.

The Tautology Algorithms

In this section we will present both a serial and a parallel tautology algorithm in coarse detail. The algorithms for multilevel functions and 1-level functions are very similar. Since the 1-level algorithm is a special case of the multilevel algorithm, we will present the multilevel algorithm with notations where the 1-level version would differ.

Serial Tautology

The multilevel tautology algorithm described by [7] is a recursive divide and conquer algorithm. A high level outline of the algorithm is shown in figure 2. The term **cover** is defined to be the description of either a 1-level or multilevel Boolean function. A brief description of each routine will be given here. For a more complete description of the multilevel and 1-level algorithms see [7] and [4] respectively.

The routine SPECIAL_CASES corresponds to the termination step of any divide and conquer algorithm [11]. The outputs of cover F are scanned: if any are set to 0 or all are set to 1, then a 0 or 1 are returned correspondingly. Otherwise a -1 is returned. In the 1-level algorithm, other attributes of 1-level cover are checked which can give an immediate answer to the tautology question. The 1-level algorithm then calls UNATE_REDUCTION [4] (not shown) if SPECIAL_CASES returns a -1. This routine checks the cover for special unate properties and if they exist allow the recursion tree to be trimmed. If an answer cannot be determined by SPECIAL_CASES (or by UNATE_REDUCTION in the 1-level case), then an input variable is heuristically selected by the routine SELECT_SPLIT. Due to the nature of the tautology problem, an optimal choice of the splitting variable is not guaranteed by SELECT_SPLIT. Next the routine COFACTOR is called twice, which creates two new covers by cofactoring F with respect to x_j and $\bar{x}_j$. In the multilevel algorithm, the cofactoring procedure copies the cover F to create new multilevel covers F_{x_j} and $F_{\bar{x}_j}$ and asserts the primary input x_j to a 1 or 0. The SIMULATE routine then propagates the value towards the primary outputs through logic simulation. In the 1-level algorithm, this step is not necessary because the input functions are only one level. ML_TAUTOLOGY is recursively called using the two simplified functions. If either returns 0, then the original function F is not tautologous. Otherwise F is tautologous.

The major differences between the 1-level algorithm and multilevel algorithm is that the simulation step is not required for the 1-level algorithm and the 1-level algorithm uses a unate filter. We will refer to the work done by the SIMULATE routine as *simulation work* and the work done by SPECIAL_CASES, SELECT_SPLIT and COFACTOR as *simplification work.*

The Parallel Tautology Algorithm

The parallel tautology algorithm is shown in figure 3. This algorithm is similar to the serial tautology algorithm up to the last call to COFACTOR. After the last call to COFACTOR, the algorithm branches to follow either a parallel processing path or a serial processing path, based on the number of processors and the current level of recursion. If the number of processors is greater than or equal to the maximum number of nodes in the tree at the next level of recursion, then the parallel processing path is taken. Otherwise, the parallel algorithm takes the serial processing path and behaves exactly like the serial algorithm. In the concurrent processing path, a new process is started on another processor. Let m be a label for the current process and n be a label for the new process on another machine. Next, process m sends the cover $F_{\bar{x}_j}$ to process n. Process m calls SIMULATE and then ML_TAUTOLOGY with F_{x_j} as the input cover, and similarly process n calls SIMULATE and ML_TAUTOLOGY with $F_{\bar{x}_j}$ as the input cover. Like the serial tautology algorithm, if either function is found not to be the tautology, then the cover F is not the tautology. If both are found to be the tautology, then F is the tautology. Note that the answer produced by process n must be sent back to its parent process m and in turn process m must wait for the answer from process n. The variable *il* indicates at what level in the recursion tree a processor began processing. If the current level of recursion is equal to *il*, then this indicates that the result from tautology should be sent back to it's parent process. Otherwise, the answer should be simply returned. The parameter *parent* contains the information regarding the identity of a process's parent. Each process must know this information in order to send the tautology answer back to it's parent.

Speedup Model for Tautology

Using the characteristics of the multilevel tautology algorithm and of our computer systems the speedup model developed previously (8) can be simplified.

The work done by a node in the recursion tree for multilevel tautology consists of simulation and simplification work. The simplification work at the level 0 node for problems which result in full recursion trees is proportional to $j2^j$, where j is the number of inputs in the cover. Further j is equal to l, the maximum level of recursion. The simplification work decreases at each level of recursion, thus it corresponds to the η term in assumption 3. We let

$$\eta_0 = l2^l. \quad (9)$$

The simulation work can be roughly modeled as the number of functions in the Boolean network, f, times a computer dependent factor $\omega(\alpha)$. Again for a set of problems which produce full recursion trees, f is proportional to l, thus

$$\gamma = \omega(\alpha) l. \quad (10)$$

Also, since the SELECT_SPLIT routine attempts to choose a variable which after cofactoring and simplification, produces two networks equal in size, the simplification work of a child node is 1/2 of that done by its parent. Thus we will let

$$k = 1. \quad (11)$$

Approximate expressions for $C_{m,n}(\alpha, i, A_N)$, $I_{m,n}(\alpha, i, A_N)$ $\Delta t(\alpha, A_1)$ and $\Delta t(\alpha, A_N)$ are now given. For the computer systems involved, the start up cost is assumed to be independent of the nature of the divide and conquer algorithm A_N, the recursion level i, and the identities of the pair of processors involved, (m,n). Thus we assume that start up cost depends only on α, the nature of the particular computer system being used. Therefore $I_{m,n}(\alpha, i, A_N)$ is modeled as

$$I_{m,n}(\alpha, i, A_N) = s(\alpha), \quad (12)$$

for all m, n, i and A. The parameter $s(\alpha)$, is the computer dependent scaling factor for startup costs. The distance between any pair of Sun 3 workstations is approximately equal to the distance between any other pair. Also, the processors were selected on the hypercube such that communications occurred only between nearest neighbor processors. Therefore we assume that the communications costs between any pair (m,n) is independent of m and n, but proportional to the amount of data transmitted. The amount of data transmitted in the tautology algorithm is proportional to the amount of simplification work done by the node receiving the data, which decreases as 2^{-i}. Thus the communications cost can be stated as

$$C_{m,n}(\alpha, i, A_N) = c(\alpha)\eta_0\left[\frac{k}{2}\right]^{i+1} \quad (13)$$

for any level $i-1$ processor m which is transmitting data to a level i processor n. The parameter $c(\alpha)$ is a computer system dependent scaling factor for communications costs. The parameters $\Delta t(\alpha, A_N)$ and $\Delta t(\alpha, A_1)$ are assumed to be constants which are only a function of the particular computer system. They are not a function of the number of processors involved and therefore we let

$$\Delta t(\alpha) = \Delta t(\alpha, A_N) = \Delta t(\alpha, A_1), \tag{14}$$

where $\Delta t(\alpha)$ is the computer dependent initialization scale factor. Now substituting (9-14) into (8) and simplifying sums we get

$$S_N = \frac{l+1+\frac{\omega l(2^{l+1}-1)+\Delta t(\alpha)}{l2^l}}{(c(\alpha)+2)\left[1-\frac{1}{N}\right]+\frac{l-\log_2 N+1}{N}+\frac{s(\alpha)\log_2 N+\Delta t(\alpha)+\omega l(\frac{2^{l+1}}{N}-1+\log_2 N)}{l2^l}} \tag{15a}$$

for $l > \log_2 N$ and

$$S_N = \frac{l+1+\frac{\omega l(2^{l+1}-1)+\Delta t(\alpha)}{l2^l}}{c(\alpha)[1+(\frac{1}{2})^l]+2-(\frac{1}{2})^l+\frac{s(\alpha)l+\Delta t(\alpha)+(l+1)\omega l}{l2^l}} \tag{15b}$$

for $l \le \log_2 N$. The speedup model has been reduced to a function of l, N, $c(\alpha)$, $s(\alpha)$ $\omega(\alpha)$, and $\Delta t(\alpha)$. For a particular computer system, N, $c(\alpha)$, $s(\alpha)$, $\omega(\alpha)$ and $\Delta t(\alpha)$ will be fixed and the multilevel speedup model will be only a function of l. Figure 4 shows a plot of speedup as predicted from the model versus l, the level of recursion, for $N = 8$. The plot contains a family of curves for different values of the startup cost scaling factor $s(\alpha)$ given specific values for the communications and initialization scaling factors $c(\alpha)$ and $\Delta t(\alpha)$ and $\omega(\alpha)=0$. Several observations can be made from this plot. Note as $s(\alpha)$ grows larger, the larger the problem (greater level of recursion) required to obtain a speedup greater than 1.0. As the problem size grows, the effect of the start up cost will become negligible and the speedups will be practically the same as for $s(\alpha)=0$. Other plots using larger values of $c(\alpha)$, not shown here, produce curves with similar shapes but with decreased speedups as expected. In a similar manner it was found that the speedup model predicts that the initialization cost $\Delta t(\alpha)$ effects only those problems which have relatively few levels of recursion. As $\Delta t(\alpha)$ is increased the speedup moves closer to 1.0 for small values of l. As either l or $s(\alpha)$ is increased the the effect of $\Delta t(\alpha)$ becomes less significant.

To obtain the speedup model for the 1-level algorithm, we set $\omega(\alpha)$ to zero, using the fact that there is no simulation work performed by that algorithm.

Figure 5 shows a plot of speedup versus maximum level of recursion for $N = 8$, $c(\alpha) = .55$, $s(\alpha) = 8000$ and $\Delta t = 100$. The plot contains several curves, each with a different value for $\omega(\alpha)$. As $\omega(\alpha)$ becomes larger, the predicted speedups for all problem sizes increases. Thus the multilevel algorithm is expected to see higher speedups than the 1-level algorithm. In fact, if the $\omega(\alpha)$ is large enough, linear speedups would be obtainable even for small problems.

The Computer Systems

The tautology algorithms were implemented on a network of eight Sun 3 workstations, a twenty processor Encore Multi-max and on a 32 node Intel hypercube. The implementations on the Multi-Max and Hypercube used the mechanisms for parallel computations provided by Encore and Intel. Two software package developed at the University of Colorado, Boulder, DPUP [1] and GRAIL [10], were used in the implementation on the Sun 3 network. The packages provide routines which which perform the necessary tasks for distributed processing. The 1-level and multilevel versions were initially implemented using DPUP. The dynamic scheduling versions of tautology used GRAIL.

Initial Results

This section presents a range of topics related to the data collected from the 3 computer systems. The description of the measured data, descriptions of the test input covers are given before the actual results are presented.

Definitions

Tests were done which would measure the real time required by the serial and concurrent tautology programs to produce an answer. On the Sun 3 network and on the Encore Multi-Max the run times include all system and communications time which result from processes being started and from data transfer. The run times from the Intel hypercube include all system and communications times from data transfer but *not process start up times*. The time to input the description of the cover was not included in the run time on any computer system. Each set of input data for tautology was run several times through the programs and the best/shortest times were used to calculate the speedup numbers. Program run times were measured on all computer systems at times when the systems had a low probability of usage by other users. If other users were observed, then those measurements were discarded.

Test Data

The example 1-level covers in [4] were used for testing 1-level tautology. Multilevel covers generated by BOLD [2] (using the 1-level covers as input) were used for testing multilevel tautology. Also, special tautologous 1-level were generated which have the following properties:

1) A full binary recursion tree is produced when run through tautology.
2) Whenever a recursion occurs the cover is split into two equally sized covers which are one-half the size of the original.
3) The depth of the recursion tree is the worst case, i.e., the maximum possible for the number of inputs.

The first and second properties fulfill the constraints of the speedup model developed previously. The final property indicates that for a particular number of inputs, the tautology algorithm will be required to do the worst case (i.e. the maximum amount) work. A set of similar multilevel covers were generated which have properties 1 and 3. **Covers with these properties are important because they match the model for speedup described previously which allows the empirical determination of the communications and startup parameters.** The 1-level covers which have these properties are called covers of all minterms. The multilevel covers with these properties are created from XNOR of the outputs of two n-bit adder circuits and will be referred to as the adder circuits.

Results from 1-level Tautology

Variations in the run times for a given input (cover) were observed, especially on the Sun 3 network. Communications collisions are probably the reason this occurs and it is more observable on the Sun 3 network because its communications is much slower relative to its processor speed as compared to the Intel hypercube or Encore Multi-max systems. Further, larger variances in run times are observed between the results from tests using 4 and 8 processors on the Sun network. This again is most likely due to the increase in the number of processors which causes more messages, more message collisions and thus a larger variance from run to run.

The maximum level and the total number of leaves in the recursion tree were recorded for each cover. Figure 6 contains plots of the speedup model for 1-level tautology ($\omega(\alpha)=0$) developed previously and the speedups obtained with the covers of all minterms for the Sun 3 network, Intel hypercube and Encore multi-max (labeled Sun 3, Hypercube, MultiMax respectively). The Sun 3 network and Encore results were collected using 8 processors and the Intel results using 32 processors. Values for $c(\alpha)$, $s(\alpha)$ and $\Delta t(\alpha)$ that best matched the data were chosen (table 1). The model corresponds very well with the special covers of all minterms. The models using the same values for the 3 parameters but for different values of N also matched very well, but are not shown here. Figure 7 shows a plot of the speedup model and the speedups from the other covers from the Sun 3 Network using 8 processors. Similar plots for the Encore and Intel systems are very similar and omitted.

As one would expect, the model applied to the three computer systems produced very different values for the scaling factors $c(\alpha)$, $s(\alpha)$ and $\Delta t(\alpha)$ (see table 1). The start up cost on the Sun 3 network is expected to be much larger due to the communications between the user program and the DPUP System [11]. The Encore computer system uses a fork call and has a much faster communications so it should have a less costly start up. The run times from the Intel system did not include processor start up times and therefore $s(\alpha)=0$. The communications scaling factor $c(\alpha)$ is not nearly as different, but again is higher on the Sun 3 network as would be expected. These differences in the speedup and communications scaling factors account for the better speedups achieved by the Encore computer versus the Sun 3 network.

Those covers which generate points below and to the right of the model's curve in figure 7, produced recursion trees with less than 2^l leaf nodes, where l is the maximum level of recursion. This indicates that the recursion tree was not full and therefore these covers do not satisfy the constraints of the model. The accuracy of the model can be increased across all problems by introducing the concept of the "effective level of recursion". A plot of speedup versus the effective level of recursion is shown in figure 8 for each of the 3 computer systems using the maximum number of processors available. The effective level of recursion is

defined as the $\log_2 R$, where R is the number of leaf nodes in the recursion tree . The model's curve is also shown in these plots. Since the model assumes a full and balanced tree, the effective level of recursion is equal to the maximum level of recursion assumed by the model, and thus the model's curve does not change. Notice that the data points have moved much closer to the model's curve. By using the effective level of recursion we are attempting to model an unbalanced recursion tree by a balanced tree with a maximum level of recursion $\log_2 R$, which is less than the maximum level of recursion for the unbalanced tree. It appears that this approximation works quite well.

In several cases it was observed that a significant increase in speedup was not observed when the number of processors was increased for a given input cover. The speedup model reveals that this behavior is to be expected for those problems which recur only a few levels. Figure 9 contains a plot containing a family of speedup model curves resulting from using different values of N and the same values for $c(\alpha)$, $s(\alpha)$ and $\Delta t(\alpha)$ found for the Sun 3 network. There is a region where the models for different N processors are very close together and if the tautology algorithm achieves a maximum level of recursion in this region for a given input, no appreciable improvements in speedup will occur even if the number of processors is increased. The plots in figure 8 show that nearly all the data points are within this saturation region for the computer systems. The Encore Multi-Max and the Intel hypercube (plots not shown) have smaller saturation regions than the Sun 3 Network. This is due to the smaller (zero) start up scaling factor $s(\alpha)$ on the Encore Multi-Max (Intel Hypercube). The smaller the saturation region the larger the set of problems which will see speedup improvements as the number of processors is increased.

Results from Multi-level Tautology

The multilevel algorithm has only been implemented on the Encore Multi-Max and on the Sun 3 workstations. The results obtained from the parallel multilevel tautology on the Encore are very similar to those seen from the 1-level tautology program. The speedup model constants were chosen to give the best fit to results from the adder circuits described above. Table 2 shows the values of the computer dependent speedup model parameters chosen. Figure 10 shows a plots of speedup versus the maximum level of recursion on the Encore. Each curve represents the speedup model's prediction for 2, 4 and 8 processors. The data points near each curve are the actual speedup obtained for the respective number of processors using the adder circuits described above. The speedup model matches the data very well. The value of $s(\alpha)$ for the multilevel algorithm is much larger than the value for the 1-level algorithm. This suggests that at least on the Encore machine that startup cost is not independent of the algorithm. The mechanism used to create a new process is the "fork". Since a fork must physically copy the program from one processor's memory to another, it is clear that the time of a fork is dependent on program size. The multilevel program was 2.6 times larger than the 1-level program therefore it is understandable that the startup time would be larger. Figure 11 shows a plot of the speedups from other examples run through the multilevel tautology algorithm using 8 processors versus maximum level of recursion . Most speedups are observed to be generally below the speedup model curve. All of the circuits which produced speedups less than what the model predicted were problems which did not produce full recursion trees. If, however, a plot of speedup versus effective level of recursion is made, there is a shift to the left of data points towards the speedup model's curve. Figure 12 is a plot of speedup versus effective level of recursion from data obtained using 8 processors and the values of the speedup model constants used in the previous plot. As in the 1-level case the speedup model curve remains the same since it assumes a full recursion tree, but the data points are now closer to the curve.

The speedups obtained from the multilevel problems appear to be higher than similar 1-level problems. A cover of all minterms problem which recurs the same level as a adder circuit does not see as great a speedup as the adder circuit does even though they both produce a full recursion tree. This better performance by the multilevel algorithm is due to the extra simulation work done at each node in the recursion tree. The speedup model for the multilevel algorithm predicts near linear speedups for reasonable recursion depths, as compared to the 1-level speedup model which predicts linear speedups at depths greater than 100, beyond a practical problem size.

Improving Results: Dynamic Processor Assignment

Clearly with this *static* processor assignment, problems which produce unbalanced recursion trees always produced low speedups. Another version of multilevel tautology has been developed which dynamicly schedules the tautology work among the processors. This algorithm begins, as the previous algorithm, by assigning a processor to each subproblem generated at each level of recursion up to level $\log_2 N$. However, when a processor is finished, any processor which has available work will attempt to give part of its problem to it. In this way, large problems which are highly unbalanced (i.e. c432) can balance the load by using processors which finish early.

The multilevel algorithm using dynamic processor scheduling has been implemented on the Encore Multi-Max and on the Sun 3 Network. The Sun 3 network implementation uses the GRAIL distributed software package.

The results obtained to date are encouraging. On a large problem, C432, a 36 input 7 output boolean network, very good results were obtained. This problem produces a highly unbalanced recursion tree. There are 2.2 million leaves in its recursion tree whereas a full tree would have a leaf count around 69 billion. Table 3 contains the results. When the dynamic processor scheduling program was run through smaller examples, the results were not as impressive. In many trivial problems, degradation in speedups occurred. Since these problems take only a short amount of time to process (less than a few minutes), we are not too concerned with these problems. However, it should be noted that even with a large problem, there is room for improvement. For example, the c432 result using 20 processors on the Encore is below 19. The cause of this less than linear speedup may be caused by a poor scheduling algorithm which results in too many small problems are being given to available processors instead of larger ones and the overhead is causing a reduction in speedup. Currently we are investigating the possibility of modeling processing time of a cover so that the scheduler can better deal out work. Early experiments suggest that this may be feasible. It is believed with an accurate model of processing time, both the small problems and the large problems will see significant improvements in speedup the dynamic processor scheduling algorithm.

Conclusions and Further Work

The parallel 1-level and multilevel tautology algorithms can be practically applied to both the distributed and multiprocessor computer environments. Covers which produce full or nearly full recursion trees produce the best speedups and these problems are the most time consuming for the serial algorithms.

Both algorithms are designed for a distributed computer environment. Each processor is given a substantial portion of work and there is not a tremendous amount of communications between processes. From the plots of the speedup model it is clear that the bipartitioning along the recursive call boundaries cannot achieve linear speedups for practical examples in the case of 1-level tautology. It would be interesting to investigate the possibilities of parallelizing the routines of the 1-level tautology algorithm.

The computer system can have a substantial effect on the speedups obtained. It was observed that communications, startup and initialization costs will affect the results. Startup costs degrade speedup for problems with covers which only recur a few levels. Because of startup costs, problems which recur only a few levels will see little or no increase in speedups if the number of processors is increased. Figure 3 shows a plot of the speedup models for several values of N which shows a region where processor saturation occurs (between maximum level of recursion 1 and 6). It is apparent that the startup cost is quite substantial on the Sun 3 network and non-trivial on the Encore Multi-Max. Data taken from the Intel hypercube did not include startup costs. Thus these results indicate the type of improvements that could be obtained by reducing the the startup costs. Research in the area of reducing startup costs, especially on the Sun 3 network should be pursued.

The speedup model for tautology has been developed under the assumption that startup and communications costs are not a function of the number of processors attempting to communicate simultaneously over the channel. However, as the number of processors increases, the number of message collisions increases. The results from the Sun 3 Network presented in this paper were obtained using at most 8 processors and the the effect of message collisions may have been very small. However, if a larger number of processors is used then this effect could become

significant. It is not clear how well the model will extend as number of processors is increased.

Finally, the model was developed in a general manner and should be easily applicable to recursive divide and conquer algorithms with the properties given previously (section titled Speedup model for Divide and Conquer Algorithms). The model can also be applied to a particular divide and conquer algorithm which has a subset of inputs which produce the properties stated above. The speedup model for tautology make several assumptions specific to the tautology algorithm and the computer systems (section titled Speedup Model for Tautology). Alterations to the communications and startup cost models will be necessary if these assumptions are not true for the particular recursive divide and conquer algorithm or the particular computer system.

Our results have shown that almost linear speedups are available for problems which are of sufficient size and lead to full or effectively full recursion trees. Examples are the covers of all minterms in the 1-level case (c.f. asymptote figure 1), and the adder circuits, other data path circuits (multipliers, alu's), and c432 in the multilevel case. We note that while c432 recurred to the full depth of its number of inputs, it still had an imbalanced tree, and so required dynamic scheduling to achieve nearly linear speedup. This scheduling algorithm did not work well, however, on problems with limited effective recursion level, and we are now working on improvements to the scheduler which correct this deficiency.

We have observed that dynamic scheduling results obtained from partially loaded computer system have produced acceptable run times. The scheduling algorithm appears to be robust enough to correctly shift work from the slower/loaded processors to the faster/unloaded processors. Although efficiency is degraded, the initial results indicate that the *available* computing resources of the loaded system are being used efficiently. Future work includes a further study of the effects of uneven processor loading, which represents a more practical setting, on the tautology run times.

We observe that in the context of multilevel logic minimization, [2], analogous tautology checking for c432 has been achieved in as little as 5 seconds, as opposed to 8 hours. Yet, further tautology requests in the same context required 5-8 hours. This nonuniformity of resource consumption is a formidable obstacle to extending the size capability of multilevel minimization. Nevertheless, it does seem that all cases that lead to "large" cpu requirements also lead to linear speedups in the corresponding parallel implementations. Thus parallel processing does appear to offer the hope of directly extending the size limitations of multilevel logic minimization algorithms, most of which are based on tautology checking or other divide and conquer type algorithms.

In fact, the parallel tautology has been implemented in the multilevel logic minimizer, MLMIN. Using this parallel MLMIN, we were able to finish the c432 problem in 10.1 hours on the Encore Multi-max using 16 processors versus 69.1 hours serially.

We have at this time no direct way to compare our results to those reported in [9], since that work was done on a different computer, and used a different (although still divide and conquer) algorithmic approach. Further, that work was designed for a testing and logic verification (rather than for a logic minimization) context. Thus it was applicable only to a restricted class of Boolean networks (in which the individual functions were required to be typical primitive logic gates, such as NAND's, NOR's, XOR's, etc). However, it does seem that the general conclusions stated above are supported by that work as well.

Acknowledgements

This work was supported in part by The National Science Foundation Grant #DMC-8419744, The Air Force Office of Scientific Research Grant AFOSR-85-0251, with additional funding provided by IBM the Corporation. We gratefully acknowledge the help of Reily Jacoby in providing theoretical consultation on tautology checking as well as detailed C code libraries for our use.

1. T. Adachi, *Proceedings 19th Design Automation Conference*, 1981, 785-791.
2. K. A. Bartlett, D. G. Bostick, G. D. Hachtel, R. M. Jacoby, M. R. Lightner, P. H. Moceyunas, C. R. Morrison and D. Ravenscroft, "BOLD: A Multiple-Level Logic Optimization System", *ICCAD87*, 1987.
3. J. Bentley, D. Haken and J. Saxe, "A General Method for Solving Divide-and-Conquer Recurrences", *SIGACT News News*, Fall 1980 , 36-44.
4. R. K. Brayton, G. D. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.
5. G. D. Hachtel and R. M. Jacoby, "Algorithms for Multi-Level Tautology and Equivalence", *Proceedings of the IEEE International Symposium on Circuits and Systems*, Kyoto, Japan, June 1985.
6. G. D. Hachtel and R. M. Jacoby, "Algorithm for Multilevel Tautology Checking", *IEEE Trans. on CAD (accepted)*, January 1986.
7. G. D. Hachtel and R. M. Jacoby, "Verification Algorithms for VLSI Synthesis", *Proceedings, NATO ASI on Logic Synthesis and Silicon Compilation for VLSI*, The Netherlands, 1987.
8. R. M. Jacoby, "Algorithms for Equivalence and Tautology Checking for Boolean Networks", Master's Thesis, University of Colorado, January 1986.
9. H. T. Ma, S. Devadas, A. Sangiovanni-Vincentelli and R. Wei, "Logic Verification Algorithms and their Parallel Implementation", *1987 Proceedings from the 24th ACM/IEEE Design Automation Conference*, Miami Beach, Florida, 1987, 283-290.
10. P. Maybee, "Grail: An Environment for Parallel Distributed Programming ", PhD Thesis, Department of Computer Science, University of Colorado, Boulder, Colorado, In Progress.
11. P. H. Moceyunas, "A Parallel Implementation of a Tautology Algorithm", M.S. Thesis, Department of Electrical Engineering and Computer Engineering, University of Colorado, Boulder, Colorado, May 1987.

Table 1

1-Level Tautology Speedup Model Constants			
Computer system	$c(\alpha)$	$s(\alpha)$	$\Delta(\alpha)$
Sun	1.0	80,000	0.0
Encore	.55	600	100.0
Intel	.1	0	600.0

Table 2

Multilevel Tautology Speedup Model Constants				
Computer system	$c(\alpha)$	$s(\alpha)$	$\Delta(\alpha)$	$\omega(\alpha)$
Encore	.55	8000	100.0	50

Table 3

Computer System	Number of Processors	Run time (hours)	Speedup
Sun 3	1	12.1	-
	8	1.6	7.6
Encore	1	33.2	-
	8	4.3	7.75
	20	1.9	17.1

Figure 1

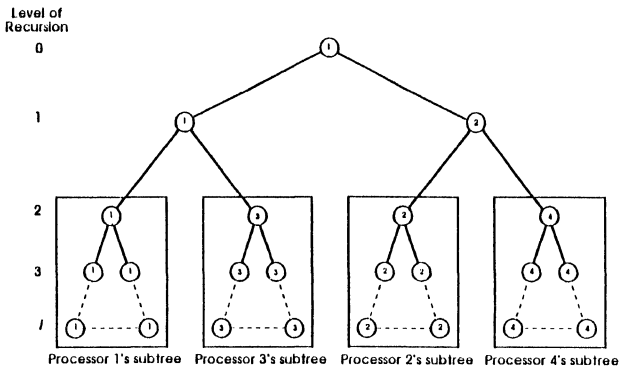


Figure 3

procedure ML_TAUTOLOGY (F , l, N, *parent*, *il*)
Inputs:
F - is a multilevel cover
i - is the current level of recursion
N - is the number of processors in the computer system
parent - is information about the parent process of this process
il - Level in the recursion tree that this processor made its first call to TAUTOLOGY
Output: Returns a 1 if the cover is a tautology otherwise returns 0

$r \leftarrow$ SPECIAL_CASES(F)
if ($r \neq -1$) **return** (r)
$j \leftarrow$ SELECT_SPLIT (F)
$F_{x_j} \leftarrow$ COFACTOR(F, x_j)
$F_{\bar{x}_j} \leftarrow$ COFACTOR($F, \bar{x}_j$)

/* Start a new process */
if ($i \leq (\log_2 N)-1$)
Start new process on another processor
Send $F_{\bar{x}_j}$ to the new processor

/* *newparent* is information of the new process's parent */
Call SIMULATE ($F_{\bar{x}_j}$) on the new processor
Call ML_TAUTOLOGY($F_{\bar{x}_j}, i+1, N$,*newparent*, $i+1$)
$F_{x_j} \leftarrow$ SIMULATE (F_{x_j})
if (ML_TAUTOLOGY($F_{x_j}, i+1, N$, *parent*, *il*) = 0)
a. **if** $il \neq i$ **return** (0)
b. **Send** a 0 back to the parent process.
Wait for the child process to send an answer back
a. **if** $il \neq i$ **return** the answer
b. **Send** the answer back to the parent process.

/* Perform Serial tautology */
else
$F_{x_j} \leftarrow$ SIMULATE (F_{x_j})
$F_{\bar{x}_j} \leftarrow$ SIMULATE ($F_{\bar{x}_j}$)
if (ML_TAUTOLOGY($F_{x_j}, i+1, N$, *parent*, *il*) = 0) **return** (0)
if (ML_TAUTOLOGY($F_{\bar{x}_j}$,$i+1$,N ,*parent* ,*il*) = 0)) **return** (0)

return(1)
end procedure

Figure 2
procedure ML_TAUTOLOGY (F)
Input: F - a multilevel cover
Output: Returns a 1 if F is a tautology otherwise returns 0

$r \leftarrow$ SPECIAL_CASES(F)
if ($r \neq -1$) **return** (r)
$j \leftarrow$ SELECT_SPLIT(F)
$F_{x_j} \leftarrow$ COFACTOR(F, x_j)
$F_{\bar{x}_j} \leftarrow$ COFACTOR($F, \bar{x}_j$)
$F_{x_j} \leftarrow$ SIMULATE(F_{x_j})
$F_{\bar{x}_j} \leftarrow$ SIMULATE($F_{\bar{x}_j}$)
if (TAUTOLOGY(F_{x_j}) = 0) **return** (0)
if (TAUTOLOGY($F_{\bar{x}_j}$) = 0)) **return** (0)
return (1)
end procedure

Figure 4

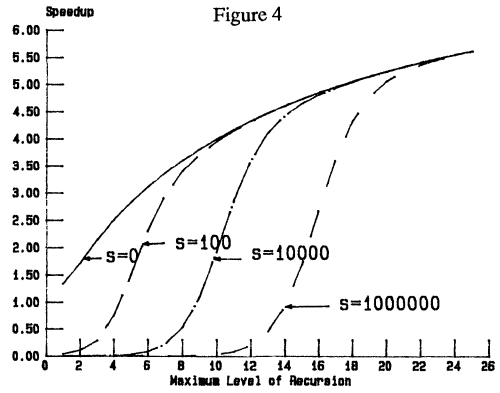


Figure 5

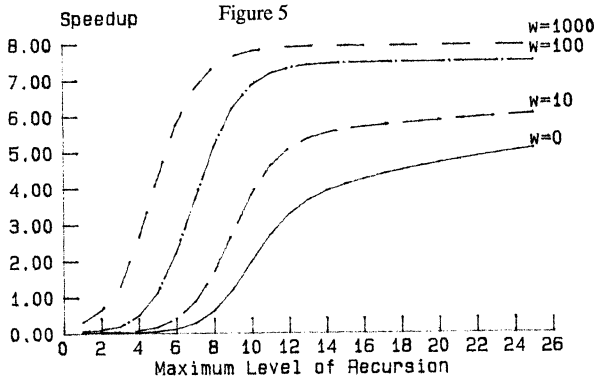


Figure 6

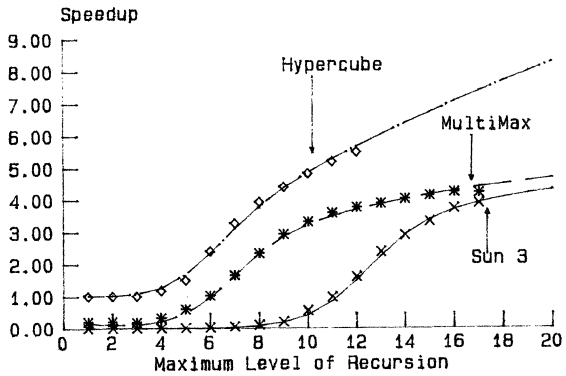


Figure 7

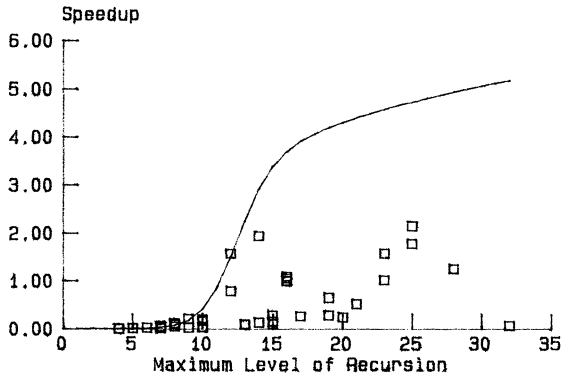


Figure 8

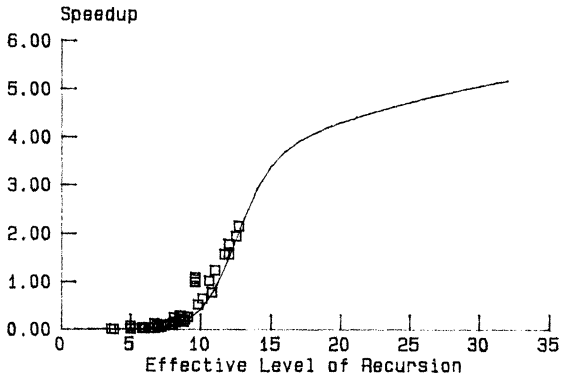


Figure 9

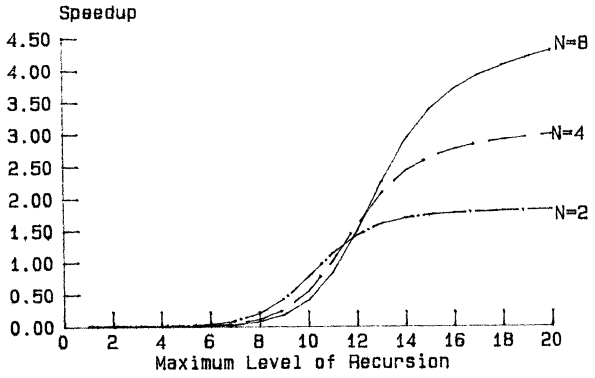


Figure 10

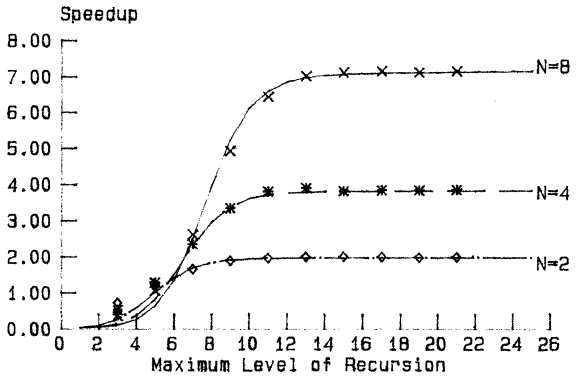


Figure 11

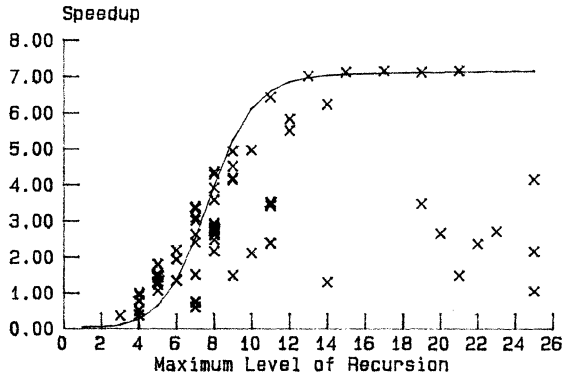
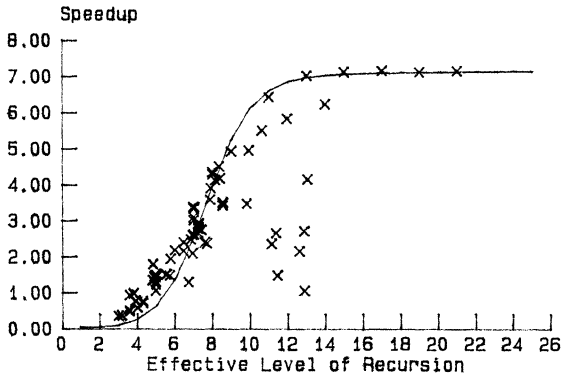


Figure 12



Parallelism in Knowledge-Based Systems with Inheritance

Michael Greenberg and Janice E. Cuny

Department of Computer and Information Science
University of Massachusetts, Amherst, MA 01003

Abstract: Applications that use domain expertise often require repeated queries of large databases; these queries typically involve the determination of attributes through inheritance. We present a parallel inheritance algorithm using inferential distance that does not require any form of network conditioning. It achieves speedup both in the parallel spreading of search activations within a single query and in the simultaneous processing of multiple queries. We also show that it is possible to significantly improve the performance of some specific networks by adding small amounts of additional processing capacity.

1 Introduction

Applications that use domain expertise — medical diagnosis, mechanical design, and computer assisted instruction, for example — often require repeated queries of large *knowledge bases*. In order to provide concise descriptions, these knowledge bases are usually organized as taxonomies of *objects* and *classes of objects* with their properties stored as high as possible within the hierarchy. This avoids a substantial amount of redundancy but it means that queries must be answered using *inheritance*. Inheritance is the process of inferring properties about objects from group membership: a prototypical chair has four legs, so we can infer that a specific chair has four legs; all mammals are warm blooded, so we can infer that dogs are warm blooded. The efficient implementation of inheritance is critical to performance.

Inheritance is complicated by *exceptions*; "Birds fly" but "penguins are birds" and "penguins don't fly." The most successful approach to dealing with exceptions [1] uses *inferential distance*, introduced by Touretzky [2]. With this approach, attributes are inherited from their "closest" ancestor according to a partial ordering:

> "The essence of this ordering is that an individual or class A is 'nearer' to class B than to class C iff A has an inference path *through* B to C." [2, p. 12]

For example, in Figure 1, Opus is nearer to Penguin than to Bird because there is a path from Opus to Bird that goes through Penguin.

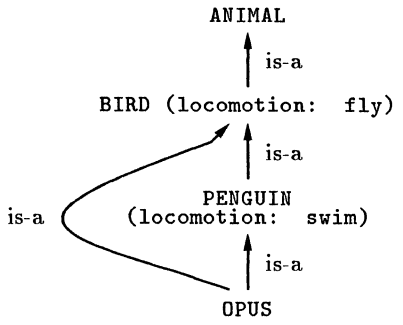


Figure 1: Sample Network. OPUS should inherit the value swim for locomotion because swim at node PENGUIN overrides fly at node BIRD.

It was first suggested that inheritance algorithms using inferential distance could not be done in parallel [3]. Later, Touretzky showed that it could be done on appropriately *conditioned* networks [2] but the conditioning requires n^2 time [1] and must be repeated after each network modification. We present a parallel inferential distance algorithm that does not require network conditioning. It has two sources of parallelism: the spreading activation of a single query and the simultaneous processing of multiple queries.

In Section 2, we describe the parallel algorithm for answering a single query and give an experimental analysis of the speedup obtained. In Section 3, we extend the algorithm to permit the simultaneous processing of multiple queries and in Section 4, we consider the effects of adding additional processing capacity to alleviate bottlenecks at frequently accessed nodes. In Section 5, we summarize our results.

2 Single Query, Parallel Inferential Distance Algorithm

A query to a knowledge base begins with the request for an attribute value and ends when all legal values satisfying that request have been sent to the originating node.

Our algorithm assumes an underlying MIMD message passing model of computation and does not make use of any global state information. Each process contains the description of a single concept and communicates with processes containing related concepts. We make no any assumptions about the transit time or the arrival order of communications.[1]

The algorithm has two variants: in the first, processes do not have local memory other than that needed to store attribute values; in the second, they do have local memory available for status information.

2.1 First Variant: No local memory

The algorithm is initiated when a request for an attribute value arrives at an *originating node*. If that node has a value locally, it is returned; otherwise, the knowledge base is searched. As the search progresses, activity spreads through the network using PLUS messages meaning "Do you have any values?" and MINUS messages meaning "Any values that you have are overridden." Both types of messages contain the name of the originating node; PLUS messages contain the requested attribute name as well. Answers are returned to the originating node using FOUND messages meaning "I have a legal value." and IGNORE messages meaning "Ignore any values that I've given you." FOUND messages contain a located attribute value and the name of the node where it was found; IGNORE messages contain the name of their source node.

Each node reacts to incoming messages as follows:

- If a node without a value for the attribute receives either a PLUS or MINUS message, then a copy of the message is sent to each parent.
- If a node with a value for the attribute receives a PLUS message, then a FOUND message is sent to the originator and a MINUS message is sent to each parent.
- If a node with a value for the attribute receives a MINUS message, then an IGNORE message is sent to the originator and a MINUS message is sent to each parent.

The final answer is computed by the originating node as the set of all "found" values minus the set of all "ignored" values. The originating node cannot complete its calculations until all message activity within the network has terminated. In order to detect this, we use a notion of *conservation of mass*:

> A *mass* is attached to each message. If a node sends out n messages in response to a received message with mass in, each outgoing message is assigned a mass of in/n. Initial requests enter the system with a mass of 1. The originator can terminate when it has received back a total mass of 1.

A fifth type of message MASS is used to transmit mass from a node without ancestors to the originator.

To see that the algorithm is correct, notice that the activation must reach every node on an ancestor path extending from the originator: nodes up to and including the first node with a value on a path receive PLUS messages and the remaining nodes on the path receive MINUS messages. If a node is on more than one path, it can receive both PLUS and MINUS messages; if it does, the MINUS indicates the presence of a value that is on an inference path from the originating node. Thus a node with a value will receive only PLUS messages in exactly the cases where it has a valid value as defined by the inferential distance criteria. Further, the algorithm must halt for networks without cycles (inheritance hierarchies, by definition, do not have cycles): mass is always conserved and all mass is eventually returned to the originating node after reaching either a node with a value or a node without a parent.

Example. If the node OPUS in the network shown in Figure 1 receives a message requesting its mode of locomotion, the following activity should result (assuming unit time for operations and message transmissions):

Time Unit 1: The initial message is sent to OPUS requesting his method of locomotion.

Time Unit 2: OPUS sends <PLUS, .5, locomotion, OPUS> to PENGUIN (a PLUS message with mass .5; the originator is OPUS) and <PLUS, .5, locomotion, OPUS> to BIRD.

Time Unit 3: PENGUIN sends <FOUND, .25, swim> to OPUS and <MINUS, .25, OPUS> to BIRD. BIRD sends <FOUND, .25, fly> to OPUS and <MINUS, .25, OPUS> to ANIMAL

Time Unit 4: BIRD sends <MINUS, .125, OPUS> to ANIMAL and <IGNORE, .125, BIRD> to OPUS. ANIMAL sends <MASS, .25> to OPUS.

Time Unit 5: ANIMAL sends <MASS, .125> to OPUS.

Time Unit 6: OPUS has received back a mass of 1 and calculates the answer is swim.

Spreading activation requires a time proportional to the depth of the search tree rather than its size; thus, there is more parallelism available in "bushy" networks. Defining *bushiness* to be the average of the number of

[1] Touretzky's model differs in several ways. His algorithm does not accumulate the answer at a specific location but leaves it distributed throughout the network. In the case where two incomparable values are found, our algorithm will report both while his algorithm reports only one. In addition, Touretzky assumes an SIMD model of computation in which the system can detect the cessation of message activity.

nodes in a search tree divided by its depth, the knowledge bases we have investigated have a bushiness value between 2 and 3. The speedup achieved for the overall processing of a query is further limited by the need to accumulate the final answer in the originating node. While some of the accumulation can overlap with the search itself, the originating node is potentially quite busy, receiving messages from every node on an ancestor path that either has a value or has no parents.

Before presenting our experimental results, we describe the second variant of our algorithm in which message traffic (including that directed to the originating node), is reduced with the use of local memory.

2.2 Second Variant: Local memory

For this version, we add a single memory cell to each node that keeps track of the history of that node's participation in a query. The cell can record one of three status values indicating that (1) neither PLUS nor MINUS messages have been received, (2) PLUS but not MINUS messages have been received, or (3) MINUS messages have been received. The use of this information can reduce message traffic. Nodes with values that receive a second message need only send their mass back to the originator (possibly in an IGNORE message). Nodes without values do the same unless they receive a PLUS followed by a MINUS in which case the MINUS must still be propagated forward. If used for the query described above, for example, this algorithm would decrease the number of time units needed from 6 to 5 and the number of messages from 10 to 8.

We have simulated the performance of both variants of our inferential distance algorithm on three knowledge bases. The first, DATATYPE1, is an 80 node data type lattice. DATATYPE1 has a single node that is the ancestor of every node in the network. Since this is not typical, our second network, DATATYPE2, is the 79 node network that resulted from removing DATATYPE1's top node; it has five top nodes. Our third network, CALL, is a 569 node call graph that has been randomly seeded with values.

The results of our simulations are shown in Figure 2. We assumed that processes could respond to incoming messages in unit time and that direct connections were available to all processes for outgoing messages; we did not charge for communication overhead. Thus, our results do not measure the parallelism that can be realized on a specific machine, but instead measure the parallelism inherent in the algorithm.

The table shows the average time needed to answer a query and the average number of messages sent. It can be seen that the algorithm's performance is improved by memory. Assuming the running time of the best sequential algorithm for inferential distance is approximately the same as the number of messages sent using the second

	Size	Bushiness	Average Time	Average Messages
DATATYPE1	80	2.0		
without memory			8.33	19.45
with memory			6.90	12.94
DATATYPE2	79	2.5		
without memory			7.53	14.96
with memory			6.35	10.81
CALL	569	2.6		
without memory			8.51	21.06
with memory			7.47	15.06

Figure 2: The effect of adding memory to the Parallel Inferential Distance Algorithm.

variant, we find that our parallel algorithm reduced the response time for single queries roughly by a factor of 2.

3 Multiple Query Algorithm

For many applications, it is necessary to provide simultaneous access to a knowledge base for more than one user. Sharing a single copy of the knowledge base avoids the problems of maintenance and coherence and provides higher utilization. In this section, we demonstrate that our algorithm permits the simultaneous processing of multiple queries without significantly affecting individual response times.

Multiprocessing of queries requires a few modifications. Messages are tagged with a *query id* allowing processes to distinguish messages relating to different queries and originating nodes are required to keep track of a number of sets of partial results. In addition, when the local memory variant is used, all nodes are required to store status information for the active queries. Since processes have fixed size memory (cells for status information) and are not notified that a query has completed, memory is managed with an Least Recently Used policy. If this results in the destruction of information before the associated query completes, the algorithm performs correctly but may execute redundant computations.

We have experimentally evaluated the processing of multiple queries on the three knowledge bases. We used a number of different memory sizes. Each experiment was performed with a fixed memory size, varying the arrival rate of queries. Queries were randomly selected from the set of possible queries and their arrival was controlled by a Poisson process. The effect of increased arrival rates on response time was observed.

Figures 3, 4, and 5 show the results of the experiment for the three knowledge bases. In each case, the y-axis is the average time to answer a query and the x-axis is the arrival rate normalized with respect to the rate at which

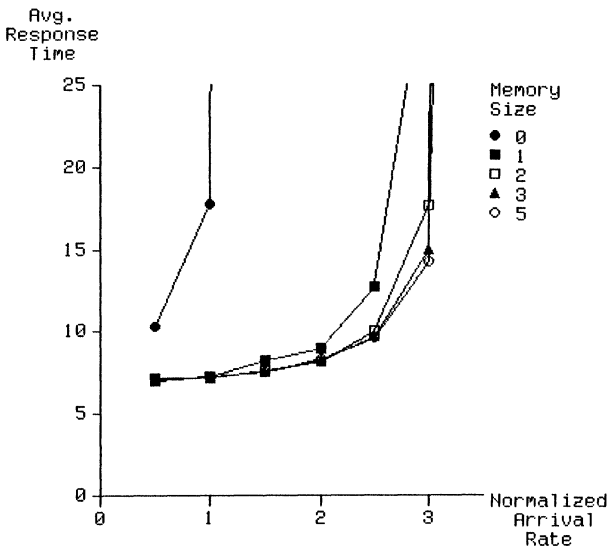


Figure 3: Multiprocessing of queries on the DATATYPE1 knowledge base.

isolated queries are processed. Thus, for example, with a memory size of 2, 3 or 5, multiprocessing allows the arrival rate to increase substantially before response time is affected.

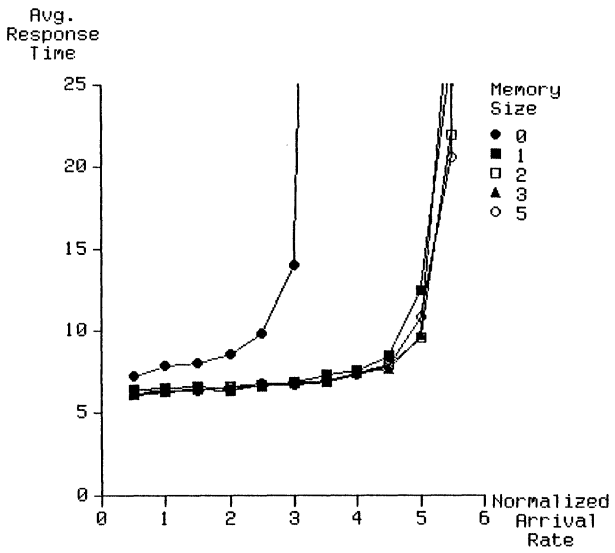


Figure 4: Multiprocessing of queries on the DATATYPE2 knowledge base. Axes labeled as in Figure 3.

As the arrival rate increases, the average time to answer a query increases. Eventually, the network reaches a saturation point at which incoming messages arrive faster than they can be processed and the average time to answer a query skyrockets. If we consider the (normalized) arrival rate just before saturation is reached as the *maximum arrival rate*, we see that the three data bases achieve maximums of 2.5, 4.5, and 13.5. The amount of improvement available depends on the structure of the network. The effect of increasing memory is twofold: the average time to answer a query is lowered and the maximum arrival rate is increased. Only small amounts of local memory are needed; a memory size of 2 or 3 appears sufficient.

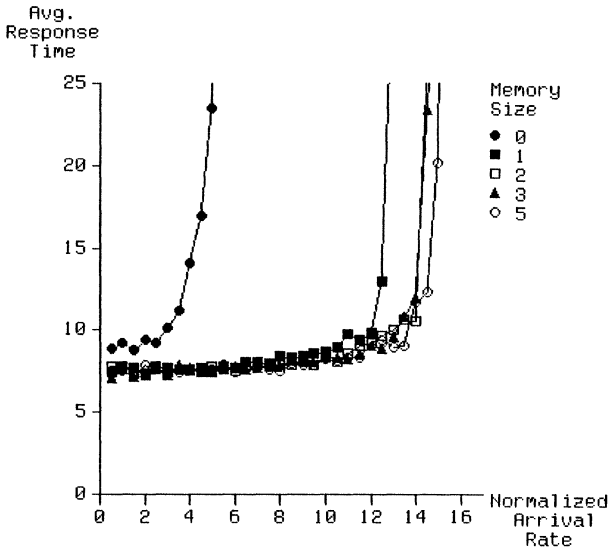


Figure 5: Multiprocessing of queries on the CALL knowledge base. Axes labeled as in Figure 3.

Examination of the simulation data showed the presence of a few bottlenecks significantly degraded performance. In the DATATYPE1 knowledge base, a single node was an ancestor to all nodes and acted as a serial bottleneck. In the next section, we explore the possibility of improving throughput by adding processing capacity at heavily utilized nodes.

4 Utilizing Additional Processing Capacity

If additional processing nodes are available for use, it is possible to alleviate congestion at heavily utilized nodes. As shown in Figure 6, heavily utilized nodes can be copied, partitioning their inputs. In the figure, C is copied so that queries originating at D and E no longer compete for its processing capacity with queries originating at F.

To investigate the effect of additional processing capacity, we used a simple processor allocation scheme. The system was run at its maximum arrival rate (just before saturation) to determine which node had the highest average input queue length. This node was duplicated and its original inputs were partitioned so that the sums of the average utilizations of the nodes for each partition were as close to equal as possible. Figure 7 shows the results for

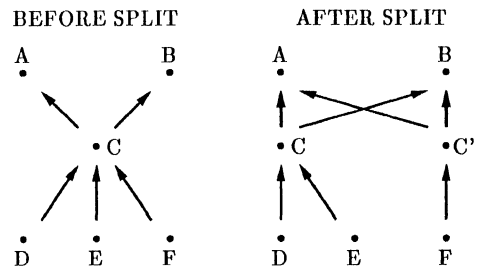


Figure 6: Example of node duplication used to reduce the processing requirements at heavily utilized nodes.

the DATATYPE1 knowledge base.[2]

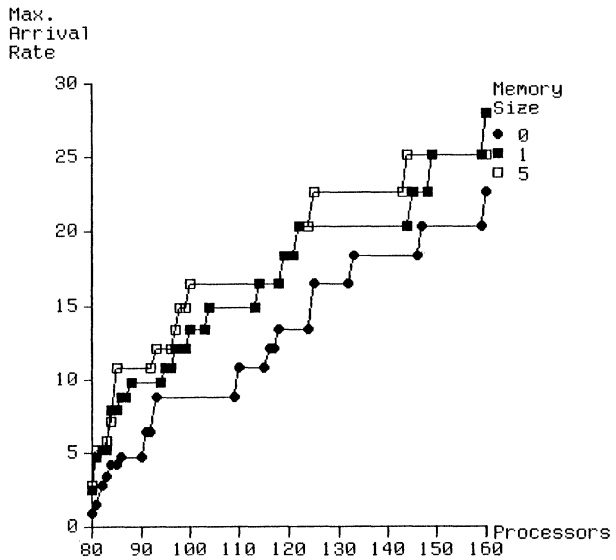


Figure 7: Speedup achieved by adding processing capacity to the DATATYPE1 network. The y-axis gives the maximum normalized arrival rate and the x-axis gives the number of processors.

The figure shows the increase in maximum arrival rates as a function of the number of nodes in the network for varying amounts of memory. The initial additions achieve substantial speedups. Adding a single node, for example, increases the throughput almost two-fold; adding five nodes increases the throughput almost four-fold. This is because initially there were a few very heavily utilized nodes in the knowledge base. As more and more processors are added, the utilization of the nodes in the system becomes more uniform and the speedup becomes linear. Note, however, that this speedup reflects the potential parallelism available in the algorithm; it does not take into account the communication overhead that would be present in a specific implementation. As the machine size increases, the effect of this overhead increases and, at some point, the performance of an actual system would degrade. Thus, we can divide the performance graph into three regions. Performance gains should be accomplished, in the first region, by expanding the existing network and, in the last region, by copying the entire network; performance gains in the middle region can be accomplished in either manner.

[2]We present data for only one network. The results for DATATYPE2 are identical after the first few splits; the results for CALL show similar improvements.

5 Conclusions

We have presented a MIMD algorithm for inferential distance inheritance that does not require network conditioning. It achieves speedup both in the parallel spreading of search activations and in the multiprocessing of queries. In each case, the parallelism available depends on the structure of the knowledge base. We have shown that it is often possible to significantly improve the performance of a specific network by adding small amounts of additional processing capacity.

This work analyzes the parallelism available in our inferential distance algorithm; it does not analyze the performance of the algorithm on a real machine. Any actual implementation will be limited by the overhead of communication. We are currently developing strategies for allocating processes on specific architectures to determine the extent to which the available parallelism can be exploited in practice.

References

[1] David W. Etherington, "More on Inheritance Hierarchies with Exceptions," *Proc. AAAI-87*, pp. 352–357 (1987).

[2] David S. Touretzky, *The Mathematics of Inheritance Systems*. Morgan Kaufman Publishers, Inc., Los Angeles California (1986).

[3] David W. Etherington and Raymond Reiter, "On Inheritance Hierarchies with Exceptions," *Proc. AAAI-83*, pp. 104–108 (1983).

ASSOCIATIVE MEMORIES ON THE CONNECTION MACHINE

Stephen D. Simmes and Charles J. Turner
Science Applications International Corporation
5151 E. Broadway, Suite 900
Tucson, Arizona 85711-3796

Abstract

We study the Hopfield associative memory on a Connection Machine. We derive a synchronous algorithm that allows up to 1600 memory patterns, each 16K in size, on a 16K processor machine. The memory recall time for such a setup is estimated to be approximately 8 minutes on a CM-1. An expression for the asynchronous energy change during an interation is derived which shows that the diagonal terms in the synaptic coupling matrix have an effect on the memory recall process. These terms are generally set to zero in studies of associative memory. We show that this may not be the best strategy.

1.0 Introduction

There has been much recent interest in computer algorithms which model or emulate biological systems. Much of this work has appeared under the appelation neural networks which, generally, refer to large networks of communicating simple processors. Such systems perform computations exhibiting qualities of living organisms such as learning, recognition, or interaction with a complex environment. One such algorithm which has attracted a wide base of interest is the associative memory as formulated by Hopfield (1982).

The Hopfield associative memory consists of a fully interconnected network of N simple processors which are capable of computing a linear combination of scalar input values along with a nonlinear thresholding operation. Each processor can be in one of two states, $S_i \in \{-1,1\}$, and the state of the entire network is represented by an N-dimensional vector $S=(S_1, S_2 \ldots, S_N)$. The memory works according to the following dynamic updating algorithm: Each processor receives the current state values from all other processors which are multiplied by synaptic weights and summed to give the net input. The processor state value is then updated according to the sign of the net input minus a threshold. Two modes of processing are of interest, synchronous and asynchronous. In the asynchronous mode, the processors are updated without global timing whereas in the synchronous mode all processors are updated in unison. The two modes are not equivalent and may lead to different limiting behavior from identical initial states.

If the synaptic weights are chosen appropriately the system is capable of exhibiting a memory property. In this case the weights store fundamental memory patterns $S^{(k)}$, k=1, . . ., M, and when M is not too large, each memory $S^{(k)}$ is a stable attractor for the dynamic updating algorithm. That is, if the system is initialized with a pattern in a sufficiently small neighborhood of one of the fundamental memories, the updating algorithm will converge to this memory as time increases. Thus, the system associates the stored memory with the noisy input version when the noise level (measured by the number of incorrect bits in the pattern) is not too large. This is a useful error correcting property in systems where the next stage of processing is sensitive to noise in the patterns.

2.0 Associative Memory Algorithm

The synaptic weight for the path connecting processor i to processor j is denoted by J_{ij}, It is assumed that $J_{ij}=J_{ji}$ so that the matrix of weights J is symmetric. The dynamic updating rule for zero threshold may then be expressed

$$S_i(t+1) = \text{sign}\left(\sum_{j=1}^{N} J_{ij} S_j(t) \right) \qquad (1)$$

where sign(x) = + 1 if $x \geq 0$ and -1 if $x < 0$. The state of the system at t = 0, S(0), is set to the noisy pattern. Equation (1) is then iterated until a stable behavior is realized.

When the i-th processor is changed independently of the other processors, the global energy

$$E(S) = -(1/2) \sum_{i,j=1}^{N} J_{ij} S_i S_j$$

is non-increasing whenever $J_{ii} \geq 0$. This is seen by the following computation. If $\Delta E^{(i)}$ denotes the change in energy when the i-th state changes by ΔS_i then

$$\Delta E^{(i)} = -\Delta S_i \left(\sum_{j=1}^{N} J_{ij} S_j(t) - J_{ii} S_i(t) \right)$$

Let $V_i(t) = \sum_{j=1}^{N} J_{ij} S_j(t)$ be the input to the i-th processor,

then $V_i(t) = S_i(t+1) \, |V_i(t)|$ and the change in energy can be written

$$\Delta E^{(i)} = -\left(J_{ii} + |V_i(t)| \right)\left(1 - S_i(t) S_i(t+1)\right)$$

$$\leq 0 \text{ if } J_{ii} \geq 0 . \qquad (2)$$

This implies that the asynchronous updating algorithm always converges to a steady state. In the synchronous mode, it is no longer true that the energy must decrease at every time step and the system converges to either a steady state or a periodic orbit.

Consider the case where the fundamental memories are orthogonal vectors, that is

$$\sum_{i=1}^{N} S_i^{(k)} S_i^{(\iota)} = N\delta_{k\iota}.$$

In this case the weights may be defined by the correlation matrix

$$J_{ij} = \sum_{k=1}^{M} S_i^{(k)} S_j^{(k)} \qquad (3)$$

This is the formulation used by Hopfield (1982) except we have put no restriction on J_{ii} (Hopfield assumed $J_{ii} = 0$). Since $J_{ii} = M$ in (3) independent of the set of memorized patterns, it is clear that the diagonal terms play no role in the storage of information. However, from equation (2) we conclude that the diagonal terms do play some role relating to the stability of the patterns. This observation is reinforced by the fact that as $J_{ii} > 0$ increases without bound (holding the off diagonal terms fixed), equation (1) predicts $S_i(t+1) = \text{sign}(J_{ii}S_i(t)) = S_i(t)$ and thus all patterns become stable. On the other hand, as $J_{ii}<0$ decreases without bound, it follows that $S_i(t+1)=\text{sign}(J_{ii}S_i(t))=-S_i(t)$ and every pattern destabilizes into a flip-flop.

Several recent papers have studied the question of stability of fundamental memories (McEliece et al (1987), Newman (1987) and Komlos and Paturi (1987)). These authors study the correlation matrix memory using probabilistic methods when the fundamental memories are random vectors. The following results have been proved by Komlos and Paturi (1987); α_a, α_s, ρ_a, ρ_s are positive constants and distance between patterns is measured by the Hamming distance (number of bits which differ).

> Asynchronous updating: If $M \le \alpha_a N$ and the initial pattern is within distance $\rho_a N$ from a fundamental memory, then the system will converge to a steady state within a distance $N\exp(-N/4M)$ from the fundamental memory. When $M<N/4\ln N$ the system will converge to the fundamental memory.
>
> Synchronous updating: If $M \le \alpha_s N$ and the initial pattern is within distance $\rho_s N$ from a fundamental memory, then in about $\ln(N/M)$ steps the system will enter a region of radius $N\exp(-N/4M)$ about the fundamental memory and remain there. When $M<N/4\ln N$ the system will converge to the fundamental memory in $0(\ln\ln N)$ time steps.

By means of numerical experiments, Hopfield (1982) showed for asynchronous updating that $\alpha_a \le 0.15$. Later, Amit, Gutfreund and Sompolinsky (1985) extended the analysis to a nonzero temperature (T) Monte Carlo process (the Hopfield dynamics correspond to T=0). They gave convincing arguments that there was a critical value $\alpha_a \cong 0.138$ above which no reliable memory was possible. A restriction on M is natural since when M gets too large the fundamental memories crowd one another and the basins of attraction begin to coalesce. In fact Komlos and Paturi (1987) have shown that there are $0(2^M)$ stable extraneous memories stored by the correlation matrix memory, equation (3).

3.0 Implementation on the Connection Machine

The Connection Machine (CM) is a system of 64K processors interconnected by a 16 dimensional hypercube network. The network is implemented as a 12 dimensional wired hypercube with 16 processors at every node. Each processor is a simple bit-serial machine with 4K bits of local memory (CM-1). The CM is controlled by instructions sent from a front-end computer. Each instruction is broadcast to all processors in parallel making the CM a synchronous machine. However, an instruction is executed only if a processor's context flag is set. Thus the context flag may be used to control the pattern of computation. Further information on the CM may be found in Hillis (1985).

It is natural to implement the updating rule of equation (1) by assigning each state component S_i to a processor in a one-to-one fashion. All processors then simultaneously execute a time step by computing the sign of the local input

$$V_i(t) = \sum_{j=1}^{N} J_{ij} S_j(t).$$

However, this method requires that the i-th processor have access to the N weights J_{ij}, j=1, . . ., N. Since these weights differ from processor to processor, there is no scheme by which they can be broadcast to all processors in N parallel steps. On the other hand, if these weights are stored in the 4K bits of local memory as 16-bit integers, then N would be restricted to a maximum of 256 processors, which is unacceptable. (The situation is actually worse than this since the CM uses local memory for dynamic stack allocation). A similar limitation arises if one attempts to assign weights to processors.

This problem does not arise when the correlation memory is incorporated directly into the computation. Using equation (3) it is possible to rewrite equation (1) as

$$S_i(t+1) = \text{sign}\left(\sum_{k=1}^{M} W^{(k)}(t)\, S_i^{(k)}\right) \qquad (4)$$

$$W^{(k)}(t) \equiv \mathbf{S}(t) \cdot \mathbf{S}^{(k)}$$

$$= \sum_{j=1}^{N} S_j(t)\, S_j^{(k)} \qquad (5)$$

Here the updating rule appears as a linear combination of fundamental memories weighted by the inner product of the current state with the fundamental memory. It is clear that this process will converge to a fundamental memory $\mathbf{S}^{(k0)}$

whenever the fundamental memories are (more or less) orthogonal and the initial pattern $S(0)$ is close enough to $S^{(ko)}$. In this case we find that $W^{(ko)} \cong N$ and $W^{(k)} \cong 0, k \neq k_o$ so that $S(1)=\text{sign}(NS^{(ko)})=S^{(ko)}$ and the algorithm converges in one time step.

Each processor has the responsibility to keep track of one state component. Instead of storing weights, however, we now store the fundamental memories distributed across the processors so that processor i stores all memory components $S_i^{(k)}$, k=1, . . .,M. The same inner products $W^{(k)}(t)$, k=1, . . .,M are used in (4) for every memory component. The elements of the k-th inner product are computed by a simple one bit parallel multiplication $S_i(t)S_i^{(k)}$. These results are then summed over all processors and rebroadcast using CM scan operations (Blelloch (1986)). Each processor then multiplies the k-th inner product times the k-th fundamental memory component and updates the sum appearing in equation (4). This process is repeated until k=M when the sign operation on the sum is executed. All memories are represented internally as binary numbers (S_i=0 or 1). This formulation uses about 2K bits of local memory for stack space, which leaves an upper bound of about M=2K memories in a network of N=64K processors. This corresponds to a capacity of $\alpha=M/N \leq 0.0312$ which is acceptable.

4.0 Numerical Experiments

Numerical experiments of the associative memory were run on a 16K processor Connection Machine (CM-1) located at Science Applications International Corporation, Tucson, Arizona. The fundamental memories consisted of 128x128 pixel images. A small number of the images M_1=27, were selected from the USC Image Processing Institute data base (Schmidt (1977)). The original USC data, which is eight bit, was mapped into binary format by either a dithering process or a thresholding operation, depending on the input scene. The dithering algorithm, which creates a halftone image similar to reproductions of pictures in daily newspapers, was found to work well on uncluttered high resolution images containing specific objects like faces, airplanes, etc. On the other hand, thresholding was found to work better on complex scenes like aerial photos, although the resulting quality was only marginal on such scenes. No attempt was made to optimize the visual quality of the fundamental memories.

The full memory capacity for 16K processors is roughly 1600 images. Since our image data base is nowhere near this large, a set of randomly generated 128x128 patterns was used to simulate the remaining $M-M_1$ fundamental memories. These patterns were generated by assigning pixels values -1 or +1 with equal probability. The patterns were not actually stored on disk but were created in the CM during the memory initialization phase. The fundamental memories were all stored and referenced in local memory as one-bit binary data via the transformation $\{-1,+1\} \Leftrightarrow \{0,1\}$.

A single recall experiment consisted of the following steps. One of the fundamental memories, $k=k_o$ was chosen as the test image. A noisy copy of the test image was made by flipping pixel values with probability $P_n<0.5$. The iteration was then initialized by loading the noisy copy into $S(0)$. The iterations were monitored by computing the distance between the current state $S(t)$ and the test memory $S^{(ko)}$ as well as the distance between successive states $S(t)$ and $S(t-1)$. The distances were computed using the metric in equation (6) which gives the fraction of pixels which differ between two images.

$$d(S_1, S_2) = \left(1-1/N \sum_{j=1}^{N} S_{1j} S_{2j} \right) / 2 \qquad (6)$$

Convergence was decided when the distance between two successive iterations fell below a threshold.

Figures 1 and 2 show the results of recall experiments for M=800 fundamental memory patterns, corresponding to α=0.0488. The noisy inputs were generated with noise levels at P_n=0.3 so that approximately 30% of the pixels were flipped from their correct value. The iterations were carried out until successive iterations differed by at most two pixels. Convergence typically occurred in about six iterations. It was found that the updating algorithm was able to process about 20 memory patterns per second. This gives a cycle time of about 40 seconds for an iteration and 4 minutes for location of the correct memory pattern in the M=800 memory experiments.

We tested the hypothesis that the diagonal weights J_{ii}, have a specific effect on the convergence of solutions of equation (1) (or equation (4)). In order to directly control the diagonal terms, equation (4) was replaced by equation (7).

$$S_i(t+1) = \text{sign}\left(\sum_{k=1}^{M} w^{(k)}(t)\, S_i^{(k)} - (M-\delta) S_i(t) \right) \qquad (7)$$

where the diagonal terms now are forced to take on the constant value $J_{ii}=\delta$. Setting $\delta=M$ regains the weights defined by equation (3) whereas $\delta=0$ gives the Hopfield (1982) formulation $J_{ii}=0$. The inner products $W^{(k)}$ are still given by equation (5). It is interesting to note that for $\delta<M$, the effect of the new term is a tendency to switch the sign of each $S_i(t)$. This tendency depends on the magnitude of the $W^{(k)}(t)$ terms and is weak when M<<N. Several experiments were run with a fixed noisy input pattern and several values of δ ranging from -M to +M. The results of these runs are summarized in Figures 3 and 4. As can be seen from these plots, there is a clear indication, for sufficiently large M, that as the diagonal terms increase from -M to +M, the convergence of the algorithm improves significantly.

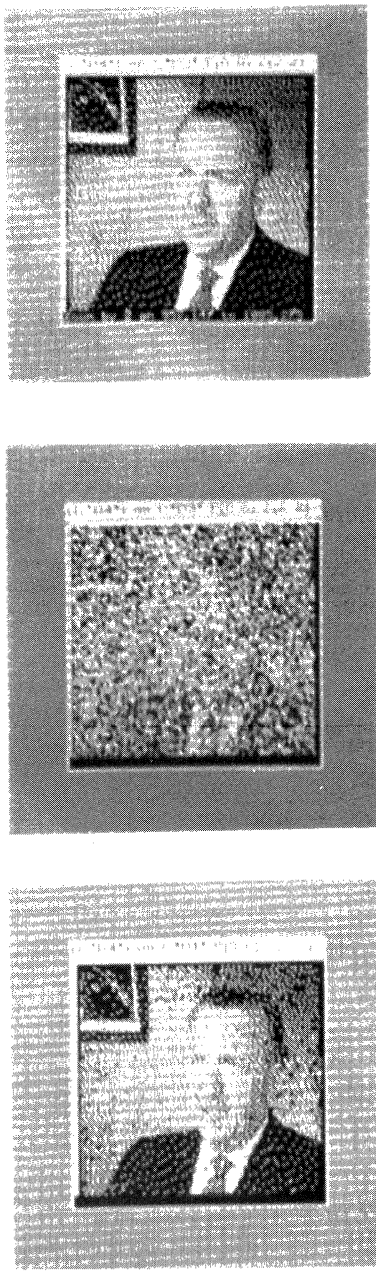


Figure 1. Results of experiment using a dithered image with M=800 memories. (top) Original, (middle) original with 30% noise, and (bottom) reconstructed image.

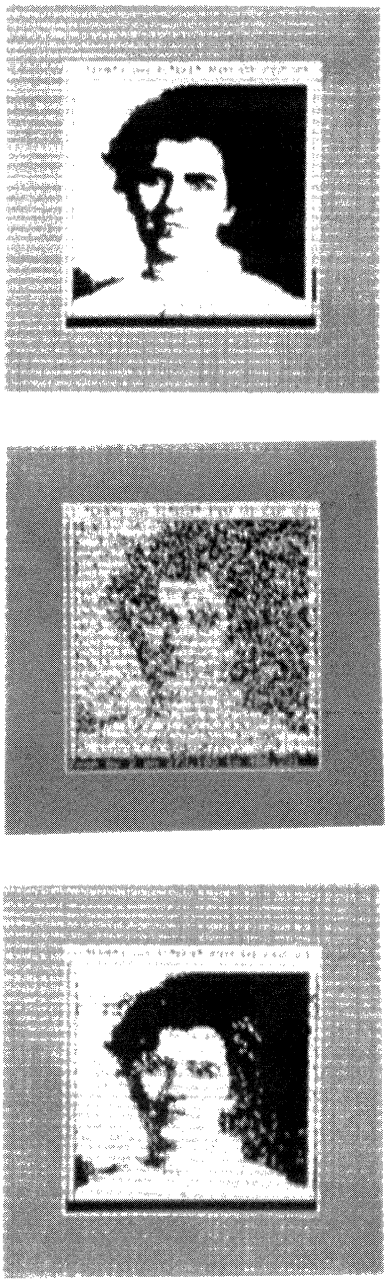


Figure 2. Results of experiment using a binary image with M = 800 memories. (top) Original, (middle) original with 30% noise, and (bottom) reconstructed image.

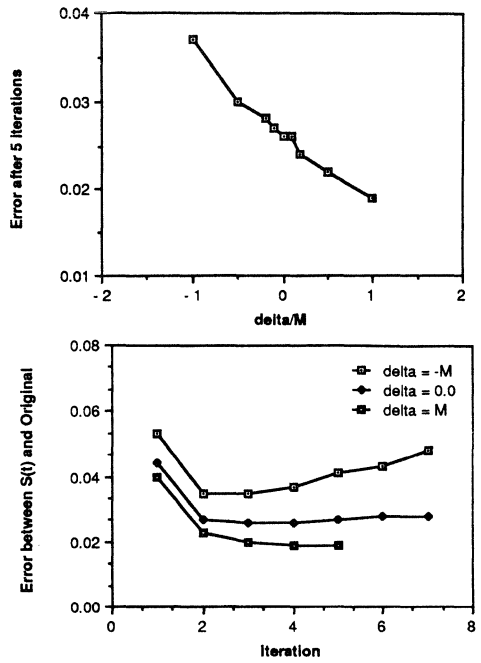


Figure 3. Convergence results for M=800 memories. (a) Error after 5 iterations for various values of delta ranging from -M to M. (b) Error at each iteration for delta = -M, 0, and M. Errors are the difference between the iteration value and the original, using equation 6.

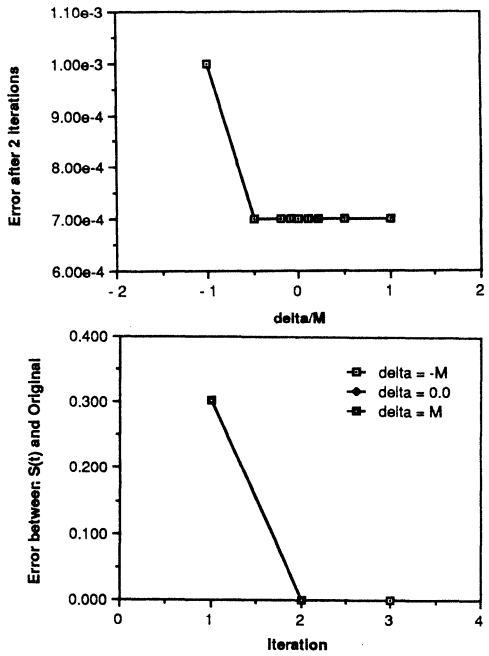


Figure 4. Convergence results for M=100 memories. (a) Error after 2 iterations for various values of delta ranging from -M to M. (b) Error at each iteration for delta = -M,0, and M. Errors are the difference between the iteration value and the original, using equation 6.

References

Amit, D. J., Gutfreund, H., and Sompolinsky, H. (1985) "Storing infinite numbers of patterns in a spin-glass model of neural networks," Phys. Rev. Lett. 55:1530-1533.

Blelloch, G. (1986) "Scans as primitive parallel operations," Thinking Machines Corporation technical report.

Hillis, D. (1985) "The Connection Machine," MIT press, Cambridge, Mass.

Hopfield, J. J. (1982) "Neural networks and physical systems with emergent collective computational abilities," Proc. Natl. Acad. Sci., USA 79:2554-2558.

Komlos, J., Paturi, R. (1987) "Convergence results in the Hopfield model," Caltech preprint.

McEliece, R. J., Posner, E. C., Rodemich, E. R., Venkatesh, S. S., (1987) IEEE Trans. Info. Theo. IT-33:461-482.

Newman, C. M., (1987) "Memory capacity in neural network models: rigorous lower bounds," Univ. of Arizona preprint.

Schmidt, R. (1977) "The USC-Image Processing Institute data base, revision 1," USC technical report.

Parallel Generation of LR Parsers

Manuel E. Bermudez
George Logothetis
Richard Newman-Wolfe

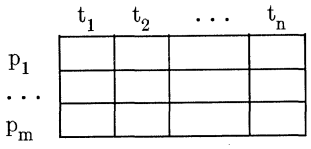
University of Florida, Gainesville, FL 32611 U.S.A.

Abstract

We present a novel technique for carrying out the automatic generation of an LR parser in a totally parallel fashion. The generation of an LR parser consists of constructing a parse table, with one row per state (in a push-down automaton), and one column per terminal symbol. Traditionally, this is carried out row by row, in which case the computation of one row depends on (potentially) all others. In contrast, our technique performs the computation by column. We show that the computation is totally independent for each column, making it ideal for parallelization. The speedup factor of the technique is min(N,M), where N is the number of processors and M is the number of terminal symbols.

1. Introduction

LR parsing [6,15] is the technique of choice for generating parsers for context-free languages. An LR parser is generated automatically from a context-free grammar, with a technique that consists of three phases. In the first, the LR(0) automaton is constructed. This automaton usually has a number of conflicts, which are of two types: shift/reduce conflicts and reduce/reduce conflicts. The second phase, lookahead computation, is invoked to resolve these conflicts. This consists of computing suitable "FOLLOW" sets for each nonterminal in the grammar. The most popular techniques for computing lookahead are SLR(1) (pronounced "Simple LR(1)") and LALR(1) (pronounced "Lookahead LR(1)"). In the third phase a transition table is built, based on the FOLLOW sets computed in the second phase. This transition table is traditionally known as the ACTION table, and it encodes the appropriate move(s) (shift, reduce, accept, or error) for each state and each terminal symbol, in the basic form shown below.



Serial SLR(1) and LALR(1) algorithms [2,7,8,16,17,21,22] fill in the "reduce" entries of the ACTION table **by row**, i.e. a lookahead set is computed for each reduce move at each state. The lookahead set is the result of unioning certain FOLLOW sets which, in turn, are obtained by unioning other FOLLOW sets with certain First sets. Thus the computation for one state (i.e. one row) depends, in general, on all the others. In contrast, our algorithms fill in the ACTION table **by column**. For each terminal symbol t (i.e. for each column), our algorithms use relations that describe how (1) t becomes a member of a First set, (2) how a member of a First set becomes a member of a FOLLOW set, (3) how a member of a FOLLOW set becomes a member of another FOLLOW set, and (4) how a member of a FOLLOW set triggers a reduce move at some state. The computation of each column is completely independent from all other columns; thus one parallel processor can be assigned to the task of filling in each column.

Previous work [3,4,5,9,10,11,18,20,23] on parallel parsing have focused on parallelizing the activity of **parsing**, rather than parallelizing the activity of **parser generation**. Thus there exist languages and compilers that allow parallel computations, and allow the development of application programs that exploit the parallel architecture, but there are no environments suitable for the use of parallel processing in the development of a compiler, or one of its components, e.g. the parser. Giving the source code of an existing parser-generator to a parallelizing compiler will not dramatically improve the parser generator's performance. At best, some of the parser-generator's operations will be vectorized. Specifically, a popular parser-generator such as YACC [14] can be compiled and run on a parallel computer, but it will use no more than one of the parallel processors, and thus will not exploit the machine's parallel architecture. This is due to the algorithms used in parser-generators, which are inherently serial, as discussed above.

In this paper we present a technique whereby the generation of an SLR(1) parser can be carried out in a totally parallel fashion. The technique allows for the totally independent decomposition of the problem into as many subtasks as there are symbols in the terminal vocabulary in the language. Thus, if one has N parallel processors, and M terminal symbols, the speedup factor in the time required to generate the parser is min(N,M). The parallelization technique also applies to LALR(1) parsers [19], but we will omit that discussion for the sake of clarity.

The remainder of this paper is organized as follows. In section 2 we present an overview of SLR(1) parsing. In section 3 we present a discussion of FOLLOW computation. In section 4 we present the parallel algorithm for generating SLR(1) parsers, analyze the storage requirements for each processor, and justify the speedup factor. Finally, in section 5 we present conclusions.

2. Overview of SLR(1) Parsing

We assume that the reader is familiar with context-free grammars and languages, and with shift-reduce parsing. We also assume some familiarity with LR parsers, in particular the construction of the LR(0) automaton. There are many good texts that cover this material, such as [1,12,13]. We also adhere to the following notational conventions.

A, B, C, ...	nonterminal symbols
t, a, b, c, ...	terminal symbols
..., x, y, z	terminal strings
..., X, Y, Z	grammar symbols
$\alpha, \beta, \gamma, ..., \omega$	strings of grammar symbols
ϵ	the empty string
$A \rightarrow \omega$	a production in a CFG
$\Rightarrow$	right-most derivation
First(α)	$\{t \mid \alpha \Rightarrow^* tx$, for some x$\}$.
p, q, r, s	states in the LR(0) automaton

A context-free grammar that generates arithmetic expressions in a typical programming language is shown in figure 1, along with its LR(0) automaton. In both the figure and the automaton, "⊥" is the end-of-file marker.

The LR(0) automaton has ten states, two of which have shift/reduce conflicts. In state 5, the parser cannot decide between reducing using production "E→E+T" and shifting on "*"; in state 6 the conflict is between shifting on

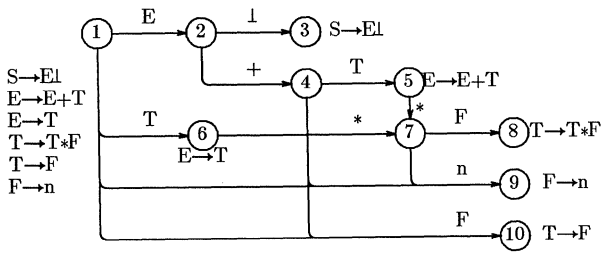


Figure 1. The LR(0) parser of a context-free grammar.

"*" and reducing using "E→T". The construction of this automaton constitutes the first phase of the generation of the parser. The second phase, namely lookahead computation, is invoked to resolve the conflicts at states 5 and 6. In the SLR(1) technique, this consists of computing FOLLOW(A), where A is the left-part of the production involved in the conflict. FOLLOW(A) is defined as $\{t \mid S \Rightarrow^* \alpha A t z\}$. This is the set of terminal symbols that can appear after A in a sentential form, i.e. a string that can be derived from the start symbol S. In our example, we should compute FOLLOW(E) for both conflicts. We will discuss in detail the computation of FOLLOW sets, which are obtained from the grammar, in the next section. For now, the reader should be easily convinced that FOLLOW(E) = {+,⊥}, since these are the only symbols that may legally appear after E in a sentential form. The FOLLOW sets are shown in the boolean table in figure 2, in which "•" in entry [A,t] indicates that t∈FOLLOW(A).

We shall see shortly that explicitly representing the FOLLOW sets in this manner is unnecessary. Instead, one parallel processor will compute each column.

	+	*	n	⊥	
E	•			•	FOLLOW(E)={+,⊥}
T	•	•		•	FOLLOW(T)={+,*,⊥}
F	•	•		•	FOLLOW(F)={+,*,⊥}

Figure 2. A FOLLOW Table.

Having computed the necessary FOLLOW sets, we proceed to the third and last phase, the generation of the parse table. As mentioned before, this is traditionally done by row, i.e. state by state. Each state is examined, and information regarding terminal transitions leading from it, as well as reductions available there, are encoded into the ACTION table. The procedure for filling the entries of the ACTION table is as follows.

```
for each state q do
    for each transition q --t--> p do
        Add "S/p" to ACTION[q,t]
        {A "shift to p" move};
    for each reduction on A→ω do
        for each t∈FOLLOW(A) do
            Add "R/A→ω" to ACTION[q,t]
            {A "Reduce using A→ω" move};
```

After these entries have been added, blank entries are filled as "error". The table should have no multiple entries. If it does, the grammar is not SLR(1). For our example, the ACTION[1] table is shown in figure 3.

State/Symbol	+	*	n	⊥
1	error	error	S/9	error
2	S/4	error	error	error
3	Accept	Accept	Accept	Accept
4	error	S/7	S/9	error
5	R/E→E+T	S/7	error	R/E→E+T
6	R/E→T	S/7	error	R/E→T
7	error	error	S/9	error
8	R/T→T*F	R/T→T*F	error	R/T→T*F
9	R/F→n	R/F→n	error	R/F→n
10	R/T→F	R/T→F	error	R/T→F

Figure 3. An SLR(1) ACTION Table.

Historically, there have been good reasons for filling in the ACTION table by row rather than by column. In practice, only a small portion of the states in the LR(0) automaton have conflicts; in our case there were two such

[1] We have neglected the nonterminal transitions in the LR(0) automaton. These are encoded into a separate table called the "GOTO" table, which is of no interest to us because the nonterminal transitions are entirely unaffected by lookahead.

states out of a total of ten. This proportion (about 20%) is in fact quite common. Thus, the table is traditionally filled by row so that states that have no conflicts (which are in the majority) incur in no lookahead computation.

It is well known that some FOLLOW sets are subsets of others. In our example, FOLLOW(E) $\subseteq$ FOLLOW(T) $\subseteq$ FOLLOW(F). Thus FOLLOW(E) must be computed before concluding the computation of FOLLOW(T), and before computing FOLLOW(F). In principle, to obtain the FOLLOW set of some nonterminal A, one may have to first compute the FOLLOW sets of all the other nonterminals. Such is the case here for F. Because of these dependencies, the FOLLOW computation has not been deemed suitable for parallelization. The same occurs for the generation of the ACTION table. Allocating two separate processors for rows p and q of the ACTION table yields no significant advantage, since there may be reductions at p and q, on productions whose left-parts are, say, A_p and A_q, whose FOLLOW sets are **not** independent.

Our approach to the problem is simply to consider filling in the ACTION table by column, rather than by row. Doing so produces completely independent computation for each column. We must first discuss in detail the computation of FOLLOW, as shown in the next section.

3. Computation of FOLLOW

We begin by reversing the order of the "for" loops in the procedure shown earlier, so that the iteration on states is nested within the iteration on terminal symbols, rather than the other way around. The resulting equivalent procedure is as follows.

```
for each symbol t do
    for each state q do
        if q --t--> p is defined then
            Add "S/p" to ACTION[q,t] ;
        for each reduction on A→ω do
            if t∈FOLLOW(A) then
                Add "R/A→ω" to ACTION[q,t] ;
```

Our intent is to allocate a separate parallel processor **to each individual terminal symbol.** The processor will fill in the entire column for that symbol. This can be accomplished because of a critical observation: the reformulated procedure requires knowledge of *whether or not* t$\in$FOLLOW(A) (a boolean value), for which the entire set FOLLOW(A) *is not necessary*. This is in contrast with the earlier procedure, in which the entire set is required because a for loop is used to *enumerate* the elements of FOLLOW(A). Thus the core of the problem of parallelizing the generation of the parser lies in computing whether or not t is an element of FOLLOW(A), without explicitly computing FOLLOW(A) itself. We now show how this can in fact be achieved. We begin by breaking down the FOLLOW set into "Direct" and "Indirect" FOLLOW sets.

$$
\begin{aligned}
\text{FOLLOW(A)} &= \text{IFOLLOW(A)} \cup \text{DFOLLOW(A)},\\
\text{IFOLLOW(A)} &= \cup \{\text{FOLLOW(B)} \mid B \rightarrow \alpha A \gamma,\ \gamma \Rightarrow^* \epsilon\},\\
\text{DFOLLOW(A)} &= \cup \{\text{First(X)} \mid B \rightarrow \alpha A \gamma X \delta,\ \gamma \Rightarrow^* \epsilon\},\\
\text{First(A)} &= \cup \{\text{First(X)} \mid A \rightarrow \gamma X \delta,\ \gamma \Rightarrow^* \epsilon\},\\
\text{First(t)} &= \{t\}.
\end{aligned}
$$

A symbol may follow nonterminal A directly, by appearing at the beginning of a phrase ($\gamma X \delta$) that immediately follows A in the right-part of some production. On the other hand, a symbol may follow nonterminal A indirectly, by appearing in FOLLOW(B), where FOLLOW(B) $\subseteq$ FOLLOW(A). The recursive nature of these equations suggests that terminal symbols propagate, from a First set to other First set(s), from there to FOLLOW set(s), and from FOLLOW set(s) to other FOLLOW set(s). The following relations capture this propagation.

Definition: X ff A if there is a production $A \rightarrow \gamma X \delta$ such that $\gamma \Rightarrow^* \epsilon$. ■

"ff" (pronounced "first-to-first") describes how one First set contributes symbols to another First set. Clearly, t$\in$First(X) iff t ff^* X.

Definition: X fF A if there is a production $B \rightarrow \alpha A \gamma X \delta$ such that $\gamma \Rightarrow^* \epsilon$. ■

"fF" (pronounced "first-to-follow") describes how a First set contributes symbols to a direct FOLLOW set, and thereby to a FOLLOW set. Clearly, t$\in$DFOLLOW(A) iff t$\in$First(X) and X fF A, for some X. Furthermore, the composition of ff^* and fF describes all symbols in DFOLLOW(A), i.e. t$\in$DFOLLOW(A) iff t $(ff^* \circ fF)$ A.

Definition: B FF A if there is a production $B \rightarrow \alpha A \gamma$ such that $\gamma \Rightarrow^* \epsilon$. ■

"FF" (pronounced "follow-to-follow") describes how one FOLLOW set contributes symbols to an indirect FOLLOW set, and thereby to a FOLLOW set. Clearly, FOLLOW(B) $\subseteq$ FOLLOW(A) if B FF^+ A. Thus, the necessary and sufficient conditions for t to be in FOLLOW(A) are (1) t $(ff^* \circ fF)$ A, i.e. t follows A directly, or (2) t $(ff^* \circ fF \circ FF^+)$ A, i.e. t follows A indirectly. Factoring out relation $(ff^* \circ fF)$ from these two cases, we conclude that t$\in$FOLLOW(A) iff t $(ff^* \circ fF \circ FF^*)$ A.

4. Parallel Generation of SLR(1) Parsers

Having characterized SLR(1) lookahead symbols, a very simple algorithm for filling the entries of the ACTION table can now be formulated. The algorithm must compute the reflexive, transitive closure of both relations "ff" and "FF". During the computation of this closure, it is desirable to avoid repeatedly visiting any nonterminals. Thus we require two bit-valued structures "ff_Visited" and "FF_Visited". The algorithm is presented below.

Algorithm Compute_SLR_Action_Table:
Input:LR(0) automaton, ff, fF, FF;
Output: ACTION table;

```
var
    ff_Visited: a bit vector indexed by symbols;
    FF_Visited: a bit vector indexed by nonterminals;

procedure Follow_to_Follow(A):
    begin
        if FF_Visited[A] then return;
        set FF_Visited[A];
        for each B such that A FF B do
            Follow_to_Follow(B);
    end;

procedure First_to_First(X):
    begin
        if ff_Visited[X] then return;
        set ff_Visited[X];
        for each A such that X fF A do
            Follow_to_Follow(A);
        for each Y such that X ff Y do
            First_to_First(Y);
    end;

begin
    for each terminal t do
    begin
        clear ff_Visited[X], for each symbol X;
        clear FF_Visited[A], for each nonterminal A;

        First_to_First(t);

        for each state q do
        begin
            if p=Go(q,t) is defined then
                Add "S/p" to ACTION[q,t];
            for each (q,A→ω) do
                if FF_Visited[A] then
                    Add "R/A→ω" to ACTION[q,t];
        end;
    end;
end;
```

In this algorithm, procedure "Add" announces that the grammar is not SLR(1) if another move already exists in ACTION[q,t]. For each terminal symbol t, the "Visited" vectors are cleared, and First_to_First(t) is called. This results in setting ff_Visited[X], and calling Follow_to_Follow(A), for all X such that t ff^* X, and for all A such that X fF A. The call to Follow_to_Follow(A), in turn, sets FF_Visited[B], for all B such that A FF^* B. Thus the net effect of calling First_to_First(t) is setting FF_Visited[A], for all A such that t ($ff^* \circ fF \circ FF^*$) A. After this call, FF_Visited is "true" for those nonterminals whose FOLLOW sets contain t. After the call, each state in the LR(0) machine is examined, and both shift and reduce entries are made in the ACTION table, as required.

It is critically important to note that the computation performed for each symbol t is *completely* independent of all the other symbols. Thus one parallel processor can be assigned to each symbol t. The parallel processors may share the memory in which the three relations ff, fF and FF are stored. Individually, each processor must 1) run its own copy of the algorithm, 2) have its own column of the ACTION table to fill in, and 3) have its own "Visited" vectors. In fact, the "Visited" vectors are more aptly named "t_is_in_First" and "t_is_in_Follow". The algorithm executed by each parallel processor is as follows.

```
Algorithm Compute_SLR_Action_Table_t:
    Input:LR(0) automaton, ff, fF, FF, t;
    Output:    ACTION: a vector indexed by states;

var
    t_is_in_First: a bit vector indexed by symbols;
    t_is_in_Follow: a bit vector indexed by nonterminals;

procedure Follow_to_Follow(A):
    begin
        if t_is_in_Follow[A] then return;
        set t_is_in_Follow[A];
        for each B such that A FF B do
            Follow_to_Follow(B);
    end;

procedure First_to_First(X):
    begin
        if t_is_in_First[X] then return;
        set t_is_in_First[X];
        for each A such that X fF A do
            Follow_to_Follow(A);
        for each Y such that X ff Y do
            First_to_First(Y);
    end;

begin
    clear t_is_in_First[X], for each symbol X;
    clear t_is_in_Follow[A], for each nonterminal A;

    First_to_First(t);

    for each state q do
    begin
        if p=Go(q,t) is defined then
            Add "S/p" to ACTION[q];
        for each (q,A→ω) do
            if t_is_in_Follow[A] then
                Add "R/A→ω" to ACTION[q];
    end;
end;
```

The storage requirements of each processor are quite reasonable: a bit-vector of length the number of symbols (t_is_in_First), another bit vector of length the number of nonterminals (t_is_in_Follow), and a vector of length the number of states (the ACTION table column for t). In the last of these, each element usually requires an integer to represent (in packed form) the corresponding shift or reduce move.

One processor must be reserved to allocate tasks to the others, but the task allocation scheme is trivial. Any processor can be assigned to any symbol, as long as the ACTION table is re-constructed in a consistent manner. If there are N processors and M terminal symbols, the computation is sped up by a factor of min(N,M). The typical grammar that describes the phrase-structure of a programming language contains several hundred terminal symbols; hence our approach can be profitably implemented on a wide range of parallel computers.

5. Conclusions

We have presented a technique for carrying out the automatic generation of an SLR(1) parser in a totally parallel fashion. The technique consists of computing (in parallel) one column of the parse table independently for each terminal symbol. We have shown that the speedup factor is min(N,M), where N is the number of processors, and M is the number of terminal symbols. We have also shown that the storage requirements are reasonable for each processor.

SLR(1) is one of the best known LR parsing methods. It is well known that LALR(1) is more powerful and more popular than SLR(1). Although not presented here for the sake of simplicity, the parallelization technique shown in this paper also applies to the LALR(1) method, as shown in detail in [19].

References

[1] Aho, A., R. Sethi and J. Ullman, **Compilers. Principles, Techniques and Tools**, Addison-Wesley, 1986.

[2] Anderson, T.J., J. Eve and J.J. Horning, **Efficient LR(1) Parsers**, Acta Informatica 2(1), pp. 12-39, 1973.

[3] Babichev, A.V., **Some theoretical aspects of parallel parsing**, Cybernetics Vol. 18, No. 2, pp. 189-195, March-April, 1982.

[4] Baer, J, and C. Ellis, **Model, Design, and Evaluation of a Compiler for a Parallel Processing Environment**, IEEE Transactions on Software Engineering, SE-3 6, Nov. 1977, pp. 394-405.

[5] Cohen, J., T. Hickey and J. Katsoff, **Upper Bounds for Speedup in Parallel Parsing**, JACM Vol. 29, 1982, pp. 408-428.

[6] DeRemer, F., **Practical Translators for LR(k) Languages**, Ph.D. Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass., 1969.

[7] DeRemer, F., **Simple LR(k) Grammars**, CACM 14(7), pp. 453-460, July 1971.

[8] DeRemer, F. and T. Pennello, **Efficient Computation of LALR(1) Look-ahead sets**, ACM TOPLAS 4(4), pp. 615-649, 1982.

[9] Donegan M.K. and S. Katzke, **Lexical Analysis and Parsing Techniques for a Vector Machine**, SIGPLAN Notices Vol. 10, No. 3, March 1975, pp. 138-145.

[10] Ellis, C.A., **Parallel Compiling Techniques**, Proc. ACM 26th National Conference, 1971, pp. 508-519.

[11] Fischer, C., **On Parsing Context-free Languages in Parallel Environments**, Ph.D. Dissertation, Computer Science Dept., Cornell University, Ithaca, N.Y., 1975.

[12] Fischer, C. and R.J. LeBlanc, **Crafting a Compiler**, Benjamin Cummings, 1988.

[13] Harrison, M., **An Introduction to Formal Language Theory**, Addison-Wesley, Reading, Massachusetts, 1978.

[14] Johnson, S.C., **YACC - Yet another compiler compiler**, Tech. Report CSTR 32, Bell Labs, Murray Hill, N.J., 1974.

[15] Knuth, D., **On the Translation of Languages from Left to Right**, Information and Control 8, pp. 607-639, 1965.

[16] Kristensen B.B. and O.L. Madsen, **Methods for Computing LALR(k) Lookahead**, ACM TOPLAS 3(1), pp. 60-82, 1981.

[17] LaLonde, W.R., **An Efficient LALR Parser Generator**, Technical Report 2, Computer Systems Research Group, University of Toronto, 1971.

[18] Lincoln, N., **Parallel Programming Techniques for Compilers**, SIGPLAN Notices 5, 10, Nov. 1970.

[19] Logothetis, G. and M. Bermudez, **LALR(1) Parser Construction in a Nutshell**, Technical Report UF-CIS-TR-87-7, Department of Computer and Information Sciences, University of Florida, November 16, 1987.

[20] Mickunas, M, and R. Schell, **Parallel Compilation in a Multiprocessor Environment**, Proc. ACM 1978 Annual Conference, 1978, pp. 241-246.

[21] Pager, D., **A Practical General Method for Constructing LR(k) Parsers**, Acta Informatica 7, pp. 249-268, 1977.

[22] Park, J.C.H., K.M. Choe and C.H. Chang, **A New Analysis of LALR Formalisms**, ACM TOPLAS 7(1), pp. 159-175, 1985.

[23] Zosel, M., **A Parallel Approach to Compilation**, Proc. ACM Symposium on Principles of Programming Languages, Oct. 1978, pp. 59-70.

CONCURRENT SYNTHESIS OF COMPOSITE EXPLANATORY HYPOTHESES

Ashok Goel, P. Sadayappan and John R. Josephson

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Abstract

The information processing task of abduction is to infer a hypothesis that best explains a set of data. A typical subtask of this is to synthesize a composite hypothesis that best explains the data set from elementary hypotheses that can explain various portions of the data. In this paper, we present a computational model for concurrent synthesis of composite explanatory hypotheses that can be realized on a distributed memory, message passing, parallel machine. In this model, a process is associated with each datum to be explained as well as with each elementary hypothesis that can explain some portion of the data, and the control of processing alternates between the data and hypotheses' processes. In each cycle of processing, the data and the hypotheses' processes view the problem solving from their perspectives, and add to the growing composite explanatory hypothesis until a best explanation is synthesized. We analyze the time complexity of the concurrent algorithms, and discuss the architectural implications of the model.

Abductive Inference

Abduction is the very general information processing task of inferring a hypothesis that best explains a set of data [2,7]. Abduction occurs, for instance, in diagnostic problem solving, where the data is in the form of manifestations (or symptoms), and the explanatory hypotheses are about component malfunctions (or diseases) [9,10]. A typical subtask of abduction is *classification* of the observed data onto stored elementary hypotheses. In simple abductive problems, *e.g.*, diagnosis under the single fault assumption, the classification subtask often yields elementary hypotheses that can individually explain the entire data. For such problems, the elementary hypothesis that most plausibly explains the data represents the best explanation. In general, however, an elementary hypothesis that can account for the entire data may not be available. Instead, a composite hypothesis has to be *synthesized* from elementary hypotheses that can explain various portions of the data. Synthesis of composite explanatory hypotheses is thus another typical subtask of abduction.

Synthesizing a composite hypothesis that best explains a set of data can be computationally very expensive, especially in the presence of certain types of interactions between the elementary hypotheses [1]. This suggests exploitation of concurrency in performing the synthesis subtask [8]. We have elsewhere reported [3] on a computational model for concurrent synthesis of composite explanatory hypotheses based on a shared memory, multiprocessor architecture. Our work on the "blackboard" model led us to think that a distributed memory, message passing parallel architecture may provide a more modular organization of knowledge and processing for the synthesis subtask. Thus in this paper we present a computational model for distributed synthesis of composite explanatory hypotheses.

Characterization of the Synthesis Task

Let $D = \{d_i | i = 1, \ldots, n\}$ be a finite set of n observed data. Let $H = \{h_j | j = 1, \ldots, m\}$ be a finite set of m elementary explanatory hypotheses. Let e be a map from subsets of H to subsets of D; $e : 2^H \rightarrow 2^D$. We interpret $e(H_j) = D_j$, where $H_j \subseteq H$ and $D_j \subseteq D$, as the *explanatory coverage* of H_j, *i.e.* H_j can explain all members of D_j. Let $V = \{v_k | k = 1, \ldots, l\}$ be a finite set of l discrete values. Let b be a map from H to V, *i.e.*, $b : H \rightarrow V$. We interpret $b(h_j)$ as the *prima facie* belief in hypothesis h_j. We may characterize the task of synthesizing a composite explanatory hypothesis by its input: D, H, e and b, and its output: C, where C is a subset of H that best explains D. The synthesis task is *linear* [1] if

$$\forall h_i, h_j \in H, (e(h_i) \cup e(h_j)) = e(\{h_i, h_j\})$$

and *monotonic* [1] if

$$\forall h_i, h_j \in H, (e(h_i) \cup e(h_j)) \subseteq e(\{h_i, h_j\})$$

In this paper we consider only the linear version of the general synthesis task. We note that linearity of the task entails monotonicity. Thus we assume that the elementary hypotheses are non-interacting and offer explanatory alternatives where their coverages overlap.

Characterization of the "Best" Explanation

This characterization of the synthesis task is incomplete since we have not yet specified what is meant by the best explanation. A general operational characterization of the "best" explanation is based on the following three criteria:

Maximal explanatory coverage of data: A hypothesis C_1 is a better explanation of D than another hypothesis C_2 if $e(C_2) \subset e(C_1)$ (strictly, if $e(C_2) \cap D \subset e(C_1) \cap D$). Ideally, the synthesized C should provide complete explanatory coverage of D.

Maximal belief in hypothesis: If $e(C_1) = e(C_2)$, then C_1 is a better explanation of D than C_2 if $b(C_1) > b(C_2)$ which denotes that $\forall d \in D$, $\forall h_2 \in C_2$ such that $d \in e(h_2)$, $\exists h_1 \in C_1$ such that $d \in e(h_1) \wedge b(h_1) \geq b(h_2)$. This criterion specifies that the elementary hypotheses in C should be locally optimal in terms of their belief values.

Minimal hypothesis: If $e(C_1) = e(C_2)$ and $b(C_1) = b(C_2)$, then C_1 is a better explanation of D than C_2, if $C_1 \subset C_2$. This global optimization condition specifies that C should be *parsimonious*.

We note the *precedence* relation between the three criteria according to which maximal coverage of the data has the highest precedence and parsimony of the composite hypothesis has the lowest. We note also that depending on the maps e and b, the synthesis task may be underconstrained, in which case the synthesized explanation would only be *a* "best" explanation.

A Concurrent Model of the Task

Decomposition of the Synthesis Task

The task of synthesizing a C that "best" explains D may be decomposed into two phases: generation, and testing. In the generation phase, a C that achieves the goal of complete explanatory coverage of the data and satisfies the local constraints on the choice of elementary hypotheses may be generated. In the testing phase, the generated C may be tested for the global constraint of parsimony and if possible, further optimized. The generation phase itself is further decomposable into three steps corresponding to the three types of hypotheses that need to be included in C. In the first step, the $h_j \in H$ that are *necessary* for explaining $d_i \in D$ may be found by a specialized form of means-ends mechanism which views explaining each d_i as a subgoal of the synthesis of C. A hypothesis h_1 is necessary for explaining some datum d if there exists no other h_2 that can explain the d. We denote this set of hypotheses as $C_{\text{Essential}}$. In the next step, the $h_j \in H$ that are the *best* explanations for some $d_i \in D - e(C_{\text{Essential}})$ are found by the same means-ends mechanism, where a hypothesis h_1 is clearly the best explanation for some datum d if its belief value $b(h_1)$ is above some high threshold θ, and there is no h_2 that can explain the d and has a belief value above θ. We denote this set of hypotheses as C_{Firm}. In the final step, the $h_j \in H$ that are needed for explaining $d_i \in D - e(C_{\text{Essential}}) - e(C_{\text{Firm}})$ are found; we denote this subset of components as C_{In}. The composite hypothesis generated is given by $C = C_{\text{Essential}} \cup C_{\text{Firm}} \cup C_{\text{In}}$.

The hypotheses in $C_{\text{Essential}}$, C_{Firm} and C_{In} differ from each other in the *firmness* with which they are included in C. The *In* components are only weakly included in C, and should to be tested for explanatory superfluousness, where a hypothesis is explanatorily superfluous if removing it from C does not reduce $e(C)$. The *Essential* components are strongly included in C, cannot be removed from C without reducing $e(C)$, and need not be tested for explanatory superfluousness, Similarly, by the manner in which C_{Firm} was generated, the *Firm* hypotheses are firmly included in C, cannot be removed without reducing $b(C)$ (relative to other composite hypotheses), and need not be tested for explanatory superfluousness.

Concurrency in the Synthesis Mechanism

It is clear that certain aspects of synthesizing C are *inherently sequential*, *e.g.*, the testing of the composite hypothesis for parsimony should serially follow the generation phase, and even in the generation phase, the sequential order of the three steps should be maintained. However, there also is significant concurrency in the process. There are really two types of questions that are raised during the synthesis of C. The first type is from the perspective of each datum $d_i \in D$ and is of the form "which hypothesis $h_j \in H$ can best explain me?". This question can be asked and answered for each datum concurrently with others. The second type of question is from the perspective of each hypothesis $h_j \in H$ and is of the form "which elements of D can I be used to explain?". Again, this question can be asked and answered for each hypothesis concurrently with others.

This suggests that a process be assigned to each $d_i \in D$ as well as to each $h_j \in H$. Let $P = \{p_i | i = 1, \ldots, n\}$ be a set of n processes, one for each $d_i \in D$. Each process $p_i \in P$ represents the perspective of the corresponding datum $d_i \in D$ in the synthesis of C. The P processes use identical algorithms and can execute concurrently. Similarly, let $Q = \{q_j | j = 1, \ldots, m\}$ be a set of m processes, one for each $h_j \in H$. Each process $q_j \in Q$ represents the perspective of the corresponding hypothesis $h_j \in H$. Again, the Q processes use identical algorithms and can execute concurrently.

A Distributed Mechanism for the Task

In distributed synthesis of composite explanatory hypotheses, each process has access only to its own local memory. Communication between the processes occurs by message passing only; there is no global updating of shared variables. Synchronization between the processes can be achieved by adopting the framework of Communicating Sequential Processes (CSP) [6]. In CSP, communication between processes occurs when one process names another as the destination and the second process names the first as the source. Synchronization is achieved by delaying the execution of an input or output command until the other process is ready with the corresponding output or input.

The messages between the P and Q processes are semantically encoded and the response of a process to a message depends on the semantics of the message it receives. For example, if process $p_1 \in P$ corresponding to datum d_1 determined that that some $h_1 \in H$ was necessary for explaining d_1, then it sends an *Essential!* message to process $q_1 \in Q$ representing h_1. On receiving this message, q_1 sends an *Explained!* message to the processes $p_i \in P$ corresponding to $d_i \in D$ that h_1 can explain. In addition to messages of this type, the processes are allowed to send (and receive) *Null* messages.

The control of processing alternates between the P processes and the Q processes. In each cycle of processing, when the P processes are executing the Q processes are idle; when the P processes have finished executing some step, they communicate their results to the Q processes, and the Q processes can start executing. Similarly, when the Q processes are executing the P processes are idle; again, when the Q processes have finished executing some step, they communicate their results to the P processes, and the P processes can start executing. Thus in each cycle of processing, the P and the Q processes view the synthesis task from different perspectives and add to the growing composite explanatory hypothesis. After three such cycles

that correspond to computing $C_{\text{Essential}}$, C_{Firm} and C_{In} respectively, a composite hypothesis C that achieves the goal of complete explanatory coverage and satisfies the local constraints on the choice of elementary hypotheses is generated. The generated C is now tested for the global constraint of parsimony in a similar manner.

Distributed Hypothesis Generation

We now present concurrent algorithms for distributed generation of C; detailed concurrent algorithms written in CSP are given in [4]. Since the n processes in P use identical algorithms, and the m processes in Q also use identical algorithms, it suffices to describe the processing from the perspectives of a datum $d \in D$ represented by a process $p \in P$ and an elementary hypothesis $h \in H$ represented by a process $q \in Q$. The process q initially has information specifying the hypothesis h that it represents, the explanatory coverage $e(h)$ of the hypothesis, the belief value $b(h)$ of the hypothesis, and the data D to be explained. Similarly, the process p initially has information specifying the datum d that it represents, the cardinality of H.

In the first cycle of processing, the *Essential* components are identified. The processing begins when process q sends an *Essential?* message to processes in P corresponding to data $d_i \in e(h)$, and *Null* messages to other processes in P. Process p receives messages from processes in Q. Now from the perspective of d, one of three things can happen:
(i) p receives no *Essential?* messages. Then d is *Unexplainable*, and p does nothing;
(ii) p receives exactly one *Essential?* message. Then the hypothesis corresponding to the process in Q from whom p received the message is necessary for inclusion in C, and p sends an *Essential!* message to that process;
(iii) p receives more than one *Essential?* message. Then p sends a *Null* message to the processes in Q from whom p had received the *Essential?* messages.

Process q receives messages from processes in P corresponding to $d_i \in e(h)$. In the second cycle of processing, the *Firm* hypotheses are identified. At the end of the first cycle, from the perspective of h one of two things can happen:
(i) q receives at least one *Essential!* message. Then q sends an *Explained!* message to processes in P corresponding to $d_i \in e(h)$;
(ii) q receives only *Null* messages. Then q sends *Firm?* messages, along with the belief value $b(h)$ of the hypothesis it represents, to processes in P corresponding to $d_i \in e(h)$.

Process p receives messages from the processes in Q corresponding to the hypotheses that can explain d. Now from the perspective of d, one of three things can happen:
(i) p receives at least one *Explained!* message. Then p does nothing;
(ii) p receives only *Firm?* messages. Then if there is some h_1 among the hypotheses that can explain d such that $b(h_1)$ is above some high threshold θ, and there is no h_2 that can explain d and has a belief value above θ, p sends a *Firm!* message to the q_1 process representing the h_1 hypothesis, and *Null* messages to processes in Q corresponding to other hypotheses that can explain the d;
(iii) p receives only *Firm?* messages, and there is no hypothesis that is clearly the best explanation for d. Then p sends *Null* messages to processes in Q corresponding to hypotheses that can explain the d.

Process q receives messages from processes in P corresponding to $d_i \in e(h)$. In the third cycle of processing, the *In* hypotheses are identified. At the end of the second cycle, from the perspective of h one of two things can happen:
(i) q receives at least one *Firm!* message. Then q sends an *Explained!* message to processes in P corresponding to $d_i \in e(h)$;
(ii) q receives only *Null* messages. Then q sends *In?* messages to processes in P corresponding to $d_i \in e(h)$.

Process p receives messages from the processes in Q corresponding to the hypotheses that can explain d. Now from the perspective of d, one of two things can happen:
(i) p receives at least one *Explained!* message. Then p does nothing;
(ii) p receives only *In?* messages. Then p selects the h with the highest belief value among the hypotheses that can explain d. If two (or more) hypotheses can explain equally well, then p breaks the tie between them by selecting the hypothesis with the largest explanatory coverage. If even this would not break the tie, then the selection is made at random. p then sends an *In!* message to the q corresponding to the selected hypothesis, and *Null* messages to processes in Q corresponding to other hypotheses that can explain d.

q receives messages from processes in P corresponding to $d_i \in e(h)$. At the end of the third cycle, a C consisting of the *Essential*, *Firm*, and *In* hypotheses has been generated.

Distributed Hypothesis Testing

If $C_{\text{In}} \subseteq C$ is empty, then the generated C is best explanation since the *Essential* hypotheses cannot be removed from C without reducing $e(C)$, and the *Firm* hypotheses cannot be removed without reducing $b(C)$. If, however, C_{In} is nonempty, then C should be tested for parsimony by testing the *In* hypotheses in it for explanatory superfluousness. Again, it suffices to present an algorithm for distributed testing of C from the perspectives of a datum $d \in D$ represented by a process $p \in P$ and a hypothesis $h \in C_{\text{In}}$ represented by a process $q \in Q$. The testing of C begins when process q sends a *Superfluous?* message to processes in P corresponding to data $d_i \in e(h)$. Process p receives messages from the processes in Q corresponding to the *In* hypotheses that can explain d. Now from the perspective of d, one of three things can happen:
(i) p receives no messages. Then p does nothing;
(ii) p receives exactly one *Superfluous?* message. Then if the d is explained by some *Essential* or *Firm* hypothesis, p sends a *Superfluous!* message to the q_j from whom it had received the message, else it sends back a *Null* message;
(iii) p receives more than one *Superfluous?* message. Again, if the d is explained by some *Essential* or *Firm* hypotheses, then p sends back a *Superfluous!* message to all processes in Q from whom it received messages. If, however, d is not thus explained,

then p selects the hypothesis with the highest belief value among the *In* hypotheses that can explain d. Again, if two (or more) hypotheses can explain d equally well, then p breaks the tie between them by selecting the hypothesis with the largest explanatory coverage, and if this would not break the tie, then the selection is made at random. p then sends a *Null* to the process in Q corresponding to the selected hypothesis and a *Superfluous!* message to processes in Q corresponding to other *In* hypotheses that can explain d.

Process q receives messages from processes in P corresponding to $d_i \in e(h_j)$. Now from the perspective of h, one of two things can happen:
q receives only *Null* messages. Then q does nothing and h remains *In*;
q receives at least one *Superfluous!* message. Then if the number of such messages received equals $|e(h)|$, h is explanatorily superfluous and is no longer *In*.

We note that under certain conditions h could be explanatorily superfluous even if the number of *Superfluous!* messages received by q in the last step is less than $|e(h)|$. This could happen, for instance, if the explanatory coverages of two or more hypotheses completely overlap and their belief values are exactly equal. In such case, each p process corresponding to some datum in the overlapping explanatory coverages would randomly select one of the candidate hypotheses for inclusion in C. Moreover, since each such p process makes its selection locally, the randomness in hypothesis selection may also result in the retention in C of more than one of the candidate hypotheses even though all but one of them is explanatorily superfluous. This problem can be solved if during testing of C for parsimony, when faced with random hypothesis selection, each p process alerts a special process, say R, of the situation by sending a message containing a list of the candidate hypotheses. Process R receives the messages from each such p process and makes a random choice between the hypotheses. It then retains the selected hypothesis in C and removes the other hypotheses from it.

A similar problem may occur if the belief values of more than two *In* hypotheses are exactly equal and their explanatory coverages overlap cyclically. Let us consider as an example a situation in which $h_1, h_2, h_3 \in C_{\text{In}}$, $b(h_1) = b(h_2) = b(h_3)$, $e(h_1) = \{d_1, d_2\}$, $e(h_2) = \{d_2, d_3\}$ and $e(h_3) = \{d_3, d_1\}$, where $d_1, d_2, d_3 \in D$. Since the belief values and the cardinality of explanatory coverages of h_1, h_2, h_3 are equal, the p processes corresponding to d_1, d_2, d_3 make their choices randomly. Again, the p processes inform process R of the situation, and again R makes a random choice between the hypotheses retaining in C, for this example, any two of the hypotheses and removing the third. At the end of processing by R, a C that "best" explains D has been synthesized.

Discussion of the Concurrent Model

Analysis of Algorithms

The worst case time complexity for distributed generation of C is given by

$$T_{\text{hypothesis generation}} = O(m + n)$$

where m is the cardinality of H and n is the cardinality of D. Similarly, the worst case complexity for distributed testing of C for parsimony is given by

$$T_{\text{hypothesis testing}} = O(m_{\text{In}} \times n)$$

where m_{In} is the number of *In* hypotheses in C. We may compare this to the complexity of a non-optimal serial algorithm that has been used to build to several major abductive knowledge-based systems [7]. The worst case time complexity for the sequential generation of C is given by [1]

$$T_{\text{hypothesis generation}} = O(n \times (m + n \times log(n))$$

and the worst case complexity of testing C for parsimony is given by

$$T_{\text{parsimony testing}} = O(m \times n \times log(n))$$

Moreover, in the sequential algorithm, the *Essential* hypotheses are not found during the generation of C. Instead, a C is assembled without regard for the *Essential* hypotheses and then tested for essentialness of the hypotheses. The worst case time complexity for testing of C for essentialness of hypotheses is given by [1]

$$T_{\text{essentialness testing}} = O(m \times n \times (m + n \times log(n)))$$

We note that the constants in the time complexities for sequential and distributed synthesis of C are comparable since in both cases they arise from linear search. It thus appears that the concurrent algorithms provide a linear speed up of processing over the sequential algorithms. However, there are several reasons for caution about this claim. Firstly, the serial algorithms are non-optimal. Secondly, the time complexities are for the worst case, and not for the "average" case since the "average" case is domain dependent. Thirdly, the complexities are valid only under the assumptions of linearity of the synthesis task. Finally, we have not accounted for the costs of communication between the P and the Q processes in calculating the complexities of the concurrent algorithms. Despite these caveats, it is clear that distributed synthesis provides an attractive model for the synthesis of composite explanatory hypotheses.

Architectural Implications

There are several interesting and somewhat unusual aspects to the computational model for concurrent synthesis of C from the viewpoint of realizing it on a distributed memory, message passing, parallel computer architecture. Firstly, the parallelism between the semi-autonomous P and Q processes is fine-grained. Secondly, at any given time during the processing, the P and Q processes are either idle or executing the same instruction on different data. Thirdly, the process of synthesizing C is communication intensive. For instance, in the first cycle during the generation of C, each process $q_j \in Q$ communicates with every

process $p_i \in P$. Fourthly, for real world problems, the number of P and Q processes is potentially very large. Even for small knowledge-based systems that use abductive inference to perform medical diagnosis in limited domains, for instance, the number of P and Q processes is typically in the hundreds. A full scale medical diagnostic system may well require thousands, or even tens of thousands of such processes.

It appears that among the existing machines, the Connection machine [5] may be the most suitable architecture for realizing this model for distributed synthesis of C. The Connection machine is a distributed memory, message passing, parallel computer which is precisely the architecture required for realizing the model. It is a single-instruction, multiple data stream machine which suits the control of processing in the model. It is based on the hypercube architecture which helps to keep the communication costs within acceptable limits. Finally, it supports the type of massive, fine-grained parallelism between a large number of small, semi-autonomous processes that is present in this model for synthesizing C.

Interactions between Hypotheses

So far we have discussed only the linear version of the general synthesis task, where we assumed that the hypotheses $h_j \in H$ are non-interacting. In fact, several distinct types of interaction between two hypotheses $h_1, h_2 \in H$ have been identified [7]:
Associativity: The inclusion of h_1 in C suggests the inclusion of h_2. Such an interaction may arise if the generator has knowledge of, say, a statistical association between h_1 and h_2.
Additivity: h_1 and h_2 cooperate additively where their explanatory capabilities overlap. This may happen if h_1 and h_2 can separately explain some $d \in D$ only partially, but collectively can explain it fully.
Incompatibility: h_1 and h_2 are mutually incompatible, *i.e.*, if one of them is included in C then the other should not be included.
Cancellation: h_1 and h_2 cancel the explanatory capabilities of each other in relation to some $d \in D$. For example, h_1 may imply that some data value will increase, while h_2 may imply that the value will decrease, thus canceling each others explanatory capability with respect to that datum.

We note that the general synthesis task is *nonlinear* in the presence of incomptability interactions and *nonmonotonic* in the presence of cancellation interactions. The concurrent model for distributed synthesis of composite explanatory hypotheses presented in this paper can be extended to accommodate the above interactions by allowing for message passing between the Q processes. This enables the hypotheses' processes to negotiate among themselves and resolve the conflicts that arise due to the presence of these interactions.

Acknowledgments

This paper has benefited from discussions with B. Chandrasekaran and N. Soundararajan, and comments by anonymous reviewers. This work has been supported by the Defense Advanced Research Projects Agency, RADC Contract F30602-85-C-0010, and the Air Force Office of Scientific Research, grant 87-0090.

References

[1] D. Allemang, M. Tanner, T. Bylander and J. Josephson, "On the Computational Complexity of Hypothesis Assembly," in *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August 1987, pages 1112-1117.

[2] E. Charniak and D. McDermott, *Introduction to Artificial Intelligence*, Reading, MA: Addison-Wesley, 1985.

[3] A. Goel, J. Josephson and P. Sadayappan, "Concurrency in Abductive Reasoning," in *Proceedings of the DARPA Workshop on Knowledge-based Systems*, St. Louis, April 1987, pages 86-92.

[4] A. Goel, P. Sadayappan, J. Josephson and N. Soundararajan, "Distributed Synthesis of Composite Explanatory Hypotheses," Technical Report, Laboratory for Artificial Intelligence Research, Department of Computer and Information Science, The Ohio State University, November 1987.

[5] W. D. Hillis, *The Connection Machine*, Cambridge, MA: MIT Press, 1986.

[6] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM* 21(8):666-677, August 1978.

[7] J. Josephson, B. Chandrasekaran, J. Smith and M. Tanner, "A Mechanism for Forming Composite Explanatory Hypotheses," *IEEE Transactions on Systems, Man and Cybernetics* 17(3):445-454, 1987.

[8] J. Pearl, "Distributed Revision of Composite Beliefs," *Artificial Intelligence* 33(2):173-215, 1987.

[9] H. Pople, "The Formation of Composite Hypotheses in Diagnostic Problem Solving: An Exercise in Synthetic Reasoning," in *Proceedings of the Fifth International Joint Conference in Artificial Intelligence*, Cambridge MA, August 1977, pages 1030-1037.

[10] J. Reggia, "Diagnostic Expert Systems Based on a Set Covering Model," *International Journal of Man-Machine Studies* 19:437-460, 1983.

ALGORITHM AND PERFORMANCE NOTES FOR BLOCK LU FACTORIZATION

by
Jim Armstrong
Research Mathematician
CONVEX Computer Corporation

Abstract

Block algorithms for matrix factorizations have gained much attention in recent years as an approach that takes advantage of computer architectures with a relatively small amount of very high speed cache memory. The advantage of this approach is efficient re-use of data while it is in cache. Block algorithms are also useful for general purpose single or multi-headed vector machines with interleaved memory. However, their performance advantage over traditional matrix-vector algorithms is very problem-dependent. An example of such behavior is the *LU* factorization of a general, dense matrix. This paper presents a block *LU* factorization algorithm that is appropriate for the CONVEX C Series machines. Performance comparisons are provided against the matrix-vector Crout algorithm.

1. Introduction

Gaussian elimination can be viewed as the reduction of a matrix, A, to upper triangular form using multiplication by transformation and row permutation matrices as follows:

$$M_{n-1}P_{n-1}M_{n-2}P_{n-2}\cdots M_1P_1A = L^{-1}A = U$$

where the P_k are row permutation matrices and the M_k are defined by

$$y_k = [\,0\;0\;\cdots\;m_k\,]^t \quad M_k = I + y_k e_k^t$$

where there are k zeros in y_k, e_k is the k-th column of the identity matrix and m_k is the vector of negative multipliers. A simple manner of applying a block p x p reduction is to develop an algorithm for the update

$$A_p := (\,M_{k+p-1}P_{k+p-1}\cdots M_{k+1}P_{k+1}M_kP_k\,)\,A_p$$

or an update based on the product of p Gauss transformation and row permutation matrices. This approach is similar to that in Dongarra [4]. The upper left-hand corner of A_p will be $a_{k+p,k+p}$ and the lower right-hand corner will be a_{nn}. The notation $x^{(j)}$ will be used to denote a section of a vector beginning at element j. Define

$$m_k^{(j)} = [\,m_{k_j}\;m_{k_{j+1}}\;\cdots\;m_{k_n}\,]^t$$
$$a_k = [\,a_{kk}\;a_{k+1,k}\;\cdots\;a_{nk}\,]^t$$
$$\bar{a}_k = [\,a_{k+1,k}\;a_{k+2,k}\;\cdots\;a_{nk}\,]^t$$

Following is the rank-p update formula for $p = 2$. Operations relating to pivoting are deleted to conserve space.

$k = 1$
while $k < n-1$
  $s_k = 1/a_{kk}$
  $m_k = s_k\bar{a}_k$
  $a_{k+1} := a_{k+1} - a_{k,k+1}m_k$

  $v_k^t = [\,a_{k,k+2}\;a_{k,k+3}\;\cdots\;a_{k,n}\,]$

  $v_{k+1}^t = [\,a_{k+1,k+2}\;a_{k+1,k+3}\;\cdots\;a_{k+1,n}\,] - m_{k_1}\,v_k^t$
  $s_{k+1} = 1/a_{k+1,k+1}$
  $m_{k+1} = s_{k+1}\bar{a}_{k+1}$

  $A_2 := A_2 - m_k^{(2)}v_k^t - m_{k+1}v_{k+1}^t$
  $k := k + 2$
end k

Any remaining portion of the computation that cannot be included in the paired updating scheme is finished off with a scalar version of the rank-1 algorithm. The algorithm for $p = 3$ is

$k = 1$
while $k < n-1$
  $s_k = 1/a_{kk}$

$m_k = s_k \bar{a}_k$
$a_{k+1} := a_{k+1} - a_{k,k+1} m_k$

$v_k^t = [\ a_{k,k+2}\ a_{k,k+3}\ \cdots\ a_{k,n}\ ]$

$t_{k+1}^t = [\ a_{k+1,k+2}\ a_{k+1,k+3}\ \cdots\ a_{k+1,n}\ ]^t$

$v_{k+1}^t = t_{k+1}^t - m_{k_1} v_k^t$
$s_{k+1} = 1/a_{k+1,k+1}$
$m_{k+1} = s_{k+1} \bar{a}_{k+1}$

$a_{k+2} := a_{k+2} - a_{k,k+2} m_k^{(2)} - a_{k+1,k+2}\ m_{k+1}$

$t_{k+2}^t = [\ a_{k+2,k+3}\ a_{k+2,k+4}\ \cdots\ a_{k+2,n}\ ]$

$v_{k+2}^t = t_{k+2}^t - m_{k_2} v_k^{(2)^t} - m_{k+1_1} v_{k+1}^{(2)^t}$
$s_{k+2} = 1/a_{k+2,k+2}$
$m_{k+2} = s_{k+2} \bar{a}_{k+2}$

$A_3 := A_3 - m_k^{(3)} v_k^{(2)^t} - m_{k+1}^{(2)} v_{k+1}^{(2)^t} - m_{k+2} v_{k+2}^t$
$k := k + 3$
end k

As before, scalar cleanup with $p = 1$ is used. The rank-2 algorithm requires two multiply-adds per two memory references in the update, which runs at full speed on a machine with one path to memory. The rank-3 approach runs even faster because the columns of the submatrices at each stage are loaded and stored fewer times. The rank-3 updates can also be coded to run the column memory references one column ahead of the current update. This eliminates the load/store pipeline latency upon branching to the top of the loop. This approach saves a significant amount memory start-up time in the updates.

These algorithms are different than traditional block *LU* algorithms in that the initial "block" reduction is not performed on a 2 x 2 or 3 x 3 block. Instead, the operations are performed in a manner similar to the standard algorithm. This method retains the same (longer) vector lengths as the standard algorithm. It is also very easy to integrate the block column and row operations with the pivoting memory traffic. In fact, the block p x p *LU* factorization is a side effect of the current algorithm! Based on the work in Armstrong [1], the best performance to date from such a method has been to use rank-3 updates and switch to rank-2 updates after the reduction reaches a certain point. The current crossover is when the submatrix order reaches 128 or less. If the entire factorization is coded in assembly language, then the first 128-element (or less) section of the last multiplier array can be retained in a vector register for use in the first block section of the rank-p update. This is currently the approach used on the C120. Because of the increased memory subsystem efficiency on the C210, this method has only a negligible advantage over the rank-3 algorithm. The performance in Megaflops (Mflops) is provided in the tables below. Dongarra's *matgen* routine (from the LINPACK benchmarks) was used to generate the system of equations and the flop count used was $2/3n^3 + 2n^2$. The M-V column represents the matrix-vector Crout algorithm implemented in FORTRAN with calls to assembly language IDAMAX and DGEMV.

C120 Rank-p update performance, 64-bit Mflops			
n	M-V	Rank-3	Rank-(3,2)
50	5.4	7.2	7.4
100	9.3	11.5	11.5
200	12.9	13.8	14.2
300	14.5	14.9	15.3
400	14.6	15.3	15.7
1000	15.0	15.9	16.6

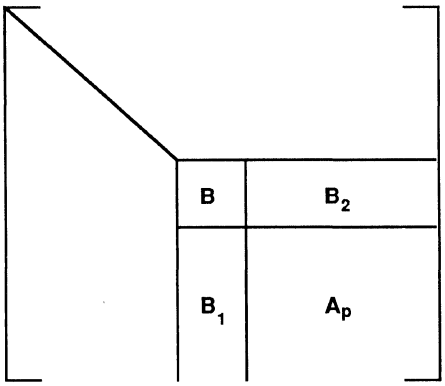
C210 Rank-3 update performance, 64-bit Mflops		
n	M-V	Rank-3
50	13.9	20.5
100	25.7	33.7
200	34.8	40.7
300	38.9	43.2
400	40.8	43.9
1000	41.1	44.0

2. General Block LU Factorization Algorithm

An excellent introduction to the rationale and development of block algorithms is provided in references [2,3,5,6]. This section presents an algorithm which generalizes the rank-p approach. The diagram below illustrates the fundamental idea behind the algorithm. The p x p block submatrix B

contains - in effect - an LU factorization of a block of the submatrix at stage k. However, the computation of multipliers and row updates is extended to the extremes of the submatrix, which produces sub-blocks B_1 and B_2. As before, the main idea of the algorithm is to compute B_1 and B_2. The block LU factorization in B is the result this computation. This is in contrast to traditional block algorithms that compute B first, and use this result to generate B_1 and B_2. The remainder of the submatrix, A_p, is updated according to a formula of the form

$$A_p := A_p + \alpha B_1 B_2$$



where $\alpha = +/-1$ depending on whether positive or negative multipliers are used. The algorithm is implemented in two parts. The first part computes the submatrices B_1, and B_2 for a given value of p. The second part simply consists of a call to _GEMM from the level III BLAS.

B_1 and B_2 are generated, from a matrix-vector algorithm that applies rank-1 updates one column and row at a time. The advantage of this approach is fewer memory references than in a standard SAXPY algorithm for applying the updates. The algorithm requires an initial pivot operation to "prime" it. The main loop consists of computing a vector of multipliers for the j-th stage. This vector, along with the previous multiplier vectors from the beginning of the block, are used to update the next column. This update can be formulated as a matrix-vector product since all the updates are not done at once. Instead, they are applied "on the fly". Next, the pivot operation for the following stage is performed. A transposed matrix-vector product is then used to apply all the previous row updates to the remainder of row $j+1$ of U.

The above summary can be mathematically illustrated by partitioning the sub-blocks of B_1 and B_2 at stage j as follows

$$B_1^j = \begin{bmatrix} B_1^{j-1} & b_1 \end{bmatrix} \qquad B_2^j = \begin{bmatrix} u_{j+1} & B_2^{j-1} \\ 0 & b_2^t \end{bmatrix}$$

Also, define l_{j+1}^t as the first row of the submatrix B_1^{j-1}. At this point, everything has been computed except b_1 and b_2^t. These vectors are generated by the following equations:

$$b_1 := b_1 - B_1^{j-1} u_{j+1} \qquad b_2^t := b_2^t - l_{j+1}^t B_2^{j-1}$$

Following is the algorithm for computing B_1, and B_2, which is a ***black bock*** in the main algorithm. The current stage of the reduction is denoted by k and the block size by p. The main loop is executed for $j = k, k+1, \ldots, k+p-2$. Inside this loop, the submatrix B_1^{j-1} has its upper left-hand corner at $a_{j+1,k}$ and is an $(n-j)$ x $(j-k+1)$ matrix. B_2^{j-1} has its upper left-hand corner at $a_{k,j+2}$ and is a $(j-k+1)$ x $(n-j-1)$ matrix. The vectors u_{j+1} and l_{j+1}^t are of length $j-k+1$. u_{j+1} has its initial element at $a_{k,j+1}$, and l_{j+1}^t starts at $a_{j+1,k}$. b_1 has its initial element at $a_{j+1,j+1}$, and b_2^t starts at $a_{j+1,j+2}$. In the following algorithm, the phrase "pivot for stage j" indicates the operation of finding the pivot row for stage j and performing the row interchange. A test for zero pivots is assumed. Also, $\overline{a}_j$ has the same definition as in previous algorithms.

pivot for stage k - primer for the algorithm

For $j = k, k+p-2$
  $s = 1/a_{jj}$
  $\overline{a}_j := s\overline{a}_j$

  $b_1 := b_1 - B_1^{j-1} u_{j+1}$

  pivot for stage $j+1$

  $b_2^t := b_2^t - l_{j+1}^t B_2^{j-1}$

end j

$s = 1/a_{k+p-1,k+p-1}$
$\overline{a}_{k+p-1} := s\overline{a}_{k+p-1}$

An interesting feature of the above algorithm is that, if the block size is set to n, the result is the same basic approach as the matrix-vector version of Crout factorization. Thus, one can derive such factorization algorithms from the viewpoint of compact elimination or from applying rank-1 updates "on the fly."

The complete block LU factorization can now be presented for a given block size p. Let $m = n-k-p+1$. Then the submatrix A_p that is updated has its upper left-hand corner at $a_{k+p,k+p}$ and is an m x m matrix. B_1 has its upper left-hand corner at $a_{k+p,k}$ and is an m x p matrix. B_2 has its upper left-hand corner at $a_{k,k+p}$ and is a p x m matrix. Assume that n/p is computed as in Fortran.

$l = n/p$
$k = 1$
For $i = 1,l$
  compute B_1 and B_2

  $A_p := A_p - B_1B_2$

  $k := k + p$
end i

Clean up leftover submatrix beginning at a_{kk} using the rank-2 update algorithm described in section 4.

3. Performance Notes

Experiments conducted on the CONVEX C120 have indicated that in order to approach the performance of the specific rank-2 and rank-3 algorithms in section 1, large block sizes are required. The asymptotic performance seems to be reached with $p = 200$. This indicates the use of a multi-algorithm approach. For $n \leq 200$, the rank-3 algorithm can be used. For $n > 200$, the general block algorithm is used with rank-(2,1) cleanup in FORTRAN.

The general block algorithm with assembly language IDAMAX, DGEMV, and DGEMM executes at 14.98 Mflops for a 300 x 300 system. For $n = 1000$, it executes at 15.75 Mflops. For $n > 1000$, the general block algorithm asymptotically performs 1.2 Mflops faster than matrix-vector Crout. On the C210, the asymptotic performance advantage is 2.3 Mflops

An advantage to the general block algorithm is that it can be easily implemented in FORTRAN with calls to the level II and III BLAS. A disadvantage over the specific rank-p update algorithms with small values of p is the inability to merge the computation of B_1 and B_2 with the pivoting memory traffic. This is reason for the performance advantage of the rank-p approach.

A later paper will describe the implementation and parallel performance on a dual-headed CONVEX C220.

4. References

[1] Armstrong, J., 1987, "Optimization of Householder Transformations. Part I: Linear Least Squares", Proceedings of 1987 International Conference on Parallel Processing, Pennsylvania State Univ. Press, University Park, PA.

[2] Bischof C. and VanLoan C., 1987, "The WY Representation for Products of Householder Matrices", SIAM SISSC 8, 2

[3] Demmell, Dongarra, Du Croz, Greenbaum, Hammarling, Sorensen, 1987, "A Prospectus for the Development of a Linear Algebra Library for High-Performance Computers", Technical Memorandum No. 97, Argonne National Laboratory, Argonne, Illinois.

[4] Dongarra, J. and Hewitt, T., 1985 "Implementing Dense Linear Algebra Algorithms Using Multitasking on the CRAY X-MP-4", Technical Memorandum No. 99, Argonne National Laboratory, Argonne, Illinois.

[5] Dongarra, Hammarling, Sorensen, 1987, "Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations", LAPACK Working Note #2, Technical Memorandum No. 99, Argonne National Laboratory, Argonne, Illinois.

[6] Dongarra, Du Croz, Duff, Hammarling, 1987, "A Proposal for a Set of Level 3 Basic Linear Algebra Subprograms", SIGNUM Newsletter, Vol. 22, No. 3, July, 1987

THE GRANULARITY OF PARALLEL HOMOTOPY ALGORITHMS FOR POLYNOMIAL SYSTEMS OF EQUATIONS

D. C. S. Allison, S. Harimoto, and L. T. Watson
Department of Computer Science
Virginia Polytechnic Institute & State University
Blacksburg, VA 24061

Abstract – Polynomial systems consist of n polynomial functions in n variables, with real or complex coefficients. Finding zeros of such systems is challenging because there may be a large number of solutions, and Newton-type methods can rarely be guaranteed to find the complete set of solutions. There are homotopy algorithms for polynomial systems of equations that are globally convergent from an arbitrary starting point with probability one, are guaranteed to find all the solutions, and are robust, accurate, and reasonably efficient. There is inherent parallelism at several levels in these algorithms. Several parallel homotopy algorithms with different granularities are studied on several different parallel machines, using actual industrial problems from chemical engineering and solid modeling.

1. Introduction.

Solving nonlinear systems of equations is a central problem in numerical analysis, with enormous significance for science and engineering. A very special case, namely small polynomial systems of equations, occurs frequently enough in solid modeling, robotics, computer vision, chemical equilibrium computations, chemical process design, mechanical engineering, and other areas to justify special algorithms. To put polynomial systems in perspective and for the purpose of this discussion, there are three classes of nonlinear systems of equations: (1) large systems with sparse Jacobian matrices, (2) small transcendental (nonpolynomial) systems with dense Jacobian matrices, and (3) small polynomial systems with dense Jacobian matrices. Sparsity for small problems is not significant, and large systems with dense Jacobian matrices are intractable, so these two classes are not counted.

Large sparse nonlinear systems of equations, such as equilibrium equations in structural mechanics, have two characterizing aspects: highly nonlinear and recursive scalar computations, and large matrix, vector operations. There is a great amount of parallelism in both aspects, but the nature of the parallelism is very different (or so it seems). Small dense transcendental systems of equations pose a major challenge, since they involve recursive, scalar intensive computation with a small amount of linear algebra. Finally, polynomial systems are unique in that they have many solutions, of which several may be physically meaningful, and there exist homotopy algorithms guaranteed to find all these meaningful solutions. The very special nature of polynomial systems and the power of homotopy algorithms are often not fully appreciated, perhaps because globally convergent probability-one homotopy methods are not widely known.

These globally convergent homotopy algorithms for polynomial systems have inherent parallelism at several levels. The purpose of the present paper is to study different granularities of parallel homotopy algorithms for polynomial systems, corresponding to different decomposition and communication strategies.

Much work has been done on solving linear systems of equations on parallel computers, mostly on vector machines [4], [5], [7], [8], [10]–[12], [14]–[16], [18], [23], [25]. Some work has been done on nonlinear equations and Newton's method [28], [31], [36], [37], and on finding the roots of a single polynomial equation [9], [27]. Parallel algorithms for polynomial systems were proposed by Morgan and Watson [22], but have not been studied much, nor have parallel homotopy algorithms for nonlinear systems of equations.

Section 2 summarizes the mathematics behind the homotopy algorithm, Section 3 discusses the special case of polynomial systems in some detail, and computational results on several parallel machines are presented and discussed in Section 4.

2. Homotopy algorithm.

Let E^n denote n-dimensional real Euclidean space. The fundamental mathematical result behind the homotopy algorithm for solving the nonlinear system of equations

$$F(x) = 0, \tag{1}$$

where $F: E^n \to E^n$ is a C^2 (twice continuously differentiable) function, is as follows:

Proposition 1 ([6], [33]). Let $\rho: E^m \times [0,1) \times E^n \to E^n$ be a C^2 map and define $\rho_a(\lambda, x) = \rho(a, \lambda, x)$. Suppose that

1. the $n \times (m+n+1)$ Jacobian matrix $D\rho$ has full rank on $\rho^{-1}(0)$, the set of zeros of ρ;
2. $\rho_a(0, x) = 0$ has a unique solution W of E^n (depending on the homotopy parameter vector a of E^m);
3. $\rho_a(1, x) = F(x)$;
4. the set of zeros of $\rho_a(\lambda, x)$ is bounded.

Then for almost all a of E^m there is a zero curve γ of $\rho_a(\lambda, x)$ along which the Jacobian matrix $D\rho_a(\lambda, x)$ has full rank, emanating from $(0, W)$ and reaching a zero $\bar{x}$ of $F(x)$ at $\lambda = 1$. Furthermore, γ has finite arc length if $DF(\bar{x})$ is nonsingular.

The general idea of the algorithm is to follow the zero curve γ of ρ_a from $(0, W)$ until a zero $\bar{x}$ of $F(x)$ is reached at $\lambda = 1$. Although the homotopy algorithm for solving the nonlinear system of equations is conceptually simple, it is nontrivial to develop a viable numerical algorithm for tracking the curve. A typical form for the homotopy map is

$$\rho_W(\lambda, x) = \lambda F(x) + (1-\lambda)(x - W) = 0, \tag{2}$$

which has the same form as a standard continuation or embedding mapping. However, two crucial differences exist. In standard continuation methods, the embedding parameter λ increases monotonically from 0 to 1 as the trivial problem $x - W = 0$ is continuously deformed to the problem $F(x) = 0$. Homotopy algorithms permit λ to both increase and decrease along γ with no adverse effect. The second difference is that in homotopy algorithms there are no "singular points" which afflict standard continuation methods. This is guaranteed by the way in which the zero curve γ of ρ_a is followed and the fact that $D\rho_a$ has full rank along γ .

The zero curve γ of the homotopy map $\rho_a(\lambda, x)$ can be tracked by many different techniques. The present work used HOMPACK [34], a software package for solving systems of nonlinear equations based on the homotopy method. HOMPACK provides three approaches for tracking γ : 1) an ODE-based algorithm with some special refinements for the homotopy context; 2) a predictor-corrector algorithm whose corrector follows the flow normal to the Davidenko flow (a "normal flow" algorithm); and 3) a version of Rheinboldt's linear predictor, quasi-Newton corrector algorithm (an "augmented Jacobian matrix" method). Since the "normal flow" algorithm is the technique used by HOMPACK to solve polynomial systems of equations, the other two algorithms will not be discussed in this paper.

The normal flow algorithm has three phases: prediction, correction, and step size estimation. In the prediction phase, the next point on the zero curve is predicted. Starting from the predicted point, the correction phase then iterates until a point on the zero curve is reached. An "optimal" step size is then estimated for the prediction of the next point on the curve.

The normal flow algorithm is so called because the iterates of the correction phase converge from the predicted point back to the zero curve along the flow normal to the Davidenko flow. The Davidenko flow is the family of zero curves formed from varying the parameter vector a of the homotopy map ρ_a.

The zero curve γ of $\rho_a(\lambda, x)$ is C^1 and can be parametrized by the arc length s. Thus $\lambda = \lambda(s)$ and $x = x(s)$ with initial conditions $\lambda(0) = 0$ and $x(0) = W$. When $\lambda(\bar{s}) = 1$, the corresponding $x(\bar{s}) = \bar{x}$ is a zero of $F(x)$.

Given that the following are available from previous calculations : two previous points on the curve, $P(s_1) = (\lambda(s_1), x(s_1))$ and

$P(s_2) = (\lambda(s_2), x(s_2))$, their corresponding tangent vectors, $P'(s_1) = (d\lambda/ds(s_1), dx/ds(s_1))$ and $P'(s_2) = (d\lambda/ds(s_2), dx/ds(s_2))$, and h, an estimate of the next "optimal" step size (in arc length) to take along γ , the next point on the zero curve can be estimated by

$$Z^{(0)} = H(s_2 + h), \tag{3}$$

where $H(s)$ is the Hermite cubic polynomial which interpolates $P(s)$ at s_1 and s_2. Thus, $H(s_1) = P(s_1)$, $H'(s_1) = P'(s_1)$, $H(s_2) = P(s_2)$, and $H'(s_2) = P'(s_2)$.

Since

$$\rho_a(\lambda(s), x(s)) = 0 \tag{4}$$

on the zero curve γ ,

$$\frac{d}{ds}\left[\rho_a(\lambda(s), x(s))\right] = D\rho_a(\lambda(s), x(s))(d\lambda/ds, dx/ds)^T = 0, \tag{5}$$

where $(d\lambda/ds, dx/ds)$ is the tangent vector to the curve. Thus, the tangent vector can be calculated by finding the kernel of the Jacobian matrix $D\rho_a(\lambda(s), x(s))$, which has rank n by Proposition 1. Once the kernel is found, the derivative $(d\lambda/ds, dx/ds)$ at a given point on the zero curve can be uniquely determined by

$$||(d\lambda/ds, dx/ds)||_2 = 1 \tag{6}$$

and continuity of the tangent vector.

Starting at the predicted point $Z^{(0)}$, the iteration in the correction phase is

$$Z^{(k+1)} = Z^{(k)} - \left[D\rho_a(Z^{(k)})\right]^{\dagger} \rho_a(Z^{(k)}), \qquad k = 0, 1, \ldots \tag{7}$$

where $\left[D\rho_a(Z^{(k)})\right]^{\dagger}$ is the Moore-Penrose pseudoinverse of $D\rho_a$. Reordering the equation, the corrector step $\Delta Z = Z^{(k+1)} - Z^{(k)}$ is the unique minimum norm solution of

$$\left[D\rho_a(Z^{(k)})\right]\Delta Z = -\rho_a(Z^{(k)}). \tag{8}$$

Fortunately ΔZ can be calculated at the same time as the kernel of $\left[D\rho_a\right]$ with just a little more effort. Normally for dense problems the kernel of $\left[D\rho_a\right]$ is found by computing the QR factorization of $\left[D\rho_a\right]$, followed by back substitution. By applying this QR factorization to $-\rho_a$ and using back substitution again, a particular solution v to (8) can be found. Let u be any non-zero vector in the kernel of $\left[D\rho_a\right]$. Then the minimum norm solution is

$$\Delta Z = v - \frac{v^t u}{u^t u} u. \tag{9}$$

Since the QR factorizations of $\left[D\rho_a\right]$ are computationally very expensive, the number of iterations required for convergence of (7) should be kept small (say <= 4) by adjusting the step size. An alternative is to use the QR factorization of $D\rho_a$ at the first predicted point $Z^{(0)}$ for several iterations. However, this results in linear convergence, which is not cost effective when compared to the asymptotically quadratic convergence of (7).

Note that the kernel of $\left[D\rho_a\right]$ is needed for the tangent vector used in the Hermite cubic interpolation at the beginning of the next step. When the iteration converges, the final iterate $Z^{(k+1)}$ is accepted as the next point on γ . Rather than calculating the tangent vector at the new point $Z^{(k+1)}$ on γ , a Jacobian matrix evaluation and a QR factorization can be saved by using the tangent vector calculated at $Z^{(k)}$. This substitution should not seriously affect the calculation of the next point on the curve since this tangent is used only in the prediction of the next point, and the tangent vector at $Z^{(k)}$ is a good estimate of the tangent vector at $Z^{(k+1)}$.

The estimation of an "optimal" step size h is an attempt to balance the number of iterations in the correction phase of the algorithm with the number of steps necessary to reach the "end" point on the zero curve where $\lambda(\bar{s}) = 1$. Increasing the step size decreases the number of steps necessary to reach the "end" of the curve. However, taking too large a step size would result in a substantial increase in the number of iterations necessary to correct the predicted point. The estimation and control of the step size h is discussed in detail in [38].

3. Polynomial systems.

Section 2 described a homotopy algorithm for finding a single solution to a general nonlinear system of equations $F(x) = 0$. Proposition 1 provided the theoretical guarantee of convergence. The rich structure and multiple solutions of polynomial systems dictate that the general theory in Section 2 must be sharpened. This section develops a globally convergent (with probability one) homotopy algorithm that finds *all* solutions to a polynomial system, and provides the theoretical justification for that algorithm.

Suppose that the components of the nonlinear function $F(x)$ have the form

$$F_i(x) = \sum_{k=1}^{n_i} a_{ik} \prod_{j=1}^{n} x_j^{d_{ijk}}, \quad i = 1, \ldots, n. \tag{10}$$

The ith component $F_i(x)$ has n_i terms, the a_{ik} are the (real) coefficients, and the degrees d_{ijk} are nonnegative integers. The total degree of F_i is $d_i = \max_k \sum_{j=1}^{n} d_{ijk}$. For technical reasons it is necessary to consider $F(x)$ as a map $F : C^n \to C^n$, where C^n is n-dimensional complex Euclidean space. A system of n polynomial equations in n unknowns, $F(x) = 0$, may have many solutions. It is possible to define a homotopy so that all geometrically isolated roots of (10) have at least one associated homotopy path. Generally, (10) will have roots at infinity, which forces some of the homotopy paths to diverge to infinity as λ approaches 1. However, (10) can be transformed into a new system which, under reasonable hypotheses, can be proven to have no roots at infinity and thus bounded homotopy paths. Because scaling can be critical to the success of the method, a general scaling algorithm [34] is applied to scale the coefficients and variables in (10) before anything else is done.

Since the homotopy map defined below is complex analytic, the homotopy parameter λ is monotonically increasing as a function of arc length [21]. Define $G : C^n \to C^n$ by $G_j(x) = b_j x_j^{d_j} - a_j, \quad j = 1, \ldots, n$, where a_j and b_j are nonzero complex numbers and d_j is the (total) degree of $F_j(x)$, for $j = 1, \ldots, n$. Define the homotopy map

$$\rho_c(\lambda, x) = (1 - \lambda)\, G(x) + \lambda\, F(x), \tag{11}$$

where $c = (a, b)$, $a = (a_1, \ldots, a_n) \in C^n$ and $b = (b_1, \ldots, b_n) \in C^n$. Let $d = d_1 \cdots d_n$ be the *total degree* of the system. The fundamental homotopy result, proved and discussed at length in [19]–[21], is:

Theorem. For almost all choices of a and b in C^n, $\rho_c^{-1}(0)$ consists of d smooth paths emanating from $\{0\} \times C^n$, which either diverge to infinity as λ approaches 1 or converge to solutions to $F(x) = 0$ as λ approaches 1. Each geometrically isolated solution of $F(x) = 0$ has a path converging to it.

A number of distinct homotopies have been proposed for solving polynomial systems. The homotopy map in (11) is from [20]. As with all such homotopies, there will be paths diverging to infinity if $F(x) = 0$ has solutions at infinity. These divergent paths are (at least) a nuisance, since they require arbitrary stopping criteria. Solutions at infinity can be avoided via the following projective transformation.

Define $F'(y)$ to be the homogenization of $F(x)$:

$$F'_j(y) = {y_{n+1}}^{d_j} F_j(y_1/y_{n+1}, \ldots, y_n/y_{n+1}), \qquad j = 1, \ldots, n. \tag{12}$$

The set of all lines through the origin in C^{n+1} is called complex projective n-space, denoted CP^n, and is a smooth compact (complex) n-dimensional manifold. The solutions of $F'(y) = 0$ in CP^n are identified with the (finite) solutions and solutions at infinity of $F(x) = 0$ in the usual way [38]. A basic result on the structure of the solution set of a polynomial system is the following classical theorem of

Bezout [21]:

Theorem. There are no more than d isolated solutions to $F'(y) = 0$ in CP^n. If $F'(y) = 0$ has only a finite number of solutions in CP^n, it has exactly d solutions, counting multiplicities.

Recall that a solution is *isolated* if there is a neighborhood containing that solution and no other solution. The multiplicity of an isolated solution is defined to be the number of solutions that appear in the isolating neighborhood under an arbitrarily small random perturbation of the system coefficients. If the solution is nonsingular (i.e., the system Jacobian matrix is nonsingular at the solution), then it has multiplicity one. Otherwise it has multiplicity greater than one.

Define a linear function $u(y_1, \ldots, y_{n+1}) = \xi_1 y_1 + \xi_2 y_2 + \cdots + \xi_{n+1} y_{n+1}$ where $\xi_1, \ldots, \xi_{n+1}$ are nonzero complex numbers, and define $F'' : C^{n+1} \to C^{n+1}$ by

$$\begin{aligned} F''_j(y) &= F'_j(y), \qquad j = 1, \ldots, n, \\ F''_{n+1}(y) &= u(y) - 1. \end{aligned} \tag{13}$$

So $F''(y) = 0$ is a system of $n+1$ equations in $n+1$ unknowns, referred to as *the projective transformation of* $F(x) = 0$. Since $u(y)$ is linear, it is easy in practice to replace $F''(y) = 0$ by an equivalent system of n equations in n unknowns. The significance of $F''(y)$ is given by

Theorem[19]. If $F'(y) = 0$ has only a finite number of solutions in CP^n, then $F''(y) = 0$ has exactly d solutions (counting multiplicities) in C^{n+1} and no solutions at infinity, for almost all $\xi \in C^{n+1}$.

Under the hypothesis of the theorem, all the solutions of $F'(y) = 0$ can be obtained as lines through the solutions to $F''(y) = 0$. Thus all the solutions to $F(x) = 0$ can be obtained easily from the solutions to $F''(y) = 0$, which lie on bounded homotopy paths (since $F''(y) = 0$ has no solutions at infinity).

The import of the above theory is that the nature of the zero curves of the projective transformation $F''(y)$ of $F(x)$ is as follows: There are exactly d (the total degree of F) zero curves, which are monotone in λ and have finite arc length. The homotopy algorithm is to track these d curves, which contain all isolated (transformed) zeros of F.

4. Computational results.

There are two extreme approaches for parallelizing the homotopy algorithm. For the coarsest form of parallelism, each individual processor tracks as many solutions (paths) as possible until all the solutions for the system of polynomial equations are found. In the other extreme, where the granularity is the finest, the primary task of tracking the solutions is delegated to one of the processors and only during polynomial system evaluations, Jacobian matrix evaluations, and other numerical calculations is the work distributed among the processors.

In the first case, the division of work is at the highest level. Initially, the parameters defining the system of polynomial equations ($F(x) = 0$) are distributed to the processors. Each processor then works independently of the other processors. When a processor completes the tracking of a path, it takes the next path to be tracked. Since there is no knowledge about the paths, they are assigned on a first-come-first-serve basis. Thus, if a "bad" path exists (one which requires a large number of function evaluations with respect to the remaining paths to be tracked), the load would not be distributed evenly among the processors. The processor with the "bad" path would cause the other processors to remain idle after all the remaining paths have been tracked.

The second approach is an attempt to balance the load more evenly, hopefully, resulting in an overall speedup over the coarser grained algorithm. In this paper, the fine-grained parallel homotopy algorithm distributes the work of evaluating the system of polynomial equations and its partial derivatives to N processors, where N is the number of equations or the maximum number of processors, whichever is smaller.

Two parallel versions, a coarse-grained version and a fine-grained version, of the homotopy algorithm for polynomial systems were developed from the parallelization of HOMPACK. They were executed on several parallel machines: Balance 21000, Elxsi 6400, Alliant FX/8, and the Intel iPSC-32. The execution times are shown in Table 1 of [38] along with those for the execution of the serial algorithm. The efficiencies are listed in Table 2 of [38]. The total degree of a problem refers to the number of solutions in the system of polynomial equations. Thus, if a machine has infinitely many processors, the maximum number of processors which can be utilized by the coarse-grained algorithm is determined by the total degree of the problem.

This paper is primarily concerned with the results obtained on bus-oriented, shared-memory parallel machines. The primary difference between implementations of parallel homotopy algorithms on shared-memory machines and implementations on distributed-memory machines is that, in distributed-memory machines, the master processor must communicate the tasks to the slave processors individually and wait for the results through some form of communication medium. After a set of results is received from one of the slave processors, the master processor assigns the next task to that free processor. Both the master processor and the slave processors must handle the processing of the communication protocols.

In shared-memory machines, the task of communicating the problem and obtaining the results is much simpler. It is handled through shared-memory, which is accessed by all the processors. The primary processor sets up the problem in the shared-memory, initiates the processors, and handles portions of the problem like the other spawned processors. Since the coordination of the processors is done through mutual exclusion of certain critical shared memory locations, no master processor is needed.

The calculations for the coarse-grained efficiencies are based on the maximum number of processors used on each of the parallel machines. The number of processors used on Sequent's Balance 21000, Elxsi's 6400, and Alliant's FX/8 are 8, 10, and 8, respectively, except for problems where the total number of paths to be tracked is less than the number of processors.

The efficiencies for the coarse-grained algorithm with various number of processors on Alliant's FX/8 are shown in Table 3 of [38]. As one would expect, the efficiency improves as the number of processors decreases.

5. Conclusions.

When the efficiencies of the two algorithms are compared using an equal number of processors, the coarse-grained algorithm is found to be more efficient than the fine-grained algorithm. In terms of speedup, the coarse-grained algorithm outperforms the fine-grained algorithm.

The problem with the coarse-grained algorithm is that when some solutions have long paths with respect to other solutions, the efficiency can be very low. This depends on the order in which the solutions are found and the total number of solutions with respect to the number of processors. Figure 1 demonstrates that by changing the order in which the solutions (of problem 602 in [38]) are found, the distribution of the work load can either be very unbalanced or very well balanced. In Figure 1, curve 1 shows that the work load can be distributed quite evenly among the processors when the "long" paths are assigned first. The distribution of the work load in the actual assignment of the paths is shown by curve 2. The worst distribution (curve 3) occurs when the "longest" path is assigned last while the rest of the paths are assigned in decreasing order in terms of the number of function evaluations. From these distributions, curve 1 has the "best" efficiency (0.99) and curve 3 has the "worst" efficiency (0.56), whereas the actual efficiency was 0.67.

In general, the coarse-grained parallel homotopy algorithm is more efficient and permits a higher degree of parallelism than the fine-grained algorithm. However, since the number of function evaluations

required for each path can not be predicted, the efficiency of the coarse-grained algorithm can be very low.

6. References.

[1] E. Allgower and K. Georg, "Simplicial and continuation methods for approximating fixed points," *SIAM Rev.*, 22 (1980), pp. 28–85.

[2] S. C. Billups, *An augmented Jacobian matrix algorithm for tracking homotopy zero curves*, M.S. Thesis, Dept. of Computer Sci., VPI & SU, Blacksburg, VA, (Sept., 1985).

[3] P. Businger and G. H. Golub, "Linear least squares solutions by Householder transformations," *Numer. Math.*, 7 (1965), pp. 269–276.

[4] A. C. Chen and C. L. Wu, "Optimum solution to dense linear systems of equations," *Proc. 1984 Internat. Conf. on Parallel Processing*, (August, 1984), pp. 417–425.

[5] M. Y. Chern and T. Murata, "Fast algorithm for concurrent LU decomposition and matrix inversion," *Proc. Internat. Conf. on Parallel Processing*, Computer Society Press, Los Alamitos, CA, (1983), pp. 79–86.

[6] S. N. Chow, J. Mallet-Paret, and J. A. Yorke, "Finding zeros of maps: Homotopy methods that are constructive with probability one," *Math. Comput.*, 32 (1978), pp. 887–899.

[7] E. Cloête and G. R. Joubert, "Direct methods for solving systems of linear equations on a parallel processor," *Proc. 8th South African Symp. on Numerical Mathematics*, Durban, South Africa, (July, 1982).

[8] M. Cosnard, Y. Robert, and D. Trystran, "Comparison of parallel diagonalization methods for solving dense linear systems," *Sessions of the French Acad. of Sci. on Math.*, (Nov., 1985), p. 781ff.

[9] G. H. Ellis and L. T. Watson, "A parallel algorithm for simple roots of polynomials," *Comput. Math. Appl.*, 10 (1984), pp. 107–121.

[10] D. D. Gajski, A. H. Sameh, and J. A. Wisniewski, "Iterative algorithms for tridiagonal matrices on a WSI-multiprocessor," *Proc. Internat. Conf. Parallel Processing*, Bellaire, MI, (Aug., 1982), pp. 82–89.

[11] W. Gentzsch and G. Schafer, "Solution of large linear systems on vector computers," *Parallel Computing 83*, North Holland, Amsterdam, (1984), pp. 159–166.

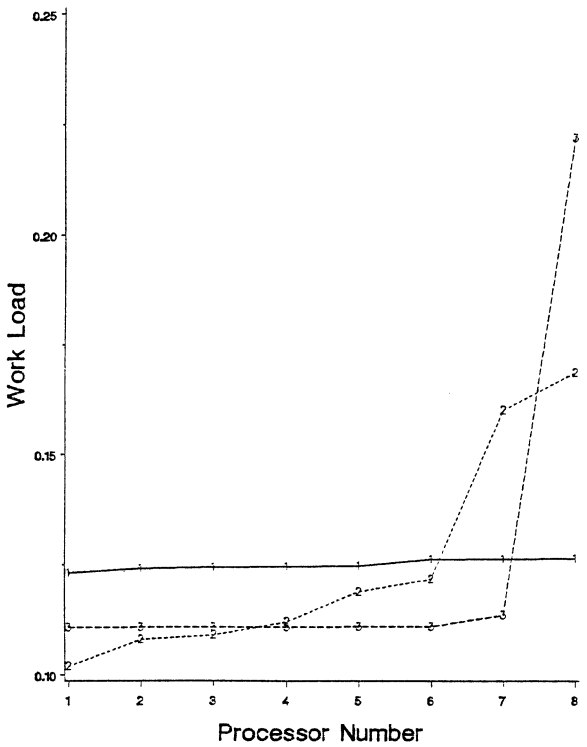


Figure 1. The distribution of work in coarse-grained parallelism for various orderings of paths for problem 602 on the Alliant FX/8.

[12] D. Heller, "A survey of parallel algorithms in numerical linear algebra," *SIAM Rev.*, 20 (1978), pp. 740–777.

[13] Intel Corporation, *iPSC Users' Manual*, Intel Corp., (1985).

[14] Y. Kaneda and M. Kohata, "Highly parallel computing of linear equations on the matrix-broadcast-memory connected array processor system," *10th IMACS World Congress*, Vols. 1–5, (1982), pp. 320–322.

[15] J. S. Kowalik, "Parallel computation of linear recurrences and tridiagonal equations," *Proc. IEEE 1982 Internat. Conf. on Cybernetics and Society*, (1982), pp. 580–584.

[16] J. S. Kowalik and S. P. Kumar, "An efficient parallel block conjugate gradient method for linear equations," *Proc. Internat. Conf. Parallel Processing*, Bellaire, MI, (Aug., 1982), pp. 47–52.

[17] M. Kubicek, "Dependence of solutions of nonlinear systems on a parameter," *ACM Trans. Math. Software*, 2 (1976), pp. 98–107.

[18] S. Lakshmivarahan and S. K. Dhall, "Parallel algorithms for solving certain classes of linear recurrences," *Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science*, Vol. 206, Springer-Verlag, Berlin, (1985), pp. 457–477.

[19] A. P. Morgan, "A transformation to avoid solutions at infinity for polynomial systems," *Appl. Math. Comput.*, 18 (1986), pp. 77–86.

[20] ——, "A homotopy for solving polynomial systems," *Appl. Math. Comput.*, 18 (1986), pp. 87–92.

[21] ——, *Solving polynomial systems using continuation for engineering and scientific problems*, Prentice-Hall, Englewood Cliffs, NJ, (1987).

[22] A. P. Morgan and L. T. Watson, *A globally convergent parallel algorithm for zeros of polynomial systems*, Dept. of Computer Sci., VPI & SU, Blacksburg, VA, TR 86-24, (Sept., 1986).

[23] D. Parkinson, "The solution of N linear equations using P processors," *Parallel Computing 83*, North Holland, Amsterdam, (1984), pp. 81–87.

[24] W. Pelz and L. T. Watson, *Message length effects for solving polynomial systems on a hypercube*, Dept. of Computer Sci., VPI & SU, Blacksburg, VA, TR 86-25, (Sept., 1986).

[25] D. A. Reed and M. L. Patrick, "A model of asynchronous iterative algorithms for solving large sparse linear systems," *Proc. 1984 Internat. Conf. on Parallel Processing*, (August, 1984), pp. 402–410.

[26] W. C. Rheinboldt and J. V. Burkardt, "Algorithm 596: A program for a locally parameterized continuation process," *ACM Trans. Math. Software*, 9 (1983), pp. 236-241.

[27] T. A. Rice and L. J. Siegel, "A parallel algorithm for finding the roots of a polynomial," *Proc. Internat. Conf. Parallel Processing*, Bellaire, MI, (Aug., 1982), pp. 57–61.

[28] H. Schwandt, "Newton-like interval methods for large nonlinear systems of equations on vector computers," *Computer Phys. Comm.*, 37 (1985), pp. 223–232.

[29] C. L. Seitz, "The cosmic cube," *Commun. ACM*, 28 (1985), pp. 22–33.

[30] L. F. Shampine and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: The Initial Value Problem*, W. H. Freeman, San Francisco, (1975).

[31] H. J. Sips, "A parallel processor for nonlinear recurrence systems," *Proc. 1st Internat. Conf. on Supercomputing Systems*, IEEE Computer Society Press, Los Alamitos, CA, (1984), pp. 660–671.

[32] L. T. Watson and D. Fenner, "Chow-Yorke algorithm for fixed points or zeros of C^2 maps," *ACM Trans. Math. Software*, 6 (1980), pp. 252–260.

[33] L. T. Watson, "A globally convergent algorithm for computing fixed points of C^2 maps," *Appl. Math. Comput.*, 5 (1979), pp. 297–311.

[34] L. T. Watson, S. C. Billups, and A. P. Morgan, *HOMPACK: A suite of codes for globally convergent homotopy algorithms*, Dept. of Industrial and Operations Eng., Univ. of Michigan, Ann Arbor, MI, Tech. Rep. 85–34, (Nov., 1985), and ACM Trans. Math. Software, 13 (1987), pp. 281–310.

[35] L. T. Watson, *Numerical linear algebra aspects of globally convergent homotopy methods*, Dept. of Computer Sci., VPI&SU, Blacksburg, VA, Tech. Report TR–85–14, (1985), and SIAM Rev., 28 (1986), pp. 529–545.

[36] R. White, "Parallel Algorithms for Nonlinear Problems," *SIAM J. Algebraic Discrete Methods*, 7 (1986), pp. 137–149.

[37] R. White, "A Nonlinear Parallel Algorithm with Application to the Stefan Problem," *SIAM J. Numer. Anal.*, 23 (1986), pp. 639–652.

[38] D. C. S. Allison, S. Harimoto, and L. T. Watson, *The granularity of parallel homotopy algorithms for polynomial systems of equations*, Dept. of Computer Sci., VPI & SU, Blacksburg, VA, TR 88–4, (Jan., 1988), 17 pp.

A New VLSI 2-D Systolic Array for Matrix Multiplication and Its Applications

Shie-Tung Peng
Moon S. Jun
Department of Computer Science
University of Maryland, Baltimore County

Abstract This paper proposes a new systolic architecture and a systolic algorithm for fast matrix multiplication. The computation time is $\frac{5}{2}m$ - 1 on the systolic array of m^2 processing elements. No any other known systolic algorithms can reach this time complexity by using a systolic array of m^2 processing elements.

To demonstrate the power of these methods, we apply them to solve the shortest path problem using a partition approach. The proposed block algorithm for the shortest path problem overcomes difficulties in the management of a large-size graph.

1. Introduction

With the advance of VLSI technology, a new technological environment is now available for manufacturing computers with high computation speed. It is now possible to implement a logic circuit consisting of hundreds of thousands of components and to build a large scale computing network with many inexpensive processing elements to perform parallel computation.

Among the several approaches to the parallel organization that can take advantage of the new technologies, the systolic arrays are of great potential to achieve high concurrency and parallel computation. Systolic arrays[H.T.Kung80, 82, S.Y.Kung 82, 87] have been developed to be efficient architectures for the solution of regular, computationally intensive peoblems.

A systolic array is an array of individual processing elements each of which is locally connected with its nearest neighbours to perform the same basic operations and to distribute the signal and the data across the entire processor array in a highly parallel and pipelined fashion. The simplicity of the processors and the uniformity of the processor interconnection allow the large systolic arrays to be implemented effectively on VLSI chips.

Matrix multiplication is one of the most important operations in diverse fields of computer sciences such as signal processing, image processing, graph theory and linear algebra.

In this paper, we propose a new systolic array and a systolic algorithm for computing matrix multiplication.

2. An Efficient VLSI 2-D Systolic Array

2.1 Description of Systolic Array Architecture

The proposed systolic array contains m^2 processing elements(PEs). Each PE can perform basic arithmetic and logic operations. There are two communication lines for each PE. The diagonal line supports two-way routing, while the vertical line supports only top-down transmission. The diagonal lines of boundary PEs are connected to the boundary PEs of the next row/column on the opposite side. There are serveral input/output buffers connected to the boundary PEs. More specifically, we have five m-by-1 array buffers, named $\mathrm{t}M^a$, $\mathrm{t}M^w$, $\mathrm{f}M^b$, $\mathrm{r}M^b$, and $\mathrm{b}M^w$. Each of the first four array buffers holds a row of the input matrices, the bottom array buffer, $\mathrm{b}M^w$, will hold the output. The connections among PEs and between array buffers and *boundary* PEs are depicted in Figure 1.

The following rules specify formally the interconnection of PEs to PEs and PEs to array buffers or vice versa.

Interconnection Rules for Architecture Design

(1) PEs to PEs

$$R_1(PE_{i,j}) = PE_{((i+1)mod(m),j)}$$
$$R_2(PE_{i,j}) = PE_{((i+1)mod(m),(j+1)mod(m))}$$
$$R_3(PE_{i,j}) = PE_{((i-1)mod(m),(j-1)mod(m))}$$

(2) PEs to array buffers and vice versa

$$\mathrm{R}(M_i^a) = \mathrm{R}(\mathrm{t}M_i^w) = PE_{1,i}$$
$$\mathrm{R}(\mathrm{f}M_i^b) = PE_{(i+1)mod(m),1)}$$
$$\mathrm{R}(\mathrm{r}M_i^b) = PE_{i,m}$$
$$\mathrm{R}(\mathrm{b}M_i^w) = PE_{m,(i+1)mod(m)}$$

2.2 A New Systolic Array Algorithm

A new systolic algorithm for computing the product matrix W of two m-by-m matrices A and B is described below. The product matrix W is defined by the formula $w_{xz} = \sum_{y=1}^{m} a_{xy} * b_{yz}$, where $w_{xy} \epsilon$ W, $a_{xy} \epsilon$ A, and $b_{yz} \epsilon$ B. The following notations will be used to describe the algorithm :

(1) $w_{xz}^0 = \begin{cases} 0 & \text{if initial} \\ previous\ result & \text{otherwise} \end{cases}$

(2) $w_{xz}^h = w_{xz}^{h-1} + (a_{xy} * b_{yz})$

(3) $w_{xz} = w_{xz}^h$

We assume that the input matrices in the memory modules can be accessed one row at a time. The ordering of rows of each input matrix that are loaded from memory modules are specified as follows: Matrices A and W are loaded from the low-numbered row to the high-numbered row starting with loading the first rows of A and W to $\mathrm{t}M^a$ and $\mathrm{t}M^w$ respectively. Matrix B is loaded in a slightly different way, two rows of B are loaded simultaneously, starting with loading the $(m/2)$th and the $(m/2 + 1)$th rows to $\mathrm{f}M^b$ and $\mathrm{r}M^b$ respectively, then the $(m/2 - 1)$th and the $(m/2 + 2)$th rows at the next computing cycle and so on until the first and the last rows are reached.

At the first step of the algorithm, elements of matrix B are piped from the column array buffers, $\mathrm{r}M^b$ and $\mathrm{f}M^b$, into the 2-D systolic array as shown in Figure 2(a). Then the contents of $PE_{1,m}$, $PE_{2,m}$, ..., $PE_{m,m}$ are $b_{(m/2)+1,m}$, ..., $b_{(m/2)+1,2}$, $b_{(m/2)+1,1}$ respectively, and the contents of $PE_{1,1}$, $PE_{2,1}$, ..., $PE_{m,1}$ are $b_{m/2,1}$, $b_{m/2,m}$, ..., $m_{m/2,2}$ respectively. Within $m/2$ steps, all elements of matrix B can be piped into the 2-D systolic array as shown in Figure 2(b).

At the $((m/2)+1)$th step, the following computations are executed in $PE_{1,1}$, $PE_{1,2}$, ..., $PE_{1,m}$ respectively, where w_{11}, w_{12}, ..., w_{1m} and a_{11}, a_{12}, ..., a_{1m} are piped from upper buffers into first row of the array.

$$w_{11}^{(1)} = w_{11}^{(0)} + a_{11} * b_{11},$$
$$w_{12}^{(1)} = w_{12}^{(0)} + a_{12} * b_{22},$$
$$\vdots$$
$$w_{1m}^{(1)} = w_{1m}^{(0)} + a_{1m} * b_{mm}$$

Then the w_{xz} is piped into the *lower&right* neighbor PE, the b_{yz} is stored in each the RAM of each PE to be used repeatedly, and the a_{xy} is piped into the *lower* neighbor PE as shown in Figure 2(c). After m-1 more steps, all elements of matrix A and W will be piped into the systolic array.

During each computation cycle, all the PEs in the array perform the same multiply-and-add operation. Each PE takes the value from its *upper&left* neighbor and adds it to $a_{xy} * b_{yz}$, and then passes the new value to its *lower&right* neighbor for use at the next step. At each computation cycle, the data stream of a_{jk} is piped one row down and w_{xz} is piped one position *lower&right* while b_{yz} is waiting in the original PE. The result w_{xz} will be piped into the bottom array buffers during the last m steps of the algorithm from the bottom *boundary* PEs.

ALGORITHM 1(Matrix Multiplication Problem)

```
Procedure Multiply( var : A, B, W ; m-bym matrices ) ;
begin
  r ⟵ 1 ;
  while ( r ≤ m/2 ) do (* Assume that m is even *)
    begin (* for 1 ≤ i ≤ m do simutanously *)
      if j ≤ m/2 in PE_ij
            then shift b_yz, left
            else shift b_yz, right ;
      r ⟵ r + 1 ;
    end ;
  store b_yz ;(*store b_yz into the RAM of each PE*)
  move a_xy ;(*move a_xy from tM^a into PE_1,i*)
  move w_xz ;(*move w_xz from tM^w into PE_1,i*)
  repeat
    begin (* all PEs that have all their data available *)
       mult a_xy,b_yz,w_xz ;
       move a_xy ;(*into the lower neighbor PE*)
       move w_xz ;(*into the lower&right neighbor PE *)
    end ;
  until there are no more data entering the PE ;
end.
```

Next, we prove that this systolic algorithm 1 works correctly by the following lemma and theorem:

(Lemma 1) : After the first $m/2$ steps of Algorithm 1, $PE_{(i,j)}$ will hold $b_{j,(j-i+1)mod(m)}$, for all $1 \le i, j \le m$.

(Proof)

(i) $b_{i,j}$ of the front memory modules at the first step will be loaded into $PE_{m-j+2,1}$ for all $1 \le j \le m$. Then $b_{i,j}$ is moved *lower&right* during the rest (i-1) steps. Therefore $b_{i,j}$ will halt at $PE_{((m-j+2)+(i-1)))mod(m),1+(i-1)} = PE_{(i-j+1)mod(m),i}$.

(ii) $b_{i,j}$ of the rear memory modules at the first step will be loaded into $PE_{m-j+1,m}$ for all $1 \le j \le$ m. $b_{i,j}$ is moved *upper&eft* during the rest m-i steps. Therefore $b_{i,j}$ will halt at $PE_{(m-j+1-(m-i))mod(m),m-(m-i)} = PE_{(i-j+1)mod(m),i}$.
By (i) and (ii), $PE_{i,j}$ holds $b_{j,(j-i+1)mod(m)}$. □

(Theorem 1): Algorithm 1 generates the product matrix W of matrices A and B.

From the algorithm, we know the following facts:
(a) $w_{i,j}$ passes through m PEs from top down.
(b) The m PEs passed by $w_{i,j}$ are
$PE_{1,j}$, $PE_{2,(j+1)mod(m)}$, $PE_{3,(j+2)mod(m)}$,...., $PE_{m,(j+m-1)mod(m)}$.
(c) The A values of the PEs in (b) while $w_{i,j}$ passes are
$a_{i,j}$, $a_{i,(j+1)mod(m)}$, $a_{i,(j+2)mod(m)}$,...., $a_{i,(j+m-1)mod(m)}$.

By Lemma 1, the B values of the PEs in (b) are $b_{j,j}$, $b_{(j+1)mod(m),j}$, $b_{(j+2)mod(m),j}$,....,$b_{(j+m-1)mod(m),j}$. .....(d)
From (c) and (d), we know that the value of $w_{i,j}$ is

$$\sum_{k=0}^{m-1} a_{i,(j+k)mod(m)} b_{(j+k)mod(m),j}$$

Therefore the algorithm generates the product of matrices A and B.□

2.3 Performance Analysis and Comparison with Other Arrays

In the proposed algorithm 1, a PE, once activated, performs the multiply-and-add operation at every computation cycle, until there are no more data entering the PE. At the peak of computation, about 100 percent of the m^2 PEs compute simultaneously. Average utilization of PEs is about 50 percent. This rate is higher than in other systolic arrays.

The computation time is $\frac{5}{2}$m - 1 time units assuming that each computing-and-routing cycle takes one unit of time. This can be verified by the following facts:

(1) In $\frac{m}{2}$ time units, all elements of matrix B are piped into the array,
(2) The time difference between the bottom and the top row elements of either A or W is (m-1) units due to startup time delay as shown in Figure 2, and
(3) The bottom row elements of either A or W are piped out of the array after m computing cycles.

It can be seen easily that our systolic algorithm has better performance than all other systolic algorithms in term of AT^2 measure for VLSI implementation as shown Table 1. For more efficient VLSI implementation, the operation of each PE is controlled locally and handles asynchronously[Peng & Jun 87, 88]. The basic idea is that a PE does not have to wait until the previous PE completes its computation. The approach speeds up the computation time by allowing the individual PE to operate independently to reduce the waiting time. The architectural design of PE will be discussed in next section.

Table.1: Comparision of Systolic Array

Systolic Array	Time Complexity	Control System
[Guibas 79]	5m	synchronous
[H.T.Kung 80]	3m+min(p,q)	synchronous
[Rote 85]	5m-3	synchronous
[McCanny 86]	4m-1	synchronous
[S.Y.Kung 87]	4m-2	asynchronous
[this paper]	$\frac{5}{2}$m-1	asynchronous

2.4 Architectural Design of Processing Element

In this section we will discuss the organization of the PE to be used as a basic module in our 2-D systolic array. In order to minimize the impact of both internal and external processor communication, each PE is partitioned into an Arithmetic Unit, three network control I/O units which can operate asynchronously, an EPROM(Errasible Programmable Read Only Memory), a RAM(Random Access Memory), and a control unit which can control the execution of both the memory unit and the arithmetic unit. The organization of the PE is illustrated in Figure 3(a).

Each PE is connected to its three neighbors via indicated busses in the control unit. This device can be used to control its speed, internal RAM, I/O pins, and byte manipulation. The data in the PE move over a multiplexed n'-bits data/address bus.

Computation is accomplished in the Arithmetic Unit, see Figure 3(b). This unit is capable of performing several formats (fixed point or floating point formats). Many operations are available such as addition, subtraction, multiplication, division, square root, trigonometric functions, etc.

The EPROM is used to store algorithms which perform data manipulation. This gives the system an efficient implementation so that the system can operate asynchronously and with local control. The RAM can be used for data storage. This memory space is important for partitioned matrix operations.

The RAM can also be used for storage of programs during algorithm development. Once the algorithms have been completed, they will be put into EPROM.

The control unit is partitioned into eight units as shown in Figure 3(d). They control RAM, EPROM, AU, and I/O units in each PE and are used to control I/O communication with the neighbor PEs in the network and allow messages to be sent to or received from several PEs simultaneously. Each NCIOU can determine whether the data are needed by the AU and/or other PEs with no intervention from the AU. Figure 3(c) 6 shows the basic architecture of one NCIOU. This unit is divided into four functional units.

3. Applications for the new 2-D systolic Array

3.1 Mapping a Partitioned Graph into a Systolic Array

In this section, we describe a partition approach for large size directed graphs. The partition method is the key to extend the computational capability of VLSI architectures and to overcome difficulties in the mapping of a large-size graph into a fixed systolic array.

A directed graph G with n nodes can be represented as a pair G = (V,E), where V is the set of vertices and E is the set of weighted edges in the graph. As shown in Figure 4(a) and (b), we use a matrix *adjacency matrix* W to represent the directed graph with six nodes as follows.

(1) $w_{ij} \doteq$ the *weight* from node i to node j if there is an edge from i to j;

(2) $w_{ij} = \infty$ if there is no edge connecting i and j;

(3) $w_{ij} = 0$ if $i = j$.

Suppose that the size of the systolic array is m^2, the adjacency matrix W of size n^2 can be partitioned into k^2 submatrices each of size m^2, where $k = n/m$. We can do computation on these submatrices separately using the proposed systolic array of size m^2 and then combine the results to get the solution for the original large size problem.

The partition approach can avoid the unrealistic assumption that an unlimited number of PEs can be used to execute required computation. The size of the systolic system should be fixed once it is built.

3.2 Shortest Path Problem

In the all-pairs shortest path problem, we are required to produce a matrix W^+ of size n^2 such that w_{ij} is the weight of the shortest path from v_i to v_j in G.

We use Floyd's approach for this problem. Let w_{ij}^k denote the length of the shortest path from v_i to v_j that has intermediate vertices with indexes smaller or equal to k. Then $w_{i,j}^n$ will be the shortest path we need. Since there are no negative weight cycles in G, we can easily modify the multiplication algorithm 1 to find W^+ by replacing "+" and "*" in the multiplication algorithm 1 with " min " and "+" as follows :

$w_{xz}^{h+1} := \min\{\ w_{xz}^h, [\ w_{xy}^h + w_{yz}^h\ ]\ \}$

Following this modified version of algorithm 1 and the Floyd algorithm, the all-pairs shortest path problem can be solved here in $5/2n$ - 1 time on the 2-D systolic array with n^2 PEs. It is illustrated by partitioning an example of a graph of order 6 using a systolic array of size 2-by-2 as shown in Figure 5(a) and (b). Here the ratio k is $n/m = 3$.

The sequences of submatrix computations for the $i\ th$ iteration, where $1 \le i \le 3$, of this algorithm are shown in Figure 5(c). At the end of $i\ th$ iteration, w_{xy} is updated to be the shortest paths through intermediate nodes contained in submatrix W_{ii}. This is done by comparing w_{xy} with $w_{xi} + w_{iy}$. The above processing is done through three steps (1),(2), and (3) of the algorithm as depicted in figure 5(c). After k iterations, all w_{xy} will have the correct values of the shortest path from x to y.

The correctness of this algorithm follows easily from the Floyd algorithm. Now, we analyze the time complexity as below: For each i, the number of submatrix computations is 1 + 2(k - 1) + $(k-1)^2 = k^2$, and each submatrix computation takes 3m - 1 units of time. Therefore, the total computing time is (3m - 1)$*k^2*$k = $3n^3/m^2$ using a systolic array of size m^2. All the computations at steps (2) or (3) in the algorithm can be done independently. Therefore, the time complexity can be further improved if there are more systolic arrays of size m^2 available.

ALGORITHM 2 (Shortest Weight Path)

```
Procedure allweight (var W: adjacencymatrix; k: integer);
begin
    for i := 1 to k do
(1)     W_ii := min( W_ii, W_ii + W_ii) ;
(2)     for j := 1 to k with j ≠ i do
            W_ij := min( W_ij, W_ii + W_ij) ;
            W_ji := min( W_ji, W_ji + W_ii) ;
        end-for ;
(3)     for h := 1 to k with h ≠ i do
            for g := 1 to k with g ≠ i do
                W_hg := min( W_hg, W_hi + W_ig) ;
            end-for ;
        end-for ;
    end-for ;
end ;
```

4. Conclusion

In this paper, we describe a new systolic system for matrix multiplication and its relative problems. The proposed systolic array achieves better performance than the previously reported systolic arrays. Other advantages of our array include: (1) no need for global propagating control signals, (2) no need for data pre-arrangement, (3) no need for data interleaving to achieve this optimal rate, and (4) only two I/O lines required.

For applications, we mention only the shortest path problem for directed graphs. There should be many other applications that the proposed system can be used. Image processing and signal processing are two possible examples. More research needs to be done to explore the potential of the new systolic system.

REFERENCES

[Floyd 62] R.W. Floyd, *Algorithm 97: Shortest Path*, Comm. ACM 5, 1962, pp.345.

[Guibas 79] L.J.Guibas, H.T.Kung, and C.D.Thompson, *Direct VLSI Implementation of Combinatorial Algorithms*, in Proc. Caltech Conference on VLSI, L.A., 1979.

[Hwang 82] K.Hwang and Y.Cheng, *Partitioned Matrix Algorithm for VLSI Arithmetic Systems.*, IEEE Transc.on Comp.,vol c-31 No.12,December 1982,pp1215-1224

[H.T.Kung 80] H.T.Kung and C.E.Leiserson *Algorithms for VLSI Processor Arrays*, in C.Mead and L.Conway, *Introduction To VLSI System.* Reading, MA: Addison-Wesley, 1980, pp.263-332.

[H.T.Kung 82] H.T.Kung, *Why Systolic Architectures?*, Computer Magazine, Vol.15,No.1,Jan.1982,pp.37-46.

[S.Y.Kung 82] S.Y.Kung, *Wavefront Array Processor: Languages, Architecture, and Applications* ,IEEE Trans.on Computers,pp.1054-1066, Nov.1982.

[S.Y.Kung 87] S.Y.Kung, S.C.Chung, and P.S.Lewis, *Optimal System Design for the Transitive Closure and the Shortest Path Problems*, IEEE Tranc. Comp., Vol.C-36, No.5, May 1987. pp.603-614.

[McCanny 86] John McCanny and John McWhirter, *The Derivation and Utilization of Bit Level Systolic Array Architectures* , "Systolic Array" in the First Int'l workshop on systolic Arrays, Oxford,2-4 July 1986, pp 47-59.

[Moldovan 83] D.I.Moldovan, *On the Design of Algorithms for VLSI Systolic Arrays*, Proc. IEEE,Vol.71,No.1,Jan.1983

[Peng & Jun 87] Shie-Tung Peng and Moon S. Jun, *A New VLSI 2-D Systolic Array for Matrix Multiplication and Its Application* , Univ. of Maryland, Baltimore Conuty, TR CS-PP-001, 1987.

[Peng & Jun 88] Shie-Tung Peng and Moon S. Jun, *A Local and Asynchronous Control for Advanced Systolic Arrays*, Univ. of Maryland, Baltimore County, TR CS-PP-003, 1988.

[Rote 85] G.Rote, *A Systolic Array Algorithm for the Algebraic Path Problem(Shortest Paths, Matrix Inversion)*, Computing, no.34, Sprin-Verleg, 1985, pp191-219.

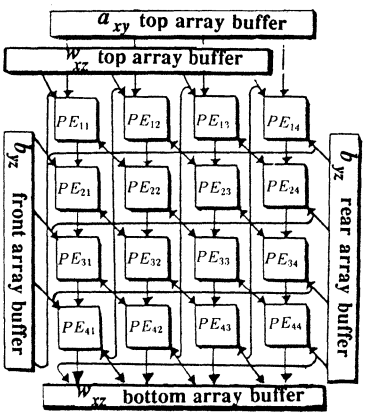


Figure.1. The connection between the *boundary* PEs and the array buffers.

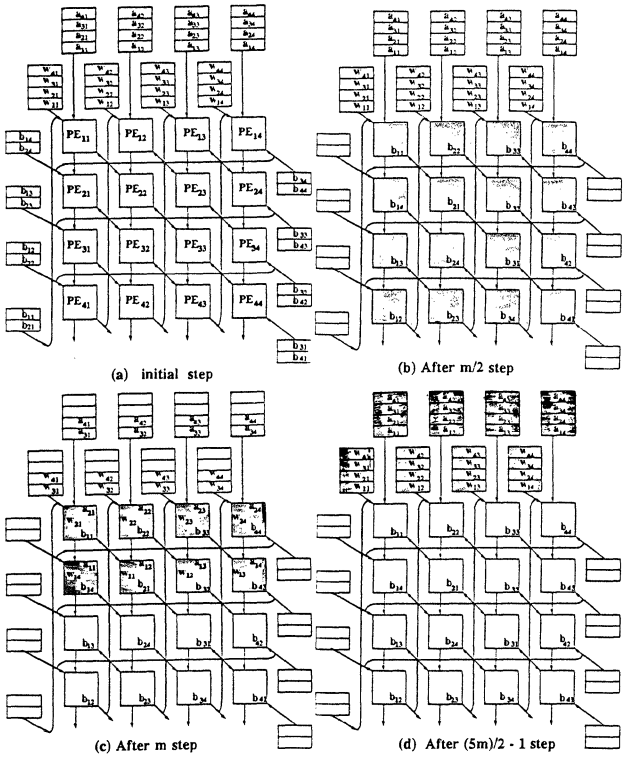
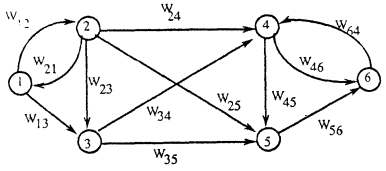


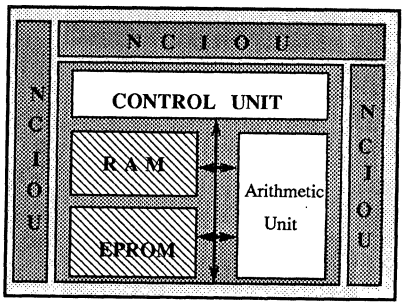
Figure.2. A new systolic matrix multiplication



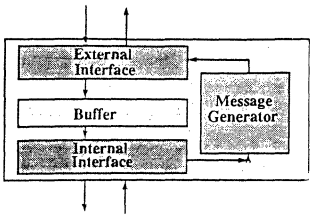
(a) A directed graph with weight.

$$W = \begin{pmatrix} 0 & w_{12} & w_{13} & \infty & \infty & \infty \\ w_{21} & 0 & w_{23} & w_{24} & w_{25} & \infty \\ \infty & \infty & 0 & w_{34} & \infty & \infty \\ \infty & \infty & \infty & 0 & w_{45} & w_{46} \\ \infty & \infty & \infty & \infty & 0 & w_{56} \\ \infty & \infty & \infty & w_{64} & \infty & 0 \end{pmatrix}$$

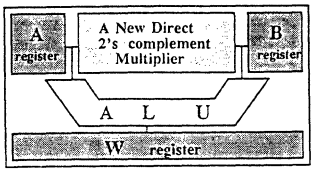
Figure.4. (b) The adjacency matrix for the directed graph.



(a) Processing Element(PE)



(b) Arithmetic Unit



(c) Network Control I/O Unit

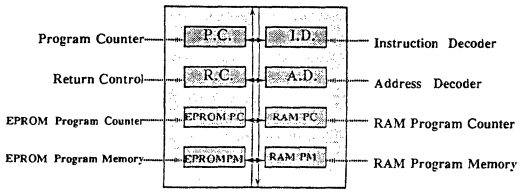
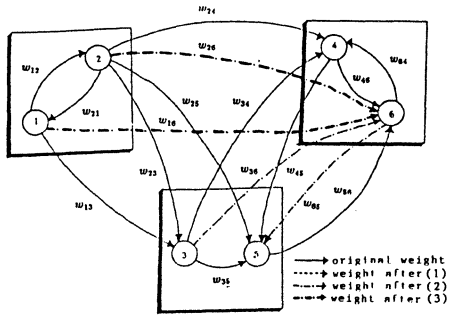


Figure.3. (d) A Control Unit



(a) The shortest path graph of Fig.3(a).

$$W^+ = \begin{pmatrix} 0 & w_{12} & w_{12} & w_{14} & w_{15} & w_{16} \\ w_{21} & 0 & w_{23} & w_{24} & w_{25} & w_{26} \\ \infty & \infty & 0 & w_{34} & w_{35} & w_{36} \\ \infty & \infty & \infty & 0 & w_{45} & w_{46} \\ \infty & \infty & \infty & w_{54} & 0 & w_{56} \\ \infty & \infty & \infty & w_{64} & w_{65} & 0 \end{pmatrix}$$

(b) The shortest path matrix W^+ of (a)

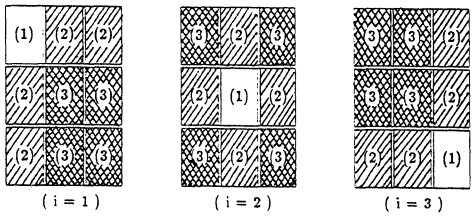


Figure.5. (c) The ordering of submatrix-operations in algorithm 2.

PARALLEL ALGORITHMS FOR MULTIPLYING VERY LARGE INTEGERS

Shie-Tung Peng (peng@umbc2.umd.edu) Thomas F. Hudson, jr.
Department of Computer Science
University of Maryland, Baltimore County
5401 Wilkens Avenue Catonsville, Maryland 21228

Abstract — Parallel algorithms for multiplying large integers are considered in the EREW PRAM model. A parallel addition algorithm is discussed, followed by a shift-and-add multiplication algorithm which runs in O(log n) time with $O(n^2)$ processors. Two FFT-based algorithms are discussed, one in the field of complex numbers, which runs in O(log n log log n) time, and in the ring of integers modulo $2^{\alpha}+1$ (Schönhage-Strassen), which runs in O(log n) time.

1. Introduction

Many problems in Applied Algebra and Number Theory require the exact product of two very large n-bit integers, where n is too large for an ordinary machine multiply instruction. We assume no bound on n. We consider only the multiplication of positive integers, since the product of negative integers can be computed from their positive counterparts with a simple test for final sign. We discuss four parallel algorithms – one for addition, one for shift-and-add multiplication, and two for multiplication using FFT's. Detailed presentations of the algorithms discussed herein can be found in [15].

1.1. Parallel Processing Model

There are many parallel processing models available [7, 9, 12, 16, 18, 20, 24], each of which has advantages and disadvantages. We work within the context of a *global shared memory model* (sometimes called the PRAM [24] model), where we have p processing elements (PE's) connected to a global memory. The PE's constitute an SIMD machine. Each PE is assumed to have local memory of size $O(\log n)$. We selected this model because it poses no architectural restrictions or delays on access to atomic data, and is consequently ideal for asymptotic time analysis.

Three modes of the global shared memory model have been identified by [20]. These affect how PE's may access the global memory. They are – CRCW (concurrent read, concurrent write), CREW (concurrent read, exclusive write), and EREW (exclusive read, exclusive write). The EREW mode is the most restrictive of the three [12]. We chose to use the EREW PRAM model, since CRCW or CREW could prevent us from easily transferring algorithms to a model or architecture which does not permit concurrent access.

2. Parallel Addition

We present an algorithm, based on ideas primarily from [11] and also from [7, 17, 18, 23], which adds two n-bit binary integers r and s, where $n = 2^N$. If we think of $r+s$ as a *modulo* 2^n addition with a carry into the n-th position, we can write this sum as two *modulo* $2^{n/2}$ sums of the upper and lower halves of r and s with carry into the n-th position and carry between these two sums (*i.e.* into the $(n/2)$-th position). Applying this recursively $\log n$ times, we reach the "bottom" where the sums are *modulo* 2 and we have carry (*modulo* 2) into each bit position. $r+s$ can be computed from the bit-wise exclusive-or of r_i, s_i, and these carries.

To compute the carries, we start at the "bottom" point and compute the carry out of the *modulo* 2 sums with a boolean operation. We then unwind the recursion to the "top" (*modulo* 2^n sum), generating "carry-out" data as we go. We then wind the recursion down again, distributing the carry-out information as "carry-in". This gives us the carry into each of the *modulo* 2 sums $r_i \oplus s_i$. The carry-in to the lowest sum is forced to be 0. This recursion requires $O(\log n)$ time with n PE's. We refer to this as ALGORITHM A.

This algorithm can compute $r-s$ by inverting s and forcing the lowest order carry-in to be 1.

BEGIN ALGORITHM A

1.0 Let r and s be two n-bit numbers where $n = 2^N, N > 0$. Let $c_{i,j}$, $f_{i,j}$, and $z_{i,j}$ represent the j-th value of c, f, and z at the i-th level of recursion. c, f, and z are used in accumulation and distribution of the carry information. The large right braces imply parallel computation, with the range indicated to the right. We compute the $(n+1)$-bit sum $t = r + s$ thus:

1.1 Initialize.

$$\left.\begin{aligned} c_{0,i} &\leftarrow r_i \wedge s_i \\ f_{0,i} &\leftarrow r_i \oplus s_i \end{aligned}\right\} 0 \le i < n$$

1.2 FOR $k \leftarrow 1$ TO N

$$\left.\begin{aligned} c_{k,i} &\leftarrow c_{k-1,2i+1} \vee (f_{k-1,2i+1} \wedge c_{k-1,2i}) \\ f_{k,i} &\leftarrow f_{k-1,2i+1} \wedge f_{k-1,2i} \end{aligned}\right\} 0 \le i < 2^{N-k}$$

ENDFOR

1.3 Compute

$$t_n \leftarrow c_{N,0}$$
$$z_{N,0} \leftarrow 0$$

1.4 FOR $k \leftarrow N$ DOWNTO 1

$$\left.\begin{aligned} z_{k-1,2i} &\leftarrow z_{k,i} \\ z_{k-1,2i+1} &\leftarrow c_{k-1,2i} \vee (f_{k-1,2i} \wedge z_{k,i}) \end{aligned}\right\} 0 \le i < 2^{N-k}$$

ENDFOR

1.5 Compute

$$t_i \leftarrow f_{0,i} \oplus z_{0,i} \Big\} 0 \le i < n$$

END ALGORITHM A

---

3. Multiplication by Direct Computation

The "natural" method of multiplication we are taught as children consists of multiplying the multiplicand by the individual digits of the multiplier, shifting each of these results to the left as the order of the multiplier digit increases, and summing the shifted results to form the final product. Binary multiplication is a special case of this method where the values summed are simply repeated copies of the multiplicand shifted to the left i places wherever the i-th bit of the multiplier is 1.

With $O(n^2)$ PE's, we can distribute copies of the multiplicand over an $O(n^2)$-sized memory in $O(\log n)$ time. We can also distribute the multiplier in $O(\log n)$ time, and logical-and the multiplier and multiplicand bits (to remove the undesired rows) in constant time. This reduces the problem to that of summing n rows of $(2n)$-bit numbers. If two rows can be added in constant time, we can sum them up in $O(\log n)$ time using a typical binary-tree summation. Clearly we cannot use ALGORITHM A for this, since summation would require $O(\log^2 n)$ time.

3.1. Carry-Save Addition

There is a valuable addition technique known as a Carry-Save Adder [17], which we abbreviate CSA. A CSA can add three n-bit numbers giving two $(n+1)$-bit numbers in time independent of n. Specifically, if a, b, c, and d are n-bit binary numbers, and e is an $(n+1)$-bit number, we can compute $d + e \leftarrow a + b + c$ by computing

$$d_i \leftarrow a_i \oplus b_i \oplus c_i \ ,$$
$$e_{i+1} \leftarrow (b_i \wedge c_i) \vee (a_i \wedge (b_i \oplus c_i)) \ ,$$
$$e_0 \leftarrow 0 \ .$$

So, if we have n rows to sum, we can reduce this to $2n/3$ rows in constant time with $O(n^2)$ PE's. Since each reduction is by a factor of 2/3, it takes $\log_{3/2} n = O(\log n)$ applications of this technique to reduce the number of rows to two, where they can be added in $O(\log n)$ time using ALGORITHM A. Therefore, we can multiply two numbers in $O(\log n)$ time with $O(n^2)$ PE's. We refer to this as ALGORITHM B.

4. FFT Techniques

Owing to the relationship between the Fourier Transform domain and convolution, the Fast Fourier Transform (FFT) can be used to multiply finite polynomials, and hence integers, very rapidly. The use of the FFT for multiplication of finite polynomials has been well studied. Presentations can be found in [1, 5, 10, 16, 19] among others. The FFT was apparently first applied to this problem by Strassen in 1968 [10].

For computation of exact convolution results, there are two common families of FFT algorithms. One family uses the ring of integers *modulo* $2^{\alpha}+1$ for some α (the Schönhage-Strassen algorithm) [1, 10, 19] and selects α large enough that the correct result is obtained. The other family uses the field of complex numbers with sufficient floating point precision to guarantee the result [10, 19]. The FFT in parallel has been discussed by [8, 16, 19, 21, 22] among others.

4.1. FFT's in the Complex Field

The FFT has its origins [5, 6, 14] in the field of complex numbers. The nth root of unity is $\omega = e^{(\sqrt{-1})2\pi/n}$ where the transform is over a vector of n values. Strassen's approach, as presented in [10, 19] works this way: Let u and v be n-bit numbers. Let

$$2n \le 2^k l < 4n \ , \quad K = 2^k \ , \quad L = 2^l \ .$$

We view u and v as K-place base-L integers, whose upper $K/2$ digits are 0. We perform k stage FFT's on u and v yielding their transforms $[\hat{u}_0, \hat{u}_1, \cdots, \hat{u}_{K-1}]$ and $[\hat{v}_0, \hat{v}_1, \cdots, \hat{v}_{K-1}]$, compute the piecewise products $[\hat{w}_0, \hat{w}_1, \cdots, \hat{w}_{K-1}]$ of the two transforms, and compute the inverse FFT $[W_0, W_1, \cdots, W_{K-1}]$ of those products.

Since the W_i's are the convolution of K-place base-L numbers, each W_i is upperbounded by KL^2 and we cannot simply concatenate these results to form the answer. The desired result can be obtained by computing $w = \sum_{i=0}^{K-1} W_i L^i$.

We now examine the algorithm in detail. Select k and l such that $k = l$. k and l are both slightly smaller than $\log n$. This makes for an awkward exact form for K, k, and l, but they are approximately $k = l = O(\log n)$ and $K = O(n/\log n)$.

All numbers are converted to binary fixed point fractions so that they will have a magnitude less than 1. Assuming we use m-bit precision, the initial error in any number is 2^{-m}. Each complex operation has one of the two forms $a \leftarrow b\,\omega + c$ or $a \leftarrow b\,\omega$. Either of these forms will cause the loss of two bits of accuracy from both the real and imaginary components.

We begin both forward FFT's by shifting the base-L integers to the right $(l+k)$ places. Shifting l places makes each number a fraction. Shifting k additional places ensures that none of the intermediate results exceed 1, since no butterfly can do more than double the intermediate values. Since the FFT is a linear operation [13], dividing the inputs by 2^{k+l} reduces the output values by the same factor. So the piecewise products have the form

$$\left[\frac{\hat{w}_{K-1}}{2^{2k+2l}}\right] = \left[\frac{\hat{u}_{K-1}}{2^{k+l}}\frac{\hat{v}_{K-1}}{2^{k+l}}\right].$$ The final output of the inverse FFT must be multiplied by 2^{2k+2l} to reconstruct the desired answer. Therefore the error must not propagate any higher than $2^{-2k-2l-1} = 2^{-4k-1}$. The k stages of the forward FFT's induce $2k$ bits of error, so the error after the forward FFT's is 2^{2k-m}. The complex multiplication of the $\hat{u}$'s and $\hat{v}$'s induces an additional two bits of error, raising the error to 2^{2k-m+2}. The inverse FFT induces another $2k$ bits of error, but the division by K reduces half of that, so after the inverse FFT the propagated error is 2^{3k-m+2}. Thus m must be at least $7k+3$ to limit the error to 2^{-4k-1}. Since $k < \log n$, we simplify and use the figure $7\log n$ as the number of bits required to obtain the correct result.

Addition and subtraction of $O(\log n)$-bit numbers can be done using ALGORITHM A in $O(\log\log n)$ time without any significant alteration. The scaling in the FFT algorithm prevents additions from overflowing into a number greater than 1. Multiplication can be done using ALGORITHM B in $O(\log\log n)$ time with the alteration that we shift rows to the right (for fractional accumulation) instead of to the left.

In each butterfly in each stage of each FFT, we must perform one complex multiplication and two complex additions. These can be done in $O(\log\log n)$ time. With $k = O(\log n)$ stages, an entire FFT requires $O(\log n \log\log n)$ time.

The multiplication of the piecewise products takes $O(\log\log n)$ time. The inverse FFT is the same as the forward FFT. The division by K and multiplication by 2^{2k+2l} at the end of the inverse FFT are simply shifts and are insignificant. Once the inverse FFT has been computed and properly shifted, we need to compute the final result $w = \sum_{i=0}^{K-1} W_i L^i$. Since each W_i is no more than $k+2l = 3\log L$ bits long, we can rearrange them into three numbers, and use one carry-save addition and one application of ALGORITHM A to compute the sum. The entire convolution requires $O(\log n \log\log n)$ time.

We now address the computation of the sinusoid table. Each entry in the table must be accurate to $7\log n$ bits. We compute the table by computing $\omega = e^{(\sqrt{-1})2\pi/K}$ using the conventional series for e^x, then computing the powers of ω from it. There are $K = O(n/\log n)$ powers of ω in the table. Each computation of a power of ω is a complex multiply which can be done in $O(\log\log n)$ time. $O(\log n)$ such computations are required, so the table can be computed and copied in $O(\log n \log\log n)$ time. Each ω_i we compute is the complex product of no more than k other ω_j values, so only $2k$ bits of error are induced. If we compute ω with $9\log n$ correct bits, we will have $7\log n$ correct bits in each power of ω. We can compute ω to this accuracy in $O(\log n \log\log n)$ time [2, 3, 4, 15].

This FFT algorithm multiplies two n-bit integers in $O(\log n \log\log n)$ time with $O(n \log n/\log\log n)$ PE's. We refer to this as ALGORITHM C.

4.2. FFT's in the Ring of Integers

The classic form of the Ring-of-Integers approach is the well-known Schönhage-Strassen algorithm. This algorithm is relatively complex, and [1, 19] give good presentations of it. The FFT is computed in the ring of integers modulo some number whose results will be smaller than the original operands. These results must then be multiplied together, which is done recursively with successively smaller and smaller FFT's until the numbers are small enough to multiply with some temporally faster technique.

Given two numbers u and v, we represent them as n-bit binary numbers where $n = 2^k$ and $u_i = v_i = 0, n/2 \le i < n$. Let $b = 2^{\lfloor k/2 \rfloor}$ and let $l = n/b$. Let u and v be represented as b (l)-bit numbers such that

$$u = \sum_{i=0}^{b-1} u_i\, 2^{li} \quad \text{and} \quad v = \sum_{i=0}^{b-1} v_i\, 2^{li} \ .$$

We compute the product of u and v *modulo* 2^n+1 using FFT's where each butterfly is computed in the ring of integers *modulo* $2^{2l}+1$. b and l are both approximately $\sqrt{n}$, so a single FFT leaves us with $\sqrt{n}$ multiplications of $(2\sqrt{n})$-bit numbers. The algorithm is used recursively to perform these multiplications. The original multiplicands are padded with 0's so the answer *modulo* 2^n+1 is exact. Recursive multiplications do not require padding since we only need the modular congruent result.

For speed in the butterflies, we use a two-valued representation for the numbers, and design each FFT butterfly computation so that it operates on this two-valued representation instead of the normal single-valued representation. If we have an x-bit number a, which is *modulo* 2^x, we can represent a as the sum of two x-bit numbers $\dot{a}$ and $\ddot{a}$ (which we call "carry-save notation" (CSN)), such that

$$a \ modulo\, 2^x \equiv (\dot{a} + \ddot{a}) \ modulo\, 2^x \ .$$

The presentation in [1] uses butterfly computations *modulo* $2^{2l}+1$. Performing binary arithmetic on numbers in this ring is awkward. In [19], Schönhage and Strassen suggest computing in the ring of integers *modulo* 2^{4l}, then converting the number back to *modulo* $2^{2l}+1$ when the computations are finished. Performing binary arithmetic *modulo* 2^{4l} is more convenient than *modulo* $2^{2l}+1$. This representation has duplicate values, but also has several noteworthy advantages, since $2^{4l} \equiv 1$ *modulo* $2^{2l}+1$ and $2^{2l} \equiv -1$ *modulo* $2^{2l}+1$.

Multiplication by 2^x is a shift left of x places. To maintain congruence *modulo* $2^{2l}+1$, since $2^{4l} \equiv 1$, each bit shifted off the left end of a number is shifted back into the right end. Thus multiplication by 2^x is simply a rotate left of x places. Addition with a CSA is the same – the carry out of the highest position is written into the unoccupied lowest order position of sum. Since $2^{2l} \equiv -1$ *modulo* $2^{2l}+1$, subtraction can be done with the formula $a - b \equiv a + 2^{2l} b$, and is consequently a combination of rotation and addition.

The CSN number $u = (\dot{u} + \ddot{u})$ *modulo* 2^{4l} can be converted back to *modulo* $2^{2l}+1$ in the following manner [19]: first add $\dot{u}$ and $\ddot{u}$ using ALGORITHM A (except that the carry out must be routed to the carry in). This converts u from CSN to a single $(4l)$-bit number. View u as $u = 2^{2l}\delta + \varepsilon$, where δ and ε are $(2l)$-bit numbers. Since $2^{2l} \equiv -1$, we can write $u = (\varepsilon - \delta)$ *modulo* $2^{2l}+1$, which is computed by

$$u = \begin{cases} \varepsilon - \delta & \varepsilon \geq \delta \\ 2^{2l}+1+\varepsilon-\delta & \varepsilon < \delta \ . \end{cases} \qquad (1)$$

With these concepts, we now give a parallel version of the Schönhage-Strassen multiplication algorithm. The majority of the presentation is similar to that given in [1], but we have made four significant changes. In the FFT's, we perform the butterfly computations *modulo* 2^{4l} instead of *modulo* $2^{2l}+1$, and represent the numbers in CSN. At the end of each FFT we resolve the $(4l)$-bit CSN form of the results to single *modulo* $2^{2l}+1$ numbers. The third change is in the multiplication *modulo* b. Schönhage and Strassen make clever use of the Karatsuba $O(n^{\log 3})$ multiplication algorithm - we use ALGORITHM B instead. The fourth change is the final reconstruction of the result in the last step. As we did with the last step of the complex-field FFT, we convert the summands to three large numbers and add them. The algorithm is shown at the end of this paper.

Resolving a *modulo* 2^{4l} number in CSN to a single-valued *modulo* $2^{2l}+1$ number using equation (1) requires a constant number of applications of ALGORITHM A and takes $O(\log l)$ time.

An FFT requires $\log b$ stages, each of which computes b butterflies *modulo* 2^{4l}. Each butterfly computation can be done in constant time. A complete FFT takes $O(\log b)$ time with $O(bl)$ PE's.

Step 2.1 takes $O(\log b)$ time with $O(bl)$ PE's.

Step 2.2 takes $O(\log l)$ time with $O(bl)$ PE's.

Step 2.3 is a recursive call to this algorithm for b pairs of $(2l)$-bit numbers. We will examine this step further in a moment.

Step 2.4 is the same as step 2.1.

Step 2.5 takes $O(\log l)$ time with $O(bl)$ PE's.

Step 2.6 takes $O(\log\log b)$ time with $O(b \log^2 b)$ PE's.

Step 2.7 takes $O(\log l)$ time with $O(bl)$ PE's.

Step 2.8 takes $O(\log n)$ time with $O(n)$ PE's.

Excluding step 2.3 for the moment, the dominant time required is $O(\log n)$ with $O(n)$ PE's. To include step 2.3, we need to know the total number of levels of recursion. We observe that $2l$ decreases from level to level in the following manner:

$$2\sqrt{n} \ , 2\sqrt{2\sqrt{n}} \ , 2\sqrt{2\sqrt{2\sqrt{n}}} \ , \cdots$$

which we can write as

$$2^{\frac{2^{i+1}-1}{2^i}} \ n^{\frac{1}{2^{i+1}}} \ , \ i = 0 \ , 1 \ , 2 \ , 3 \ , \cdots$$

and is approximately $4\, n^{\frac{1}{2^{i+1}}}$. Taking the log, we get $2^{-i-1}\log n + 2$ which is a constant when $i = \log\log n - 1$, so the recursion proceeds through $O(\log\log n)$ levels. Since the time required is $O(\log n)$ for a single level of recursion on arguments of size n, the $2^{-i-1}\log n$ form for the log of the argument size gives an overall time requirement of $O(\log n)$ with $O(n \log\log n)$ PE's.

In [19], Schönhage and Strassen give a depth complexity of $O(\log n)$ for a logic net which implements their algorithm. We have shown that we can achieve the same time in the EREW PRAM model by representing the numbers in CSN and using CSA's in the butterfly computations.

5. Conclusions and Further Research

The FFT is well-known as a useful tool for serial multiplication of large integers, and we have shown that it is quite powerful for parallel integer multiplication as well.

Since the original preparation of this paper, more detailed work has been performed. Work has been done in three models - a version of the EREW PRAM model where each memory cell contains 1 bit and each PE can perform 1-bit boolean operations (abbrev. PRAM-1), a version of the EREW PRAM model where each memory cell contains 32 bits and each PE can perform 32-bit boolean and arithmetic operations (including multiplication) (abbrev. PRAM-32), and an interconnection model where 32-bit PE's were connected via a barrel shifter (abbrev. Bsh). The performance was estimated based on running time and on cost (running time × number of PE's).

In all models, ALGORITHM C had a consistently higher cost than ALGORITHM D. This is due largely to the high constant factor associated with the $O(\log\log n)$ time required by the butterfly computations in ALGORITHM C. ALGORITHM B was faster than other algorithms, and for small values of n had a lower cost, but the cost grows much more rapidly, crossing ALGORITHM D at about $n = 2^9$. In the PRAM-32 model, the appearance of a 32-bit addition capability had a dramatic effect. ALGORITHM A is no longer necessary, since probabilistic arguments weigh heavily in favor of a simple ripple-carry addition approach with a small expected running time relatively independent of n. A faster version of ALGORITHM B was developed which used the 32-bit addition instructions instead of carry-save addition. The cost of this crosses the cost of ALGORITHM D at about $n = 2^{12}$. The 32-bit multiplications make ALGORITHM C faster than ALGORITHM D for small values of n, but ALGORITHM D is still faster after about $n = 2^{14}$.

In the Bsh model, ALGORITHMs C and D require about the same time. With a fast 32-bit multiplier and communication delays more than about four times the instruction time, ALGORITHM C is definitely faster than ALGORITHM D. Across the board, however, the cost of ALGORITHM D is still consistently less than that of ALGORITHM C. This apparent improvement in ALGORITHM C is actually the result of the difficulty encountered by ALGORITHM D when moved to an interconnection network. Except for the butterfly-to-butterfly communication, which is well understood, ALGORITHM C only needs to transfer data over a range of $O(\log^2 n)$ PE's in the floating point multiplications, while ALGORITHM D needs to transfer data over a range of $O(\sqrt{n})$ PE's for the *modulo* 2^{4l} rotations. Once communication delays become a function of the distance travelled, the time required by ALGORITHM D increases.

There are still a number of open questions, including the specific type of FFT to use and the possibility of VLSI implementation. We suspect VLSI implementation of ALGORITHM C would be more effective since it is simpler and has shorter PE-to-PE communication requirements. An investigation into the possible improvements which come from an MIMD implementation is also warranted.

BEGIN ALGORITHM D

Input: The input is two n-bit integers, u and v, where $n = 2^k$. At the topmost level of recursion, the uppermost $n/2$ bits of u and v must be 0. This is not required at any subsequent level of recursion.

Output: The output is the $(n+1)$-bit product of u and v *modulo* $2^n + 1$.

2.0 If n is small, multiply u and v *modulo* $2^n + 1$ using any method. Otherwise let $b = 2^{\lfloor k/2 \rfloor}$ and let $l = n/b$. Express u and v as $u = \sum_{i=0}^{b-1} u_i 2^{li}$ and $v = \sum_{i=0}^{b-1} v_i 2^{li}$, and convert each to CSN *modulo* 2^{4l}.

2.1 Compute the FFT's *modulo* $2^{2l}+1$ (using *modulo* 2^{4l} CSN) of $[u_0, \psi u_1, \psi^2 u_2, \cdots, \psi^{b-1} u_{b-1}]$ and $[v_0, \psi v_1, \psi^2 v_2, \cdots, \psi^{b-1} v_{b-1}]$ where $\psi = 2^{2l/b}$ and $\omega = \psi^2$ is a b-th root of unity.

2.2 Resolve the CSN *modulo* 2^{4l} results $[\hat{u}'_0, \cdots, \hat{u}'_{b-1}]$ and $[\hat{v}'_0, \cdots, \hat{v}'_{b-1}]$ to their single-valued *modulo* $2^{2l}+1$ equivalents $[\hat{u}_0, \cdots, \hat{u}_{b-1}]$ and $[\hat{v}_0, \cdots, \hat{v}_{b-1}]$ using equation (1).

2.3 Compute the pairwise *modulo* $2^{2l}+1$ products $[(\hat{u}_0 \times \hat{v}_0)$ *modulo* $2^{2l}+1, \cdots, (\hat{u}_{b-1} \times \hat{v}_{b-1})$ *modulo* $2^{2l}+1]$ of the two FFT's by recursive use of ALGORITHM D.

2.4 Compute the inverse FFT *modulo* $2^{2l}+1$ of the pairwise products from step 2.3. The result is $[\hat{w}_0, \psi\hat{w}_1, \cdots, \psi^{b-1}\hat{w}_{b-1}]$ where each $\psi^i \hat{w}_i$ product is *modulo* 2^{4l} and is in CSN.

2.5 Compute the single-valued numbers $w''_i = w_i$ *modulo* $2^{2l}+1$ by first computing $\hat{w}_i = (\psi^i \hat{w}_i \times \psi^{-i})$ *modulo* 2^{4l}, then computing $w''_i = \hat{w}_i$ *modulo* $2^{2l}+1$ using equation (1).

2.6 Compute $w'_i = w_i$ *modulo* b by computing $w'_i \leftarrow ((u_i$ *modulo* $b) \times (v_i$ *modulo* $b))$ *modulo* b using ALGORITHM B.

2.7 Compute the exact w_i values by computing $w_i = (2^{2l}+1)((w'_i - w''_i)$ *modulo* $b) + w''_i$, where each w_i is positive and no more than $b\,2^{2l}$.

2.8 Construct three n-bit numbers with non-overlapping sequences from $\sum_{i=0}^{b-1} w_i 2^{li}$ *modulo* 2^n+1 and add them *modulo* 2^n+1. This is the desired result.

END ALGORITHM D

References

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, Massachusetts (1974).
2. Carl M. Bender and Steven A. Orszag, *Advanced Mathematical Methods for Scientists and Engineers*, McGraw-Hill Book Company, New York (1978).
3. P. B. and J. M. Borwein, *Pi and the AGM*, John Wiley and Sons, New York, (1987).
4. P. B. and J. M. Borwein, "Ramanujan and Pi," *Scientific American* (Feb. 1988).
5. E. O. Brigham, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey (1974).
6. J. W. Cooley, P. A. Lewis, and P. D. Welch, "The Fast Fourier Transform Algorithm and its Applications," *IBM Research report* **RC 1743** (Feb 9, 1967).
7. Eliezer Dekel and Sartaj Sahni, "Binary Trees and Parallel Scheduling Algorithms," *IEEE Transactions on Computers* **C-32, No. 3** pp. 307-315 (March 1983).
8. B. Fornberg, "A vector implementation of the fast Fourier transform," *Math. Comput.* **36**, pp. 189-191 (1981).
9. Kai Hwang and Fayé A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, New York (1984).
10. Donald E. Knuth, *The Art of Computer Programming - Volume 2 / Seminumerical Algorithms*, Second Edition, Addison-Wesley Publishing Company, Reading, Massachusetts (1981).
11. V. M. Krapchenko, "Asymptotic Estimation of Addition Time of a Parallel Adder," *Syst. Theory. Res.* [trans. from Probl. Kilbern, 19 (1967), 107-122], **19** pp. 105-122 (1967).
12. Clyde P. Kruskal, Larry Rudolph, and Marc Snir, "Efficient Parallel Algorithms for Graph Problems," *IEEE 1986 International Conference on Parallel Processing*, pp. 869-876 (1986).
13. Clare D. McGillem and George R. Cooper, *Continuous and Discrete Signal and System Analysis*, Holt, Rinehart, and Winston, Inc., New York (1974).
14. Alan V. Oppenheim and Ronald W. Schafer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey (1975).
15. Shie-Tung Peng and Thomas F. Hudson, jr., "Parallel Algorithms for Multiplying Very Large Integers," *UMBC Dept. of Computer Science tech. report* **TR CS-PP-002** (February, 1988).
16. Michael J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill Book Company, New York (1987).
17. John E. Savage, *The Complexity of Computing*, John Wiley & Sons, Inc., New York (1976).
18. Udo Schendel, translated by B. W. Conolly, *Introduction to Numerical Methods for Parallel Computers*, Ellis Horwood Limited, Chichester (1984).
19. Arnold Schönhage and Volker Strassen, "Schnelle Multiplikation großer Zahlen," *Computing* **7** pp. 281-292 (1971).
20. Marc Snir, "On Parallel Searching," *ACM Symposium on Distributed Computing*, pp. 242-253 (August 1982).
21. Paul N. Swarztrauber, "Multiprocessor FFT's," *Parallel Computing* **5** pp. 197-210 (1987).
22. Paul N. Swarztrauber, "Vectorizing the FFT's," pp. 490-501 in G. Rodrigue, ed., *Parallel Computations*, Academic Press, New York (1982).
23. Shmuel Winograd, "On the Time Required to Perform Addition," *Journal of the ACM* **12, No. 2** pp. 277-285 (April 1965).
24. J. C. Wyllie, "The Complexity of Parallel Computations," *Ph.D. dissertation, Dept. of Computer Science,* Cornell University, (1979).

A Parallel Pivoting Algorithm on a Shared Memory Multiprocessor

with Fill-in Control

Gita Alaghband
Department of Electrical Engineering and Computer Science
University of Colorado at Denver
1200 Larimer St., Campus Box 110
Denver, Colorado 80204†

Abstract -- During L/U decomposition of a sparse matrix, it is possible to perform computation on many diagonal elements simultaneously. Pivots that can be processed in parallel are related by a compatibility relation and are grouped in a compatible set. The collection of all maximal compatibles yields different maximum sized sets of pivots that can be processed in parallel. Generation of the maximal compatibles is based on the construction of an incompatible table which gives information about pairs of incompatible pivots. The algorithm to generate all maximal compatibles involves a binary tree search and is exponential in the order of the matrix. A technique to obtain an ordered compatible set directly from the ordered incompatible table is given. The ordering is based on the Markowitz number of the pivot candidates. This technique generates a set of compatible pivots with the property of generating few fills. A new heuristic algorithm is presented that combines the idea of an ordered compatible set with a limited binary tree search to generate several sets of compatible pivots in linear time. An elimination set to reduce the matrix is generated and selected on the basis of a minimum Markowitz sum number. The parallel pivoting technique is a stepwise algorithm and can be applied to any submatrix of the original matrix. Thus it is not a preordering of the sparse matrix and is applied dynamically as the decomposition proceeds. Parameters are suggested to obtain a balance between parallelism and fill-ins. A sample result of applying the proposed algorithms on an application matrix using the HEP multiprocessor is presented.

1. Introduction

In this paper we present a parallel triangularization algorithm for solving large, sparse systems of linear equations. The algorithms described are designed for a shared-memory MIMD model for parallel computation, in which the total memory address space is accessible uniformly to all parallel units. This computational model provides synchronization mechanisms to allow multiple memory updates. If multiple updates are aimed at the same memory cell, the penalty paid is a short delay in access time.

The triangulation of an $n \times n$ matrix $A = [a_{ij}]$ can be described by the following procedure.

for $K = 1,2,...,n-1$ and for *each* $a_{jk} \neq 0$

$$a_{jk} \leftarrow a_{jk}/a_{kk} \qquad j>k \tag{1.1}$$

For each pair $a_{ik} \cdot a_{kj} \neq 0$

$$a_{ij} \leftarrow a_{ij} - a_{ik} \times a_{kj} \qquad i>k,\ j>k \tag{1.2}$$

In (1.2) if $a_{ij}=0$ but $a_{ik} \cdot a_{kj} \neq 0$, a fill-in is generated. If we have sufficient processors, the divide operations (1.1) for each column K can be done in parallel. Also, for each k the update operation (1.2) for all pairs $a_{ik} \cdot a_{kj} \neq 0$ can be done in parallel. Our experience in employing this approach has indicated that the sparsity of application matrices leaves parallel processes with little work to perform if only reduction for a single pivot is done in parallel [1]. During Sparse LU decomposition it is possible to perform computation on many diagonal elements simultaneously. In parallel LU decomposition of general unsymmetric sparse matrices several key issues must be considered:

a) Parallelism and fill-in are two competing issues and a balance between the two must be obtained. In other words minimizing fill-ins results in limited parallelism, and maximizing parallelism results in uncontrolled generation of fill-ins.

b) A test for numerical stability of pivots must be made to ensure the accuracy of the solution process.

c) In applications where the sparse linear system must be solved repeatedly, it must be possible to decompose structurally identical matrices using the information produced for the first decomposition of such matrix.

d) A storage structure suitable for parallel processing must be determined.

In the remaining of this paper we will present the design of a heuristic algorithm [2] [3], and [4] which identifies parallel pivot candidates and allows the matrix to be reduced for multiple pivots simultaneously while it minimizes fill-ins. Other parallel pivoting strategies have been suggested [5], [6], [7], [8], [9], [10], [11], [12]. The proposed algorithm is a dynamic algorithm which can be applied at any point during the decomposition phase and does not require a preordering of the input matrix. Therefore pivots can be tested for numerical stability and unsymmetric permutations are possible between consecutive applications of the parallel pivoting algorithm to the sparse matrix under consideration. This technique also allows the decomposition of structurally identical matrices to be carried out as required in (c).

A description of the parallel pivoting algorithm is given in section 2. Section 3 gives an analysis of the order of the algorithm. Section 4 describes several program-controlled parameters to control generation of fill-ins. Finally in section 5 some results and concluding remarks are given.

2. Parallel Pivoting Algorithm

In the technique described here parallel pivot candidates are obtained from the diagonal elements of the matrix, thus allowing only symmetric permutations during the application of the parallel pivoting algorithm. However, since it is a stepwise algorithm, it is possible to perform unsymmetric permutations in between the applications of the parallel pivoting procedure.

Pivots that can be processed in parallel are related by a compatibility relation and are grouped in a compatible. In other words pivots P_{ii}, P_{jj}, P_{kk} are compatible and can be processed in parallel if and only if elements $a_{ij}, a_{ji}, a_{ik}, a_{ki}, a_{jk}, a_{kj}$ are all zero. A compatibility relation classifies the elements of a set into nondisjoint subsets, so that all members of a subset are compatible. Thus collection of all maximal compatibles [13], yields different maximum sized sets of pivots that can be processed in parallel. Several methods for generating maximal compatibles exist and they are all based on the construction of an implication (incompatible) table. The incompatible table gives information about pairs of incompatible pivots. Let the incompatible table be represented by an array, *imptbl*, of dimension n, order of the matrix, with elements of *imptbl* being sets of n elements each. Each set corresponds to a column of the table. If we assume the diagonal elements of the matrix are numbered 1 through n, then column i of the table, $imptbl_i$ holds the incompatible information for pivot i of the matrix. Figure 2.1.b shows the incompatible table for the matrix of Figure 2.1.a in the given order. Column one of the table is constructed by scanning row and column 1 of the matrix. For each nonzero encountered, the corresponding position in $imptbl_1$ is marked. Therefore pivot p_1 is incompatible with pivots p_7 and p_8. Next, ignoring row/column 1 of the matrix, $imptbl_2$ is constructed by scanning row/column 2. The process is repeated until all (n-1) columns of the table are complete.

A systematical approach for extracting the maximal compatibles involves a binary tree search. This approach is exponential in the order of the matrix, however, its description is essential in the derivation of the target algorithm. Initially, it is assumed that all pivots are compatible. They are grouped in one set consisting of all pivot. This set, S, will be at the root of a binary tree, level zero. Next, the set of pivots incompatible with, P_1, obtained from $imptbl_1$ is used to split S into a left S_1 and a right S_2 set, constituting level one. S_1 consists of all elements of its parent S except those incompatible with P_1. S_2 consists of the same elements as S except P_1 itself. Next using $imptbl_2$ we split every set at level 1 to

†Research was supported in part by NASA Contract No. NAS1-17070 and by the Air Force Office of Scientific Research under Grant No. AFOSR 85-1089 while the author was in residence at ICASE, NASA Langly Research Center, Hampton, VA 23665.

	1	2	3	4	5	6	7	8
1	x						x	
2		x			x	x		
3		x	x				x	
4				x	x			
5		x		x	x		x	
6		x				x	x	x
7	x				x	x	x	
8	x		x		x			x

Figure 2.1.a Matrix A1

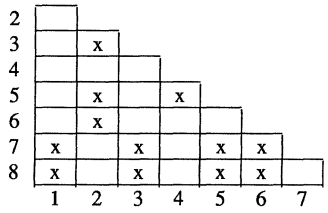


Figure 2.1.b Incompatible Table

generate the four sets at level 2. This process is repeated until no more splitting of the sets are possible. Figure 2.1.c illustrates this procedure for the matrix of Figure 2.1.a. The leaf sets are checked and any set included in a larger set is eliminated. The remaining sets constitutes all possible maximal compatibles.

Among the maximal compatibles generated, sets b, c, and d are of maximum size 4. This suggests we need a mechanism to select a set among all the candidate sets which would generate fewest fill-in. The technique used here is based on the Markowitz criterion for minimizing fill-ins in sparse matrices in sequential programming. The Markowitz [14] number of an element a_{ij} is defined by $(r_i-1)(c_j-1)$, where r_i and c_j are the number of nonzeros in row i and column j of the reduced matrix respectively. At each step, the element with minimum Markowitz number is selected as pivot. The strategy used here is called *Markowitz Sum* and is the sum of the Markowitz number of all pivots in a set. At each step, among all the maximal compatibles of equal maximum size we select the one with minimum Markowitz Sum to reduce the matrix. Sets b, c, and d have Markowitz Sum of 11, 22, and 22 respectively. Therefore the compatible pivots 1,3,4,6 in set b is the one to be selected to reduce the matrix in parallel.

The exponential characteristics of this algorithm prohibits its use. However, over a number of analysis done for several small test cases arising from electronic circuits, we found that many parallel computation steps are possible and during these steps the matrices are often reduced completely. Most importantly our results show that by reducing the amount of parallel work at each step slightly it is possible to reduce the generation of fill-in significantly and still reduce the matrix in the same number of steps [2]. The goal of our heuristic algorithm therefore is to obtain enough parallel work by just considering a sub-maximal set of compatible pivots at each step.

By a more careful analysis of the incompatible table we can produce a compatible without searching the binary tree. Column i of *imptbl* yields all pivots p_j, $j>i$, incompatible with p_i. If $imptbl_i$ is null, then p_i is compatible with every pivot p_j with $j>i$. By scanning the table, we find a set of pivots, *compset*, whose columns in the table are null. Clearly, such pivots are compatible. Furthermore, if the set of incompatible pivots of p_k are disjoint from *compset*, then p_k is also compatible with every pivot in *compset*. The procedure to produce an *ordered compatible*, *compset*, can be summarized as follows:

```
scan imptbl from right to left
  for each column i of imptbl do
    begin
    if ( imptbl_i ∩ compset is empty) then
    (*add [i] to the set of compatibles*)
    compset = compset + [i]
    else
    delete row i of imptbl
    end
```

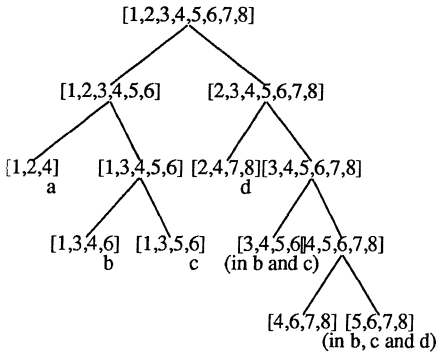


Figure 2.1.c: Binary Tree Search to Obtain the Set of Maximal Compatibles

Applying this procedure to the incompatible table of Figure 2.1.b results in:

Null columns = (7,8), *Compset* = (4,7,8)
Markowitz Sum = 1 + 12 + 3 = 16

Pivot 7 is the pivot with maximum Markowitz number in the matrix. We would like to have compatible pivots with as low Markowitz numbers as possible in order to minimize fill-in. It is clear that pivots with low Markowitz number generally have fewer incompatibilities. Changing the order of construction of the incompatible table can eliminate the inclusion of pivots with high Markowitz number in the *ordered compatible*. The table is constructed for pivots in decreasing order of Markowitz number. This is illustrated for the same matrix of Figure 2.1.a in Figure 2.1.d, where a larger *compset*, size 4, of lower Markowitz Sum, 11, is obtained.

Now we can combine the idea of an *ordered compatible* with the tree search algorithm to obtain a limited tree search algorithm which produces an acceptable set of compatible pivots for reducing the matrix. This is done by partially searching the binary tree up to a given level, *ULEVEL*, generating a number of sets. Each of these sets are some subset of the root and can be considered as a starting set. Thus by scanning the incompatible table corresponding to each of the starting sets at *ULEVEL*, we can construct an *ordered compatible*.

Among all *ordered compatibles*, the one of maximum size and minimum Markowitz Sum is chosen as the *elimination set* to reduce the

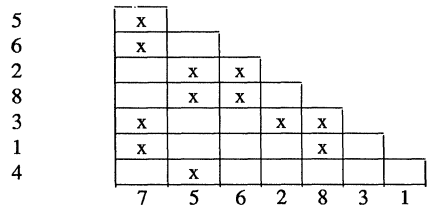


Figure 2.1.d Ordered Incompatible Table
Null columns: (1,3,4), Compset: (1,3,4,6)
Markowitz Sum: 2+2+1+6 = 11

matrix. Different orderings for producing the starting sets at *ULEVEL* have been considered [2], [3]. The most promising one is to split the sets with pivots in increasing order of their Markowitz numbers. This process seems to give a good balance to the binary tree for the first few levels used to generate the starting sets. It also has the property to keep pivots of low Markowitz numbers within the starting sets.

3. Order of Parallel Pivoting Algorithm

The algorithm is no longer exponential in time. It consists of the following steps:

1. Construct the incompatible table.
2. Sort pivots according to Markowitz numbers.

3. Produce starting sets at *ULEVEL* .

4. Generate an ordered compatible for each of the starting sets at *ULEVEL*.

The algorithm involves some set manipulation operations. These operations are listed for each section separately as required.

1. Construction of the incompatible table requires scanning the non-zero elements of the matrix. The set operations involved are adding an element to a set and test for membership which are both $O(1)$ operations. Therefore, the incompatible table can be constructed in $O(NZ)$ operations where NZ is the number of nonzeros in the matrix.

2. A sorting algorithm is needed to order the pivots according to their Markowitz numbers. The Batcher sort [15] is used here and it has been shown that with enough parallel operations, sorting is completed in $1/2\lceil \log n \rceil (\lceil \log n \rceil + 1)$ steps. Employing an efficient sort would improve the performance of the algorithm.

3. Production of all starting sets at *ULEVEL* takes a constant time, say K, proportional to the number of starting sets. It involves the set operations intersection, difference, deletion of an element from the set, and test for a null set. These operations are of order n with a constant factor equal to the inverse of the number of bits per computer word. The set operations are usually implemented in machine language or microcode and thus have a small time factor. They could be considered to have a constant time (rather than order of n) compared to the time taken to execute a high level language statement. Therefore, an efficient implementation of the set operations is important to the efficient execution of the algorithm. If we denote the time to do a set operation with *setop* then this section can be done in $O(K \cdot setop)$.

4. Generation of an ordered compatible from the incompatible table requires scanning n sets corresponding to the columns of the table and performing intersection and difference operations on the sets. Thus ordered compatibles can be produced in $O(K \cdot n \cdot setop)$, where K is the number of starting sets at *ULEVEL*. For reasonable values of *ULEVEL*, all ordered compatibles can be derived in parallel for different starting sets.

4. Fill-in Minimization

It is possible to minimize generation of fill-ins significantly by reducing the amount of parallel work slightly. Trading off parallelism for fill-in reduction is done according to the size of the *elimination set* and a number of parameters:

1. Shrinkage parameter: By allowing a small percentage of the *elimination set* to be discarded we can control the number of compatible pivots to a degree that does not limit our parallel work by too much.
2. Upper limit parameter: This limit would allow just enough parallel work to keep our parallel processes busy.
3. Threshold parameter: In shrinking the size of an *elimination set* only pivots with Markowitz number higher than a threshold value in the ordered list of pivots may be discarded. Pivots with low Markowitz numbers do not tend to generate many fills and need not be discarded.

5. Result and Conclusion

The above algorithm has been simulated on a VAX 11/780 and implemented on the HEP pipelined shared-memory computer [16]. Test cases from application programs such as the SPICE circuit simulation [17] and SPAR structural analysis program [18] have been used and results of different analysis are available [2]. Here we represent the timing results of running the parallel program over a 144 by 144 matrix from the circuit of an 8-bit full adder and employing different values for trade off parameters. Figure 5.1 represents the execution time of parallel *LU Decomposition* program for different numbers of processes from 1 to 25. The result is for the case when maximum parallelism is used. For $NPROC=1$, the matrix is completely reduced in 10 parallel steps. The number of compatible pivots at each step is 72, 25, 16, 11, 6, 5, 3, 2, 2, and 1 respectively. Note that in the first step half of the matrix is reduced in parallel. The execution time decreases with an increase in the number of processes up to $NPROC=11$. In fact there is $1/NPROC$ reduction in execution time for small values of *NPROC* as new processes make efficient use of the execution pipeline. This decrease in execution time bottoms out as the pipeline becomes full (the execution pipeline on the HEP has eight steps, resulting in a speed up of about 8). The slope of the linearly rising tail of the curve indicates the length of time spent in critical sections in various points in the program. A complete model for analysis of parallel programs can be found in [19]. Defining the speed up to be:

$$S = T(1) / T(NPROC)$$

where $T(1)$ is the time to execute the program with one process and $T(NPROC)$ is the same time using *NPROC* processes. Then a speed up of 4.82 is obtained for 11 processes. Note that this is not speed up measured with respect to the best sequential algorithm, but only gives insight to the parallelism in this program.

For a small number of processes, execution time versus number *NPROC* of processes can be represented as:

$$T(NPROC) = C_1 + C_2 / NPROC$$

where C_1 represents the sequential portion of the work and C_2 the parallel portion. A simple least squares fit to determine C_1 and C_2 is applied to a linear portion of the execution time versus *NPROC* curve to estimate the degree of parallelism. This analysis shows that the code is 87% parallel. The speed up of 5.81 was obtained when the following trade off parameters where used

Threshold	1/3	Shrinkage Parameter	30%
Upper Limit	25	*ULEVEL*	4

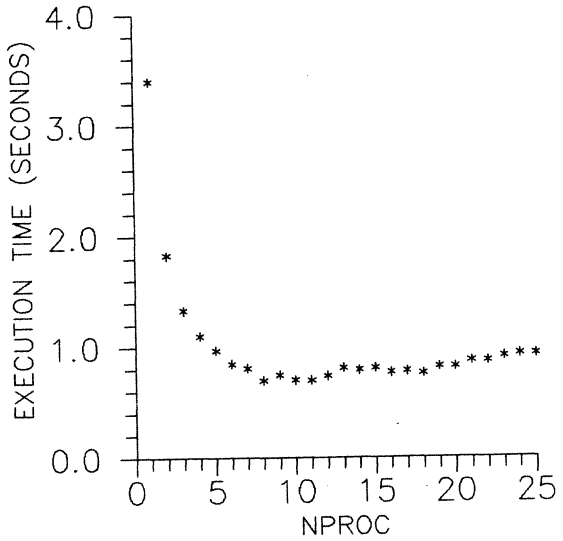


Figure 5.1 Execution Time vs. Number of Processes
No Trade off
144×144, NZ=616, 8-Bit Full Adder

The higher speed up indicates that by employing the above parameters a better balance between number of compatible pivots generated at different steps is achieved. A reduction of 23% in fill is obtained as the result of the above parameter variations which compares reasonably with results from the best sequential program. The fill-in can further be decreased by assigning different values to trade off parameters.

The sparse LU decomposition technique described in this paper employs a parallel pivoting strategy to solve the problem of having enough parallelism in sparse matrices. The main features of the heuristic algorithm can be summarized as follows:

-It can identify a good set of parallel pivots in linear time.

-It is a stepwise algorithm and can be applied to any submatrix of the original matrix. Thus it is not a preordering of the sparse matrix and is applied dynamically as the decomposition proceeds.

-Pivots can be tested for numerical stability and unsymmetric permutations can be performed if necessary.

-Trade off between parallelism and fill-in is possible under several program controlled parameters.

-The information produced by the algorithm can be stored to decompose structurally identical matrices.

We have presented the parallel reduction combined with parallel pivoting technique, control over the generation of fills and check for numerical stability, all done in parallel with work being distributed over the active processes. The program verifies that it is actually possible to do parallel pivoting in sparse matrices on multiprocessors and take advantage of the existing parallelism in the problem and in the hardware.

References

[1] G. Alaghband and H. F. Jordan "Parallelizing a Sparse Matrix Package," *Report CSDG 83-3,* Computer System Design Group, Electrical and Computer Engineering Department, University of Colorado, Boulder, June 1983.

[2] G. Alaghband *"Multiprocessor Sparse LU Decomposition with Controlled Fill-in,"* Ph.D. thesis, Department of Electrical and Computer Engineering, University of Colorado, Boulder, May 1986.

[3] G. Alaghband and H. F. Jordan "Multiprocessor Sparse L/U Decomposition with Controlled fill-in," *ICASE Report No. 85-48,* NASA Langley Research Center, Hampton, Virginia 23665, 1985.

[4] G. Alaghband "Parallel Pivoting Combined with Parallel Reduction," *ICASE Report No. 87-75,* NASA Langley Research Center, Hampton, Virginia 23665, 1987.

[5] D. A. Calahan, "Parallel Solution of Sparse Simultaneous Linear Equations," *Proc. 11-th Annual Allerton Conf. Circuits and System Theory,* pp. 729-735, Oct. 1973.

[6] J. W. Huang and O. Wing "Optimal Parallel Triangulation of a Sparse Matrix," *IEEE Trans. on Circuits and Systems,* vol. CAS-26, No. 9, pp. 726-732, Sept. 1976.

[7] Y. F. Zhou "Optimal Parallel Triangulation of a Sparse Matrix- A Graphical Approach," *IEEE 1981 Symp. on Circuits and Systems.*

[8] K. Nakajima "A Graph Theoretical Approach to Parallel Triangulation of a Sparse Asymmetric Matrix," *Proceedings of 1984 Conf. on Information Science and Systems.*

[9] J. A. G. Jess and H. G. M. Kees "A Data Structure for Parallel LU Decomposition," *IEEE Trans. on Computers,* vol. C-31, no. 3, pp. 231-239, March 1982.

[10] F. J. Peters "Parallel Pivoting Algorithms for Sparse Symmetric Matrices," *Parallel Computing 1,* pp. 99-110, 1984.

[11] M. R. Leuze "Parallel Triangularization of Substructured Finite Element Problems," *ICASE Report no. 84-47,* Sept. 1984.

[12] I. S. Duff "Parallel Implementation of Multifrontal Scheme," Argonne National Laboratory, Mathematics and Computer Science Division, *Technical Memorandum no. 49,* March 1985.

[13] Z. Kohavi *"Switching and Finite Automata Theory,"* Computer Science Series, Second Edition, McGraw Hill Book Company, 1978.

[14] H. M. Markowitz "The Elimination Form of the Inverse and its Application to Linear Programming," *Management Science, 3,* pp. 255-269, 1957.

[15] K. E. Batcher *Proc. AFIPS Spring joint Computer Conference, 32,* pp. 307-314, 1968.

[16] J. S. Kowalik *The HEP Supercomputer and Its Applications,* Ed., MIT Press, 1985.

[17] L.W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," *Memorandum ERL-M520,* Electronics Research Laboratory, University of Calif., Berkeley, CA 94720, 1975.

[18] "SPAR," *NASA CR 158970-1,* Engineering Information Systems Inc., San Jose, CA, Dec. 1978.

[19] H. F. Jordan "Interpreting Parallel Processor Performance Measurements," *Report CSDG 85-1,* Computer System Design Group, Electrical and Computer Engineering Department, University of Colorado, Boulder, November 1985.

Optimal Decomposition of Matrix Multiplication on Multiprocessor Architectures

Whelan, M., Gao, Guang R.[1] and Yum, T. K.[2]
Philips Laboratoris, North American Philips Corporation,
Briarcliff Manor, NY 10510

1. Introduction. Exploiting parallelism in matrix multiplication on a shared memory multiprocessor will encounter problems common to many other applications: determination of both an appropriate grain size for the parallel tasks for each processor, and an allocation which keeps the processors usefully busy.

In this paper, we study the problem of decomposing matrix multiplication so that it achieves optimal speedup on a multiprocessor. In scientific and engineering applications, matrix multiplication is frequently used as a kernel benchmark in the evaluation of supercomputer performance [4, 5]. Many other linear algebra algorithms may be transformed or reduced to contain matrix multiplication as their kernel [1].

Our approach is based on the observation that matrix multiplication has a regular iterative nature and that task decompositions based on iterations have fixed computation and communication patterns. Therefore, an analytical model calculating the cost of such computation/communication patterns is possible. Compared with other recent related work [2, 7], our work computes the effect of computation and communication cost in the case of the execution of a single matrix multiply as opposed to the overlapped execution of many matrix multiplies. The results of this work derives optimal partitioning and allocation strategies for matrix multiplication on a shared memory multiprocessor.

In section 2, we describe the model of a multiprocessor architecture which is used in the analysis. In section 3, we formulate the decomposition problem, and derive analytically the optimal condition for program partitioning and an optimal allocation strategy. In section 4 we present a summary of simulation experiments which were performed to determine the validity of assumptions made in the analysis. In section 5, derive an optimal speedup function. In section 6, we discuss computation / communication tradeoffs, and in section 7 we conclude with a comparison of our work with other related work.

2. Architecture Model. The main features of the machine model our study is based on are:

1. There are n identical processors (PEs) connected by a single bus.
2. The bus is a packet oriented data transfer medium, as opposed to a backplane interconnect .
3. The memory system is organized in two levels. At the first level, each PE has its private local memory (PM) and a cache (CM); and at the second level, there is a common shared memory (SM).

The time to transfer a block of L words from SM to a CM is T_l where $T_l = T_s + L \times T_w$. Where T_s and T_w are respectively the packet overhead and the transfer time per element of data after start-up. Define $T_1 = (T_s + L \times T_w)/L$ i.e., T_1 is the average time to transfer a data element. Each PE can compute multiply accumulate operations in the context of a matrix multiplication at the rate of one per T_* time units. It must be noted that this cost must include the average cost of the conditional tests, loop counters, and address arithmetic which must be performed on a real machine.

In determining the load a task imposes on the shared bus, we do not account for the following items.

1. Fetching code from the shared memory.
2. Cache organization, e.g. replacement strategy.
3. Finite cache sizes.

Our rationale is that 1 is minimal since the code required to specify matrix multiplication is very small. Points 2 and 3 are justified on the assumption that even for relatively small caches, the frequencies with which cache collisions occur is very small, thus once a variable is accessed once by a processor, it will thereafter with very high probability reside in that processor's cache. It is our contention that within certain constraints, the analysis will still accurately predict the properties of a realistic machine. The results of simulations, which do account for these effects, verify this assumption.

[1]Current address: McGill Univ., School of Computer Science, 805 Sherbrooke St. West, Montreal, Canada H3A 2K6

[2]Current address: Nynex Corporation, White Plains, NY

3. Optimal Decomposition. The impact of decomposition strategy on the performance of matrix multiplication can be demonstrated by considering the following three decompositions for $C = A \times B$ (where A and B are $M \times M$ matrices): (1) uniprocessor; (2) M processors with each computing one row of C; (3) M^2 processors each computing one elements of C.

	T_{cp}	T_{cm}	$O(T_{tot})$
1.	$M^3 T_*$	$2M^2 T_1$	$M^3 T_*$
2.	$M^2 T_*$	$(1+M) M^2 T_1$	$M^3 T_1$
3.	$M T_*$	$2 M^3 T_1$	$2 M^3 T_1$

Figure 1: Three simple partitioning cases.

The first two columns of the table in Figure show the computation time per processor T_{cp} and communication time per processor T_{cm} required for each of the three cases. The third column of the table lists the dominant term of the total time. We assume that the matrices A and B are stored in shared memory and their elements will be transferred to processors as required by the computation. We may immediately observe that between cases 2 and 3 the total time doubled although M times as many processors are used.

For simplicity, we study the multiplication of two $M \times M$ square matrices A and B, although the extension to arbitrary matrix multiplication is relatively straightforward [8]. In this paper, the cost metric being used is total elapsed time. Thus, optimizing performance means minimizing the total time required to complete a single matrix multiplication. It should be noted that this is not the cost function used in other related work [2], where throughput is used. Throughput is an appropriate cost measure if one has many such computations to be performed and the computations may be overlapped. However, if one does not have many such computations, or, if the computations cannot be overlapped (e.g because of dependencies), then elapsed time is more appropriate.

The resultant matrix C is partitioned into rectangular submatrices to be computed by each PE using the basic block matrix multiplication algorithm [6]. The two aspects of decomposition strategy are closely related to each other, and must be considered together in order to determine the shape and size of the submatrices as the optimal condition for decomposition. The computation and communication of each task should be arranged so that optimal performance may be achieved.

It is natural to speculate that a square shape would be an optimal choice of the submatrices. This is indeed the case, although it is also the case that any row column interchange equivalent to an optimal square partitioning is also optimal [8]. For the remainder of this paper, we will assume that square partitioning is used, that is the result matrix C is partitioned into square submatrices to be computed by individual PE's. We must now determine the optimal size of the submatrices, and the manner in which a processor will compute its submatrix.

3.1 Problem Formulation. The matrix multiplication is partitioned into $H \times H$ square submatrices of C. For simplicity assume M is divisible by H. The number of PEs required is $n = M^2/H^2$.

The manner in which PEs compute their subtasks must now be chosen. We will use the outer product method, with a parameter R determining the size of the subtask. The partitioning strategy is:

1. Each PE is assigned the task of computing a submatrix of C, e.g. $C_{I,J}$, I,J in {0,1,2...(M/H-1)} .
2. Tasks assigned to PEs are further divided into M/R subtasks $(1 \leq R \leq M)$, each subtask computes a partial result of $C_{I,J}$ using the outer product method.

This task decomposition can be expressed algorithmically as :

```
FORALL I,J in (0 ... (M/H - 1))/* M*M/(H*H) tasks*/
 FOR i,j in (1..H) C(I*H+i,J*H+j)=0; ENDFOR/*init*/
 FOR l in ( 0 .. (M/R - 1 ) )      /* the subtasks */
  FOR k in (1 .. R)
   FOR i,j in (1 .. H)
    C(I*H+i,J*H+j) = C(I*H+i,J*H+j)
                   + A(I*H+i,l*R+k)*B(l*R+k,J*H+j);
   ENDFOR
  ENDFOR
 ENDFOR
ENDALL
```

The allocation of subtasks to the PEs is done in an overlapped fashion. The subtasks are grouped into sets with the k^{th} set containing the subtasks computing the k^{th} partial results for each PE. Hence there are a total of M/R sets and the number of subtasks in each set is n. The computation and communication is performed in *rounds*, each round computes one set of the subtasks. For our purposes, each subtask can be considered as one indivisible access and computation process, i.e. $H \times R$ elements from A and $R \times H$ elements of B are read into by a procesor and the partial result is computed.

3.2 Partition Regions. In Figure 2, we show several different partitions and allocations (for M=16), where shaded and unshaded boxes indicate communication

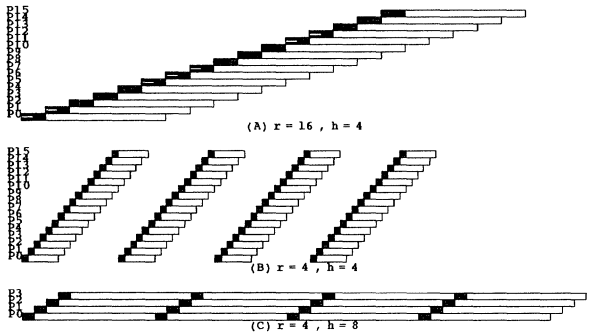


Figure 2: Computation/communication overlap.

and computation respectively. Let T_{tot} be the total time for performing the entire matrix multiplication. Let t_{cp}, t_{cm} be the time for the computation and communication time for an individual subtask. In Figure 2 (a) $R = M$, note that there is only one *round* in the process. In Figure 2 (b), the partition is such that the communication time dominates the total time. We can see that the bus will be busy most of the time. During the process, processors become idle while awaiting access to the bus. In Figure 2 (c), the partition is such that the computation time is dominant. In this case the processors are busy most of the time, but the bus becomes idle from time to time.

As illustrated by the examples, the system may operate in two different regions depending on the partitioning: *computation bounded* or *communication bounded.*

3.3 Overlapped Computation/Communication. For a subtask, we have:

$$t_{cp} = H^2 \times R \times T_* \tag{3.1}$$

$$t_{cm} = 2 \times H \times R \times T_1 \tag{3.2}$$

where T_* and T_1 are as defined in Sections 2. In the computation-bounded region the total time is

$$T_{tot_p} = \frac{M}{R} \times (t_{cp} + t_{cm}) + (n-1) \times t_{cm} \tag{3.3}$$

$$= H^2 \times M \times T_* + 2H \times T_1 \times (M + R(M^2/H^2 - 1)) \tag{3.4}$$

In the communication-bounded region the total time is

$$T_{tot_c} = n \times \frac{M}{R} \times t_{cm} + t_{cp} \tag{3.5}$$

$$= 2 \times \frac{M^3}{H} \times T_1 + H^2 \times R \times T_* \tag{3.6}$$

We note that as R increases, both equations will result in longer finishing times. Therefore we should choose the minimum possible R to reduce the total time in either case. Therefore, letting $R = 1$, equations (3.4) and 3.6 become:

$$T_{tot_p} = H^2 \times M \times T_* + 2H \times T_1 \times (M + (M/H)^2 - 1) \tag{3.7}$$

$$T_{tot_c} = 2 \times \frac{M^3}{H} \times T_1 + H^2 \times T_* \tag{3.8}$$

Now let us determine the value of H which will minimize the total time, this will be referred to as H_{opt}. Let us first determine the value of H at which the computation and communication times are totally overlapped. This happens precisely when the computation time of a subtask on one processor can be exactly overlapped with the communication time of the other $(n\text{-}1)$ processors. Under this condition, for example, the i^{th} round is ready to start to use the bus for the first subtask at exactly the moment when the communication for the $(i\text{-}1)^{th}$ round completes. In this case, we have:

$$(n-1)t_{cm} = t_{cp} \tag{3.9}$$

Substituting (3.1) and (3.2) into (3.9) and noting that $n = M^2/H^2$, we obtain:

$$H^3 \times T_* + 2 \times H^2 \times T_1 - 2 \times M^2 \times T_1 = 0 \tag{3.10}$$

Equation (3.10) establishes the key condition for maximum overlap between computation and communication time for matrix multiplication.

We have derived the condition of partitioning for maximum overlap from the argument of optimum tiling of the computation/communication timing graph. We have also deduced separately the formula for computing the total time valid for either the computation-bounded or the communication-bounded region such as (3.7) and (3.8). It is easy to show that the value of H at the intersection point of $T_{tot_p}(H)$ and $T_{tot_c}(H)$ satisfies equation (3.10). This can be shown by subtracting (3.8) from (3.7) and setting the result to zero, which yields equation (3.10). Thus the maximum overlap condition happens at precisely the intersection of the computation and communication bounded region.

3.4 Optimal Partition Size. It seems intuitive that the value of H which satisfies equation (3.10) should lead to a minimum total time because the processors and the bus are kept usefully busy all the time (except for initial and terminal transient behavior). From (3.7), we can compute the derivative of T_{tot} in the computation-bounded region and rewrite it in the following form:

$$\frac{dT_{tot}}{dH} = \frac{2 \times M}{H^2} \times (H^3 \times T_* + H^2 \times T_1 - \frac{1}{M} \times H^2 \times T_1 - M \times T_1) \tag{3.11}$$

According to (3.10), when $H > H_{opt}$ we have

$$H^3 \times T_* + H^2 \times T_1 > 2 \times M^2 \times T_1 - H^2 \times T_1 \quad (3.12)$$

Therefore, from (3.11) and (3.12) we deduce that when $H > H_{opt}$ the following holds in computation-bounded region:

$$\frac{dT_{tot}}{dH} > \frac{2 \times M}{H^2} \times T_1 \times (2 \times M^2 - H^2 - \frac{1}{M} \times H^2 - M) \quad (3.13)$$

Since $M \geq H$, from (3.13) we claim when $H > H_{opt}$ the following is true for all $M \geq 2$:

$$\frac{dT_{tot}}{dH} > 0 \quad (H > H_{opt}) \quad (3.14)$$

Similarly, from (3.8) we can show that in communication-bounded region

$$\frac{dT_{tot}}{dH} < 0 \quad (H < H_{opt}) \quad (3.15)$$

From (3.14) and (3.15), we know that the total time in the communication bounded region is a decreasing function of H until $H = H_{opt}$. After that, in the computation-bounded region the total time become an increasing function of H. Therefore, H_{opt} must be the one and only value in the range 1 . . . M at which T_{tot} reaches its minimum. We have thus proved the fact that the maximum overlapping point is indeed the optimal partitioning point.

Since equaion (3.10) is cubic in H, and that there is no linear term of H present, it can be shown [8] that there always exists a unique solution in the range 1 . . . M for H_{opt} which satisfies equation (3.10). For large M, the solution can be approximated by

$$H_{opt} \approx (2 \times M^2 \times \frac{T_1}{T_*})^{1/3} \quad (3.16)$$

4. Simulation. In this section, we describe a simulation study which was performed to validate the assumptions made in the preceeding analysis. A sample of the simulation results are shown in the following plot. The solid line indicates the results predicted using the analysis presented in the previous sections, while the simulation results are represented by point markers.

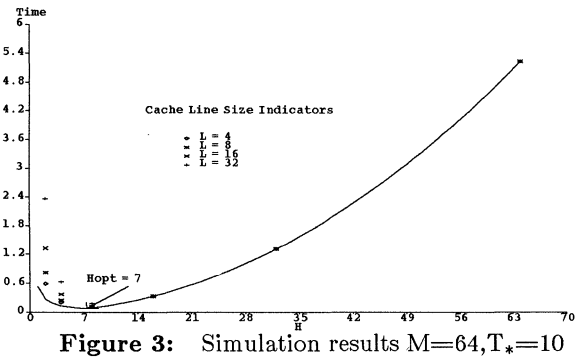


Figure 3: Simulation results M=64,T_*=10

In Figure 3, the difference of execution times for different cache line sizes are significant for small values of H, ($H<8$). For larger values of H, i.e. $H \geq 8$, the difference is much less substantial. As was shown in Section 4, the execution time becomes computation bounded for large values of H, i.e. $H > H_{opt}$. Therefore, the change of communication cost due to different cache line size has little impact on the execution time. On the other hand, for H less than H_{opt}, the execution time is communication bounded, therefore, any increase in the communication cost will directly contribute to execution time.

From Figure 3, it can be seen that the line size can have a significant effect on execution time (a factor of 3) in terms of the optimum attainable performance. The smaller the line size (from the data shown) the better would seem to be the case. However, for very small line sizes, where *small* depends on the packet overhead, the additional overhead causes performance degradation. Thus there is a *window* of appropriate cache line sizes. In summary, the simulation results do indeed validate the analysis.

5. Optimal Speedup. The optimal decomposition can be used to derive a speedup function. Let us derive the optimum speedup, denoted by S_{opt}, for our partitioning scheme. Here the speedup means the total uniprocessor time T_u divided by the multiprocessor processing time T_{tot}, i.e. $S = T_u / T_{tot}$. We can derive the optimal speedup from either T_{tot_p} or T_{tot_c}. For example, substitute T_{tot_c} in (5.1), we arrive at

$$S = \frac{T_u}{T_{cm} + t_{cp}} \quad (5.1)$$

Since $T_u = n \times M \times t_{cp}$, $T_{cm} = n \times M \times t_{cm}$, and $t_{cp} = (n-1) \times t_{cm}$ at optimal partition, we have

$$S_{opt} = \frac{n \times M}{n \times M + (n-1)} \times \frac{t_{cp}}{t_{cm}} \quad (5.2)$$

Assuming $M >> 1$ and consider the optimal condition, we have

$$S_{opt} \approx \frac{1}{2} \times H_{opt} \times \frac{T_*}{T_1} \quad (5.3)$$

From (5.3) and (3.16), we get

$$S_{opt} \approx \frac{1}{2} \times (2 \times M^2)^{1/3} \times (\frac{T_*}{T_1})^{2/3} \quad (5.4)$$

Comparing (5.4) with the corresponding result in [2], we can observe that, in both approaches, the optimal speedup is proportional to $(T_* / T_1)^{2/3}$. However, our

result is different from the result in [2] by the factor $((2M^2)^{1/3})/2$. This factor implies that the optimal speedup increases with the size of the matrix.

6. Communication/Communication Tradeoffs. The study reported here is interesting because the task in question, matrix multiplication, is regular enough that it can perform well on very powerful processing elements, but has sufficient parallelism to be exploited on very large numbers of processing elements. What the results of this study indicate is that for a fixed communications speed, one can use less powerful processing elements to achieve comparable speedup, provided one has sufficiently many of them.

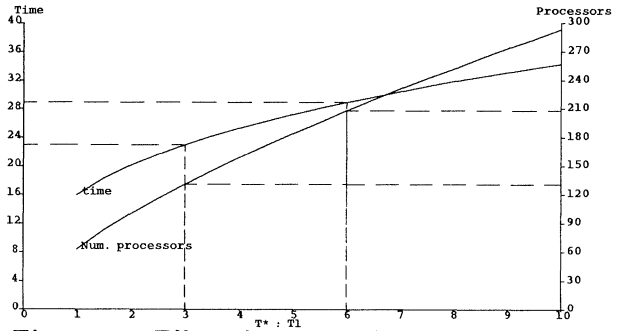


Figure 4: Effect of proc. speed on opt. performance

Figure 4 shows the time required for a large matrix multiply at the optimal partition size as the processor speed is varied. The two identified points represent $T_* = 3$ and $T_* = 6$, i.e. a halving of the speed of each PE, while maintaining the communications costs fixed. The resulting optimal performance shows a decrease of about 26%, with an increase in the number of processing elements to achieve this performance of about 50%. Thus we pay a penalty for the loss of processor speed, but it is much less than a halving of performance. Hence in considering alternatives, one should keep in mind the non linear relationship between individual element speed, and system speed.

7. Discussion. Some related results are reported in the recent work described by Vrsalovic et, al, and Cvetanovic [2, 7]. In contrast to the work reported in [2], however, we analyse the time for a single matrix multiplication, as opposed to the rate of execution of matrix multiplies (i.e. latency vs throughput).

We are able to show analytically, for the ideal machine model, square partitioning is necessary for optimality and our analysis is applicable for both square and non-square matrices [8]. Jalby and Meier [3] in the Cedar Project have reported a study on optimal partitioning of matrix multiplication for their machine. Under a different partitioning scheme, their results show that a rectangular submatrix may be the best choice for a processor architecture with combined vector and parallel capabilities. However, their work does not provide a complete analysis of the combined computation and communication time. Instead, by stating that such combined minimization problem is difficult, they consider each as an independent problem by considering only the dominant terms.

We have studied the critical problem of computation/communication tradeoffs for multiprocessor systems. We have demonstrated that there exists an optimal decomposition under which the matrix multiplication problem can achieve a maximum speedup on a shared memory multiprocessor with single bus interconnection. The size for the optimal partitioning is analytically characterized, and an allocation is formulated which can achieve optimal performance. Simulation results have confirmed our analytic frame work.

1. Aho A, Hopcroft J, Ullman J, *The Design and Analysis of Computer Algorithms*, Addison-Wesley 1974
2. Cvetanovic Z, *The Effects of problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems*, IEEE Trans. on Computers C-36(4), April 1987
3. Jalby W, Meier U, *Optimizing matrix Operations on a Parallel Multiprocessor with a Hierarchical Memory System*, Proceedings of 1986 ICPP Conference.
4. Masaaki Shimasaki *Performance Analysis of Vector Supercomputers by Hockney's Method* Proceedings of the 2nd International Conference on Supercomputers, Santa Clara, CA May 1987
5. Neves K, Simon H, *Supercomputer Performance Evaluation: Benchmarking Applications* Proceedings of the 2nd International Conference on Supercomputers, Santa Clara, CA May 1987
6. Strang G, *Linera Algebra and its Applications* Academic Press, 1976
7. Vrsalovic D, Gehringer E, Segall Z and Siewiorek D, *The Influence of Parallel Decompositiopn Strategies on the Performance of Multiprocessor Systems* Proceedings 12th Ann. Int. Symposium on Computer Architecture, Boston June 1985
8. Whelan M, Gao G, Yum T.K., *Decomposition of Matrix Multiplication for Computation on Multiprocessor Architectures*, TR-87-044, Philips Laboratories, Briarcliff Manor, NY

Performance Analysis of an Optimistic Concurrency Control Algorithm in Replicated Database Systems

Mukesh Singhal
Dept. of Computer & Information Science
The Ohio State University
Columbus, OH 43210

Abstract: In this paper, we present an analysis of a concurrency control algorithm for replicated database systems. We present a model of distributed database systems, which provides a framework to study the performance of different concurrency control algorithms and use the model in the analysis of a concurrency control algorithm. We show that even after making some assumptions, detailed performance model of a concurrency control algorithm is so complicated that it is impossible to find its closed-form solution. We circumvent this problem by making two assumptions: first, we assume that the state of a site is statistically independent of the state of other sites, which permits us to analyze a single site rather than the whole system. Second, we assume that an update sees the average state of the system and all the updates exhibit the average steady-state behavior, which permits us to work with averages rather than with probability distributions. Therefore, the technique used in the analysis is approximate and iterative.

1. Introduction

In this paper, we analyze the performance of an optimistic concurrency control algorithm for replicated database systems. In optimistic concurrency control algorithms, an update is executed concurrently with other updates without performing any synchronization. However, before its computed values are written into the database, it enters a *validation phase* which determines if the update has conflicted with any concurrent update. In case of conflicts, lower priority update is aborted, else its computed values are written into the database. Depending upon the way intersite communication and validation phase are carried out, several optimistic algorithms have appeared in the literature. The optimistic algorithm which is analyzed in this paper functions in the following way: A site S_i maintains three queues — $SuspendQ_i$ which contains local updates which can not be executed due to conflicts with local updates awaiting the results of their validation; $LocalQ_i$ which contains local updates that have been tentatively executed and are awaiting the results of validation; $RemoteQ_i$ which contains remote updates that are awaiting the results of their validation.

When a site S_i receives a user request to execute an update U, it performs the following sequence of actions: if there is no entry in $LocalQ_i$ that has w-r conflict with U, then it assigns U a timestamp TS(U), places an entry for U into $LocalQ_i$, tentatively executes U, and sends out *validation(write_set(U), TS(U))* messages to all other sites. When a site S_j, $i \neq j$, receives this message, it places (write_set(U), TS(U)) into $RemoteQ_j$, updates its clock, and returns a *reply* message to S_i which contains the current timestamp of S_j. After S_i has received a message with timestamp larger than TS(U) from all other sites (called condition R1), it checks if there is an entry in $RemoteQ_i$ which has w-r conflict with U and has smaller timestamp than TS(U). (Negation of this condition is denoted by R2.)

If condition R2 passes, then U is committed: site S_i writes the computed values in its database copy and sends these values in *update-commit* messages to all other sites. When a site S_j receives these values, it writes them into its database copy, discards the corresponding entry from $RemoteQ_j$, and aborts all local updates from $LocalQ_j$ which have r-w conflict with U. If condition R2 fails, then U is aborted, its entry is removed from $LocalQ_i$, and *update-abort* messages are sent to all sites. On the receipt of this message, a site removes the entry for U from its RemoteQ. A site executes write action of an update using Thomas-Write rule (TWR) [4] where write to a data object is ignored if it has already been written by an update of higher timestamp.

2. Performance Model

We model arrivals of updates by a Poisson process. This is justified because a database is usually shared by a large number of independent users. We assume that the size of the readset and the writeset of updates are identically and independently distributed random variables with Geometric distribution. This assumption is justified because the updates that reference a small number of data objects occur more frequently than the updates that reference a large number of data objects. We assume that access to data objects is uniformly distributed across the entire database, i.e., every data object is accessed with equal likelihood.

A computer system, or site, provides the users with the facilities such as CPU for processing and main memory and secondary memory (usually disk) for storage capability. Execution of an update requires service at CPU for queue manipulation, update computations, message handling, etc. Secondary memory holds the data objects of a database, and in practice, it may consists of several disks. For simplicity, we assume only one disk in the model. An update makes a disk access for reading data objects or for writing computed values into the database. Disk service time depends upon seek-time, rotational delay, and data-transfer delay. We assume that the disk service time is exponentially distributed and the disk serves requests in FCFS order. Usually, concurrency control algorithms require data structures such as queues, lock tables, graphs, etc. We assume that the main memory at each site is large enough to hold these data structures.

Communication medium affects the performance by introducing finite delay in every intersite communication. This delay is modeled by an infinite server, i.e., the communication medium serves all intersite communication in parallel. In practice, this usually holds in store and for-

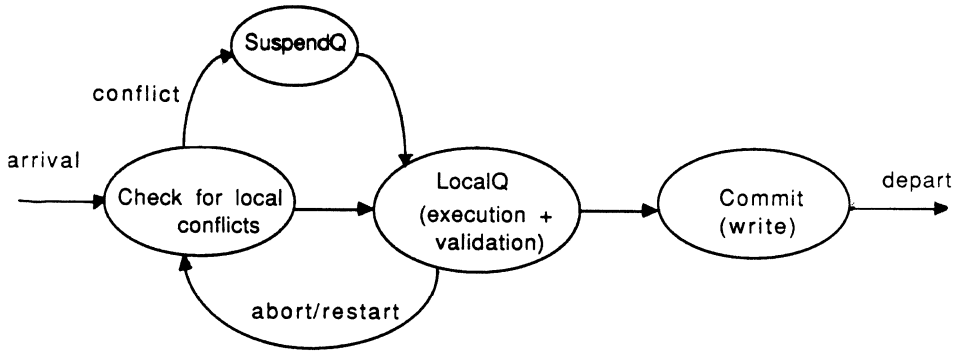


Figure 1. Flow diagram of the life-cycle of an update

ward networks. To simplify the model, we assume that the service time of the communication medium (i.e., time taken by a message to travel from one site to another site) is constant.

Parameters of the Model

(1) N: The number of sites in the system.

(2) λ: Update arrival rate which is the rate of update arrivals at a site. (Poisson distributed with parameter λ.)

(3) rs/ws: Update size which is the average number of data objects in the readset/writeset of updates. (Geometrically distributed with mean rs/ws.)

(4) M: Size of database which is the number of data objects in each copy of the database.

(5) T: Message propagation time which is the average time taken by a message to propagate through the communication medium.

(6) $1/\mu$: Disk access time which is the average time to read data objects in the readset or write data objects in the writeset of an update. (Exponentially distributed with parameter μ.)

Time spent by CPU in comparing timestamps, manipulating queues, computing updates, and processing messages is usually much smaller than disk access time and message propagation time. Therefore, we neglect CPU in the performance analysis.

In the performance study, we will be interested in computing *update response time* which is the time interval between the instants when an update is submitted by a user to a site and when the update is completely executed at that site.

3. Performance Analysis of the Algorithm

The life-cycle of an update is depicted in figure 1. When an update arrives, it is diverted to SuspendQ if its host site already has a conflicting entry in its LocalQ. Otherwise, the update proceeds with its tentative execution where it performs its read and compute operations and performs all its writes on a temporary storage. Then the site performs validation for the update by exchanging messages. If the update fails the validation, it aborts and restarts, else it commits its writes to the database at all the sites using TWR.

3.1. Difficulties in Analyzing the System

Computation of update response time requires computation of the probability of going to SuspendQ (denoted by P_{sus}), average wait time in SuspendQ, average wait in LocalQ, the probability of passing validation (denoted by P_{pass}), and time taken to commit the writes. Computation of P_{sus} and P_{pass} for an update requires detailed knowledge of the system state when these decisions/checks are made. For example, to determine P_{sus} for an update, we should know the exact data requirements for each update in LocalQ; likewise, we should know the exact data requirements of all global updates to determine P_{pass}. We can model such a system by a Markov chain by making appropriate assumptions about the probability distribution of service rates. However, the Markov chain will have such a large number of states and such a complex structure that it will not be feasible to analyze it even for a database system of small size.

3.2. Approach Taken

We handle the state space explosion by making two approximations. First, we assume that the state of a site is statistically independent of the states of other sites. As a result, we can analyze each site in isolation and the effect of other sites on a site can be reflected by write/update activity due to other sites. Since we assume that the system is homogeneous, on the average all sites will perform identically and performance measures can be obtained by analyzing only one site. (Note that it is easier to analyze one site rather than the entire system.) Such decomposition technique has already been applied in the analysis of load balancing algorithms [1], concurrency control algorithms [3], multiple access protocols and store-and-forward packet switching networks [2].

After making appropriate assumptions about the probability distribution of service rates, we can model a single site by a Markov chain whose states are detailed enough to capture concurrency control activities at that site, e.g., number of running/blocked transactions, data objects held/acquired by them, etc. It is not difficult to see that even for a database system of small size, Markov chain for a site will have such a huge state space with very complex structrure that obtaining its closed-form solution will be practically impossible. Here we make the second approximation: rather than working with the probability distributions, we work with the averages — we assume that an update sees the average state of the system and all updates exhibit average behavior.

We exploit the interdependence among the variables to derive a set of equations and solve it using an iterative technique. Initially, we assume that the probability of an update restart is zero, or very small, and compute waiting

time in different stages of an update. From these waiting times, we estimate new probability of an update restart which in turn gives new waiting times. This process is repeated until the difference in the value of a variable of interest (e.g., the probability of restart or wait time) between two successive iterations is less than a desired value.

The probability that two independently selected groups of data objects of size a and b out of the M data objects have a conflict is (denoted by $\Phi(a, b)$):

$\Phi(a, b)$ = 1 - Prob(there is no data object common between the groups)

$$= 1 - \binom{M-a}{b} / \binom{M}{b} \simeq a*b/M.$$

This result is used heavily in the analysis.

3.3. Performance Analysis

An update may undergo execute-restart cycle several times before it actually commits. If the average waits in SuspendQ and LocalQ are W_{sus} and W_{loc}, respectively, then the average duration of an execute-restart cycle, R, is:

$$R = P_{sus}*W_{sus} + W_{loc} \quad ...(1)$$

We assume that the probability of an update restart is independent of the probabilities of its earlier restarts. Therefore, the number of restarts an update undergoes before it commits is Geometrically distributed. If the probability of an update restart is P_{res} (= 1 - P_{pass}) and the average time to commit writes of an update is W_{write}, then the update response time is,

$$Resp = \left[(1-P_{res})*R + P_{res}(1-P_{res})*2R + \ldots.\right] + W_{write}$$

$$Resp = \frac{R}{(1-P_{res})} + W_{write} \quad ...(2)$$

If on the average there are N_{loc} entries in LocalQ, then an arriving update will go to SuspendQ if it intends to read any data objects which belongs to the writeset of any of the N_{loc} updates; that is, there is a data object common between rs and N_{loc}*ws data objects. The probability of this happening is,

$$P_{sus} = \Phi(rs, N_{loc}*ws) = \frac{N_{loc}*ws*rs}{M} \quad ...(3)$$

The effective rate of update arrival at a site is $\lambda/(1-P_{res})$ because aborted updates are retried. Therefore, the average number of entries in LocalQ is (Little's law),

$$N_{loc} = \lambda/(1-P_{res})*W_{loc} \quad ...(4)$$

Computation of W_{loc}

An entry stays into LocalQ until it is aborted by a committed entry in RemoteQ (the committing entry has w-r conflict with the aborted update and has priority over it) or until condition R1 holds for it (at that instant, it may commit or abort depending upon whether it passes condition R2 or not). If there are N_{rem} entries in RemoteQ on the average, then $N_{rem}*(1-P_{res})$ of them will commit on the average and can potentially abort entries in LocalQ. Since a committing entry in RemoteQ aborts all the entries in LocalQ which have r-w conflict with it, the probability that an entry in LocalQ is aborted before condition R1 holds for it is,

$$P_1 = \Phi(rs, N_{rem}*(1-P_{res})*ws) = \frac{N_{rem}*(1-P_{res})*ws*rs}{M} \quad ...(5)$$

For an entry in LocalQ, condition R1 holds for it after a delay of 2T after it has been placed in LocalQ. An entry can get aborted at *any instant* with equal probability before R1 holds for it (an assumption). Therefore, on the average an entry stays in LocalQ for duration T given that it gets aborted before R1 holds for it. If an entry is not aborted before R1 holds for it (this happens with the probability 1-P_1), it may get aborted after R1 holds for it provided there is a smaller timestamp entry in RemoteQ which has w-r conflict with it (i.e., R2 does not hold). Since entries arrive in RemoteQ at a rate $(N-1)\lambda/(1-P_{res})$, on the average $2T*(N-1)\lambda/(1-P_{res})$ entries in RemoteQ can cause an update to abort due to R2 (only if they commit and have w-r conflict with this update). The probability of this happening is:

P_2 = Prob(R2 does not hold for Ul U is not aborted before R1 holds for it)

$$= \Phi(rs, \frac{2T(N-1)\lambda*ws}{(1-P_{res})}) = \frac{2T(N-1)\lambda*ws*rs}{(1-P_{res})M}$$

The update is committed if condition R2 also holds for it when condition R1 holds for it. The probability of this happening is:

$P_{pass} \equiv 1 - P_{res}$

= Prob(R2 holds for Ul U is not aborted before R1 holds for it)* Prob(U is not aborted before R1 holds for it)

$=(1-P_2)*(1-P_1)$

$$= \left[1 - \frac{2T(N-1)\lambda*ws*rs}{(1-P_{res})M}\right]*(1-P_1) \quad ...(6)$$

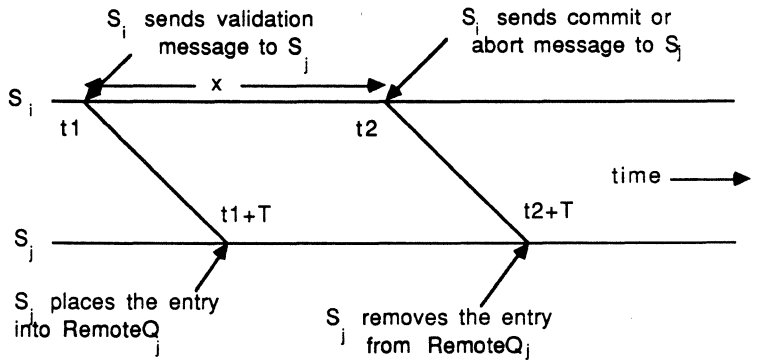


Figure 2.

Irrespective of the outcome of R2 check, the update on the average waits for 2T time units. Also, an entry waits in LocalQ until it is aborted before R1 holds for it or after R1 holds for it. Therefore,

W_{loc} = E[wait in LocalQ | update is aborted before R1 holds]* Prob(update is aborted before R1 holds) + E[wait in LocalQlupdate is not aborted before R1 holds]* Prob(update is not aborted before R1 holds)

$$W_{loc}=T*P_1+2T*(1-P_1) \quad ...(7)$$

Computation of W_{rem}

If on the average an entry waits in RemoteQ for W_{rem} time units, then from Little's law (note that entries arrive in RemoteQ at a rate $(N-1)\lambda/(1-P_{res})$):

$$N_{rem}=W_{rem}*(N-1)\lambda/(1-P_{res}) \quad ...(8)$$

Next, we compute the average wait in RemoteQ, W_{rem}. To determine W_{rem}, we first establish a relationship between the time interval entries for an update stay in LocalQ of its host site and RemoteQ of a remote site. Consider the scenario shown in figure 2. At instant t1 site S_i sends out a *validation* message for an update to site S_j which reaches S_j at instant t1+T and site S_j then places an entry for the update in $RemoteQ_j$.

At instant t2, S_i commits or aborts the update and sends the corresponding message to S_j. S_j receives the message at instant t2+T and removes the corresponding entry from $RemoteQ_j$. (Interval x, for which the entry stays in $LocalQ_i$, depends upon whether the update was aborted before condition R1 held for it or not.) Note that the duration for which the corresponding entry stays in $RemoteQ_j$, (t2+T) - (t1+T) = x, is the same as the duration for which the corresponding entry stays in $LocalQ_i$. Therefore, for an update, its entries stay in LocalQ and RemoteQs for the same amount of time and

$$W_{rem}\equiv W_{loc}=P_1T+2T(1-P_1) \quad ...(9)$$

Computation of W_{sus}

An entry in SuspendQ is checked for the possibility of getting unblocked whenever an entry departs (commits/aborts) from LocalQ. The average interdeparture time of updates at a site, Γ, is $(1-P_{res})/\lambda$ because update departure rate is the same as the effective arrival rate, $\lambda/(1-P_{res})$, in the equilibrium. Since a departing update releases ws data object on the average, the probability that a data object previously unavailable to an update in SuspendQ, now becomes available is $ws/(N_{loc}*ws) = 1/N_{loc}$. If the average conflict size ‡ is cs, then the probability that an update in SuspendQ gets unblocked when an update departs from LocalQ is $p= (1/N_{loc})^{cs}$ (which is the probability that all the conflict-causing data objects are freed by the departing update). If we assume that the probability of an update in SuspendQ getting unblocked is independent for different departures of entries in LocalQ, then the number of departures before an update in SuspendQ gets unblocked follows a Geometric distribution with parameter p. As a result, the average wait in SuspendQ is,

$$W_{sus} = \frac{\Gamma}{p} = \frac{(N_{loc})^{cs}(1-P_{res})}{\lambda} \quad ...(10)$$

‡ Conflict size is the number of conflict creating data objects; i.e., the data objects which are common in two conflicting updates.

Computation of W_{write}

Data objects at a site are stored on a secondary storage device, say a disk. Therefore, an access to the disk is made when read and write actions of updates are executed. Note that the read and the write actions arrive the disk at rates $\lambda/(1-P_{res})$ and $N\lambda$, respectively. If we assume that the read and the write requests arrive at the disk according to Poisson distribution with parameter $\lambda' = \lambda/(1-P_{res}) + N\lambda$ and the disk service time is exponentially distributed with parameter μ for all these requests, then the average response time at the disk is (M/M/1 server),

$$T_{disk}=\frac{1}{\mu-\lambda'}\equiv W_{write} \quad ...(11)$$

We obtain the performance of the algorithm by solving the system of equations in the following manner: we start with small values for waiting time in LocalQ and RemoteQ and compute the probabilities of update restarts - P_{res} and P_1 - using equations (3), (4), (5), (6), (7), and (8). We use these values in equations (7) and (9) to compute new values for waiting times - W_{loc} and W_{rem}. This process is repeated until difference of waiting times between two successive iterations is less than small quantity, say 5 percent. Finally, the average update response time is computed using equations (1), (2), (3), (10), and (11).

4. Concluding Remarks

We have presented a performance model of distributed database systems and used it to analyze the performance of an optimistic concurrency control algorithm in replicated database systems. We have shown that even after making some simplifying assumptions, detailed performance model of a concurrency control algorithm is so complicated that it is impossible to find its closed-form solution. We have solved this problem by assuming (i) that the state of a site is statistically independent of the state of other sites, which permits us to analyze a single site rather than analyzing the whole system and (ii) that an update sees the average state of the system and all the updates exhibit the average steady-state behavior, which permits us to work with averages rather than with probability distributions. The performance analysis not only provides a simple and quick method to compute the performance measures of an optimistic concurrency control algorithm but also it gives an approximate method to analyze other distributed concurrency control algorithms.

References

1. EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J., "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions of Software Engineering*, pp. 662-675 (May 1986).
2. LAM, S. S., "Store-and-Forward Buffer Requirements in a Packet Switching Network," *IEEE Transactions on Communications*, pp. 394-403 (April 1976).
3. SINGHAL, MUKESH AND AGRAWALA, A. K., "Performance Analysis of an Algorithm for Concurrency Control in Replicated Database Systems," *Performance Evaluation Review, Vol. 14, No. 1*, (May 1986).
4. THOMAS, R. H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. of Database Systems*, pp. 180-209 (June 1979).

Hypercube Algorithms for Some String Comparison Problems*

Oscar H. Ibarra, Ting-Chuen Pong and Stephen M. Sohn

Department of Computer Science
University of Minnesota
Minneapolis, Minnesota 55455

Abstract

We give parallel algorithms for solving some string comparison problems on the hypercube. For strings x and y with length(x) = m, length(y) = n, and assuming $n \geq m$, we show the following: the string edit problem, the longest common subsequence problem and the minimum-length time-warping problem can be solved in $O(\log^2 n)$ time using $O(1)$ space per processing element (PE) on an SIMD hypercube of $O(n^3/\log^2 n)$ PE's. We also show that the longest common substring problem can be solved in $O(\log n)$ time using $O(1)$ space per PE on an SIMD hypercube of $O(n^2)$ PE's. Finally we show that the substring problem (where typically m is much smaller than n) can be solved in $O(m+\log n)$ time using $O(m)$ space per PE on an MIMD hypercube of $O(n/m)$ PE's. We note that this algorithm has an optimal processor-time product if $m = \Omega(\log n)$. The results of implementing this algorithm on the NCUBE/7 hypercube machine are also presented.

1 Introduction

String comparison techniques are important in many diverse fields such as text processing, image and signal processing, pattern recognition and artificial intelligence [20]. The *string edit problem* is to compute the minimum cost of transforming one string into another string using the edit operations insert, delete and replace [22]. Each edit operation has an associated cost which is a function of the alphabet from which the strings were composed. The cumulative cost is a measure of the dissimilarity or distance between two strings and is called the *edit distance* or the *weighted Levenshtein distance* [18]. The *longest common subsequence problem* (LCS) is to determine the length of the longest subsequence (not necessarily a contiguous substring) common to both strings [22]. The *minimum-length time-warping problem* has particular importance in the domain of speech processing and is concerned with matching strings based on compression and expansion [20]. Many other sequence comparison problems may be found in [20].

The three problems just mentioned may be broadly characterized as *inexact string matching problems*. A slightly less difficult class of problems concerns finding only exact matches between strings or substrings. The *substring problem* asks whether a shorter string (called the *pattern*) is contained in a longer string (called the *text*) [16].The *longest common substring problem* (LCG) is to find the longest substring that is common to both strings [20].

2 The String Edit Problem

The string edit problem is formally defined as follows [22]: let $x = x_1 x_2 \ldots x_m$ and $y = y_1 y_2 \ldots y_n$ be strings over a fixed finite alphabet. We are given the edit operations insert, delete and replace in order to transform x into y. Associated with these operations are costs I_a, D_b, and $R_{a,b}$, respectively, for all symbols a and b in the alphabet. The minimum cumulative cost (called the *edit distance*) can be found using dynamic programming: let M be a $(m+1) \times (n+1)$ table such that $M(i,j)$ is the minimum cost of transforming $x_1 \ldots x_i$ into $y_1 \ldots y_j$. The table entries can be computed using the recurrence:

$$\begin{aligned} M(0,0) &= 0; \\ M(i,0) &= \sum_{r=1}^{i} D_{x_r}; \\ M(0,j) &= \sum_{r=1}^{j} I_{y_r}; \end{aligned}$$

$$M(i,j) = \min \begin{cases} M(i-1,j-1) + R_{x_i,y_j}, \\ M(i-1,j) + D_{x_i}, \\ M(i,j-1) + I_{y_j}. \end{cases}$$

The longest common subsequence problem is a special case of the string edit problem using the following cost function [22]:

$$I_a = D_a = 1;$$

$$R_{a,b} = \begin{cases} 0 & \text{if } a = b, \\ 2 & \text{otherwise} \end{cases}$$

for all symbols a and b in the alphabet. The minimum-length time-warping problem is solved by restricting the form of the recurrence as well as the cost function (also, the table is $m \times n$) [20]:

$$\begin{aligned} M(0,0) &= 0; \\ M(i,j) &= \min \begin{cases} M(i-1,j-1) + \tau R_{x_i,y_j}, \\ M(i-1,j) + 1/2\tau R_{x_i,y_j}, \\ M(i,j-1) + 1/2\tau R_{x_i,y_j}. \end{cases} \end{aligned}$$

Now suppose that $n \geq m$. It has been observed that virtually all sequence comparison problems are variants of the string edit problem and hence can be solved by the same dynamic programming technique [18]. This approach takes $O(mn)$ time sequentially [22]. An asymptotically faster algorithm using a divide-and-conquer approach taking $O(mn/\min\{m, \log n\})$ time was given in [19] (All logarithms mentioned in this paper are base two.) In [18] it was shown that the edit distance can be computed in parallel on a two-dimensional systolic array in $O(n)$ time using mn processing elements (PE's). An algorithm yielding the actual edit sequence in $O(n)$ time using a one-way two-dimensional iterative array of mn PE's was given in [5]. The number of PE's can be reduced to n using a one-way one-dimensional systolic array with the same time bound of $O(n)$ [6, 12]. A similar algorithm with these same bounds but using an SIMD parallel machine that can simulate a linear array was given in [8]. On a bus automaton the LCS can be computed in constant time [3].

Our algorithm is essentially a parallelization of the dynamic programming technique, as are the parallel algorithms just noted. We will first indicate how to reformulate the recurrence relation as a weighted directed graph. Let us define a family $\{G_n\}$, $n \geq 1$, of weighted directed graphs as follows: $G_n = \langle V, E \rangle$ where

$$V = \{(i,j) \mid 0 \leq i, j \leq n-1\},$$

$$E = \{(i,j) \rightarrow (k,l) \mid k = i+1 \text{ or } l = j+1 \text{ or both, for } 0 \leq i,j,k,l \leq n-1\}.$$

Associated with each edge is a weight $c_{(i,j)\rightarrow(k,l)}$. The graph G_4 is illustrated in Figure 1.

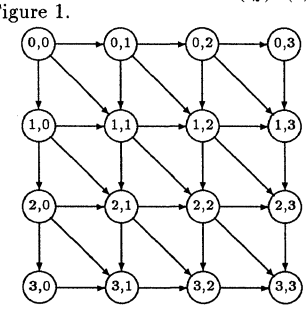


Figure 1: The graph G_4.

*This research was supported in part by NSF Grants DCR-8420935, DCR-8604603 and ECS-8505662.

Now focus on the string edit problem since the other two problems are subproblems and hence solvable using this same technique. Edge weights are assigned to the graph G_{n+1} as follows:

$$c_{(i,j)\to(k,l)} = \begin{cases} I_{y_l} & \text{for } i = k, l = j+1, \\ D_{x_k} & \text{for } k = i+1, j = l, \\ R_{x_k,y_l} & \text{for } k = i+1, l = j+1 \end{cases}$$

for $0 \le i,k \le m$, $0 \le j,l \le n$. If $m \ne n$ then the remaining edges are weighted with $+\infty$. It is not difficult to see that the edit distance between x and y is equal to the length of the shortest path from vertex $(0,0)$ to vertex (m,n). Furthermore, the sequence of edges comprising this shortest path correspond to the optimal sequence of edit operations used.

The algorithm we give uses divide-and-conquer to decompose the problem. Even though only one path is ultimately desired, it will be necessary to compute many paths in parallel at early stages of the algorithm. The *perimeter-pairs shortest path problem* for this family of graphs is to determine the lengths of the shortest (directed) paths between all pairs of vertices lying on the perimeter of G_n.

2.1 Overall Scheme

Initially the n^2 vertices of a given graph G_n are partitioned into n^2 subgraphs, called *blocks*, one vertex per block. These blocks are then merged pairwise to form a collection of $n^2/2$ blocks, each containing two vertices and the edge that connects them. We then solve the perimeter-pairs problem for each of these blocks taken individually, which is trivial for these blocks. The resulting blocks are then merged pairwise again and the perimeter-paths problem is solved for these enlarged blocks. This procedure is repeated until one block containing all of the graph remains. At this point, we have solved the problem for G_n. Notice that at a given level of the merging, the perimeter-pairs problem is being solved for disjoint subgraphs. Hence it can be performed on every block in parallel. Also, it is not difficult to solve the perimeter-pairs problem for a newly merged block if we previously have solved the problem for the two component blocks.

The initial partitioning and subsequent merging must preserve the topology of the graph. In other words, we must merge subgraphs that are adjacent in the graph. For this family of graphs there is a natural and symmetric merging procedure that obeys this constraint. For $k = 2^i$, $0 \le i \le \log n - 1$, we alternately merge $k \times k$ blocks pairwise horizontally and then merge the resulting $k \times 2k$ blocks pairwise vertically. This is illustrated in Figure 2 for the graph G_4.

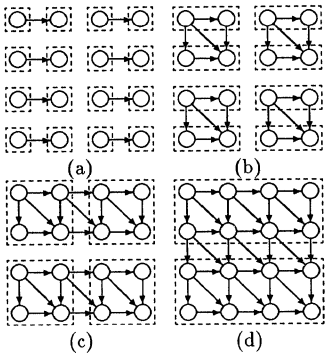


Figure 2: (a) First, (b) second, (c) third, and (d) fourth stages of merging for the graph G_4.

2.2 A Graph-theoretical Discussion

Consider the graph G_n where n is a power of 2. Now focus on a given stage where we are merging two $k \times k$ blocks horizontally. Denote the left-hand block as the A-block and the right-hand block as the B-block. Also denote the resulting adjoined block as the AB-block. The perimeter pairs problem has previously been solved for these $k \times k$ blocks at earlier stages of the procedure. Let P_A be the set of vertices lying on the perimeter of the A-block and let P_B be similarly defined. P_{AB} is the set of vertices that lie on the perimeter of the adjoined AB-block. Denote by $lsp(u,v)$ the length of the shortest path between vertex u and vertex v, or $+\infty$ if there is no path from u to v. We can define the lengths of the perimeter-pairs paths for the A-block as a set of triples $A = \{(u,v,lsp(u,v)) \mid u,v \in P_A\}$. B (AB) is similarly defined for the B-block (AB-block). Let D be the edges going directly from the A-block to the B-block.

In order to compute AB we need only A, B and D. Initially, we will indicate how to compute $\widehat{AB} = \{(u,v,lsp(u,v)) \mid u,v \in (P_A \cup P_B)\}$ from which AB is easily obtained. $\widehat{AB}$ can be written as the union of four disjoint sets: $\widehat{AB} = A \cup B \cup A \rightsquigarrow B \cup B \rightsquigarrow A$ where

$$A \rightsquigarrow B = \{(u,v,lsp(u,v)) \mid u \in P_A, v \in P_B\},$$

$$B \rightsquigarrow A = \{(u,v,lsp(u,v)) \mid u \in P_B, v \in P_A\}.$$

We can immediately simplify $B \rightsquigarrow A = \{(u,v,+\infty) \mid u \in P_B, v \in P_A\}$.

Consider splitting the edges of D and inserting a unique "imaginary" vertex within each edge as follows: for each $(r \to s) \in D$ define two new weighted edges $(r \to w_s^r)$ and $(w_s^r \to s)$ where w_s^r is a new vertex that now lies directly between r and s. Let $I = \{w_s^r \mid (r \to s) \in D\}$ be this set of imaginary vertices. Furthermore, define the weights of these new edges as $c_{r \to w_s^r} = c_{r \to s}$ and $c_{w_s^r \to s} = 0$. This is illustrated in Figure 3.

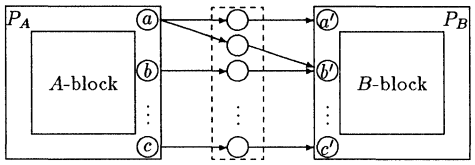


Figure 3: The set of imaginary vertices lies between the A-block and the B-block.

The motivation for this construction is to write an expression for $A \rightsquigarrow B$ in terms of the shortest paths to and from the imaginary vertices:

$$A \rightsquigarrow B = \left\{ \left(u, v, \min_{w \in I} \{lsp(u,w) + lsp(w,v)\} \right) \mid u \in P_A, v \in P_B \right\}.$$

$A \rightsquigarrow B$ can be thought of as follows: the shortest path from $u \in P_A$ to $v \in P_B$ must include exactly one edge of D. Due to the construction of the imaginary vertices and new edges given above, we can equivalently state that the shortest path from $u \in P_A$ to $v \in P_B$ must pass through exactly one vertex $w \in I$. Also, because the path from u to v is the shortest path, the paths from u to w and from w to v must also be shortest paths. Therefore, having computed these component paths $A \rightsquigarrow I$ and $I \rightsquigarrow B$, they can be merged to yield the desired $A \rightsquigarrow B$. The case where two $k \times 2k$ blocks are merged vertically is handled in a similar manner.

2.3 Using the Hypercube

Dekel, Nassimi and Sahni [7] have given a matrix multiplication algorithm for the hypercube. It was shown that two $n \times n$ matrices can be multiplied in $O(n/p + \log p)$ time when $n^2 p$, $1 \le p \le n$ PE's are available. Conceptually, they treat the hypercube as an $p \times n \times n$ array. We refer to this virtual configuration as the *cube* configuration, as distinguished from the actual physical configuration of the hypercube.

The sets mentioned above are implemented as matrices which are then manipulated via matrix operations with suitable transform matrices. It is difficult to describe hypercube parallel operations in terms of atomic PE operations. Our algorithm is written as a sequence of matrix operations that are successively applied to the input (the edge costs). This algorithmic representation is a straightforward and simple way to express a parallel algorithm. The matrix of path costs for a merged block of vertices is formed by computing the "product" of the constituent path cost matrices, where the "product" operation is obtained by replacing the multiplication and addition operations of ordinary matrix multiplication with the addition and minimum operations, respectively. The hypercube is partitioned into *sub-cubes* and the merging of the path cost matrices is done in parallel for the particular block of vertices assigned to each sub-cube. The sub-cubes themselves are then coalesced to form larger sub-cubes for the next level of merging. For the sake of brevity we omit further details of the algorithm,

which can be found in [13]. It can be shown that the overall time needed is $O(\log^2 n)$ for a hypercube of $O(n^3/log^2 n)$ PE's using $O(1)$ space per PE (a detailed analysis is given in [13]). Furthermore, the actual sequence of edges (corresponding to the edit sequence) can be found in the same time and space by a parallel divide-and-conquer search through the intermediate path cost matrices.

3 The Longest Common Substring Problem

A related but distinct measure of string distance is the length of the *longest common substring* (LCG) [20]. Let x and y be strings over a fixed finite alphabet, $|x| = m$, $|y| = n$ and $n \geq m$. (Henceforth we abbreviate length$(x) = |x|$.) There is a straightforward sequential method that is obtained by considering all possible alignments between the two strings which takes $O(mn)$ time [20]. Subsequently this time bound was improved to $O((m+n)\log(m+n))$ [15] and finally to the optimal bound $O(m+n)$ [23] (see also [1]). We give a parallel adaptation of the straightforward method where the actual substring satisfying the LCG constraint as well as its length are obtained. On an SIMD hypercube of size $O(n^2)$, we show that the LCG can be found in $O(\log n)$ time using $O(1)$ space per processing element. Consider the following definitions:

Definition 1 *Given strings x and r_x, r_x is said to be a* rotation *of x if there exist strings u and v such that $r_x = uv$ and $x = vu$.*

Definition 2 *Given strings $x = x_1x_2\ldots x_n$ and $y = y_1y_2\ldots y_n$, the* aligned longest common substring *(ALCG) of x and y is the LCG u of x and y with the additional constraint that $u = x_ix_{i+1}\ldots x_k = y_iy_{i+1}\ldots y_k$ for some $1 \leq i \leq k \leq n$.*

Now suppose that we wish to find the LCG of x and y, where $n = |y| \geq |x|$. Let "#" and "$" be two distinct symbols not occurring in the alphabet of x and y. Further suppose that we already have a function ALCG(u, v) that returns the ALCG of u and v for strings such that $|u| = |v|$. The following is a reformulation of the straightforward algorithm using the terms just defined.

```
input(x, y);
u := v := ε; //ε is the empty string//
y' := y#;
x' := x$^{|y'|-|x|};
for all rotations r_{y'} of y' do begin
    u := ALCG(x', r_{y'});
    if |u| > |v| then
        v := u
end;
output(v);
```

Definition 3 *Given strings $x = x_1x_2\ldots x_n$, $y = y_1y_2\ldots y_n$ and $c = c_1c_2\ldots c_n$, c is said to be the* characteristic matching string *of x and y if*

$$c_i = \begin{cases} 1 & \text{if } x_i = y_i, \\ 0 & \text{otherwise.} \end{cases}$$

Now note that the ALCG(x, y) function returns the substring of x and y corresponding to the longest substring of 1's in the characteristic matching string of x and y.

Again suppose that we are given strings x and y, $n = |y\#| \geq |x|$. Also suppose that n is a power of 2. If n is not a power of 2 then y can be padded on the right end with additional "#"s to form the string $y\#^{2^{\lceil \log n \rceil} - n}$. From now on we will denote the padded string as y. We pad the right end of x to yield the string $x\$^{2^{\lceil \log n \rceil} - |x|}$. Again, from now on we will denote the padded string as x. We will need a hypercube of n^2 PE's. The hypercube can be conceptualized as a two-dimensional array where each row and column of the array is itself connected as a hypercube (e.g., see [4]). In this configuration we can alternately describe a PE by either its row-major array coordinates (i, j), $1 \leq i, j \leq n$, or its binary address in the overall hypercube. The mapping from coordinate form to binary index form is: PE (i, j) has index $(i-1)\cdot n + (j-1)$. Initially suppose that x and y are stored in the hypercube such that PE $(1, j)$ contains both x_j and y_j, $1 \leq j \leq n$. The algorithm can be divided into four phases:

Phase 1 We wish to store the symbols x_j and y_j in PE (i, j), $1 \leq i, j \leq n$. It is well known that a datum can be broadcast to all nodes of a hypercube of size n in exactly $\log n$ time [7]. Each column of the conceptual array is a hypercube of size n and x_j and y_j are initially stored in PE $(1, j)$. Hence, the broadcasting is done in parallel on the n columns with the result that PE (i, j) holds x_j and y_j after $O(\log n)$ time.

Phase 2 String y is now circularly shifted $(i-1)$ places to the right along row i, $1 \leq i \leq n$. The result is that y_j is now stored in PE $(i, (j+i-1 \bmod n)+1)$ in row i. This shifting is done on every row in parallel. Each row of the conceptual array is a hypercube of size n. It has been shown that circular shifting of k positions for any $0 \leq k \leq n-1$ can be performed in $O(\log n)$ time on a hypercube of size n [17, 7]. Hence Phase 2 takes $O(\log n)$ time overall.

Phase 3 In Phase 2, all n possible rotations of y have been generated, one unique rotation per row. The ALCG will now be computed in parallel for each row i using x and $r_y^{(i)}$, where $r_y^{(i)}$ is the rotation of y on row i. The output of this phase will be the initial index (with respect to x) and length of the ALCG solution for each row. These values will be stored in PE's $(i, 1)$, $1 \leq i \leq n$, corresponding to the rows. Let us assume that each PE has stored its column index. We will focus on the ALCG computation for a particular row i but the following procedure will be performed in parallel on every row $1 \leq i \leq n$. Initially the characteristic matching string c is formed for x and $r_y^{(i)}$, one symbol per PE. This is easily done in parallel in $O(1)$ time by comparing x_j and $r_{y_j}^{(i)}$ at PE (i, j) and setting some register to 1 if $x_j = r_{y_j}^{(i)}$, otherwise 0. Therefore, the problem is reduced to finding the initial index and length of the longest substring of 1's in the string c, which is distributed over the n PE's of row i one symbol per PE. Since this subproblem is of independent interest we will state it as a lemma.

Lemma 1 *Given a string x composed of 0's and 1's, $|x| = n$, the length of the longest substring of 1's in x can be found in $O(\log n)$ time using $O(1)$ space per PE on an SIMD hypercube of size $O(n)$. Furthermore, the initial index of the satisfying substring also can be found in the same time, space and number of PE's if each PE has stored its hypercube index.* (A proof can be found in [13].)

Therefore, after computing the characteristic matching string, we can apply Lemma 1 with the result that the initial index and length of the ALCG of x and r_y^i is stored in PE $(i, 1)$, $1 \leq i \leq n$ after $O(\log n)$ time.

Phase 4 The overall solution to the LCG problem now is found by taking the maximum of the length values stored in PE $(i, 1)$, $1 \leq i \leq n$. The indices corresponding to the lengths are carried along in the computation. Since this column of PE's is a hypercube, the maximum can be computed in a straightforward way in $O(\log n)$ time [7].

4 The Substring Problem

Let x and y be strings over a fixed finite alphabet. The *substring problem* asks whether x is a substring of y. Typically the string x (called the *pattern*) is much shorter than the string y (called the *text*) [16]. Sequential solutions to the substring problem have been extensively studied. The straightforward algorithm consists of aligning the pattern starting at the beginning of the text and comparing symbols pairwise in order from left to right. If the pattern matches the text for this alignment then the occurrence is noted. Otherwise, the pattern is shifted one position to the right and the procedure repeated (see [1]). In the worst case this approach takes $O(mn)$ time. An optimal algorithm taking $O(m+n)$ time can be obtained by using the results of the previous partial match [16]. Some related results and extensions were given in [2, 10] while a general discussion of this technique can be found in [1].

Parallel algorithms have also been given. On a concurrent-read concurrent-write parallel random access machine (CRCW PRAM) of $O(n/\log n)$ processing elements (PE's), a time bound of $O(\log n)$ was obtained [9, 21]. We will show that on an MIMD hypercube of $O(n/m)$ PE's, the substring problem can be solved in $O(m + \log n)$ time using $O(m)$ space per PE. This algorithm possesses an optimal processor-time product if $m = \Omega(\log n)$, i.e., m is asymptotically greater than $\log n$. Our algorithm will use the sequential algorithm given in [16] as a subroutine. Therefore it is appropriate to consider the MIMD model since this subroutine can be performed asynchronously on each node in parallel.

4.1 The Algorithm

We need an MIMD hypercube of n/m PE's. (Without loss of generality assume that m evenly divides n; if not, then the text y can be padded on the right with some symbol not occurring in the alphabet of the pattern x and the text y.) Let $x = x_1x_2 \ldots x_m$ and $y = y_1y_2 \cdots y_n$. It is well known that a hypercube can optimally simulate a two-way one-dimensional array of processors (e.g., see [4]). This technique uses a Gray code to map the PE's $0, 1, \ldots, n/m-1$ such that PE i is directly connected to PE $i-1$ and $i+1$. Initially suppose that the pattern x is stored in PE 0 and the text y is distributed over the PE's such that PE i contains $y_{mi+1}y_{mi+2} \cdots y_{m(i+1)}$. The algorithm is divided into four phases:

Phase 1 We wish to broadcast the pattern x to all other PE's. A technique given in [11] uses an embedding of spanning trees in order to obtain the optimal time bound of $O(m + \log(n/m))$ for broadcasting m items to all PE's in a hypercube of size n/m. Each PE stores the pattern as it is received.

Phase 2 After Phase 1 PE i has stored the pattern x and a segment of the text $y_{mi+1}y_{mi+2} \cdots y_{m(i+1)}$ Now it is possible that the pattern occurs in the text overlapping a boundary imposed by this segmentation. This is alleviated by reading a portion of the text stored in PE $i+1$ and then searching for the pattern in this enlarged segment. Hence we want PE $i+1$ to send $y_{m(i+1)+1}y_{m(i+1)+2} \cdots y_{m(i+2)-1}$ to PE i, for all $i+1 \leq n/m-1$ in parallel. Using the one-dimensional array connections this can be done in $O(m)$ time in parallel.

Phase 3 After Phase 2 PE i, $i < n/m-1$, holds an enlarged segment of the text $y_{mi+1}y_{mi+2} \cdots y_{m(i+2)-1}$ which is $2m-1$ symbols long. PE $n/m-1$ has stored $y_{n-m+1}y_{n-m+2} \cdots y_n$. In parallel each PE now searches for x in its respective segment. Using the technique given in [16] this takes $O(m)$ time. The PE then notes whether or not the pattern was found.

Phase 4 In $O(\log(n/m))$ time the n/m PE's can be polled in parallel to see if any match occurred [7].

4.2 Implementation on the NCUBE/7

While the above algorithm has an optimal processor-time product for $O(n/m)$ PE's, in practice the size of a given hypercube machine is fixed. Hence, it is desirable to obtain an algorithm that yields an acceptable speed-up using a fixed number of PE's for a pattern and text of arbitrary size. With slight modifications, our algorithm was implemented on the NCUBE/7 hypercube machine using a maximum of 64 PE's. The overall time for a given run can be decomposed into two parts: communication and computation. The communication time consists of the time needed to broadcast the pattern to all PE's plus the time needed to send and receive the appropriate portions of text between neighboring PE's in the conceptual linear array. The computation time is the time needed to search for the pattern within the enlarged segment of text. The results are summarized in Figure 4 for a text of length 10,000 and a pattern of length 12. Additional experiments were performed to consider various ways of loading the text and pattern into the hypercube, as well as various searching techniques [14].

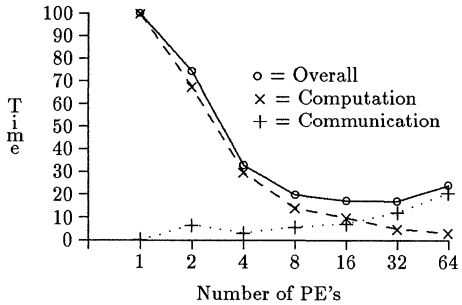


Figure 4: Normalized timing values for the NCUBE/7 implementation of the substring algorithm.

5 Summary

We have given parallel algorithms for solving some string comparison problems on the hypercube. The problems include both inexact and exact matching problems. Our algorithm for the inexact matching problems, of which the string edit problem is the most general representative, was presented in terms of a shortest path problem for a special family of graphs. The exact matching problems were solved in a more direct fashion. Our algorithms use many different conceptualizations of the hypercube architecture. For example, at certain instances it may be useful to treat the hypercube as a mesh or array. At other times a tree-like structure may be appropriate. The hypercube is very well suited for this due to its high connectivity.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, pp. 346–361, 1974.

[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, 20, pp. 762–772, 1977.

[3] D. M. Champion and J. Rothstein, "Immediate Parallel Solution of the Longest Common Subsequence Problem," in *Proc. 1987 Int'l Conf. on Parallel Processing*, pp. 70-77, 1987.

[4] T. F. Chan and Y. Saad, "Multigrid algorithms on the hypercube multiprocessor," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 64–75, Jan. 1987.

[5] J. H. Chang, O. H. Ibarra and M. A. Palis, "Parallel parsing on a one-way array of finite-state machines," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 64–75, Jan. 1987.

[6] H. D. Cheng and K.-S. Fu, "VLSI Architectures for Pattern Matching Using Space-Time Domain Expansion," in *Proc. IEEE Int'l Conf. on Computer Design ICCD*, pp. 181–184, 1985.

[7] E. Dekel, D. Nassimi and S. Sahni, "Parallel matrix and graph algorithms," *SIAM J. Comput.*, vol. 10, no. 4, pp. 657–675, Nov. 1981.

[8] E. Edmiston and R. A. Wagner, "Parallelization of the Dynamic Programming Algorithm for Comparison of Sequences," in *Proc. 1987 Int'l Conf. on Parallel Processing*, pp. 78–80, 1987.

[9] Z. Galil, "Optimal parallel algorithms for string matching," in *Proc. 16th ACM Symp. on Theory of Computing*, pp. 240–248, 1984.

[10] Z. Galil and J. I. Seiferas, "Time-space-optimal string matching," *J. Comput. Syst. Sciences*, 26, pp. 280–294, 1983.

[11] C.-T. Ho and S. L. Johnsson, "Distributed Routing Algorithms for Broadcasting and Personalized Communication in Hypercubes," in *Proc. 1986 Int'l Conf. on Parallel Processing*, pp. 640–648, 1986.

[12] O. H. Ibarra and M. A. Palis, "VLSI algorithms for solving recurrence equations and application," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, no. 7, pp. 1046–1064, July 1987.

[13] O. H. Ibarra, T.-C. Pong and S. M. Sohn, "Hypercube Algorithms for Some String Comparison Problems," Tech. Report 88-8, Univ. of Minnesota, 1988.

[14] O. H. Ibarra, T.-C. Pong and S. M. Sohn, "Hypercube Algorithms for the Substring Problem," Tech. Report, Univ. of Minnesota, 1988.

[15] R. M. Karp, R. E. Miller and A. L. Rosenberg, "Rapid identification of repeated patterns in strings, trees, and arrays," in *ACM Symp. on Theory of Computing*, vol. 4, pp. 125–136, 1972.

[16] D. E. Knuth, J. H. Morris and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, June 1977.

[17] V. K. P. Kumar and V. Krishnan, "Efficient Image Template Matching on Hypercube SIMD Arrays", in *Proc. 1987 Int'l Conf. on Parallel Processing*, pp. 765–771, 1987.

[18] H.-H. Liu and K.-S. Fu, "VLSI arrays for minimum-distance classifications," in *VLSI for Pattern Recognition and Image Processing*, K.-S. Fu, ed., pp. 45–63, 1984.

[19] W. Masek and M. Paterson, "A faster algorithm for computing string-edit distances," *J. Comput. Syst. Sci.*, vol. 20, pp. 18–31, 1980.

[20] D. Sankoff and J. B. Kruskal, ed.s, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Reading, Mass.: Addison-Wesley, 1983.

[21] U. Vishkin, "Optimal parallel pattern matching in strings," in *Proc. 12th ICALP, Lecture Notes in Computer Science 194*. Springer-Verlag, pp. 497–508, 1985.

[22] R. Wagner and M. Fischer, "The string-to-string correction problem," *J. Assoc. Comput. Machinery*, vol. 21, pp. 168–173, 1974.

[23] P. Weiner, "Linear pattern matching algorithms," in *IEEE Symp. on Switching and Automata Theory*, vol. 14, pp. 1-11, 1973.

Time Lower Bounds for Sorting on Multi-Dimensional Mesh-Connected Processor Arrays

Yijie Han[†*] and *Yoshihide Igarashi*[††]

[†]Department of Computer Science
University of Kentucky
Lexington, KY 40506-0027

[††]Department of Computer Science
Gunma University
Kiryu, 376 Japan

Abstract

We provide a useful technique called the Chain Theorem to derive good time lower bounds for sorting on the multi-dimensional mesh-connected model. For any $d \geq 2$ we derive a lower bound which is significantly better than the distance bound of dn on the d-dimensional model. We also distinguish between indexing schemes by showing that there exists a poorer indexing scheme than the snake-like indexing scheme on the multi-dimensional model. All these results are obtained using the chain theorem.

1. Introduction

A mesh-connected processor array is widely accepted as a realistic model of a parallel computer. The problem of sorting on a mesh-connected processor array has been studied by many researchers[3-7, 9-14]. It is known that that $(2d-1)n$ steps are optimal computing time within the leading term for sorting n^d items into d-dimensional snake-like order on the d-dimensional mesh-connected model[3, 4, 13]. However, up to now we do not know whether the snake-like order is the best indexing scheme for sorting. A question whether the distance bound $2n-2$ is ultimately achievable on a two dimension $n \times n$ mesh-connected model by using some super indexing scheme has also been raised[6].

The authors of the present paper have shown that $2.2247n$ steps are a time lower bound independent of indexing schemes for sorting n^2 items on the $n \times n$ mesh-connected model[1]. Thus, the question posed by Ma *et al.*[6] has been answered. Time lower bounds for various indexing schemes on the $n \times n$ mesh-connected model and the existence of a poor indexing scheme with $4n-2\sqrt{2n}-3$ time lower bound have also been shown[1]. These results have been obtained using a new technique called the chain argument[1].

In this paper we develop the chain argument in order that we can apply its extended version to derive nontrivial lower bounds for sorting. We show a theorem that gives a relation between computing time for sorting n^d items and the number of processors in a certain region of the mesh-connected model. We can numerically calculate the best lower bound obtainable from the theorem. For each $d \geq 2$ our result is significantly better than the distance bound of dn on the d-dimensional model.

[*] Work reported herein was partially supported by the University of Kentucky research initiation grant.

2. Preliminary

We consider a general model of a synchronous d-dimensional mesh-connected processor array consisting of n^d identical processors. It is denoted by $M[1..n, ..., 1..n]$ or $M[(1..n)^d]$. Each processor at location $(i_1,..., i_d)$, $1 \leq i_1, \ldots, i_d \leq n$, is denoted by $M[i_1,..., i_d]$. The distance between $M[i_1,..., i_d]$ and $M[j_1,..., j_d]$ is defined to be $\sum_{k=1}^{d} |i_k - j_k|$ and denoted by $dis((i_1,...,i_d), (j_1,...,j_d))$. Processor $M[i_1,..., i_d]$ is directly connected with processor $M[j_1,..., j_d]$ if and only if $dis((i_1,..., i_d), (j_1, \ldots, j_d))=1$. All n^d processors work in parallel with a single clock, but they may run different programs. As for sorting computation, the initial contents of $M[(1..n)^d]$ are assumed of n^d linearly ordered items, where each processor has exactly one item. The final contents of $M[(1..n)^d]$ are the sorted sequence of the items in a specific order. In one step each processor can communicate with all of its directly-connected neighbor processors. The interchange of items in a pair of directly-connected processors or the replacement of the item in a processor with the item in one of its neighbor processors can be done in one step. The computing time is defined as the number of parallel steps of the basic operations to reach the final configuration.

An indexing on processor array $M[(1..n)^d]$ is a one-to-one mapping from $\{1, ..., n\}^d$ to $\{1, ..., n^d\}$. For an indexing I, the index of $M[i_1,..., i_d]$ is denoted by $I(i_1,..., i_d)$. Some indexing schemes on the 2-dimensional model are shown in Fig. 1.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

(a)

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

(b)

Fig. 1. (a) Row-major indexing
(b) Snake-like row-major indexing

A subset of $M[(1..n)^d]$ is called a region. The distance between processor $M[i_1,..., i_d]$ and region S is defined by $\min\{dis((i_1, \ldots, i_d), (j_1,...,j_d))|M[j_1,...,j_d]$ is in $S\}$ and denoted by $dis((i_1, \ldots, i_d), S)$. A sequence $<(i_{11},...,i_{1d}),..., (i_{c1},..., i_{cd})>$ is called a chain under indexing I if and only if $<I(i_{11},..., i_{1d}),.., I(i_{c1},..., i_{cd})>$ is a consecutive integer sequence in increasing order. For the above chain its length is $c-1$. Processor $M[i_1,..., i_d]$ is called a corner if and only if for each j $(1\leq j\leq d)$ i_j is 1 or n. For an integer i we denote integer $n-i+1$ by $\overline{i}$. If $M[i_1,..., i_d]$ is a corner and k is a positive integer, $\{M[j_1,...,j_d]|dis((i_1, \ldots, i_d),(j_1,...,j_d))<k\}$ is called a corner region and denoted by $CREG((i_1,..., i_d), k)$. If S is a region, the cardinality of S is defined as the number of processors in S and denoted by $||S||$. An ordered pair of corner regions $CREG((i_1,..., i_d), k_1)$ and $CREG((\overline{i_1},..., \overline{i_d}), k_2)$ is called a sweep. The first corner region of the sweep is called the residing region and the second one is called the stretching region. The length of the sweep is defined to be $d(n-1)+k_1-k_2$.

3. The Chain Theorem

Sweeps play an important role in deriving time lower bounds for sorting on the mesh-connected model. Our first theorem gives a relation among the computing time, the length of a sweep and the length of a chain.

Theorem 1 (Chain Theorem): For an indexing I on the d-dimensional mesh-connected model and a sweep of length T, if there is no chain in its residing region such that its length is equal to the cardinality of the stretching region, then there is no algorithm of time complexity less than T for sorting n^d items on the model into the order specified by I.

Proof: Let $(CREG((i_1,..., i_d), k_1), CREG((\overline{i_1},..., \overline{i_d}), k_2))$ be a sweep, where each $i_j(1\leq j\leq d)$ is 1 or n. Then the length of the sweep is $T=d(n-1)+k_1-k_2$. Let S be the cardinality of the stretching region. Suppose that an algorithm of time complexity $T-1$ is executed on the d-dimensional mesh-connected model. The effect of the initial contents of the stretching region to corner $M[i_1,..., i_d]$ does not appear before $((n-1)d-k_2+1)$-st step of the computation. Let a be the item in $M[i_1,..., i_d]$ immediately after the $((n-1)d-k_2)$-th step of the computation. The destination of item a depends on the initial contents of the stretching region. By assigning different initial values to the processors in the stretching region, we can force item a into $S+1$ different sorted positions. These different positions form a chain of length S, and should be within the residing region since the computing time is $T-1$. □

Kunde[3] has shown a time lower bound for sorting in lexicographic order on the multi-dimensional mesh-connected model. We first show how to use the Chain Theorem by applying it to derive Kunde's lower bound.

Lemma 1: Let $R=\{(i_1, \ldots, i_d)|$ for each j $(1\leq j\leq d)$ i_j is a positive integer and $\sum_{j=1}^{d}(i_j-1)<k\}$. Then $k^d/d!\leq||R||\leq(k+d-1)^d/d!$, where $||R||$ is the cardinality of R.

Proof: $||R|| = \binom{k+d-1}{d}$, and $\int_0^k dx_1 \int_0^{k-x_1} dx_2 \cdots \int_0^{k-\sum_1^{d-2} x_i} dx_d <$ $||R|| < \int_0^{k+d-1} dx_1 \int_0^{k+d-1-x_1} dx_2 \cdots \int_0^{k+d-1-\sum_1^{d-2} x_i} dx_d$. Therefore, $k^d/d! < ||R|| < (k+d-1)^d/d!$. □

Corollary 1: Let $M[i_1, \ldots, i_d]$ be a corner and V be the cardinality of $CREG((i_1,..., i_d), k)$. If $1\leq k\leq n$, $k^d/d!<V<(k+d-1)^d/d!$.

Theorem 2: A time lower bound for sorting n^d items on the d-dimensional mesh-connected model into lexicographic order is $(2d-1)n-\lceil(d!n^{d-1})^{1/d}\rceil-2d+2$ steps.

Proof: Consider the sweep $(CREG((1, n,..., n), (d-1)n-d+2), CREG((n, 1, ..., 1), \lceil(d!n^{d-1})^{1/d}\rceil))$. The length of the sweep is $(2d-1)n-\lceil(d!n^{d-1})^{1/d}\rceil-2d+2$. From Corollary 1 the cardinality of the stretching region is greater than n^{d-1}. The length of the longest chain in the residing region is $n^{d-1}-1$. Therefore, from the Chain Theorem any algorithm for sorting n^d items on the model into lexicographic order takes at least $(2d-1)n - \lceil(d!n^{d-1})^{1/d}\rceil - 2d + 2$ steps. □

The above lower bound is also a lower bound for the snake-like order.

4. A Poor Indexing Scheme

Kunde[4] has shown that within the leading term, $(2d-1)n$ steps are the asymptotically optimal computing time for sorting n^d items into snake-like order. We show the existence of a poorer indexing scheme than the snake-like indexing.

Theorem 3: There exists an indexing scheme such that any algorithm for sorting n^d items by the indexing scheme on the d-dimensional mesh-connected model takes at least $2dn-2\lceil(d!)^{1/d}n^{1/2}\rceil-2d+1$ steps.

Proof: Let $k=\lceil(d!)^{1/d}n^{1/2}\rceil$. Consider the sweep $(CREG((1,.., 1), d(n-1)-k+1), CREG((n,..., n), k))$. The length of the sweep is $2dn-2\lceil(d!)^{1/d}n^{1/2}\rceil-2d+1$. From Corollary 1 the cardinality of the stretching region is not smaller than $\lceil n^{d/2}\rceil$. We define an indexing scheme as follows: The first $\lceil n^{d/2}\rceil$ sorted positions are in the residing region, the $(\lceil n^{d/2}\rceil+1)$-st sorted position is in the stretching region, the next $\lceil n^{d/2}\rceil$ sorted positions are in the residing region, the $(2\lceil n^{d/2}\rceil+2)$-nd sorted position is in the stretching region, and so on. Then the length of the longest chain in the residing region is $\lceil n^{d/2}\rceil-1$. This length is smaller than the cardinality of the stretching region. Therefore, from the Chain Theorem this theorem holds. □

5. Cardinality of Various Regions

In this section we describe how cardinalities of corner regions, center regions and unions of corner regions are evaluated. The union of k-corner regions on the d-dimensional model is defined by $\bigcup_{(i_1,..., i_d)\in\{1, n\}^d} CREG((i_1,.., i_d), k)$, and denoted by $UCREG(k, d)$. The r-center region of the d-dimensional model is denoted by $CENT(r, d)$ and defined as follows: $CENT(0, d)$ is the empty set. If n is even, $CENT(1, d)$ is

$\{M[i_1, \ldots, i_d]|$ for each j $(1 \le j \le d)$ i_j is $n/2$ or $n/2+1\}$, and otherwise $CENT(1, d)$ is $\{M[\lceil n/2 \rceil, \ldots, \lceil n/2 \rceil]\}$. For $r \ge 2$ $CENT(r, d) = \{M[i_1, \ldots, i_d] | dis((i_1, \ldots, i_d), CENT(1, d)) < r\}$.

Lemma 2: Let $tn \le k \le (t+1)n$, $0 \le t \le d-1$. Then for each $(c_1, \ldots, c_d)$ in $\{1, n\}^d$, the following inequalities hold:

$$\frac{k^d}{d!} - \frac{(k-n)^d}{(d-1)!} + \frac{(k-2n)^d}{2!(d-2)!} - \cdots + (-1)^t \frac{(k-tn)^d}{t!(d-t)!}$$
$$< ||CREG((c_1, \cdots, c_d), k)||$$
$$< \frac{(k+d-1)^d}{d!} - \frac{(k-n+d-1)^d}{(d-1)!} + \frac{(k-2n+d-1)^d}{2!(d-2)!} - \cdots$$
$$+ (-1)^t \frac{(k-tn+d-1)^d}{t!(d-t)!}.$$

Proof: Let us first evaluate the volume of a region on the d-dimensional real space. Let $R(k)$ be $\{(i_1, \ldots, i_d)|$ for each j $(1 \le j \le d)$ i_j is a positive real number, and $\sum_{j=1}^{d} i_j < k\}$, and let $V(k)$ be $\{(i_1, \ldots, i_d)|$ for each j $(1 \le j \le d)$ i_j is a positive real number not greater than n, and $\sum_{j=1}^{d} i_j < k\}$. If S is a region on the d-dimensional real space, the volume of S is denoted by $||S||$. Then

$$||R(k)|| = \int_0^k dx_1 \int_0^{k-x_1} dx_2 \cdots \int_0^{k-\sum_1^{d-2} x_i} dx_d = k^d/d!, \quad \text{and}$$

$||V(k)|| < ||CREG((c_1, \ldots, c_d), k)|| < ||V(k+d-1)||$. Let $\{a_1, \ldots, a_d\}$ be the set of properties on elements in $R(k)$, where a_i is the property that the value of the i-th coordinate is greater than n. Let $R(a_i, k)$ be the subregion of $R(k)$ having property a_i, and let $N(a_i)$ be the volume of $R(a_i, k)$. We also define $R(a_i', k)$ as the subregion of $R(k)$ not having property a_i and $N(a_i')$ as the volume of $R(a_i', k)$. Since an element in $R(k)$ can have more than one property, we also use the following notations: $N(a_{i_1}, \ldots, a_{i_t})$ is the volume of the subregion of $R(k)$ having properties $a_{i_1}, \ldots, a_{i_t}$ and $N(a_{i_1}', \ldots, a_{i_t}')$ is the volume of the subregion of $R(k)$ not having properties $a_{i_1}, \ldots, a_{i_t}$. Then $||V(k)|| = N(a_1', \ldots, a_d')$. From the principle of inclusion and exclusion[8], $||V(k)|| = ||R(k)|| - \sum N(a_{i_1}) + \sum N(a_{i_1}, a_{i_2}) - \sum N(a_{i_1}, a_{i_2}, a_{i_3}) + \cdots + (-1)^d N(a_1, a_2, \ldots, a_d)$, where the sum of $\sum N(a_{i_1}, \ldots, a_{i_t})$ is taken over all combinations of t properties. If $k \le tn$, then all the terms after the t-th term in the right hand side of the above formula are 0. By a simple integration as the one in Lemma 1, we have $N(a_{i_1}, \ldots, a_{i_t}) = \frac{(k-tn)^d}{d!}$ if $k \ge tn$. Hence $||V(k)|| = \frac{k^d}{d!} - \binom{d}{1}\frac{(k-n)^d}{d!} + \binom{d}{2}\frac{(k-2n)^d}{d!} - \cdots + (-1)^t \binom{d}{t}\frac{(k-tn)^d}{d!} = \frac{k^d}{d!} - \frac{(k-n)^d}{(d-1)!} + \frac{(k-2n)^d}{2!(d-2)!} - \cdots + (-1)^t \frac{(k-tn)^d}{t!(d-t)!}$, where $k \ge tn$. Therefore the lemma holds. □

We consider that each element in $UCREG(k, d)$ belongs to its nearest corner region. That is, we define $DCREG((c_1, \ldots, c_d), k)$ to be the set $\{M[i_1, \ldots, i_d] | dis((c_1, \ldots, c_d), (i_1, \ldots, i_d)) < k$ and for each j $(1 \le j \le d)$ $|c_j - i_j| \le \lfloor n/2 \rfloor - 1\}$. If n is even, $UCREG(k, d) = \bigcup_{(c_1, \ldots, c_d) \in \{1, n\}^d} DCREG((c_1, \ldots, c_d), k)$. If n is odd, $UCREG(k, d) \subseteq \bigcup_{(c_1, \ldots, c_d) \in \{1, n\}^d} DCREG((c_1, \ldots, c_d), k)$.

Lemma 3: If $t\lfloor n/2 \rfloor \le k \le (t+1)\lfloor n/2 \rfloor$ and $0 \le t \le d-1$, then

$$\frac{(2k)^d}{d!} - \frac{(2k-n)^d}{(d-1)!} + \frac{(2k-2n)^d}{2!(d-2)!} - \cdots + (-1)^t \frac{(2k-tn)^d}{t!(d-t)!}$$
$$\le ||UCREG(k, d)||$$
$$\le \frac{(2(k+d-1))^d}{d!} - \frac{(2(k+d-1)-n)^d}{(d-1)!} + \frac{(2(k+d-1)-2n)^d}{2!(d-2)!} - \cdots$$
$$+ (-1)^t \frac{(2(k+d-1)-tn)^d}{t!(d-t)!}.$$

Lemma 4: If $(d-t-1)\lfloor n/2 \rfloor \le k \le (d-t)\lfloor n/2 \rfloor$ and $0 \le t \le d-1$, then

$$n^d - \frac{(dn-2k)^d}{d!} + \frac{((d-1)n-2k)^d}{(d-1)!} - \frac{((d-2)n-2k)^d}{2!(d-2)!} + \cdots$$
$$+ (-1)^{t+1} \frac{((d-t)n-2k)^d}{t!(d-t)!}$$
$$\le ||UCREG(k, d)||$$
$$\le n^d - \frac{(dn-2(k+d-1))^d}{d!} + \frac{((d-1)n-2(k+d-1))^d}{(d-1)!}$$
$$- \frac{((d-2)n-2(k+d-1))^d}{2!(d-2)!} + \cdots + (-1)^{t+1} \frac{((d-t)n-2(k+d-1))^d}{t!(d-t)!}.$$

The proofs of Lemmas 3 and 4 are similar to that of Lemma 2[2]. The values of the formulae bounding $||UCREG(k, d)||$ in Lemma 3 is exactly the same as the values of the formulae in Lemma 4. If $k \le dn/4$ then the evaluation of $||UCREG(k, d)||$ by Lemma 3 is easier than by Lemma 4, and otherwise the evaluation by Lemma 4 is easier than by Lemma 3.

6. A Lower Bound for an Arbitrary Sorting Order

In this section we derive a time lower bound independent of indexing schemes on the d-dimensional model.

Theorem 4: Let $V = ||UCREG(k_1, d)||$ and $||CREG((1, \ldots, 1), k_2)|| \ge n^d - \lceil V/2 \rceil$, where $1 \le k_1, k_2 \le (n-1)d+1$. Then a lower bound for sorting n^d items by any indexing scheme on the d-dimensional mesh-connected model is $2d(n-1) - k_1 - k_2 + 1$ steps.

Proof: We consider an arbitrary indexing I on the d-dimensional mesh-connected model. There exists a position (*i.e.*, a processor) in $UCREG(k_1, d)$ such that $\lceil V/2 \rceil \le I(b) \le n^d - \lfloor V/2 \rfloor + 1$ or $\lfloor V/2 \rfloor \le I(b) \le n^d - \lceil V/2 \rceil + 1$. Such a position b is in at least one corner region $CREG((i_1, \ldots, i_d), k_1)$. Without loss of generality we may assume that b is in $CREG((n, \ldots, n), k_1)$. Consider the sweep, $(CREG((1, \ldots, 1), (n-1)d - k_1 + 1), CREG((n, \ldots, n), k_2))$. Since b is outside of the residing region, the length of the longest chain in the residing region is at most $n^d - \lceil V/2 \rceil - 1$. Since the cardinality of the stretching region is not less than $n^d - \lceil V/2 \rceil$, from the Chain Theorem a lower bound for sorting n^d items by indexing I on the model is $2d(n-1) - k_1 - k_2 + 1$ steps. □

Good time lower bounds for arbitrary sorting order can be obtained from Theorem 4 by minimizing the value of $k_1 + k_2$. Although it is difficult to give a general formula of the maximized lower bound as a function of n and d, we can numerically derive it for an arbitrary d.

Proposition 1: A time lower bound for sorting n^d items into an arbitrary order on the d-dimensional mesh-connected model is $\lfloor (d - 0.5 + (0.5)^{1/d})n \rfloor - d$ steps.

Proof: Let k_1 be $\lfloor n/2 \rfloor$. Then 2^d corner regions $CREG((i_1, \ldots, i_d), k_1)$ are mutually disjoint. From Lemma 1 $V=||UCREG(k_1, d)||$ is not less than $\lfloor n^d/d! \rfloor$. Let k_2 be $d(n-1)+1-\lfloor (1/2)^{1/d} n \rfloor$. Then the cardinality of corner region $CREG((i_1, \ldots, i_d), k_2)$ is not less than $n^d - \lceil V/2 \rceil$. Hence, from Theorem 4, a lower bound for sorting n^d items by any indexing scheme on the d-dimensional mesh-connected model is $\lfloor (d-0.5+(0.5)^{1/d})n \rfloor - d$ steps. □

Since we are mainly interested in asymptotic lower bounds, we hereafter omit minor terms, ceilings and floors in formulae of lower bounds.

From Lemma 3 $V=||UCREG(n, d)|| \geq ((2n)^d - dn^d)/d!$. If the cardinality of $CREG((i_1, \ldots, i_d), r)$ is not less than $n^d - V/2$, from Theorem 4 a lower bound for sorting n^d items on the d-dimensional model is $(2d-1)n-r$ steps. In this case there exists such r in the range between $(d-1)n$ and $(d-2)n$. Let $r=(d-1-t)n$, where $0<t<1$. Then $||CREG((i_1,\ldots, i_d), r)|| \geq n^d - V/2$ if and only if $(1+t)^d - dt^d \leq (2^d-d)/2$. Let $t+1=((2^d-d)/2)^{1/d}$. Then $||CREG(i_1, \ldots, i_d), r)|| \geq n^d - V/2$. Therefore, $(((2^d-d)/2)^{1/d}+d-1)n$ is a time lower bound for sorting n^d items on the model. Hence we have the next proposition.

Proposition 2: A time lower bound for sorting n^d items into an arbitrary order on the d-dimensional mesh-connected model is $(((2^d-d)/2)^{1/d}+d-1)n$ steps.

Time lower bounds given by Proposition 2 are listed in Table 1. These lower bounds are not the best ones obtainable from Theorem 4.

d	time lower bound
2	$2.0000n$
3	$3.3572n$
4	$4.5650n$
5	$5.6829n$
6	$6.7528n$
7	$7.7969n$
8	$8.8267n$

Table 1: Asymptotic lower bounds obtained by Proposition 2.

For $d=2$ we choose $k_1=(1-\sqrt{6}/6)n$ and $k_2=(2-\sqrt{6}/3)n$. Then $||CREG(i_1, \ldots, i_d), k_2)|| \geq n^2 - ||UCREG(k_1, d)||/2$. Therefore, from Theorem 4, $4n-k_1-k_2 \approx 2.2247n$ is a time lower bound[1].

Theorem 5: An asymptotic time lower bound for sorting n^2 items into any sorting order on the 2-dimensional mesh-connected model is $(1+\sqrt{6}/2)n \approx 2.2247n$ steps.

For $d=3$, let $k_1=0.87n$ and $k_2=1.7294n$; for $d=4$, let $k_1=1.12n$ and $k_2=2.2667n$; for $d=5$, let $k_1=1.395n$ and $k_2=2.7893n$. From Theorem 4 we have the following theorems.

Theorem 6: An asymptotic time lower bound for sorting n^3 items into any sorting order on the 3-dimensional mesh-connected model is $3.4086n$ steps.

Theorem 7: An asymptotic time lower bound for sorting n^4 items into any sorting order on the 4-dimensional mesh-connected model is $4.6133n$ steps.

Theorem 8: An asymptotic time lower bound for sorting n^5 items into any sorting order on the 5-dimensional mesh-connected model is $5.8207n$ steps.

Lower bounds listed in Theorems 6, 7, 8 are the best ones obtainable from Theorem 4. The proofs of these theorems are given in [2]. For each d we can derive a time lower bound better than the lower bound given in Table 1.

7. Concluding Remarks

The question whether there exists a sorting algorithm with less than $(2d-1)n$ time for some indexing scheme still remains open. The snake-like row-major order and its trivial variations are only the known indexing schemes for which optimal sorting algorithms within the leading term have been found. We are interested in finding optimal sorting algorithms for other indexing schemes on the multi-dimensional mesh-connected model.

References

[1]. Y. Han and Y. Igarashi, Time lower bounds for sorting on a mesh-connected processor array, 1988 Proceedings of Aegean Workshop on Computing, (Lecture Notes in Computer Science, Springer).

[2]. Y. Han and Y. Igarashi, Time lower bounds for parallel sorting on a multidimensional mesh-connected processor arrays, TR No. 107-88. Dept. Comput. Sci., Univ. of Kentucky-Lexington(Jan. 1988).

[3]. M. Kunde, Lower bounds for sorting on mesh-connected architectures, Acta Informatica, 24(1987), pp. 121-130.

[4]. M. Kunde, Optimal sorting on multi-dimensionally mesh-connected computers, 4th Symp. on Theoretical Aspects of Computer Science, LNCS 247 Springer, 1987, pp. 408-419.

[5]. H.-W. Lang, M. Schimmler, H. Schmech and H. Schröder, Systolic sorting on a mesh-connected network, IEEE Trans. Comput., C-34(1985), pp. 652-658.

[6]. Y. Ma, S. Sen and I.D. Scherson, The distance bound for sorting on mesh-connected processor arrays is tight, 27th Symp. on Foundations of Comput. Sci., IEEE, 1986, pp. 255-263.

[7]. D. Nassimi and S. Sahni, Bitonic sort on a mesh-connected parallel computer, IEEE Trans. Comput., C-27(1979), pp. 2-7.

[8]. F.S. Roberts, Applied Combinatorics, Prentice-Hall, Englewood Cliffs, 1984.

[9]. K. Sado and Y. Igarashi, A divide-and-conquer method of the parallel sort, IECE of Japan, Tech. Commit. of Automata and Languages, AL84-68, pp. 41-50(1985).

[10]. K. Sado and Y. Igarashi, Fast parallel sorting on a mesh-connected processor array, Proc. of Japan-U.S. Joint Seminar, Discrete Algorithms and Complexity Theory(Johnson, D.S. et al. eds), Academic Press, New York, 1986, pp. 161-183.

[11]. K. Sado and Y. Igarashi, Some parallel sorts on a mesh-connected processor array, J. Parallel and Distributed Computing, Vol. 3(1986), pp. 389-410.

[12]. I.D. Scherson, S. Sen and A. Shamir, A true two-dimensional sorting technique for VLSI networks, Proc. 1986 Int. Conf. on Parallel Processing, 1986, pp. 903-908.

[13]. C.P. Schnorr and A. Shamir, An optimal sorting algorithm for mesh-connected computers, Proc. 18-th ACM Symp. on Theory of Computing, 1986, pp. 255-263.

[14]. C.D. Thompson and H.T. Kung, Sorting on a mesh-connected parallel computer, Commun. ACM, 20(1977), pp. 263-271.

Accounting for Parallel Tree Search Overheads

E. Altmann*, T.A. Marsland, T. Breitkreutz

Computing Science Department
University of Alberta
Edmonton, Alberta
Canada T6G 2H1

Abstract

Loosely coupled parallel systems are an appealing architecture for increased computing power, because of the comparative simplicity of coordinating individual networked processors to work on a single problem. However, the appeal of such systems for many search problems is offset by their marked failure to achieve linear speedup as the degree of parallelism grows.

Here, the *overheads* (losses) that result in sub-linear speedup for search are examined in light of the *vertex cover* problem. Although this task is less complicated than other combinatorial search problems, such as chess, it exhibits similar overheads when solved using loosely coupled parallel systems. Such overheads arise because pruning causes the search trees to become skewed, which in turn makes it difficult to schedule work to keep all processors productive.

The combined overheads comprise solution time *overrun*, the amount by which a solution time is slower than linear speedup. We present and discuss experimental data in an attempt to account for solution time overrun and communication and synchronization losses, as they occur in loosely coupled parallel search.

Acknowledgements. Financial support from the Canadian Natural Sciences and Engineering Research Council through Grant A7902 made the experimental work possible.

1 Introduction

Using a loosely coupled network of processors is a simple way to increase the processing power available to an application. This simplicity is appealing, especially given the increasing number of computing facilities that feature networked workstations. Any such facility is a candidate for becoming a loosely coupled parallel system, with the addition of software to control processor communication and scheduling. Thus loosely coupled parallel systems have the potential to be a comparatively simple, low-cost answer to the demand for increased computing power.

However, for important problems such as game tree search, such systems can exhibit heavy computational overheads that undermine their efficiency and limit their effective speed. Our primary motivation for this study was the investigation of a particular type of overhead, termed *synchronization* loss (defined in Section 2), which can severely impair the performance of computer chess programs. Control of this overhead is made especially difficult in the chess case by the complexity of the program itself and the difficulty in subdividing the work into smaller chunks that can be distributed equitably across all processors. Therefore we sought a simpler application whose parallel implementation exhibited a similar serious synchronization overhead. We chose the *vertex cover* problem, which may be stated as follows: *Given an undirected graph, find the smallest set of vertices such that every edge in the graph is incident to at least one vertex in the set* [5].

2 Types of Overheads

Overheads in loosely coupled parallel search fall into three broad categories:

1. *Communication overhead*, in which processors wait while they exchange information that may improve their efficiency. Typically this exchange involves updating or retrieving data from a global shared table, or sending and receiving messages.

2. *Search overhead*, in which more nodes are searched in the parallel implementation than in a sequential one. One form of search overhead occurs when processors unwittingly do redundant calculations. Because work is not being done in the strictly sequential style of a single processor, information must be shared if processors are to detect and avoid the duplication of work; redundant calculations arise when information that should be shared does not arrive in time to prevent the initiation of extra work. Another form of search overhead occurs when work is deliberately assigned to processors on a speculative basis, in the absence of anything better for them to do. Here duplicated or unnecessary effort is a direct trade-off against idleness.

3. *Synchronization overhead*, in which processors become idle after completing their assigned work, and cannot continue until some (even all) others finish completely. Clearly, if processors are idle due to poor synchronization, then overall speed of the system is far less than optimal.

Normally there is a trade-off between communication, search, and synchronization overheads. For example, if speculative computing is employed to reduce synchronization loss, there will normally be some increase in search overhead, and perhaps also extra communication.

3 Effective Power of Parallel Solutions

The power of parallel solutions is often demonstrated via the solution of classical combinatorial problems [13]. Especially popular is the *traveling salesman* problem, since all combinations might have to be searched and hence nearly linear speedup is possible [9]. Almost all combinatorial search problems are well-suited to a multiprocessor solution, provided that most work is independent (i.e., negligible data sharing is needed and subproblems can be solved in any order). In the ideal case, subproblems are also predictable in size, allowing nearly perfect processor scheduling.

If search tree pruning techniques are used in a sequential solution, then in parallel adaptations of that solution processors must share information that leads to cutoffs. Information-sharing entails communication cost. More importantly, pruning in a parallel solution introduces synchronization overhead because it renders the size of a chunk of work unpredictable. Thus pruning, while improving sequential solutions, contributes to overheads in parallel solutions, making ideal (linear) speedup over the best available sequential solution difficult to achieve.

The effective power of a parallel solution may be overestimated if the uniprocessor solution against which it is compared is *not* the fastest available. Consider sequential minimax search of a uniform tree (i.e., exhaustive search of a decision tree of fixed depth and constant branching factor). Simple tree-splitting will yield close to ideal speedups. There is no pruning, so there is no need for communication, and subproblems are of uniform size. However, in current parallel implementations of the $\alpha - \beta$ pruning algorithm [10], tree-splitting clearly fails to achieve linear speedup.

Given that pruning effects are equally desirable in a parallel solution, methods must be devised to control the overheads that stem from them.

*Present address: Computer Science Department, Carnegie Mellon University, Pittsburgh PA, 15213-3890.

One such method is to dynamically reconfigure communication paths so that idle processors can be assigned to those that are still busy [12]. An alternative that is appealing for the relative simplicity of its control structure is to tailor a static processor configuration to the application at hand.

4 Parallel Solutions to Vertex Cover

A multiprocessor solution to vertex cover was one of several potential applications for the MANIP architecture proposed by Wah and Ma [14]. A follow-up study by Zariffa [15] used a 7 processor system to gain working experience with some pragmatic aspects of multicomputer systems. Sufficient implementation details were provided in that study for us to replicate its experiments, and thereby compare computer systems and processor configurations. This paper presents data gathered in the course of our replication.

In the original experiments, 15 graphs were searched with 2, 4, and 7 processors, statically configured, and scheduled using three different schemes. Of the scheduling methods, *dynamic first level* (*dfl*) proved the most effective, and is the method on which our study is based. *Dfl* assigns a first-level tree node to each processor, then dynamically assigns the remaining ones on a first-come, first-serve basis.

The original search algorithm prunes branches whose solutions cannot better the best found so far, and therefore builds trees similar to chess trees in that they are somewhat skewed to the left. We devised a faster algorithm that builds smaller and highly skewed trees [1], but used the original one in our experiments to allow comparison of results. The existence of this faster sequential algorithm implies that our speedups do not necessarily reflect the effective power of parallel vertex cover solutions.

We note that Zariffa's data on the occurrence of acceleration anomalies [6,4] is inconsistent. In our own experiments acceleration anomalies were not observed, although they do occur in game trees [8].

5 Implementation Details

In this section we present implementation factors that bear on the analysis that follows in Section 6.

5.1 Non-homogeneous Processor Systems

The hardware in Zariffa's system consisted of Data General Nova 4s and one Nova 3. The system was non-homogeneous, the Nova 3 being roughly 10 percent faster than the others. However, because the Nova 3 provided disk access for the system, it participated in all experiments.

Our system had an analogous feature: six Motorola 68010s, running with limited operating system support, were roughly 20 percent faster than the seventh, which was identical but ran under UNIX[1] and provided full multitasking. The UNIX-based processor (named *sunshine*) was also the only one that provided disk access for the system. To reduce operating system overhead to a minimum, the kernel of the other processors (referred to as *standalones*) supported Ethernet access as the only I/O function, and did not support multiple application processes. Sunshine necessarily participated in all the experiments, but usually only as a master data gatherer.

5.2 Processor Configurations

The topology of communication paths in our system (henceforth the *processor configuration*) was different than that in the original work, where constraints imposed by the hardware dictated that a particular ring configuration was the simplest and most effective.

In this form of ring, a record of control information, such as a list of completed chunks of work, is maintained by each processor. When the record changes, as upon the selection of a new chunk by a processor, the updated record is passed around the ring. Because P copies of the control information are maintained, updating the information requires that a packet travel $P-1$ successive hops before the update is complete. In the original experiments, an update was delayed at least 10 milliseconds (*ms*) before reaching the last processor. Thus it was possible for two nodes to claim the same chunk of new work within the period of communication latency, since they used their own copy of the control information when looking for work [15].

A broadcast bus (Ethernet) and our Virtual Tree Machine (VTM) [11] allowed us greater flexibility in designing a processor configuration. We used a single-level process tree, in which slave processes executed the search, and spoke only with a master process. (Although arbitrary and dynamic configurations are possible with the VTM, such static processor trees remain the most common [11].) The slave processes resided on the standalones, and the master, responsible for file I/O, resided on sunshine. The master also coordinated the search, maintaining the unique copy of the control record. The advantages of such a configuration are twofold:

1. Faster broadcasting of updates. Although information must still travel P hops for an update (1 slave-to-master message and $P-1$ master-to-slave messages), the processor configuration imposes no sequential ordering on messages. The master queues all outgoing messages in a tight loop, without waiting for any particular one to be sent. Thus the $P-1$ messages are broadcast nearly simultaneously.

2. No possibility of duplicating a chunk of work. All reads and writes of search control information are executed sequentially on a single copy of the record, so the information is always consistent.

A master/slave configuration can be susceptible to message-processing bottlenecks in the master, particularly at higher degrees of parallelism [15]. Our systems did not exhibit significant communication overhead, implying that such a bottleneck is not a factor up to parallelism of degree 7 [1].

Another issue that arises from employing a master process is the selection of a physical processor on which to place it. In our system the master, requiring file I/O capability, had to reside on sunshine for all experiments. Each slave process was assigned to its own standalone processor in our 2 and 4 processor systems. However, because of limited hardware availability when our experimental work began, it was initially convenient to allow sunshine to host both the master and a slave for the 7 processor experiments. Later the effects of a doubled versus an independent master were explored, using a seventh (newly acquired) standalone processor; the results are presented in Section 6.2.

5.3 Message Passing Operations

Three message passing operations occur in our systems: *polls*, *sends*, and *receives*. *Sends* and *receives* copy information into and out of system buffers, respectively, taking small and constant amounts of time. *Polls* are of two sorts, *non-blocking* and *blocking*. *Non-blocking* polls are used by slaves to check for new bounds before node expansion, and so take small and bounded time.

Immediately after a slave *sends* results to the master, it issues a *blocking* poll so that it can wait for new work. The interval spanned by this poll incorporates the time for the results to reach the master, the time for the master to process them and build a new piece of work, and the time for the new work to reach the slave's system buffer. Thus the polling interval embodies all time spent in communication between processes during the search, as well as time spent waiting for work.

6 Identifying Overheads

Our averaged speedup figures show the declining effectiveness of additional processors as the degree of parallelism is increased. For the graphs in our test set the speedups were as follows: 1.81 for 2 processors, 3.26 for 4 processors, and 4.69 for 7 processors. In this section we identify and quantify the overheads that limited the performance of our parallel systems.

We note that a positive correlation between problem size and speedup was discovered, which indicates that experiments with larger graphs might yield different figures. We have addressed the generation of large

[1] UNIX is a registered trademark of Bell Laboratories.

graphs having specified statistical properties, and have explored a variety of binary-tree processor configurations for solving vertex cover [2].

6.1 Estimating Communication Overhead

Data that measures idle time per processor suffices to quantify communication overhead. In the 4 processor case slaves ran as the only process on their respective machines, while in the 7 processor case the master process and one slave were *doubled up*. Our prediction that the doubled-up processor would be capable of less work as a slave is borne out by lower-than-average node counts (counts of nodes searched) [1].

The salient feature in the 4 processor data agrees well with intuitive prediction. For each problem at least one processor has effectively 0 idle time relative to solution time. (Solution times for 4 processors appear in the *Elapsed* column of Table 2. Idle times were between 30 and 40 *ms*, with once exception of 118 *ms*.) Since the calculation of idle time includes time spent on communication, the communication loss for that processor is also effectively zero. If we assume that total communication overhead is equally distributed across processors, it follows that communication overhead is negligible for all processors. This assumption is intuitively rational, there being no apparent reason to suspect that the "busy" processor is biased toward a lesser degree of communication.

The 7 processor data exhibits an interesting anomaly in the elapsed times of the last processor to finish. When the doubled-up processor is last to finish, it usually has an idle time of exactly 0. Each of the other processors, when last to finish, has an idle time in the range 76 to 196 *ms*, a roughly constant discrepancy from the range of minimum times found in the 4 processor data. This noise may be an effect of the doubling-up of master and slave, a factor for which control was lacking. We subsequently investigated the effects of doubling-up, and present a discussion of the results in Section 6.2.

Two other aspects of the 7 processor data bear mention. First, the noise in the idle-time data reinforces the need to experiment with sufficiently large problem instances, so that solution times overwhelm small and constant overheads that are difficult to identify. Second, there are examples where several processors have long idle times (on the order of 30% of the total running time), highlighting the inefficiency of *dfl* scheduling.

6.2 Placement of Master Process

In Table 1 we present data on the effect of doubling a master and a slave on one machine. For this experiment five of the computationally more expensive problems were chosen, and a seventh standalone processor was used for the 7 processor independent master tests. Solution times with an independent master appear in the *Master Alone* columns, and the increases in solution time when the master was doubled up appear in the *Master Doubled* columns.

	Times (seconds)					
	2 Processors		4 Processors		7 Processors	
Prob. #	Master Alone	Master Doubled	Master Alone	Master Doubled	Master Alone	Master Doubled
11	288.64	+48.99	158.59	+17.91	106.32	-1.56
12	188.60	+25.52	109.39	+12.39	75.00	-0.06
13	251.36	+35.01	127.61	+14.11	86.33	+10.78
14	310.72	+45.86	163.50	+33.71	107.56	+4.30
15	226.05	+51.60	128.94	+2.93	76.61	+4.95
av.	253.07	+41.40	137.61	+16.21	90.36	+3.68

Table 1: Timing Effects of Doubling Master and Slave

The data shows that for 2 and 4 processor systems an independent master increases performance by a non-trivial amount. However, with 7 processors an independent master led to some slower solution times (Problems 11 and 12). For these instances the increase in processing power was small enough to be offset by random, detrimental changes in the order of work allocation. Our conclusion is that, as the degree of parallelism increases, the benefit from an independent master tends to zero.

We infer that having one slightly *faster* processor will have just as little effect with 7 or more processors as having a slightly *slower* one. This justifies a direct comparison [1] between our 7 processor system and the one used in the original study. On the other hand, in the 4 processor case our solution times may overestimate the power of our system, since the master resided on an independent fifth processor. The general conclusion drawn from the data is that the greater the degree of parallelism, the less significant the effect of having a processor of different capability.

6.3 Estimating Synchronization Overhead

Table 2 compares solution time *Overrun* with estimates of synchronization loss for the 4 processor case. The estimates, which appear in the *Clocking* and *Node Counts* columns, are derived from two different measurement methods. Solution times appear in the *Elapsed* column.

Overrun is simply *Elapsed* − *Linear Speedup*, where *Linear Speedup* is the sequential solution time divided by the number of processors. Thus overrun quantifies the difference between the effective and the ideal power of a parallel solution.

The *Clocking* method of estimating synchronization loss is just the sum of clocked polling intervals over all slaves, divided by the number of slaves. The results appear in the *Average Idle* column under *Clocking*.

The *Node Counts* method is slightly more complex. A *cost per node* value is derived for a given search tree, by dividing the largest of the processor *node counts* into the solution time. The *working time* for each processor is computed as *cost per node* × *node count*, where *node count* is for that processor. The idle time for each processor is then *Elapsed* − *working time*. The average of these times appears in the *Average Idle* column under *Node Counts*.

	Times (seconds)			Synch. Loss Estimators			
				Clocking		Node Counts	
Pr. #	Elapsed	Linear Speedup	Overrun	Av. Idle	Error (%)	Av. Idle	Error (%)
1	8.47	5.79	2.69	2.28	15.2	1.02	62.2
2	9.35	5.82	3.53	1.83	48.2	1.68	52.3
3	24.63	13.19	11.44	8.05	29.6	7.94	30.6
4	37.56	33.44	4.12	4.57	-10.7	4.32	-4.9
5	10.74	8.45	2.29	1.33	41.9	0.97	57.8
6	11.26	8.98	2.28	1.64	28.1	1.41	38.3
7	22.02	17.14	4.88	3.87	20.9	3.47	28.9
8	22.86	19.34	3.52	3.56	-1.1	3.20	9.1
9	90.75	64.63	26.13	17.43	33.3	18.15	30.5
10	29.78	21.59	8.19	5.90	28.0	5.51	32.7
11	158.59	135.19	23.40	17.47	25.3	17.05	27.2
12	109.39	81.75	27.64	18.66	32.5	18.87	31.7
13	127.61	114.20	13.41	10.50	21.7	9.89	26.3
14	163.50	142.40	21.10	13.52	36.0	13.49	36.1
15	128.94	106.74	22.20	11.14	49.8	10.96	50.6
av.	63.69	51.91	11.78	8.12	31.1	7.86	33.3

Table 2: Synchronization Overhead Estimates for 4 Processors

We interpret the data in Table 2 with the following reasoning:

1. Communication overhead is negligible, as found in Section 6.1. Since *Clocking* embodies only communication and synchronization losses, it accurately measures the latter. Now, comparing *Average Idle* columns in Table 2, the two measurement methods correlate well, especially for larger problems. Thus the *Node Counts* method is also accurate as a measure of synchronization loss.

2. *Cost per node* has the same expected value for all regions of the search tree, given the correctness of *Node Counts*. If this were not so, then some regions would be more expensive to search than others, and *Node Counts* would not generate average idle times that consistently agree with *Clocking*. Moreover, clocked working times per processor correlate well with node counts per processor [1], showing that *cost per node* is uniform over subtrees.

3. Poor speedups cannot be blamed on a higher distribution of expensive nodes in a parallel search tree, given a uniform *cost per node*. Therefore, sub-linear speedup, and hence overrun, are directly linked to lost potential in the parallel solution. More specifically, overrun is linked to synchronization loss, since communication and search overheads are negligible [1].

Our reasoning derives the conclusion that each of *Clocking*, *Node Counts*, and *Overrun* accurately measure synchronization loss. Unfortunately, although the two former methods correlate well, there is a non-trivial discrepancy when they are compared to the latter. The discrepancies are presented as percentages of *Overrun* in Table 2.

This inconsistency between arguably valid results is a matter for further investigation, pointing to the more general difficulty of determining the exact cause of reduced effective power in parallel solutions. Overheads other than communication, search, and synchronization might be involved. It is also possible that a systematic variation in *cost per node*, not manifested in our data, causes parallel search tree nodes to be more expensive than sequential ones on average.

One hypothesis was that losses due to additional startup costs in a parallel solution must be factored into the analysis. However, such overheads are invariant over problems, and the discrepancy between methods is proportional to *Overrun*, so the category of constant overheads was ruled out as a possible explanation.

6.4 Variance in Solution Times

In order to be certain that there was no significant variance in solution times under different Ethernet traffic, we solved one particular problem ten successive times [1]. There is a slight variation in node counts between runs, confirming the evidence of the 7 processor data in Table 1, namely that parallel search can be affected by small changes in the timing of inter-processor communication. However, the differences are not significant, and we conclude that variance in solution times was not a factor in our experiments.

7 Summary and Conclusions

Our goal in this study was to identify and quantify overheads that reduce the effective power of loosely coupled parallel systems. We chose vertex cover as a problem simpler than chess, but whose parallel solutions suffer comparable synchronization losses [8]. Given the analogy, we expect that insights that help to characterize overheads will transfer between the two problems.

Through replicating a previous study [15], we were able to compare two hardware and software systems, and assess how their differences influenced solution speed and choice of processor configuration. The greater speed of our systems identified a need to experiment with bigger problem instances, so that data is not affected by small and random variances introduced through parallelism. Using a single-level process tree led us to assess where best to place the master process, and to the conclusion that adding a non-standard processor to an otherwise homogeneous set has decreasing effect as the degree of parallelism increases.

In implementing the original search algorithm, we devised a faster algorithm that searches highly skewed trees [1]. The existence of this faster algorithm makes it clear that the effective power of a parallel solution can only be assessed relative to the fastest sequential solution available. Design of processor configurations to effectively search highly skewed trees in parallel has been undertaken [2].

Our experiments with the original search and processor scheduling algorithms showed negligible effects from search and communication overhead, isolating synchronization overhead as the detrimental factor. Estimates were computed using three empirical methods, only two of which converged to similar numbers. The discrepancy between the *Overrun* method and the *Clocking* and *Node Count* methods is a matter for further research, and highlights the difficulty of accurately identifying and quantifying overheads.

Once the source of losses in loosely coupled systems is clearly identified, analysis can be extended to processor scheduling methods more complex than *dfl*. One such method currently under study is the dynamic re-allocation of processors when they become idle, either on a buddy basis or by stopping a given subtree search and re-starting with more processors [3]. Another approach employs speculative computing, in which newly idle processors continue with the next phase of their work, after assuming that the outcome of searches underway on other processors will not affect their work choice [10].

References

[1] E. Altmann, T. Breitkreutz, and T.A. Marsland, "Overheads in Loosely Coupled Parallel Search," TR87-15, Computing Science Department, University of Alberta, pp. 1-27, Edmonton, July 1987.

[2] T. Breitkreutz, T.A. Marsland, and E. Altmann, "Parallel Search of Skewed Trees," TR87-16, Computing Science Department, University of Alberta, pp. 1-27, Edmonton, August 1987.

[3] R. Hyatt, "Parallel Search with Cray Blitz," *presented at ACM/IEEE FJCC Computer Chess Workshop*, p. abstract, Dallas, October 1987.

[4] Ten-Hwang Lai and Sartaj Sahni, "Anomalies of Parallel Branch-and-Bound Algorithms," *Communications of the ACM*, pp. 594-602, June 1984.

[5] Eugene L. Lawler, "Covering Problems: Duality Relations and a New Method of Solution," *SIAM Journal on Applied Mathematics*, pp. 1115-1132, September 1966.

[6] Guo-jie Li and Benjamin W. Wah, "Coping with Anomalies in Parallel Branch-and-Bound Algorithms," *IEEE Transactions on Computers*, pp. 568-573, June 1986.

[7] T.A. Marsland, M. Olafsson, and J. Schaeffer, "Multiprocessor Tree-Searching Experiments," pp. 37-51, 1985.

[8] T.A. Marsland and F. Popowich, "Parallel Game-Tree Search," *IEEE Transactions on PAMI*, pp. 442-452, July 1985.

[9] J. Mohan, "A Study of Parallel Computation - The Traveling Salesman Problem," TR CMU-CS-82-136, Computer Science Department, Carnegie Mellon University, Pittsburgh, August 1982.

[10] M. Newborn, "Unsynchronised Iteratively Deepening Parallel Alpha-Beta Search," *IEEE Transactions on PAMI*, 1988.

[11] M. Olafsson and T.A. Marsland, "A Unix Based Virtual Tree Machine," *CIPS'85 Congress Proceedings.*, pp. 176-181, Montreal, June 1985.

[12] Jonathan Schaeffer, "Improved Parallel Alpha-Beta Search," *ACM/IEEE FJCC Proceedings*, pp. 519-527, Dallas, 1986.

[13] Benjamin W. Wah, Guo-jie Li, and Chi Fen Yu, "Multiprocessing of Combinatorial Search Problems," *IEEE Computer*, pp. 93-108, June 1985.

[14] Benjamin W. Wah and Y. W. Eva Ma, "MANIP – A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems," *IEEE Transactions on Computers*, pp. 377-390, May 1984.

[15] N. Zariffa, "Implementation and Analysis of Three Parallel Branch-and-Bound Algorithms for the Vertex Covering Problem," M.Sc. Thesis, School of Computer Science, McGill University, pp. 1-96, Montreal, March 1986.

SORTING WITH LINEAR SPEEDUP ON A VLSI NETWORK

Peter Varman Kshitij Doshi

Electrical and Computer Engineering
Rice University
Houston, Texas 77251-1892.

Abstract

This paper presents a parallel algorithm to sort n data items on an architecture consisting of p processor-memory-switch modules interconnected in the topology of a binary cube. The switches form a pipelined, packet switched interconnection network that is used to route data between the processors. The time complexity, including both communication and computation costs is $\theta(n \log n / p + \log^2 p)$, yielding linear speedup for all p, $1 \le p \le n/\log n$.

1. Introduction

Sorting is a fundamental computational problem that has been investigated for several decades. Several parallel sorting algorithms that achieved poly logarithmic execution times and linear speedups on the idealized parallel random access machine model (e.g [3,5,7,8,11,16,17,21]) have been discovered. Efficient sorting on network and VLSI computational models are described in [1,4,6,13,14,18,20]. A coarse-grained parallel sorting algorithm for a (non-pipelined) hypercube [9] achieves linear speedup only for for $1 \le p \le \log n$. The reader is referred to [2,10] for several sorting methods, and for further references.

In this paper we present a coarse-grained parallel sorting algorithm that can be mapped onto a *pipelined hypercube architecture* of p PEs. The latter is one of a class of parallel architectures referred to as ensemble architectures [19]. These represent a cost-effective means of implementing parallel systems and are rapidly becoming available commercially. Our sorting algorithm requires $O(n \log n / p + \log^2 p)$ time, including both computation and communication costs, and thereby achieves linear speedup for all p, $1 \le p \le n/\log n$.

The rest of the paper is organized as follows. In Section 2, the pipelined hypercube architecture is described. Section 2.1 introduces communication graphs (CGs) and formalizes their relation to routing on the pipelined hypercube. In section 2.2 we show that communication traffic patterns arising in the sorting algorithm can be mapped onto the hypercube, to achieve conflict-free routing. Section 3 describes the sorting algorithm and its implementation on the pipelined hypercube.

2. Architectural Model

A *pipelined hypercube* network consists of $p = 2^k$ nodes, $k \ge 0$, indexed from 0 to $2^k - 1$, and connected in the topology of a *binary cube* of dimension k; i.e nodes i and j are connected, whenever the binary representations of i and j differ in exactly one bit. Figure 1(a) shows a binary cube for $p = 8$.

In the following $i_{k-1}i_{k-2}...i_1i_0$ denotes the binary representation of integer i, $0 \le i \le 2^k$ -1, and i_j denotes the integer whose binary representation differs from that of i in the bit numbered j. Each node in the binary cube is a processor-memory-switch (PMS) module. PMS[i] consists of a processor-memory module (denoted by PE[i]), and k switch boxes, $(0, i)$, $(1, i)$, ..., $(k-1, i)$, one for each dimension. PE[i] is connected by a *shared bus* to the switches in its module. Each switch (l, i), $0 \le l \le k-2$, is connected by a bidirectional, full-duplex *intra-module* link to switch $(l+1, i)$. *Inter-module* links connect switches in different PMS modules. Formally, for each pair of nodes, i and $\bar{i}_l$, that are connected in the binary cube, there is a bidirectional, full-duplex link between switch (l, i) and switch $(l, \bar{i}_l)$. Figure 1 (b) shows a PMS module for a three-dimensional cube.

The switches form a synchronous, pipelined packet-switched network that is used to transfer blocks of data between the PMS modules. Three types of communication traffic that arise in the sorting algorithm must be supported by the network. These will be referred to as *forward routing*, *reverse routing* and *cube routing* respectively. We now describe the functional requirements of the switches for each type of routing.

A cycle of a switch consists of an *odd* phase and an *even* phase. The odd phase consists of data transfer between switches in the same PMS module along intra-module links; in the even phase data is transferred between different modules using inter-module links. In forward routing, communication during the odd phase is from switch (l, i) to $(l+1, i)$, while for reverse routing it is from (l, i) to $(l-1, i)$. On receiving a packet, switch (l, i) decodes the destination address associated with the packet, and buffers it for transmission on either the intra-module link or the inter-module link to $(l, \bar{i}_l)$ as appropriate. If the packet is buffered for transmission on the intra-module link, the packet will be transferred to the switch $(l + 1, i)$ (or $(l - 1, i)$) in the odd phase of the next cycle. Otherwise, it will be transferred to $(l, \bar{i}_l)$ in the even phase of the current cycle.

Cube routing is employed to emulate the point-to-point connections of a binary cube. We require at most one switch to send (or receive) a packet to (from) the PE in its PMS module in the same cycle. Thus a shared bus between a PE and the switches in its module represents an adequate connection.

In the next section, a formal graph-theoretic model of the interconnection network is presented, and communication traffic patterns that arise in the sorting algorithm are proved to be conflict free.

2.1. Communication Graphs

A CG (communication graph) is a directed graph whose nodes represent switches and whose edges represent unidirectional communication links between the switches. Nodes with no incoming (outgoing) edges will be called *sources* (*sinks*). We define two CGs, F and R, on which the required traffic patterns are proved to be conflict free. We then show that a conflict-free set of routes in either F or R corresponds to conflict free routing on the pipelined hypercube.

Both F and R have p $(k + 1)$ nodes arranged in $k+1$ levels, with $p = 2^k$ nodes at each level. A node is denoted by (l, i), where l is the *level* number, $0 \le l \le k$, and i is the *index* of the node within the level, $0 \le i \le p - 1$. In F, a node (l, i) at level l, $0 \le l \le k-1$ is connected to the two nodes $(l+1, i)$ and $(l+1, \bar{i}_l)$, by edges directed from the former into the latter. In R, a node (l, i), $0 \le l \le k - 1$, is connected to the two nodes $(l+1, i)$ and $(l+1, \bar{i}_{k-l-1})$. F will be referred to as the *forward* network and R will be referred to as the *reverse* network. (See figure 2.)

A switch at level l, $0 \le l \le k$, examines a bit of the address associated with a packet, and passes it at the next cycle to a switch at level $l + 1$. We describe two routing operations that the switches support, namely *least significant bit* (LSB) routing and *most significant bit* (MSB) routing. The switches in F employ *LSB* routing while those in R employ *MSB* routing.

In LSB routing, node (l, i) of F $0 \le l \le k-1$, routes a packet to either node $(l+1, i)$ or to node $(l+1, \bar{i}_l)$ depending on whether the l^{th} bit $0 \le l \le k-1$, of the address field A_F matches the l^{th} bit of i or not, respectively. In MSB routing, node (l, i) of R, $0 \le l \le k-1$, routes a packet to either node $(l+1, i)$ or to node $(l+1, \bar{i}_{k-l-1})$ depending on whether the $k-l-1^{th}$ bit , $0 \le l \le k-1$, of the address field A_R matches the l^{th} bit of i or not, respectively.

The switches in R also support a variant of MSB routing referred to as *MSB routing with copy*. This is used to implement a *broadcast* facility, in which a data packet can be sent simultaneously from a node $(0, i)$ to *all* to the consecutively indexed destination nodes, (k, START_i), (k, START_i+1), ... , (k, END_i-1), (k, END_i). The address field A_R now consists of the pair of integers $(\text{START}_i, \text{END}_i)$, $\text{START}_i \le \text{END}_i$, which define the limits within which the packet must be sent. Each switch node (l, j), $0 \le l \le k-1$, performs the following actions on receiving a packet of this form. If the $k-l-1^{th}$ bits of START_i and END_i are the same, the node implements the usual MSB routing to route the packet to the node

indicated by the address START_i. If the two bits are different, then the packet is forwarded to *both* the nodes $(l+1, j)$ and $(l+1, \overline{j_{k-l-1}})$. However, the addresses START_i and END_i that are forwarded to the two nodes are updated as follows. The copy forwarded to the node with the smaller index will have END_i set to $2^k - 1$ and that forwarded to the node with the larger index will have START_i set to zero.

Definitions : The *route* in F (R) from node $(0, i)$ to node (k, j) is the ordered sequence of nodes in F (respectively R), $((0, i), (1, i^1), \ldots (l, i^l), \ldots (k, j))$, that a packet with address $A_F = j$ (respectively $A_R = j$) passes through. The sequence of edges between nodes in the route is the *path* of the route. A route in F is referred to as a *forward route*, while a route in R is referred to as a *reverse-route*. Two routes are said to be *conflict free* if they are node disjoint. A set of routes is conflict free if they are pairwise node disjoint.

We now relate F and R to the pipelined hypercube of section 2.1, and show how conflict-free routes in F (or R) imply link-disjoint routes in the pipelined hypercube.

In the following let H_F (H_R) refer to the graph obtained from F (respectively R) by replacing the directed edge $\langle (l, u), (l+1, \overline{u_l}) \rangle$ (respectively $\langle (l, u), (l+1, u_{k-l-1}) \rangle$) with the directed edge $\langle (l+1, u), (l+1, \overline{u_l}) \rangle$ (respectively $\langle (l+1, u), (l+1, \overline{u_{k-l-1}}) \rangle$). Figure 3 shows H_F and H_R corresponding to F and R of figure 2. To demonstrate the similarity of the switches and links of the pipelined hypercube network to the nodes and edges of H_F and H_R, the nodes in figure 2 have been renumbered as follows. Node (l, u) in H_F is renamed $(l-1, u)$, and node (l, u) in H_R is renamed $(k-l, u)$, $1 \le l \le k$. Node $(0, u)$ is shown as PE[u]. It may be seen that with this renaming of nodes, H_F maps directly onto the switches and links of the hypercube network used for forward routing, and H_R to the switches and links used for reverse routing.

Definition : A route in H_F is obtained from a route in F by replacing every edge $\langle (l, u), (l+1, \overline{u_l}) \rangle$ in the latter path, by the two directed edges $\langle (l, u), (l+1, u) \rangle$ and $\langle (l+1, u), (l+1, \overline{u_l}) \rangle$, $0 \le l \le k-1$. Similarly, a route in H_R is obtained from a route in R by replacing every edge $\langle (l, u), (l+1, \overline{u_{k-l-1}}) \rangle$ by the two directed edges $\langle (l, u), (l+1, u) \rangle$ and $\langle (l+1, u), (l+1, \overline{u_{k-l-1}}) \rangle$, $0 \le l \le k-1$.

Theorem 2.1 : Let P_1 and P_2 be the paths of two node disjoint routes in F (or R) and $\hat{P_1}$ and $\hat{P_2}$ be the corresponding paths in H_F (respectively H_R). Then $\hat{P_1}$ and $\hat{P_2}$ are edge disjoint.

Proof: By contradiction. If an edge e exists that is common to $\hat{P_1}$ and $\hat{P_2}$ then,

if $e = \langle (l, u), (l+1, u) \rangle$, the node (l, u) is common to both P_1 and P_2,

else $e = \langle (l, u), (l, \hat{u}) \rangle$, where $\hat{u} = \overline{u_{l-1}}$ or $\overline{u_{k-l}}$ according to whether P_1 and P_2 are from F or R, and it follows that the node $(l-1, u)$ is common to the two paths.

The theorem shows that if the required set of routings can be shown to be conflict-free in F (or R), then the routes using forward (respectively reverse) routing on the pipelined hypercube will be link disjoint.

2.2. Conflict Free Routing

In a series of lemmas we describe several routing patterns that arise in the sorting algorithm, and show that they are conflict-free in F or R. Closely related results for performing certain of these routings on a binary cube have been previously reported in [15]. As a consequence of Theorem 2.1, these routings can be performed without link conflict in the pipelined hypercube using forward or reverse routing respectively.

Lemma 2.1: Let (l, u) be a node on the route from $(0, i)$ to (k, j) in F (R). Then the binary representation of u is $i_{k-1}i_{k-2}...i_l\ j_{l-1}j_{l-2}...j_0$ (respectively, $j_{k-1}j_{k-2}...j_{k-l}\ i_{k-l-1}i_{k-l-2}...i_0$).

Proof: Direct consequence of LSB (MSB) routing.

Lemma 2.2:Let $(0, s(i))$, $(k, t(i))$), $0 \le i \le r-1$, be a collection of r pairs such that, $0 \le s(0) < s(1) < \ldots < s(r-1) \le 2^k - 1$, $0 \le t(0) < t(1) < \ldots < t(r-1) \le 2^k - 1$, and $s(i+1) - s(i) \ge t(i+1) - t(i)$, for all i, $0 \le i \le r-2$. Then the set of routes in F from each node $(0, s(i))$ to the node $(k, t(i))$ is conflict-free.

Proof:Consider any pair i, j and without loss of generality assume that $i > j$. Let $u = s(i)$, $v = s(j)$, $x = t(i)$ and $y = t(j)$.

Assume, contrary to the lemma, that (l, w) is a node that is common to the two routes. From lemma 2.1 above, $w = u_{k-1}u_{k-2}...u_l\ x_{l-1}x_{l-2}...x_0 = v_{k-1}v_{k-2}...v_l\ y_{l-1}y_{l-2}...y_0$. Then, $u - v < 2^l$ and $x - y \ge 2^l$, which contradicts the fact that $u - v \ge x - y$. Since i and j were arbitrary, the set of routes is conflict free.

A similar lemma holds for the routes in R:

Lemma 2.3 : Let $(0, s(i))$, $(k, t(i))$), $0 \le i \le r-1$ be a collection of r pairs such that, $0 \le s(0) < s(1) < \ldots < s(r-1) \le 2^k - 1$, $0 \le t(0) < t(1) < \ldots < t(r-1) \le 2^k - 1$, and, $s(i+1) - s(i) \le t(i+1) - t(i)$, for all i, $0 \le i \le r-2$. Then the set of routes in R from each node $(0, s(i))$ to the node $(k, t(i))$ is conflict-free.

Definition: *Broadcast* routing is defined as follows. Let $\{ (0, i) \mid 0 \le i \le r-1 \}$, be a set of sources in R. Associated with each source $(0, i)$, is a pair of integers, START_i and END_i such that $\text{END}_i \ge \text{START}_i$. For all i, $0 \le i \le r-2$, $\text{START}_{i+1} = \text{END}_i + 1$, and $\text{END}_{r-1} = p-1$. Route data from $(0, i)$ to all (k, u), $\text{START}_i \le u \le \text{END}_i$.

Broadcast routing is performed using *MSB with copy* in R. The following lemma can be shown to hold (see [22]) for broadcast routing:

Lemma 2.4 : Let $(0, i)$ and $(0, j)$ be two sources involved in a broadcast operation. Then the routes from $(0,i)$ to (k, u) and from $(0, j)$ to (k, v), for any pair u, v, $\text{START}_i \le u \le \text{END}_i$ and $\text{START}_j \le v \le \text{END}_j$ are conflict free.

Corollary : *Broadcast* routing is conflict free.

Combinations of special cases of the routings implied in the above lemmas arise in the sorting algorithm. The special case of lemma 2.3 when $t(i) = i$, is known as *Concentrate*. A *Concentrate* followed by a *Broadcast* is known as *Scatter*. The special case of lemma 2.3, when $s(i) < t(i)$, $0 \le i \le r-1$, is known as *Right Transfer* routing; the case when $s(i) > t(i)$ for all i, is correspondingly referred to as *Left Transfer*. A *Left Transfer* followed by a *Right Transfer* is referred to as a *Weave*. All of these routings can be performed using link disjoint routes on the pipelined hypercube.

3. Main Algorithm

The sorting procedure is a parallelization of the standard merge-sort as shown below in the procedure *Parallel Merge Sort*. The crux of the procedure is the parallel algorithm for merging two ordered lists, as described in section 3.1. Section 3.2 contains the complexity analysis.

Parallel Merge Sort ($W[0 .. N-1]$, M)

*/*Sort array W using $M = 2^K$ PEs. Let $m = N/M$ */*

/ Let $W[im .... (i+1)m - 1]$ be denoted by W_i */*

1. PE[i], i, $0 \le i \le M - 1$, is loaded with W_i.
2. Each PE[i] independently sorts W_i into increasing order.
3. For $j = 0$ to $K - 1$ do
   For each s, $0 \le s \le 2^{K-1-j} - 1$, do independently and concurrently :
   / Let $i = s\ 2^{j+1}$ */*
   / Let $A_s = (W_i ... W_{i+2^j-1})$ and $B_s = (W_{i+2^j} ... W_{i+2^{j+1}-1})$ */*
   Merge A_s and B_s using procedure *Coarse Merge*($A_s, B_s, 2^{j+1}$).
   End.

End Parallel Merge Sort.

3.1. The Merging Procedure *CoarseMerge*

Let A[0 ...$n-1$] and B[0 ...$n-1$] be the two sorted arrays of elements to be merged, where the elements are drawn from a totally ordered set. The arrays will be merged using $2p$, $p = 2^{k-1}$, PEs. Let $m = n/p$. The elements A[im...$(i+1)m$-1] are stored in PE[i] and elements B[im...$(i+1)m$-1] in PE[$p + i$], $0 \le i \le p-1$, and are referred to as *blocks* A_i and B_i respectively. The output of the merge is the sorted array C[0...$2n-1$], such that PE[i] contains elements C[im...$(i+1)m$-1].

The merging algorithm consists of three phases. The first phase consists of a *fine-grained* merging procedure that is performed on a list of representative elements, one from each block. The outcome of the fine-grained merge is used to *pair* each block of A with some block of B and each block of B with some block of A. If block A_u (B_u) is paired with block B_v (A_v), then in the second phase, B_v (A_v) is transferred to the PE containing A_u (B_u). After this data transfer step, a PE will have two blocks A_u and B_v. The PE merges the two blocks into a single sorted list, removing duplicates in the process. In the final phase, the individually merged blocks are routed to the appropriate processors, so that the output is ordered according to the chosen convention.

In the first phase, elements A[im] (and B[im]), the smallest elements in each of the blocks and denoted by a_i (respectively b_i), are merged into a single sorted subsequence $\Gamma = t_0, t_1 \ldots t_{2p-1}$.

Definitions : An element t_i, $0 \le i \le 2p-1$, in Γ is a *border* element if either $t_i = a_v$ and $t_{i+1} = b_w$, or $t_i = b_v$ and $t_{i+1} = a_w$ $(0 \le v, w \le p-1)$. It is convenient to define non-existent elements t_{-1} and t_{2p} as border elements. A subsequence of Γ consisting of elements $t_i, t_{i+1}, \ldots t_{i+r}$ is a *run* if none of $t_i, t_{i+1}, \ldots t_{i+r-1}$ are border elements. If $i = 0$, then the run is called the *first run.* For each i, $0 \le i \le 2p-1$, *partner* $(i) = k$ if t_k is a border element, and $t_{k+1}, t_{k+2}, \ldots t_i$ is a run. If t_i occurs in the first run then *partner* (i) = -1. For each element t_i, $0 \le i \le 2p-1$, if $t_i = a_j$, $0 \le j \le p-1$, then define *block* T_i to be A_j ; if $t_i = b_j$ then define T_i to be B_j. It is convenient to consider the non-existent block T_{-1} as a block of m elements, each of value $-\infty$ (the smallest possible value), and the non-existent block T_{2p} as a block of m elements each of value $+\infty$ (the highest possible value).)

For phase 2 of the algorithm, block T_i is paired with $T_{partner(i)}$. An example for the case $p = 4$, is shown in figure 4. PE[0] through PE[3] contain the 4 blocks of A, A_0, A_1, A_2 and A_3, and PE[4] through PE[7] contain the blocks B_0, B_1, B_2 and B_3 respectively. The pairings of blocks, as required by the *partner* relation are shown by arrows. In the example, B_0, B_1 and B_2 are all paired with block A_1, A_2 is paired with B_2, B_3 is paired with A_2 and A_3 is paired with B_3. Each block is sequentially merged with the block with which it is paired. When merging block T_i with $T_{partner(i)}$, the PE_i removes duplicates by enforcing the following *uniqueness conditions*:

(i) discard all elements less than t_i (ii) discard all elements greater than or equal to t_{i+1}.

In the example, the merge of B_0 and A_1 retains only the elements in the range $[b_0, b_1)$. Similarly the merge of A_2 with B_2 retains only the elements in the range $[a_2, b_3)$; and the merge of B_3 with A_2 retains only the elements in the range $[b_3, a_3)$.

Let U_i, $0 \le i \le 2p-1$ be the block resulting from the merge of T_i and $T_{partner(i)}$ and the removal of duplicates. $|\ U_i\ |$ denotes the cardinality of U_i. Let $U_i = [u_{i,0}, u_{i,1}, \ldots, u_{i,r}]$, with $|\ U_i\ | = r + 1$. Define $rank(u_{i,j})$, $0 \le j \le r$, as the number of elements of A and B that are less than $u_{i,j}$. Let α_i and β_i be the number of elements removed due to uniqueness conditions (i) and (ii) respectively. (Define $\alpha_i = m$ for blocks paired with T_{-1}, and define $\beta_{2p-1} = 0$.)

In the final step, we construct the block C_i of the sorted sequence by discarding the first $m - \alpha_i$ elements of U_i and concatenating to the end of the list that remains, the first $m - \alpha_{i+1}$ elements of U_{i+1}. Corollary to lemma 3.4 in this section will establish that this can be done concurrently. (Proofs of the following lemmas have been omitted due to space limitations and are available in [22].)

Lemma 3.1 : For each i, $0 \le i < 2p-1$, $\beta_i + \alpha_{i+1} = m$.

Lemma 3.2 : For each i, $0 \le i \le 2p-1$, $|\ U_i\ | = 2m - (\alpha_i + \beta_i)$.

Lemma 3.3 : For each i, $0 \le i \le 2p-1$, $rank(u_{i,0}) = m\ (i-1) + \alpha_i$.

Lemma 3.4 : For each i, $0 \le i \le 2p-1$, $|\ U_i\ | > m - \alpha_i$.

Corollary : $(m - \alpha_i)$ elements can be simultaneously transferred from U_i to U_{i-1}, $0 \le i \le 2p-1$.

Theorem 1 : The sequence of blocks $C_0\ C_1\ C_2 \ldots C_{2p-1}$ is sorted, with $|\ C_i\ | = m$ for each i, $0 \le i \le 2p-1$.

Proof : $|\ C_i\ | = |\ U_i\ | + \alpha_i - \alpha_{i+1} = m$ from lemmas 3.1 and 3.2. That all elements of C_{i-1} are in sorted order and smaller than elements of C_i is immediate from the fact that U_i are in sorted order.

Following is the formal description of the procedure *Coarse Merge*.

Procedure Coarse Merge (A, B, p)

/ Merge 2 sorted sequences A and B, each of size ½m p, on a p PE system. The first p/2 PEs contain elements of A, while the last p/2 contain elements of B. */*

denote : the processor element i by PE_i, the data in PE_i by D_i, and the minimum element of D_i by d_i.

define the following macros :

left(i) = {-1 if i = 0 or $p/2$; i-1 otherwise}

right(i)= {p if $i = p/2$-1 or p-1; i+1 otherwise}

border(i) = {TRUE iff RIGHTRANK(i) > RANK(i) + 1}.

firstrun(i) = {(($i < p/2$) ∧ (RANK(i) = i)) ∨ (($i \ge p/2$) ∧ (RANK(i) = $i - p/2$))} /* BOOLEAN */

The following integers are used by each PE_i:

RANK(i) : *the rank of* d_i *following the fine-grained merge. Define* RANK[-1] = -1 *and* RANK[p] = p.

RIGHTRANK(i) : *the rank of* $d_{right(i)}$.

PARTNER(i) : *the block with which* D_i *is paired.*

RIGHTLIMIT(i) : = d_j, *where* RANK[j] = RANK[i]+1.

NEXT(i) : *defined if* (*border*(i)); = *Id. of next ranked block.*

BEGIN-SCATTER(i) : *defined if* (*border*(i)); = *Id. of the first block to pair with* D_i.

END-SCATTER(i) : *defined if* (*border*(i)); = *Id. of the last block to pair with* D_i.

α_i : *Elements discarded due to condition* 1.

COUNT(i) : *Number left after duplicate removals.*

BORDERRANK(i) = *number of* PE_j, $j < i$, *such that both* j *and* i *are from the same half of the PE array and border*(j) *is* TRUE.

begin

1 (a). Each PE_i, $0 \le i \le p-1$, creates a record $X_i = \langle$ VAL, INDEX $\rangle$ with VAL = d_i and INDEX = i.

1 (b). The processors sort $\{ X_i \}$ using a fine-grained parallel computation. At the end of this step, each PE_i, has RANK(i) and RIGHTLIMIT(i) correctly initialized.

2 (a). Every PE_i except PE_0 and $PE_{p/2}$ creates the record $R = \langle$ rightrank $\rangle$, with rightrank = RANK(i), and sends it to PE_{i-1}.

2 (b). Every PE_i except $PE_{p/2-1}$ and PE_{p-1} receives records R sent in step 2 (a). All PE_i perform the following assignments:

1. if (*right*(i) ≠ p) then RIGHTRANK(i) = R.rightrank
   else RIGHTRANK(i) = p.
2. if (*NOT*(*firstrun*(i)) then PARTNER(i) = *RANK*(i) - i + $p/2$ -1.
3. if(*border*(i)) then
   if(*firstrun*(i)) then NEXT(i) = *rank*(i) - i + $p/2$
   else NEXT(i) = *right*(PARTNER(i))
   BEGIN-SCATTER(i) = NEXT(i)
   if(right(i) ≠ p) then END-SCATTER(i) = NEXT(i) + RIGHTRANK(i) - RANK (i) - 2
   else if($i = p/2 - 1$) then END-SCATTER(i) = $p - 1$
   else END-SCATTER(i) = $p/2 - 1$.

3. Using FINE-GRAINED PREFIX, each PE_i such that *border*(i) is TRUE computes BORDER-RANK(i).

/ Border PEs scatter data blocks */*

4. if(*border*(i)) then **scatter**(D_i, $A_f(i)$, {$A_{r1}(i)$, $A_{r2}(i)$}) where, A_f = BORDER-RANK(i), A_{r1} = BEGIN-SCATTER(i), and A_{r1} = END-SCATTER(i).

5. Each PE_i, $0 \le i \le p-1$ such that *firstrun*(i) = FALSE receives the block $D_{\text{PARTNER}(i)}$ sent in step 4, merges the block D_i with the block $D_{\text{PARTNER}(i)}$, discards duplicates (elements $< d_i$ or $\ge$ RIGHTLIMIT(i)), and sets α_i.
   Each PE_i such that *firstrun*(i) is TRUE, sets $\alpha_i = m$.
   Let V_i denote the data that remains in PE_i. COUNT(i) = $|\ V_i\ |$.

/ Permute data blocks into increasing order of rank */*

6 (a). Each PE_i sends (V_i, COUNT(i), α_i) to $PE_{RANK(i)}$.

6 (b). Each PE_i receives the data sent out from some PE_j (such that i = RANK(j)) in step 6 (a). Let U_i denote the data received by PE_i; i.e., $U_i = V_j$. PE_i sets $\alpha = \alpha_j$ and COUNT = COUNT(j).

/ Equalize data in each PE */*

7 (a). Each PE_i, $0 < i \le p-1$ sends the first m-α elements of U_i to PE_{i-1}, and deletes them from U_i.

7 (b). Each PE_i, $0 \le i < p-1$ receives the elements sent in step 6(a), and concatenates them to U_i.

end

Steps 1 and 3 of *Coarse Merge* consist of fine-grained parallel computations. By fine-grained computation we mean computations which deal with sparse data sets, one data record per processor. In step 1 the two sequences of representative elements $a_0 \ldots a_{p/2-1}$ and $b_0 \ldots b_{p/2-1}$ are

merged into the sorted sequence Γ. This is most simply performed using the well known bitonic merge algorithm to merge the records X_i, $0 \le i \le p/2-1$ and X_i, $p/2 \le i < p-1$. (The implementation is standard and details appear in [22].) Following bitonic merge, PE[i], $0 \le i \le p-1$, contains the record X_j, whose rank is equal to i. The *rank* and the value of the next higher element are returned to PE[j]. At the end of step 1, each PE[i] has its local variable RANK(i) set equal to the position of its representative in Γ and its variable RIGHTLIMIT(i) set to the value of $t_{RANK(i)+1}$, the representative immediately to its right in Γ. Figure 5(a) shows the situation assuming the sorted list of representatives of Figure 4.

In step 2, each PE determines if its representative is a *border* element, and if so determines the range of addresses over which it has to broadcast its data block. PE[i], $i \neq p/2-1$ and $i \neq p-1$, is a border PE only if RANK(i+1) > RANK(i) + 1. (PE[$p/2-1$] and PE[$p-1$] are handled specially.) In step 3 each border PE[i], $0 \le i \le p-1$ determines BORDER-RANK(i), the number of border PEs to its left and in its half of the processor array. This information is used to route the data blocks in a conflict free manner in step 4.

In step 4, the requisite data blocks are broadcast. The details are discussed in section 3.1.1. With respect to the example (see figure 5(b)), PE[1] broadcasts block A_1 to PE[4], PE[5] and PE[6], A_2 is broadcast to PE[7], B_2 to PE[2] and B_3 to PE[3]. In step 5, processors merge the resident block with the block broadcast to it, and remove duplicates to obtain the sorted blocks, V_i, $0 \le i \le p-1$. Note that the blocks will be in permuted order, the permutation being determined by RANK(i) (see figure 5(c)). In step 6 the blocks are permuted into rank order. Finally in step 7, each PE[i] transfers $m - \alpha_i$ elements to PE[$i-1$], to equalize the number of elements in each PE.

The FINE-GRAINED-PREFIX operation of step 3 of Coarse Merge is implemented as follows. Each PE[i] such that *border*(i) is true sets a flag, *Flag*(i) to one; the other PEs set *Flag*(i) to zero. Each half of the processor array performs a parallel prefix computation [15] (also referred to as partial sums computation), to determine the sum of the flag values that lie in its half and with indices smaller than it. (The details of the implementation on a hypercube appear in [22].)

3.1.1. Mapping on the Network

In this section we discuss how the various steps of procedure *Coarse Merge* can be implemented on the pipelined hypercube. Step 5 consists of a sequential merging algorithm executed concurrently and independently by each PE. Transfers of data blocks are required in steps 4, 6 and 7. We discuss each of these now, starting with the simplest.

Step 7 consists of a *Left-Shift* by one. By Lemms 2.2 (or 2.3), this can be accomplished by the in a conflict-free manner, by using *LSB routing* (or MSB routing) on the *forward network*.

The data transfer required by step 6 is a *Weave* routing. It is therefore performed in two stages. The first stage is a *Right-Transfer* where all blocks A_i are transferred from PE[i], $0 \le i \le p/2-1$, to PE[RANK(i)]. The second stage is a *Left-Transfer* where all blocks B_i are transferred from PE[$i + p/2$], to PE[RANK(i)]. Note that for each pair A_i, A_j, (and for each pair B_i, B_j), $j > i$, RANK[j] - RANK[i] $\ge j - i$. Hence, in both cases the conditions for conflict-free routing are satisfied. Each stage can be accomplished using *MSB routing* on the *reverse network*.

Data routing required in step 4 is also done in two stages. In the first stage, all blocks from a *border* PE[i] are transferred to PE[BORDER-RANK(i)]. This is a *Concentrate* operation performed independently on each half of the processor array and is conflict-free under LSB routing on the *forward* network (Lemma 2.2). In the second stage, the concentrated blocks are broadcast from PE[BORDER-RANK(i)] to PEs in the range PE[BEGIN-SCATTER(i)] to PE[END-SCATTER(i)]. Note that both the broadcast operations can occur concurrently.

The remaining steps can be performed directly using the *cube routing* portion of the network. At the end of step 1, ranks of X_i are returned to PE[i], which is exactly the reverse of that occuring in step 5 -- except that only 1 record per PE is involved. This can be accomplished in a 2 step conflict-free manner: first all PE[i] containing representatives from A send their records using LSB routing; next those containing records from B do the same. The time required is upper-bounded by 2 log p routing steps.

3.2. Complexity Analysis

The time required for procedure *Coarse Merge* can be upper bounded as follows. Note that the algorithm uses p PEs and each PE has m data items. The time required for the fine-grained computations in steps 1,2 and 3 is O(log p). The data routing required in steps 4, 6 and 7 can be performed in O(m + log p) time. The sequential merging of two blocks of size m each in step 5, requires O(m) time. Thus the time for procedure *Coarse Merge* is given by :

$$T_{merge}(m, p) = O(m + \log p).$$

The time required for procedure *Parallel Merge Sort*, $T_{sort}(N, P)$, is derived as follows. Note that the procedure sorts a list of N items using $P = 2^K$ PEs. The time required to perform the independent sort in each PE, using sequential mergesort (for instance) is O(N/P log(N/P)). The time for the j^{th} iteration, $1 \le j \le K$, is $T_{merge}(N/P, 2^j)$. Hence :

$$T_{sort}(N, P) \le c_1(N/P + N/P \log(N/P)) + c_2 \sum_{j=0}^{K} (N/P + j)$$

which is O(($N \log N$)/P + $\log^2 P$).

Selected References

1. M. Ajtai, J. Komlos, E. Szemeredi, "An O(n log n) Sorting Network," *Combinatorica* **3**. (1983)
2. S.G. Akl, "Parallel Sorting Algorithms," Notes and Reports in Comp. Sci. and App. Math., Academic Press, Orlando. (1985)
3. S.G. Akl and N. Santoro, "Optimal Parallel Merging and Sorting Without Memory Conflicts," *IEEE Trans. Comput.*, C-36, 11.
4. K.E. Batcher, "Sorting Networks and their Applications," *Proc. AFIPS Spring Joint Computer Conf.*, 32, (1968) pp. 307-314.
5. G. Bilardi and A. Nicolau, "Bitonic Sorting with O(n log n) Comparisons," *Proc. 20th Ann. Conf. Info. Sci. and Sys.*, (1986).
6. G. Bilardi and F. P. Preparata, "A Minimum Area VLSI Network for O(log n) Time Sorting," *IEEE Trans. Comput.*, C-34, 4. (1985).
7. A. Borodin and J. Hopcroft, "Routing, Merging and Sorting on Parallel Models of Computation" *Proc. 14th ACM Symp. Theory of Computing*.
8. R. Cole, "Parallel Merge Sort," *Proc. 27th IEEE Symp. Foundations of Computer Science*.
9. L. Johnsson. "Combining Parallel and Sequential Sorting on a Boolean n-Cube," *Proc. Intl. Conf. on Parallel Processing*. (1979).
10. D. E. Knuth, The Art of Computer Programming Vol. 3., *Addison-Wesley*, Reading MA, (1973).
11. C.P. Kruskal, "Searching, Merging and Sorting in Parallel Computation," *IEEE Trans. Comput.*, C-32, 10. (1983).
12. C. P. Kruskal, L. Rudolph and M. Snir, "The Power of Parallel Prefix," *IEEE Trans. Comput.*, C-34, 10. (1985).
13. F.T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Trans. Comput.*, C-34, 4. (1985).
14. D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Trans. Comput.*, C-28, 1.
15. D. Nassimi and S. Sahni, "Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network," *JACM*, 29(3).
16. F.P. Preparata, "New Parallel Sorting Schemes," *IEEE Trans. Comput.*, C-27, 7. (1978).
17. Y. Shiloach and U. Vishkin, "Finding the Maximum, Merging, and Sorting in a Parallel Computation Model," *J. Algorithms*, 2. (1981).
18. H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. Comput.*, C-20, 2. (1971).
19. C. Sietz "Concurrent VLSI Architectures" *IEEE Trans. Comput.*, C-33, 12. (1984).
20. C. D. Thompson and H. T. Kung, "Sorting on a Mesh-Connected Parallel Computer," *Comm. ACM*, v. 20, 4.
21. L. Valiant, "Parallelism in Comparison Problems," *SIAM J. Comput.*, 4, 3. (1975).
22. P. Varman and K. Doshi, "Sorting With Linear Speedup on a VLSI Network," TR- 8802, Rice University. (Feb. 1988.)

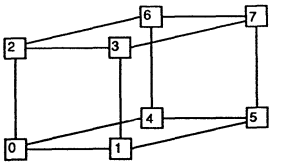
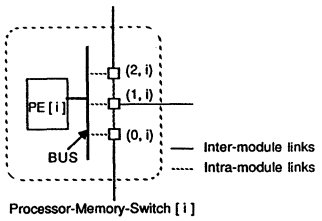
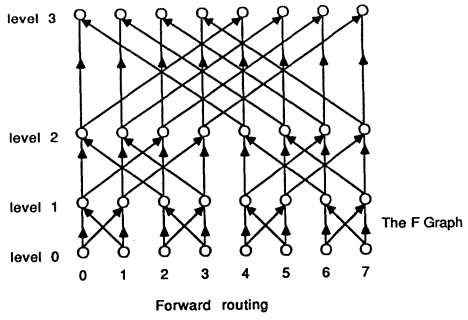


Figure 1(a)

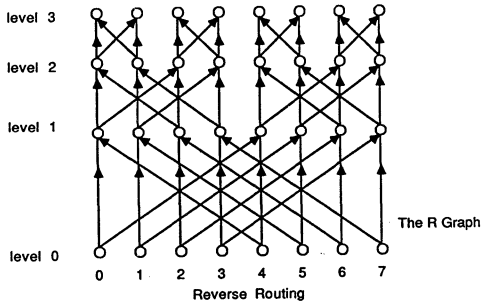


Processor-Memory-Switch [i]

Figure 1(b)



Forward routing



Reverse Routing

Figure 2

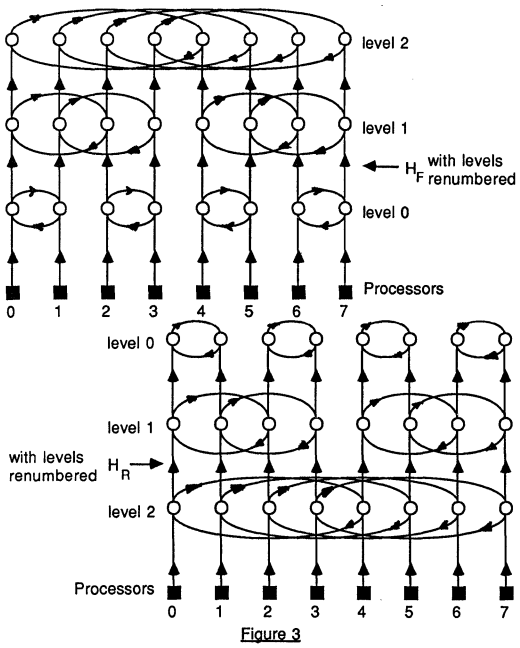
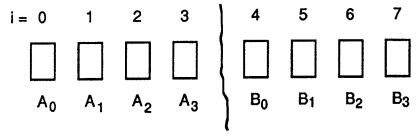


Figure 3



RANK[i] = 0 1 5 7 2 3 4 6

BORDER[i] = T T T T

Note that a_1 , b_2 , a_2 and b_3 are border elements

Sorted sequence (of representatives) : $a_0 a_1 b_0 b_1 b_2 a_2 b_3 a_3$

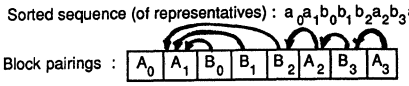
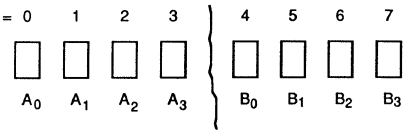


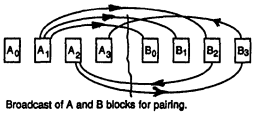
Figure 4.



RANK[i] = 0 1 5 7 2 3 4 6

RIGHTLIMIT[i] = a_1 b_0 b_3 – b_1 b_2 a_2 a_3

Fig 5(a) : Situation after step 1 of Coarse Merge, for the example of figure 4.



Broadcast of A and B blocks for pairing.

i	A_f =BORDER-RANK[i]	A_{r1}=BEGIN-SCATTER[i]	A_{r2}=END-SCATTER [i]
1	0	4	6
2	1	7	7
6	4	2	2
7	5	3	3

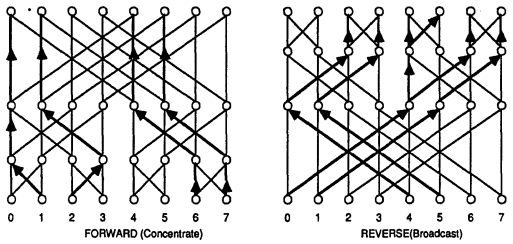
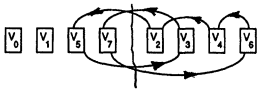


Figure 5(b)



The blocks are permuted into rank order.

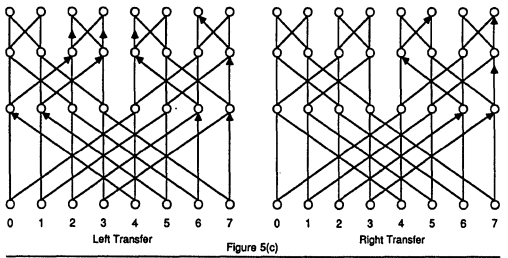


Figure 5(c)

Concurrent Insertions and Deletions in a Priority Queue*

V. Nageshwara Rao and Vipin Kumar†
Department of Computer Sciences,
University of Texas at Austin,
Austin, Texas 78712

Abstract

The heap is an important data structure used as a priority queue in a wide variety of parallel algorithms (e.g., multiprocessor scheduling, branch-and-bound). In these algorithms, contention for the shared heap limits the obtainable speedup. This paper presents an approach to allow concurrent insertions and deletions on the heap. Our scheme has much lower overheads and gives a much better performance than a previously reported scheme. The scheme also retains the strict priority ordering of the serial access heap algorithms; i.e., a delete operation returns the best element of all elements that have been inserted or are being inserted. Our experimental results on the BBN Butterfly parallel processor demonstrate that the use of concurrent heap algorithms in parallel branch-and-bound improves its performance substantially.

1 Introduction

The heap is an important data structure used as a priority queue in a wide variety of parallel algorithms (e.g., multiprocessor scheduling, branch-and-bound[10]). In these algorithms each processor performs an access-think cycle. Every processor executes its current subproblem at hand (thinking), then accesses the shared heap to insert subproblems if it generated any and takes the best available subproblem in the heap to solve next. Since many processors are sharing the heap and they may access the heap at the same time, the simplest way to provide consistency in updates is to serialize the updates. A lock is associated with the heap and the processors access the heap under mutual exclusion. This serial access scheme limits the number of processors that can be used to speedup the problem. If T_{think} is the mean think time and T_{access} is the mean access time, then clearly the maximum speedup achievable is

$$\leq \frac{T_{access} + T_{think}}{T_{access}}$$

T_{think} is a characteristic of the problem being solved. T_{access} depends on the priority structure being used. For the heap, T_{access} is $O(\log N)$, where N is the size of the heap.

One way to alleviate the limitation is to let many processors access the heap simultaneously. Updates on different parts of a heap can proceed concurrently provided they do not interact with each other. Let us view the heap as a binary tree with the root at the top and leaves at the bottom. In the ordinary serial heap algorithm, the deletes manipulate the heap level by level going from top to bottom, while inserts manipulate it from bottom to top. Hence many insertions (or many deletions) can be executed in parallel by using a simple locking scheme[1]. But inserts and deletes can not be active together, as they proceed in opposite directions and hence can deadlock. Biswas and Browne[1] present a scheme to handle this problem. But their scheme has a substantial overhead, and performs worse than the sequential heap unless the heap size N is very large.

This paper presents a new concurrent heap access scheme that has small overhead, and is able to perform better than the sequential heap even for small heaps. Two important ingredients of this scheme are (i) a heap insertion algorithm which manipulates the heap from top to bottom; and (ii) a scheme to combine a delete operation with the most recent unfinished insertion operation. Since these new insertions and the deletions move from top to bottom in the heap, they can both be active together without causing deadlock.

2 Preliminaries

A heap is a complete binary tree of depth d[5]. with the property that the value of the key at any node is less than the value of the keys at its children (if they exist).

It is efficient to implement the heap using an array. The root occupies location 1 and the node i occupies location i. The children of node i occupy locations 2i and 2i+1. The parent of node i is at $\lfloor \frac{i}{2} \rfloor$. We assume that each node in the heap has a key pointing to a field of data. Key(i) denotes the key located at node i. VALUE(i) denotes the value or priority order of the key at node i. Empty nodes in the heap are assumed to have keys with value MAXINT ($= \infty$).

We denote the left son and the right son of node i by LSON(i) and RSON(i) respectively. The parent of node i is denoted by PARENT(i). Associated with the heap are the data fields **lastelem** and **fulllevel**[1]. **lastelem** is the index of the last non-empty node of the heap. The keys of all nodes beyond **lastelem** is MAXINT. **fulllevel** is the index of the first node in the deepest level of the heap (that contains at least one non-empty node). For an empty heap, lastelem = fulllevel = 0. Fig 1 shows a sample heap of twelve elements, and the value of lastelem and fulllevel.

*This work was supported by Army Research Office grant # DAAG29-84-K-0060 to the Artificial Intelligence Laboratory, and Office of Naval Research Grant N00014-86-K-0763 to the computer science department at the University of Texas at Austin.

†Tel:512-471-9571,Arpanet: Kumar@sally.utexas.edu

[1]The conventional delete and insert operations[5] do not need to maintain **fulllevel** but it is needed for the insert_t operation which traverses the heap from top to bottom.

The operations supported on a heap are insertion and deletion. The insert operation inserts a new key, nkey, in the heap and the delete operation returns the smallest key in the heap. The reader is referred to [5] for the details of these serial-access heap algorithms.

3 Inserting from Top

It is possible to perform insertions from the top by using the following (informally stated) algorithm[2]:

```
k ←1;
if VALUE(k) > VALUE(nkey)
then Exchange(key(k),nkey)) ;
while (k has both successors)
        k ←any successor of k;
        if VALUE(k) > VALUE(nkey)
        then Exchange(key(k),nkey)) endif
endwhile
Put nkey at one of the empty leaves of k.
```

This naive insertion algorithm is not guaranteed to grow the heap level-by-level, which is crucial for the efficiency of insertions and deletions.[3] Our new insertion algorithm, which we call insert_t, performs reheapification in such a way that each insertion adds a key to the first empty node in the heap (just as in the conventional insert operation).

Let **target** be the first empty node in the heap. The **insertion path** is the path between the root and target. This path is unique (and can be easily computed) because the heap has a tree structure. Values of the nodes on the insertion path (from root to target) are nondecreasing. To insert a new key in the heap, we need to put the new key at a proper node on the insertion path, and move all the keys at and below this node one level down (filling the target node). The conventional insert algorithm does this by visiting the nodes on the insert path from bottom to top. The **insert_t** algorithm given below does it in the opposite order.

```
insert_t(nkey,heap)
Lock(heap)
lastelem ←lastelem + 1 ;
target ←lastelem ;
if (lastelem ≥ fulllevel* 2)
then fulllevel = lastelem endif
i ←target - fulllevel ;
/* i is the displacement of target */
j ←fulllevel/2 ; /* j = 2^(length of insertion path −1) */
k ←1 ; /* k is the current position
                in the insertion path */
/* Reheapification Loop */
while (j ≠ 0)
        if(VALUE(k) > VALUE(nkey))
        then Exchange(nkey,key(k)) endif
        if (i ≥ j)
        then {k ←RSON(k); i ←i - j;} /* Go Right*/
        else {k ←LSON(k)}; /* Go Left */
        endif
        j ←j/2 ;
endwhile
key(k) ←nkey ;
Unlock(heap) ;
end_insert_t
```

Note that the insertion path is being computed on the fly. Let i be the displacement of target at the last level (i.e., i = lastelem - fulllevel), and p be the length of the insertion path. If we view i as a p bit binary number, then the bits of the binary representation of i (from the most significant to the least significant) tell us whether to go right (if 1) or left (if 0) when we go from the root downward. For example, the first element at the last level (given by fulllevel) has displacement 0 and its path is left,left,left$\cdots$. Fig 2 shows the twelve element heap of Fig 1 to which a thirteenth element is being added. It also shows the values of fulllevel, lastelem, and i just before the execution of the reheapification loop in insert_t. For a proof of correctness of insert_t, see [9].

Status code	*Meaning*
Present	A key exists at the node.
Pending	An insertion is currently in progress which will ultimately insert a key at the node
Wanted	A deleter is waiting for the key.
Absent	No key is present at the node.

Table 1: Meaning of various status codes.

4 Concurrent access algorithms

A simple locking strategy is embedded into delete and insert_t routines to achieve concurrency in access maintaining consistency in updates and avoiding deadlocks. Instead of locking the whole heap (as done in the serial access scheme), we lock only a small portion of the heap at a time. This portion is called **window**. It consists of 3 nodes for the delete algorithm and 1 node for the insert operation. In order to allow window locking, we associate a lock with every element. Each processor accesses the contents of a node only after locking it to ensure mutual exclusion. The two other data fields of the heap, fulllevel and lastelem, are modified only in the initialization phase of the insert_t and delete routines. Hence we associate the lock of node 1, the root, with these fields also; i.e., a processor can access these locations only when the root has been locked.

Although insert_t and delete both manipulate the heap from top to bottom, there is one problem in letting them work together. Recall that the delete operation deletes the key at the root and replaces it with the most recently inserted leaf key (and starts reheapification). If the last insert_t operation is still in progress, then this last leaf node does not have a key. If delete picks up the key of any other leaf node, then the resulting heap may become unbalanced. If the delete operation waits for the last insertion to finish, then we loose concurrency.

To solve this problem, we associate a field called status with every node in the heap. The status of a node can have four values, each associated with the semantics given in Table 1. When an insertion starts, the status of its target is set to pending. If a deleter starts working when an insertion is still in progress, it changes the status of the target of the last inserter to wanted,

[2]Throughout the paper we present algorithms in a C-like English pseudo-code.

[3]If the heap becomes unbalanced, then inserts and deletes can take up to O(N) operations rather than O(log N) operations.

and waits. After every step of reheapification on the insertion path, the inserter checks to see if the status of target has become wanted. If this is the case, then nkey is placed at the root and the inserter quits. Once the key is placed at the root, deleter starts working. The concurrent deletion and insertion algorithms are presented below.

```
Concurrent Delete(heap)
Lock(1);
/* Lock the root of the heap */
if (lastelem = 0)
then {Unlock(1); Return(NULL)} endif
least ←key(1) ;
i ←1 ;
j ←lastelem ;
lastelem ←lastelem - 1 ;
if (lastelem < fulllevel)
then fulllevel ←fulllevel/2 endif
if (j=1)
then{ key(1) ←MAXINT; status(1) ←ABSENT;
Unlock(1); Return(least)} endif
Lock(j) ;
if (status(j) = PRESENT)
  then {key(1) ←key(j); status(j) ←ABSENT;
        key(j) ←MAXINT;}
  else {status(1) ←ABSENT ; status(j) ←WANTED};
endif
Unlock(j) ;
while (status(i) = ABSENT) do {Wait()}
  endwhile /* i = 1 at this point */

Lock(LSON(i)) ; Lock(RSON(i)) ;
/* Reheapification Loop */
/* Let MIN(i) give index of the son of i which
has lower VALUE*/
/* Let MAX(i) give index of the son of i which
has higher VALUE*/
while (VALUE(i) > VALUE(MIN(i))) do
      Exchange(key(i),key(MIN(i))) ;
      Unlock(i) ; Unlock(MAX(i)) ;
      i ←MIN(i) ;
      Lock(LSON(i)) ; Lock(RSON(i)) ;
endwhile
Unlock(i) ; Unlock(LSON(i)) ; Unlock(RSON(i)) ;
Return(least) ;
end_Concurrent_Delete


Concurrent Insert(nkey,heap)
Lock(1) /* Lock root of the heap */
lastelem ←lastelem + 1 ;
target ←lastelem ;
if (lastelem ≥ fulllevel* 2)
then fulllevel ←lastelem endif
i ←target - fulllevel ;
/* i is the displacement of target */
j ←fulllevel/2 ; /* j = 2^(length of insertion path -1) */
k ←1 ; /* k is the current position
          in the insertion path */
status(target) ←PENDING ;

/* Reheapification Loop */
while (j ≠ 0)
      if (status(target) = WANTED)
        then break endif
      if (VALUE(k) > VALUE(nkey))
        then Exchange(nkey,k); endif
      if (i ≥ j)
        then /* Go Right */
        {Lock(RSON(k));Unlock(k);
        k ←RSON(k); i ←i - j}
        else /* Go Left */
        {Lock(LSON(k)); Unlock(k);
        k ←LSON(k)};
      endif
      j ←j/2 ;
endwhile
if (status(target) = WANTED)
      then /*Some deleter is waiting at the root
              to pick the key at target */
      {key(1) ←nkey; status(target) ←ABSENT;
              status(1) ←PRESENT}
      else
      {key(target) ←nkey;
              status(target) ←PRESENT};
endif
Unlock(k) ;
end_Concurrent Insert
```

Whenever an inserter or a deleter moves down 1 level by incrementing k or i, it first locks the next node and then releases the current lock. This ensures that concurrent deletes or inserts proceeding in the same path progress in a strict queue order without any interference. Since the locking sequence is in the strict increasing order of node indices, there are no deadlocks. See [9] for a proof of correctness.

5 Experimental Evaluation

We have implemented the concurrent access heap algorithms and the serial access heap algorithms on BBN Butterfly to test their performance. Using each scheme, P processors performed a total of 1000 delete or insert operations (each processor performed 1000/P operations. P was varied between 1 and 30). Think time T_{think} was set to roughly 5 times the heap access time of the serial operation. The speedup was computed as follows:

$$\frac{\textit{Time taken by P processors for } \frac{1000}{P} \textit{operations}}{\textit{Time taken by 1 processor for 1000 operations}}.$$

Relative performance of the concurrent heap was studied for the following cases.

Case I: Deletes

In this case each processor performed one delete operation in each access-think cycle. A total of 1000 delete operations were performed on a heap that initially had 2048 elements. Thus, the depth of the heap remained 10 for all the deletions. For the serial access scheme, the speedup was fairly linear up to 5 processors, but saturated after that. For the concurrent heap, the speedup did not saturate until over 11 processors. These results are shown in Fig. 3.

Case II: Inserts

In this case, each processor performed one insert operation in each access-think cycle. A total of 1000 insert operations were performed on a heap that initially had 1024 elements. Thus, the depth of the heap remained 10 for all the insertions. If the values of the inserted keys are randomly distributed, then the number of iterations of the heapification loop executed by insert_b are very small[5]. On the other hand, insert_t executes strictly $\log(N-1)$ iterations (N is the size of the heap). Hence for inserting keys with random key values, our concurrent heap scheme does not perform better than the serial access heap insert. (The speedup figures are roughly the same for both cases.)

In parallel branch-and-bound algorithms[8,10] the inserted keys tend to have small values. For such keys, both insert_b and insert_t would execute roughly the same number of iterations. To test the performance in this case, we generated keys whose values were in the decreasing order. In this case, just as in Case II, the speedup of the concurrent heap scheme saturated

much later than the serial access heap scheme. Fig. 4 shows the speedup curves.

Case III: Two Inserts and One Delete

In this case, each processor performs one delete and two insert operations in each access-think cycle. The heap initially has 1024 elements. This case simulates the behavior of a typical parallel branch-and-bound algorithm in which each processor picks a least cost node from the heap, generates 2 successors and puts them back on the heap. In this case, the concurrent heap scheme is able to provide a speedup of 13.5, where as the serial access scheme saturates at 5.

6 Related Research

Biswas and Browne[1] present a scheme, called CHEAP, that allows insertions and deletions to proceed in parallel. In their scheme, an insert or delete operation is decomposed into a sequence of update steps at different levels of a heap. An auxiliary task queue stores the steps of insertions and deletions currently in progress. By appropriately scheduling these update steps, a set of service processes concurrently perform insertions and deletions without causing deadlocks. If enough service processors are available, then this scheme can perform insertions and deletions in constant time. This approach is not able to perform better than the serial access scheme except for very large heaps due to the overheads associated with scheduling window updates through the server queue.

Unlike the scheme in [1], our scheme does not require special server processors to update the heap. Also the number of locks needed for each operation are much smaller. Unlike their scheme, our scheme also retains the strict priority ordering of the serial access heap algorithms; i.e., a delete operation returns the best element of all elements that have been inserted or are being inserted at the time the delete operation is started. The scheme presented in this paper was motivated by the work of Biswas and Browne. Initially, we wanted to incorporate CHEAP in our parallel branch-and-bound algorithms to improve their performance. But experiments conducted by Biswas [4] showed that CHEAP was not able perform better than the serial access scheme even for heaps with 1,000 elements.

Ellis and Gaffar[5] have developed a scheme that also does not require the use of separate special service processors. In this scheme, inserts and deletes proceed in opposite directions, but avoid deadlock using a "sliding-lock" scheme. Performance results of this scheme are not yet available.

A number of concurrent-access schemes have been developed for manipulating dictionaries that are represented as balanced trees[4,7], B-trees [3], and the balanced cube[2]. Most of these concurrent schemes allow $O(\log N)$ operations (delete the smallest key, delete a key, insert a key, search for key, etc.) to be done simultaneously. A major exception is the balanced cube which permits $O(N)$ search, insert and delete operations to done concurrently. However, even the balanced cube permits only $O(\log N)$ operations "delete-the-smallest-key" operations at a time. In a priority queue, the only operations of interest are "delete-the-smallest-key" and "insert-a-key". For these operations, on a sequential processor, the heap is clearly a more efficient data structure than B-tree, balanced trees and the balanced cube. Since our concurrent-access heap scheme has the same degree of concurrency as others and has smaller overhead, it is better than other concurrent schemes for manipulating a strict priority queue.

[4]Personal communication

[5]Private communication with Carla Ellis

7 Conclusions

We have presented a new concurrent heap access scheme that has small overhead, and is able to perform better than the sequential heap even for small heaps. The insert and delete operations of this scheme keep the heap balanced; hence each operation still takes $O(\log N)$ steps, where N is the size of the heap. The scheme also retains the strict priority ordering of the serial access heap algorithms; i.e., a delete operation returns the best key of all keys that have been inserted or are being inserted at the time delete is started. In this scheme, $O(\log N)$ processes can manipulate the heap simultaneously. A detailed analysis of the expected performance is reported in [9], where we also discuss a number of possible improvements that can be made to reduce the overhead of the scheme. We have incorporated the concurrent heap scheme in a parallel branch-and-bound algorithm for solving the traveling salesman problem, and have obtained much better speedups than with the serial access schemes[6].

Note that even in the concurrent-access heap scheme, atmost $O(\log N)$ processors can manipulate the heap concurrently. To allow greater concurrency, it seems necessary to relax the strictness of the priority queue. In [6], we present several "distributed" formulations of priority queue that permit $O(N)$ concurrency, and test their effectiveness in parallel branch-and-bound.

Acknowledgements: We would like to thank Jit Biswas for many useful discussions concerning CHEAP.

References

[1] Jit Biswas and James C. Browne. Simultaneous update of priority structures. In *Proceedings of International conference on Parallel Processing*, page XXXXX, 1987.

[2] William Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publ, Boston, Massachusetts, 1987.

[3] Carla S. Ellis. Concurrent search and insertion in 2–3 trees. *Acta Informatica*, 14:63–86, 1980.

[4] Carla S. Ellis. Concurrent search and insertion in avl trees. *IEEE Transactions on Computers*, C-29 No 9:811–817, Sept 1980.

[5] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville, Maryland, 1978.

[6] V. Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel heuristic search of state-space graphs: a summary of results. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, August 1988. Also AI Lab Tech. Report 88-70, University of Texas at Austin, March 88.

[7] U. Manber and R.E. Ladner. Concurrency control in a dynamic search structure. *ACM Trans. on Database Systems*, 9, 3:439–455, 1984.

[8] Joseph Mohan. Experience with two parallel programs solving the traveling salesman problem. In *Proceedings of International conference on Parallel Processing*, pages 191–193, 1983.

[9] V. Nageshwara Rao and V. Kumar. *Concurrent Access of Priority Queues.* Technical Report TR88-06, Computer Science Dept., Univ. of Texas at Austin, February 1988.

[10] V. Nageshwara Rao, V. Kumar, and K. Ramesh. *Parallel Heuristic Search on a Shared Memory Multiprocessor.* Technical Report AI TR87-45, Univ. of Texas at Austin, January 1987.

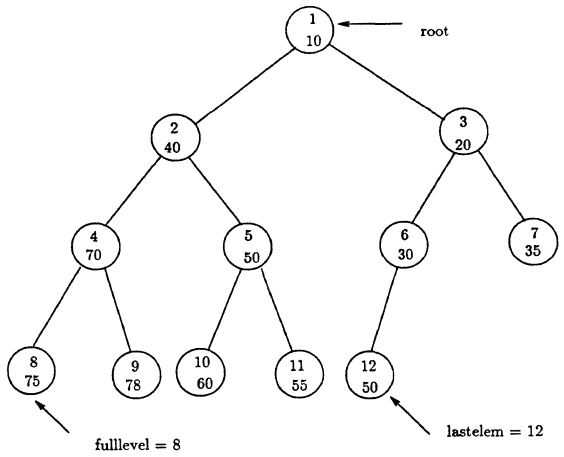


Figure 1: A heap of twelve elements. Upper half of the circle contains the node number, and the lower half contains the value of the key.

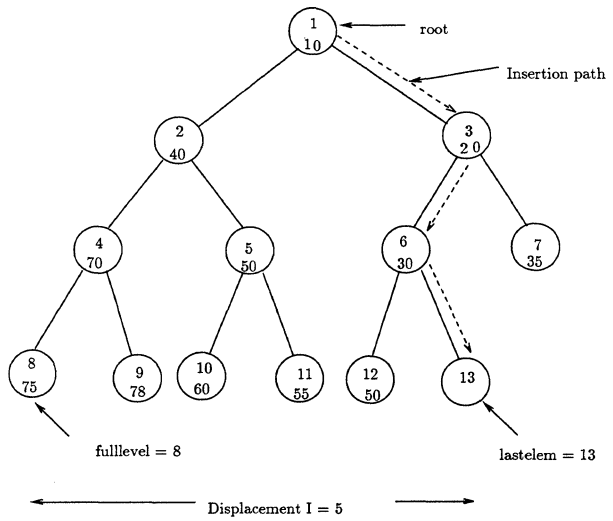


Figure 2: An example of how the insertion path is computed in insert_t. A new key is inserted into the heap at node 13. I = 5 = (101) in the binary representation; length of the insertion path = 3.

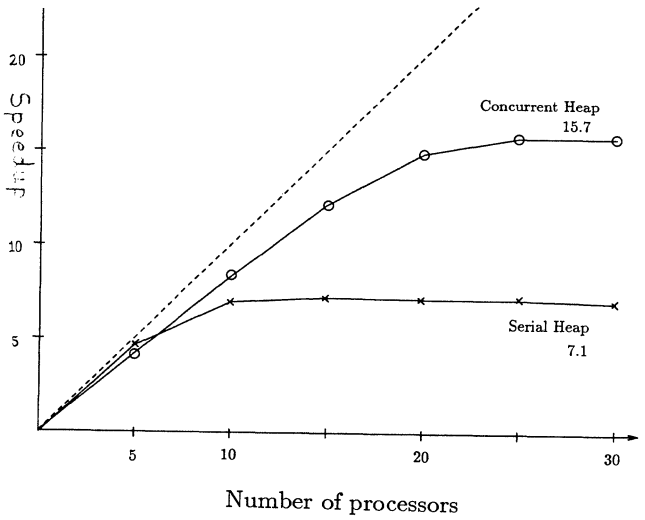


Figure 3: Plot of speedups obtained in execution of access-think cycles for delete operation.

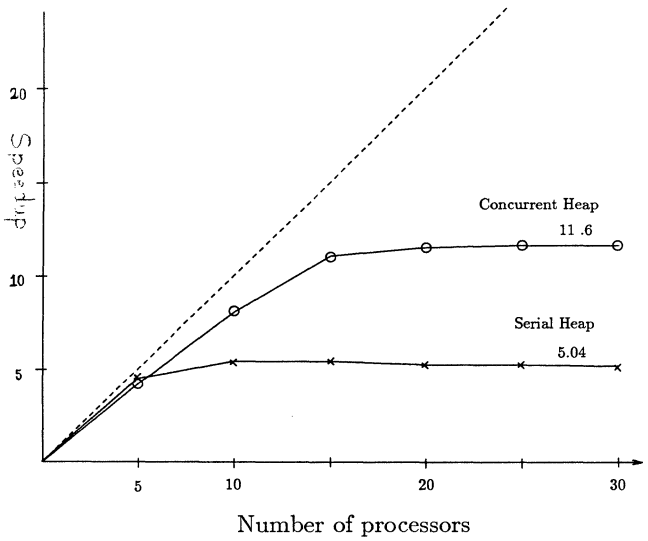


Figure 4: Plot of speedups obtained in execution of access-think cycles for insert operation. The numbers inserted are in a decreasing order.

CONVOLUTION ON SIMD MESH CONNECTED MULTICOMPUTERS [+]

Sanjay Ranka and Sartaj Sahni

University of Minnesota

Abstract

Convolution is an important primitive in computer vision and image processing. In this paper, we present efficient and optimal algorithms for convolution on a mesh connected computer. Our algorithms do not assume any broadcast feature for data values as assumed by previously proposed algorithms.

Keywords and Phrases

Convolution, mesh connected multicomputer

1. INTRODUCTION

The inputs to the image template matching problem are an $N \times N$ image matrix $I[0..N-1, 0..N-1]$ and an $M \times M$ template $T[0..M-1, 0..M-1]$. The output is an $N \times N$ matrix C2D where

$$C2D[i, j] = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} I[(i+u) \bmod N, (j+v) \bmod N] * T[u,v]$$

$$0 \leq i, j < N$$

$C2D$ is called the two dimensional convolution of I and T. Template matching, i.e., computing $C2D$, is a fundamental operation in computer vision and image processing. It is often used for edge and object detection; filtering; and image registration [ROSE82, BALL85]. Because of the fundamental nature of this problem and because of its high complexity ($O(M^2N^2)$ on a single processor computer), much attention has been devoted to the development of efficient fine grain multicomputer parallel algorithms. For example, Chang, Ibarra, Pong and Sohn [CHAN87] have studied this problem on an SIMD pyramid computer; Fang, Li and Ni [FANG85], Fang, Li and Ni [FANG86], Prassana Kumar and Krishnan [PRAS86], and Ranka and Sahni [RANK87a] and [RANK87b] have considered it on a hypercube multicomputer; Fang, Li and Ni [FANG86] have considered perfect shuffle multicomputers; Kung and Song [KUNG81] have considered systolic arrays; and Lee and Aggarwal [LEE87], and Maresca and Li [MARE86] have considered mesh connected computers.

In this paper, a parallel algorithm for 2-D convolution is presented for an SIMD mesh connected multicomputer. Our algorithm differs from those of [LEE87] and [MARE86] in that our algorithm does not use any broadcast of data values. Further, the amount of result value movement in our algorithms is an order of magnitude less than in the algorithms of [LEE87] and [MARE86]. Thus when the size of the image and template values is small (e.g, binary images and templates) as compared to the convolution values, our algorithms will be more efficient.

Section 2 describes our computer model. In addition, notation and some fundamental data movement operations are developed in this section. In Section 3, we develop fine grained algorithms for one dimensional convolution. These form the basic component of our two dimensional convolution algorithms which are developed in Section 4.

2. PRELIMINARIES

2.1. Mesh Connected Multicomputer

[+] This research was supported in part by the National Science Foundation under grants DCR84-20935 and MIP 86-17374

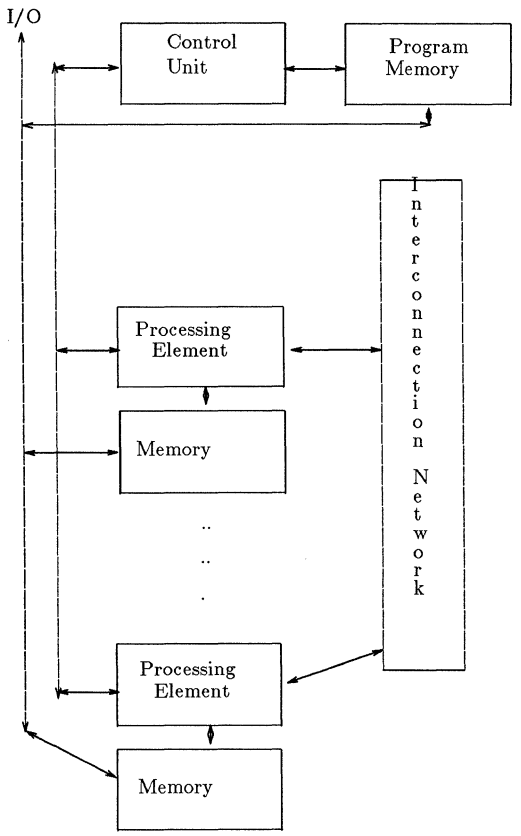


Figure 1 : An SIMD multicomputer

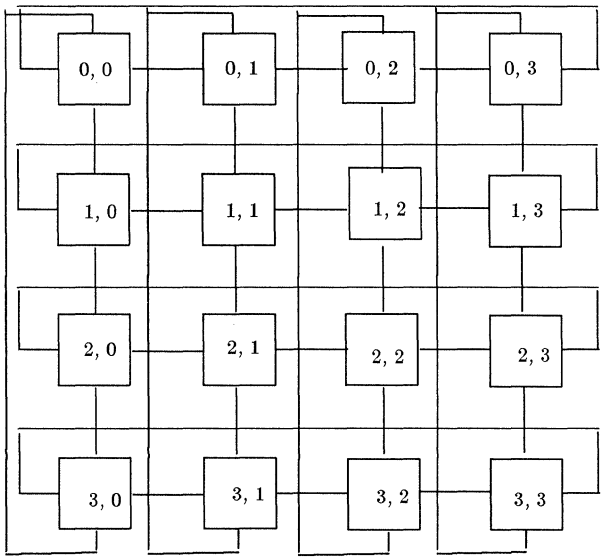


Figure 2 : A 4 × 4 mesh connected computer

A block diagram of an SIMD mesh connected multicomputer is given in Figures 1 and 2. The important features of such a multicomputer and the programming notation we use are:

1. There are P × P processing elements connected together via a mesh interconnection network (to be described later). Each PE has a unique index (0..P-1,0..P-1). Sometimes we will use a one dimensional indexing of the mesh. This is obtained using the standard row major mapping in which (i, j) is mapped $iP + j$. The local memory of each PE can hold data only (i.e no executable instructions). Hence PEs need be able to perform only the basic arithmetic operations (i.e., no instruction fetch or decode is needed). Throughout this paper, we shall use brackets([ ]) to index an array and parentheses('()') to index the PEs. So, A[i, j] refers to i,j'th element of the matrix A while A(i, j) refers to the A register of PE(i, j).

2. There is a separate program memory and control unit. The control unit performs instruction sequencing, fetching, and decoding. In addition, instructions and masks are broadcast by the control unit to the PEs for execution. An *instruction mask* is a boolean function used to select certain PEs to execute an instruction. For example, in the instruction
$$A(i, j) := A(i, j) + 1, \quad (i \ mod \ 4 = 0)$$
($i \ mod \ 4 = 0$) is a mask that selects only those PEs whose row index satisfies this property. I.e, all PEs with indices which are multiples of 4 increment their A register by 1.

3. The topology of a 16 node mesh connected computer is shown in Figure 2. A P × P mesh contains P^2 PEs. PE(i, j) is connected to PE((i–1) mod P, j), PE((i+1) mod P, j), PE(i, (j–1) mod P) and PE(i, (j+1) mod P).

4. Interprocessor assignments are denoted using the symbol ←, while intraprocessor assignments are denoted using the symbol :=. Thus the assignment statement:
$$B((i+1) \ mod \ N, j) \leftarrow B(i, j)$$
implies that each processor transmits its B register value to the B register of the processor on its right.

5. In a *unit route*, data may be transmitted from one processor to another only if the two are directly connected. We assume that the links in the interconnection network are unidirectional. Hence at any given time, data can be transferred either from PE((i+1) mod N, j) to PE(i, j) or vice versa.

6. Since the asymptotic complexity of all our algorithms is determined by the number of unit routes, our complexity analysis will count only these.

2.2. Basic Data Manipulation Operations

In this section, we develop algorithms for some basic operations on a one dimensional array. Such an array with P PEs has the topology shown in Figure 3. The PEs are indexed 0 through P-1 left to right.

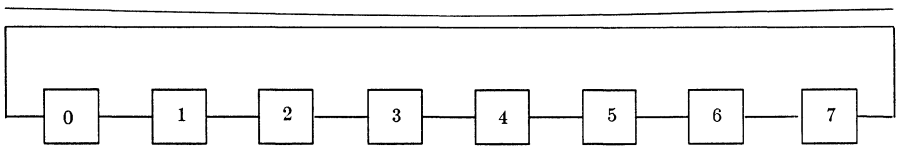


Figure 3: One dimensional array

2.2.1. Shift

SHIFT(A,i) shifts the A register data circularly counter clockwise by i. It can be performed in | i | unit routes unless P = 2. In this case, a shift of ±1 requires 2 unit routes because of the assumption of unidirectional links. For convenience, we shall henceforth assume P > 2.

2.2.2. Data Accumulation

For this operation, PE j has an array $A[0..M-1]$ of size M. The notation $A[i](j)$ refers to $A[i]$ in PE j. In addition, each PE has a value in its I register. After the data accumulation, the M elements of A in PE j are such that:
$$A[i](j) = I((j + i) \ mod \ P), \ 0 \leq i < M, \ 0 \leq j < P$$

This operation can be performed in (M-1) unit routes using the algorithm given in Figure 4.

```
procedure ACCUM(A, I, M)
{each PE accumulates in A, the I values of the next M PEs
 including itself}
begin
    A[0] := I;
    for i := 1 to M-1 do
    begin
      SHIFT(I, -1);
      A[i] := I;
    end
end {ACCUM}
```

Figure 4: Data accumulation

2.2.3. Adjacent Sum

For each PE, p, $0 \leq p < P$, the sum
$$T(p) = \sum_{i=0}^{M-1} A[i]((p + i) \ mod \ P)) \qquad (1)$$
is to be computed. M ≤ P is a parameter to the operation. This can be performed in 2(M -1) unit routes using the algorithm of Figure 5. The strategy here is that each processor initiates a T value that circulates through the M processors containing its A terms (cf. eq(1)). Once the M terms have been accumulated, the T values need to move back to the originating PEs. This requires a clockwise shift of M-1.

```
procedure AdjacentSum(A, M)
begin
    T:=A[0];
    for i := 1 to M-1 do
    begin
      SHIFT(T, 1);
      T := T + A[i];
    end
    SHIFT(T, -M +1 );
end {of AdjacentSum}
```

Figure 5: Adjacent Sum

3. ONE DIMENSIONAL CONVOLUTION

The inputs to the one dimensional convolution problem are vectors $I[0..N-1]$ and $T[0..M-1]$. The output is the vector C1D where:

$$C1D[i] = \sum_{v=0}^{M-1} I[(i+v) \bmod N] * T[v] \quad , \ 0 \leq v < N$$

We use the computation of C1D as a basic step in our algorithms to compute C2D. In this section, we develop algorithms for C1D on a one dimensional processor array. We consider the two cases:

(i) Each PE has $O(M)$ memory

(ii) Each PE has $O(1)$ memory

Our algorithms assume that the controller cannot broadcast data values to the processors. There are P=N processors and the vector I is mapped onto the one dimensional processor array such that processor p contains $I[p]$. Further, there are N/M copies of T in the processor array with one copy in each block of M processors (processors $iM+j, 0 \leq j < M$ form a block for each i, $0 \leq i < N/M$). Within a block, the mapping of T is the same as that of I. The case when N=16 and M=4 is given in Figure 6. The first row of this figure gives the processor index.

3.2. O(1) Memory

When only O(1) memory per PE is available, we begin by first pairing I values in the processors. The pair in processor p is (A(p), B(p)) = $(I[(q+2k) \bmod N], I[(q+2k+1) \bmod N])$ where $q = \lfloor P/M \rfloor M$ and $k = p \bmod M$. Figure 8 gives the initial AB pairs in each PE for the case $N=16, M=4$. This pairing is easily obtained in M unit routes. Once the AB pairing has been obtained, the C1D may be computed by rotating the AB and T values clockwise. Throughout the algorithm, the product of A(p) and T(p) will give one of the terms needed to compute $C1D(p)$, $0 \leq p < P$. B(p) will be the next I value needed. Initially, this is true for all processors except those with $p \bmod M = M-1$. This situation is remedied by replacing B with I in these processors to get the first column labeled AB'. Following a rotation of AB, we get the second column labeled AB. Now, the B value in processors with $p \bmod M = M-2$ needs to be changed to $I(p)$. With this insight, one arrives at the algorithm of Figure 9. Its correctness is easily established. The number of unit routes (including those for pairing) is at most $3M$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3

Figure 6: Initial configuration for 1-D convolution

3.1. O(M) Memory

When each processor has $O(M)$ memory, the most effective way to compute C1D is to first perform a data accumulation on I (Figure 4). Following this, each processor has all the I values needed to compute the corresponding entry of C1D. Next, the T values are circulated through the M processors. During this circulation, the T values are multiplied by I values and the C1D values computed. It is worth noticing that, unlike the algorithms of [LEE87] and [MARE86], no results are moved in our algorithms. Thus when the size of the results is much larger than the size of the template or image values, our algorithm will perform better. The algorithm is presented in Figure 7. The total number of unit routes is 2M -1. Note that while the last iteration of SHIFT(T, -1) is unnecessary for the computation of C1D, it restores the original T values. This is required by our C2D algorithm.

```
procedure C1D_M(M)
{O(M) memory algorithm for one dimensional convolution}
begin
        ACCUM(A, I, M);
        C1D := 0;
        for j := 0 to M-1 do
        begin
                C1D := C1D + A[(j + p) mod M] * T;
                SHIFT(T, -1);
        end
end; {of C1D_M}
```

Figure 7: $O(M)$ memory computation of C1D

4. TWO DIMENSIONAL CONVOLUTION

We assume that C2D is to be computed on an N × N mesh connected SIMD multicomputer. Further, we assume that $I[i, j]$ is initially in the I register of $PE(i, j)$, the result C2D is to be computed such that $C2D[i, j]$ is in the C2D register of $PE(i, j)$, and that N is a multiple of M. Thus the N × N array may be viewed as composed of N^2/M^2 arrays each of size M × M (Figure 10). We also assume that processor PE(i, j) contains T$[i \bmod M, j \bmod M]$ in its T register.

4.1. O(M) Memory

In this case, PE(i,j), $0 \leq i < N$, $0 \leq j < N$ first computes M one dimensional convolutions $S[q]$, $0 \leq q < M$ as defined below

$$S[q] = \sum_{r=0}^{M-1} I[(i, (j+r) \bmod N] * T[q, r]$$

Next, C2D is obtained by performing an adjacent sum operation along the columns of the N × N PE array. A high level description of the algorithm is given in Figure 11. The total number of unit routes is $M^2 + O(M)$. Notice that result movement is restricted to adjacent sum, which takes O(M) time. The algorithms of [LEE87] and [MARE87] require $O(M^2)$ result movement.

4.2. O(1) Memory

We develop two algorithms for this case. The first is conceptually simpler but requires $4M^2 + O(M)$ unit routes. The second requires only $2M^2 + O(M)$ unit routes. For the first algorithm, we rewrite the definition of C2D as

$$C2D[i, j] = \sum_{r=0}^{M-1} CXD[i, r, j]$$

i	I	AB	T	AB'	AB	T	AB'	AB	T	AB'	AB	T	AB'
0	I_0	I_0I_1	T_0	I_0I_1	I_1I_2	T_1	I_1I_2	I_2I_3	T_2	I_2I_3	I_3I_4	T_3	I_3I_0
1	I_1	I_2I_3	T_1	I_2I_3	I_3I_4	T_2	I_3I_4	I_4I_5	T_3	I_4I_1	I_1I_2	T_0	I_1I_2
2	I_2	I_4I_5	T_2	I_4I_5	I_5I_6	T_3	I_5I_2	I_2I_3	T_0	I_2I_3	I_3I_4	T_1	I_3I_4
3	I_3	I_6I_7	T_3	I_6I_3	I_3I_4	T_0	I_3I_4	I_4I_5	T_1	I_4I_5	I_5I_6	T_2	I_5I_6
4	I_4	I_4I_5	T_0	I_4I_5	I_5I_6	T_1	I_5I_6	I_6I_7	T_2	I_6I_7	I_7I_8	T_3	I_7I_4
5	I_5	I_6I_7	T_1	I_6I_7	I_7I_8	T_2	I_7I_8	I_8I_9	T_3	I_8I_5	I_5I_6	T_0	I_5I_6
6	I_6	I_8I_9	T_2	I_8I_9	I_9I_{10}	T_3	I_9I_6	I_6I_7	T_0	I_6I_7	I_7I_8	T_1	I_7I_8
7	I_7	$I_{10}I_{11}$	T_3	$I_{10}I_7$	I_7I_8	T_0	I_7I_8	I_8I_9	T_1	I_8I_9	I_9I_{10}	T_2	I_9I_{10}
8	I_8	I_8I_9	T_0	I_8I_9	I_9I_{10}	T_1	I_9I_{10}	$I_{10}I_{11}$	T_2	$I_{10}I_{11}$	$I_{11}I_{12}$	T_3	$I_{11}I_8$
9	I_9	$I_{10}I_{11}$	T_1	$I_{10}I_{11}$	$I_{11}I_{12}$	T_2	$I_{11}I_{12}$	$I_{12}I_{13}$	T_3	$I_{12}I_9$	I_9I_{10}	T_0	I_9I_{10}
10	I_{10}	$I_{12}I_{13}$	T_2	$I_{12}I_{13}$	$I_{13}I_{14}$	T_3	$I_{13}I_{10}$	$I_{10}I_{11}$	T_0	$I_{10}I_{11}$	$I_{11}I_{12}$	T_1	$I_{11}I_{12}$
11	I_{11}	$I_{14}I_{15}$	T_3	$I_{14}I_{11}$	$I_{11}I_{12}$	T_0	$I_{11}I_{12}$	$I_{12}I_{13}$	T_1	$I_{12}I_{13}$	$I_{13}I_{14}$	T_2	$I_{13}I_{14}$
12	I_{12}	$I_{12}I_{13}$	T_0	$I_{12}I_{13}$	$I_{13}I_{14}$	T_1	$I_{13}I_{14}$	$I_{14}I_{15}$	T_2	$I_{14}I_{15}$	$I_{15}I_0$	T_3	$I_{15}I_{12}$
13	I_{13}	$I_{14}I_{15}$	T_1	$I_{14}I_{15}$	$I_{15}I_0$	T_2	$I_{15}I_0$	I_0I_1	T_3	I_0I_{13}	$I_{13}I_{14}$	T_0	$I_{13}I_{14}$
14	I_{14}	I_0I_1	T_2	I_0I_1	I_1I_2	T_3	I_1I_{14}	$I_{14}I_{15}$	T_0	$I_{14}I_{15}$	$I_{15}I_0$	T_1	$I_{15}I_0$
15	I_{15}	I_2I_3	T_3	I_2I_{15}	$I_{15}I_0$	T_0	$I_{15}I_0$	I_0I_1	T_1	I_0I_1	I_1I_2	T_2	I_1I_2

Figure 8: Execution Trace N = 16, M = 4

```
procedure C1D_1 (M)
{O(1) memory C1D algorithm}
begin
        PAIRING(M); {obtain initial AB pairs}
        C1D := 0;
        for j := 0 to M-1 do
        begin
                B(p) := I(p); {p mod M = M-1-j)
                C1D := C1D + A * T;
                SHIFT(A, -1);
                C := B; B := A; A:= C; {Exchange A and B}
                SHIFT(T, -1);
        end;
end; { of C1D_1}
```

Figure 9: $O(1)$ memory computation of C1D

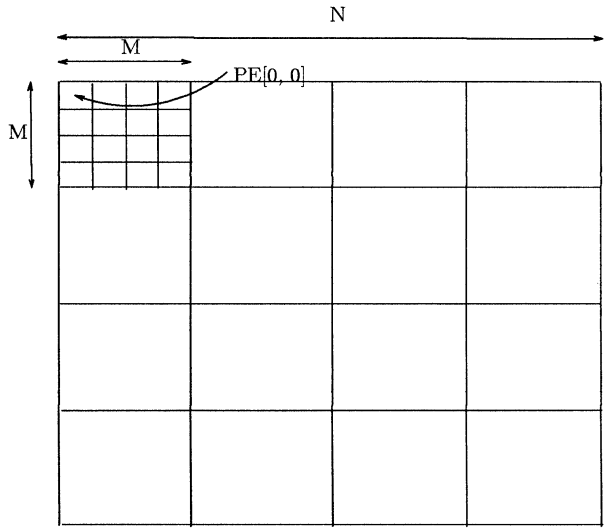


Figure 10: A N × N array viewed as N^2/M^2 M × M arrays

```
procedure C2D_M(N, M)
        {assumes O(M) memory per PE}
Step1:  Regard the N × N mesh as N one dimensional array
        processors. Each row forms one such array. Perform a
        data acumulation on I. Now each PE contains the M I
        values it needs to compute its S(q)'s.
Step2:  Compute the S[q]'s. Each S[q] is a one dimensional con-
        volution. However, the data accumulation step of the
        algorithm of Figure 7 may be omitted as the I values
        have already been accumulated in Step 1. To go from
        one S to another, the T values need to be shifted along
        the columns of each M × M block. This can be done us-
        ing the vertical inter PE connections.
Step3:  Compute C2D(i, j) = sum_{r=0}^{M-1} S[r]((i + r) mod N, j) This is
        done using the adjacent sum algorithm of Section 2 on
        the columns of the N × N PE array
end
```

Figure 11: High level description of two dimensional convolution with each PE having O(M) Memory

```
procedure C2D(N, M)
        {O(1) memory C2D algorithm}
Step1:  Repeat Steps 2, 3 and 4 for q := 0 to M-1
Step2:  PE(i + r, j) computes C1D(i + r, j) = CXD[i, r, j]
        where i mod M = q and 0 ≤ r < M. This is done using
        procedure C1D_1 of Figure 9.
Step3:  PE(i, j) for i mod M = q computes
        C2D(i, j) = sum_{r=0}^{M-1} CXD[(i + r) mod N, j] by repeatedly
        shifting the C1D values up the columns of the processor
        array.
Step4:  T(i, j) ←T((i + 1) mod N, j)
end; {of C2D}
```

Figure 12: O(1) memory computation of C2D

$$CXD[i, r, j] = \sum_{a=0}^{M-1} I[(i + r) \bmod N, (j + a) \bmod N] * T[r, a]$$

A high level description is provided in Figure 12. The number of unit routes is $4M^2 + O(M)$. In the q'th iteration of Steps 1 and 2 of this algorithm C2D(i, j) is computed for all PEs with $i \bmod M = q$. This is done by first having PE$(i + r, j)$. Compute $C1D(i + r, j) = CXD[i, r, j]$ for $i \bmod M = q$ and $0 \leq r < M$. Next $C2D(i, j)$ is computed by summing $CXD[i, r, j]$ for $i \bmod M = q$ and $0 \leq r < M$. This summation is done only in PEs (i, j) with $i \bmod M = q$ and requires shifting the CXD$[i, r, j]$ values up along the columns. Each iteration of Steps 2 and 3 takes $4M + O(1)$ unit routes. Thus the unit route count for the entire algorithm is $4M^2 + O(M)$. Note also that this algorithm requires $M^2 - M$ movement of the C1D values.

The strategy for the second algorithm is similar to that used in computing C1D when only O(1) memory is available. We may rewrite the definition of C2D as

$$C2D[i, j] = \sum_{r=0}^{M-1} X[i, j, r] * Y[r]$$

where $X[i, j, r]$ is the $1 \times M$ vector $I[(i+r) \bmod N, j .. (j+M-1) \bmod N]$ and $Y[r]$ is the $1 \times M$ vector $T[r, 0 .. M-1]$. Thus C2D is viewed as the one dimensional convolution of X and Y where each term X and Y is a vector. The algorithm is presented in Figure 13. Figure 14 shows the initial AB, I1 I2, A1 A2, and B1 B2 pairs created by the first four PAIRING operation when N=8, and M=4.The total number of unit routes is $2M^2 + O(M)$. Unlike the first algorithm, there is no result movement in this algorithm.

```
procedure C2D_1(N, M)
{assumes O(1) memory per PE}
begin
        PAIRING(M) along rows of I; {obtain AB pairs}
        PAIRING(M) along columns of I; {obtain I1 I2 pairs}
        PAIRING(M) along columns of A; {obtain A1 A2 pairs}
        PAIRING(M) along columns of B; {obtain B1 B2 pairs}
        C2D := 0;
        for a := 0 to M-1 do
        begin
                A2(i, j) := A(i, j);  ( i mod M = M-1-a)
                B2(i, j) := B(i, j);  ( i mod M = M-1-a)
                I2(i, j) := I(i, j);  ( i mod M = M-1-a)
                AA := A1 ; BB := B1;
                for b := 0 to M-1 do
                begin
                        BB(i, j) := I1(i, j);  ( j mod M = M-1-b)
                        C2D(i, j) := C2D(i, j) + AA(i, j) * T(i, j);
                        SHIFT(AA, -1) along rows;
                        C := BB; BB := AA; AA := C;
                        SHIFT(T, -1) along rows;
                end
                SHIFT(A1, -1) along columns;
                SHIFT(B1, -1) along columns;
                SHIFT(I1, -1) along columns;
                C := A1; A1 := A2; A2 := C;
                C := B1; B1 := B2; B2 := C;
                C := I1; I1 := I2; I2 := C;
                SHIFT(T, -1) along columns;
        end;
end{of C2D_1}
```

Figure 13: Two dimensional convolution with each PE having O(1) Memory

5. CONCLUSION

In this paper, we have developed optimal algorithms for 1-D convolution and image template matching (2-D Convolution) on a mesh connected SIMD multicomputer. None of our algorithms require a data broadcast. Further, our algorithms require less or no movement of results. Hence, these algorithms will be more efficient when the size of the image and template values is small as compared to the size of the convolution values.

6. REFERENCES

[BALL85] D. H. Ballard and C. M. Brown, "*Computer Vision*", **1985**, Prentice Hall, New Jersey.

[CHAN87] J. H. Chang, O. Ibarra, T. C. Pong, and S. Sohn, "Convolution on a Pyramid Computer", *International Conference on Parallel Processing*, **1987**, pp 780-782.

[DEKE86] E. Dekel, D. Nassimi and S. Sahni, " Parallel matrix and graph algorithms", *SIAM Journal on computing*, **1981**, pp. 657-675.

[FANG85] Z. Fang, X. Li and L. M. Ni, "Parallel Algorithms for Image Template Matching on Hypercube SIMD Computers", *IEEE CAPAMI workshop*, **1985**, pp 33-40.

[FANG86] Z. Fang, X. Li and L. M. Ni, "Parallel Algorithms for 2-D convolution", *International Conference on Parallel Processing*, **1986**, pp 262-269.

[HORO85] E. Horowitz and S. Sahni, "*Fundamentals of Data Structures in Pascal*", Computer Science Press, **1985**.

[KUNG82] H. T. Kung and S. W. Song, "A Systolic 2-D Convolution Chip", *Multicomputers and Image Processing: Algorithms and Programs, editors: Preston and Uhr (Academic Press, New York)*, **1982**, pp 373-384.

[LEE87] S. Y. Lee and J. K. Aggarwal, "Parallel 2-D convolution on a mesh connected array processor", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **July 1987**, pp 590-594.

[MARE86] M. Maresca and H. Li, "Morphological Operations on Mesh-connected Architecture : A generalized convolution Algorithm", *Proceedings of 1986 IEEE Computer Society Workshop on Computer Vision and Pattern Recognition* ,**1986**, pp 299-304.

[PRAS87] V. K. Prasanna Kumar and V. Krishnan, "Efficient Image Template Matching on SIMD Hypercube Machines", *International Conference on Parallel Processing*, **1987**, pp 765-771.

[RANK87a] S. Ranka and S. Sahni, "Image Template Matching on an SIMD hypercube multicomputers", *University of Minnesota Tech. Report*, **1987**.

[RANK87b] S. Ranka and S. Sahni, "Image Template Matching on MIMD hypercube multicomputers", *University of Minnesota Tech. Report*, **1987**.

[ROSE82] A. Rosenfeld and A. C. Kak, "*Digital Picture Processing*", Academic Press, **1982**

AB I1 I2 A1 A2 B1 B2	$I_{0,0}I_{0,1}$ $I_{0,0}I_{1,0}$ $I_{0,0}I_{1,0}$ $I_{0,1}I_{1,1}$	$I_{0,2}I_{0,3}$ $I_{0,1}I_{1,1}$ $I_{0,2}I_{1,2}$ $I_{0,3}I_{1,3}$	$I_{0,4}I_{0,5}$ $I_{0,2}I_{1,2}$ $I_{0,4}I_{1,4}$ $I_{0,5}I_{1,5}$	$I_{0,6}I_{0,7}$ $I_{0,3}I_{1,3}$ $I_{0,6}I_{1,6}$ $I_{0,7}I_{1,7}$	$I_{0,4}I_{0,5}$ $I_{0,4}I_{1,4}$ $I_{0,4}I_{1,4}$ $I_{0,5}I_{1,5}$	$I_{0,6}I_{0,7}$ $I_{0,5}I_{1,5}$ $I_{0,6}I_{1,6}$ $I_{0,7}I_{1,7}$	$I_{0,0}I_{0,1}$ $I_{0,6}I_{1,6}$ $I_{0,0}I_{1,0}$ $I_{0,1}I_{1,1}$	$I_{0,2}I_{0,3}$ $I_{0,7}I_{1,7}$ $I_{0,2}I_{1,2}$ $I_{0,3}I_{1,3}$
AB I1 I2 A1 A2 B1 B2	$I_{1,0}I_{1,1}$ $I_{2,0}I_{3,0}$ $I_{2,0}I_{3,0}$ $I_{2,1}I_{3,1}$	$I_{1,2}I_{1,3}$ $I_{2,1}I_{3,1}$ $I_{2,2}I_{3,2}$ $I_{2,3}I_{3,3}$	$I_{1,4}I_{1,5}$ $I_{2,2}I_{3,2}$ $I_{2,4}I_{3,4}$ $I_{2,5}I_{3,5}$	$I_{1,6}I_{1,7}$ $I_{2,3}I_{3,3}$ $I_{2,6}I_{3,6}$ $I_{2,7}I_{3,7}$	$I_{1,4}I_{1,5}$ $I_{2,4}I_{3,4}$ $I_{2,4}I_{3,4}$ $I_{2,5}I_{3,5}$	$I_{1,6}I_{1,7}$ $I_{2,5}I_{3,5}$ $I_{2,6}I_{3,6}$ $I_{2,7}I_{3,7}$	$I_{1,0}I_{1,1}$ $I_{2,6}I_{3,6}$ $I_{2,0}I_{3,0}$ $I_{2,1}I_{3,1}$	$I_{1,2}I_{1,3}$ $I_{2,7}I_{3,7}$ $I_{2,2}I_{3,2}$ $I_{2,3}I_{3,3}$
AB I1 I2 A1 A2 B1 B2	$I_{2,0}I_{2,1}$ $I_{4,0}I_{5,0}$ $I_{4,0}I_{5,0}$ $I_{4,1}I_{5,1}$	$I_{2,2}I_{2,3}$ $I_{4,1}I_{5,1}$ $I_{4,2}I_{5,2}$ $I_{4,3}I_{5,3}$	$I_{2,4}I_{2,5}$ $I_{4,2}I_{5,2}$ $I_{4,4}I_{5,4}$ $I_{4,5}I_{5,5}$	$I_{2,6}I_{2,7}$ $I_{4,3}I_{5,3}$ $I_{4,6}I_{5,6}$ $I_{4,7}I_{5,7}$	$I_{2,4}I_{2,5}$ $I_{4,4}I_{5,4}$ $I_{4,4}I_{5,4}$ $I_{4,5}I_{5,5}$	$I_{2,6}I_{2,7}$ $I_{4,5}I_{5,5}$ $I_{4,6}I_{5,6}$ $I_{4,7}I_{5,7}$	$I_{2,0}I_{2,1}$ $I_{4,6}I_{5,6}$ $I_{4,0}I_{5,0}$ $I_{4,1}I_{5,1}$	$I_{2,2}I_{2,3}$ $I_{4,7}I_{5,7}$ $I_{4,2}I_{5,2}$ $I_{4,3}I_{5,3}$
AB I1 I2 A1 A2 B1 B2	$I_{3,0}I_{3,1}$ $I_{6,0}I_{7,0}$ $I_{6,0}I_{7,0}$ $I_{6,1}I_{7,1}$	$I_{3,2}I_{3,3}$ $I_{6,1}I_{7,1}$ $I_{6,2}I_{7,2}$ $I_{6,3}I_{7,3}$	$I_{3,4}I_{3,5}$ $I_{6,2}I_{7,2}$ $I_{6,4}I_{7,4}$ $I_{6,5}I_{7,5}$	$I_{3,6}I_{3,7}$ $I_{6,3}I_{7,3}$ $I_{6,6}I_{7,6}$ $I_{6,7}I_{7,7}$	$I_{3,4}I_{3,5}$ $I_{6,4}I_{7,4}$ $I_{6,4}I_{7,4}$ $I_{6,5}I_{7,5}$	$I_{3,6}I_{3,7}$ $I_{6,5}I_{7,5}$ $I_{6,6}I_{7,6}$ $I_{6,7}I_{7,7}$	$I_{3,0}I_{3,1}$ $I_{6,6}I_{7,6}$ $I_{6,0}I_{7,0}$ $I_{6,1}I_{7,1}$	$I_{3,2}I_{3,3}$ $I_{6,7}I_{7,7}$ $I_{6,2}I_{7,2}$ $I_{6,3}I_{7,3}$
AB I1 I2 A1 A2 B1 B2	$I_{4,0}I_{4,1}$ $I_{4,0}I_{5,0}$ $I_{4,0}I_{5,0}$ $I_{4,1}I_{5,1}$	$I_{4,2}I_{4,3}$ $I_{4,1}I_{5,1}$ $I_{4,2}I_{5,2}$ $I_{4,3}I_{5,3}$	$I_{4,4}I_{4,5}$ $I_{4,2}I_{5,2}$ $I_{4,4}I_{5,4}$ $I_{4,5}I_{5,5}$	$I_{4,6}I_{4,7}$ $I_{4,3}I_{5,3}$ $I_{4,6}I_{5,6}$ $I_{4,7}I_{5,7}$	$I_{4,4}I_{4,5}$ $I_{4,4}I_{5,4}$ $I_{4,4}I_{5,4}$ $I_{4,5}I_{5,5}$	$I_{4,6}I_{4,7}$ $I_{4,5}I_{5,5}$ $I_{4,6}I_{5,6}$ $I_{4,7}I_{5,7}$	$I_{4,0}I_{4,1}$ $I_{4,6}I_{5,6}$ $I_{4,0}I_{5,0}$ $I_{4,1}I_{5,1}$	$I_{4,2}I_{4,3}$ $I_{4,7}I_{5,7}$ $I_{4,2}I_{5,2}$ $I_{4,3}I_{5,3}$
AB I1 I2 A1 A2 B1 B2	$I_{5,0}I_{5,1}$ $I_{6,0}I_{7,0}$ $I_{6,0}I_{7,0}$ $I_{6,1}I_{7,1}$	$I_{5,2}I_{5,3}$ $I_{6,1}I_{7,1}$ $I_{6,2}I_{7,2}$ $I_{6,3}I_{7,3}$	$I_{5,4}I_{5,5}$ $I_{6,2}I_{7,2}$ $I_{6,4}I_{7,4}$ $I_{6,5}I_{7,5}$	$I_{5,6}I_{5,7}$ $I_{6,3}I_{7,3}$ $I_{6,6}I_{7,6}$ $I_{6,7}I_{7,7}$	$I_{5,4}I_{5,5}$ $I_{6,4}I_{7,4}$ $I_{6,4}I_{7,4}$ $I_{6,5}I_{7,5}$	$I_{5,6}I_{5,7}$ $I_{6,5}I_{7,5}$ $I_{6,6}I_{7,6}$ $I_{6,7}I_{7,7}$	$I_{5,0}I_{5,1}$ $I_{6,6}I_{7,6}$ $I_{6,0}I_{7,0}$ $I_{6,1}I_{7,1}$	$I_{5,2}I_{5,3}$ $I_{6,7}I_{7,7}$ $I_{6,2}I_{7,2}$ $I_{6,3}I_{7,3}$
AB I1 I2 A1 A2 B1 B2	$I_{6,0}I_{6,1}$ $I_{0,0}I_{1,0}$ $I_{0,0}I_{1,0}$ $I_{0,1}I_{1,1}$	$I_{6,2}I_{6,3}$ $I_{0,1}I_{1,1}$ $I_{0,2}I_{1,2}$ $I_{0,3}I_{1,3}$	$I_{6,4}I_{6,5}$ $I_{0,2}I_{1,2}$ $I_{0,4}I_{1,4}$ $I_{0,5}I_{1,5}$	$I_{6,6}I_{6,7}$ $I_{0,3}I_{1,3}$ $I_{0,6}I_{1,6}$ $I_{0,7}I_{1,7}$	$I_{6,4}I_{6,5}$ $I_{0,4}I_{1,4}$ $I_{0,4}I_{1,4}$ $I_{0,5}I_{1,5}$	$I_{6,6}I_{6,7}$ $I_{0,5}I_{1,5}$ $I_{0,6}I_{1,6}$ $I_{0,7}I_{1,7}$	$I_{6,0}I_{6,1}$ $I_{0,6}I_{1,6}$ $I_{0,0}I_{1,0}$ $I_{0,1}I_{1,1}$	$I_{6,2}I_{6,3}$ $I_{0,7}I_{1,7}$ $I_{0,2}I_{1,2}$ $I_{0,3}I_{1,3}$
AB I1 I2 A1 A2 B1 B2	$I_{7,0}I_{7,1}$ $I_{2,0}I_{3,0}$ $I_{2,0}I_{3,0}$ $I_{2,1}I_{3,1}$	$I_{7,2}I_{7,3}$ $I_{2,1}I_{3,1}$ $I_{2,2}I_{3,2}$ $I_{2,3}I_{3,3}$	$I_{7,4}I_{7,5}$ $I_{2,2}I_{3,2}$ $I_{2,4}I_{3,4}$ $I_{2,5}I_{3,5}$	$I_{7,6}I_{7,7}$ $I_{2,3}I_{3,3}$ $I_{2,6}I_{3,6}$ $I_{2,7}I_{3,7}$	$I_{7,4}I_{7,5}$ $I_{2,4}I_{3,4}$ $I_{2,4}I_{3,4}$ $I_{2,5}I_{3,5}$	$I_{7,6}I_{7,7}$ $I_{2,5}I_{3,5}$ $I_{2,6}I_{3,6}$ $I_{2,7}I_{3,7}$	$I_{7,0}I_{7,1}$ $I_{2,6}I_{3,6}$ $I_{2,0}I_{3,0}$ $I_{2,1}I_{3,1}$	$I_{7,2}I_{7,3}$ $I_{2,7}I_{3,7}$ $I_{2,2}I_{3,2}$ $I_{2,3}I_{3,3}$

Figure 14: Configuration after the four PAIRING operations in Figure 13 for the case N = 8 and M = 4

Parallel Solutions to Geometric Problems on the Scan Model of Computation*

Guy E. Blelloch and James J. Little
M.I.T. Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139

Abstract

This paper describes several parallel algorithms that solve geometric problems. The algorithms are based on a vector model of computation — the *scan-model*. The purpose of this paper is both to illustrate how the model can be used and to describe a set of simple algorithms.

We describe a k-D tree algorithm that, for n points, requires $O(\lg n)$ calls to the primitives, a line-drawing algorithm that requires $O(1)$ calls to the primitives, a line-of-sight algorithm that requires $O(1)$ calls to the primitives, and finally two convex-hull algorithms. All these algorithms should be noted for their simplicity rather than complexity; many of them are parallel versions of known serial algorithms or variants of known parallel algorithms.

Most of the algorithms discussed in this paper have been implemented on the Connection Machine, a highly parallel single instruction multiple data (SIMD) computer.

1 Introduction

The purpose of this paper is twofold. Firstly, it describes a set of elegant, practical algorithms for solving a diverse set of problems in computational geometry and graphics. Secondly, it helps demonstrate that the *scan-model* is a viable model of computation. These two purposes complement each other: the model allows a simple description of the algorithms, and the algorithms demonstrate the power of the model.

Researchers have suggested several synchronous parallel models of computation. The most popular of these models are the parallel random access machine (P-RAM) models [13]. A P-RAM consists of a set of conventional processors attached to a single shared memory. Processors communicate through the shared memory: one processor can write a value into the memory and another processor can read this value. Researchers have suggested several variations of the P-RAM models. These variations mostly differ in whether or not they permit concurrent reads from, or concurrent writes to, a unique memory location. By assuming that memory references take *unit-time*, the P-RAM models have been used to determine the asymptotic running time of many parallel algorithms.

We suggest another class of synchronous parallel models of computation defined in terms of a set of primitive operations that operate on arbitrarily long vectors of atomic values. We call these models vector models [8]. The models differ from P-RAM models both in that they are single instruction multiple data (SIMD) models, and in that there is no concept of a memory shared among many processors. Elements in a vector communicate through a permutation primitive rather than a shared memory. As with the P-RAM models, vector models can be used to analyze the asymptotic running time of algorithms, by making assumptions about the relative running times of the primitives.

Since vector models are SIMD, they can be efficiently mapped onto a wider range of architectures than P-RAM models can. As well as being implementable on standard serial computers and on multiple instruction parallel computers, they can be efficiently implemented on vector processors, such as the vector processor of the CRAY systems [24], or single instruction parallel computers, such as the Connection Machine [16]. On the other hand, since P-RAM models are multiple instruction multiple data (MIMD) models, they are more powerful than vector models. As should become evident in this paper, and as argued elsewhere [8], this additional power is not necessary for a broad range of practical algorithms. We also believe that vector models tend to lead to simpler and more concrete algorithm descriptions than do P-RAM models.

The *scan-model* is a particular vector model. The name comes from the inclusion of a *scan*, also known as prefix, primitive. In this paper we describe several algorithms based on the scan-model. Most of the algorithms we describe in this paper have been implemented on the Connection Machine. All the algorithms described in this paper are described in more detail in [10,8]. Before describing the algorithms, we define the scan-model and introduce some techniques based on the model.

2 The Scan-Model

The scan-model [8] is defined in terms of a set of primitive operations that operate on arbitrarily long vectors of atomic values. By a vector we mean a one dimensional array (an ordered set). By atomic values we mean values that can be represented in $O(\lg n)$ bits — in this paper we only use integers, floating point numbers and boolean values. We assume that the primitives require approximately an equivalent duration of time when operating on equal length vectors. The scan-model has three classes of primitives: elementwise arithmetic and logical operations, permutation operations, and scan operations, a type of parallel prefix computation.

Each *elementwise* primitive operates on equal length vectors, producing a result vector of equal length. The element i of the result is an elementary arithmetic or logical primitive — such as $+, -, *$, or and not — applied to element i of each of the input vectors. For example:

A	=	[5	1	3	4	3	9	2	6]
B	=	[2	5	3	8	1	3	6	2]
A + B	=	[7	6	6	12	4	12	8	8]
A × B	=	[10	5	9	24	3	27	12	12]

The *permutation* primitive takes two vector arguments — a *data vector* and an *index vector* — and permutes each element in the data vector to the location specified in the index vector. For example:

Vector Index	=	[0	1	2	3	4	5	6	7]
A (data vector)	=	[o	t	e	m	e	r	g	y]
I (index vector)	=	[2	5	4	3	1	6	0	7]
permute(A, I)	=	[g	e	o	m	e	t	r	y]

It is an error for more than one element to have the same index — the permutation must be one-to-one. This restriction is similar to the restriction made in the exclusive read exclusive write (EREW) P-RAM model. To allow communication between vectors of different sizes, we include a version of the permute primitive that returns a vector of different length than the source vectors by masking out elements or putting in defaults.

The scan primitives execute a scan operation, sometimes called a prefix computation, on a vector. The scan operation takes a binary associative operator $\oplus$, and a vector $[a_0, a_1, ..., a_{n-1}]$ of n elements, and returns the vector $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$. In this paper we will only use plus, maximum, minimum, or and and as operators for the scan primitives. We will henceforth call these scan operations +-scan, max-scan, min-scan, or-scan and and-scan. Some examples:

* This report describes research done within the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology. Support for the A.I. Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Army contract DACA76-85-C-0010 and in part under the Office of Naval Research contract N00014-85-K-0124.

A	=	[5	1]	[3	4	3	9]	[2	6]
B	=	[1	0]	[2	0	3	1]	[0	1]
+-scan(A)	=	[5	6]	[3	7	10	19]	[2	8]
permute(A, B)	=	[1	5]	[4	9	3	3]	[2	6]

Figure 1: Examples of the segmented versions of the primitive operations.

A	=	[5	1	3	4	3	9	2	6]
+-scan(A)	=	[5	6	9	13	16	25	27	33]
max-scan(A)	=	[5	5	5	5	5	9	9	9]

In [9,8] we argue why, in the analysis of algorithms, the scan primitives should be considered no more expensive (timewise) than the permutation primitive. The basic argument is that the scan primitives can be implemented, both in practice and in theory, to run as fast as the permutation primitive.

In the description of algorithms we will often loosely refer to vectors in which each element contains some fixed number of atomic values. At the primitive level such a structure vector would be implemented with multiple vectors but a higher level language could support record-like vectors in which each element has some constant number of values.

2.1 Segments

This section describes a method that allows a programmer to take a vector routine defined to operate on a single set of data and then apply it to many sets in parallel. For example, if we had a vector routine that sorted a set of values, we could apply it to sort many sets of data in parallel. Or, if we had a vector routine that, given endpoints, determines the pixels on a line, we could apply it to draw many lines in parallel.

The technique involves dividing a vector into segments and placing one set of data in each segment. To keep track of how a data vector is segmented, we associate with the data vector a *segment-descriptor*. A *segment-descriptor* is itself a vector which has as many elements as segments of the data vector; each of these elements contains an integer which specifies the length of the segment[1]. For example:

A'	=	[5	1	3	4	3	9	2	6]
segment-descriptor	=	[2	4	2]					
A	=	[5	1]	[3	4	3	9]	[2	6]

Henceforth, the notation

$$A = [5\ 1]\ [3\ 4\ 3\ 9]\ [2\ 6]$$

is shorthand for a pair of vectors: the data vector along with its *segment-descriptor*.

For each primitive of the scan-model we define a segmented version that operates independently within each segment. Figure 1 illustrates examples of segmented versions of the primitives. The segmented version of the permutation primitive bases its indices relative to the beginning of each segment so values permute within a segment — it is an error for an index to reference outside of the segment. The segmented version of the scans primitives restart at the beginning of each segment[2]. The segmented version of the elementwise operations are unchanged. All the segmented versions can be simulated with a small constant number of calls to the unsegmented versions [8].

> **The Segment Lemma:** With a segmented version of all the primitives of the scan-model, we can apply any routine defined in terms of those primitives to operate on a single set of data, to multiple sets of data independently and in parallel.

We won't prove this lemma in this paper, but it should be intuitive; a proof is given in [8]. This lemma allows great simplification of the code needed to describe parallel algorithms.

[1] There are several other ways of representing segments [8] but we find this representation the most convenient.

[2] A similar operation was suggested by Schwartz [26].

A	=	[7	3	8]						
L	=	[2	4	2]						
I	=	[0	2	1]						
B	=	[5	1]	[3	4	3	9]	[2	6]	
C	=	[1	0]	[2	1	3	0]	[0	1]	
F	=	[T	F]	[T	F	F	T]	[T	T]	
distribute(A, L)	=	[7	7]	[3	3	3	3]	[8	8]	
index(L)	=	[0	1]	[0	1	2	3]	[0	1]	
element(A, I)	=	[5	3	6]						
+-reduce(B)	=	[6	19	8]						
max-reduce(B)	=	[5	9	6]						
pack(B, F)	=	[5]	[3	9]	[2	6]				
split(B, F)	=	[1]	[5]	[4	3]	[3	9]	[]	[2	6]
delete-split(B, F)	=	[1]	[5]	[4	3]	[3	9]	[2	6]	
rank-split(C, F)	=	[0]	[0]	[0	1]	[1	0]	[]	[0	1]

Figure 2: Examples of a set of simple operations based on the primitives.

2.2 Some Simple Operations

In this section we describe several useful, simple operations that can be implemented with a small constant number of calls to the primitive operations [9]. As with the segmented versions of the primitives, these operations are useful enough that they might themselves be considered primitives and be implemented directly. Many of these operations are similar to primitives of APL [18]. Figure 2 illustrates examples of each of the operations.

The distribute operation takes a vector of *values* and a vector of *lengths* and distributes each value into a segment of length specified by *lengths*. The index operation takes a vector of *lengths*, creates a segment for each length, and returns the index of each element within each segment. The element operation takes a segmented vector *values*, and a vector of *indices* with one element per segment. Each index i is used to extract the i^{th} element from the corresponding segment in *values*. The reduce operation takes a segmented vector of *values* and combines all the elements in each segment using one of five binary operators: +, maximum, minimum, or or and. It returns a vector with as many elements as segments.

The append operation takes two segmented vectors of *values* with the same number of segments and appends the two vectors segmentwise. The pack operation takes a segmented vector of *values* and a segmented boolean vector of *flags*, and packs all the elements with a T in their flag into consecutive elements, deleting elements with an F in their flag. The split operation takes a segmented vector of *values* and a segmented boolean vector of *flags*, and packs all the elements with an F in their flag to the bottom of each segment and elements with a T in their flag to the top of each segment. It also splits each segment in two at the boundary between the T and F elements. We also define a delete-split operation which is the same as split but deletes any empty segment. The rank-split operation is similar to the split operation except that the *ranks* argument must be a valid set of indices for the permutation primitive. As well as splitting these indices, the rank-split operation renumbers them so they are valid within the new segments but maintain the same order.

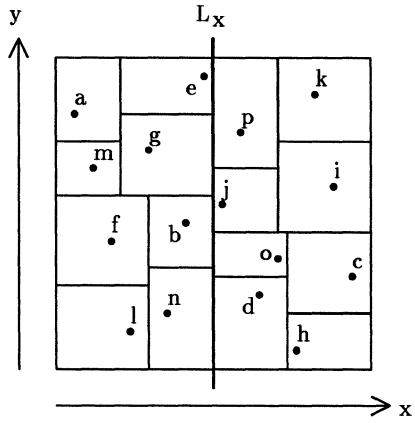
2.3 Recursive Splitting

The segment abstraction and the primitives we described allow simple definitions of recursive routines that start with some set of values, split this set into subsets and recursively solve the problem on each subset. We will call this technique *recursive splitting*. As an example of such a technique, consider the following parallel version of quicksort. As with the serial algorithm, the algorithm picks one of the keys as a pivot value, splits the keys into two sets, one with greater valued keys and one with lesser valued keys, and then recurses on each set.

The parallel version picks a random element from each segment as a pivot value using the element operation[3]. The algorithm distributes this pivot value over each segment using a distribute operation, and splits the keys based on whether a key is greater or less than the pivot using the delete-split operation[4]. The algorithm is now applied recursively to

[3] I assume that there is a primitive elementwise random operation which in each element takes an integer and returns a pseudo-random number less than that integer.

[4] We use the delete-split operations instead of the split operation so that we never



point	=	[a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p]
x-rank	=	[0	6	15	10	7	2	4	12	14	8	13	3	1	5	11	9]
y-rank	=	[13	7	4	3	15	6	11	0	9	8	14	1	10	2	5	12]
above-split-line?		[F	F	T	T	F	F	F	T	T	T	T	F	F	F	T	T]
rank-split x-rank	=	[0	6	7	2	4	3	1	5]	[7	2	4	6	0	5	3	1]
rank-split y-rank	=	[6	3	7	2	5	0	4	1]	[2	1	0	5	4	7	3	6]

Figure 3: An example of a 2-D tree. The top diagram shows the final splitting. The vectors below are generated during the first step — when splitting along the line L_x.

the result. When the numbers within all segment are in non decreasing order, we return. As with the serial algorithm, this algorithm is expected to complete in $O(\lg n)$ steps[5]. In the scan-model, each step requires a small constant number of operations.

The code needed to implement quicksort in the scan-model is as follows:

```
define quicksort(keys){
    if-any (shift-left(keys) < keys)
        then
            pivots ← element(keys, random(length(keys)));
            flags ← distribute(pivots, length(keys)) ≤ keys;
            quicksort(delete-split(keys, flags ));
        else keys}
```

This general recursive splitting technique can be used in many divide and conquer algorithms. In this paper we will use it in the k-D tree algorithm discussed in Section 3, and the quickhull algorithm discussed in Section 6.1.

3 Building a k-D Tree

A k-D tree is technique for splitting n points in a k dimensional space into n regions each with a single point [6]. It starts by splitting the space in two along one of the coordinates using a $k-1$ dimensional hyperplane. It then recursively splits each of the subspaces in two. Figure 3 illustrates an example of a 2-D tree. At each step the algorithm must select which dimension to split within each subspace; the criterion for selection depends on how the tree will be used. A common criterion is to select the dimension along which the spread of points is greatest.

The k-D tree is often used as a step in other algorithms. 3-D trees are used in ray tracing algorithms for rendering solid objects. In such algorithms, objects need only be stored in the regions they penetrate and rays need only examine regions they cross. This can greatly reduce the number of objects each ray needs to examine. k-D trees are also used in many proximity algorithms such as the all closest pairs problem [15] or the closest pair problem. k-D trees have also been suggested for use in some machine learning algorithms [20].

The algorithm we describe here is a parallel version of a standard serial algorithm [22]. For n points, our algorithm takes $O(k \lg n)$ calls to the primitives on vectors of length n. This algorithm is optimal in the sense that even if simulated on a serial machine, it will run in the same asymptotic running time as the best serial algorithm.

Our algorithm consists of one step per split. Each step requires $O(k)$ calls to the primitives. Before executing any steps, the algorithm sorts the set of points according to each of the k dimensions. The sorting can be executed with the quicksort algorithm discussed earlier, an enumerate-pack sorting algorithm discussed in [9], or a version of Cole's sorting algorithm [11]. Instead of keeping the actual values in sorted order for each dimension, we keep the rank of each point along each dimension. The rank of a point is the position the point would be located at if the vector were sorted. We call the vectors that hold these ranks, *rank-vectors* — there is one *rank-vector* for each dimension. Figure 3 illustrates an example for a 2-D tree, the initial *rank-vectors*, and the result of the first step.

At each step of the algorithm the *rank-vectors* will contain a segment for each subspace, and the ranks within each segment will be the correct ranks for that subspace. It suffices to demonstrate that we can execute a split along any dimension and generate new ranks within the two subspaces. The algorithm is then correct by induction.

To split along a given dimension the algorithm distributes the cut line and determines for each point whether it is above or below the line[6]. The algorithm now uses the rank-split operation defined in Section 2.2 to split each *rank-vector* based on whether a point is below or above the split line. The rank-split operation as defined correctly generates the rank within each subspace. Each step therefore requires $O(k)$ calls to the primitives: some operations to determine whether each point is below or above the split, and k rank-split operations. Since there are $O(\lg n)$ steps, the whole algorithm requires $O(k \lg n)$ calls to the primitives.

A two dimensional closest-pair algorithm can be implemented based on the k-D tree algorithm. This algorithm is a parallel version of an algorithm of Bentley and Shamos [7] and is described in [10]. For n points in a two dimensional space, our algorithm requires $O(\lg n)$ calls to the primitives using vectors of length $O(n)$.

4 Line Drawing

Two dimensional line drawing is the problem of: given a pair of points on a two dimensional grid (the two endpoints of a line), determine what pixels in a finite resolution grid lie on a line between the endpoints. Line drawing is used extensively in practice in generating computer images, especially in computer aided design. In this section we describe a very simple line drawing routine. It generates the same set of pixels as generated by the simple digital differential analyzer (DDA) serial technique [19]. The routine takes a small constant number of calls to the primitives on vectors at most as long as the number of pixels in the output. Because of the **segment lemma** (Section 2.1), the routine can be used to draw many lines in parallel. The routine we describe has been extended by Salem [25] to render solid objects.

The basic idea of the routine is to calculate the number of pixels in a line and allocate a set of vectors of that length with the line information distributed across the vectors. Then based on the line information and a unique index for each element, the elements can calculate their final position on the grid. The number of pixels in a line is one more than the maximum of the x and y differences — we will call this number L. We distribute one endpoint and the slope of the line across vectors of length L using the distribute operation and generate the unique index for each position in the vector with the index operation. Based on the index, the endpoint, and the slope, we can calculate the position of each pixel using some simple arithmetic. This is described in more detail in [10].

5 Line of Sight

Given an $\sqrt{n}$ by $\sqrt{n}$ grid of altitudes and an observation point on or above the surface, a line of sight algorithm finds all points on the grid

have more segments than elements.

[5] This is actually only true if either the keys are unique, or we split into three groups at each step ($<, =, >$), or we switch between $\leq$ and $\geq$ on alternating steps.

[6] As stated earlier, the method for choosing a cut line will depend on the particular use of the k-D tree.

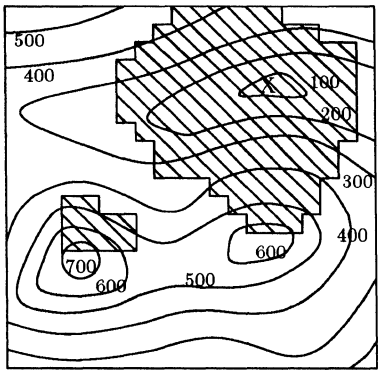


Figure 4: An example of a line of sight problem. The X marks the observation point. The numbers represent the altitude of each contour line. The elements visible from the observation point are shaded.

visible from the observation point. Figure 4 illustrates an example. A line of sight algorithm can be applied to help determine where to locate potential eyesores. For example, when designing a building, a highway or a city dump, it is often informative to know from where the "eyesore" will be visible.

The algorithm we describe requires $O(1)$ calls to the primitives using vectors of length $O(n)$. The basic idea is to allocate a segment in a vector for every ray that propagates in the plane from the observation point, henceforth referred to as X, to a boundary position. Based on some calculations on the points in each ray, we can determine if the point is visible.

The algorithm consists of four basic steps. Each point p in the grid calculates the vertical angle between the horizontal plane that passes through X (the observation point) and the line from p to X. Secondly, the algorithms allocates a set of rays — one for each boundary grid point — and distributes the angles from each point p in the grid to all the rays it belongs to. Each ray is a segment in a vector we will call the *ray structure*. Thirdly, following a ray from X to the boundary, a point p is visible if its angle is greater than all the angles that precede it in the ray. This can be determined for all points in all rays with a single segmented max-scan, and a comparison. Fourthly, visibility information is returned back to the grid points. Since a grid point can have a position in many rays, the visibility flags are combined using an or-reduce. Some permutations are required to distribute the information to the ray structures and to reduce it back to the grid; this is described in [10].

The longest vectors required by the algorithm will be the vectors of the copy and ray structures. It is not hard to show that for a $\sqrt{n}$ by $\sqrt{n}$ grid, independent of the location of X, these vectors will have length $2n$.

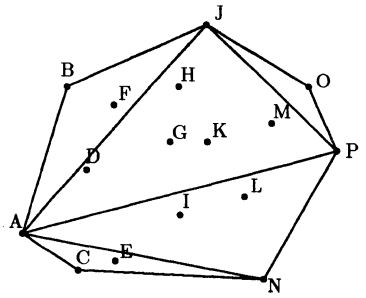
6 Convex Hull

The planar convex hull problem is: given n points in the plane, find which of these points lie on the perimeter of the smallest convex region that contains all points. The planar convex hull problem is probably the most studied problem in computational geometry, both because it is a simple problem, making it easy to study, and because it has many applications — applications range from computer graphics [14] to statistics [17].

In this section we describe two scan-model based algorithms for determining the convex hull of a set of points. The first algorithm, a parallel quickhull algorithm, is very simple and likely to perform well in practice but is not provably optimal. The second algorithm is more complicated and impractical but is theoretically optimal. The algorithm is based on a parallel algorithm designed for the concurrent read exclusive write (CREW) P-RAM model [1,3].

6.1 QuickHull

This is a parallel version of the quickhull algorithm [22,12]. The quickhull algorithm was given its name because of its similarity with the quicksort algorithm. Like quicksort, the quickhull algorithm picks a pivot element, a point; splits the data based on the pivot; and is then recursively applied to each of the split sets. Also like quicksort, the pivot element is not guaranteed to split the data into sets with any particular ratio of sizes, so that in the worst case, the algorithm can require n steps.



[A B C D E F G H I J K L M N O P]

A [B D F G H J K M O] P [C E I L N]

A [B F] J [O] P N [C E]

A B J O P N C

Figure 5: An example of the quickhull algorithm. Each vector shows one step of the algorithm. The line AP is the original split line. J and N are the farthest points in each subspace from AP, and are therefore used for the next level of splits. The values outside the brackets are hull points that have already been found.

Figure 5 illustrates an example of the quickhull algorithm. The algorithm first splits the points into two sets with a line that passes between the two x extrema — lets call these points l and r. In the scan-model this is executed with a few reduce and distribute operations, some elementwise arithmetic calculations, and a split operation.

The algorithm now recursively splits each of the two subspaces into two using the following steps. It determines for each point p in the subspace the perpendicular distance from the point to the line lr. This can be calculated with a cross product of the lines lr and lp. The algorithm selects the farthest point from the line lr and distributes it to all other elements in the subspace — lets call this point t. It should be clear that t lies on the convex hull. Points within the triangle ltr cannot be on the convex hull and are eliminated with a pack operation. The point t is now used to further split each segment based on which of the two sides of the triangle, lt or rt, they fall. The algorithm is now applied to the new segments recursively. The algorithm is completed when all segments are empty.

Each step requires a small constant number of calls to the primitives. As with the serial quickhull, for m hull points, the algorithm runs in $O(\lg m)$ steps for well distributed hull points, and has a worst case running time of $O(m)$ steps.

6.2 $\sqrt{n}$ Merge Hull

This algorithm is a variation of a parallel algorithm suggested in [1] and independently in [3]. Their algorithm is based on the concurrent read, exclusive write (CREW) P-RAM model. We cannot use their algorithm directly because the scan-model does not permit concurrent access to a single value, a necessary part of their algorithm. The variation we describes keeps all elements that require the same data in a contiguous segment so the data can be distributed using a distribute operation. The contribution of our version is showing how the concurrent read operation can be replaced by the distribute operation and involves a tree search method discussed in the next section. Like the original algorithm, the variation we describe runs with $O(\lg n)$ calls to the primitives.

Unfortunately, we do not have space here to review the CREW algorithm and we refer the reader to the two papers mentioned above. A review of the CREW and a more complete description of our variation can also be found in [10]. The only difficult part of converting the CREW algorithm to the scan-model is in the step that finds the upper tangent line-segments when merging the $\sqrt{n}$ subhulls into a single

convex hull. The step uses the algorithm of Overmars [21] to find the upper tangent line segment for all pairs of subhulls. Overmars method executes a binary search alternating between the two subhulls, and requires $O(\lg n)$ time. At the k^{th} step of the binary search, an element will either go down the left branch, the right branch or will stay still in the search tree.

The merging step cannot be implemented directly on the scan-model since each pair of subhulls independently finds the upper tangent-line segments using the algorithm of Overmars, and will therefore require concurrent reads: several pairs, while executing the binary search, will require access to the same elements. To avoid the concurrent read, we place each of the sets of $\sqrt{n}$ points that belong to the same subhull in its own segment. We then use a general binary search method described in [10] to execute the binary search. This search will require $O(\lg n)$ time. The basic idea of this binary search method is to use the **split** operation to split the elements going down each branch of the search, and to use the **append** operation to append the elements that stay at a vertex of the tree to the elements that come down from a parent. This guarantees that all elements at any vertex of the search tree are in a contiguous segment so that the **distribute** operation can be used to distribute information to them. Each element also needs to keep a pointer to the matching element in the other subhull to which it is trying to find an upper tangent line.

Our variation of the CREW algorithm runs with the same number of calls to the primitives as the original since, as with the original, the sort runs in $O(\lg n)$ time, and, as argued above, the merge also runs in $O(\lg n)$ time.

7 Conclusions

This paper introduces the idea of a vector model of computation; defines a particular vector model, the scan-model; and describes several algorithms implemented on the scan-model. Since many of the algorithms discussed in this paper are variants of known algorithms, we believe that much of the contribution of this paper is to methodology rather than to algorithms.

We believe that the algorithms we describe are very practical for implementation on a wide range of architectures, both serial and parallel, and should in most cases be almost as fast on a particular architecture as algorithm designed specifically for that architecture[7]. This generality is one of the main advantages of the scan-model over the P-RAM models. The advantage arises both because the scan-model is a vector model, allowing efficient implementations on vector processors and single instruction parallel processors, and because it treats the scan operation as taking no more time than a permutation, a realistic assumption for almost all architectures. The algorithms described in this paper have been implemented on the Connection Machine.

In more recent research we have been considering the effect of including other operations as primitives. The operation we have found most promising is a variation of the merge operation[8]. This operation can be implemented efficiently on a wide range of architectures and is useful for many algorithms. To implement the merge operation on serial architectures we can use the standard merge operation, and on parallel architectures we can use a variation of Batcher's bitonic merge network [5]. Algorithms to construct and manipulate the plane-sweep tree data structure [2,1,4,23] can be greatly simplified with a primitive merge operation. We have also found the merge primitive useful for manipulating sets. We have also considered sorting as a primitive, but we find it hard to argue that sorting should be assumed to require the same time as a permutation.

We hope that the paper will help spur further interest in designing algorithms for vector models of computation.

Acknowledgments

We would like to thank Charles Leiserson and Guy Steele for their contributions. We would also like to thank Thinking Machines for the opportunity to work on the Connection Machine.

[7] This is not true for architectures with low connectivity such as grid architectures or tree architectures.

[8] Given two vectors A and B of numbers, it returns a vector C of length A with indices into the vector B. These indices point to where in B an element in A should merge.

References

[1] Alok Aggarwal, Bernard Chazelle, Leo Guibas, Colm Ò'Dùnlaing, and Chee Yap. Parallel computational geometry. In *Proceedings Symposium on Foundations of Computer Science*, pages 468–477, October 1985.

[2] Mikhail J. Atallah, Richard Cole, and Michael T. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. In *Proceedings Symposium on Foundations of Computer Science*, pages 151–160, October 1987.

[3] Mikhail J. Atallah and Michael T. Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3(4):492–507, December 1986.

[4] Mikhail J. Atallah and Michael T. Goodrich. Efficient plane sweeping in parallel. In *Proceedings ACM Symposium on Theory of Computing*, pages 216–225, 1986.

[5] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[6] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[7] Jon L. Bentley and Michael I. Shamos. Divide-and-conquer in multidimensional space. In *Proceedings ACM Symposium on Theory of Computing*, pages 220–230, 1976.

[8] Guy E. Blelloch. *Scans and Other Primitives for Parallel Computation.* PhD thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, to be completed June 1988 (hopefully).

[9] Guy E. Blelloch. Scans as primitive parallel operations. In *Proceedings International Conference on Parallel Processing*, pages 355–362, August 1987.

[10] Guy E. Blelloch and James J. Little. *Parallel Solutions to Geometric Problems on the Scan Model of Computation.* Technical Report TM-952, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1988.

[11] Richard Cole. Parallel merge sort. In *Proceedings Symposium on Foundations of Computer Science*, pages 511–516, October 1986.

[12] W. F. Eddy. A new convex hull algorithm for planar sets. *ACM Trans. Math. Software*, 3(4):398–403, 1977.

[13] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[14] H. Freeman. Computer processing of line-drawing images. *Computer Surveys*, 6:57–97, 1974.

[15] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.

[16] William D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[17] J. G. Hocking and G. S. Young. *Topology.* Addison-Wesley, Reading, MA, 1961.

[18] Keneth Iverson. *A Programming Language.* Wiley, New York, 1962.

[19] William M. Newman and Robert F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, New York, 1979.

[20] Stephen M. Omohundro. Efficient algorithms with neural network behavior. *Complex Systems*, 1, 1987.

[21] Mark H. Overmars and Jan Van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[22] Franco P. Preparata and Michael I. Shamos. *Computational Geometry — An Introduction.* Springer-Verlag, New York, 1985.

[23] John H. Reif and Sandeeep Sen. Optimal randomized parallel algorithms for computational geometry. In *Proceedings International Conference on Parallel Processing*, pages 270–277, August 1987.

[24] Richard M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[25] James Salem. **Render: A Data Parallel Approach to Polygon Rendering.* Technical Report, Thinking Machines Corporation, January 1988.

[26] Jacob T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, October 1980.

PARALLEL TEMPLATE MATCHING ALGORITHMS

Xiaoqing Qu and Xiaobo Li
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract -- This paper describes two parallel template matching algorithms on an SIMD array processor with a hypercube interconnection network. These algorithms improve the algorithm proposed by Fang et al[1]. The first algorithm proposed in this paper modifies the local address computation scheme, so that only one permute-multiply phase is needed. The computation time is reduced to half. The second algorithm treats window columns the same way as rows. Permute-multiply operations in Gray code sequence are also implemented among columns. For an N by N image and M by M window, the overall time complexity is reduced from $O(M^2+MlogN)$ to $O(M^2+logN)$. The trade-off is that the local memory size of each PE is increased from M to M^2.

Introduction

Image template matching is a basic image processing operation. A large N by N image G is searched with an M by M template (window). The similarity measure between the window and the M by M subimage of G is defined as $C_{ij} = \sum_{s=0}^{M-1}\sum_{t=0}^{M-1} G_{i+s,j+t} * W_{st}$ where $G(i,j)$ is the upper-left corner element of the subimage. In filtering operation, array C is the final result. The tremendous computation load ($O(N^2M^2)$) and the independence of subimages make the template matching suitable for SIMD machines.

Several parallel template matching algorithms have been proposed [1-2]. Fang's algorithm [1] uses hypercube interconnection network, because of its popularity in commercial products[3-6]. The algorithm on N^2 PEs proposed in [1]. with the time complexity $O(M^2+MlogN)$. requires that each PE has M local memory locations to store a window column and a register initialized to image element G_{ij}. The window elements are broadcast to PEs one column at a time. Within each image column, image elements in registers are permuted according to Gray code, then multiplied with the corresponding window element in local memory. The computation of local address is designed to provide correct correspondence. This broadcast-permute-multiply operation above is repeated M times, one window column each.

In this paper, we propose two algorithms improved over the above algorithm. Section 2 describes a different scheme for computing the local address, so that the inter-PE communication time is cut in half. In Section 3, an algorithm is given to implement the Gray code permutation to window columns as well as to window rows. Since the permutation-multiplication is implemented in 2 dimensions instead of one dimension, the overall inter-PE communication time is reduced to M^2+logN.

The second author is supported in part by the Canadian National Science and Engineering Research Council under Grant A9198.

Local Address Computation

In this section, we present an algorithm CUBE-N2-1, an improved version of CUBE-N2 given in [1], on a hypercube network with N^2 PEs. It is assumed that image G has been distributed in N^2 PEs, i.e. $PE(i,j)=G_{ij}$. Each column of image G is divided to N/M segments, where each segment has M consecutive elements. The window elements are broadcast from control memory to local memory, one column at a time during executing the algorithm. Also, we assume that $N=2^n$ and $M=2^m$.

In the original algorithm CUBE-N2 given in [1], there are two phases (phase 2 and 3) to perform permute-multiply-accumulate operations. Careful investigation reveals that in phase 2 the PEs without "*" mark could also perform multiplication which is not used in generating C_{ij} at its own location but for $C_{i-M,j}$. The algorithm CUBE-N2-1 presented in this section has only one phase, phase 2, to perform permute-multiply-accumulate steps. Every PE is permitted to perform multiplication in phase 2. The result of multiplications with mark "*" is accumulated to one register, say C, whereas the result of multiplications of the other PEs without "*" mark is accumulate to another register, say B. After phase 2, the partial result in B is rotated one segment up and added to C to get the final result of C_{ij} at each $PE(i,j)$. In order for all PEs to perform correct multiplications, a new Lemma 3 is presented below for local address computation. Since the address may become negative, to obtain correct local addresses, the window column is viewed as a circular array.

Lemma 3: After executing $A(p^{(k)})$<—$A(p)$, the $MAR(p)$ is incremented by 2^k if p_k=0, decremented by 2^k if p_k=1.

The algorithm CUBE-N2-1 is given below. This algorithm computes $C(i,j) = C_{ij}$, $0 \le i,j \le N-1$. The procedure uses three registers, A, B and C. It is assumed that $A(i,j)$ is initialized to G_{ij}. B and C are used to accumulate the partial results during execution and initialized to 0. The PEs are indexed by p or (i,j) with $p=iN+j$. A data movement example for N=32 and M=8 is given in Fig. 1.

```
procedure CUBE-N2-1(A,B,C)
begin
1   C(p):=0;
2   B(p):=0;
3   for t:=0 to M-1 do begin
4     MAR(p):=0;
5     for s:=0 to M-1 do
6       CMAR:=α+t*M+s;
7       M[MAR(p)]<=CM[CMAR];
8       MAR(p):=MAR(p)+1;
9     end for;
```

```
10      MAR(p):=0;
11      C(p):=C(p)+A(p)*M[MAR(p)];
12      PHASE-2;
13      A(p^(n+m-1))<—A(p);
14      ROTATE(A,0,n-1,0);
15    end for;
16    ROTATE(B,n+m,2n-1,0);
17    C(p):=C(p)+B(p);
end
```

procedure PHASE-2

```
begin
 1    MAR(p):=0;
 2    for k:=n to n+m-1 do begin
 3      A(p^(k))<—A(p);
 4      MAR(p):=(MAR(p)+2^(k-n)) mod M  (p_k=0);
 5      MAR(p):=(MAR(p)-2^(k-n)) mod M  (p_k=1);
 6      C(p):=C(p)+A(p)*M[MAR(p)] (p_k=0);
 7      B(p):=B(p)+A(p)*M[MAR(p)] (p_k=1);
 8      F:=false;
 9      U:=k-n;
10      Gray(U,F);
11    end for
end
```

procedure GRAY(U,F)

```
begin
  if U=0 then return else
  begin
 1    flag:=true;
 2    GRAY(U-1,flag);
 3    A(p^(U+n-1))<—A(p);
 4    if F then
 5      MAR(p):=(MAR(p)+2^(U-1)) mod M  (p_(U+n-1)=0);
 6      MAR(p):=(MAR(p)-2^(U-1)) mod M  (p_(U+n-1)=1);
 7    else
 8      MAR(p):=(MAR(p)-2^(U-1)) mod M  (p_(U+n-1)=0);
 9      MAR(p):=(MAR(p)+2^(U-1)) mod M  (p_(U+n-1)=1);
10    end if;
11    C(p):=C(p)+A(p)*M[MAR(p)]  (p_k=0);
12    B(p):=B(p)+A(p)*M[MAR(p)]  (p_k=1);
13    flag:=false;
14    GRAY(U-1,flag);
  end if;
end;
```

In the algorithm above, It takes M^2 steps to broadcast window elements to local memory, and $O(M^2)$ multiplications to compute the C_{ij}. With the same initialization, these two terms are the same for any network. So we only consider the inter-PE communication time.

In the loop t of CUBE-N2-1, PHASE-2 takes approximately M unit routes[1]. Rotating one column left of line 14 takes $logN$ unit routes. Outside the loop t, rotating one image segment up of line 16 requires $logN-logM$ unit routes. The total inter-PE communication time is $M(M+logN)+logN-logM$, which is $O(M^2+MlogN)$. Comparing with the $2M^2+2M(logN-logM)+MlogN$ inter-PE communication time given in [1], the time saving is about half, though it is still in the same order as that in algorithm CUBE-N2.

A New Algorithm on Hypercube with N^2 PEs

In this section, a new algorithm CUBE-N2-2 on hypercube network with N^2 PEs is presented. This algorithm not only permutes image rows but also permutes image columns as well. The algorithm first computes the window column address and then computes the row address within a window column, both using Lemma 3. Since all the window columns should be held in local memory, M^2 local memory are needed. The algorithm CUBE-N2-2 is given below. A data movement example with $N=8, M=4$ is given in Fig. 2.

The procedure CUBE-N2-2(A,B,C,D,E) uses five registers from A to E. It is assumed that $A(i,j)$ is initialized to $G(i,j)$. Registers C, B, D and E are used to accumulate the partial results of C_{ij} during execution and all of them are initialized to 0. Match_along_column of line 13 computes partial results of C_{ij} at each image element $G(i,j)$ and stores them to register C,B,D and E respectively. Lines 14-17 rotate the contents of D and E one segment left and add them to B and C. Lines 18-19 rotate the content of B one segment up and add it to register C. Upon completion, the register C of each PE contains the final result of C_{ij} at image element $G(i,j)$.

In the procedure Match_along_column, variable t is the window column index, initialized to 0. The loop l of lines 3-12 executes the permute-multiply-accumulate along the column direction. The procedure Match_within_column(A,B,C,t) has four parameters. Register A holds image data, C and B are used to accumulate the partial results. Variable t is the window column index and it is different for different PE columns. Variable s is window row index, initialized to 0. Line 2 calculates the local address of the window element of each PE. The loop k executes permute-multiply-accumulate along row direction.

procedure CUBE-N2-2(A,B,C,D,E)

```
begin
 1    B(p):=0;
 2    C(p):=0;
 3    D(p):=0;
 4    E(p):=0;
 5    MAR(p):=0;
 6    for t:=0 to M-1 do
 7      for s:=0 to M-1 do
 8        CMAR:=α+t*M+s;
 9        M[MAR(p)]<=CM[CMAR];
10        MAR(p):=MAR(p)+1;
11      end for;
12    end for;
13    Match_along_column;
14    ROTATE(D,m,n-1,0);
15    C(p):=C(p)+D(p);
16    ROTATE(E,m,n-1,0);
17    B(p):=B(p)+E(p);
18    ROTATE(B,n+m,2n-1,0);
19    C(p):=C(p)+B(p);
end;
```

procedure Match_along_column

```
begin
 1    t:=0;
 2    Match_within_column(A,C,B,t);
```

```
3    for l:=0 to m-1 do begin
4      A(p^(l))<—A(p);
5      t:=(t+2^l) mod M  (p_l=0);
6      t:=(t-2^l) mod M  (p_l=1);
7      Match_within_column(A,C,B,t)  (p_l=0);
8      Match_within_column(A,D,E,t)  (p_l=1);
9      F:=false;
10     U:=l;
11     Gray_column(U,F);
12   end for;
end;
```

procedure Match_within_column(A,C,B,t)

```
begin
1    s:=0;
2    MAR(p):=t*M;
3    C(p):=C(p)+A(p)*M[MAR(p)];
4    for k:=n to n+m-1 do begin
5      A(p^(k))<—A(p);
6      s:=s+2^(k-n) mod M  (p_k=0);
7      s:=s-2^(k-n) mod M  (p_k=1);
8      MAR(p):=t*M+s;
9      C(p):=C(p)+A(p)*M[MAR(p)]  (p_k=0);
10     B(p):=B(p)+A(p)*M[MAR(p)]  (p_k=1);
11     F:=false;
12     U:=k-n;
13     Gray_row(U,F);
14   end for;
15   A(p^(n+m-1))<—A(p);
end;
```

procedure Gray_column(U,F)

```
begin
1    if U=0 then return else
2    begin
3      flag:=true;
4      Gray_column(U-1,flag);
5      A(p^(U-1))<—A(p);
6      if F then
7        t:=t+2^(U-1) mod M  (p_(U-1)=0);
8        t:=t-2^(U-1) mod M  (p_(U-1)=1);
9      else
10       t:=t-2^(U-1) mod M  (p_(U-1)=0);
11       t:=t+2^(U-1) mod M  (p_(U-1)=1);
12     end if;
13     Match_within_column(A,C,B,t)  (p_l=0);
14     Match_within_column(A,D,E,t)  (p_l=1);
15     flag:=false;
16     Gray_column(U-1,flag);
17   end if;
end;
```

procedure Gray_row(U,F)

```
begin
1    if U=0 then return else
2    begin
3      flag:=true;
4      Gray_row(U-1,flag)
5      A(p^(U+n-1))<—A(p);
6      if F then
7        s:=s+2^(U-1) mod M  (p_(U+n-1)=0);
8        s:=s-2^(U-1) mod M  (p_(U+n-1)=1);
9      else
10       s:=s-2^(U-1) mod M  (p_(U+n-1)=0);
11       s:=s+2^(U-1) mod M  (p_(U+n-1)=1);
12     end if;
13     MAR(p):=t*M+s;
14     C(p):=C(p)+A(p)*M[MAR(p)]  (p_k=0);
15     B(p):=B(p)+A(p)*M[MAR(p)]  (p_k=1);
16     flag:=false;
17     Gray_row(U-1,flag);
18   end if;
end;
```

In the algorithm CUBE-N2-2, there is no rotation in Match_along_column, nor in Match_within_column. The procedure Match_within_column takes M unit routes. In lines 7-8 of the loop l of Match_along_column, the procedure Match_within_column is called twice. Thus Match_along_column requires $2M^2$ unit routes. Three ROTATEs of lines 14-18 of CUB2-N2-2 needs $3(logN-logM)$ unit routes. Therefore the total inter-PE communication time is $2M^2+3(logN-logM)$, which is $O(M^2+logN)$.

Conclusion

Two template matching algorithms on hypercube SIMD computer with N^2 PEs are presented. The first algorithm improved the local address computation scheme over the algorithm given by Fang[1]. The time complexity is reduced to half. The second algorithm extends the permute-multiply-accumulate operations in Gray code sequence to both column and row direction. The inter-PE communication time is reduced to $O(M^2+logN)$. The template matching computation is intrinsically a shift(rotate)-multiplication in nature, while the hypercube network can only perform exchange. Therefore the rotation must be implemented by sequences of exchanges, which takes $O(logN)$ steps. In the second algorithm CUB-N2-2, we have the minimum rotation time, thus the $O(M^2+logN)$ inter-PE communication time is optimal in this sense. The trade-off is that the local memory of each PE is increased to M^2.

References

[1] Z. Fang, X. Li and L.M. Ni, "Parallel Algorithm for Image Template Matching on Hypercube SIMD Computers," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. PAMI-9, NO.6, Nov. 1987.

[2] V.K. Prasanna Kumar and V. Krishnan, "Efficient image template matching on SIMD hypercube machines", *Proceedings of International Conference on Parallel Processing*, 1987, pp. 765-771.

[3] G. Fox and S. Otto, "Algorithms for concurrent processors," *Physics Today*, pp. 50-59, May 1984.

[4] W.D. Hillis, *The connection Machine*. Cambridge, MA: M.I.T. Press, 1985.

[5] F. Preparata and J. Vuillemin, "The cube-connected cycles: A versatile network for parallel computation," in *Proc. IEEE Symp. Foundations of Computer Science*, 1979, pp. 140-147.

[6] A.P. Reeves and C.H. Francfort, "Data mapping and rotation functions for the Massively Parallel Processor," in *Proc. 1985 IEEE Comput. Soc. Workshop Computer Architecture for Pattern Analysis and Image Database Management*, 1985, pp. 412-419.

			k=n		k=n+1		u=1		k=n+2		u=1		u=2		u=1		recover
PE(0)	*0	0	*1	1	*3	3	*2	2	*6	6	*7	7	*5	5	*4	4	0
PE(1)	*0	1	7	0	*1	2	*2	3	*6	7	*5	6	*3	4	*4	5	1
PE(2)	*0	2	*1	3	7	1	6	0	*2	4	*3	5	*5	7	*4	6	2
PE(3)	*0	3	7	2	5	0	6	1	*2	5	*1	4	*3	6	*4	7	3
PE(4)	*0	4	*1	5	*3	7	*2	6	6	2	7	3	5	1	4	0	4
PE(5)	*0	5	7	4	*1	6	*2	7	6	3	5	2	3	0	4	1	5
PE(6)	*0	6	*1	7	7	5	6	4	2	0	3	1	5	3	4	2	6
PE(7)	*0	7	7	6	5	4	6	5	2	1	1	0	3	2	4	3	7
PE(8)	*0	8	*1	9	*3	11	*2	10	*6	14	*7	15	*5	13	*4	12	8
PE(9)	*0	9	7	8	*1	10	*2	11	*6	15	*5	14	*3	12	*4	13	9
PE(10)	*0	10	*1	11	7	9	6	8	*2	12	*3	13	*5	15	*4	14	10
PE(11)	*0	11	7	10	5	8	6	9	*2	13	*1	12	*3	14	*4	15	11
PE(12)	*0	12	*1	13	*3	15	*2	14	6	10	7	11	5	9	4	8	12
PE(13)	*0	13	7	12	*1	14	*2	15	6	11	5	10	3	8	4	9	13
PE(14)	*0	14	*1	15	7	13	6	12	2	8	3	9	5	11	4	10	14
PE(15)	*0	15	7	14	5	12	6	13	2	9	1	8	3	10	4	11	15
PE(16)	*0	16	*1	17	*3	19	*2	18	*6	22	*7	23	*5	21	*4	20	16
PE(17)	*0	17	7	16	*1	18	*2	19	*6	23	*5	22	*3	20	*4	21	17
PE(18)	*0	18	*1	19	7	17	6	16	*2	20	*3	21	*5	23	*4	22	18
PE(19)	*0	19	7	18	5	16	6	17	*2	21	*1	20	*3	22	*4	23	19
PE(20)	*0	20	*1	21	*3	23	*2	22	6	18	7	19	5	17	4	16	20
PE(21)	*0	21	7	20	*1	22	*2	23	6	19	5	18	3	16	4	17	21
PE(22)	*0	22	*1	23	7	21	6	20	2	16	3	17	5	19	4	18	22
PE(23)	*0	23	7	22	5	20	6	21	2	17	1	16	3	18	4	19	23
PE(24)	*0	24	*1	25	*3	27	*2	26	*6	30	*7	31	*5	29	*4	28	24
PE(25)	*0	25	7	24	*1	26	*2	27	*6	31	*5	30	*3	28	*4	29	25
PE(26)	*0	26	*1	27	7	25	6	24	*2	28	*3	29	*5	31	*4	30	26
PE(27)	*0	27	7	26	5	24	6	25	*2	29	*1	28	*3	30	*4	31	27
PE(28)	*0	28	*1	29	*3	31	*2	30	6	26	7	27	5	25	4	24	28
PE(29)	*0	29	7	28	*1	30	*2	31	6	27	5	26	3	24	4	25	29
PE(30)	*0	30	*1	31	7	29	6	28	2	24	3	25	5	27	4	26	30
PE(31)	*0	31	7	30	5	28	6	29	2	25	1	24	3	26	4	27	31

Fig. 1. A data movement example of the algorithm CUBE-N2-1 for N=32 and M=8.

	t	2		2		2		2		2		2		2		2	
s	j / i	0		1		2		3		4		5		6		7	
1	0	C	12	C	12	D	12	D	12	C	12	C	12	D	12	D	12
3	1	B	32	B	32	E	32	E	32	B	32	B	32	E	32	E	32
1	2	C	12	C	12	D	12	D	12	C	12	C	12	D	12	D	12
3	3	B	32	B	32	E	32	E	32	B	32	B	32	E	32	E	32
1	4	C	12	C	12	D	12	D	12	C	12	C	12	D	12	D	12
3	5	B	32	B	32	E	32	E	32	B	32	B	32	E	32	E	32
1	6	C	12	C	12	D	12	D	12	C	12	C	12	D	12	D	12
3	7	B	32	B	32	E	32	E	32	B	32	B	32	E	32	E	32

Fig. 2. One step of a data movement example of the algorithm CUBE-N2-2 for N=8 and M=4 is given here. In the table above, i, j is PE index, s, t is window index, A, B, C, D are registers to accumulate the result.

The table shows the combined result of the loop k of lines 4-14 for k=n=3 in Match_within_column for both p_l=0 and p_l=1 of line 13-14 in Gray_column of line 11 in Match_along_column.

LINEAR QUADTREE ALGORITHMS ON THE HYPERCUBE

S. K. Nandy, Rajat Moona†, and S. Rajagopalan‡
Center for Computer Aided Design
†Department of Computer Science and Automation
‡Department of Electrical Communication Engineering
Indian Institute of Science
Bangalore 560 012 INDIA

Abstract -- In this paper we provide an adaptation of the algorithms for neighbor finding (NFA) and boundary following (BFA) on the hypercube architecture. We encode quadtree on a linear array and analyse two schmes for embedding them on the hypercube. We observe that the communication delay in NFA using block embedded linear quadtree is low. Finally, we provide a parallel adaptation of the BFA on the hypercube based on block embedding and derive an expression for its speedup.

1. Introduction

Region representation of images play an important role in image processing, VLSI design and computer graphics[2,3,5-9]. Most commonly used algorithms deal with finding the boundary of a region, finding the neighbors of nodes in the tree, and performing operations on these trees[2,3,5-9]. With the advent of multiprocessor systems, the need for development of parallel and efficient algorithms is being increasingly realised. In this paper we attempt to provide parallel adaptations for the existing algorithms[2,3] on the hypercube architecture[4].

2. ENCODING QUADTREES IN A LINEAR ARRAY[6-9]

A node in a quadtree is a number called K_value where, K_value=$k_n k_{n-1} \ldots k_1$ and $0<=k_i<5$. Each digit k_i denotes the path to be taken at level i, (0=terminal 1=NW, 2=NE, 3=SW, 4=SE) to reach the node from root which is assigned a level 0. For an image and its corresponding quadtree shown in Fig. 2.1, node 10 has a K_value 0441. The K_value gives a unique path from root to any node in the tree and the most significant non-zero entry in K_value specifies its sontype.

A K_value yields to a unique decimal number $4^{n-1}*k_n + 4^{n-2}*k_{n-1} + \ldots 4*k_2 + k_1$ which may be used to index the linear array represention with dimension $[0..(4^{n+1}-1)*4/3]$ for a quadtree of depth n. Each element of this array contains 2 bits of information used to represent the colortype of the node as follows.

```
type colortype=(white,gray,not_used,black);
```

A white or black node is a terminal node and has no children. However, with regard to the array representation of quadtrees, the nodes allocated for their children are labelled "not_used". In the rest of the paper, we adopt the definitions and algorithms given in [2,3].

3. ALGORITHMS USING LINEAR QUADTREES[7-10]

The BFA and the related NFAs have been reproduced here in pseudo Pascal.

Algorithm NFA 3.1 EQUAL_ADJ_NEIGHBOR

```
function EQUAL_ADJ_NEIGHBOR (K:K_value;
    var K_ret:K_value; d:side):boolean;
{ finds an equal size node K_ret which is
adjacent along the d side of node K. Function
returns FALSE if such a node doesn't exist }
begin
  if K=root node then EQUAL_ADJ_NEIGHBOR:=FALSE
  else begin
    i:=max_level;
    while K[i]=0 do
      begin
        K_ret[i]:=0;i:=i-1
      end;
    repeat K_ret[i]:=mirror(K[i],d);i:=i-1
    until (adjacent(K[i+1],d)=FALSE) or (i<1);
    if adjacent(K[i+1],d)=TRUE then
      EQUAL_ADJ_NEIGHBOR:=FALSE
    else begin
      for i:=i downto 1 do K_ret[i]:=K[i];
      EQUAL_ADJ_NEIGHBOR:=TRUE
    end
  end
 end;
```

Algorithm NFA 3.2 CORNER_ADJ_NEIGHBOR

```
function CORNER_ADJ_NEIGHBOR (K:K_value;
  var K_ret:K_value; d:side; c:quadrant):boolean;
{finds a non-gray neighbor K_ret which is adjacent
along side d of K and is aligned to the other side
d' such that c=quadrant(d,d'). Function returns
FALSE if such a node doesn't exist }
begin
  if K=root node then CORNER_ADJ_NEIGHBOR:=FALSE
  else begin
    i:=max_level;
    while K[i]=0 do
      begin
        K_ret[i]:=0;i:=i-1
      end;
    repeat K_ret[i]:=0;i:=i-1
    until (adjacent(K[i+1],d)=FALSE) or (i<1);
    if (adjacent(K[i+1],d)=FALSE) then
      begin
        for j:=1 to i do K_ret[j]:=K[j];
        i:=i+1;
        while gray(K_ret) do
          begin
            if level(K)>level(K_ret) then
              K_ret[i]:=mirror(K[i],d)
            else K_ret[i]:=mirror(c,d);
            i:=i+1
          end;
        CORNER_ADJ_NEIGHBOR:=TRUE
      end
    else CORNER_ADJ_NEIGHBOR:=FALSE
  end
end;
```

Algorithm NFA 3.3 ALIGNED

```
function ALIGNED (K1,K2:K_value; d:side):boolean;
{Given two nodes K1 and K2 such that K2 is
adjacent along counterclockwise side of d of K1,
function returns TRUE if K1 and K2 are aligned
along d side of K1 else a FALSE. }
begin
  if (K1=root node) or (K2=root node) then
    ALIGNED:=FALSE
  else begin
    i:=1;
```

```
    while (K1[i]<>0) or (K2[i]<>0) do i:=i-1;
    if level(K1)=level(K2) then ALIGNED:=TRUE
    else if level(K1)<level(K2) then
      interchange(K1,K2);
    repeat
      if (K2[i]=0) and adjacent(K1[i],d) then
        i:=i-1
    until (i<1) or (K2[i]<>0) or
          not adjacent(K1[i],d);
    if i<1 then ALIGNED:=FALSE
    else if K2[i]=0 then ALIGNED:=FALSE
        else ALIGNED:=TRUE
  end
end;
```

Algorithm BFA 3.4 BFA

```
procedure BFA(K1,K2:K_value;d:direction);
{ Given two nodes K1(black) and K2(white) which
are adjacent along direction d of K1, the
algorithm traces the boundary of the region. The
algorithm outputs a sequence of strings giving the
direction and the length of the path to be
traversed along that direction }
```

The BFA algorithm is not reproduced here due to space limitation and can be found in [10]

4. EMBEDDING LINEAR QUADTREES ON THE HYPERCUBE

The hypercube is a MIMD machine having a d-dimensional cube interconnection topology with one processor module at each node of the cube. Each edge in the cube is constituted by a physical communication link[4]. A hypercube of dimension four is considered for embedding of quadtrees. The address of a PE is denoted by $A = (a_3a_2a_1a_0)_2$ and the K_value of a node in the quadtree is denoted by $k_nk_{n-1}...k_1$ where $0 <= k_i < 5$. The quadtree can be embedded in the hypercube in two ways viz. Block Embedding and Complete Embedding. Details of the embedding are given below.

4.1 Block Embedding

In this case, sub-trees of the quadtree are assigned to different PE's in the hypercube as shown in Fig. 4.1. The address of a PE in which a node in the quadtree would lie is given by the K_1 and K_2 digits of the node's K_value.

4.2 Complete Embedding

All the quadtree nodes are distributed to the PE's in the hypercube. The hashing procedure to find the PE address (pe_addr) of a quadtree node, is as follows :

```
initial pe_addr ← (0000)2.
for i := 1 to max_level do
  if ki<>0 then complement (ki)th bit of pe_addr;
```

5. COMPLEXITY ANALYSIS OF NFA

Our adaptations of NFAs involves manipulation of K_value. The K_values of interest are initial and final ones and execution time depends on the communications delay between PEs in which the nodes reside. We assume a random image comprising many uniformly distributed connected regions such that a node is equally likely to appear in any position and level in a quadtree and the perimeters of regions have low standard deviation.

Consider a block embedded quadtree. With regard to the NFA Equal_adj_neighbor the possible communication delay is between the neighbors which fall on the boundary such that two adjacent neighbors are on two different processors.

For 16 processor allocation, there will be 6 boundaries of total 2^n leafnodes each (see Fig. 4.1a). The adjacent neighbor pairs of size one leafnode falling on the boundary will be $6 * 2^n$. Similarly for pairs of size 4^a falling on the boundary will be $6 * 2^{n-a}$. Total number of neighbor pairs falling on the boundary will therefore, be

$$6 * (2^n + 2^{n-1} + ... + 1) = 6 * (2^{n+1} - 1).$$

However, total number of neighbor pairs in the image [3] is

$$\sum_{i=0}^{n-1} 2^{n-i} * (2^{n-i} - 1)$$

Communication requirement (C) on the average will therefore be,

$$\frac{6 * (2^{n+1} - 1)}{\sum_{i=0}^{n-1} 2^{n-i} * (2^{n-i} - 1)}$$

or, $C = 9 * 2^{-n}$ for n very large.

Assuming an average communication length of two hops between processors, the total communication delay can be expressed as

$(C*2)/16 = (9/8)*2^{-n}$ for large n.

Similarly, it can be shown that all the other NFAs will also have the same communication delay, for large n.

Whereas for a completely embedded quadtree, it is difficult to derive similar expressions for the average number of hypercube nodes visited. Programs were run for different NFAs and the results are presented in Table 5.1.

TABLE 5.1

Algorithm	Average number of hops
Equal_adjacent_neighbor	2.0000
GT_Equal_adj_neighbor	2.0000
Equal_corner_neighbor	1.8450
Corner_Adj_neighbor	2.0000
Aligned	1.1200

6. BFA ON THE HYPERCUBE

Prior to the description of BFA on the hypercube, we lay down the following assumptions.

1. An image comprises $\Re$ regions which are uniformly distributed over the entire area A of the image. The area A is defined as number of pixel in the image-$2^n \times 2^n$ for a n-level tree.
2. The density of the image D is specified as ratio of black area to total area. Total area occupied by all regions, will therefore, be (D*A).
3. The number of regions $\Re$ in the image is very large. ($\Re$ >> number of PEs)
4. The image has been block embedded

```
procedure Parallel_BFA;
  var start1,start2:array [1..ℜ] of K_value;
  cobegin
    for i:=1 to ℜ do {for all regions in image}
      begin
        locate start1[i] and start2[i];
        BFA(start1[i],start2[i],N)
      end
  coend;
```

6.1 Speedup

For this algorithm, the time required to trace one boundary of perimeter $\wp$ is $O(\wp)$. The computation time on the parallel machine will therefore be $O(\mathcal{P}/16)$, $\mathcal{P}$ being the total perimeter. The maximum communication delay will occur only when boundary of the region coincides with the processor boundary, and all black and white node pairs are at leafnode level. In such a case, there can be a maximum $6 * 2^n$ communications. If the communication time is t_c and execution time is t_e then total time required on the parallel system will be, $(\mathcal{P}/16) * t_e + (6 * 2^n) * t_c$
and the speedup will be,

$$\frac{(\mathcal{P} * t_e)}{((\mathcal{P}/16) * t_e + (6 * 2^n) * t_c)}$$

Assuming square regions, a maximum of $\Re$ regions, and average area per region given by $(D*A/\Re)$, the total perimeter $\mathcal{P}$ of all the regions in the image can be expressed as $4*(D*A*\Re)^{\frac{1}{2}}$

Further, $A=2^n*2^n$, and with regards to VLSI Layouts[1], $\Re \propto A$, or $\Re=\mu*A$. Hence speedup is

$$\frac{16}{1 + 24 * 2^{-n} * (D * \mu A)^{-\frac{1}{2}} * (t_c / t_e)}$$

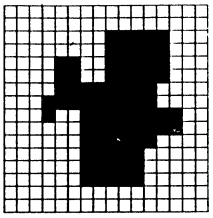
$\simeq 16$ (for large n).

7. CONCLUSIONS

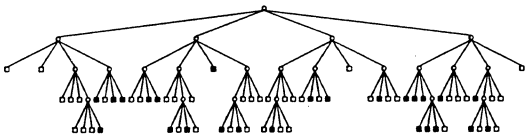
From the simplicity and efficiency of implementation, it can be concluded that our approach to parallel processing can be readily extended to CAD tools, viz. design rule checker[5], circuit extractor, routers etc. that are designed based on NFAs and BFA.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the contribution of Prof. V. Rajaraman for initiating this research. Help rendered by YVSP Rao towards manuscript preparation is also acknowledged.

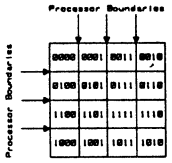


(a) Image (16 X 16)

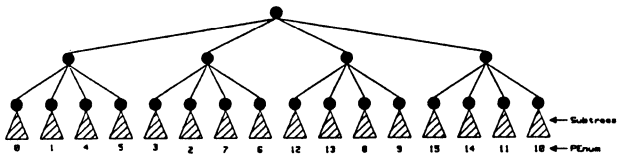


(b) Quadtree for the Image

FIG. 2.1 An Image and its corresponding Quadtree



(a) 16 Blocks of an Image and their PE assignments



(b) Quadtree corresponding to the Image in (a)

FIG. 4.1 An Image and its partitions

REFERENCES

[1] Bentley, J L, Haken, D and Hon, R W "Statistics on VLSI Designs", Department of Computer Science, CMU, CS-80-111 (April 1980).

[2] Dyer, R, Rosenfeld, A and Samet H "Region Representation: Boundary Codes from Quadtrees", CACM, Vol. 23, No. 3, March 1980, pp. 171-179.

[3] Samet, H "Neighbor Finding Techniques for Images Represented by Quadtrees", CGIP, Vol. 18, 1982, pp. 37-57.

[4] Tuazon, J, Peterson, J, Pniel, M and Leberman, M "Caltech/JPL Mark II Hypercube Concurrent Processor", Proc. 1985 Parallel Processing Conference, pp. 663-673.

[5] Nandy, S K and Patnaik, L M "Linear Time Geometrical Design Rule Checker Based on Quadtree Representation of VLSI Mask Layouts", Computer-Aided Design, Vol. 18, No. 7, September 1986, pp. 380-388.

[6] Samet, H "The quadtree and related hierarchical data structures", ACM Comput. Surv., Vol. 16, June 1984, pp. 187-260.

[7] Samet, H "Region Representation : Quadtrees from Binary Arrays",CGIP,Vol. 13, 1980, pp.88-93.

[8] Gautier, N K, et.al. "Space and Time efficiency of the Forest of Quadtrees representation", Jr. on Image and Vision Computing, Vol. 3, No. 2, May 1985.

[9] Ravindran, S and Manohar, M "An Algorithm for converting Forest of Quadtrees to Binary Array", Jr. on Image and Vision Computing, Nov. 1987.

[10] Nandy, S K, Rajat Moona and S. Rajgopalan "Primitive Quadtree Algorithms based on K_values", Tech. Report, Center for CAD, I.I.Sc., Jan 88.

OPTIMISING A RECONFIGURABLE MIMD TRANSPUTER MACHINE FOR LINE-OF-SIGHT CALCULATIONS ON LARGE DIGITAL MAPS

J S Ward and J B G Roberts
Royal Signals and Radar Establishment
Great Malvern
UK

Abstract

A major demand for computing power in simulators and in systems for evaluating and optimally siting sensors is the tracing lines of sight to determine intervisibilities and calculating 'depths of shadow' between arbitrary points above 3-D landscapes represented by large, high resolution digital maps.

We show how a MIMD parallel machine with a switched interprocessor connection topology can be configured and programmed for these problems; the machine topology and allocation of tasks being rearranged to achieve spectacular performances on maps of differing resolution and for various graphical display requirements. The machine architecture is a modular network of Transputers capable of making arbitrary interconnections between as many as 1000 processors.

1. Philosophy of Machine Architecture

Compact computers of very high potential performance can now be assembled rather easily from the present generation of single chip processing elements (PE). Realising a high proportion of the potential performance may be difficult however, particularly with irregular or data dependent algorithms. Even systematic problems need careful allocation of the tasks between processors to avoid serious under utilisation of the machine, and the communication strategy between processors assumes a vital role in preventing wasteful bottlenecks. However VLSI allows us to implement complex interconnection networks between processors to alleviate this problem.

With several partners in an ESPRIT consortium [1], we are developing multi-transputer [2] processor machines based on the Communicating Sequential Processes (CSP) [3]/OCCAM model of computing [4]. A machine built on this model is characterised by distributed memory and point to point communication links. This model allows great hardware and software flexibility and versatility. For example it allows MIMD processing, modular construction of machines, fixed or floating point PEs and individual processor memory sizes, all augmented by the extra freedom to reconfigure the machine, dynamically if desired, by software control of the switches.

This architecture, known as RTP (Reconfigurable Transputer Processor) differs crucially from machines based on shared memory or shared communications buses, both of which entail heavy hardware costs in achieving high memory and communication bandwidths [5]. Like 'hypercube' machines, it is a coarse grain MIMD machine using 16 or 32-bit transputer PE's which execute distinct programs. However the RTP topology consists of one-to-one channels set up by switches rather than a fixed low-diameter network through which data messages are routed.

Any connection topology possible with the four bidirectional links available on each transputer can be realised for as many as 1000 PEs, including the use of multiple links between transputers. The RTP philosophy being to employ hardware techniques to match the machine to the problem. The network is set up automatically from the 'harness' (known as the wiring diagram) which is always written to describe the allocation of processes to processors. In cases where a problem involves several phases such as I/O, data sorting, scatter/gather operations etc., the configuration may optimised for each and set accordingly. There is no hardware obstacle to altering the switch connections dynamically, but operating system tools are not yet available to decide routings and to prevent the interruption of current transfers.

2. Hardware

Although the RTP is still in development, several prototypes have been built and programmed in order to verify ideas. The standard hardware module for RTP machines is a 'SuperNode' built around 16 'worker' PE's which are normally Floating Point transputers (T800s). This transputer was developed as part of the ESPRIT project and is capable of calculation at a sustained rate of between 1 and 2 MFlop/s. The worker transputers in a supernode are connected to each other, to a memory server, a hard disc unit and to other supernodes through two 72 x 72 crossbar link switches, controlled by a 'control' transputer. These all communicate via standard Inmos links but the control transputers have an independent low bandwidth bus connected to all the transputers in the machine to

allow diagnostics information to be extracted without disturbing the state of the RTP links. The 72 x 72 crossbar switch has been implemented using two 15000-gate CMOS gate arrays.

In the prototype RTP machines each worker PE has 256 KBytes of static RAM in addition to the T800's 4 KBytes of internal memory. In later versions this will be optionally extended to 4 MBytes of dynamic RAM each.

Larger machines are built up by interconnecting 'supernodes' together using a further layer of switches. Sufficient links are available for the internode switches to allow the switching network to make any possible transputer graph in a 'rearrangeable' manner [6].

3. Terrain Visibility and Shadowing

A demanding application we have programmed on the RTP is the calculation of lines of light using digital maps. The programs calculate the ground that is visible to an observer at a given height and position over the map and also the depth of concealment (or 'shadow' if we think of the observer as a light source).

Computation on digital maps presents an interesting combination of benchmark tests. These computations are numerically intensive and can be done with either integer or floating point arithmetic. The maps involved can be large and the distribution of this database amongst many processors to allow multiple access to the database is in itself an interesting problem. Furthermore these applications are considerably enhanced by good graphical displays. Manipulating images to provide rapid response times is a further computational load and exercises the machines I/O capability.

A major factor in the performance of the visible area calculations is the number of rays traced. Computing N^2 rays for an N x N map is highly redundant and a reduction by a factor of order N is possible. A profile of the ground along each ray is formed and the the look elevation angle to each point on it. The program then decides if each point along the the profile is visible or not and colours the display accordingly.

Two versions of the programs developed are described in sections 3.1 and 3.2 below.

3.1 Small Map

When implementing this algorithm for a 256 x 256 byte map of the Isle of Wight, the future need to use much larger maps, which because of their size could not be stored in the memory of a single processor, was allowed for. With this in mind a pipelined solution with different processors doing different tasks was adopted. Four stages of the algorithm were identified:

i) Calculate the (x,y) coordinates along each ray using Bressenham's Algorithm [7]. The change in x and/or y per step is at most 1.

ii) Using the (x,y) coordinates of a ray fill in the heights of the terrain along the ray creating a profile.

iii) Starting from the observer end of the ray decide if each point along the ray can be seen. This is done by comparing the tangent of the angle between the observer and the height at that point along the ray with the maximum tangent calculated so far. If the tangent at that point is less than the maximum tangent so far that point along the ray is not visible, if greater it is visible.

iv) From the result of stage (iii) decide what colour to display on the graphics screen and where, for each point along the ray. The colour convention adopted had the underlying terrain heights displayed as a red/green (or brown) colour bar and areas not visible coloured black or as a grey scale in the depth of shadow computations.

For the small map implementation (256 x 256 bytes) the first two stages above were implemented in one processor. The network of processors used is shown in figure 1. There are 4 parallel pipelines which work on 8 adjacent rays - the rays are chosen to be adjacent to ensure good load balancing. The first processor in each pipeline performs stages (i) and (ii) above, each pipeline then splits into two, each sup-pipeline performing the shadowing computation (iii). The two sub-pipelines in each pipeline then come together in the the fourth processor which does the colour computation (iv). Several other networks were used during program development but this one was chosen as it had gave the most balanced load amongst the processors. The load on each of the processors by the use of an 'efficiency monitor' which shows how busy each of the processors was in the network.

3.1.1 Performance

Table 1 gives the execution times for a two metre high observer both at the centre of the map and in one corner.

The depth of shadow calculation shown in the table is a calculation of how high a target would have to be to seen at that point on the map if the ground is not directly visible from the observer. The

shadow is displayed as a graduated grey scale that saturates black at a user definable depth.

In its final form the program ran in the times shown in table 1 on 16 T414 17 MHz 'worker' transputers, a T414-15 B007 graphics board with 0.5 MByte Video RAM and 0.5 MByte DRAM and a B004. The B004 is a transputer card with 2 MBytes of DRAM that acts as a host in an IBM-AT.

The use of an efficiency monitor showed that for many cases the rate of execution of the program was limited by the speed at which the graphics processor could display the results. In this case there is nothing to be gained from using more 'worker' processors. Cases where the graphics processor load dominates are characterised by similar times for integer and floating point calculations.

With sub-second calculation times the use of a tracker-ball or mouse to move the position of the observer gives impressive interactive displays.

3.2 Larger Maps

Comparison of the 256 x 256 byte map of the Isle of Wight with an Ordnance Survey (paper) map revealed several inaccuracies and a more detailed digital map was obtained. This map was 1024 x 512 points of 16-bits, each spaced on a 50m grid as opposed to the 100m grid for the earlier map.

As the B007 graphics board drives a 512 x 512 screen, it was decided to limit the shadowing calculation to a 512 x 512 section of the map at any one time, the rest of the map being held in RAM on the B004. With this limitation the storage required by the used area of the map was 0.5 MBytes. As each of the processors in the network has 256 KBytes of RAM, at least three processors have to be dedicated to the storage of each copy of the map. Splitting the map up amongst processors then allows parallel access to parts of the map, speeding up the height lookup process.

The final network of processors used is shown in figure 2. This has been optimised to approximately equalise the load across the processors for shadowing calculations using integer arithmetic on T414s. The loading on the processors was measured using the efficiency monitor. Figure 2 should be viewed as two pipelines each of which split into two. The four sub-pipes process adjacent rays to balance the load between pipelines. Various configurations were readily explored because the machine configuration is automatically set up from the normal Occam description of the program.

The main differences between the 256 x 256 byte map and 512 x 512 16-bit map are discussed in the following sections

3.2.1 Map Look Up

As noted above, the map storage is split amongst three processors. The x.y.list process calculates where the each ray enters or leaves each of the three map sections and passes these as parameters to the three look-up processors. This was found to be the most efficient way of organising the lookup.

3.2.2 Performance

Table 2 summaries the performance achieved on the larger map with the same processors described in section 3.1.1 above. Performance could be improved by using faster processors (20 or 30MHz parts) and faster links with overlapped acknowledge. The processors used for the timings below link used 10 Mbit/s links.

3.3.4 Displays

The two photographs illustrate some of the points mentioned above, both photographs how part of the larger map.

Photograph 1 shows the area visible to a 2m high observer on St Catherines Down at the cursor position. The version of the program used to calculate this required 5.89 sec and used integer arithmetic throughout. In the bottom left hand corner of all these photographs is a display of the utilisations of all the processors in the network during the calculation. The level of the white line indicates 100% utilisation. The first 8 columns refer to the top pipeline in figure 2, the next 8 the lower pipeline and the most righthand column is the B007 graphics processor. The order of display within the pipelines is:

(x.y.list, lookup 0, look up 1, look up 2, top shadow, top colour, lower shadow and lower colour.)

Notice that no processor is 100% utilised. This shows that either the processors are waiting for inputs whilst doing no useful work or that the link bandwidth is saturated. The network was configured to maximise processor usage on visible area integer calculations.

The second photograph is a display of the depth of shadow for an observer at the same point. The depth of shadow is displayed as a grey scale which saturates black when the the depth of shadow exceeds 100m. This calculation has been done using floating point arithmetic and took 17.35 sec. Notice here that the 4 processors performing the the shadowing computations are 100% used and limit the computation. The network was not optimised for this mode of calculation.

T800 floating point transputers execute the point point version of the code faster than the integer version of the code. This is because the floating point code is much simpler than the integer version as it applies no range checking or scaling of numbers.

4. Summary and Conclusions

This paper has described the performance of a small (1 cubic ft) prototype node for large MIMD machines built from transputers. The key features of this machine are distributed memory and reconfigurable point-to-point communications.

The Intervisibility calculations demonstrate the power and versatility of the machine. The different processor configurations were easy to generate and program using the switches. This is the great strength and flexibility of the machine. Without the switches the tendency is to set up one network and use that regardless of how inefficient it is.

The processing power of the machine can be seen from the execution times quoted above. The speed of execution of the visibility calculations allows the effects of arithmetic precision, interpolation etc to be quickly and easily investigated. The speed of execution allows searches for optimum sensor sites a realistic proposition. These investigation are rarely done on conventional machines due to excessive CPU time requirements.

6. Acknowledgement

This work has been partly funded by the Advanced Information Processing program of ESPRIT.

7. References

1. Harp J G,
Phase 2 of the Reconfigurable Transputer Project - P1085,
ESPRIT '87: Achievements and Impact, pp 583 - 591
North Holland 1987.

2. INMOS Ltd,
Transputer Reference Manual,
Prentice-Hall, London, 1988.

3. Hoare C A R
Communicating Sequential Processes,
Prentice-Hall, London, 1985.

4. INMOS Ltd,
Occam2 Reference Manual,
Prentice-Hall, London, 1988.

5. Hwang K,
Advanced Parallel Processing with Supercomputer Architectures,
Proc IEEE, Vol 75, Oct 87,
pp1348-1379.

6. Nicole D A, LLoyd E K and Ward J S,
Switching Networks for transputer Links,
8th Occam Users group Meeting,
Sheffield, UK, March 88.

7. Bressenham J E,
Algorithm for Computer Control of a digital plotter,
IBM System Journal, Vol 4, No 1,
1955.

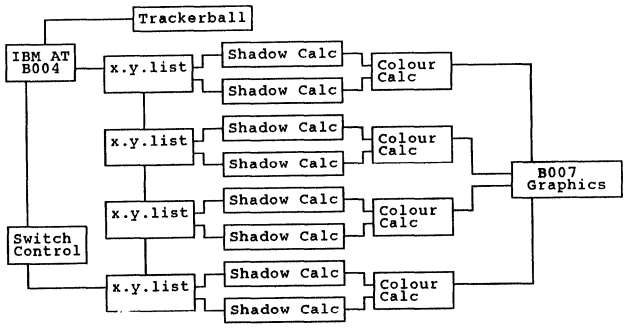


Figure 1
Small Map 256 x 256 bytes,
Rays from Edge of picture.

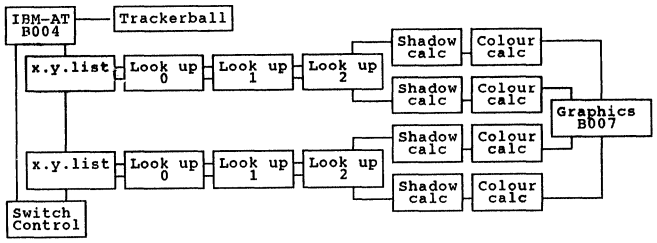


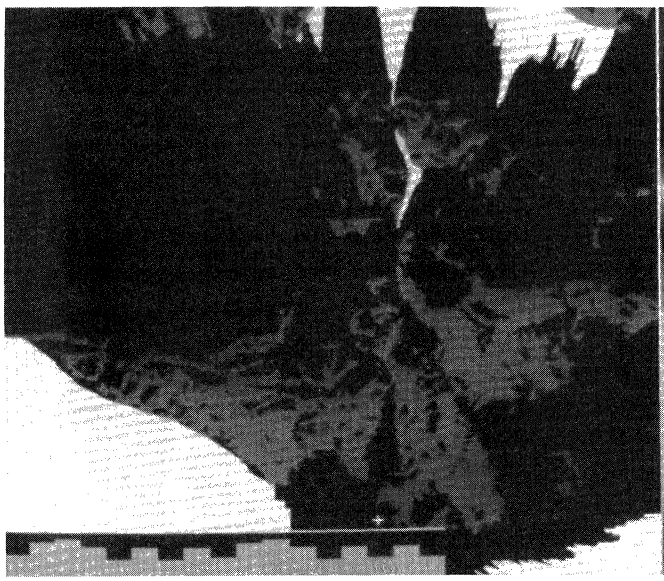
Figure 2
Large Map Processor Layout

Observer Position x	y	Calculation type	Arithmetic type: Floating pt	Arithmetic type: Integer
128	128	Shadow	1.13	0.51
0	0	Shadow	1.73	1.74
128	128	Depth of shadow	2.00	2.00
0	0	Depth of shadow	2.85	2.84

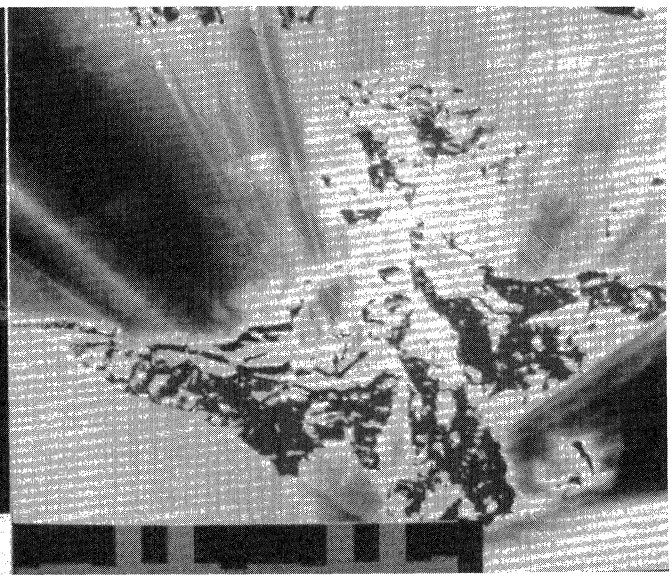
Table 1 Summary of execution times in seconds for 256 x 256 map
These timings are largely limited by the speed of the graphics processor

Relative Position x	y	Calculation type	Arithmetic type: Floating pt	Arithmetic type: Integer
256	256	Shadow	8.93	3.93
0	0	Shadow	13.00	5.68
256	256	Depth of shadow	17.27	5.15
0	0	Depth of shadow	18.74	9.10

Table 2 Summary of execution times in seconds with 512 x 512 map



Photograph 1
Shadowing Calculation with Integer Arithmetic



Photograph 2
Depth of Shadow Calculation with Floating Point Arithmetic

IMPLEMENTATION AND ANALYSIS OF A NAVIER-STOKES ALGORITHM ON PARALLEL COMPUTERS

Raad A. Fatoohi

Sterling Software, Inc.
Palo Alto, CA 94303

Chester E. Grosch

Old Dominion University
Norfolk, VA 23529

Abstract -- This paper presents the results of the implementation of a Navier-Stokes algorithm on three parallel/vector computers. The object of this research is to determine how well, or poorly, a single numerical algorithm would map onto three different architectures. The algorithm is a compact difference scheme for the solution of the incompressible, two-dimensional, time dependent Navier-Stokes equations. The computers were chosen so as to encompass a variety of architectures. They are: the MPP, an SIMD machine with 16K bit serial processors; Flex/32, an MIMD machine with 20 processors; and Cray/2. The implementation of the algorithm is discussed in relation to these architectures and measures of the performance on each machine are given. Simple performance models are used to describe the performance. These models highlight the bottlenecks and limiting factors for this algorithm on these architectures. Finally conclusions are presented.

I. Introduction

Over the past few years a significant number of parallel computers have been built. Some of these have been one of a kind research engines, others are offered commercially. Both SIMD and MIMD architectures are included. A major problem now facing the computing community is to understand how to use these various machines most effectively. Theoretical studies of this question are valuable. However, we believe that comparative studies, wherein the same algorithm is implemented on a number of different architectures, provide an equally valid way to this understanding. These studies, carried out for a wide variety of algorithms and architectures, can highlight those features of the architectures and algorithms which make them suitable for high performance parallel processing. They can exhibit the detailed features of an architecture and/or algorithm which can be bottlenecks and which may be overlooked in theoretical studies. The success of this approach depends on choosing "significant" algorithms for implementation and carrying out the implementation over a wide spectrum of architectures. If the algorithm is trivial or embarrassingly parallel it will fit any architecture very well. We need to use algorithms which solve hard problems which are attacked in the scientific and engineering community.

In this paper we present the results of the implementation of an algorithm for the numerical solution of the Navier-Stokes equations, a set of nonlinear partial differential equations. In detail, the algorithm is a compact difference scheme for the numerical solution of the incompressible, two dimensional, time dependent Navier-Stokes equations. The implementation of the algorithm requires the setting of initial conditions, boundary conditions at each time step, time stepping the field, and checking for convergence at each time step. Equally important to the choice of algorithm is the choice of parallel computers. We have chosen to work on a set of machines which encompass a variety of architectures. They are: the MPP, an SIMD machine with 16K bit serial processors; Flex/32, an MIMD machine with 20 processors; and Cray/2. The basic comparison which we make is among SIMD instruction parallelism on the MPP, MIMD process parallelism on the Flex/32, and vectorization of a serial code on the Cray/2. The implementation is discussed in relation to these architectures and measures of the performance of the algorithm on each machine are given. In order to understand the performances on the various machines simple performance models are developed to describe how this algorithm, and others, behave on these computers. These models highlight the bottlenecks and limiting factors for algorithms of this class on these architectures. In the last section of this paper we present a number of conclusions.

II. The numerical algorithm

The Navier-Stokes equations for the two-dimensional, time dependent flow of a viscous incompressible fluid may be written, in dimensionless variables, as:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \tag{2.1}$$

$$\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = \zeta, \tag{2.2}$$

$$\frac{\partial \zeta}{\partial t} + \frac{\partial}{\partial x}(u\,\zeta) + \frac{\partial}{\partial y}(v\,\zeta) = \frac{1}{\mathrm{Re}}\nabla^2 \zeta, \tag{2.3}$$

where $\vec{u} = (u,v)$ is the velocity, ζ is the vorticity and Re is the Reynolds number.

The numerical algorithm used to solve equations (2.1) to (2.3) was first described by Gatski, et al. [6]. This algorithm is based on the compact differencing schemes which require the use of only the values of the dependent variables in and on the boundaries of a single computational cell. Grosch [8] adapted the Navier-Stokes code to ICL-DAP. Fatoohi and Grosch [3] solved equations (2.1) and (2.2), the Cauchy-Riemann equations, on parallel computers. The algorithm is briefly described here.

Consider equations (2.1) to (2.3) in the square domain $0 \le x \le 1$, $0 \le y \le 1$ with the boundary conditions $u = 1$ and $v = 0$ at $y = 1$ and $u = v = 0$ elsewhere. Subdivide the domain into rectangular cells. The center of a cell is at $(i+1/2,j+1/2)$. Apply the centered difference operator to

equations (2.1) to (2.2), to get

$$\delta_x U_{i+1/2,j+1/2} + \delta_y V_{i+1/2,j+1/2} = 0, \qquad (2.4)$$

$$\delta_x V_{i+1/2,j+1/2} - \delta_y U_{i+1/2,j+1/2} = \zeta_{i+1/2,j+1/2}. \qquad (2.5)$$

The adaptation of this algorithm to different parallel architectures can be simplified by the introduction of box variables to represent $\vec{U}$. The box variables, $\vec{P}$, are defined at the corners of the cells so that the average of two adjacent $\vec{P}$'s is equal to the $\vec{U}$ on the included side. The set of difference equations and boundary conditions in terms of the box variables are solved using a cell relaxation scheme which is equivalent to an SOR method [6], [8].

The compact difference approximation to equation (2.3) results in an implicit set of equations which are solved by an ADI method [4]. This method consists of two half steps to advance the solution one full step in time. Let Δt be the full time step and apply finite difference operators to equation (2.3), to get

$$\beta_{i,j}^{(x)}\, \zeta_{i-1,j}^{n+1/2} - (1 + 2\alpha_j^{(x)})\, \zeta_{i,j}^{n+1/2} + \gamma_{i,j}^{(x)}\, \zeta_{i+1,j}^{n+1/2} = F_{i,j}, \qquad (2.6)$$

$$\beta_{i,j}^{(y)}\, \zeta_{i,j-1}^{n+1} - (1 + 2\alpha_i^{(y)})\, \zeta_{i,j}^{n+1} + \gamma_{i,j}^{(y)}\, \zeta_{i,j+1}^{n+1} = G_{i,j}, \qquad (2.7)$$

where

$$F_{i,j} = -\beta_{i,j}^{(y)}\, \zeta_{i,j-1}^{n} - (1 - 2\alpha_i^{(y)})\, \zeta_{i,j}^{n} - \gamma_{i,j}^{(y)}\, \zeta_{i,j+1}^{n},$$

$$G_{i,j} = -\beta_{i,j}^{(x)}\, \zeta_{i-1,j}^{n+1/2} - (1 - 2\alpha_j^{(x)})\, \zeta_{i,j}^{n+1/2} - \gamma_{i,j}^{(x)}\, \zeta_{i+1,j}^{n+1/2},$$

$$\alpha_j^{(x)} = \Delta t \,/\, 2(\Delta x)_j^2\, \mathrm{Re}, \qquad \alpha_i^{(y)} = \Delta t \,/\, 2(\Delta y)_i^2\, \mathrm{Re},$$

$$\beta_{i,j}^{(x)} = \alpha_j^{(x)} + \Delta t\, U_{i-1,j}\,/4(\Delta x)_j, \quad \beta_{i,j}^{(y)} = \alpha_i^{(y)} + \Delta t\, V_{i,j-1}/4(\Delta y)_i,$$

$$\gamma_{i,j}^{(x)} = \alpha_j^{(x)} - \Delta t\, U_{i+1,j}\,/4(\Delta x)_j, \quad \gamma_{i,j}^{(y)} = \alpha_i^{(y)} - \Delta t\, V_{i,j+1}/4(\Delta y)_i.$$

The velocity field is not defined at the corners of the cells in this scheme; however, it can be computed using the box variables at the two immediate interior neighbors along the vertical and horizontal lines. Equation (2.6) represents a set of independent tridiagonal systems (one for each vertical line of the domain). Similarly, equation (2.7) represents a set of independent tridiagonal systems (one for each horizontal line of the domain). The ADI method for equation (2.3) is applied to all interior points of the domain. The values of ζ on the boundaries are computed using equation (2.2), see [2] for details.

The key to the adaptation of the relaxation scheme for solving equations (2.1) and (2.2) to parallel computers is the realization that each $\vec{P}$ is updated four times in a sequential sweep over the array of cells. This fact is utilized by using reordering to achieve parallelism. The computational cells are divided into four sets of disjoint cells so that the cells of each set can be processed in parallel [3]. It is therefore clear that the cell iteration for the box variables is a four "color" scheme. Thus four steps are necessary for a complete relaxation sweep.

The main issue in implementing the ADI method for equation (2.3) on parallel computers is choosing an efficient algorithm for the solution of tridiagonal systems. Two algorithms are considered here: Gaussian elimination and cyclic elimination, [4], [9]. The Gaussian elimination algorithm is based on an LU decomposition of the tridiagonal matrix. This algorithm is inherently serial because of the recurrence relations in both stages of the algorithm. However, if one is faced with solving a *set* of independent tridiagonal systems, then Gaussian elimination will be the best algorithm to use on a parallel computer; all systems of the set are solved in parallel. The cyclic elimination algorithm is a variant of the cyclic reduction algorithm [9] applying the reduction procedure to all of the equations and eliminating the back substitution phase of the algorithm. Cyclic elimination is most suitable for machines with a large natural parallelism, like the MPP.

The solution procedure for the Navier-Stokes equations can be summerized as follows:

(1) Assume that ζ is zero everywhere at $t = 0$. The variables and boundary values are initialized.

(2) The vorticity at the corners of the domain, undefined in this scheme, is approximated using the values of its neighboring points. The values of $\zeta_{i+1/2,j+1/2}$ are computed using the values of ζ at the corners of the cells.

(3) The relaxation process is implemented for each "color", i.e. four times in order to complete a sweep. The maximum residual is computed and tested against the convergence tolerance. The whole process is repeated until the iteration converges.

(4) The coefficients $\alpha_j^{(x)}$, $\alpha_i^{(y)}$, $\beta_{i,j}^{(x)}$, $\beta_{i,j}^{(y)}$, $\gamma_{i,j}^{(x)}$, $\gamma_{i,j}^{(y)}$ for both passes of the ADI method are computed.

(5) The values of ζ on the boundaries are computed.

(6) The tridiagonal equations distributed over columns, equation (2.6), are solved.

(7) The tridiagonal equations distributed over rows, equation (2.7), are solved.

These steps were implemented using the following subprograms: *setbc,* step (1); *zcntr,* step (2); *relaxd,* step (3); *cof,* step (4); *zbc,* step (5); *triied,* step (6); and *trijed,* step (7). The repetition of steps (2) through (7) yields the values of the velocity and vorticity at any later time.

III. Implementation on the MPP

The Massively Parallel Processor (MPP) is a large-scale SIMD processor built by Goodyear Aerospace Co. for NASA Goddard Space Flight Center [1]. The MPP is a back-end processor for a VAX-11/780 host, which supports its program development and I/O needs.

The MPP's high level language is MPP Pascal [7]. It is a machine-dependent language which has evolved from Parallel Pascal, an extended version of Pascal with a syntax for specifying array operations. These extensions provide a parallel array data type and operations on these arrays.

The Navier-Stokes algorithm, described in section II, was implemented on the MPP using 127×127 cells (128×128 grid points). The computational cells are mapped onto the array so that each corner of a cell corresponds to a processor. The seven subprograms of this algorithm (see section II) were written in MPP Pascal.

These subprograms were executed entirely on the MPP; only I/O routines were run on the VAX.

The relaxation process, subprogram *relaxd,* was implemented on the array using the four color relaxation scheme [3]. The ADI method, subprograms *triied* and *trijed,* was implemented by solving two sets of 128 tridiagonal systems using the cyclic elimination algorithm. This is done in parallel on the array with a tridiagonal system of 128 equations being solved on each row or column.

One of the problems in solving Navier-Stokes equations on the MPP is the size of the PE memory. The relaxation subprogram uses almost all of the 1024 bit PE memory; 22 parallel arrays of floating point numbers, all but 5 of which are temporary. Although the staging memory can be used as a backup memory, this causes an I/O overhead and reduces the efficiency. This problem was solved by declaring all parallel arrays as global variables and using them in procedures for more than one purpose. Beside this memory problem, there are problems in using MPP Pascal to perform vector operations and to extract elements of parallel arrays. Operations on vectors are performed by expanding them to matrices and performing matrix operations; thus the processing rate is 1/128 of that for matrix operations. MPP Pascal does not permit extracting an element of a parallel array. This means that scalar operations involving elements of parallel arrays need to be expanded to matrix operations or performed on the VAX.

The relaxation subprogram is quite efficient; almost all of the operations are matrix operations, no vector and only two scalar operations per iteration, with data transfers only between nearest neighbors. The ADI subprograms are reasonably efficient; mostly matrix operations with few scalar and no vector operations. However, both algorithms have some hidden defects. In updating the box variables for each set in the relaxation scheme only one forth of the processors do useful work; the remaining processors are masked out. This is because only one corner of each cell of a set is updated each time. For each level of the elimination process in the cyclic elimination algorithm, a set of data is shifted off the array and an equal set of zeros is shifted onto the array. This means that some of the processors are not doing useful work; here they are either multiplying by zero or adding a zero. This is a problem with many algorithms on SIMD machines.

Table I contains the execution time for each subprogram of the algorithm, that for one iteration in the case of *relaxd;* the percentage of the total time spent in that subprogram; and the processing rate. It is clear, from Table I, that the majority of the time was spent in *relaxd* for this particular run. This is because the average time step requires about 270 iterations and the total time spent in the other subprograms (*zcntr, cof, zbc, triied, trijed*) is only about the time to do two iterations of *relaxd.* The number of iterations in *relaxd* per time step depends on the data used during a given run. A different input data set could result in a smaller number of iterations per time step and relatively less time spent in the relaxation subprogram.

Table I. Measured execution time and processing rate of the Navier-Stokes subprograms for the 128 × 128 problem on the MPP.

Sub-program	Execution time (msec)	Perc. of time (%)	Processing rate (MFLOPS)
setbc	0.587	0.00	84
zcntr	2.694	0.06	24
relaxd	15.265*	99.23	156
cof	1.933	0.05	136
zbc	1.833	0.04	1.1
triied	12.717	0.31	125
trijed	12.725	0.31	125
overall#	41.597	100.00	155

* per iteration.

\# for ten time steps (execution time is in seconds here).

The processing rates in Table I are determined by counting only the arithmetic operations which truly contribute to the solution. Scalar and vector operations which were implemented as matrix operations are counted as scalar and vector operations. This is the reason why the subprograms *zbc* and *zcntr* have low processing rates; *zbc* has only vector operations while *zcntr* has some scalar operations implemented as matrix operations. The subprogram *setbc* has mostly scalar and data assignment operations which reduce its processing rate. Beside these three subprograms, the processing rate ranges from 125 to 155 MFLOPS with an average rate of about 140 MFLOPS.

In order to estimate the execution time of an algorithm on the MPP, the numbers of arithmetic and data transfer operations are counted and the cost of each operation is measured. This is illustrated in the following model. Only operations on parallel arrays are considered.

The execution time of an algorithm on the MPP, T, can be modeled as:

$$T = T_{cmp} + T_{cmm}, \tag{3.1}$$

$$T_{cmp} = t_c \, (N_a \, C_a + N_m \, C_m + N_d \, C_d), \tag{3.2}$$

$$T_{cmm} = t_c \, (N_{sh} \, C_{sh} + N_{st} \, C_{st}), \tag{3.3}$$

where T_{cmp} and T_{cmm} are the computation and communication times; t_c is the machine cycle time (t_c = 100 nsec); N_a, N_m, N_d, N_{sh}, and N_{st} are the numbers of additions, multiplications, divisions, shift operations, and steps shifted; and C_a, C_m, C_d, C_{sh}, and C_{st} are the numbers of cycles for addition, multiplication, division, startup shift operation, and each step of shift operation. Table II contains the measured values of the basic floating point operations.

Table II. Measured execution times (in machine cycles) of the floating point operations in MPP Pascal.

Add	Multiply	Divide	One step shift	k step shift
965	811	1225	168	136 + 32 k

Table III contains the operation counts per grid point for the Navier-Stokes subprograms on the MPP using the cyclic elimination algorithm for solving the tridiagonal systems. Note that scalar and vector operations (in *zcntr* and *zbc),* which were implemented as matrix operations, are considered here as matrix operations. Table IV contains the estimated computation and communication times using equations (3.2) and (3.3) and Tables II and III. The cost of scalar operations is not included in this model; this explains the differences between the estimated and measured times for *setbc* and *cof.* Apart from these two subprograms, the difference between the total estimated and measured times ranges between 3% to 8% of the measured times. The amount of time spent on data transfers is quite modest; from 6% for *relaxd* to 25% for *triied* and *trijed.* This is because this algorithm does not contain many data transfers and these transfers are only between nearest neighbors except for the tridiagonal solvers.

Table III. Operation counts per grid point for the Navier-Stokes subprograms on the MPP, using the cyclic elimination algorithm for solving the tridiagonal systems.

Sub-program	Add	Multiply	Divide	Shift	Steps shifted
setbc	1	1	1	-	-
zcntr	15	9	-	19	28
*relaxd**	119	26	-	42	84
cof	8	8	-	8	8
zbc	5	7	4	8	11
triied	30	45	22	44	764
trijed	30	45	22	44	764

* per iteration.

Table IV. Estimated execution times (in milliseconds) of the Navier-Stokes subprograms on the MPP.

Sub-program	Comp. time	Comm. time	Total est. time	Measured time
setbc	0.300	-	0.300	0.587
zcntr	2.177	0.348	2.525	2.694
relaxd	13.592	0.840	14.432	15.265
cof	1.421	0.134	1.555	1.933
zbc	1.540	0.144	1.684	1.833
triied	9.239	3.043	12.283	12.717
trijed	9.239	3.043	12.283	12.725

IV. Implementation on the Flex/32

The Flex/32 is an MIMD shared memory multiprocessor based on 32 bit National Semiconductor 32032 microprocessor and 32081 coprocessor [5]. The results presented here were obtained using the 20 processor machine at NASA Langley Research Center.

The machine has ten local buses; each connects two processors. These local buses are connected together and to the common memory by a common bus. The 2.25 Mbytes of the common memory is accessible to all processors. Each processor contains 4 Mbytes of local memory. Each processor has a cycle time of 100 nsec.

The Navier-Stokes algorithm, described in section II, was implemented on the Flex/32 using 64×64 grid points (63×63 cells) and 128×128 grid points (127×127 cells). The main program as well as the seven subprograms of the algorithm were written in Concurrent Fortran, which comprises the standard Fortran 77 language and extensions that support concurrent processing.

The parallel implementation of the Navier-Stokes algorithm is done by assigning a strip of the computational domain to a process and performing all the steps of the algorithm in each process. The main program performs only the input and output operations and creates and spawns the processes on specified processors. In our implementation, we used 1, 2, 4, 8, and 16 processors of the machine. The domain is decomposed first vertically for the first six subprograms (*setbc, zcntr, relaxd, cof, zbc,* and *triied*) and then horizontally for the subprogram *trijed.* The relaxation scheme for each strip was implemented locally. After relaxing each set of cells, each process exchanges the values of the interface points with its two neighbors through the common memory. The tridiagonal equations were solved using the Gaussian elimination algorithm for both passes of the ADI method. Data is stored in the common memory, in the local memory of each processor, or in both of them.

In order to satisfy data dependencies between segments of the code, a counter is used. This counter, which is a shared variable with a lock assigned to it, can be incremented by any process and be reset by only one process. It is implemented as a "barrier" where all processes pause when they reach it. A set of flags are also used for synchronization in the relaxation subprogram.

Table V contains the speedups and efficiencies as functions of the number of processors for the 64×64 and 128×128 problems. The measured execution times and processing rates using 16 processors are listed in Table VI. The efficiency of the algorithm ranges from about 94%, for the 64×64 problem using 16 processors, to about 99%, for the 128×128 problem using two processors.

Table V. Speedup and efficiency as functions of the number of processors, p, of the Navier-Stokes algorithm on the Flex/32.

p	64×64 points		128×128 points	
	speedup	efficiency	speedup	efficiency
1	1.000	1.000	1.000	1.000
2	1.959	0.980	1.976	0.988
4	3.893	0.973	3.941	0.985
8	7.715	0.964	7.850	0.981
16	15.027	0.939	15.483	0.968

Table VI. Measured execution times for ten time steps and processing rates for the Navier-Stokes algorithm using 16 processors of the Flex/32.

Problem size (grid points)	Execution time (sec)	Processing rate (MFLOPS)
64×64	268.7	1.09
128×128	2587.1	1.13

The performance model is based on estimating the values of the overheads resulting from running the algorithm on more than one processor. The execution time of an algorithm on p processors of the Flex/32, T_p, can be modeled as:

$$T_p = T_{cmp} + T_{ovr}, \tag{4.1}$$

where T_{cmp} is the computation time and T_{ovr} is the overhead time. Let f_{ld} be a load distribution factor where $f_{ld} = 1$ if the load is distributed evenly between the processors and $f_{ld} > 1$ if at least one processor has less work to do than the other processors. Then the computation time on p processors can be computed by

$$T_{cmp} = f_{ld}\, T_1 / p, \tag{4.2}$$

where T_1 is the computation time using a single processor.

The overhead time can be modeled by:

$$T_{ovr} = T_{cmo} + T_{spn} + T_{syn}, \tag{4.3}$$

where T_{cmo} is the common memory overhead time, T_{spn} is the spawning time of p processes, and T_{syn} is the synchronization time. Three components of the common memory overhead time can be identified:

$$T_{cmo} = T_{cma} + T_{cpl} + T_{cml}, \tag{4.4}$$

where T_{cma} is the common memory additional time - this results from storing additional variables in the common memory; T_{cpl} is the common plus local memory time - this results from storing variables in both the common and local memories; T_{cml} is the common minus local memory time - this results from storing variables in the common memory instead of local memory. The values of T_{spn}, T_{syn}, T_{cma}, T_{cpl}, and T_{cml} can be estimated as follows:

$$T_{spn} = p\, t_{spn}, \tag{4.5}$$

$$T_{syn} = p\, k_{lck}\, t_{lck}, \tag{4.6}$$

$$T_{cma} = n\, k_{cma}\, f_{bc}(p)\, t_{cma}, \tag{4.7}$$

$$T_{cpl} = n\, k_{cpl}\, (f_{bc}(p)\, t_{cma} + t_{lma}), \tag{4.8}$$

$$T_{cml} = n\, k_{cml}\, (f_{bc}(p)\, t_{cma} - t_{lma}), \tag{4.9}$$

where t_{spn} is the time to spawn one process - a reasonable value is 13 *m*sec; t_{lck} is the time to lock and unlock a variable - a reasonable value is 47 μsec; t_{cma} is the time to access a variable in common memory - a reasonable value is 6 μsec; t_{lma} is the time to access a variable in local memory - a reasonable value is 5 μsec; k_{lck} is the number of times a variable is locked and unlocked for each process; k_{cma} is the number of times an additional variable is referenced; k_{cpl} is the number of times a variable is stored in both local and common memory; k_{cml} is the number of times a variable is stored in common memory instead of local memory; and $f_{bc}(p)$ is the bus contention factor - it is a function of p. It is assumed that all memory operations are performed on vectors of length n.

The performance of the Navier-Stokes algorithm is heavily influenced by the performance of the relaxation subprogram; about 98% of the total time was spent in this subprogram. Since the number of cells is not divisible by the number of processors used, the last processor has less work to do than the other processors. Therefore, the load distribution factor, equation (4.2), can be computed by

$$f_{ld} = \left\lceil \frac{n-1}{p} \right\rceil \left(\frac{p}{n-1} \right). \tag{4.10}$$

Using the performance model, equations (4.1) through (4.10), the overhead time represents at most 5% of the execution time of the algorithm, including the load distribution factor. The overhead time of the relaxation subprogram dominates the total overhead time. The values of k_{lck} and k_{cma} for each iteration of the relaxation process are 1 and 8. The spawning time has a minor impact on the overhead time because the processes are spawned only once during the lifetime of the program. The synchronization time is insignificant because the routines that provide the locking mechanism are very efficient and overlap with the memory access. The bus contention factor is very small. The common memory additional time, T_{cma}, dominates the overhead time. This overhead results from accessing the interface points for each iteration of the relaxation subprogram. The other components of the common memory overhead time, T_{cpl} and T_{cml}, have a negligible impact on the total overhead time because these operations are performed only once during every time step.

V. Implementation on the Cray/2

The Cray/2 is an MIMD supercomputer with four Central Processing Units, a foreground processor which controls I/O and a main memory. The results reported here were obtained using the old Cray/2 at NASA Ames Research Center; the new one has a shorter main memory access time than the old one.

The Navier-Stokes algorithm, described in section II, was implemented on one processor of the Cray/2 using 64×64 and 128×128 grid points. The reordered form of the relaxation scheme, the four color scheme, was implemented on the Cray/2 with no major modifications. The reordering process removes any recursion because each of the four sets (colors) contains disjoint cells. The two sets of the tridiagonal systems were solved by the Gaussian elimination algorithm for all systems of each set in parallel. This was done by changing all variables of the algorithm into vectors running across the tridiagonal systems. The inner loops of all of the seven subprograms of the Navier-Stokes algorithm were fully vectorized. The local memory was used to store some of the variables, whenever that was possible. This reduces main memory conflicts and speeds up the calculation.

Tables VII and VIII contain the execution time for each subprogram, the percentage of the total time spent in that subprogram, and the processing rate for the 64×64 and 128×128 problems. Most of the time was spent in *relaxd,* and the average time step requires about 110 iterations for the 64×64 problem and about 270 iterations for the 128×128 problem. The subprogram *setbc* has a low processing rate because it has mostly memory access and

scalar operations; however, this subprogram is called only once during the lifetime of the program. Beside this subprogram, the processing rate ranges from 57 to 97 MFLOPS with an average rate of about 70 MFLOPS for the subprograms of both problems.

Table VII. Measured execution time and processing rate of the Navier-Stokes subprograms for the 64 × 64 problem on one processor of the Cray/2.

Sub-program	Execution time (msec)	Perc. of time (%)	Processing rate (MFLOPS)
setbc	0.480	0.02	25
zcntr	0.252	0.08	63
relaxd	2.719*	99.02	96
cof	0.720	0.24	85
zbc	0.015	0.01	66
triied	1.007	0.33	57
trijed	0.928	0.30	62
overall#	3.048	100.00	96

* per iteration

\# for ten time steps (execution time is in seconds here).

Table VIII. Measured execution time and processing rate of the Navier-Stokes subprograms for the 128 × 128 problem on one processor of the Cray/2.

Sub-program	Execution time (msec)	Perc. of time (%)	Processing rate (MFLOPS)
setbc	1.651	0.01	29
zcntr	1.059	0.03	61
relaxd	11.001*	99.60	97
cof	3.036	0.10	84
zbc	0.034	0.00	59
triied	4.014	0.13	59
trijed	3.870	0.13	62
overall#	30.286	100.00	97

* per iteration

\# for ten time steps (execution time is in seconds here).

Based on the fact that Cray vector operations are "stripmined" in sections of 64 elements, the time required to perform arithmetic and memory access operations on vectors of length L_{vcr} can be modeled as follows:

$$T_{f1} = \left(\left\lceil \frac{L_{vcr}}{64} \right\rceil L_f + L_{vcr}\right) N_{f1}\, CP, \qquad (5.1)$$

$$T_{f2} = \left(\left\lceil \frac{L_{vcr}}{128} \right\rceil L_f + \frac{L_{vcr}}{2}\right) N_{f2}\, CP, \qquad (5.2)$$

$$T_{m1} = \left(\left\lceil \frac{L_{vcr}}{64} \right\rceil L_m + R_1 L_{vcr}\right) N_{m1}\, CP, \qquad (5.3)$$

$$T_{m2} = \left(\left\lceil \frac{L_{vcr}}{128} \right\rceil L_m + R_2 \frac{L_{vcr}}{2}\right) N_{m2}\, CP, \qquad (5.4)$$

where T_{f1} and T_{f2} are the times to perform floating point operations with strides of 1 and 2; T_{m1} and T_{m2} are the times to perform main memory access operations with strides of 1 and 2; CP is the clock period (CP = 4.1 nsec); L_m is the length of main memory to registers path (L_m = 56 CPs); L_f is the length of floating point functional unit (L_f = 23 CPs); R_1 and R_2 are the data transfer rates through main memory with strides of 1 and 2 (reasonable values are R_1 = 1 and R_2 = 3.5, although competition from other processors causes a lower transfer rates and hence increased values of R_1 and R_2); N_{f1} and N_{f2} are the numbers of floating point operations with strides of 1 and 2; and N_{m1} and N_{m2} are the numbers of main memory access operations with strides of 1 and 2.

Table IX contains the operation counts per grid point for the Navier-Stokes subprograms using the Gaussian elimination algorithm for solving the tridiagonal systems. These operations are performed on all grid points of the domain except for *zbc* where they are performed on vectors. Tables X and XI contain the estimated times of the Navier-Stokes subprograms for the 64 × 64 and 128 × 128 problems. These times are obtained using equations (5.1) to (5.4) and Table IX. It is assumed that each division takes four times the multiplication time. The main memory access time for each subprogram represents about 50% to 70% of the total estimated and measured time. This shows that the Cray/2 is a memory bandwidth bound machine. The memory stride of 2 in *relaxd* causes more than a 50% slowdown in data transfer rate. The difference between the total estimated and measured values can be attributed to several causes. Among these are: the memory access and arithmetic operations can overlap, the time to perform scalar operations is not included, and there is up to 20% offset on the results depending on the memory traffic and the number of the active processes. Finally, this model does not take into account the overlapping between segments of long vectors for the same operation. However, it was found that this overlapping is insignificant for Fortran programs.

Table IX. Operation counts per grid point for the Navier-Stokes subprograms on the Cray/2, using the Gaussian elimination algorithm for solving the tridiagonal systems.

Sub-program	Add	Multiply	Divide	Memory access
setbc	1	1	1	8
zcntr	3	1	-	5
*relaxd**	46	20	-	31
cof	8	8	-	16
zbc#	5	7	4	20
triied	6	7	2	17
trijed	6	7	2	17

* per iteration # vector operations.

Table X. Estimated and measured execution times (in milliseconds) of the Navier-Stokes subprograms for the 64 × 64 problem on one processor of the Cray/2.

Sub-prog.	Mem. time	Add time	Mult. time	Est. time	Measured time
setbc	0.246	0.022	0.111	0.379	0.480
zcntr	0.154	0.067	0.022	0.243	0.252
relaxd	1.915	1.206	0.551	3.672	2.719
cof	0.480	0.173	0.173	0.826	0.720
zbc	0.010	0.002	0.008	0.020	0.015
triied	0.510	0.130	0.324	0.964	1.007
trijed	0.510	0.130	0.324	0.964	0.928

Table XI. Estimated and measured execution times (in milliseconds) of the Navier-Stokes subprograms for the 128 × 128 problem on one processor of the Cray/2.

Sub-prog.	Mem. time	Add time	Mult. time	Est. time	Measured time
setbc	0.996	0.090	0.450	1.536	1.651
zcntr	0.622	0.270	0.090	0.982	1.059
relaxd	6.826	4.144	1.802	12.772	11.001
cof	1.967	0.711	0.711	3.389	3.036
zbc	0.019	0.004	0.016	0.039	0.034
triied	2.090	0.533	1.333	3.956	4.014
trijed	2.090	0.533	1.333	3.956	3.870

VI. Comparisons and Concluding Remarks

There are a number of measures that one can use to compare the performance of these parallel computers using a particular algorithm. One is the processing rate and another is the execution time (see Tables I, VI, VII and VIII). However it must be borne in mind that both of these measures depend on the architectures of the computers, the overhead required to adapt the algorithm to the architecture, and the technology, that is, the intrinsic processing power of each of the computers.

If we consider a single problem, a ten time step run of the algorithm on a 128 × 128 grid, then the processing rate is a maximum for the MPP, 155 MFLOPS, compared to 97 MFLOPS for the Cray/2, and only 1.13 MFLOPS on 16 processors of the Flex/32. The low processing rate of the algorithm on the 16 processors of the Flex/32 is simply due to the fact that the National Semiconductor 32032 microprocessor and 32081 coprocessor are not very powerful. Although the algorithm has a higher performance rate on the MPP than on the Cray/2, it takes less time to solve the problem on the Cray/2 than on the MPP. This is due to the algorithm overhead involved in adapting the algorithm to the MPP. As shown in Tables III and IX, each iteration of the relaxation process has 145 arithmetic operations per grid point on the MPP compared to 66 operations per grid point on the Cray/2. Also, the cyclic elimination algorithm, used on the MPP, has 92 arithmetic operations per grid point while the Gaussian elimination algorithm, used on the Cray/2, has only 10 operations per grid point; not including computation of the forcing terms.

The implementation of the algorithm on the Flex/32 has the same number of arithmetic operations per grid point as on the Cray/2; there is only a reordering of the calculations and no additional arithmetic operations in the overhead. The algorithmic overhead for the Flex/32 version is the cost of exchanging the values of the interface points and setting the synchronization counters for the relaxation scheme and accessing the common memory for the ADI method. This means that the code on each processor is the serial code plus the overhead code. When the code is run on one processor, it is just the serial code with the overhead portion removed.

Another measure of performance is the number of machine cycles required to solve a problem. This measure reduces the impact of technology on the performance of the machine. For the 128 × 128 problem, for example, the ten time step run requires about 416 billion cycles on the MPP, 7387 billion cycles on the Cray/2, and 25871 billion cycles on 16 processors of the Flex/32. This means that the MPP outperformed the Cray/2, by a factor of 18, and the latter outperformed the Flex/32, by a factor of 3.5, in this measure. This also means that one processor of the Cray/2 outperformed 16 processors of the Flex/32 even if we assume that both machines have the same clock cycle. The problem with the Flex/32 is that, although each processor has a cycle time of 100 nsec, the memories (local and common) have access times of about 1 μsec.

One simple comparison between the MPP and Cray/2 is the time to perform a single arithmetic operation using the models developed in sections III and V. Using equation (5.1), the time to perform a single floating point operation (addition or multiplication) on an array of size 128 × 128 elements on the Cray/2, excluding the memory access cost, is 91.3 μsec. The time to perform the same operation on the MPP using MPP Pascal, see Table II, ranges from 81.1 μsec (for multiplication) to 96.5 μsec (for addition). This shows that the processing power of a single functional unit of the Cray/2 is comparable to the processing power of the 16384 processors of the MPP. However, much of the overhead is not included in this comparison: memory access cost on the Cray/2, data transfers on the MPP, and so on.

This experiment showed that by reordering the computations we were able to implement the relaxation scheme on three different architectures with no major modifications. Two different algorithms, Gaussian elimination and cyclic elimination, were used to solve the tridiagonal equations on the three architectures; the two algorithms were chosen to exploit the parallelism available on these architectures. The algorithm exploits multiple granularities of parallelism. The algorithm vectorized quite well on the Cray/2. A fine grained parallelism, involving sets of single arithmetic operations executed in parallel, is obtained on the MPP. Parallelism at higher level, large grained, is exploited on the Flex/32 by executing several program units in parallel.

The performance model on the MPP was fairly accurate on predicting the execution times of the algorithm. The performance model on the Flex/32 showed the impact

of various overheads on the performance of the algorithm. The performance model on the Cray/2 was based on predicting the execution costs of separate operations. This model is used to identify the major costs of the algorithm and reproduced the measured results with an error of at most 35%.

The ease and difficulty in using a machine is always a matter of interest. The Cray/2 is relatively easy to use as a vector machine. Existing codes that were written for serial machines can always run on vector machines. Vectorizing the unvectorized inner loops will improve the performance of the code. Unlike parallel machines, vector machines do not have the problem of "either you get it or not". The Flex/32 is not hard to use, except for the unavailability of debugging tools which is a problem for many MIMD machines (a synchronization problem could cause a program to die). On the other hand, the MPP is not a user-friendly system. The size of the PE memory is almost always an issue. MPP Pascal does not permit vector operations on the array nor does it allow extraction of an element of a parallel array. The MCU has 64 Kbytes of program memory. This memory can take up to about 1500 lines of MPP Pascal code. This means that larger codes can not run on the MPP. Finally, input/output is somewhat clumsy on the MPP. However, other machines with architectures similar to the MPP may not have the same problems that the MPP does.

There is one further observation of interest. This algorithm can be implemented concurrently on four processors of the Cray/2 (multitasking). The code will be similar to the Flex/32 version except that most of the variables should be stored in the main memory. Adapting this algorithm to a local memory multiprocessor with a hypercube topology should be relatively easy. A high efficiency is predicted in this case because all data transfers are to nearest neighbors and their cost should be very small compared to the computation cost.

Acknowledgements

We would like to thank David Wildenhain of SAR, Tom Crockett of ICASE, and Chris Willard of NAS for their assistance. This research was supported in part by NASA Contract NAS1-18107 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665. Also, research was supported in part by NASA Contract NAS2-11555 while the first author was an employee of Sterling Software under contract to the Numerical Aerodynamic Simulation Systems Division at NASA Ames Research Center, Moffett Field, CA 94035.

References

[1] Batcher, K. E., "Design of a Massively Parallel Processor," **IEEE Trans. Comput.,** Vol. C-29, 1980, pp. 836-840.

[2] Fatoohi, R. A., **Implementation and Performance Analysis of Numerical Algorithms on the MPP, Flex/32 and Cray/2,** Ph.D. dissertation, Old Dominion Univ., Norfolk, VA, 1987.

[3] Fatoohi, R. A. and Grosch, C. E., "Implementation of a Four Color Cell Relaxation Scheme on the MPP, Flex/32 and Cray/2," **Proc. 1987 Int. Conf. Par. Proc.,** pp. 424-426.

[4] Fatoohi, R. A. and Grosch, C. E., "Implementation of an ADI Method on Parallel Computers," **J. Scientific Computing,** Vol. 2, No. 2, 1987, pp. 175-193.

[5] Flexible Computer Co., **Flex/32 Multicomputer System Overview,** Publication No. 030-0000-002, 2nd ed., Dallas, TX, 1986.

[6] Gatski, T. B., Grosch, C. E., and Rose, M. E., "A Numerical Study of the Two-Dimensional Navier-Stokes Equations in Vorticity-Velocity Variables," **J. Comput. Phys.,** Vol. 48, No. 1, 1982, pp. 1-22.

[7] Goddard Space Flight Center, **MPP Pascal Programmer's Guide,** Greenbelt, MD, 1987.

[8] Grosch, C. E., "Adapting a Navier-Stokes code to the ICL-DAP," **SIAM J. Scientific & Statistical Computing,** Vol. 8, No. 1, 1987, pp. s96-s117.

[9] Hockney, R. W. and Jesshope, C. R., **Parallel Computers: Architecture, Programming and Algorithms,** Adam Hilger, Bristol, England, 1981.

SOLUTION OF VISCOUS FLUID FLOWS ON A DISTRIBUTED MEMORY CONCURRENT COMPUTER

Mark E. Braaten

Engineering Systems Laboratory

GE Research and Development Center
P.O. Box 8
Schenectady, New York 12301

ABSTRACT

A concurrent algorithm for the solution of the Navier-Stokes equations expressed in curvilinear coordinates has been developed for execution on a distributed memory parallel computer. This algorithm offers the ultimate promise of near-supercomputer performance on relatively low-cost parallel computers. The new algorithm is based on an existing serial pressure-correction-based algorithm, and partitions the problem onto the processors using areal decomposition. The algorithm is demonstrated on an Intel iPSC for a complicated two-dimensional laminar flow problem, for various grid sizes and numbers of processors. Speedup per iteration approaches 100% parallel efficiency as the grid size is increased. However, the convergence rate of the concurrent procedure tends to deteriorate somewhat relative to the original serial algorithm as the grid size and the number of processors are increased, limiting the maximum speedup that was achieved to a factor of 9.16 with 16 processors. The degradation in convergence rate is traced to a poorer solution of the pressure correction equation that is obtained in the concurrent procedure, and several remedies are proposed. Overall, the results are very encouraging.

1. INTRODUCTION

A major obstacle to the increased use of computational fluid dynamics in engineering design continues to be the long run times and high cost of the computer simulations. As an example, a run of a 3-D finite volume combustor code developed at the GE Research and Development Center [1] requires one to two hours of cpu time on a Cray-XMP supercomputer, for a grid with 75,000 grid points. Such a calculation would require hundreds of hours on a minicomputer or engineering workstation, making it impractical for routine design purposes. More exotic codes such as those used for the direct simulation of turbulence require even greater computational resources.

Parallel processing offers the promise of greatly reducing the execution times for CFD codes by using many processors to attack the problem simultaneously. Recent advances in VLSI technology have led to the development of a class of relatively low-cost concurrent computers, which use a moderate number of inexpensive processors, that offer near-Cray performance at a fraction of the cost, making them appear attractive for more routine use. The computational power of the individual processors ranges from that of a 16-bit microcomputer, in machines such as the Intel iPSC, and Ametek System 14, to fast vector processors in machines such as the Alliant FX/8 and the Intel iPSC/VX. Other parallel supercomputers with small numbers of supercomputer processors are available or under development (such as the ETA^{10} and Cray 3) which offer higher ultimate performance, but their high cost and limited availability make them less attractive for routine use for the foreseeable future.

If low-cost processors are to be used, many will be required to provide the level of performance required for CFD codes. Concurrent computers of this type can either be of a shared memory or distributed memory architecture. Memory conflicts can limit the number of processors that can be effectively used in a shared memory system. Distributed memory systems do not suffer from this limitation, and can be scaled up to hundreds of processors while retaining high parallel efficiencies. The disadvantage of distributed memory architectures is that the problem to be solved must be explicitly partitioned by the programmer onto the various processors in such a way that load balancing is maintained and communication between processors is minimized. For some problems, it may not be easy or even possible to find a satisfactory means of doing this partitioning. Fortunately, for finite volume fluid mechanics algorithms, a simple and natural geometrical partitioning of the problem meets the requirements very nicely. Consequently, distributed memory architectures appear very well suited for the solution of CFD problems, as has been noted by other researchers [2,3], and are the focus of attention here.

It is interesting to note that the evolution of both architectures appears to be toward hybrid systems offering aspects of both local and global memory, which will tend to blur the distinction between the two types of machines. Examples are the shared memory Flex/32 [2], which allows the memory to be allocated either locally or globally, and distributed memory machines like the experimental J-machine under development at MIT [4], in which the communication between the processors is so rapid that a virtual global memory addressing system can be implemented. In any case, since the field of parallel processing is developing so rapidly, with machines appearing (and disappearing) daily, the concurrent algorithm developed here was designed to be generally applicable to the general class of distributed memory computers, rather than to a particular machine.

In this work, the development of a concurrent algorithm for the solution of the Navier-Stokes equations expressed in curvilinear coordinates is described. First, the original serial algorithm is reviewed, and then the development of the concurrent algorithm is detailed. A theoretical analysis of the potential speedup available from the concurrent algorithm is followed by a demonstration of the algorithm on an Intel hypercube for a complicated 2-D laminar flow problem.

2. REVIEW OF ORIGINAL ALGORITHM

The numerical algorithm developed here for execution on a distributed memory parallel processor is a direct extension of the serial incompressible flow algorithm described in [5,6]. Only a brief outline of the original algorithm is given here; the reader is referred to the original references for the details. For simplicity, the discussion is limited to incompressible laminar flows, but the

algorithm is extendable to turbulent, compressible, and chemically reacting flows.

The governing conservation equations typically can be written in the Cartesian coordinates for the dependent variable ϕ in the following form

$$\frac{\partial}{\partial x}(\varrho u\phi) + \frac{\partial}{\partial y}(\varrho v\phi) = \frac{\partial}{\partial x}\left(\Gamma \frac{\partial \phi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\Gamma \frac{\partial \phi}{\partial y}\right) + R(x, y) \tag{1}$$

where Γ is the effective diffusion coefficient and R is the source term. When new independent variables ξ and η are introduced, Eq. (1) changes according to the general transformation $\xi = \xi(x, y)$, $\eta = \eta(x, y)$. Equation (1) can be rewritten in (ξ, η) coordinates as follows:

$$\frac{1}{J}\frac{\partial}{\partial \xi}(\varrho U\phi) + \frac{1}{J}\frac{\partial}{\partial \eta}(\varrho V\phi) = \frac{1}{J}\frac{\partial}{\partial \xi}\left[\frac{\Gamma}{J}(q_1\phi_\xi - q_2\phi_\eta)\right] + \frac{1}{J}\frac{\partial}{\partial \eta}\left[\frac{\Gamma}{J}(-q_2\phi_\xi + q_3\phi_\eta)\right] + S(\xi, \eta) \tag{2}$$

where U and V are the contravariant velocity components, q_1, q_2, and q_3 are metric terms arising from the coordinate transformation, J is the Jacobian of the transformation, and $S(\xi, \eta)$ is the source term in the $\xi-\eta$ coordinates.

A staggered grid system is adopted, following the standard practice for incompressible finite volume algorithms. The scalar variables (p, ϱ) are located at the center of the control volumes and the Cartesian and contravariant velocity components are located on the faces of the control volumes. Discretization of Eq. (2), along with suitable interpolation for the variables whose values are unknown on the control volume faces, leads to the following general form of the conservation equation for the variable ϕ:

$$A_P\phi_P = \sum_{i = E,W,N,S} A_i\phi_i + (S_\phi)_P \tag{3}$$

The subscripts P,E,W,N, and S refer to the grid point at the center of the control volume and the four neighboring grid points, respectively. The term $(S_\phi)_P$ includes the original source term in the equation, plus the additional terms that cannot be approximated by the values of ϕ at the five grid points. A successive-line-underrelaxation method is used to solve the resulting finite difference equations for each variable ϕ.

The momentum and continuity equations, along with appropriate boundary conditions, make up the complete description of a laminar flow. The solution of these coupled equations makes up the kernel of any computational fluid mechanics algorithm. Laminar flows are commonly used to test the performance of numerical algorithms since the effects of the pressure-velocity coupling, which usually controls the convergence of the algorithm, are most clearly evident for such flows [7].

The method used to solve the coupled system of momentum and continuity equations is a pressure-correction method similar to that described in [8]. Following the notation of Ref. [5], the momentum equations can be written as

$$A_P^u u_P = \sum_{i = E,W,N,S} A_i^u u_i + D^u + (B^u p_\xi + C^u p_\eta) \tag{4}$$

$$A_P^v v_P = \sum_{i = E,W,N,S} A_i^v v_i + D^v + (B^v p_\xi + C^v p_\eta) \tag{5}$$

The Ds represent coefficients arising from the viscous cross-derivative terms, and the Bs and Cs represent the projected areas acted on by the pressure gradients in the ξ and η directions, respectively. The momentum equations can be solved, for a given pressure distribution p^*, to yield a tentative velocity field u^*, v^*. Since u^*, v^* do not necessarily satisfy the continuity equation, they and the guessed pressure field p^* must be updated. The corrected velocities and pressure are obtained from:

$$\begin{aligned} p &= p^* + p' \\ u &= u^* + u' \\ v &= v^* + v' \end{aligned} \tag{6}$$

Through manipulation of the momentum equations, and the formulas defining the contravariant velocities U, V in terms of u, v and the various metric derivatives, the velocity corrections U', V' can be expressed in terms of p', through the relations:

$$\begin{aligned} U' &= (B^u y_\eta - B^v x_\eta)\, p'_\xi \\ V' &= (C^v x_\xi + C^u y_\xi)\, p'_\eta \end{aligned} \tag{7}$$

These expressions are then substituted into the discrete continuity equation:

$$(\varrho U)_e - (\varrho U)_w + (\varrho V)_n - (\varrho V)_s = 0 \tag{8}$$

leading to the final form of the pressure correction equation:

$$\begin{aligned} &(\varrho U^*)_e + (\alpha\varrho)_e (p'_E - p'_P) - (\varrho U^*)_w \\ &- (\alpha\varrho)_w (p'_P - p'_W) + (\varrho V^*)_n + (\beta\varrho)_n (p'_N - p'_P) \\ &- (\varrho V^*)_s - (\beta\varrho)_s (p'_P - p'_S) = 0 \end{aligned} \tag{9}$$

where α and β are the coefficients derived by combining Eq. (7) and Eq. (8). This equation is solved, and the pressure and the velocity components are updated, completing one global iteration. Due to the nonlinearity of the problem, a number of global iterations are required to obtain a converged solution.

3. CONCURRENT IMPLEMENTATION

Effective implementation of the algorithm described in Section 2 on a distributed memory concurrent computer requires the satisfactory resolution of three major computational issues, namely: 1) load balancing, 2) minimization of communications costs, and 3) the development of an efficient concurrent algorithm. The development of an efficient concurrent algorithm is the critical step in making effective use of any parallel architecture. In the attempt to keep all of the processors busy generating floating point numbers at impressive combined mflop rates, it is easy to lose sight of the fact that the true goal is to achieve the same solution to the physical problem in less time than with a single processor.

The concurrent algorithm described in this section was implemented on a 16-node, memory-enhanced Intel hypercube. Each processor is rated at only 0.03 mflop; consequently, ideal performance with a 16-node system is only about 0.5 mflops. Consequently, the emphasis here is not on the absolute performance of the code, but on the speedup obtained with multiple processors relative to a single processor. The ultimate goal is to run the code on a machine with faster processors to achieve near-supercomputer performance. Although the basic concurrent algorithm is applicable to any distributed memory parallel processor, details of the implementation motivated by the specific architecture of the iPSC are mentioned when appropriate.

A widely used technique for partitioning the solution of a partial differential equation onto a number of processors is the areal decomposition method [9]. This method represents an extension of the classical Schwarz alternating method [10] to a parallel architecture. The solution domain is divided up into a number of overlapping subdomains, and each subdomain is assigned to a different processor. Overlapping is necessary so that each interior grid point is treated as an interior point in at least one subdomain. In parallel, the coefficients of the equation are calculated in each subdomain, and an iterative solution is obtained in each region to some reasonable level of convergence. The boundary values are then exchanged with the neighboring subdomains, and the solution is iterated further. When some suitable global convergence criterion is satisfied, the solutions on each subdomain can be assembled into the complete solution for the entire domain.

Since the areal decomposition method appears to be a natural (and general purpose) way of partitioning problems involving the solution of pde's by either finite volume or finite element methods for execution on a parallel computer, it was the method adopted here. The solution domain is divided up into a number of overlapping subdomains, one per processor, with the number of grid points in each subdomain approximately equal. Since the work required for each node point is roughly the same, this ensures that load balancing is excellent.

In a curvilinear coordinate code, much of the storage burden is taken up by the storage of the grid point positions and the many metric derivative terms that arise from the coordinate transformation. Since the grid is held fixed in the course of the calculation, it is far more computationally efficient to compute the metric terms once and for all at the beginning and store the results, rather than recomputing the metric terms repeatedly throughout the calculation. A geometrical decomposition has the advantage that this metric information needs only be stored in the local memory of the processor that is assigned to that region of the domain, and does not need to be communicated between processors. Only the interfacial values of the solution variables, which are updated in the course of the solution, need to be passed between processors. The only storage penalty that arises from the areal decomposition method is due to the need to store the metric information and solution variables in the overlapping regions twice.

The solution domain can be divided into strips, boxes, or arbitrarily shaped regions with roughly the same number of points. The key factor for minimizing the communications costs is to maximize the ratio of computation in each processor to the communication between processors. The computational work in each subdomain is dependent on the number of control volumes (volume) of the subdomain, while the amount of communication between subdomains depends on the number of boundary cells (surface area) of the subdomain. For a large enough problem and a moderate number of processors, decomposition by strips leads to a smaller surface-to-volume ratio of the subdomains than does a boxwise decomposition. With the structured grid used in a finite volume formulation, there is no need to resort to arbitrarily shaped regions, although they are useful in a finite element context. The disadvantage of stripwise decomposition is that the number of processors that can be effectively used is limited to something on the order of n, where n is the largest number of control volumes in any given direction. For both two- and three-dimensional problems, n will seldom exceed 100; consequently, the number of processors that can be effectively used is similarly limited. In three-dimensional problems, boxwise decomposition may be a better choice since it will allow a larger number of processors, and consequently more computational power, to be applied to the problem.

The best choice for the partitioning of the problem depends on the number of processors available and the ratio of computational speed to communications speed for the machine under consideration. The Intel iPSC hypercube consists of an Intel 80286-based host machine and a computational cube, consisting of up to 128 Intel 80286-based processors connected in a hypercube arrangement. All communication is slow relative to the speed of computation. With this number of processors and communication speed, stripwise decomposition is a good choice for this machine.

Due to the nature of the staggered grid used, two rows of grid points must be overlapped to ensure that all interior u velocities appear as interior points in at least one subdomain. This also causes one extra row of v velocities and pressures to be solved redundantly in each subdomain, which increases the computational effort for the concurrent algorithm somewhat over that of the original serial algorithm.

There are a number of characteristics of the iPSC that influence the detailed coding of the algorithm required to achieve good concurrent performance. There is a significant overhead associated with routing messages from one processor to the next, putting a premium on nearest neighbor communication. This overhead is overcome by mapping the hypercube to the required linear array through the use of binary reflected Gray codes [11]. Gray codes are sequences of n-bit binary numbers with the properties that any two successive numbers differ only in one bit, and all binary numbers with n bits are included. Each processor is then a nearest neighbor to the two processors that are handling the two adjacent subdomains. As the code is implemented here, the host machine is used only to read in the grid file and the input variables, monitor convergence, and write out the final solution. Note that on the existing iPSC, all I/O must be done on the host machine. An

efficient concurrent broadcast routine, contained in Intel's protocode library, is used to pass all of the input information from the host to each processor in the form of identical messages. In parallel, each processor then extracts the portion of the grid file that it needs, and the input variables from these messages. This procedure is much more efficient than having the host machine sequentially create and send individualized messages to each node. The host machine is slow, and the cost of host-to-processor messages is high. The concurrent broadcast routine requires only one host-to-processor message, with all the other messages passed via a spanning tree from processor to processor, with some message passing occurring in parallel. Convergence is monitored by sending the mass residual to the host machine and comparing it to the prescribed convergence criteria. A partial mass residual is computed by each processor and summed together using a concurrent concentration routine that forms the total mass residual and sends it to the host via a spanning tree, in a reverse manner to what was done in the concurrent broadcast. This convergence check is performed every fifth iteration to reduce the overhead of the node-to-host communication.

The existing serial solution algorithm solves the governing equations in the following order: x-momentum, y-momentum, and finally pressure correction. The same structure is retained in the concurrent implementation. In parallel, the coefficients for the x-momentum equation are computed for each subdomain. In parallel, a few sweeps of an iterative, block-corrected, line-by-line solution procedure [12] are performed in each subdomain. During this process, the boundary values for u in each subdomain are held fixed. Next, the boundary values for u are exchanged between subdomains, and the line-by-line solution procedure is repeated. The number of repetitions of the line-by-line solution and the exchange of the boundary values is prescribed by the user.

Upon completion of the x-momentum equation, the procedure is repeated for the y-momentum equation, and finally for the pressure correction equation. This equation by equation procedure was selected for several reasons. First, it reduces to the original algorithm for a single processor. Thus, comparison of the speedup obtained with p processors over p subdomains is made relative to the original algorithm on a single processor over the entire solution domain, which reflects a comparison of the parallel algorithm with the best serial algorithm, in the spirit of S'_p as defined by Ortega and Voigt [13]. Second, if enough repetitions of the line-by-line solver and the exchange of boundary points are performed so that a converged solution is obtained for each equation before proceeding to the next equation, then the overall rate of convergence of the algorithm will be the same as for the serial algorithm if each equation is also solved to convergence at each stage. Although it is not the usual practice to solve each equation fully to convergence at each step, since the coefficients are only tentative, this similarity suggests that the new concurrent algorithm will show comparable convergence behavior to the serial algorithm, which has proved to converge well for a large variety of problems.

4. THEORETICAL SPEEDUP ANALYSIS

Theoretical estimates can be made of the speedup that can be attained with p processors relative to a single processor for the current algorithm. The speedups achieved will be somewhat less than linear due to the following factors: 1) the additional computational work due to the overlapping regions, 2) the cost of the message passing, and 3) the reduction in convergence rate due to the change in the solution procedure from line-by-line over a single domain to the concurrent Schwarz alternating procedure.

Expressions for the degradation factors resulting from overlapping and communication costs can be formulated by counting the number of arithmetic operations and messages passed as a function of the problem size and the number of processors. The overlapping of the subdomains causes more lines of control volumes to be solved than in the serial algorithm, increasing the computational work, and the cost of message passing represents an overhead not found in the serial algorithm.

The computational effort in each subdomain is proportional to the number of control volumes in the subdomain, and can be expressed as:

$$(t_{\text{cpu}})_p = k_{\text{cpu}} \left(\frac{NI+p-1}{p}\right) NJ \tag{10}$$

where k_{cpu} represents the cpu time per control volume per iteration of the algorithm. Each processor exchanges essentially the same mix of messages with its two adjacent processors at each iteration. On the Intel iPSC, the overhead for message passing is so high that messages less than 1024 bytes all take essentially the same amount of time to transfer [9]. Since most of the message traffic consists of short messages, the time per iteration that a processor spends passing messages can be expressed as

$$(t_m)_p = k_m (2n_m) \tag{11}$$

Here k_m represents the average time required to send and receive a short message, and n_m represents the number of messages exchanged with an adjacent processor per iteration. In this simple analysis, the time processors spend idle waiting for messages that have not yet been sent, due to a lack of synchronization between processors, is included in this term.

The ratio of cpu time per iteration, including both computation and message passing, for the concurrent algorithm with p processors to the cpu time for the original serial algorithm becomes

$$\frac{t_p}{t_s} = \frac{k_{\text{cpu}} \left(\frac{NI+p-1}{P}\right) NJ + 2k_m n_m}{k_{\text{cpu}}\ \text{NI} \cdot NJ} \tag{12}$$

The ratio of the total computational time for the concurrent algorithm to that for the serial algorithm is given by

$$\frac{T_p}{T_s} = \frac{i_p t_p}{i_s t_s} \tag{13}$$

where i_p and i_s are the number of iterations required to obtain a solution to the same level of convergence for the parallel and serial algorithms, respectively. After rearrangement, the following expression for the speedup obtained with the concurrent algorithm with p processors over the original serial algorithm on one processor is obtained:

$$S = \left(\frac{i_s}{i_p}\right) \left[\frac{1}{1 + \frac{p-1}{NI} + \frac{k_m}{k_{\text{cpu}}} \frac{2n_m p}{NI \cdot NJ}}\right] p \tag{14}$$

In a general sense, the speedup S can be expressed as:

$$S = \epsilon p = \left(\frac{i_s}{i_p}\right)\left[\frac{1}{1+O+C}\right] p \qquad (15)$$

Here, the term ϵ equals the parallel efficiency, which is less than unity due to the three factors listed above, namely the overlapping penalty O, the communication cost C, and the reduction in convergence rate (i_s/i_p).

The important thing to note from the above expressions is that the penalties due to overlapping and communication become smaller and smaller as the problem size gets bigger, with everything else fixed. Hence, for a large enough problem, the only degradation from a linear performance improvement will result from any reduction in convergence rate that may result. Unfortunately, this degradation cannot be predicted analytically and must be determined from computational experiments. Note, that due to the particular formulation of the concurrent algorithm adopted here, the same convergence rate as the original serial algorithm can always be achieved by increasing the number of line-by-line sweeps. However, since this increases the cpu work per iteration, the total cpu time may or may not decrease. The optimum number of sweeps for each variable for the concurrent algorithm must also be determined via numerical experiments, and is not necessarily the same as for the original serial algorithm.

5. DEMONSTRATION RUNS

A series of demonstration runs of the new concurrent algorithm was made on a 16-node, memory-enhanced Intel iPSC hypercube at the University of Lowell, MA. The concurrent version of the algorithm was first developed and tested using Intel's hypercube simulator, running under Unix 4.2 on a SUN 3/160 workstation. Although it certainly would be useful to confirm the performance of the concurrent algorithm for a much wider range of flow configurations, the similarity of the concurrent algorithm to the serial algorithm, which has been widely tested, gives confidence that the limited results presented here will be representative of the performance of the algorithm in general.

The test case selected involves steady laminar flow in the axisymmetric afterburner configuration shown in Figure 1. As described earlier, it is useful to study laminar flows since the solution of the coupled momentum and continuity equations forms the kernel of any computational fluid dynamics algorithm and can be most clearly studied in laminar flow. The solutions presented here are a first step towards a realistic afterburner simulation, which will include equations for turbulence and combustion that can be solved by the same basic procedure.

Due to limitations on cpu time, very large grid sizes could not be attempted. Two body-fitted grids, one with 32 × 20 nodes, and the second with 64 × 20 nodes, were used in the calculations. Both grids are shown in Figure 2. For reference, the converged solution for the fine grid is shown in Figure 3, by means of plots of the calculated velocity vectors and the streamlines. Although the flow is laminar, the flowfield is not simple and contains wake regions behind the flameholders and a recirculation zone near the trailing edge of the centerbody.

Although the performance of the original serial algorithm is affected by the choice of such parameters as underrelaxation factors for the velocities and pressure, and the number of line sweeps for each variable, previous studies [14] have shown the sensitivity is not that great, provided reasonable values are used. In this work, no attempt has been made to optimize these factors for each run; rather, reasonable values of these parameters were held fixed for all runs. The underrelaxation factors for the x- and y-momentum equations were taken equal to 0.3, and that for pressure was taken equal to 0.5. Two sweeps of the line-by-line procedure were used for both x- and y-momentum, with three sweeps taken for pressure correction.

The number of iterations required for convergence is dependent on the choice of convergence criterion. Here, the solutions were taken to be converged when the normalized mass residual was less than 10^{-3}. It was confirmed that the converged solutions obtained with the concurrent algorithm were independent of the number of processors used, and identical to those obtained with the original serial algorithm.

5.1 Results

As discussed earlier in Section 4, as the grid size is increased, the speedup per iteration of the algorithm should approach a linear speedup with the number of processors, since the penalties associated with overlapping and communication become less significant. Figure 4 demonstrates that the speedups obtained on the test problem for the two grid sizes exhibit this trend. With 16 processors, the maximum speedup per iteration increased from

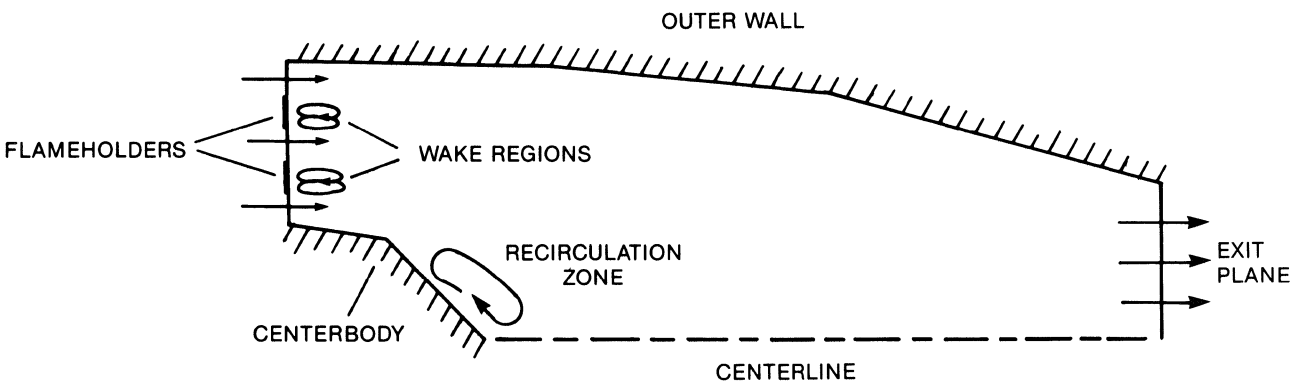


Figure 1. Axisymmetric afterburner configuration.

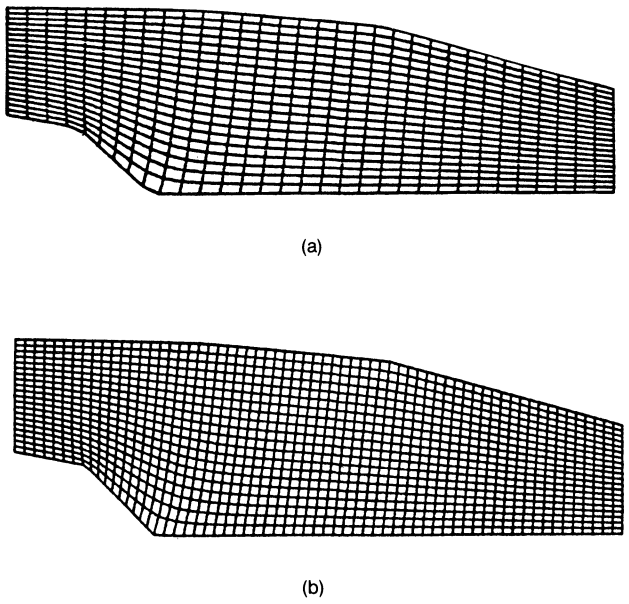


Figure 2. Computational grids for test problem, a) coarse 32 × 20 grid, b) fine 64 × 20 grid.

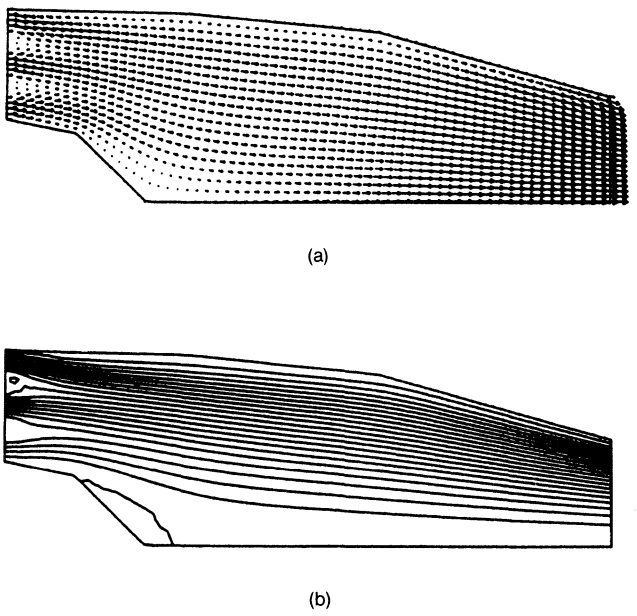


Figure 3. Converged solution for the fine grid, a) velocity vectors, b) streamlines.

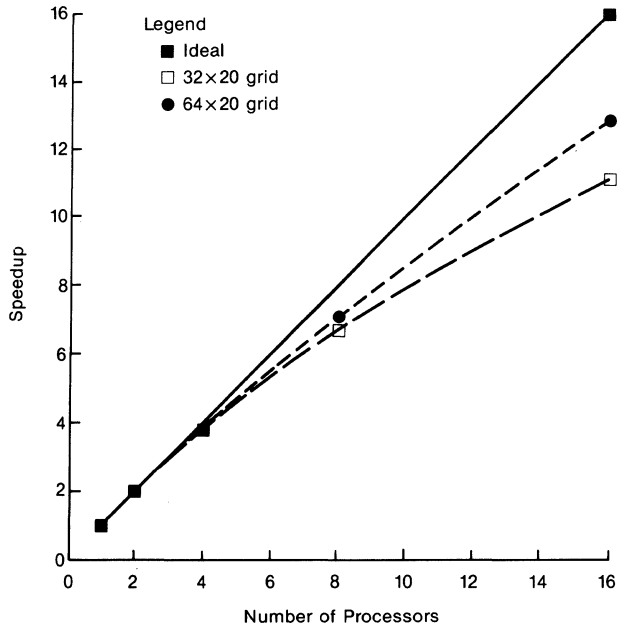


Figure 4. Speedup per integration for concurrent algorithm on Intel iPSC.

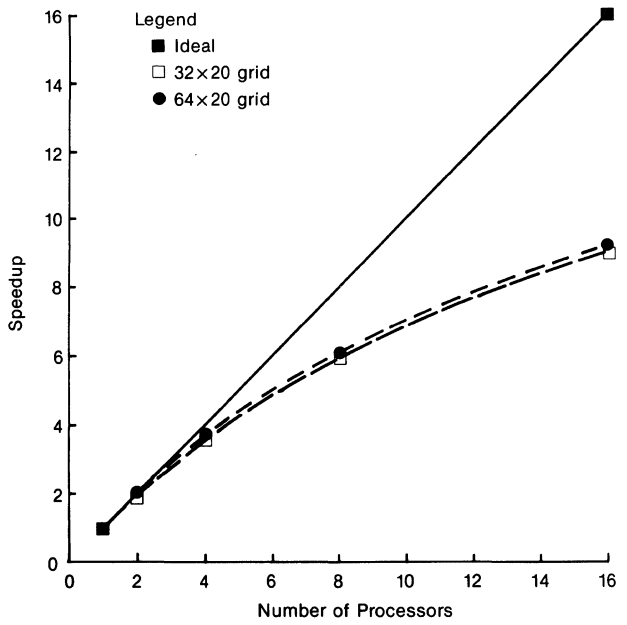


Figure 5. Speedup of total time for concurrent algorithm on Intel iPSC.

11.12 for the coarse grid to 12.84 for the fine grid, representing parallel efficiencies of about 70% and 80%, respectively. The major contribution to the efficiency being less than 100% comes from the overlapping of the subdomains, rather than from the communications costs, which appear minimal.

However, the speedup in terms of total computational time, shown in Figure 5, does not show the same improvement as the number of grid points is increased, due to a degradation in the convergence rate of the algorithm as the number of grid points and the number of processors is increased. The convergence paths, as a function of the number of processors, are shown for the coarse grid and the fine grid in Figures 6 and 7, respectively. Note that for both grids, the original serial algorithm (1 node) shows a smooth monotonic reduction in the mass residual, reaching an asympototic rate of convergence that is substantially less than the

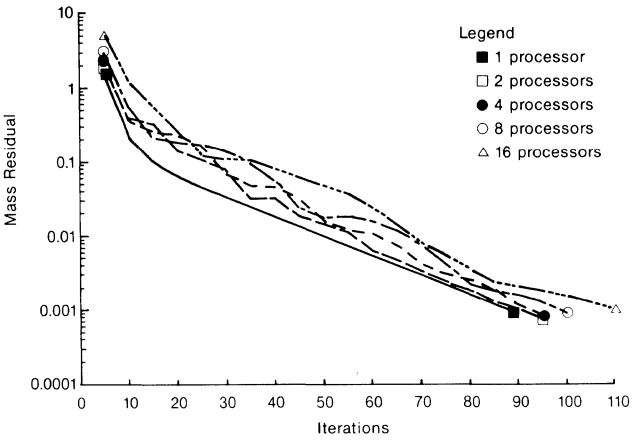


Figure 6. Convergence behavior of concurrent algorithm on coarse 32 × 20 grid.

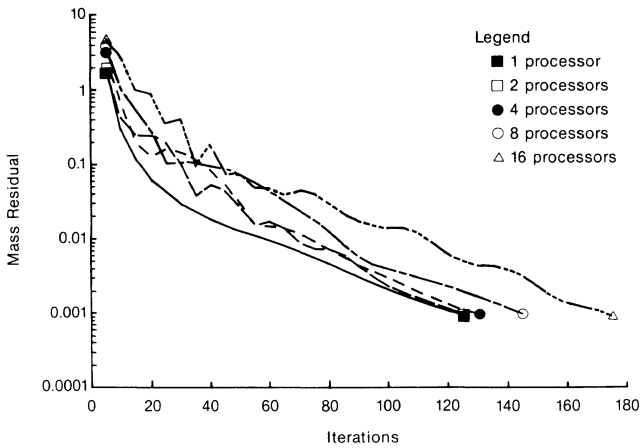


Figure 7. Convergence behavior of concurrent algorithm on fine 64 × 20 grid.

initial convergence rate. The concurrent algorithm shows a noisier convergence path with a slower initial rate of convergence, but with a similar asymptotic rate of convergence. With 16 processors, the convergence rate was 24% less for the coarse grid, and 40% less for the fine grid, than the corresponding rates for the serial algorithm.

Earlier studies of the serial algorithm have shown that the elliptic nature of the pressure correction equation makes it more difficult to converge than the momentum equations, which tend to be convection-dominated [15]. Schwarz's alternating method is also known to be slowly convergent for highly elliptic problems when many subdomains are used [16]. A series of runs was made varying the number of line sweeps for the pressure correction equation, for the fine grid test case using the concurrent algorithm with 16 processors. Figure 8 shows that the convergence rate of the concurrent algorithm approaches that of the serial algorithm as the number of pressure correction sweeps is increased, as expected. This indicates that the reason for the slower convergence of the concurrent algorithm in the original case was a poorer solution of the pressure correction equation with the same number of sweeps used in the serial procedure. It is interesting to note from Figure 9 that the total computational time required by the concurrent algorithm is relatively insensitive to the number of line sweeps used for pressure correction, provided that at least three sweeps are performed. Any improvement in convergence rate achieved by performing more than five sweeps is more than offset by the increased cost per iteration that results.

6. CONCLUDING REMARKS

A concurrent algorithm for the solution of the Navier-Stokes equations expressed in curvilinear coordinates has been developed and demonstrated on a distributed memory parallel processor. The speedup per iteration approaches 100% parallel efficiency as the grid size is increased. However, a reduction in the convergence rate of the algorithm as the grid size and the number of processors are increased, caused by a poorer solution of the pressure correction equation, limits the speedup in terms of the total computational time relative to the original serial algorithm to less than

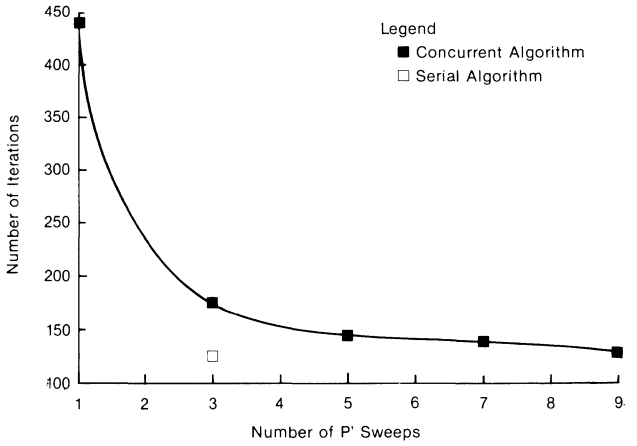


Figure 8. Effect of number of line sweeps for pressure correction equation on number of iterations required for convergence (64 × 20 grid, 16 processors).

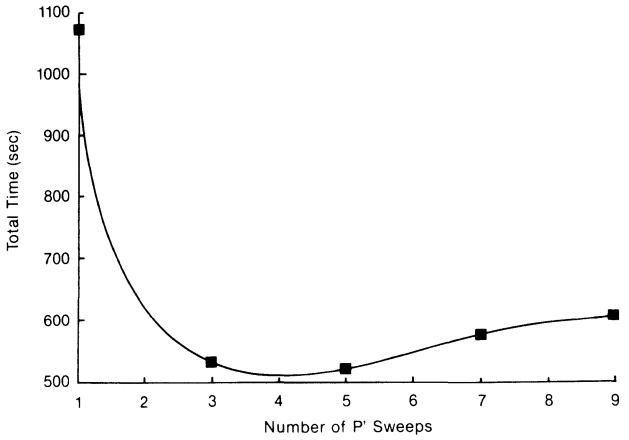


Figure 9. Effect of number of line sweeps for pressure correction equation on total time required by concurrent algorithm (64 × 20 grid, 16 processors).

linear. Despite this, the algorithm looks very promising, with the potential for giving near-supercomputer performance on a distributed memory machine with faster processors.

Further work is needed in the following areas:

1) Block correction or multigrid techniques should be investigated to improve the parallel solution of the pressure correction equation, in an attempt to minimize the degradation in convergence rate observed here for large grids and large numbers of processors. Multigrid methods have been successfully implemented on hypercubes [17], and have been shown to improve the convergence of the pressure correction equation for serial algorithms [18].

2) A convergence criterion should be used for the solution of each individual equation, rather than a fixed number of line sweeps, to ensure the best tradeoff between cost per iteration and overall convergence rate.

3) More test problems should be studied to verify the performance of the concurrent algorithm under a wider range of conditions. Problems involving turbulence and chemical reactions should be included in the study.

4) The concurrent algorithm should be run on a distributed memory machine with vector processors, such as Intel's iPSC/VX, to measure the speedup relative to a single processor and the total computational time. A successful demonstration will prove the practical utility of such machines for running CFD codes.

7. ACKNOWLEDGMENTS

The author would like to thank his colleague S.S. Tong of the GE Research and Development Center for his assistance in getting the hypercube simulator operational on his SUN workstation, Professors G. Pecelli and S. Smith of the Computer Science department of the University of Lowell for allowing me access to the university's Intel hypercube, and to P. Kautz and G. MacDonald of Intel for their assistance. Thanks are also due to my colleagues R. Mani and W. Shyy for helpful discussions, and to C. Cowsert for typing the equations.

8. REFERENCES

[1] W. Shyy, M.E. Braaten, and D.L. Burrus, "A Numerical Study of Flows in an Annular Gas-Turbine Combustor," GE TIS Report 86CRD257, Schenectady, NY, 1986.

[2] J. Townsend, T.A. Zang, and D.L. Dwoyer, "Fluid Dynamics Parallel Computer Development at NASA Langley Research Center," in A.K. Noor, ed., *Parallel Computations and Their Impact on Mechanics*, ASME, New York, 1987.

[3] D.E. Keyes and M.D. Smooke, "A Parallelized Elliptic Solver for Reacting Flows," in A.K. Noor, ed., *Parallel Computations and Their Impact on Mechanics*, ASME, New York, 1987.

[4] W.J. Dally, "Concurrent Computer Architecture," in A.K. Noor, ed., *Parallel Computations and Their Impact on Mechanics*, ASME, New York, 1987.

[5] W. Shyy, S.S. Tong, and S.M. Correa, "Numerical Recirculating Flow Calculation Using a Body-Fitted Coordinate System," *Num. Heat Trans.*, 8, 99-113, 1985.

[6] M.E. Braaten and W. Shyy, "A Study of Recirculating Flow Computation Using Body-fitted Coordinates: Consistency Aspects and Mesh Skewness," *Num. Heat Trans.*, 9, 559-574, 1986.

[7] G.D. Raithby and G.E. Schneider, "Numerical Solution of Problems in Incompressible Fluid Flow: Treatment of the Velocity-Pressure Coupling," *Num. Heat Trans.*, 2, 417-449, 1979.

[8] S. V. Patankar, *Numerical Heat Transfer and Fluid Flow*, Hemisphere Publishing Co., New York, 1980.

[9] O.A. McBryan and E.F. Van de Velde, "Hypercube Algorithms and Implementations," *SIAM J. Sci. Stat. Comput.*, 8, 5227-5287, 1987.

[10] Q.V. Dihn, R. Glowinski, and J. Periaux, "Solving Elliptic Problems by Domain Decomposition Methods with Applications," in G. Birkhoff, A. Schoenstadt, eds., *Elliptic Problem Solvers II*, Academic Press, Inc., New York, 1984.

[11] T.F. Chan, "On Gray Code Mapping for Mesh-FTTS on Binary N-Cubes," RIACS Technical Report 86.17, NASA Ames Research Center, Moffett Field, CA, 1986.

[12] S.V. Patankar, "A Calculation Procedure for Two-Dimensional Elliptic Situations," *Num. Heat Trans.*, 4, 409-425, 1981.

[13] J.M. Ortega and R.G. Voigt, "Solution of Partial Differential Equations on Vector and Parallel Computers," *SIAM Review*, 27, 149-240, 1985.

[14] W. Shyy, "Numerical Outflow Boundary Condition for Navier-Stokes Flow Calculations by a Line Iterative Method," *AIAA Journal*, 23, 1847-1848, 1985.

[15] W. Shyy and M.E. Braaten, "Three-Dimensional Analysis of the Flow in a Curved Hydraulic Turbine Draft Tube," *Int. J. Num. Meth. Fluids*, 6, 861-882, 1986.

[16] M.E. Braaten, "Development and Evaluation of Iterative and Direct Methods for the Solution of the Equations Governing Recirculating Flows," PhD Thesis, Department of Mechanical Engineering, University of Minnesota, Minneapolis, MN, 1985.

[17] T.F. Chan and R.S. Tuminaro, "Implementation of Multigrid Algorithms on Hypercubes," RIACS Technical Report 86.30, NASA Ames Research Center, Moffett Field, CA, 1986.

[18] M.E. Braaten and W. Shyy, "Study of Pressure Correction Methods with Multigrid for Viscous Flow Calculations in Nonorthogonal Curvilinear Coordinates," *Num. Heat Trans.*, 11, 417-442, 1987.

Parallelizing An Electron Transport Monte Carlo Simulator (MOCASIN 2.0)

Herb Schwetman and Stephen Burdick

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, TX 78759

Abstract -- Electron transport simulators are important tools for studying electrical properties of semiconducting materials and devices. As demands for modeling more complex devices and new materials have emerged, so have demands for more processing power. This paper documents a project to convert an electron transport simulator (MOCASIN 2.0) to a parallel processing environment. In addition to describing the conversion, the paper presents PPL, a parallel programming version of C running on a Sequent multiprocessor system. In timing tests, models that simulated the movement of 2,000 particles for 100 time steps were executed on ten processors, with a parallel efficiency of over 97%. In this revision to MCC Technical Report ACA-ST/CAD-328-87, an additional table has been added to explain the apparent discrepancy in the timing results found in Table 1.

Introduction

Device simulators play an important role in the characterization and advancement of semiconductor technology. They are used to characterize and predict the electrical behavior of various devices, such as transistors and diodes, fabricated from different materials including germanium, silicon and gallium arsenide. Currently, the most common device simulators are based on the drift-diffusion approximation [7, 8, 12]. Drift-diffusion based simulators enjoy the properties of familiarity (due to widespread use), rapid execution (due to the simplifying approximations made about the physical properties of the devices), and reliable predictions for a large class of problems. Unfortunately, the drift-diffusion approximation becomes less accurate as device sizes shrink, as the electric field inside of the devices becomes stronger, and as the electric field varies rapidly in space or time. These trends are becoming more evident as semiconductor technology advances.

Device simulators that model transport of a carrier (an electron or a hole) at the scattering level and that use Monte Carlo simulation remove several of the assumptions made in the drift-diffusion simulators about device behavior [5, 6]. This new form of simulator has several advantages over the drift-diffusion based simulators, including: (1) they can be easily modified to include new scattering processes; (2) they can model intrinsic noise phenomena; and (3) they represent the physical behavior more accurately. However, a disadvantage is that it can require enormous amounts of CPU time; runs requiring hours or days of computing time on a Sun 3/260 workstation are common.

At MCC, a program to model electron transport in planar MESFET's (MEtal Semiconductor Field Effect Transistors) was developed as part of the Computer Aided Design (CAD) project. This program, named MOCASIN 2.0 [2], was implemented for the the Apollo 660 workstation and the Sun 3 workstation [BuBl86] over a two year period. In the middle of 1987, a collaboration between the CAD project and the Parallel Processing project at MCC led to the development of a parallel version of MOCASIN 2.0 which runs on the Sequent Balance multiprocessor system. This paper describes the the implementation of parallel MOCASIN 2.0, with emphasis on the changes required to convert the serial version to the equivalent parallel version. A key factor in facilitating this conversion was PPL [9, 10, 11], a superset of C with enhancements to facilitate parallel processing.

The Simulator - MOCASIN 2.0

MOCASIN 2.0 is a two-dimensional, scattering-level transport simulator for planar MESFET's fabricated from III-V compounds. The program places charged particles (representing the carriers) in a rectangular simulation region (representing the MESFET) and solves for the electric field due to charged particles, ionized dopants, and boundary conditions. The particles are moved according to the local electric field and scattered according to local material properties. After each particle has been moved for a very short time interval (5×10^{-14} seconds), the electric field is updated. These two steps - field determination and particle transport - are repeated until the device behavior has been simulated for the specified time interval.

The carrier transport process also consists of two stages. In the first stage, each particle is moved under the influence of the local electric field for a randomly selected length of time. During the second stage, a scattering process is randomly selected from a set of possible processes, and the particle's energy-momentum state is randomly altered according to the selected scattering process.

This procedure is very general and uses the more accurate approximations mentioned above. In particular, drift-diffusion based simulators assume that the carriers instantaneously reach a mean velocity determined by the local electric field. This assumption prevents drift-diffusion based simulators from accurately predicting the electrical behavior of submicron sized MESFET's in Gallium Arsenide. Scattering-level based transport simulations, such as MOCASIN 2.0, model the experiment more accurately, but they require much more CPU time.

To address the problem of long running times, a project was established to implement a parallel version of the simulator. The first step of this project was to select a portion of the simulator code to execute in parallel. The next steps were to define a process structure and redefine the data structures for parallel execution. After implementation of the parallel version, tests were conducted, to verify correct operation and to measure the impact of the changes.

The section of the simulator chosen as the portion of the simulator to be converted to parallel execution was the particle transport section, for three reasons:

- it was easily isolated,
- it consumed a major percentage of the CPU time in the serial version, and
- the movement of each particle is independent of the movement of all of the other particles.

As will be seen later on, this selection was fortuitous.

Once this selection was made, MOCASIN 2.0 was moved to the parallel environment used for this project. This environment was the Sequent Balance 8000 multiprocessor system [13], together with PPL, a parallel programming language based on C. PPL had been developed at MCC for the Balance system and, based on prior experiences [4], seemed to be a good choice for the conversion project. The remainder of this paper gives a brief overview PPL and then details the major changes required to arrive at a parallel version of the simulator.

Overview of PPL

As stated above, PPL is a superset of C. The extensions all deal with creating and controlling multiple (parallel) processes within a single C program. The primary construct is the PPL process. PPL processes correspond to tasks or microtasks in some other parallel environments. The underlying runtime support package uses UNIX† processes (created using the *fork* system call) to serve as "workers". In the remainder of this paper, the term "process" will refer to a PPL process, and the term "worker" will refer to a UNIX process which is associated with a processor. PPL processes which are eligible to execute are placed on a single "ready-to-run" queue. As processes are placed on this queue, the idle workers are stimulated to examine this queue; one of the idle workers will remove the next eligible process and start its execution. PPL is designed to encourage the use of processes, and it completely hides the existence of the workers. The only control a programmer has over the workers is to specify the number of workers dedicated to the program. Each worker uses one physical processor, so the number of workers corresponds to the maximum level of parallelism which could be exploited by the program. A PPL program can have many simultaneously active processes with only a few workers. In fact, executing a PPL program with only one worker assigned (and consequently with all processes executing sequentially on one processor) is a useful debugging technique.

Processes are created dynamically. There are no implied relationships between processes. Processes can terminate any time after they are created. In addition, they can suspend execution, to await the occurrence of an event caused by another active process. Processes can initiate additional instances of themselves or other processes. The input arguments and variables local to a process are automatically preserved as the process progresses through alternating periods of being active and then being suspended. Processes can call functions and procedures, just as normal C procedures do. The PPL programmer is able to concentrate on implementing the parallel program, while the details of process management and process scheduling are handled by procedures in the PPL runtime library.

In an environment with simultaneously active processes, there must be mechanisms for synchronizing the activities of these processes. Often, these synchronization mechanisms are used to guarantee that only one process at time is active in a critical section of the code. A classic example would be updating a variable which is located in shared memory (see below). PPL supports two kinds of synchronization operations: those based on suspending and resuming processes and those which use a "busy wait". The first kind has the advantage that while a process is in the suspended state in some queue of suspended processes, the worker it was using can be assigned to another eligible process. However, there is some processing overhead required to place a process in this queue and to start another process. The second kind has very low overhead, but it does "waste" a worker while the process is waiting for something to happen. PPL programmers are encouraged to use the suspend-resume form of synchronization if there is a high probability of being forced to suspend or if the process will be in the critical section for a long period of time. The busy-wait form is appropriate if there is a low probability of being forced to wait or if the time in the critical section will be short.

While PPL has several different kinds of synchronization mechanisms, the only ones used in the parallel version of MOCASIN 2.0 are *locks*, a *mailbox*, and a *counter*. On the Sequent, locks are busy-wait mechanisms which are implemented using the test-and-set-bit instruction and an array of lock bits. PPL provides statements to declare and initialize locks and to lock and unlock these locks. Mailboxes and counters are both suspend-resume synchronization mechanisms. A mailbox is used to coordinate the exchange of messages between processes. A process can *send* a message (either an number or a string of bytes) to a mailbox. Another process can do a *receive* operation on that mailbox. If there is a pending message, the receiving process will consume the message and continue; if there are no pending messages, the receiving process will be suspended, until a message arrives at the mailbox. A counter is often used by a parent process to wait until

† UNIX is a trademark of Bell Laboratories.

all of the child processes have completed their processing. A counter used in this fashion is initialized to an integer value (the number of children to be initiated). After the children are initiated, the parent suspends by performing a *c_wait* operation on the counter. As each child process completes, it does a *c_set* operation on the counter. As the last child does this, the parent process resumes and continues execution.

In the parallel version of MOCASIN 2.0, several locks are used to guarantee that only one process at a time updates a variable in shared memory. A mailbox is used to implement a queue of particles which need to be moved during a time step, and a counter keeps up with the number of moved particles, so that the parent can resume processing after all of the particles have been moved in a single time step.

The Sequent system provides a (logical) segment of memory which can be shared by all of the processes associated with a program. The C compiler provided by Sequent has been modified, to accept *shared* as part of the declaration of a variable. Variables declared as shared are located in the shared segment and can be accessed by PPL processes. Thus, there are three kinds of storage which can be used by PPL programs: storage which is globally defined and located in the shared segment, storage which is globally defined, but located in the private segment of each worker, and storage which is automatically allocated on the runtime stack as each process or procedure is entered. Placing data in the correct class of storage is a crucial consideration in the design and implementation of a parallel program.

The Parallel Version of MOCASIN 2.0

As stated earlier, it was decided to parallelize that portion of MOCASIN 2.0 which moved the particles within a time step, and to leave the portions dealing with determining the electric field and advancing to the next time step in serial mode. It was hoped that by restricting the portion to be modified, the anticipated performance enhancements could be quickly obtained.

MOCASIN 2.0 can be viewed as a main program that contains a time-step for-loop. (See Figure 1). This outer loop steps time from zero to the final simulation time. The electric field is determined at each time step as a function of the charge locations and boundary conditions. Then each particle is moved in the local electric field. In the serial version, this was accomplished with a particle for-loop that steps through an array containing each of the simulated particles. This particle loop was the only section of MOCASIN 2.0 to be parallelized. The particle loop was parallelized by sequentially assigning the movement of each particle in turn to the next free processor. The lack of interaction between particles during the movement phase vastly simplified the serial-to-parallel conversion.

```
MAIN:
  FOR EACH TIME STEP {
    DETERMINE ELECTRIC FIELD;
    MOVE_PARTICLE PROCEDURE {
      FOR EACH PARTICLE
        PERFORM ONE MOVE STEP;
      }
    }
```

Figure 1
Structure of MOCASIN 2.0.

Two kinds of changes were made to parallelize MOCASIN 2.0. One was to convert from serial execution to parallel execution, to take advantage of the architecture of the Sequent. These modifications were accomplished using the constructs provided by PPL [9]. The parallel version was modified still further, to insure that repeated simulations of the same model would yield identical results. The possibility for different (but correct) results comes from the use of a single stream of random numbers in the serial version. In the parallel version, it is likely that particles would be moved in different ways on repeated executions of the program, because different sequences of random numbers would control each particle on successive runs. An alteration of the parallel version (see below) removed this kind of variability and guaranteed repeatable behavior. While not essential to correct operation of the model, these changes greatly simplified verification of the parallel version. Both kinds of changes could be implemented and tested on a single processor machine, in the PPL environment.

As indicated in Figure 2, the user-requested number of *mover* processes is created prior to entering the time step loop. Then, the time-step loop is entered and, after the electric field is determined, the mailbox is filled by sending it a sequence of *messages*, each containing the index of a particle that must be moved. Control is turned over to the mover processes until they signal completion. This parallel phase replaces the serial for-loop control structure. When all of the particles have been moved, the main procedure regains control and the next time step is started.

Another change required for parallelization was to insure that all variables that were being altered by the particle moving processes were located in shared memory. Some of the data structures in MOCASIN had to be relocated to shared memory, to enable correct operation of the parallel version. Conflicts can arise when multiple mover processes try to simultaneously modify data values in that part of main memory which is shared by all of the processes. These conflicts are resolved by using *locks*.

Some additional changes were required, to insure that the parallelized version of MOCASIN 2.0 would give reproducible results for different runs that used the same random number seed. There were two such

changes. The first was to associate an individual random seed with each electron being simulated. With this change, each particle was guaranteed to be moved in the same way, regardless of the temporal ordering of events in successive runs of the program. This is similar to the technique reported by Fredrickson, et al [3].

```
MAIN:
   CREATE MOVER PROCESSES; (one per processor)
   FOR EACH TIME STEP {
      DETERMINE ELECTRIC FIELD;
      MOVE_PARTICLE PROCEDURE {
         FOR EACH PARTICLE
            SEND MESSAGE (particle index);
         WAIT FOR COMPLETION
            OF ALL PARTICLES MOVES;
         }
      }

MOVER PROCESS:
   DO FOREVER {
      RECEIVE MESSAGE (particle index);
      PERFORM ONE MOVE STEP;
      SIGNAL COMPLETION OF ONE PARTICLE;
      }
```

Figure 2
Structure of Parallel Version of MOCASIN 2.0.

The second change was to modify the procedure used to delete particles that left the simulation region through a contact. In the serial version, each particle was deleted from the array of active particles and replaced with a particle from the end of the array, which kept the array in a compact form. When executing in parallel, this technique would have meant that the order of the particles in the resulting array would have been different each time the parallel version was executed. The parallel version was changed so that as a particle was deleted from the region, its index was added to a list. At the end of the moving phase, this list was sorted, the particles then deleted in order, and the "holes" in the array filled as before. In addition to preserving the order of the particles in the array, this technique was required for two other reasons: (1) using the list allowed the program to postpone deleting particles until the end of the transport phase so that the size and structure of the array did not change during the parallel phase of execution, and (2) sorting the list of deleted particles meant that they would be removed in descending order by indices and the delete procedure from the serial version could be reused.

Timing Results

Correct operation of the parallel version of MOCASIN 2.0 was verified by comparing the results with those from the original serial version. The new results were not identical, because of the different ordering of movements caused by the new procedure for generating random numbers. However, these results where judged to be valid, based on a thorough analysis of them. Next, performance of the parallel version was measured using "N" processors, with N ranging from 1 to 10. These tests simulated the same MESFET with two different sets of parameters [2], as follows:

- A short set of parameters, with 100 electrons, for 10 time steps, and
- A long set of parameters, with 2000 electrons, for 100 time steps.

As mentioned earlier, the system used in the project was the Sequent Balance 8000, at MCC. This system has 12 CPU's; each CPU has a NS32032 (10Mhz) processor, a floating point coprocessor, a memory management unit, and 8 kilobytes of local cache. The size of the main memory is 16 megabytes, and all of this memory can be accessed by any processor. The local caches employ a write-through strategy, and special circuitry insures that the contents of each cache remain consistent, even when cached data is being updated by another processor.

Some results from the experiments on the Sequent are shown in Table 1. The speed up factor in Table 1 is the elapsed time with N = 1 processors divided by the elapsed times for other values of N.

	Short Problem		Long Problem	
	Elapsed Time	Speed-up	Elapsed Time	Speed-up
N = 1	991	1.0	171,422	1.0
N = 2	523	1.9	85,310	2.0
N = 4	294	3.4	43,334	3.9
N = 8	181	5.5	21,943	7.8
N = 10	159	6.2	17,512	9.7

Table 1
Elapsed Times (Seconds) and Speed-up Factors on Sequent

A common way of evaluating parallel speed up is to compare the elapsed times of the parallel version to the elapsed time of the purely serial version. When this was done with the two versions of MOCASIN 2.0, the serial version required more time than the parallel version with one processor (by about 10 to 20 per cent). This result was counterintuitive and more studies were performed, to try to understand the cause. Additional analysis pointed out that the user mode times for the serial version and the parallel version were about the same, but that the system mode time for the serial version was much greater than for the parallel version. Some detailed runtime statistics showed that the number of page reclaims (page faults which were satisfied without doing any I/O) for the serial version was nearly a factor of between 5 and 10 greater than the number of page reclaims in the parallel version. In addition to introducing parallel processes, the parallel

version also used shared memory, while the serial version used unshared memory. The data in Table 2 summarizes these findings. It is fair to conclude that in the version of the Sequent operating system being used for these tests, page replacement is biased in favor of keeping pages in shared memory over keeping pages in unshared memory, to such a degree that comparisons of running times for programs using shared memory and programs using nonshared memory can be invalid.

	Elapsed Time	User Time	System Time	Page Reclaims
Serial	1070	937	128	130,126
N = 1	991	963	24	15,767
N = 2	523	1010	20	14,769
N = 4	294	1111	28	18,931
N = 8	181	1323	39	18,327
N = 10	159	1434	45	16,861

Table 2
Times (Seconds) and Page Faults for Short Problem

Summary

The goals of the project were met. First, a C program written for a single processor system was quickly (about two man weeks) converted to run on a multiprocessor system. Second, measurements made on the parallel version showed that this program could make very efficient use of multiprocessor system. That fact that the program, executing the large model, ran 9.7 times faster on a 10 processor system demonstrates efficient performance.

The project, while achieving the stated goals, also raises some additional issues. For example, how critical is the fact that in this problem, the particle movements were all independent from each other (in the same time step)? Could an equally efficient implementation be done, even when there are some dependencies between particles as they are moved? Another issue involves the *style* of the parallel version of MOCASIN 2.0. The current parallel version uses the simple technique of sending a message for each particle which needs to be moved. There are other techniques, such as assigning blocks of particles to each of the *mover* processes at each time step, which would eliminate the need for sending 2000 messages (in the long problem) at each time step. Also, there are probably other approaches which would eliminate the sort (of the deleted particles) at the end of each time step while maintaining a reproducible set of results. Such a change could reduce the CPU time required at each step.

Finally, there was no attempt to extrapolate the performance efficiencies which would be obtained with increasing numbers of processors. The current project was limited to ten processors by the system being used. It would be interesting to continue the study by increasing the number of processors. In the extreme case, how would a 2000 processor system handle this problem? How does efficiency "tail off" as the number of processors increases?

The current project was important in one other respect, namely it provided a concrete example of a useful parallel program. The practicality of parallel programming, to solve scientifically interesting problems, has not yet been generally established. This project is another step in the process of discovering both the usefulness and the limitations of parallel processing.

List of References

[1] Burdick, S.A. and P. Blakey, "MOCASIN 1.0 User's Manual", *MCC Technical Report*, CAD-028-87, Feb. 1987.

[2] Burdick, S.A., "MOCASIN 2.0 User's Manual", *MCC Technical Report*, CAD-303-87, Sept. 1987.

[3] Frederickson, P., Hiromoto, R. and J. Larson, "A Parallel Monte Carlo Transport Algorithm Using a Pseudo-random Tree to Guarantee Reproducibility", *Parallel Computing*, (4), North-Holland, pp. 281-290, 1987.

[4] Glenn, R. and H. Schwetman, "The Performance of the Burstein Hierarchical Router Using MCC's Parallel Programming Language on the Sequent Balance B-8000", *MCC Technical Report*, PP-322-86, August 29, 1986.

[5] Moglestue, C., "Monte Carlo Particle Simulation of a GaAs MESFET with Gate Trench Sloping Towards the Source", *IEE Proceedings,* (132), 1985, p. 217.

[6] Moglestue, C., "A Self-Consistent Monte Carlo Particle Model to Analyze Semiconductor Microcomponents of any Geometry", *IEEE Transactions on Computer Aided Design,* March, 1986, pp. 326 - 345.

[7] Pinto, M., "Two-Carrier Simulation of Complex Device Structures in Minicomputer Environments", *2nd SIAM/IEEE Conference on Numerical Simulation of VLSI Devices,* Boston, MA, November 13, 1984.

[8] Price, C., *Two-Dimensional Numerical Simulation of Semiconductor Devices,* Ph.D. Thesis, Stanford University, 1982.

[9] Schwetman, H.D., "PPL: A Parallel Programming Language Based on C", *MCC Technical Report*, PP-096-86, March 1986.

[10] Schwetman, H.D., "PPL Reference Manual", *MCC Technical Report*, PP-219-86, July 1986.

[11] Schwetman, H., "Parallel Programming with PPL", *MCC Technical Report*, ACA-ST-007-88, January, 1988.

[12] Selberherr, S. and A. Schutz, "MINIMOS - A Two-Dimensional MOS Transistor Analyzer", *IEEE Transactions on Electron Devices,* (27), 1980, pp. 1540 - 1550.

[13] Sequent Computer Systems, Inc., *Balance 8000 System Technical Summary,* Order No. MAN-0110-00, 1984.

[14] Sequent Computer Systems, Inc., *Balance 8000 Guide to Parallel Programming,* 1003-41030 Rev. A, November 26, 1985.

PROTEIN SEQUENCE COMPARISON ON A DATA PARALLEL COMPUTER

Eric Lander[†]
Whitehead Institute for Biomedical Research
9 Cambridge Center, Cambridge, Massachusetts 02142
and
Harvard University
Cambridge, Massachusetts 02138

Jill P. Mesirov
Washington Taylor IV
Thinking Machines Corporation,
245 First Street, Cambridge, Massachusetts 02142-1214

(May 4, 1988)

Abstract

In this paper we discuss the issues involved in implementing a general dynamic program on a data parallel computer to compare proteins for good subsequence matches, based on a variety of scoring metrics. A standard serial algorithm can be optimally parallelized. Careful allocation of machine resources has enabled us to compare an entire database of 2000 proteins against itself in about the same time that it would take to run one protein against the database using conventional computers. The results gleaned from this program provide information about scoring metrics and allow clustering of groups of related proteins. This information can be of assistance in determining the biochemical function of some proteins.

[†] Eric Lander was supported in part by National Science Foundation grant #NSF-DCB-8611317 and System Development Foundation grant #SDF612

1 Introduction

In many fields, including molecular biology, speech recognition, and cryptology, there are problems whose solutions involve comparing sequences of symbols to determine correlations between them. Often, this task involves finding an optimal alignment of the sequences in question, which may require introducing gaps into one or both of these sequences. One way of measuring the optimality of an alignment is by computing a score based on a matrix of weights reflecting the similarity between pairs of symbols. In some situations a penalty is subtracted for each gap introduced. Such a score can be computed by a dynamic programming algorithm in time proportional to the product of the lengths of the sequences [1,12].

In biology, this technique is particularly useful. Although it is possible to determine the amino acid sequence for virtually any protein, there is no general method for determining the protein's biochemical function from this information alone. One of the most successful approaches has been to find a similarity between a subsequence of a newly sequenced protein and one of a protein of known function [5]. However, because comparison of a single protein to a database of 4,000-5,000 proteins using the complete dynamic programming algorithm can take several hours on conventional computers, biologists frequently employ more approximate, heuristic methods [11]. Also, because of the computational difficulty of performing exhaustive studies of many alternative scoring systems, the few scoring systems that are typically used are relatively ad hoc. Only a few computationally intensive studies, using a single scoring system, have been done [3,14].

In order to examine the methodology underlying these scoring systems more completely and to initiate a systematic search through the databases for statistically significant correlations between pairs or groups of proteins, we have implemented a parallel version of this dynamic programming algorithm on the data parallel CM-2 Connection Machine® system. Through an analysis of the data layout and careful coding of the inner loop we have written a program which enables us to run every protein in a database of 2000 proteins against the entire database in under an hour.

In this paper, we discuss the implementation issues involved in this application. We also mention preliminary results from our systematic investigation of some scoring systems. Section 2 discusses the architecture of the CM-2 Connection Machine system, Section 3 reviews the basic dynamic programming algorithm for subsequence comparison, and Section 4 indicates how it can easily be parallelized. Section 5 contains a detailed description of the implementation on the CM-2, Section 6 discusses generalizations of the algorithm in Section 3, and Section 7 contains timings and a brief description of some of the results obtained using this software.

2 The CM-2 Architecture

The algorithms for this project were programmed on the Connection Machine CM-2 data parallel computer [4,7]. The CM-2 is composed of a microsequencer and a maximum of 64K single-bit processing elements. The processors run in SIMD mode, with the instruction stream broadcast by the sequencer. The sequencer is controlled by an external front end machine, usually a SYMBOLICS® Lisp Machine, or VAX® . Parallel extensions of familiar programming languages allow the user to program the front end to do serial computation, with an expanded instruction set providing access to the "data parallel" part of the system. Each processor is associated with 64K

bits of RAM, and there is a single high-speed floating point unit for every 32 processors. The processors are connected in a boolean n-cube topology, however the communication system is general enough that an arbitrary connection scheme can be imposed by the user. Thus, data can be rapidly exchanged between the memories of different processors as necessary to complete any computation. One very communication efficient way to configure the machine is as a k dimensional grid, which is automatically superimposed by the system software onto the boolean cube using a multidimensional gray-code mapping. In this paper we will deal with the CM-2 as if the processors were configured in a 1-d grid. In this situation every processor has a unique identifying coordinate denoted by p, such that processor p only communicates with processors $p-1$ and $p+1$, where p ranges from 0 to $N-1$, where N is the number of processors in the CM-2.

3 The Subsequence Matching Problem

The subsequence problem can be formulated as follows:

Given two sequences A, B, of symbols chosen from a domain $\mathcal{F}$

$$A = (a_1, a_2, \ldots, a_n),\ B = (b_1, b_2, \ldots, b_m), \qquad a_i, b_j \in \mathcal{F},$$

find the subsequences

$$A' = (a_{i_1}, a_{i_2}, \ldots, a_{i_z}),\ B' = (b_{j_1}, b_{j_2}, \ldots, b_{j_z})$$

[where $1 \le i_1 < i_2 < \ldots < i_z \le n,\ 1 \le j_1 < j_2 < \ldots < j_z \le m$] which maximize the comparison function $C(A', B')$. C can depend on the symbols a_{i_l}, b_{j_k} in A' and B' and on the numbers of symbols in A and B which are omitted between successive symbols in A' and B' (gaps).

We denote by $\sigma_l = i_{l+1} - i_l - 1, \tau_k = j_{k+1} - j_k - 1$, the "gap" sizes between a_{i_l} and $a_{i_{l+1}}$, b_{j_k} and $b_{j_{k+1}}$ respectively, for $1 \le l, k < z$. Let $A' \prec A$ denote that A' is a subsequence of A, and define $A_{(j)}$ to be the subsequence of the first j elements of A, i.e., $A_{(j)} = (a_1, \ldots, a_j)$.

In this paper, we consider primarily scoring systems which are defined recursively as follows:

$$\begin{aligned} C(A', B') &= C(A'_{(z)}, B'_{(z)}) \\ &= C(A'_{(z-1)}, B'_{(z-1)}) + D(a_{i_z}, b_{j_z}) + g \cdot (\sigma_{z-1} + \tau_{z-1}) \\ &= \sum_{k=1}^{z} D(a_{i_k}, b_{j_k}) + g \cdot (i_z - i_1 - z + j_z - j_1 - z + 2), \end{aligned}$$

where the gap constant $g < 0$, and D is a correlation function between single elements of $\mathcal{F}$.

For such comparison functions, one can use a dynamic programming algorithm to determine the best subsequence match for a given pair of sequences A, B in serial time $O(mn)$ where n and m are the lengths of the sequences A and B.

This dynamic programming algorithm can best be understood by considering the matrix

$$C_{r,s} = \max\{C(A', B') + g \cdot (r - i_z + s - j_z) \mid i_z \le r, j_z \le s\},$$

Where max is taken across all $A' = (a_{i_1}, \ldots, a_{i_z})$, $B' = (b_{j_1}, \ldots, b_{j_z})$.

It is then clear [13] that

$$C_{r,s} = \max \begin{cases} 0 \\ C_{r-1,s-1} + D(a_r, b_s) \\ C_{r-1,s} + g \\ C_{r,s-1} + g \end{cases}$$

Thus, to compute $\max_{A',B'}\{C(A', B') \mid A' \prec A, B' \prec B\}$, one need only compute $\max C_{r,s}$ where the matrix elements $C_{r,s}$ can be computed inductively in time $O(mn)$.

4 Parallel Implementation

A parallel version of the dynamic programming algorithm is quite straightforward to derive [see for example [6]]. Since computing the value of $C_{r,s}$ only depends on knowing the values of $C_{r-1,s}, C_{r,s-1}$, and $C_{r-1,s-1}$, we see that all of the elements on one anti-diagonal of the matrix can be computed simultaneously if the values along the two previous anti-diagonals are known. That is, for a fixed value of t, the matrix elements $C_{t-s,s}$ can be computed simultaneously for all s provided that they are known for $t-1$ and $t-2$. Thus, one can parallelize the above algorithm by computing successive anti-diagonals of the matrix $C_{r,s}$ on successive time steps. This is represented schematically in Figure 1. The algorithm requires $n+m-1$ time steps and m processors to compare proteins of length m and n. If the proteins are ordered so that $m \le n$, then we obtain optimal speedup since $O(m)O(n) = O(mn)$.

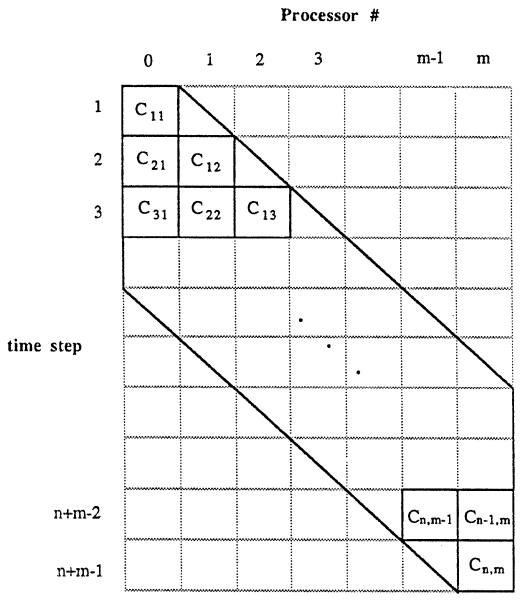


Figure 1: Diagram indicating activity of processor p at time step t. If $1 \le t\text{-}p \le n$, then processor p computes $C_{t-p,p+1}$ at step t. Otherwise, the processor is inactive.

5 CM-2 Implementation

The basic goal was to develop software which would calculate the scores between every pair of proteins in a given database according to an arbitrary scoring metric of the kind described above. Since many individual pairwise comparisons have to be done for a single database search, it is worthwhile to efficiently pack the data so that the overall time spent is minimized. (Note that here $\mathcal{F}$ is the set of all amino acids with several extra symbols added to denote unknowns. Since this domain consists of only 23 distinct symbols, we can represent each as a five bit integer, ranging from 1 to 23.)

5.1 Data Mapping

Since the CM-2 has far more processors than most proteins have amino acids, it is possible to perform the dynamic programming algorithm on many proteins at the same time. To this end, the proteins in the chosen data base are first sorted by length and then partitioned sequentially into sets S_d such that the total number of amino acids in the proteins of each S_d is as large as possible but is less than or equal to the total number of processors available in the CM-2. The proteins in each set S_d are then compared in parallel to the entire database, one protein at a time.

We denote the nth smallest protein in the database by P_n, and its length by $l(P_n)$. If the CM-2 we are working on has N processors, then we set $S_d = \{P_{\alpha_{d-1}}, \ldots, P_{\alpha_d - 1}\}$, where $\alpha_0 = 1$, and

$$\sum_{a=\alpha_{d-1}}^{\alpha_d - 1} l(P_a) \leq N < \sum_{a=\alpha_{d-1}}^{\alpha_d} l(P_a).$$

is used to determine α_d inductively from α_{d-1}.

We will denote the ith protein in S_d by P_i^d, i.e.,

$$P_i^d = P_{\alpha_{d-1}+i-1},$$

and the qth amino acid in this protein by $P_i^d(q)$.

If M is the total number of such sets S_d, and K is the total number of proteins in the database, then the basic structure of the serial part of the program to compare all pairs of proteins is:

for *d = 1* **to** *M* **do**
  set up S_d *on CM-2*
  for *k = 1* **to** *K* **do**
    compare in parallel all proteins in S_d *to* P_k
    retrieve results and store on front end

where setting up S_d involves laying out the amino acids in the proteins in S_d in linear fashion into the memory of the processors with the CM-2 configured as a 1-d grid as described in Section 2 [see Figure 2]; the inner loop which compares S_d to P_k is implemented using the parallelized dynamic programming algorithm just described. Thus, on each pass through the inner loop, all the proteins in S_d will be compared simultaneously with some protein P_k which we will designate henceforth as a "target" protein. We will use p_i^d to denote the coordinate of the processor containing $P_i^d(1)$. Thus,

$$p_i^d = \sum_{j=1}^{i-1} l(P_j^d).$$

We make the following comments:

1. Since in practice the correlation function D is usually symmetric, in order to compare all pairs of proteins in the database it is only necessary to compare the proteins in S_d to the proteins in all $S_{d'}$ where $d' \geq d$. This is more efficient than using the proteins in $S_{d''}$ where $d'' \leq d$ as target proteins since this uses the longer proteins as target proteins as recommended in Section 4.

2. It is preferable not to have large variations in the lengths of the proteins in a set S_d. Observe that the time needed to compare S_d to a protein P is

$$\begin{aligned} T(d,P) &= l(P) + \max\{l(P') \mid P' \in S_d\} - 1 \\ &= l(P) + l(P^d_{last}) - 1. \end{aligned}$$

   This was the motivation for sorting the database before partitioning into sets S_d. Even though this does not necessarily yield an optimal packing of the proteins into groups of bounded size, in most situations it seems to do well at optimizing $\Sigma T(d,P)$, which is the value we really want to minimize. This heuristic is related to the First Fit Decreasing algorithm for bin packing, which can be shown to come within a factor of 11/9 of the minimal number of S_d needed [9].

3. From Figure 1, it is clear that processors are idle some fraction (less than half) of the time. One might try to reduce the number of cycles in which processors are idle by overlapping comparisons with consecutive target proteins. Although this might give a slight performance improvement, the increased complexity of the inner loop code makes it unclear what, if any, gain would be made.

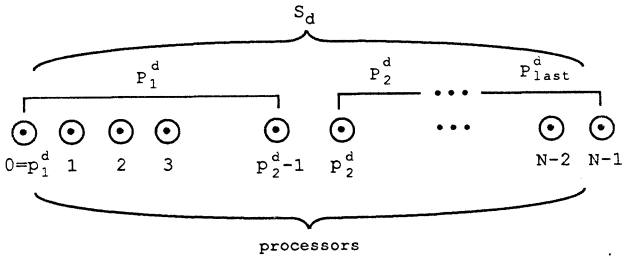


Figure 2: Diagram indicating mapping of amino acids in proteins in $S_d = \{P_1^d, P_2^d, \ldots, P_{last}^d\}$ onto CM processors.

5.2 Inner loop implementation

In this section we will describe the actual Connection Machine implementation of the parallel algorithm for the inner loop of our code. We assume that the proteins in some set S_d have been put into the CM-2's memory according to the scheme described in the previous section. For a given target protein P_k, at each iteration step t the value of $C_{t-q,q}(P_k, P_i^d)$ is computed by processor $p = p_i^d + q - 1$ We will denote a parallel variable *foo* by *foo*[], where *foo*[p] is the value of the variable *foo*[] in processor p. $foo_t[p]$ is the value of *foo*[p] at the end of iteration t. Thus, we can describe the data from S_d as being stored in a parallel variable (also called a "pvar") SD[], so that

$$\text{SD}[p_i^d + q - 1] = P_i^d(q),$$

recalling that $P_i^d(q)$ is the q*th* amino acid in P_i^d in its five bit integer form. At the time of initialization of SD[] for S_d we also initialize a boolean pvar H[] marking the beginning of each protein, i.e.,

$$\text{H}[p] \leftarrow \begin{cases} 1 \Leftrightarrow \text{processor } p \text{ contains } P_i^d(1) \text{ for some } i \\ 0 \text{ otherwise} \end{cases}$$

The target protein P_k is stored in the memory of the front end machine. The amino acids from the target protein will be sent from the front end to the Connection Machine memories one per time step. They will shift across a pvar TGT, and thus be aligned with the amino acids sequences held in the pvar SD, so that the comparisons can be performed [see Figure 3].

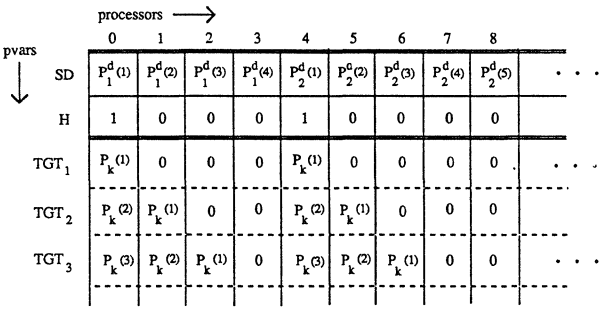


Figure 3: Data Layout for the first three iterations of the inner loop for a set of proteins in a typical S_d, where length(P_1^d)=4.

We will make use of the following pvars in the code

SD holds the amino acid sequences for the proteins in S_d as described above.

H indicates the beginning of each protein in SD.

CNOW$_t$ gets the values of the C matrix on the anti-diagonal currently being computed, i.e., $C_{t-q,q}$. (It starts with the values from the previous row of C, $C_{t-q-1,q}$.)

CLAST$_t$ gets the values of the C matrix entries from the previous column of C, i.e., $C_{(t-1)-(q-1),q-1} = C_{t-q,q-1}$.

CDIAG$_t$ gets the values of the C matrix entries from the 2nd previous anti-diagonal of C, i.e., $C_{(t-2)-(q-1),q-1} = C_{t-q-1,q-1}$.

CMAX$_t$ gets the cumulative maximum score for the corresponding column of the C matrix, i.e., $\max_{t' \le t} C_(t' - q, q)$.

TGT holds the shifting copies of the amino acid sequence for P_k

The code for the inner loop now proceeds as follows:

```
        /* initialize pvars */
(1)     CLAST[p] ← 0
        CNOW[p] ← 0
        CMAX[p] ← 0
        TGT[p] ← 0
    /* compute score matrix C */
    for t = 1 to T(d, P_g) do
(2)     TGT[p] ← TGT[p − 1]
(3)     where H[p] = 1,
(4)           TGT[p] ← P_k(t)
(5)     CDIAG[p] ← CLAST[p] + D(SD[p], TGT[p])
(2)     CLAST[p] ← CNOW[p − 1]
        CNOW[p] ← max{0, CLAST[p] + g, CNOW[p] + g,
                      CDIAG[p] + D(SD[p], TGT[p])}
        CMAX[p] ← max{CMAX[p], CNOW[p]}
(6) CMAX[] ← max-scan (CMAX[]), with segmentation pvar H[]
```

At the completion of the algorithm, we assert that:

$$\text{CMAX}[p_{i+1}^d - 1] = \max\{C(A', B') \mid A' \prec P_i^d, B' \prec P_k\},$$

so the results of the comparison can be directly retrieved from CM-2 memory and stored on the front end for later use.

Algorithm notes:

1. All assignment statements are assumed to be carried out in all processors, unless otherwise indicated. Thus *foo*[p] ← 0 means that the pvar *foo*[] is given a value of 0 simultaneously in all processors.
2. Assignment statements involving pvar data in different processors are accomplished using 1d grid communications.
3. The **where** command evaluates some expression in every processor simultaneously, and only carries out the consequent operation in those processors in which this logical expression is true.
4. We assume that $P_k(t) = 0$, for $t > l(P_k)$.
5. The evaluation of the function D(SD[p], TGT[p]) was implemented using an indirect table lookup, where the 5-bit bytes SD[p] and TGT [p] are concatenated into a 10-bit word used as an offset into a locally stored table. Setting $D(0, a) = D(a, 0) = 0$ for all a ensured that the values of C computed in processors during "inactive" time steps of were less than the maximum valid value of C computed in the same processor during the same comparison. Thus, it was not necessary to explicitly determine which processors were active at each time step.

6. The scan operation is a primitive used in many parallel algorithms [2,8,10]. Scans can be defined as follows: given an associative operator $\oplus$ ($(a \oplus b) \oplus c = a \oplus (b \oplus c)$), the function $\oplus$ *-scan* ($f[]$) returns a pvar $z[]$ whose value at processor p is given by

$$z[p] = f[1] \oplus f[2] \oplus \ldots \oplus f[p].$$

So-called "segmentation" bits are boolean pvars used to break up the scan into disjoint segments, over which the scan is independently performed. For example,

$$z[p] \leftarrow \oplus\text{-}scan\ (f[])\ with\ segmentation\ pvar\ \text{H}\ []$$

means

$$z[p] \leftarrow f[\psi_p = \max\{p' | p' \leq p, h[p'] = 1\}] \oplus f[\psi_p + 1] \oplus \ldots \oplus f[p]$$

Thus, line (4) of the above pseudocode results in:

$$\text{CMAX}[p_i^d + q] = \max_{0 \leq p' \leq q} \{\text{CMAX}[p_i^d + p']\}$$

from which it can be seen that the above assertion about CMAX follows.

6 Generalizations

In many situations one wishes to know more than just the score of the best subsequence matches. For instance, it is often useful to know where along the protein these matches occur or how many gaps are inserted into them. The algorithm described above can be modified to retain such information without suffering a significant decrease in performance. As an illustration, we sketch the modified algorithm used to determine the locations of the initial and final amino acids of the best subsequences for each comparison, as well as the scores.

Assume that for every comparison of proteins A and B, we wish to compute not only $\max\{C(A', B') | A' \prec A, B' \prec B\}$, but also the positions of the first and last amino acids in the subsequences A', B'. That is, if $A' = (a_{i_1}, a_{i_2}, \ldots, a_{i_z}), B' = (b_{j_1}, b_{j_2}, \ldots, b_{j_z})$ maximize $C(A', B')$, we want the dynamic programming algorithm to give us the values of $C(A', B')$, i_1, i_z, j_1, and j_z. The algorithm can be modified to do this by associating with each matrix entry $C_{r,s}$ the values of i_1 and j_1 for the subsequences ending at r, s which yield the maximum score for $C_{r,s}$. These can be stored in matrices $I_{r,s}$, $J_{r,s}$, and when the values of $C_{r,s}$ are computed inductively, the values of $I_{r,s}$ and $J_{r,s}$ and be simply carried over from the previous values of the appropriate C matrix entries. If we introduce the notation $A|B$ to denote the concatenation of two variables, and define

$$\max\{A|B, C|D\} = \begin{cases} A|B, & A > C \text{ or } (A = C \text{ and } B \geq D) \\ C|D, & \text{otherwise} \end{cases}$$

then our new recursion relation can be defined as:

$$C_{r,s}|I_{r,s}|J_{r,s} = \max \begin{cases} 0|r|s \\ (C_{r-1,s-1} + D(a_r, b_s))|I_{r-1,s-1}|J_{r-1,s-1} \\ (C_{r,s-1} + g)|I_{r,s-1}|J_{r,s-1} \\ (C_{r-1,s} + g)|I_{r-1,s}|J_{r-1,s} \end{cases}$$

And the values of C, i_1, j_1, i_z, and j_z for the optimal subsequence are given by:

$$C|i_1|j_1|i_z|j_z = \max_{r,s}\{C_{r,s}|I_{r,s}|J_{r,s}|r|s\}.$$

It is fairly easy to modify the code described in Section 5.2 to implement this algorithm. One could use a similar approach to retain other information about the best subsequences, such as the number of times a particular pair of amino acids appear as a_{i_n}, b_{j_n} for any n, or the values of z for each sequence comparison.

In fact, one can also generalize the dynamic programming algorithm to deal with a broader class of comparison functions C. For instance, one might wish to use a scoring function of the form:

$$C(A', B') = \sum_{k=1}^{z} D(a_{i_k}, b_{j_k}) + \sum_{k=1}^{z-1} [g \cdot (\sigma_k + \tau_k) + \chi(\sigma_k) + \chi(\tau_k)]$$

$$\text{where } \chi(n) = \begin{cases} 0, & n = 0 \\ G \text{ otherwise} \end{cases}$$

This function is similar to the general case considered above, however it incurs an extra penalty for having any gap at all between consecutive amino acids in a sequence, making the penalty for continuing a gap once it has begun relatively smaller. This is of interest in comparing proteins since the random process of mutation is more likely to remove an entire group of amino acids by breaking protein sequences than to remove the same number of individual proteins from different locations on the same protein.

To implement such a function, one must modify the dynamic program slightly more than in the previous example, but the main idea remains the same. For a scoring system such as this, one sets up a new set of variables which keep track of the best scores in all four of the possible combinations of situations where the subsequences A' and B' either end in gaps or do not. Actually, since any situation in which both subsequences end in gaps can be expressed as a gap in one of the sequences followed by a gap in the other, these values can be put into three matrices defined as follows:

$$C00_{r,s} = \max\{C(A', B') | i_z = r, j_z = s\},$$

$$C10_{r,s} = \max\{C(A', B') + G + g \cdot (r - i_z) | i_z < r, j_z = s\},$$

$$C01_{r,s} = \max\{C(A', B') + G + g \cdot (s - j_z) | i_z = r, j_z < s\}.$$

Where max is taken across all $A' = (a_{i_1}, \ldots, a_{i_z})$, $B' = (b_{j_1}, \ldots, b_{j_z})$.

We then have the recursion relations:

$$C00_{r,s} = \max \begin{cases} 0 \\ C10_{r-1,s-1} + D(a_r, b_s) \\ C01_{r-1,s-1} + D(a_r, b_s) \\ C00_{r-1,s-1} + D(a_r, b_s) \end{cases}$$

$$C01_{r,s} = \max \begin{cases} C00_{r,s-1} + G + g \\ C10_{r,s-1} + G + g \\ C01_{r,s-1} + g \end{cases}$$

$$C10_{r,s} = \max \begin{cases} C00_{r-1,s} + G + g \\ C01_{r-1,s} + G + g \\ C10_{r-1,s} + g \end{cases}$$

which, as in the basic algorithm, allow us to compute the best subsequence by computing $\max C00_{r,s}$. Thus, the dynamic programming method allows us to do sequence comparisons with a variety of scoring metrics, and to save any information about those comparisons which could be useful.

7 Results

Using the software described here we have begun to experiment on databases of proteins using a variety of scoring metrics. The inner loop of the basic algorithm takes about 300 μseconds on a CM-2 with a clock speed of 7Mhz when no position information is computed, and about 600 μseconds when the positions of the best subsequences are calculated as in Section 6. We ran a database of 2192 proteins against itself using a simple scoring metric of the form $D(a_i, a_j) = (2\delta_{ij} - 1) + \alpha, g = \beta < 0$, where α, β determine the relative penalties for mismatches and gaps.

This is one of the matrices most frequently used by biologists. The run took about 6 hours on an 8K CM-2 to compute scores between all pairs of proteins and the positions of the subsequences giving these scores. (This represents about 1.8×10^{11} matrix entry computations, and would have taken $\approx$ 45 minutes on a 64K machine.) We have begun to analyze the results of this comparison search using various clustering heuristics to find multiple-sequence similarities.

Another set of experiments which we carried out compared a variety of scoring systems on several small databases. It can be shown mathematically [15] that when one compares random proteins with a scoring metric of the form $D(a_i, a_j) = (2\delta_{ij} - 1) + \alpha, g = \beta$, the mean length of the subsequences in the optimal matches varies according to the values of α and β in a regular manner. The $\alpha\beta$ plane divides into two regions, one where the mean length of optimal matches varies as the log of the lengths of the proteins, and another where the optimal matches vary linearly with the protein lengths. There is a region on the $\alpha\beta$ coordinate plane where the transition between these two domains occurs, but an explicit determination of this phase transition point has never been made. By computing the mean

$$\langle \frac{i_z - i_1 + j_z - j_1}{2} \rangle$$

across all pairwise comparisons for varying values of α, β, we were able to locate this transition region empirically, and to determine the qualitative behavior of this scoring function under changes in the parameters α and β, [see Figure 4]. We repeated this experiment for actual and random proteins, i.e., sequences of amino acids generated according to the observed distribution of amino acids in the set of proteins in the database, and obtained essentially the same results, indicating that the conclusions drawn here could be applied to comparisons of actual databases.

8 Summary

The use of highly parallel computers has enabled us to make progress in several areas of biological research (Section 7), which would have been difficult without software tools to do high-speed subsequence comparisons using arbitrary scoring metrics. The methods used here could easily be modified to be used in other contexts such as speech recognition.

β= \ α=	-0.88	-0.75	-0.63	-0.50	-0.38	-0.25	-0.13	0.00	0.13	0.25	0.38	0.50	0.63	0.75	0.88	1.00
-0.13	7.54	16.14	58.95	244.81	322.33	346.73	358.98	365.62	369.38	372.25	374.19	375.87	377.12	377.98	380.70	384.58
-0.25	6.54	7.54	10.96	16.14	28.71	58.95	150.75	244.81	296.36	322.33	338.04	346.73	362.69	370.86	376.42	381.68
-0.38	6.47	6.81	7.54	10.05	12.20	16.14	23.43	85.48	58.95	118.87	245.05	316.42	348.64	363.21	371.83	378.77
-0.50	6.47	6.54	6.89	7.54	9.83	10.96	12.93	16.14	24.19	43.71	122.78	260.23	327.23	353.94	366.51	375.42
-0.63	6.47	6.50	6.81	6.92	7.54	9.76	10.60	12.80	17.01	26.28	56.69	158.85	296.20	343.40	362.42	373.40
-0.75	6.47	6.47	6.54	6.81	6.93	7.80	9.94	11.21	13.81	19.97	34.23	92.21	254.00	332.04	357.56	371.00
-0.88	6.47	6.47	6.50	6.81	6.89	7.16	7.88	10.91	12.50	16.34	25.89	59.28	195.25	318.96	353.41	368.97
-1.00	6.47	6.47	6.50	6.54	6.81	7.02	7.27	8.43	11.87	14.30	21.66	42.19	141.88	299.36	348.82	365.26
-1.13	6.47	6.47	6.47	6.50	6.81	6.93	7.23	7.68	10.77	13.29	18.71	33.18	98.68	274.22	344.08	364.92
-1.25	6.47	6.47	6.47	6.50	6.54	6.94	7.16	7.68	9.96	11.89	17.17	27.96	75.72	256.65	340.97	364.23
-1.38	6.47	6.47	6.47	6.50	6.50	6.94	7.09	7.63	9.56	11.20	16.25	25.13	64.74	251.55	337.93	363.18
-1.50	6.47	6.47	6.47	6.47	6.50	6.70	6.97	7.54	9.39	10.84	14.99	22.56	55.51	216.93	334.54	362.16

Figure 4: A protein database consisting of fifty (50) proteins with lengths between 192 and 220 was compared with itself using a scoring matrix $D(a_i, a_j) = (2\delta_{ij}-1) + \alpha$, $g = \beta$. The figure gives the mean length of the best subsequence match for various α, β pairs. The phase transition region is highlighted in gray.

References

[1] Bellman, Richard. 1957. Dynamic Programming. Princeton University Press. Princeton, NJ.

[2] Blelloch, G.E. "Scans as Primitive Parallel Operations", Proc. Int. Conf. on Parallel Processing, 1987.

[3] Collins, J.F. and Coulson, A.F.W. 1984. "Applications of parallel processing algorithms for DNA sequence comparison." *Nucleic Acid Research* 12, 181-192.

[4] *Connection Machine® Model CM-2 Technical Summary* Thinking Machines Corporation. 1987.

[5] Doolittle, Russell F. 1986. Of Urfs and Orfs, A Primer on How to Analyze Derived Amino Acid Sequences. University Science Books. Mill Valley, CA.

[6] Edmiston, Elizabeth and Robert A. Wagner. 1987. *Parallelization of the Dynamic Programming Algorithm*

for Comparison of Sequences. Proceedings of the 1987 International Conference on Parallel Processing. Penn State Press. Pennsylvania, PA. 78-80.

[7] Hillis, W. Daniel. 1985. The Connection Machine. MIT Press. Cambridge, MA.

[8] Hillis, W. Daniel and Guy L. Steele, Jr. 1986. "Data Parallel Algorithms." *Communications of the ACM.* Vol. 29, No. 12., 1170-1183.

[9] Johnson, D.S. 1973. "Near-Optimal Bin Packing Algorithms." Doctoral Dissertation, Department of Mathematics, Massachusetts Institute of Technology. Cambridge, MA.

[10] Ladner, R.E. and M.J. Fischer. 1980. "Parallel Prefix Computation". *J. Assoc. Comput. Math.* Vol. 27, No. 4, 831-838.

[11] Lipman, D.J. & W.R. Pearson (1985) "Rapid and Sensitive Protein Similarity Searches." *Science* 227, 1435-1441

[12] Needleman, S.B. and C.D. Wunsch. 1970. "A General Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins." *J. Mol. Biol.* 48, 444-453.

[13] Smith, T.F. and Waterman, M.S. 1981. "Identification of common molecular subsequences." *Journal of Molecular Biology* 147, 195-197.

[14] Smith, T.F., Waterman, M.S. and Burks, C. 1985. "The statistical distribution of nucleic acid similarities." *Nucleic Acid Research* 13, 645-656.

[15] Waterman, M.S., L. Gordon, and R. Arratia. 1987. "Phase Transitions in Sequence Matches and Nucleic Acid Structure." *Proc. Natl. Acad. Sci. USA* 84, 1239-1243.

LINEAR OPTIMIZATION VIA MESSAGE-BASED PARALLEL PROCESSING

Craig B. Stunkel
Computer Systems Group
Coordinated Science Laboratory
University of Illinois
1101 West Springfield
Urbana, Illinois 61801

ABSTRACT

Linear optimization problems are commonly solved via an iterative technique known as the simplex method. This paper proposes and examines the performance of several, parallel variants of the simplex and revised simplex algorithms on the Intel iPSC, a message-based parallel system. Linear optimization test data are drawn from commercial sources and represent *realistic* problems. Analysis shows that the speedup obtained is sensitive to both the structure of the underlying data *and* the data partitioning.

1. INTRODUCTION

Linear optimization problems are *underconstrained* linear systems that maximize or minimize some objective function. These linear optimization problems are natural formulations of many business plans and often contain hundreds of equations with thousands of variables. Changing economic conditions dictate that many organizations solve these large linear optimization problems daily.

Historically, linear optimization problems have been solved via the simplex method [Luen73]. Although it is well-known that the computational complexity of the simplex method is not polynomial in the number of equations, experience has shown that its average case behavior is linear. Despite the excellent performance of the simplex method, the size of the optimization problems and the frequency of their solution make linear optimization a computationally taxing endeavor. This computational complexity, coupled with the wealth of research on the simplex method, make parallel solution of linear optimization problems an attractive research problem. Implementations of simplex utilizing special purpose VLSI arrays have been proposed [BeBo87, OnNa84].

This paper examines the performance of several, parallel variants of the simplex and revised simplex algorithms on a message-passing system. A review of the linear optimization problem including a discussion of the simplex and revised simplex algorithms is presented in §2. In §3, possible parallelizations of the simplex algorithm are discussed, together with their potential advantages and disadvantages, and results of benchmark studies of the alternatives are shown. In §4, this discussion is continued for the revised simplex algorithm, and the simplex and revised simplex algorithm performance is compared using linear optimization data drawn from commercial sources. This performance analysis shows that the speedup obtained is sensitive to both the structure of the underlying data *and* the data partitioning. A summary of the work is presented in §5.

2. LINEAR OPTIMIZATION AND THE SEQUENTIAL SIMPLEX ALGORITHM

2.1. General Linear Optimization Problem

Mathematically, the linear optimization problem can be stated as:

$$\text{Minimize:} \quad c^T x \tag{1}$$

$$\text{Subject to:} \quad Ax = b, \quad \text{where} \quad b \geq 0$$

$$x \geq 0$$

Acknowledgment: This research was supported in part by a Shell Doctoral Fellowship and in part by a Digital Faculty Incentives for Excellence Award.

Here, c^T is an n vector of variable coefficients that defines the objective function (i.e., the function being minimized). For a maximizing problem, the negative of the objective function can be minimized. The objective function can thus be viewed as a cost function, where the object is to minimize total cost. The $m \times n$ linear system $Ax = b$ defines the linear constraints on the objective function x. Each of the m rows of the matrix A defines a constraint on the n variables of the objective function.

The optimization problem arises because the linear system $Ax = b$ is underconstrained (i.e., m is smaller than n, and the matrix A contains many more columns (variables) than rows (constraints)). Consequently, there are many possible x vectors that satisfy the system $Ax = b$. A fundamental theorem of linear programming states that an optimal solution, if it exists, occurs when $n - m$ elements of x are zero (i.e., when there are precisely m non-zero elements of x). This corresponds to the solution of an $m \times m$ linear system, the *basis*, obtained by selecting m of the n columns of the matrix A. The simplex method is a search algorithm that decreases the value of the objective function at each iteration by selecting a non-zero element of x, a so-called *basic* variable, and replacing the corresponding column of A with another column.

In this paper, two forms of the simplex method are examined: the simplex algorithm and the revised simplex algorithm. Although these two algorithms are based on the same underlying principles, they use different types of operations to reach the solution. In the remainder of this section a brief, and by no means complete, overview of the algorithms is presented. For more thorough treatments of the simplex method and optimization theory, see [Foul81, Murt83, Llew64, Luen73]. See [KuTZ71, Solo84] for helpful hints on practical implementation of the simplex and revised simplex algorithms.

2.2. Simplex Algorithm

In most practical problems some or all of the equations in $Ax = b$ are inequalities. There is a simple method of transforming these inequalities into equalities while maintaining the $x \geq 0$ constraint. In addition, one must obtain an initial feasible solution (an initial solution that satisfies the constraint $x \geq 0$). Thus, each equation forming $Ax = b$ is transformed as follows:

If $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$, a *slack* variable is added: $\sum_{j=1}^{n} a_{ij}x_j + s_i = b_i$

If $\sum_{j=1}^{n} a_{ij}x_j \geq b_i$, a *surplus* variable and an *artificial* variable are added: $\sum_{j=1}^{n} a_{ij}x_j - s_i + r_i = b_i$

If $\sum_{j=1}^{n} a_{ij}x_j = b_i$, an artificial variable is added: $\sum_{j=1}^{n} a_{ij}x_j + r_i = b_i$

The artificial variables must be zero when the optimum value is found. They aid in obtaining an initial feasible solution. The slack and surplus variables may or may not be zero when the optimum is found. Clearly, the $x \geq 0$ constraint can be still be satisfied by these new equations.

To explain the notion of a feasible solution, the notion of a *basis* must be reviewed. A basis of A is a linearly independent collection of m columns of A. It can be represented as $B = [A_{j_1} \cdots A_{j_m}]$. With a valid basis of A, a basic solution can be found by setting all components of x corresponding to columns of A not in B to zero and solving the resulting m equations to determine the remaining components of x. These are the basic variables. If the solution to these equations satisfies $x \geq 0$, then it is a basic feasible solution. The initial basis is commonly chosen as the

A columns corresponding to the slack and artificial variables, because this basis always results in a basic feasible solution.

The simplex method systematically moves from basic feasible solution (bfs) to bfs. This movement, called *pivoting*, will terminate in an optimal solution if one exists. This pivoting consists of three steps: (1) finding a new basis column that decreases the objective (cost) function value, (2) finding the column to remove from the basis that maximizes the decrease in the objective function value while maintaining the requirements of a bfs, and (3) replacing the old basis column with the new one.

If B is an $m \times m$ matrix representing the current basis, and $A_{B(i)}$ is the column of A that is currently the i-th basis vector of B, any column A_j in the original A matrix can written in terms of the basis vectors of B:

$$A_j = \sum_{i=1}^{m} \bar{a}_{ij} A_{B(i)} \tag{2}$$

Let $x_{B(i)}$ represent the i-th value of the current bfs, and let $c_{B(i)}$ represent the element of the objective row corresponding to this variable $x_{B(i)}$. Equation (2) can be interpreted as meaning that for every unit of the variable x_j that enters the basis, an amount $\bar{a}_{ij}$ of the each of the variables $x_{B(i)}$ must leave. Thus a unit increase in the variable x_j results in a net change in the objective function equal to:

$$\bar{c}_j = c_j - \sum_{i=1}^{m} \bar{a}_{ij} c_{B(i)} \tag{3}$$

It is then profitable to make x_j basic and to bring A_j into the basis when $\bar{c}_j < 0$. Also, if all $\bar{c}_j \geq 0$, then a local (and global) optimum has been reached, and simplex terminates.

$\bar{A}_j$ is the representation of A_j with respect to B, and can be obtained as $\bar{A}_j = B^{-1}A_j$. Also, because b is simply another column, it can also be represented in terms of the basis as $\bar{b} = B^{-1}b$. The value of $\bar{c}_j$ can now be calculated in terms of B:

$$\bar{c}_j = c_j - \sum_{i=1}^{m} \bar{a}_{ij} c_{B(i)} = c_j - c_B^T \bar{A}_j = c_j - c_B^T B^{-1} A_j$$

$$\text{or } \bar{c}_j = c_j - \pi^T A_j \qquad \text{where } \pi^T = c_B^T B^{-1}$$

Any x_j with $\bar{c}_j < 0$ can become basic and decrease the objective row function, but generally the x_j corresponding to the minimum $\bar{c}_j$ is chosen. Although there is no theory that guarantees this to be the best choice for decreasing the objective row function, empirical data show that it suffices. The corresponding A column entering the basis is also termed the *pivot column*.

Now, a column must be chosen to leave the basis. The column must be chosen such that replacing it with the entering basis column will guarantee that a bfs still exists with the new basis. Although the details will be omitted, the following criterion guarantees that a new bfs will be reached (for entering basis column = A_{piv_col}):

$$\text{find the row } i \text{ which minimizes } \frac{\bar{b}}{\bar{a}_{i,piv\_col}} \text{ for all } \bar{a}_{i,piv\_col} > 0$$

Intuitively, the purpose of this criterion is to ensure that the new solution $\bar{b}$ of the bfs is non-negative. This row i (also known as the *pivot row*) indicates that column $A_{B(i)}$ of the basis representation should be replaced by A_{piv_col}. Correspondingly, x_{piv_col} replaces $x_{B(i)}$ as a basic variable. This completes one iteration of the simplex algorithm.

It can been shown [Foul81] that simple matrix row operations via Gauss-Jordan elimination produce $\bar{A}$, $\bar{c}$, and $\bar{b}$ without altering the set of feasible solutions. The simplex algorithm uses this Gauss-Jordan transformation of the tableau to move from bfs to bfs. Figure 1 shows the initial simplex tableau, with the starting values for $\bar{A}$, $\bar{b}$, and $\bar{c}$ simply equal to A, b, and c. x_0 is the current optimum value of the objective function. Figure 2 enumerates the computational steps for each iteration. The third step uses Gauss-Jordan elimination to manipulate the entire coefficient matrix in a pivot operation. This places a 1 at the pivot point (the intersection of the pivot column and row) and zeros elsewhere in the pivot column, including the objective row, and updates $\bar{A}$, $\bar{b}$, and $\bar{c}$.

Analysis of Figure 2 shows that the complexity of the simplex algorithm is $O(mn-m^2+n+m)$ additions and comparisons, $O(mn-m^2+m)$ multiplications, and $O(m+1)$ divisions per iteration.

a_{11}	a_{12}	.	.	a_{1n}	b_1
a_{21}	a_{22}	.	.	a_{2n}	b_2
.	.	.	.	.	.
a_{m1}	a_{m2}	.	.	a_{mn}	b_m
c_1	c_2	.	.	c_n	x_0

Figure 1. Simplex Tableau

Phase	Operation
Locate *piv_col*	min_obj = minimum $(\bar{c}_j)$ $\quad j=1,...,n$ if $min\_obj \geq 0$ then exit, optimum has been found! $piv\_col = j$ for the column with this minimum $\bar{c}_j$
Locate *piv_row*	$min\_ratio = \min_{i=1,...,m} \frac{\bar{b}_i}{\bar{a}_{i,piv\_col}}$ for $\bar{a}_{i,piv\_col} > 0$ $piv\_row = i$ for the row with this minimum ratio
Gauss-Jordan Elimination	for $i \neq piv\_row$ $factor_i = \frac{\bar{a}_{i,piv\_col}}{\bar{a}_{piv\_row,piv\_col}}$ $\bar{a}_{i,j} = \bar{a}_{i,j} - factor_i \bar{a}_{piv\_row,j}$ $\quad j=1,...,n$ $\bar{b}_i = \bar{b}_i - factor_i \bar{b}_{piv\_row}$ end for $\bar{c}_j = \bar{c}_j - \frac{\bar{c}_{piv\_col}}{\bar{a}_{piv\_row,piv\_col}} \bar{a}_{piv\_row,j}$ $\quad j=1,...,n$ $\bar{a}_{piv\_row,j} = \frac{\bar{a}_{piv\_row,j}}{\bar{a}_{piv\_row,piv\_col}}$ $\quad j=1,...,n$ $\bar{b}_{piv\_row} = \frac{\bar{b}_{piv\_row}}{\bar{a}_{piv\_row,piv\_col}}$

Figure 2. Simplex Algorithm

2.3. Revised Simplex Algorithm

Unlike the simplex algorithm, which continually updates all of the $\bar{A}$ columns, the revised simplex algorithm [Solo84] maintains only enough information to reproduce an updated objective row or any updated column of $\bar{A}$, and relies on B and π as in the explanation of the general simplex method. The $\bar{b}$ column is handled identically to the simplex algorithm. The original A matrix and objective row are never changed, and since A is usually quite sparse, it can be stored in sparse form. The B matrix is never needed in computations, but its inverse B^{-1} is used for calculating π and $\bar{A}$ values. Although the proof is beyond the scope of this introduction, B^{-1} and $-\pi$ can be maintained through elementary row operations.

The search for the pivot column (the "find *piv_col*" step) in a revised simplex iteration uses $-\pi$ to calculate the updated $\bar{c}^T$ row as shown in Figure 3, and the search for the pivot row (the "find *piv_row*" step) uses B^{-1} to calculate the updated $\bar{A}_{piv\_col}$. Matrix B^{-1} is initially an $m \times m$ identity matrix, and $-\pi$ is initially a $\mathbf{0}$ vector. Once the pivot column and pivot row have been identified, the B^{-1} matrix and $-\pi$ vector can be updated through a process similar to Gauss-Jordan elimination, using $\bar{A}_{piv\_col}$ to determine the row fractions as shown in Figure 3.

In the revised simplex, more time is spent in finding the minimum objective and ratio values, but less time is required for updating the

matrix. If s is a fraction representing the percentage of non-zero values in the sparse A matrix, then the number of addition and compare operations per iteration is $O(mns+m^2+n+m)$. Similarly, the number of

Phase	Operation
Find *piv_col*	$\bar{c}_j = c_j - \pi^T A_j \qquad j=1,...,n$ min_obj = minimum($\bar{c}_j$), $\qquad j=1,...,n$ if $min\_obj \geq 0$ then exit, optimum has been found ! $piv\_col = j$ for column with this minimum $\bar{c}_j$
Find *piv_row*	$\bar{A}_{piv\_col} = B^{-1}A_{piv\_col}$ $min\_ratio = \min_{i=1,...,m} \frac{\bar{b}_i}{\bar{a}_{i,piv\_col}}$ for $\bar{a}_{i,piv\_col}>0$ $piv\_row = i$ for row with this minimum ratio
Gauss-Jordan Elimination	for $i \neq piv\_row$ $factor_i = \frac{\bar{a}_{i,piv\_col}}{\bar{a}_{piv\_row,piv\_col}}$ $\vec{B_i} = \vec{B_i} - factor_i \vec{B_{piv\_row}} \qquad j=1,...,m$ $\bar{b}_i = \bar{b}_i - factor_i \bar{b}_{piv\_row}$ end for $-\pi_j = -\pi_j - \frac{\bar{c}_{piv\_col}}{\bar{a}_{piv\_row,piv\_col}} \vec{B_{piv\_row}} \qquad j=1,...,m$ $\vec{B_{piv\_row}} = \frac{\vec{B_{piv\_row}}}{\bar{a}_{piv\_row,piv\_col}} \qquad j=1,...,m$ $\bar{b}_{piv\_row} = \frac{\bar{b}_{piv\_row}}{\bar{a}_{piv\_row,piv\_col}}$

Figure 3. Revised Simplex Algorithm

multiplications performed per iteration is $O(mns+m^2+m)$. The number of divisions is $O(m+1)$. Overall, it can be seen the algorithm requires $O(2mns+2m^2)$ floating point operations per iteration, compared to $O(2mn-2m^2)$ for standard simplex. For a sparse A tableau and $m < n - m$, revised simplex requires fewer floating point operations. Another advantage of revised simplex is the algorithm's computational robustness. Revised simplex does not update the entering variable column until it actually enters the basis. Because of this, the accumulation of round-off errors on the coefficients of the column is reduced.

3. PARALLEL SIMPLEX ALGORITHMS

The message passing algorithms of this paper were developed on an Intel iPSC hypercube with 16 processors, each with an extended memory of 4.5 Megabytes. A 32 node Intel hypercube with 0.5 Megabytes of memory per node was also used for experiments measuring communication times. The extended memory on the 16 node machine supported larger problem sizes, and made it possible to measure single node performance for calculating speedups and efficiencies on large problems.

In this implementation, there is little difference between a serial version of the simplex or revised simplex algorithm and the parallel version running on only 1 node, except for down-loading and up-loading of data from and to the host at the start and the finish of the program. This facilitated the measurements of serial execution time. Serial execution time was defined as the time between when the last data is down-loaded from the host to the time when up-loading of the answer to the host begins (i.e., all host-node interactions were excluded). Parallel execution time for 1 or more nodes included the time for down-loading and up-loading data. Hence, calculated speedups for a single node are less than 1.

The linear optimization test data are drawn from commercial sources and represent realistic problems. We benchmarked several problems with a range of sizes from "afiro" (m=27 and n=59) to "bandm" (m=305 and n=777) on 1, 2, 4, 8, and 16 nodes of the iPSC. Larger problems were also run on 16 nodes, but did not fit into the memory of a single hypercube node, hindering speedup calculations.

In message passing architectures, many implementation details are determined once the data distribution decisions are made. This is primarily a consequence of the relatively high cost of communication versus local memory access. The simplex and revised simplex algorithms share many characteristics with linear equation solution, matrix multiplication, and other matrix operations. Previous work on distributed linear system solvers has advocated row or column partitioning of matrices [GeHe85, Mole85, AyOz87]. Similar methods are pursued here, and two data distribution strategies are presented for solving the simplex algorithm efficiently on hypercubes.

3.1. Column Partitioned Simplex

In the column partitioned method (method A) complete columns (including the objective value) are divided equally among the processors. The pivot column determination requires two steps: (1) finding the local minimum of the objective values for those columns at each node, and (2) using a global minimum communication process to identify and distribute *piv_col*, the identity of the column containing the minimum objective value. Implementation of the global minimum communication is discussed in §3.3.

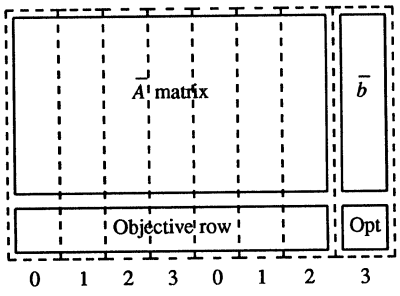
Because one processor contains the entire pivot column $\bar{A}_{piv\_col}$, that processor can now determine the pivot row if it also possesses the vector column $\bar{b}$. Hence, one must balance distributing and maintaining this $\bar{b}$ column at each node against adding a communication step to pass the $\bar{b}$ column from its resident processor to the processor containing $\bar{A}_{piv\_col}$. Once the pivot row is determined, $\bar{A}_{piv\_col}$ is passed to every processor node along with the value of *piv_row*. Each node then uses $\bar{A}_{piv\_col}$ to perform the Gauss-Jordan elimination, exchanging the basic columns, and the simplex iteration is complete.

Another consideration for column partitioning is the assignment of processors to tableau columns. Both the column block and the column wrap methods [GeHe85] were considered, however in the standard tableau format, basic columns were initially concentrated on the right side of the A matrix. Basic columns require less work to maintain because they need not participate in the Gauss-Jordan elimination. To equally divide these basic columns among the processors, the column wrapping method of partitioning was chosen.

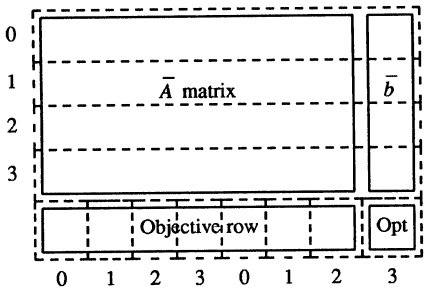
The potential performance advantage of distributing $\bar{b}$ to every node was investigated. Benchmarks showed that, for the Intel hypercube with dimensions 0 to 5, keeping the $\bar{b}$ column on only one node and sending it to the node possessing $\bar{A}_{piv\_col}$ was more efficient than distributing the $\bar{b}$ column. Figure 4(a) summarizes the column partition data placement. Figure 5 shows the speedups obtained both for a distributed $\bar{b}$ column and for $\bar{b}$ maintained at only one node. This figure illustrates the effect of problem size on speedup. For a given hypercube dimension, the time required for the parallel Gauss-Jordan elimination step increases with both m and n, whereas the overhead for global communication remains constant. Also, the time required for the serial computation of *piv_row* and the global send of $\bar{A}_{piv\_col}$ increases with m. Thus, as the product of m and n increases, the parallel steps consume a larger portion of the total execution time, so speedup and efficiency increase. For the problems shown, keeping the $\bar{b}$ column on one node provides marginally greater speedups. For the "scsd1" problem, the difference in speedup is quite small. This is because updating the $\bar{b}$ column is inexpensive (small m) in comparison to the total cost of updating $\bar{A}$ (large n, resulting in many columns per node) through Gauss-Jordan elimination.

3.2. Row Partitioned Simplex

In the row partition strategy, complete rows of the tableau (including the value from the $\bar{b}$ column) are divided equally among the processors. There are two different options for distributing the $\bar{c}^T$

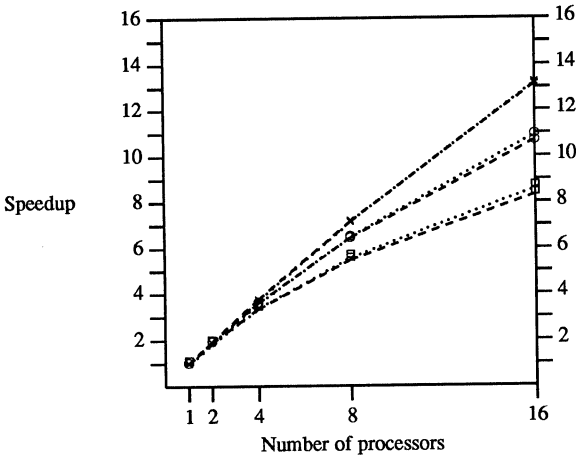


(a) Column partitioning



(b) Row partitioning

Figure 4. Data placement for simplex column and row partitioning. (for m = 4, n = 7, and 4 processors)



Simplex column method A, b on every node:
- 220 x 469 brandy
- 77 x 837 scsd1
- 305 x 777 bandm

Simplex column method A, b on only one node:
- 220 x 469 brandy
- 77 x 837 scsd1
- 305 x 777 bandm

Figure 5. Speedups for the simplex column method with two different $\bar{b}$ column distributions

(objective) row: equally divide the elements of the $\bar{c}^T$ row among the processors, or give each node the entire $\bar{c}^T$ row.

If each node possesses the entire objective row, then the search for *piv_col* can be done separately by each processor and requires no communication. If each node contains only a portion of the objective row, then a global search is required to find *piv_col*, but each processor can find its locally minimum *piv_col* in parallel with the other processors. These options again illustrate the communication versus computation trade-off. The benchmark results show that it is more efficient to partition the $\bar{c}^T$ row and proceed with the global minimization, even for the smaller problem sizes that were tested. For the larger problems the difference was large (e.g., in one case speedup increased from 8 to 11 when the $\bar{c}^T$ row was partitioned).

After *piv_col* is determined, *piv_row* is calculated. Each node possesses a portion of the $\bar{A}_{piv\_col}$ column along with the $\bar{b}$ column elements of the same rows. Thus each node has enough information to find the minimum ratio $\bar{b}_i/\bar{a}_{i,piv\_col}$ for all of its $\bar{a}_{i,piv\_col} > 0$. A global minimum of each locally minimum *piv_row* is needed to find the global *piv_row*, and *piv_row* is distributed to every processor.

For the Gauss-Jordan elimination step, the entire row corresponding to *piv_row* must be resident on every processor. Therefore, the processor holding this row sends it to every other node, and the elimination step executes. Figure 4(b) shows the row partition data placement, and Table 1 compares differences in the simplex algorithm variants, along with the revised simplex algorithm variants to be discussed later.

3.3. Methods for Global Minimum

Each simplex algorithm partitioning requires at least one computation of a global minimum during each iteration. For all cases, the result of this global minimum is needed at every node. Two different methods were compared for obtaining the global minimum. These two methods are illustrated in Figure 6. Method EXCHANGE pairs nodes for an exchange of minima during a step. On each step, nodes that differ by a single bit of a particular power of 2 (i.e., a particular dimension) are paired. Log N of these exchange steps are required, with a different dimension used to pair processors for each step. Method CONDENSE starts by passing all local minima from the upper half of the hypercube to the lower half. Then the lower half splits and passes its newly calculated minima from its own top half to its bottom half again. This process continues until one node (node 0 in this implementation) contains the global minimum. This phase of CONDENSE can be viewed as an inverse global send. Node 0 then globally sends this minimum to every other node. 2 log N steps are required, but at each step, a processor is either sending or receiving, but not both as in EXCHANGE. In addition, no intermediate computation is needed during the log N steps required to globally send the minimum.

Table 2 shows the results benchmarking the comparison of these two methods. The numbers are normalized to the CONDENSE communication times to show relative communication costs; these costs include waiting time. The results show that CONDENSE is slightly less efficient than EXCHANGE for a 2-node hypercube but becomes increasingly more efficient as the dimension of the hypercube increases.

Simplex Algorithm Partitioning	**Major step within the algorithm**	
	Find *piv_row*	Gauss-Jordan elim.
Column (A)	Sequential computation	$\bar{A}_{piv\_col}$ global send Parallel computation
Row (B)	Parallel computation Global minimization	$\bar{A}$ row global send Parallel computation
Revised (C), entire A on node	Parallel computation Global minimization	B^{-1} row global send Parallel computation
Revised (D), partitioned A	A_{piv_col} global send Parallel computation Global minimization	B^{-1} row global send Parallel computation

Table 1. Summary of Hypercube algorithm differences (finding *piv_col* is similar for all partitionings)

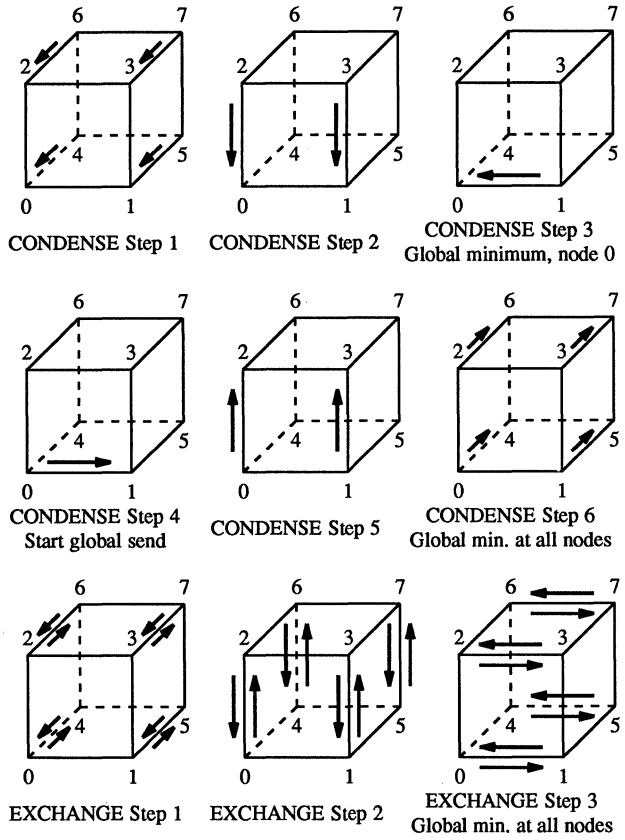


Figure 6. CONDENSE and EXCHANGE methods for finding a global minimum.

Because the Intel iPSC hypercube does not support simultaneous bi-directional transmission, the EXCHANGE method actually requires $2 \log N$ node-to-node message delays and $2 \log N$ comparisons of minima at each node. The CONDENSE method requires $2 \log N$ message delays and only $\log N$ comparisons of minimums. The main reason for CONDENSE's superiority is that messages received out of order will not destroy the global minimum calculation (e.g., if node 0 receives the current minimum from node 2 before it receives the minimum from node 4 in Figure 6). This is because each node does not *send* any information until it first *receives* all of its required minimum information. The EXCHANGE method has no similar property, and messages arriving out of order must be saved until the proper time, so more synchronization overhead is inherent. The chance of out of order messages rises with increasing hypercube dimension, favoring CONDENSE for larger hypercubes. There is a further advantage to the CONDENSE method - most communication steps require only a subset of the processors, so any computation occurring concurrently will proceed with fewer interruptions.

In light of these results, the CONDENSE method was chosen for all global minimum calculations. On a hypercube implemented with two physical links between nodes, the EXCHANGE method should be superior, since the number of steps will reduce to $\log N$.

3.4. Performance Comparisons between simplex methods A and B

The simplex column partition method A and the row partition method B use an identical method for finding *piv_col*. Finding *piv_row*, however, is quite different. The column method is a serial computation with no communication. The row method is a parallel computation but requires a global minimization step. This provides another computation versus communication trade-off.

Hypercube Dimension	Problem			
	"share1b" problem		"israel" problem	
	CONDENSE	EXCHANGE	CONDENSE	EXCHANGE
1	1.00	0.98	1.00	0.99
2	1.00	1.04	1.00	1.00
3	1.00	1.05	1.00	1.01
4	1.00	1.21	1.00	1.02
5	1.00	1.54	1.00	1.18

Table 2. Normalized global minimization communication delays (normalized to the CONDENSE delays)

The Gauss-Jordan step requires a column global send for the column partition and a row global send for the row partition. In most cases, $m < n$, and the column global send is more efficient. However, the column partition requires each node to calculate a row multiplication factor for all rows during the Gauss-Jordan step. In the row partition, each node only calculates these factors for the rows it possesses.

Figure 7 compares the column method A to the row method B for various small to medium sized linear optimization problems. For most problems that were tested, row partitioning achieves a higher speedup for 2 - 16 processors. The extra serial computation required for the column method is more costly than the extra global minimum required for the row method. Also, as the problem size increases, the cost of the serial portion of the column method increases faster than the costs for finding *piv_row* in row method B. One interesting exception is "scsd1," which has a small number of rows m with a number of columns $n \gg m$. For this small number of rows, the serial *piv_row* phase of method A is not expensive, but for method B the communication of the long pivot row required for Gaussian elimination is expensive, and method A has much better speedup.

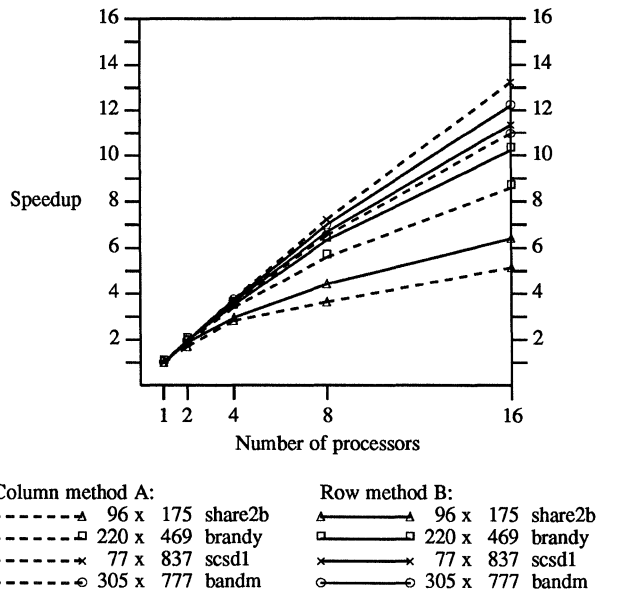


Figure 7. Speedups for simplex column and row partitioning methods

4. REVISED SIMPLEX PARTITIONING METHODS

The revised simplex algorithm usually offers a reduced amount of computation, but increases the complexity of interaction between various elements of the data structure, and thus potentially incurs more communication overhead. The A matrix is static and sparse - typically between 5 and 10% [Solo84] of the elements are non-zero - and all operations involving this matrix use an entire A column (vector-vector or matrix-vector multiplication). A rows carry no significance in revised simplex. Thus, either partitioning columns equally among the nodes or placing the entire matrix in each processor's memory are the viable options. Since the A matrix is not manipulated, there is no penalty for keeping the entire matrix at every node unless there is not enough memory space. For now, it is assumed that the entire A matrix and objective row are stored at each node, and A is stored in a sparse data structure.

It was found in the simplex algorithm that the *piv_col* calculation should be performed through a global minimization, and this is even more valid for revised simplex since the *piv_col* search must be preceded by the computationally intensive update of the objective row ($\bar{c}_j = c_j - \pi^T A_j$). Every node also needs a copy of $-\pi$ before the *piv_col* search begins. In the Gauss-Jordan elimination, $-\pi$ is functionally another row of B^{-1}. Also, a node participating in the *piv_row* search needs elements of A_{piv_col} and $\bar{b}$ from each row it is assigned to search. These observations will facilitate the B^{-1} discussion that follows.

Storing the B^{-1} matrix (m by m) generally requires more memory than the tableau, and in sequential computation the Gauss-Jordan elimination step on the matrix is the most expensive step. Hence a parallel implementation should partition B^{-1} among the processors to parallelize the Gauss-Jordan elimination. The natural ways of partitioning this matrix (which is also used in the $\bar{A}_j = B^{-1}A_j$ calculation of the new pivot column) are by rows or by columns. Unlike in the simplex algorithm, however, partitioning by rows is *clearly* better than partitioning by columns. A step by step comparison will make this apparent. For the *piv_col* step, assume column partitioning of B^{-1}. Then $-\pi$ must be distributed among the nodes, since the update of $-\pi_i$ depends upon elements in the i-th column of B^{-1}. Hence $-\pi$ needs to be gathered and sent to every node (similar communication complexity to global minimization) before each step to find *piv_col*. If B^{-1} is row partitioned, the entire $-\pi$ row can reside on one node (necessitating a global send of $-\pi$ during each iteration), or on every node (no communication for $-\pi$ needed). Either of these row partition options for the $-\pi$ vector are superior to the column partition's option.

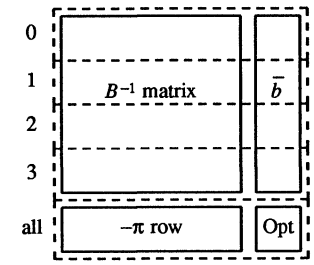
In the *piv_row* step with column partitioned B^{-1}, $\bar{A}_{piv\_col} = B^{-1}A_{piv\_col}$ produces m partial sums which must then be combined through a global sum communication. In contrast, row partitioning allows $B^{-1}A_{piv\_col}$ to produce complete sums for those rows of B^{-1} that are on that node. In addition, if the $\bar{b}$ elements of the same row are present, a local *piv_row* minimum can be found with no communication.

The Gauss-Jordan elimination step is similar for both partitions. Because of the inherent advantages in finding *piv_col* and *piv_row*, the B^{-1} row partitioning was chosen for method C, with $\bar{b}$ partitioned as if it were an extra column of the B^{-1} matrix.

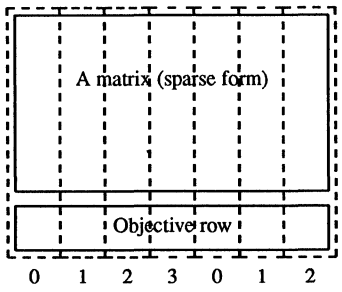
Now the options for the $-\pi$ distribution will be readdressed (maintain $-\pi$ at one node or at every node). It was found in simplex method A that the $\bar{b}$ column should reside on only one node and be sent the node possessing A_{piv_col}. In contrast, the $-\pi$ vector is needed by *every* node during every iteration, requiring a global send if $-\pi$ is maintained by only one node. Benchmark tests showed that it was less expensive to maintain $-\pi$ through Gauss-Jordan elimination on every node than to perform this global send. The data partitioning for method C is shown in Figure 8. Figure 9 demonstrates that $-\pi$ should be maintained by every node.

4.1. Overlapped Communication and Computation

There is potential for overlapping the communication required in each new iteration's *piv_col* global minimization step with the Gauss-Jordan elimination step that terminates the previous iteration. Overlapping these two procedures can reduce the waiting and synchronization time that is inevitable in the global minimization. The Gauss-Jordan elimination step can compute the new $\bar{c}$ row (for simplex)

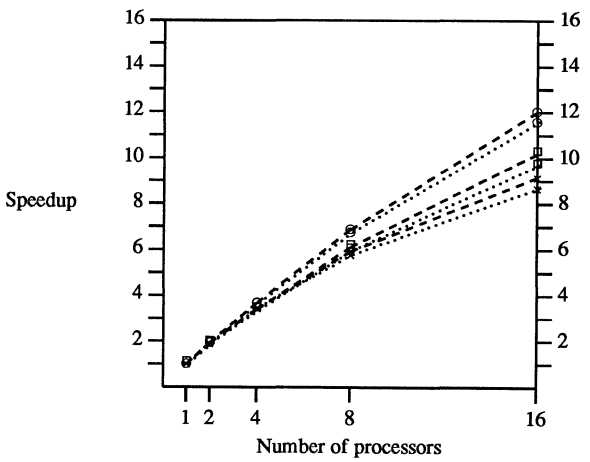


B^{-1} row partitioning (methods C and D)



Sparse tableau partitioning for method D
(in method C, entire A matrix & objective row resides on each node)

Figure 8. Data partitioning for the revised simplex methods C & D
(for m = 4, n = 7, and 4 processors)



Revised simplex method C, $-\pi$ on every node:
- 220 x 469 brandy
- 77 x 837 scsd1
- 305 x 777 bandm

Revised simplex method C, $-\pi$ on only one node:
- 220 x 469 brandy
- 77 x 837 scsd1
- 305 x 777 bandm

Figure 9. Speedups for revised simplex method C with two different $-\pi$ distributions

or the new $-\pi$ row (for revised simplex) before any other rows. With this information the locally minimum *piv_col* can be found at each node, and the global minimization communication steps can begin. Instead of waiting for sends or receives to complete, the processors perform the Gauss-Jordan elimination, checking the status of pending messages before starting the elimination step on a new row. When a status check reveals that a message is ready to be sent or received, the processor handles the communication step immediately, and then resumes the Gauss-Jordan step. This communication overlap was implemented for simplex method B and revised simplex method C (and method D to be discussed shortly). A modest increase in speedup was observed, but not as much as expected (only about 2 - 3% for large problems). This is partly because the computation processor on the Intel hypercube is utilized for much of the sending and receiving work. It is expected that overlaps of this type will have a greater effect on hypercubes with more separation of computation and communication hardware.

4.2. Revised Simplex with Partitioned A Tableau

For the problem sizes tested and the small number of nodes on the hypercubes used in this study, storing the entire sparse A tableau at each node was no more of a problem than storing the partitioned B^{-1} matrix. However, as the number of nodes increases, the B^{-1} matrix partition at each node becomes smaller, while the A tableau storage remains constant. So, for larger problems and large numbers of nodes, it may not be practical to keep the entire A matrix at each processor. Therefore a solution (method D) was investigated that involved a column partitioned A matrix (see Figure 8). The major change required is an extra communication step after *piv_col* is found. The node possessing A_{piv_col} globally sends this sparse column to every other node. Figure 10 shows the speedup of the original revised simplex method C and the new method D for various problem sizes. The difference in speedup for the two methods decreases as the problem size increases, because the time required for method D's extra communication is relatively constant while the computation time is increasing. In fact, for the problems tested in this study, A_{piv_col} was sparse enough that its message length was always under the 1K packet size of the Intel machine, and so the communication overhead was fixed. For large problems the change in speedup is minimal.

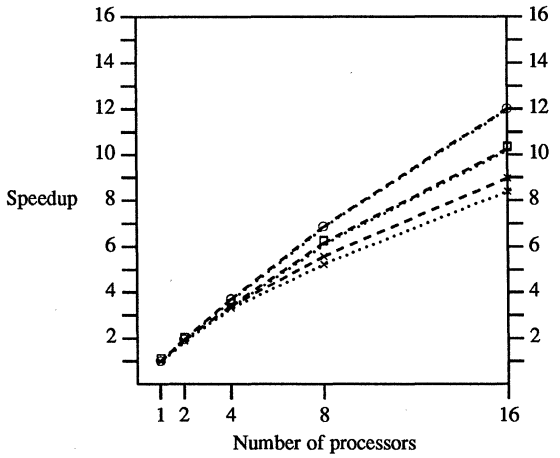


Figure 10. Performance speedup for revised simplex methods.

4.3. Comparison of Simplex B and Revised Simplex C Methods

Despite the more complicated structure of the revised simplex algorithm, a method of partitioning the data was found such that communication requirements are quite similar to that of the row-partitioned simplex method. Each method requires two global minimization steps, and each method requires a global send before the Gauss-Jordan elimination step can proceed. The global send for the simplex case is a full row of the A tableau (length of n), but for the revised simplex case a row of the B^{-1} matrix (length of m) is sent. Since m is typically less than n, the revised simplex method requires smaller messages. Figure 11 compares execution times of methods B and C for several optimization problems. Execution time is used instead of speedup since the basic sequential algorithms for the two methods are different.

The key observation to make from Figure 11 is that the method (simplex or revised simplex) that performs fastest sequentially will also perform fastest in parallel. This is a consequence of the similar communication structure, which makes it easy to predict relative performance of the parallel algorithms. For most larger problems, $m < n - m$, which favors the revised simplex method. A large difference in execution time is seen for "scsd1", for which $m \ll n$. "Share2b" has $m > n - m$, which favors the standard simplex.

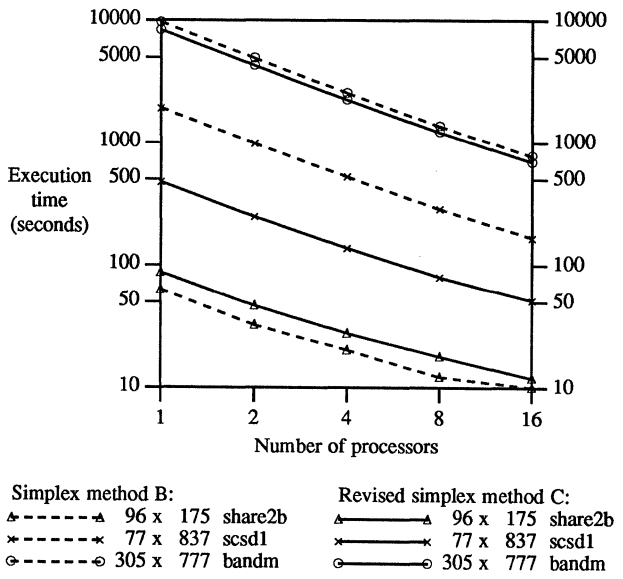


Figure 11. Execution times for method B and revised method C
(Execution times plotted on log scale)

5. SUMMARY

Several partitioning and communication strategies were explored for executing the simplex and revised simplex algorithms on a hypercube. Column and row partitionings were compared for the simplex algorithm, and the row partitioning method was found to be generally superior. Column partitioning is more efficient when the number of rows m is small, and the number of columns n is much greater than m.

Although revised simplex is a more complex sequential algorithm than simplex, it was found that by partitioning the B^{-1} matrix by rows and maintaining both the $-\pi$ vector and the static A matrix on every processing node, communication costs could be kept roughly equivalent to that of the row-partitioned simplex. Comparisons made between actual execution times of these simplex and revised simplex algorithms showed that whichever algorithm performed better sequentially also performed faster in parallel. If $m < n - m$, the revised simplex version should execute faster.

A revised simplex algorithm that partitioned the A matrix by columns among the processors was also investigated. This partitioning reduces the memory requirement for each node but necessitates an extra message during each iteration. This message is generally small and showed very little impact on performance for the larger problems we tested.

Although in theory the simplex and revised simplex algorithm calculations can be almost completely parallelized, communication costs are a significant factor in the actual iPSC execution time. This is primarily a result of the two global minimizations. Identical calculations done by each node, such as maintaining $-\pi$ in the revised simplex method, also contribute to a drop in efficiency as the number of nodes increases. Options were explored for optimizing the communication required for obtaining global minima. The best strategy is to use an "inverse" global send communication pattern to collect the global minimum at one node and then send the answer to every other node.

In summary, the revised simplex algorithms presented seem superior to the simplex algorithms. The revised simplex method with a partitioned A matrix provides the best alternative for practical linear optimization of large problems on a hypercube because of its reduced memory requirement. The speedups obtained show the feasibility of using hypercubes for linear optimization, particularly because many practical problems are far larger than tested here, and because the ratio of communication to computation costs were relatively high for the Intel iPSC.

Acknowledgments

Thanks go to Dan Reed, who originally suggested studying the linear optimization problem, made many valuable suggestions for performance experiments, and provided access to his Intel iPSC/d5 hypercube. Thanks also go to W. Kent Fuchs for his support and guidance during the experiments and the work on this paper.

REFERENCES

[AyOz87] C. Aykanat and F. Ozguner, "Large Grain Parallel Conjugate Gradient Algorithms on a Hypercube Multiprocessor," *Proc. Int. Conf. on Parallel Processing*, pp. 641-644, August 1987.

[BeBo87] A. A. Bertossi and M. A. Bonuccelli, "A VLSI Implementation of the Simplex Algorithm," *IEEE Trans. on Computers*, Vol. C-36, No. 2, pp. 241-247, Feb. 1987.

[Foul81] L. R. Foulds, *Optimization Techniques: An Introduction*, Springer-Verlag, New York, 1981.

[GeHe85] G. A. Geist and M. T. Heath, "Matrix Factorization on a Hypercube Multiprocessor," in *Hypercube Multiprocessors 1986*, ed., M. T. Heath. Philadelphia: SIAM, pp. 161-180, 1986.

[HoJo87] C.-T. Ho and S. L. Johnsson, "Algorithms for Matrix Transposition on Boolean n-cube Configured Ensemble Architectures," *Proc. Int. Conf. on Parallel Processing*, pp. 621-629, August 1987.

[KuTZ71] H. P. Kunzi, H. G. Tzschach, and C. A. Zehnder, *Numerical Methods of Mathematical Optimization*, Academic Press, New York, 1971.

[Luen73] D. G. Luenberger, *Introduction to Linear and Nonlinear Programming*, Addison-Wesley, Reading, Massachusetts, 1973.

[Mole85] C. Moler, "Matrix Computation on Distributed Memory Multiprocessors," in *Hypercube Multiprocessors 1986*, ed., M. T. Heath. Philadelphia: SIAM, pp. 181-195, 1986.

[Murt83] K. G. Murty, *Linear Programming*, John Wiley and Sons, New York, 1983.

[OnNa84] K. Onaga and H. Nagayasu, "A Wavefront-Driven Algorithm for Linear Programming on Dataflow Processor-Arrays," *Proc. Int. Computer Symp.*, pp. 79-94, 1984.

[Solo84] D. Solow, *Linear Programming: An Introduction to Finite Improvement Algorithms*, North-Holland, New York, 1984.

RESULTS OF A MULTIPROCESSOR IMPLEMENTATION FOR SEQUENTIAL DECISION PROCESSES

Marc Diamond, Jim Newhouse, and Jeff Kimbel

FMC Corporation
Advanced Systems Center
1300 South Second Street
Minneapolis, Minnesota 55459

Abstract

We present a multiprocessor realization for sequential dynamic programming problems defined on a state space which is represented by a directed acyclic graph. Our approach applies, in particular, to problems in which an upper and lower bound on the (intially unknown) cost can be determined for each node in the search graph as soon as it is generated, a framework often referred to as an "informed model". We demonstrate how a recursive relationship between the bounds on successive states can be exploited to develop a technique in which state space generation and pruning are carried out in an asynchronous homogeneous manner on a loosely coupled architecture. The process is controlled by a message passing scheme utilizing three basic message types for (1) generating and (2) pruning nodes, and (3) backing up costs. Issues related to the design of the messages, correctness of the approach, and potential problems created by race conditions and deadlock are discussed. Results obtained in the context of a large scale sequential (Markov) decision problem are presented which indicate that near 100% efficiency in the use of processors can be achieved. Because many common problems in game playing, combinatorial optimization, and discrete state optimal control can be adapted to an informed model framework, the techniques presented here are quite general in their potential application.

1 Introduction

Background. The techniques presented here apply to any member of a broad class of discrete-state discrete-time sequential decision problems defined on a directed acyclic graph, referred to as a *decision graph*. The decision graph is an explicit representation of state space and possible state transitions. It has a node associated with each state in the system and arcs from a given node N_1 to another node N_2 if there exists a means by which the system can transit from the state associated with node N_1 to the state associated with node N_2. A certain subset of the set of all nodes in the system are distinguished as *decision nodes* from which a controller has the ability to influence the next state to which the system will progress. Other nodes (outcome nodes) represent states in which an opponent or nature has control over the next state transition.

There is a real-valued *cost* $c(N)$ associated with each node N. The cost associated with each leaf or terminal node is assumed to be directly computable. The cost associated with a non-terminal node, N, is given by a (recursive) function of the costs associated with the set, $SUCC(N) = \{N_1, N_2, ..., N_r\}$, of successors of N. For example, if the objective is to *minimize* the overall cost[1], then the cost of each decision node is given by: $c(N) = \min\{c(N_1), c(N_2), ..., c(N_r)\}$.

Minimax[2] [1] and discrete space and time Markov [2,3] decision processes are perhaps the most prevalent examples of sequential decision processes, although, in the most abstract sense, any combinatorial optimization problem, e.g. an integer program solved using implicit enumeration (branch and bound), could also be considered a form of polyadic sequential decision making in which all nodes are decision nodes. For a Markov decision process the cost associated with an outcome node is given by the weighted average: $c(N) = \sum_{i=1}^{r} p_i\, c(N_i)$, where p_i is the probability of making the transition from state N to state N_i. For a minimax decision process, the cost associated with an outcome node is given as $c(N) = \max\{c(N_1), c(N_2), ..., c(N_r)\}$.

We will assume here that the decision graph is finite, and that there exists a root node which corresponds to

[1] as will be assumed, without loss of generality, in all further discussion

[2] In most discussions of minimax decision processes, decision nodes are maximizing nodes. In all references to minimax in the sequel, we assume the opposite in order to maintain uniformity with discussions on Markov decision processes and our application as discussed in section 4. Extension to the case in which decision nodes are maximizing should be obvious. Similar extensions can easily be made to the equivalent negamax and negamin formulations.

a unique starting state for the process. A generative dynamic programming algorithm is used to determine what action to apply at every decision point in order to minimize total cost. The process [1] involves an expansion queue (or "open" list) to which a mechanism for generating the state of all successors of a node is applied according to some strategy to generate the decision graph[3]. The part of the decision graph that has, at a given point in time, already been generated is called the *search graph*. The *optimal* decision graph is a subgraph of the decision graph generated from the root node and all optimal decisions from that point.

Informed Models. In an *informed model* [4] each node, N, in the state space is assigned an upper bound, $U(N)$, and a lower bound, $L(N)$, on its actual cost $c(N)$ as soon as it is generated. We require that $U(N) \geq c(N) \geq L(N)$ by definition. When $U(N) = L(N) = c(N)$ the node N is said to be *fathomed*. Finally, we will assume that $U(N) = L(N) = c(N)$ for any terminal node N.

Recursive formulation of bounds. We assume that nodes and bounds are generated and updated at discrete points in time. We denote the "current" upper, lower, and cost values for node N at time t by $U^t(N)$, $L^t(N)$, and $c^t(N)$. There is a recursive relationship that exists between the updated bounds for node N and the bounds of its offspring. Assume the cost $c(N)$ is given as some function (depending on the node type) $f(N_1, N_2, ..., N_r)$ of the successors, $N_1, N_2, ..., N_r$, of N. Then we have [4]:

$$U^{t+1}(N) = \min\{U^t(N), f(U^t(N_1), U^t(N_2), ..., U^t(N_r))\},$$

and,

$$L^{t+1}(N) = \max\{L^t(N), f(L^t(N_1), L^t(N_2), ..., L^t(N_r))\}.$$

For the Markov decision process we have,

$$U^{t+1}(N) = \min\{U^t(N), \min_{N_i \in SUCC(N)}\{U^t(N_i)\}\}, \quad (1)$$

and,

$$L^{t+1}(N) = \max\{L^t(N), \min_{N_i \in SUCC(N)}\{L^t(N_i)\}\}, \quad (2)$$

if N is a decision node, and,

$$U^{t+1}(N) = \min\{U^t(N), \sum_{N_i \in SUCC(N)} p_i\, U^t(N_i)\},$$

and,

$$L^{t+1}(N) = \max\{L^t(N), \sum_{N_i \in SUCC(N)} p_i\, L^t(N_i)\},$$

if N is an outcome node.

For a minimax decision process, equations 1–2 give the recursive formulation at a decision (in our case, minimizing) node, and we have

$$U^{t+1}(N) = \min\{U^t(N), \max_{N_i \in SUCC(N)}\{U^t(N_i)\}\}, \quad (3)$$

and,

$$L^{t+1}(N) = \max\{L^t(N), \max_{N_i \in SUCC(N)}\{L^t(N_i)\}\}, \quad (4)$$

if N is an outcome node.

Pruning. The upper bound is a non-increasing, and the lower bound is a non-decreasing, function of t. Thus we can use the bounds associated with nodes in an informed model to curtail generation of the entire state space. Let N be a decision node, and assume the process generates a *tree*. If $L^t(N_i) \geq U^t(N_j)$ for $N_i, N_j \in SUCC(N)$, the node N_i can be "removed from further consideration", since no optimal decision tree will contain it. For minimax decision processes, a similar relationship, i.e., $U(N_i) \leq L(N_j)$ can be used to prune the successor N_i from an *outcome* node N, provided that we assume that our "opponent" is working with the same objective function. In contrast, in a directed graph, a node may have more than one predecessor. As such, it cannot be assumed that once a node becomes pruned (or, "deactivated") that it will not become the offspring of some node (hence, "activated"), possibly in the optimal decision graph, at some future point in time. This has significant implications for the design of our approach, as discussed below.

Relation to other work. The process discussed above defined on informed models provides a generalization for most common search strategies, because we can always assign $U(N) = +\infty$ and $L(N) = -\infty$ for any non-terminal node N when it is first generated. In this case, for the minimax decision processes, if bounds are backed up according to equations 1–4 and pruning is implemented as discussed above, the familiar alpha-beta pruning strategy results[4]. (Refer to [5,6,7,8] for other discussions on parallel implementations of alpha-beta pruning). For branch and bound enumeration of integer programs we can assign as an upper bound (in a minimization problem) the objective value of any incumbant solution or the objective value of any feasible completion of a given partial solution as generated for

[3]The process of expanding a node is assumed here to be atomic, although a more general case can be considered.

[4]Certain aspects of alpha-beta, particularly deep cutoffs, require minor extensions to the message passing scheme proposed here.

example by a "greedy" algorithm. Linear programming relaxations are often used to generate lower bounds.

Parallel search algorithms have received treatment in a broad range of contexts [9,10,11,12], although, for the most part, the assumption is that the underlying process generates a tree. Hence the focus of attention tends to be on issues that differ from those discussed here. Most notable are the problem of mapping nodes to the interconnection topology of the processors in a manner that preserves adjacency, or focusing processor resources on specific areas of the search tree. In [6,7,12] issues related to anomalies that occur in parallel search are discussed. Often, search processes that endeavor only to find a *feasible* solution (e.g. a "path" through a state space from a start state to a goal, or a proof tree for a predicate logic theorem) will be represented as numerical optimization problems in order to guide the search process itself [13]. In this case, the techniques discussed here are also applicable.

Search algorithms defined on informed models for specific problem areas have received some study, particularly among the AI community. Most notably, Berliner's B* [14] and descendent algorithms [15,16,17,18] show performance gains over earlier alpha-beta variants. Recently, Ibaraki, *et al.* [4] have generalized the analysis of informed models, and introduced the algorithm H* which is a member of a class of algorithms that typically outperform alpha-beta. We are aware of no other work treating the parallel implementation of search processes defined on informed models, in particular, when the state space for the process is a directed acyclic graph, which is the most general case.

Outline. In the sequel, section 2 defines the message passing approach, section 3 contains a discussion on the correct operation of the procedure we have defined, and section 4 covers results derived from an implementation of our approach. A synopsis is given in section 5.

2 Approach

In our approach, a hashing function, applied, e.g., to the binary representation of the state vector for the node, is used to determine which processor is responsible for a node once it is generated. Clearly it would be more desirable to map nodes participating in a predecessor-successor relationship to adjacent processors. But because each node may have more than one predecessor, this is difficult to do in general without very specific knowledge of the topology of the decision graph. In order to realize the distributed algorithm, three basic message types are required, corresponding to the basic actions required in the state space generation and pruning. They are: (1) add predecessor link, (2) remove predecessor link, and (3) re-evaluate bounds.

```
ADD PREDECESSOR

IF the target node has not been created THEN
    create the node with predecessor link to sender
    and put node on the expansion queue
ELSE
    add predecessor link back to sender
    IF the offspring node is marked inactive THEN
        mark the node as active
    IF the node has not been expanded THEN
        put node on expansion queue
    ELSE
        send add predecessor messages
          to all successors
```

Figure 1: THE ALGORITHM FOR THE ADD PREDECESSOR MESSAGE HANDLER.

The ADD PREDECESSOR message. The purpose of the add predecessor message is to (re-)establish a link from a successor (offspring) node to a parent (predecessor). The receipt of an add predecessor message indicates to the offspring node that there is a predecessor in the decision graph that (1) is in the current search graph, and (2) requires the minimal expected cost (as well as the current upper and lower bounds) associated with the offspring node in its current and future computations. As such, if the offspring node is inactive at the time the message is received, it must re-activate itself in order to resume the on-going process of generating more and more refined upper and lower bounds, leading eventually to a fathomed node. If the node has not yet been created, then the host processor must create the target node, generate its upper and lower bounds, and put it on the expansion queue. The algorithm for handling an add predecessor message is given in figure 1.

The REMOVE PREDECESSOR message. A remove predecessor message is sent from a parent (predecessor) node to an offspring (successor) node when the parent node no longer needs current and updated state information from the offspring node in calculating its current state. This can occur for two reasons: (1) the parent node has become inactive, or (2) the branch of the search graph rooted at the offspring node has become pruned.

Upon receipt of a remove predecessor message, the offspring node removes the link back to the (sending) predecessor node. If the offspring node remains active (i.e., it still has predecessors in the current search graph) then no further action is required. However, removal of the link back to the parent node may reduce message

REMOVE PREDECESSOR

Remove link to predecessor node

IF list of predecessor nodes is now empty THEN
    IF the node is on the expansion queue THEN
        remove the node from the expansion queue
    ELSE
        send remove predecessor messages to all offspring

Figure 2: THE ALGORITHM FOR THE REMOVE PREDECESSOR MESSAGE HANDLER.

traffic by removing a potential channel along which re-evaluate bounds messages can pass. Otherwise, if the list of predecessor links becomes empty as a result of the removal of the link, then the node becomes inactive. It removes itself from the expansion queue if it is on it, or else it sends remove predecessor messages to all of its offspring. The algorithm used by the handler for a remove predecessor message is given in figure 2.

The RE-EVALUATE BOUNDS message. The re-evaluate bounds message indicates to the receiving (parent) node that the bounds of the sending (offspring) node have changed, and hence it is appropriate for the parent to recalculate its bounds according to the recursive formulation given in section 1. The re-evaluate bounds message is sent when the bounds of an offspring node have changed. This can occur either if the offspring node itself has received a re-evaluate bounds message, or when the bounds are set for the first time (i.e. when its data structure is established). The algorithm used by the handler for a re-evaluate bounds message is given in figure 3.

RE-EVALUATE BOUNDS

Re-calculate the upper and lower bounds according to the recursive formulation for the receiving node

IF the bounds have changed THEN
    send re-evaluate bounds messages to all predecessors
IF one or more successors can be pruned THEN
    send remove predecessor messages to those successors
IF the upper bound is equal to the lower bound THEN
    mark the node as fathomed

Figure 3: THE ALGORITHM FOR THE RE-EVALUATE BOUNDS MESSAGE HANDLER.

3 Correctness of the Approach

Several potential problems exist with processes defined on graphs that do not occur in the context of problems defined on a tree. Observing the message behavior around a typical node reveals that it may be generated, deactivated, and reactivated several times during the course of expanding the decision graph. We must show, therefore, that when (and if) the process terminates, any node in an optimal decision tree is active. Furthermore, we must show that there is no possibility for "infinite message loops", caused by a sequence of message initiations that eventually "loop back" to the original node in the sequence, thereby resulting in a ever-increasing flow of message traffic. In the following discussion we show that the message passing scheme proposed here is guaranteed to produce the correct result, and terminate in a finite amount of time assuming the decision graph is finite[5].

In the sequel, the following strategy will be used. First we define the concept of the *state* of the system at a given time t. We then show that the system has a finite number of states, provided the number of nodes in the decision graph is finite. Next we show that the state transition diagram is finite. From these two results it can be concluded that the algorithm will terminate, provided that the number of message initiations is finite. We will then show that at termination, all nodes on the optimal decision graph are fathomed, and hence the root node is fathomed. Finally, we show that the number of message initiations is always finite. This is done by first showing that the result holds when the process generates a tree, and then extending this result to the case where the process generates a graph.

Definitions. The state of a node N at a point in time t will be given as

$$S^t(N) = (U^t(N), L^t(N))$$

where $U^t(N)$ (respectively, $L^t(N)$) is the upper (respectively, lower) bound of node N at time t. If node N has not been generated (i.e., not in the current search graph) at time t, then we set $U^t(N) = +\infty$ and $L^t(N) = -\infty$. The *distance* between two nodes in a directed graph is the length of the shortest (in this case, directed) path between the two nodes. The *depth*,

[5] Many of the arguments provided here are given in outline form only due to space limitations. Full details can be found in [19].

$d(N)$, of a node N in a directed acyclic graph is the distance from the root node to N. The *separation* of two nodes is the length of the *longest* direct path between them. The *sound*, $\sigma(N)$, of a node N is the separation between the root node and N.

3.1 Finite Termination

Lemma 3.1. The number of states for any node in the system is finite.

Proof: The proof proceeds by induction on the sound of a node N in the decision graph. The assertion holds for any leaf node N_L since it can be in only two states: $S^t(N_L) = (+\infty, -\infty)$ if it is not in the current search graph, and $S^t(N_L) = (U(N_L), L(N_L))$ otherwise, where $U(N_L)$ and $L(N_L)$ are the bounds generated by the initial bound generation procedure[6]. Since a leaf node has no successors, its upper and lower bounds will never be changed once it is generated. This provides the basis.

Assume the result holds for all nodes at a sound greater than or equal to s, and let $\sigma(N)$ be $s-1$. Note that for any successor N_i of N, we must have $\sigma(N_i) \geq s$, so N_i has a finite number of states by the inductive hypothesis. The set of states for node N is contained in a set generated by the application of the recursive formulation for the costs to the cross product of the set of states for each of its successors. The latter set is finite from the inductive hypothesis. ■

Theorem 3.2. The number of states for the system is finite.

Proof: By its definition:

$$\{S^t : 0 \leq t \leq \infty\} \subseteq \times_N \{S^t(N) : 0 \leq t \leq \infty\}.$$

Every element in the cross product is finite by lemma 3.1, and there is a finite number of nodes in the decision graph. ■

Theorem 3.3. The state transition diagram for the process is acyclic.

Proof: The upper bound for any node is a non-increasing function, and the lower bound for any node is a non-decreasing function of t. Hence each element in the sequence of states through which a given node passes is unique. ■

Because the number of states is finite and the state transition diagram is acyclic, we have the following.

Theorem 3.4. Given that the number of nodes in the decision graph is finite, then the system will reach a terminal state in finite time, provided that the number of message initiations is finite. ■

[6] As noted above, it is reasonable to assume that $U(N_L) = L(N_L) = c(N_L)$ for any leaf node N, that is, that leaf nodes are fathomed upon generation. This is not required for this result, however.

Next we wish to show that when the system reaches a terminal state, the root node is fathomed. That is, the upper bound for the root node is equal to the lower bound. First we first show that the optimal decision graph is always generated.

Theorem 3.5. Let N_1 and N_2 be two nodes in the search graph such that N_1 is a parent of N_2 and that (1) N_1 and N_2 are both in an optimal decision graph, (2) there is a directed path in the current search graph from the root node to N_1 that is contained entirely within an optimal decision graph, and, (3) there is an arc from N_1 to N_2 in the current search graph. Then, the arc from N_1 to N_2 will never be removed and the node N_2 will never become inactive.

Comment: This result can be established [19] by induction on the depth of a node in the search graph. ■

Corollary 3.6. If the node N_2 in the conditions for theorem 3.5 is on the expansion queue, it will never be removed from the expansion queue.

Theorem 3.7. At any point in the operation of the distributed algorithm, as above, either (1) there is at least one processor, with at least one node in an optimal decision graph on its expansion queue, or (2) the optimal decision graph is generated entirely in the current search graph.

Comment: This result can be established by a simple inductive argument on the sequence $T = t_0, t_1, ..., t_n, ...$ of times at which the system changes state. ■

Theorem 3.8. The optimal decision tree is always generated in its entirety by the distributed algorithm.

Proof: This result follows directly from corollary 3.6 and theorem 3.7. ■

Theorem 3.9. Upon termination, every node in the optimal decision graph is fathomed.

Proof: The proof is again by induction on the separation from the root node to a particular node N in the optimal decision graph. For any leaf node, the result follows directly from theorem 3.8, that is, when the leaf node is generated it will be fathomed according to our basic assumptions as stated above.

For a node, N, at a sound $s-1$, consider the list, $N_1, N_2, \ldots, N_r$ of its successor nodes. Let N_i be the last of these successors to become fathomed. When that happens, a re-evaluate bounds message is sent to each of its predecessors, including node N. Upon receipt of the re-evaluate bounds message, the node N will re-evaluate its bounds. Since all of its offspring have been fathomed at this point, the node N itself becomes fathomed. ■

This result shows, in particular, that the root node is fathomed when the system reaches a terminal state.

3.2 Finite Message Initiation

Most of the discussion presented above has been predicated upon the fact that the message initiations are finite. We now show that this holds.

Theorem 3.10. In the case where the process generates a search tree, the number of messages of each type that are generated is finite.

Proof: In a search tree, each node will have at most one predecessor (parent). Because the upper bounds are non-increasing, and the lower bounds are non-decreasing, once a node becomes inactive in a search tree, it remains inactive for all future time. Hence, a node is added and removed from the parent list of an offspring node at most once. As such, there can be no more than one add predecessor and one remove predecessor message per node. Since, by assumption, the number of nodes is finite, the number of add and remove predecessor messages is finite.

A re-evaluate bounds message is sent either as a result of a state change or in response to an add-predecessor message. The number of states is finite, as are the number of add predecessor messages. Hence, the number of re-evaluate bounds messages that are initiated are finite. ■

Theorem 3.11. In the case where the process generates a decision graph, the number of message initiations of each type is finite.

Outline of Proof: Any directed acyclic graph can be "unwound" to a tree which is in some sense "equivalent" to the original graph. This is achieved by duplicating nodes with multiple parents, giving each parent an equivalent copy of the original node. This process proceeds recursively, starting at the root node and working down the graph until no nodes with multiple parents remain. Because the state vectors are duplicated when the nodes are, the decision process defined on the equivalent tree is the same as that defined on the original graph, in the sense that (1) the optimal cost associated with each node in the graph is the same as the optimal cost associated with the equivalent nodes in the decision tree, and (2) the *optimal* decision graph maps to the optimal decision tree.

In the strategy used to prove this theorem, we show that the number of message initiations are bounded by the number of message initiations for the equivalent search tree. Consider first the add predecessor messages. Assume that there there is a node, N which during the course of expanding the search graph receives two add predecessor messages from the same predecessor N_n (otherwise, there would be at most one add predecessor message sent along each arc in the decision graph, which is finite by assumption). From the conditions under which the add predecessor message is initiated (figure 1), it can be shown that for this to happen there must exist:

(1) a directed path from some ancestor N_1 of N_n along which least two sequences of add predecessor messages has been transmitted, leading to the two add predecessor messages from N_N to N, and,

(2) two distinct predecessors N_0 and $N_0{}'$ of N_1 such that an add predecessor message from N_0 to N_1 was responsible for generating the first of the two sequences, and an add predecessor message from $N_0{}'$ to N_1 was responsible for generating the second of the two sequences.

Because of the procedure used in generating the equivalent search tree from the search graph, the two sequences of add predecessor messages would therefore occur on distinct branches of the tree, one rooted at N_0 and the other rooted at $N_0{}'$. Using this fact as a basis, it can then be shown that there can be at most one message sequence in the graph for each message sequence in the corresponding tree, which, from theorem 3.10 establishes the result for the add predecessor messages.

The proof for the remove predecessor messages is similar. As in theorem 3.10, the fact that a finite number of re-evaluate bounds messages are sent is established from the fact that the add and remove predecessor message transmissions are finite and the assumption that the state space is finite. ■

Race Conditions and Deadlock. In general, race conditions created by messages between two nodes arriving out of sequence will not create a problem for the operation of the distributed algorithm being discussed here, although some precautions must be taken as outlined below.

First, if messages between two specific nodes are certain to arrive in the same order they are sent, then it is easy to show [19] that the process will always generate the correct result regardless of the order in which those messages arrive relative to messages sent by *other* nodes. In practice, it is difficult to guarantee this, unless special precautions are taken in the communication protocols. Arguments very similiar to that used in the proof of theorems 3.10 and 3.11, can be used to show that the order of arrival of re-evaluate bounds messages is still not important. But problems can occur in sequences of add and remove predecessor messages.

For example, consider a node N_1 and its offspring N_2, and a sequence of:

add predecessor, remove predecessor, add predecessor

which is assumed to be received as,

add predecessor, add predecessor, remove predecessor.

If the second add predecessor message is ignored by node N_2, then upon receipt of the remove predecessor message, node N_2 will remove the link back to N_1. Any subsequent re-evaluate bounds messages will therefore not be propagated from N_2 to N_1 and it is possible that the final result will be incorrect. In our implementation, these types of problems are managed by maintaining a count of the number of add predecessor messages minus the number of remove predecessor messages received from a given predecessor, in the successor node. If the count is zero or negative the successor considers itself to be pruned. Otherwise, it is not, and re-evaluate bounds messages are propagated back to it.

Deadlock does not occur in the context of the scheme presented here because the processor activities are driven by the work in the message handling and expansion queues, and no message explicitly requires a reply before its desired "effect" on the search graph is completed.

4 Results

The results reported here apply to a large scale resource allocation problem modeled as a Markov decision process. The specific application is to a problem in naval command and control. The process of solving the problem involves expanding a decision graph to enumerate all possible ways (in the worst case) in which resources can be allocated to tasks over time. The reader is referred to [20,21] for more detailed discussions of the application. The message passing scheme discussed above was implemented in "C" on a BBN Butterfly parallel processor[7]. The Butterfly machine consists of processor-memory units, with memory on any processor accessible to any other processor through a delta network as described in detail in several recent reports [22,23,24]. Although the Butterfly is a tightly coupled machine, our implementation simulates a loosely coupled environment that has a crossbar interconnection topology.

In our emulation of a loosely coupled architecture, each processor's outbound message packets are stored directly into a buffer on the receiving processor, then enqueued for processing. Memory access time across the switch has been reported to be 4 to 7 times slower than access to local memory, and contention for message buffer space (controlled by semaphores provided with the Butterfly's "Chrysalis" operating system) created other delays in the transmission of messages. However, no further delays were introduced into the simulated "channels". Although it might be argued that the results discussed here may not accurately reflect the situation that would be obtained in a truly loosely coupled environment, there are good arguments to the contrary as discussed further below. We are currently implementing the approach on a network of Transputer[8] processing elements, which should provide a resolution to this issue in the near future.

There are several different measures for the performance of parallel algorithms. We will use *relative efficiency of parallelism*, E_p, as defined in [25], which gives $E_p = \frac{S_p}{p}$. where S_p is the speedup factor over a single processor obtained by dividing the elapsed time of the message passing algorithm on one processor by that on p processors. We present results related to efficiency as a function of problem size (number of nodes in the search graph) and the number of processors in the system. To this end, experiments were conducted against a range of problem sizes, by varying the value of the major input parameters. Data was collected for a range of problems resulting in search graphs with 6 to 1832 nodes, with the number of processors ranging from 1 to 15. Memory limitations of the system configuration on which the initial experiments were conducted precluded the generation of larger search graphs, which would have been highly desirable[9]. Nonetheless, the trends from the data that are available are fairly clear, and hence we believe, will be consistent with results derived from larger experiments.

A search graph consisting of 1832 nodes requires about 133 seconds to generate on a single Butterfly processor unit (a BPN2 module which consists of a Motorola 68020/68881 processor), or about .072 seconds/node. The time required to generate a node in the search graph decreases with smaller problems in our application, primarily because the size of the state vector associated with each node is a function of the problem inputs. For example, a search graph consisting of 81 nodes requires 1.79 seconds on a single processor, or about .022 seconds/node.

Figure 4 shows efficiency as a function of nodes generated for various numbers of processors. Figure 5 shows efficiency as a function of number of CPUs for various fixed "problems" (as defined by the problem inputs). Note that both graphs exhibit trends that are more characteristic of a problem running on a pipelined architecture rather than a distributed asynchronous algorithm, the important factor apparently being the number of nodes per processor (figure 6).

[7] Butterfly is a trademark of Bolt, Beranek, and Newman, Advanced Computers Inc.

[8] Transputer is a trademark of INMOS, Inc.

[9] This is a reflection of the size of the state vector for nodes in our application, which tends to be very large. We will be conducting experiments and reporting results on larger systems in the near future.

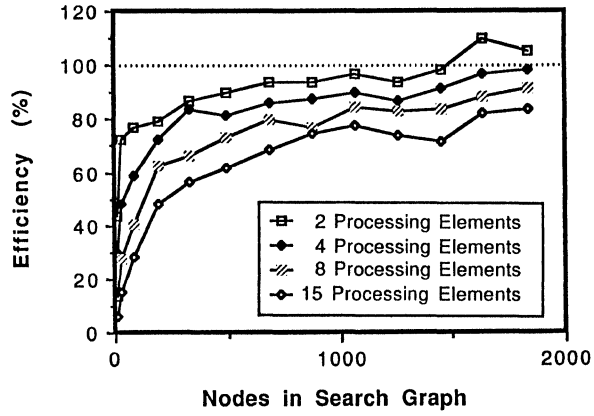


Figure 4: EFFICIENCY AS A FUNCTION OF THE NUMBER OF NODES IN THE SEARCH GRAPH FOR VARIOUS NUMBERS OF PROCESSING ELEMENTS IN THE SYSTEM.

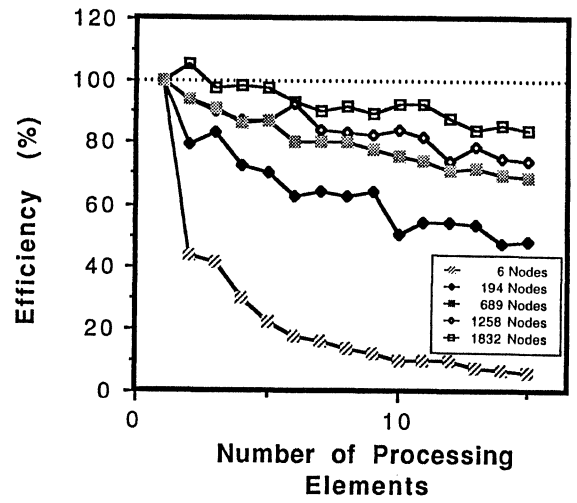


Figure 5: EFFICIENCY AS A FUNCTION OF THE NUMBER OF PROCESSING ELEMENTS FOR VARIOUS PROBLEM SIZES.

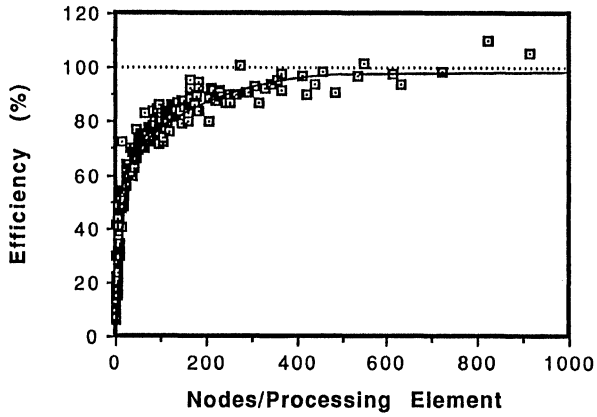


Figure 6: EFFICIENCY AS A FUNCTION OF THE NUMBER OF NODES PER PROCESSING ELEMENT.

5 Discussion

We have presented an overview of an approach to solving sequential decision problems in a loosely coupled environment. The approach is based on a message passing scheme, using three basic message types. We have outlined a proof of the correctness of the algorithm and discussed issues related to race conditions that can potentially occur in the passing of messages. We have demonstrated initial results that indicate that the approach yields a very high degree of processor utilization for problems of a sufficient size.

Note how efficiency asymptotically approaches 100% as the number of nodes/processor increases. Boundary conditions existing during the early stages of the generation of the search graph (that is, when the first few nodes are being generated, and the message and expansion queues for all but a few processors are empty) dominate when this ratio is small, yielding poor relative efficiency, but become less of a factor when the search graph is very large. The initialization effect is further exacerbated by the hashing scheme for distributing nodes among the various processors, which tends to provide a more uniform distribution of nodes to processors as the number of nodes in the search graph becomes large. For small problems, even small variances in the number of nodes per processor can mean that the expansion and message queues of a processor become empty so that the processor becomes idle.

It has often been reported that in many distributed algorithms implemented on loosely coupled architectures, communication overhead becomes the dominant factor as the problem size and number of processors grow large. This effect depends on (1) the extent to which communication between processors introduces a synchronizing or "serial" component to the process, (2) the relative amount of processor overhead required for transporting and routing messages, and (3) the extent to which communication bottlenecks occur. The first factor is not a significant issue in our approach, because no explicit synchronization is necessary. Furthermore, except for the initial phase of search graph generation, each processor maintains a "back-log" of work, so that the amount of time a message takes to transit the network should not have a significant impact on overall efficiency. This explains our conjecture that overall efficiency will not be seriously degraded in an implementation on an actual loosely coupled system, at least as far as our application is concerned. The effect of the second factor can obviously be minimized by an appropriate hardware design, which off-loads message routing responsibilities from the processors. The last factor is dependent on the interconnection topology of the system.

In determining the significance of the approach discussed here, it is important to also consider how it compares to the best, or even commonly available, single processor algorithms[10]. We find that the message passing scheme proposed here when run on a *single* processor runs approximately two thirds as fast as the best single processor scheme that we have developed to date, in which costs and bounds are recursively backed up the graph. Thus, for example, the distributed algorithm runs 10 times faster on 15 processors than a "good" sequential algorithm running on a single processor. The ratio has been observed to be more or less constant, independent of problem size. Furthermore, we have found that the ordering of nodes on the expansion queue (producing depth first, breadth first, and by various criteria, best first search) has very little effect on the run times for the recursive version of the algorithm.

By profiling the runs, we have determined that the difference is in the extra time required to build and handle the messages, which in our application involves a certain amount of effort to encode the state vector associated with each node, allocate and deallocate message buffer space, and so forth. Furthermore, the codes for handling messages have been implemented relatively recently and have not been optimized to the same extent as the codes for the recursive version. The important factor is that the extra processing time is (1) a constant factor, and hence does not change the relative *order* of complexity of the algorithm, and (2) is due to processing *local to each node*. It is not, for example, the result of a serial component or a synchronization effect inherent in the design of the approach. It would be a penalty incurred by any approach for distributing the work load on a loosely coupled system by allocating nodes uniformly among the processors, since there is no other way in which the information about the state of an offspring node can be conveyed to a processor responsible for that offspring. Although several approaches have been proposed which maintain subsections of search trees local to a specific processor [12,26], they would not extend to processes defined on directed acyclic graphs, particularly if little is known about the structure of the graph at the outset, because a given node may have (and in our case, does have) predecessors from diverse sections of the graph. Furthermore, approaches that maintain locality have other disadvantages, including redundant search, communication, and synchronization overhead.

The reader has no doubt noted the fact that efficiencies greater than 100% are obtained in some cases. This effect has also been observed for other types of parallel search algorithms [6,7,12]. The explanation that has often been given, is that a fortuitous ordering of nodes on the expansion queue finds an optimal solution earlier in a multiprocessor environment. This would probably not be the best explanation of this anomaly in our case. Among other things, we find that the number of nodes generated in the search graph tends to be fairly constant[11], independent of the number of processors in the system. On the other hand, we have observed a signficant decrease – up to 37% – in the number of re-evaluate bounds messages for a fixed problem size as the number of processors are increased, and in fact, this decrease, factored by the amount of time required to generate and handle a re-evaluate bounds message explains the difference in elapsed time very well. One explanation for the reduction in the number of these messages is that the more parallelism that is employed in propagating bound constraining information between a node and a given ancestor, the faster its bounds become constrained. If the ancestor node is ultimately to be pruned, it happens sooner, on average, in the multiprocessor implementation and hence the channel between that ancestor and any of *its* predecessors becomes closed to any further message traffic earlier in the search graph generation process. We suggest that further research of this phenomenon is warranted.

[10] To prove that a given algorithm for a complex process is optimal is a difficult task at best, so let us suffice with a discussion of "best commonly available".

[11] typically, within 0.3 percent

References

[1] N. Nilsson, *Principles of Artificial Intelligence.* Palo Alto: Tioga Press, 1980.

[2] C. Derman, *Finite State Markovian Decision Processes.* New York: Academic Press, 1970.

[3] J. Shapiro, *Mathematical Programming: Structures and Algorithms.* New York: Wiley-Interscience, 1979.

[4] T. Ibaraki, "Generalization of alpha-beta and SSS* search procedures," *Art. Intell.*, vol. 29, pp. 73–117, 1986.

[5] S. Akl, D. Barnard, and R. Doran, "Design, analysis, and implementation of a parallel tree search algorithm," *IEEE Trans. on Pat. Anal. and Mach. Intell.*, vol. PAMI–4, pp. 192–203, March 1982.

[6] G. Baudet, *The Design and Analysis of Algorithms for Asynchronous Multiprocessors.* PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, April 1978.

[7] R. Finkel and J. Fishburn, "Parallelism in alpha-beta search," *Art. Intell.*, vol. 19, pp. 89–106, 1982.

[8] R. Finkel and J. Fishburn, "Improved speedup bounds for parallel alpha-beta search," *IEEE Trans. on Pat. Anal. and Mach. Intell.*, vol. PAMI-5, pp. 89–92, 1983.

[9] J. Fishburn, *Analysis of Speedup in Distributed Algorithm.* Ann Arbor, Michigan: UMI Research Press, 1984.

[10] V. Kumar and L. Kanal, "Parallel branch and bound formulations for AND/OR tree search," *IEEE Trans. on Pat. Anal. and Mach. Intell.*, vol. PAMI-6, pp. 768–778, 1984.

[11] T. Marsland and M. Campbell, "Parallel search of strongly ordered game trees," *Computing Surveys*, vol. 14, pp. 533–551, 1982.

[12] R. Finkel and U. Manber, "DIB—a distributed implementation of backtracking," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 235–256, April 1987.

[13] G. Lipovski and M. Hermenegildo, "B-log: a branch and bound methodology for the parallel execution of logic programs," in *Proceedings of the 1985 IEEE International Conference on Parallel Processing*, (Penn. State Univ.), pp. 560–567, August 1985.

[14] H. Berliner, "The B* tree search algorithm: a best-first proof procedure," *Art. Intell.*, vol. 12, pp. 23–40, 1979.

[15] M. Campbell and T. Marsland, "A comparison of minimax tree search algorithms," *Art. Intell.*, vol. 20, pp. 347–367, 1983.

[16] N. Darwish, "A quantitative analysis of the alpha-beta pruning algorithms," *Art. Intell.*, vol. 21, pp. 405–433, 1983.

[17] A. Palay, "The B* tree search algorithm: new results," *Art. Intell.*, vol. 19, pp. 145–163, 1982.

[18] I. Roizen and J. Pearl, "A minimax algorithm better than alpha-beta? yes and no," *Art. Intell.*, vol. 21, pp. 199–220, 1983.

[19] M. Diamond, J. Kimbel, and J. Newhouse, "A multiprocessor implementation for polyadic sequential dynamic programming problems," Tech. Rep., FMC Corporation Advanced Systems Center Technical Report, Minneapolis, MN, 1988.

[20] M. Diamond and O. Carducci, "Decision processes for large scale resource allocation problems," in *Proc. 8^{th} MIT/ONR Workshop on C^3I Systems*, (Cambridge, MA), July 1984.

[21] M. Diamond, J. Newhouse, and J. Kimbel, "Decision processes for large scale resource allocation problems — extensions and results," in *Proce. JDL 1987 Symposium on C^2 Research*, (Washington, DC), June 1987.

[22] *Proceedings of the Fall 1987 Butterfly Users Group Meeting*, (Cambridge, MA), BBN Advanced Computers, October 1987.

[23] *Inside the Butterfly Plus.* BBN Advanced Computers, Cambridge, MA, October 1987.

[24] T. Leblanc and S. Jain, "Crowd control: coordinating processes in parallel," in *Proceedings of the 1985 IEEE International Conference on Parallel Processing*, (Penn. State Univ.), pp. 434–441, August 1985.

[25] D. Parkinson, "Practical parallel processors and their uses," in *Parallel Processing Systems*, (D. Evans, ed.), Cambridge, MA: Cambridge University Press, 1982.

[26] T. Marsland and F. Popowich, "Parallel game tree search," *IEEE Trans. on Pat. Anal. and Mach. Intell.*, vol. PAMI–7, no. 4, pp. 442–452, 1985.

SPACE-EFFICIENT AND FAULT-TOLERANT MESSAGE ROUTING IN OUTERPLANAR NETWORKS

(Preliminary version)

Greg N. Frederickson [1]

Department of Computer Science
Purdue University
West Lafayette, IN 47907

Ravi Janardan [2]

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

Abstract

A fault-tolerant routing scheme for outerplanar networks is presented, which stores routing information succinctly and routes messages along near-shortest paths. For an n-node network containing t node and edge faults, the total space and communication is $O(t\alpha n)$ and the routings generated are within a factor of $((\alpha+1)/(\alpha-1))^t$ of optimal, where $\alpha > 1$ is an odd-valued integer parameter. Thus the routings can be tuned as desired. Efficient algorithms are given for setting up the routing scheme.

1. Introduction

A primary function in a distributed network is the routing of message between pairs of nodes. Often, a cost is associated with each edge, making it desirable to route along shortest, or near-shortest, paths. Although this can be accomplished easily by storing a complete routing table at each of the n nodes of the network, such an approach is expensive, using a total of $\Theta(n^2)$ items of routing information, where each item is a node name. Thus, recent research has focused on reducing the amount of routing information stored, while still retaining good routings. Compact routing schemes have been designed for numerous classes of networks, ranging from simple networks such as trees, rings, complete networks, and complete bipartite networks [8,9,10] to more complex networks that possess a certain embedding property (the simplest of which are the outerplanar networks) [3], and to networks exhibiting certain separator properties, such as the c-decomposable networks and planar networks [4]. These schemes examine the problem in the context of being free to assign suitable short names to the nodes at the time the network is set up. The idea here is to encode useful information about the network within the names and to then use this information to generate good routings. All the above schemes use considerably less space than complete routing tables, keep node names to $O(\log n)$ bits, and still route along shortest or near-shortest paths.

The problem of compacting routing information has implications that go beyond merely saving space in the network. The study of this problem has led to several new insights into the issues of naming nodes and compactly encoding information within node names [3,4], as well as to fast sequential algorithms for computing all-pairs shortest paths in planar graphs [2].

Unfortunately, none of the above routing schemes can handle node and edge faults, which can invalidate the stored routing information and result in arbitrarily bad routings. Although the problem can be overcome by recomputing the routing information for the resulting network from scratch, this approach can involve as much as $\Theta(n^2)$ communication overhead even for sparse networks containing a single fault. It is thus desirable to design compact routing schemes which can adapt efficiently to faults and still route well.

In this paper we present a space- and communication-efficient routing scheme for the class of outerplanar networks. An *outerplanar network* is a network which can be embedded in the plane so that all nodes lie on the boundary of a single face, usually the exterior face [7]. In our approach, the network adapts to faults by distributively computing a small amount of *additional routing information* which is then used in conjunction with compact routing information for the original network to restore good routings. For any combination of node and edge faults that do not disconnect the network, our scheme restores near-optimal routings using only a constant amount of additional routing information per fault. Specifically, let t be the number of faults and $\alpha > 1$ an odd-valued integer parameter. Then our scheme uses a total of $O(t\alpha n)$ items of additional routing information and generates routings which are, in worst-case, at most $((\alpha+1)/(\alpha-1))^t$ times longer than optimal. (Note that this bound is less than $(\alpha+t)/(\alpha-t)$, for $\alpha \geq t$.) Thus the routings can be made as close to optimal as desired by choosing α appropriately large. Furthermore, the additional routing information is computed efficiently using only $O(t\alpha n)$ messages.

Briefly, our approach is as follows. The worst-case occurs when all faults are interior edges, i.e., edges not on the exterior face. (As we shall see, optimal routings can be reinstated if all the faults are nodes and exterior edges.) There are now essentially two candidate paths. To choose between them we make use of an interesting *monotonicity property* of distance differences in outerplanar networks. Using this property, for each failed edge we suitably partition the exterior face boundary into α segments. Information about these segments is stored at each node and the routing from a source to a destination is performed based on the relative positions of the segments containing the two nodes.

[1] Research supported in part by NSF grant CCR-86202271 and ONR contract N 00014-86-K-0689.

[2] Research supported in part by NSF grant DCR 8320124.

A noteworthy feature of our scheme is that if there is just one interior edge fault among the t faults, then the additional information can, in fact, be precomputed at the time the network is set up. For each interior edge, information about the α segments is precomputed to handle the potential failure of the edge. This information is stored at the endpoints of the edge and is broadcast through the network when the edge fails. We give an efficient sequential algorithm to precompute this information for *all* interior edges in $O(\alpha n \log n)$ time.

Throughout we model our network by an undirected graph. (For graph-theoretic terms not defined here, see [7].) In the next section we review the compact routing scheme for fault-free outerplanar networks presented in [3], which is a component of our fault-tolerant scheme. Section 3 describes the fault-tolerant routing scheme. Due to space constraints, most proofs are either omitted or abbreviated in this preliminary version.

2. Interval routing in outerplanar networks

We first summarize the interval routing method presented in [8] for trees and rings. The nodes are named appropriately with the integers from 1 to n. For trees, the names are depth-first numbers. For rings, the names are assigned consecutively, going clockwise around the ring. For any vertex v of degree d, let $w_1, w_2, \ldots, w_d$ be the neighbors of v indexed in clockwise order around the exterior face starting from v. Each edge incident with v is labeled by an interval, with the intervals from all edges incident with v forming a partition of $[1,n]-v$. Wraparound is allowed in the intervals. For instance, the interval $[i,j)$, $i > j$, contains $\{i, i+1, \ldots, n, 1, \ldots, j-1\}$. Denote the intervals by $[l_i, l_{i+1})$, for $i = 1, 2, \ldots, d$, where $l_{d+1} = v$, and let interval $[l_i, l_{i+1})$ label edge $\{v, w_i\}$. This interval labeling has the property that $\{v, w_i\}$ is the first edge on a shortest path from v to any node whose name is in $[l_i, l_{i+1})$. The values l_i, $i = 1, 2, \ldots, d$, are stored in a table at node v, each with a pointer to associated edge $\{v, w_i\}$. When a message arrives at node v, if its destination u is not equal to v, then the table is searched for the entry l_i such that $l_i \leq u < l_{i+1}$. The message is then sent out on edge $\{v, w_i\}$. Since the values l_i, $i = 1, 2, \ldots, d+1$ form a rotated list [6,1], the table can be searched in $O(\log d)$ time using a modified binary search.

As the following theorem shows, the interval labeling method also works for outerplanar networks under a suitable naming of the nodes [3]. The nodes are assigned integer names from 1 to n in consecutive order by proceeding clockwise around the exterior face; if any node v is visited more than once in this traversal, implying that v is an articulation point of the network, then v may be named on any one of the visits. We call such a naming of the nodes a *clockwise* node naming. An outerplanar network with a clockwise node naming is shown in Figure 1.

Theorem 1. ([3]) Let G be an n-vertex outerplanar graph with a clockwise naming of its vertices. For any assignment of nonnegative costs to its edges, the end of every edge incident with any vertex v can be labeled with a subinterval of $[1,n]$ such that the edge is the first edge on a shortest path from v to any vertex in the subinterval. ∎

Figure 2 illustrates an interval labeling of the edges of the network in Figure 1.

In our fault-tolerant scheme, shortest paths routing information for the original (fault-free) outerplanar network, G, is stored in interval form. Moreover, the interval information is set up to favor an edge to a path of two or more edges, in the case of ties. Also, we assume that the edge costs of G satisfy the *generalized triangle inequality*, i.e., each edge is a shortest path between its endpoints.

3. Compact fault-tolerant outerplanar routings

We consider node faults and edge faults separately and later show how to handle combinations of node and edge faults. As a running example, we will use the unit-cost network G shown in Figure 3.

3.1 Handling node faults

Consider a single node fault. Let v be the failed node and S the set of edges which are on the faces containing v but not incident with v. The edges of S form a simple path and graph $G - v$ is the union of a number of subgraphs, each defined by an edge of S, as follows. Let $\{a, b\}$ be any edge of S such that v is in the interval (a, b). Then the subgraph defined by $\{a, b\}$ is the induced subgraph of $G-v$ on the nodes in the interval $[b, a]$. Call each such subgraph an S-component, and the nodes a and b the *gateway nodes* of the S-component. Note that the nodes in the subgraph attached to a are in the interval $[a, v)$ and those in the subgraph attached to b are in the interval $(v, b]$.

Figure 4 shows the network of Figure 3 after the failure of node 5. The edges in S are $\{4,2\}$, $\{2,1\}$, $\{1,8\}$, and $\{8,6\}$, shown bold. The corresponding S-components have nodes in the intervals $[2,4]$, $[1,2]$, $[8,1]$, and $[6,8]$ respectively.

When v fails, each node w in $G - v$ determines the following additional routing information: the name v, the names of the nodes in each S-component H to which w belongs, and the names of the nodes in the subgraphs attached to each gateway node of H. This is done as follows. One of the endpoints of the simple path formed by the edges of S, which is a neighbor of v, sends out over the path a message containing the name v. Each node on the path forwards the message out on the path, and also broadcasts its own name and the name v within each of the S-components to which it belongs. Thus w receives the name v and the names a and b of the gateway nodes of H. These names v, a, and b succinctly represent the desired additional routing information for w. Since any node is in at most two S-components, $O(1)$ items of additional routing information are stored per node, hence $O(n)$ overall. The total number of messages exchanged is $O(n)$.

As an example, node 11 in Figure 4 receives the names 1 and 8 of the gateway nodes of its S-component, and the name 5 of the failed node. From these it deduces that its S-component consists of the nodes in the interval $[8,1]$, and that the nodes in the subgraphs attached to gateway nodes 1 and 8 are in the intervals $[1,5)$ and $(5,8]$ respectively.

The routing from a source s to a destination d is as

follows. Let u be any node participating in the routing, inclusive of s and d. If u is d then the routing terminates. Otherwise, if u and d are in the same S-component, then u routes to d using interval routing. Otherwise, u uses interval routing to route to that gateway node of its S-component such that d is in the subgraph attached to this gateway node.

The routing from u to d will be along a shortest path in $G - v$. Suppose that u and d are in the same S-component. Since the edge from S contained in this component satisfies the generalized triangle inequality, the distance between u and d in $G - v$ equals the distance between them in G. Furthermore, since the interval routing information favors an edge to a path of two or more edges, the routing from u to d will be along a shortest path confined to the component. If u and d are in different S-components, then any shortest (u, d)-path must use that gateway node of u's S-component to which the subgraph containing d is attached. The routing strategy correctly determines this gateway node and routes to it along a shortest path.

A similar approach works for $t > 1$ node faults also. Let $v_1, v_2, \ldots, v_t$ be the failed nodes. Let S_j be the set of edges on the faces containing v_j but not incident with v_j, $1 \leq j \leq t$, and let $S = \bigcup_{j=1}^{t} S_j$. Graph $G' = G - \{v_1, v_2, \ldots, v_t\}$ is the union of a number of S-components, where an S-component can now contain more than one edge from S, but at most one from any S_j. Each S-component H is defined by its edges of S as follows: Suppose that H contains an edge from each of the sets $S_{i_1}, S_{i_2}, \ldots, S_{i_l}$, $1 \leq l \leq t$, where the sets are indexed such that $v_{i_1} < v_{i_2} < \cdots < v_{i_l}$. Let $\{a_{i_j}, b_{i_j}\}$ be the edge from S_{i_j} in H such that v_{i_j} is in the interval (a_{i_j}, b_{i_j}), $1 \leq j \leq l$. Then, H is the induced subgraph of G' on the nodes in $[b_{i_1}, a_{i_2}] \cup [b_{i_2}, a_{i_3}] \cup \cdots \cup [b_{i_l}, a_{i_1}]$. The gateway nodes of H are the nodes a_{i_j} and b_{i_j}, $1 \leq j \leq l$. The nodes in the subgraph attached to any gateway node u of H are as follows. In general, u will be an endpoint of more than one edge from $S_{i_1}, S_{i_2}, \ldots, S_{i_l}$, and so receives labels of both types 'a' and 'b'. For each label a_{i_j} that u receives, $1 \leq j \leq l$, a subset of the nodes in the subgraph attached to it are contained in the interval $[a_{i_j}, v_{i_j})$. Similarly, for each label b_{i_j} that u receives, a subset of the nodes in the subgraph attached to it are contained in the interval $(v_{i_j}, b_{i_j}]$. The set of all nodes in the subgraph attached to u will be the union of these intervals.

As additional routing information, each node w of G' determines the names of the failed nodes, the composition of each S-component H containing w, and the composition of the subgraphs attached to the various gateway nodes of H. A recovery message, containing the name v_{i_j}, is sent out on the simple path defined by S_{i_j}, and each node a_{i_j} and b_{i_j} broadcasts its own name and the name v_{i_j} into H, $1 \leq j \leq l$. The names v_{i_j}, a_{i_j}, and b_{i_j} succinctly represent the additional routing information for w. The total number of message exchanges within H will be $O(t|E(H)|)$. Thus there are $O(tn)$ message exchanges overall. Each node maintains $O(t)$ items of additional routing information for each S-component of G' to which it belongs. Since a node can belong to at most as many S-components as its degree in G', the total number of items is $O(t|E(G')|)$, which is $O(tn)$.

The routing strategy is exactly as before and the routings generated will be optimal.

Figure 5 illustrates the network of Figure 3 after the failure of nodes 5 and 13, with the edges of S shown bold. Let H be the S-component consisting of nodes 8, 9, 10, and 1. H contains edges $\{a_{i_1}, b_{i_1}\} = \{1, 8\}$ and $\{a_{i_2}, b_{i_2}\} = \{10, 1\}$ from the sets S_{i_1} and S_{i_2} corresponding to $v_{i_1} = 5$ and $v_{i_2} = 13$. Thus each node of H receives the names 1 and 8, and the name of the associated failed node 5, and also the names 10 and 1, and the name of the associated failed node 13. From this it deduces that the nodes of H are in $[8, 10] \cup [1, 1]$. The nodes in the subgraph attached to 1 are in $[1, 5) \cup [13, 1)$, those in the subgraph attached to 8 in $(5, 8]$, and those in the subgraph attached to 10 in $[10, 13)$.

3.2 Handling edge faults

An exterior edge fault can be viewed as the failure of a fictitious node in the middle of the edge. Thus this case can be handled essentially as before, with one of the endpoints of the failed exterior edge initiating the recovery.

Interior edge faults are more difficult to handle, however. Unlike node and exterior edge faults where there is essentially just one choice of path to route over, there are now two candidate paths. This introduces nonoptimality in the routings, since the correct choice of path is not always apparent. However, for the routing scheme that we present, the quality of the routings can be improved to any desired extent by using a correspondingly larger amount of additional routing information.

First consider a single interior edge fault. We present two approaches. In the first, additional routing information is precomputed for each interior edge at network setup time, to handle the potential failure of the edge, and is stored at one of the endpoints of the edge. We also give an efficient algorithm for precomputing this information. We then describe an alternative approach where the additional information is computed distributively, as the faults occur. We then build upon the second approach to handle multiple interior edge faults.

3.2.1 Handling a single interior edge fault

Let $e = \{v_1, v_2\}$ be the failed edge and F the boundary of the face that results from the deletion of e from G. Graph $G - e$ is the union of a number of subgraphs, each defined by an edge of S, as follows. Let $\{a, b\}$ be any edge of S, with a immediately following b clockwise around F. Then the subgraph defined by $\{a, b\}$ is the induced subgraph of $G - e$ on the nodes in the interval $[b, a]$. Each such subgraph is called an *F-component* and the nodes a and b are called *gateway nodes* of the F-component.

Figure 6 shows the network of Figure 3 after interior edge $\{1, 8\}$ has failed. The edges $\{5, 1\}$, $\{8, 5\}$, $\{10, 8\}$, and $\{1, 10\}$ of F are shown bold. The corresponding F-components are the subgraphs induced on the nodes in the intervals $[1, 5]$, $[5, 8]$, $[8, 10]$, and $[10, 1]$ respectively.

Assume that v_1 initiates the recovery. This involves propagating additional routing information to each node. This information consists of the name v_1, the names of the gateway nodes of the F-component containing the node, and information about α points on the boundary of the exterior face, each at a prescribed distance from v_1 in $G-e$. The latter is precomputed and stored at v_1. Thus α items are stored for each interior edge, hence $O(\alpha n)$ in total. The total number of message exchanges is $O(\alpha n)$.

The information precomputed at v_1 is as follows: For any nodes x and y in $G-e$, let x_c and y_c be the first and last nodes of F encountered when going clockwise from x to y around the exterior face. If no node from F is encountered, then take x_c and y_c to be x and y respectively. A *shortest clockwise path from x to y* consists of a shortest path in $G-e$ from x to x_c, followed by the edges of F going clockwise from x_c to y_c, followed by a shortest path in $G-e$ from y_c to y. Denote by $\rho_c(x,y)$ the length of this path. A routing along this path is called a *shortest clockwise routing from x to y*. Such a routing is easy to perform, since the shortest paths from x to x_c and y_c to y can be realized using the interval routing information for G. Similarly for a *shortest counterclockwise path/routing from x to y*, whose length we denote by $\rho_{cc}(x,y)$.

The following lemma, which is a special case of a more general result proved in [2], reveals an interesting monotonicity property of distance differences in outerplanar networks.

Lemma 1. View the boundary of the exterior face of the embedding of $G-e$ as a continuum of points and extend the distance functions $\rho_c(\cdot,\cdot)$ and $\rho_{cc}(\cdot,\cdot)$ to points. The function $\rho_c(p,v_1)-\rho_{cc}(p,v_1)$ is monotonically nondecreasing for points p encountered going counterclockwise from v_1 around the exterior face. ∎

Let C be the total cost of the edges of F. Thus $\rho_c(\cdot,v_1)-\rho_{cc}(\cdot,v_1)$ changes monotonically from $-C$ to C in traversing the exterior face boundary counterclockwise from v_1. Let $\alpha > 1$ be an odd integer parameter. Precompute and store at v_1 the α *division points* $v_1 = z_0, z_1, \ldots, z_{\alpha-1}$, lying on the boundary of the exterior face, where the points are indexed counterclockwise from v_1. Division point z_i satisfies $\rho_c(z_i,v_1)-\rho_{cc}(z_i,v_1) = -C+(2C/\alpha)i$, $0 \le i \le \alpha-1$. It is represented as a 3-tuple (i,u,v), where u and v are the endpoints of the exterior edge $\{u,v\}$ on which it falls. (If it coincides with a node then u and v are both the name of this node.)

We illustrate division points with reference to Figure 6, with $\alpha = 3$. Division point z_0 coincides with node $v_1 = 1$; z_1 lies on edge $\{9,10\}$, at a distance of $5/6$ from node 10; and, z_2 lies on edge $\{5,6\}$, at a distance of $1/6$ from node 6.

The routing from a source s to a destination d is as follows. If s and d are both in the same F-component then s routes using the interval routing information for G. Otherwise, s uses the division points to route to d. Let s be between $z_{(k-1) \bmod \alpha}$ and z_k, and d between $z_{(k'-1) \bmod \alpha}$ and $z_{k'}$, where $0 \le k,k' \le \alpha-1$. In general, if a vertex coincides with a division point z_i, then we take the vertex to be between $z_{(i-1) \bmod \alpha}$ and z_i. If $k \le k'$ (i.e., s precedes d counterclockwise from v_1), then if $k'-k \ge (\alpha+1)/2$, then s performs a shortest clockwise routing to d; otherwise it performs a shortest counterclockwise routing. If $k' \le k$ (i.e., d precedes s), then if $k-k' \ge (\alpha+1)/2$, then s performs a shortest counterclockwise routing to d; otherwise it performs a shortest clockwise routing.

Theorem 2. Let $\rho(s,d)$ be the distance in $G-e$ between any nodes s and d, and let $\hat{\rho}(s,d)$ be the length of the routing generated by the above strategy. Then $\hat{\rho}(s,d)/\rho(s,d) < (\alpha+1)/(\alpha-1)$.

Proof. If s and d are in the same F-component, then the routing will be along a shortest (s,d)-path. Suppose that s and d are in different F-components and $\hat{\rho}(s,d) > \rho(s,d)$. We claim that $\hat{\rho}(s,d)-\rho(s,d) < C/\alpha$. First we note that from the positions k and k' of s and d, we have

$$\begin{aligned} -C+(k-1)(2C/\alpha) &< \rho_c(s,v_1)-\rho_{cc}(s,v_1) \\ &\le -C+k(2C/\alpha) \\ -C+(k'-1)(2C/\alpha) &< \rho_c(d,v_1)-\rho_{cc}(d,v_1) \\ &\le -C+k'(2C/\alpha). \end{aligned}$$

The proof of the claim involves considering various cases. Suppose that $k \le k'$ (i.e., s precedes d counterclockwise from v_1), and $k'-k \ge (\alpha+1)/2$. Let u be the first node from F encountered going counterclockwise from s to d. Note that u exists since s and d are in different F-components. Clearly, $\rho_{cc}(s,v_1) = \rho_{cc}(s,u)+\rho_{cc}(u,v_1)$, and $\rho_c(d,v_1) = \rho_c(d,u)+\rho_c(u,v_1)$. Substituting into the above inequalities gives

$$\begin{aligned} -C+(k-1)(2C/\alpha) &< \rho_c(s,v_1)-\rho_{cc}(s,u)-\rho_{cc}(u,v_1) \\ &\le -C+k(2C/\alpha) \\ -C+(k'-1)(2C/\alpha) &< \rho_c(d,u)+\rho_c(u,v_1)-\rho_{cc}(d,v_1) \\ &\le -C+k'(2C/\alpha). \end{aligned}$$

Note that $\rho_{cc}(s,d) = \rho_{cc}(s,u)+\rho_{cc}(u,d) = \rho_{cc}(s,u)+\rho_c(d,u)$, $\rho_c(s,d) = \rho_c(s,v_1)+\rho_c(v_1,d) = \rho_c(s,v_1)+\rho_{cc}(d,v_1)$ and $\rho_{cc}(u,v_1)+\rho_c(u,v_1) = C$. By the routing strategy, if $k'-k \ge (\alpha+1)/2$, then $\hat{\rho}(s,d) = \rho_c(s,d)$. Thus $\rho(s,d) = \rho_{cc}(s,d)$. Then

$$\begin{aligned} \hat{\rho}(s,d)-\rho(s,d) &= \rho_c(s,d)-\rho_{cc}(s,d) \\ &= (\rho_c(s,v_1)+\rho_{cc}(d,v_1))- \\ &\quad (\rho_{cc}(s,u)+\rho_c(d,u))+ \\ &\quad (C-\rho_{cc}(u,v_1)-\rho_c(u,v_1)) \\ &< (-C+k(2C/\alpha))- \\ &\quad (-C+(k'-1)(2C/\alpha))+C \\ &\le C-((\alpha+1)/2)(2C/\alpha)+(2C/\alpha) \\ &= C/\alpha. \end{aligned}$$

Similarly for the case $k'-k < (\alpha+1)/2$ and for $k' \le k$, which establishes the claim.

Furthermore, it can be shown that whenever $\hat{\rho}(s,d) > \rho(s,d)$, then $\hat{\rho}(s,d) + \rho(s,d) = \rho_c(s,d) + \rho_{cc}(s,d) \geq C$. Since $\hat{\rho}(s,d) - \rho(s,d) < C/\alpha$, it follows that $\rho(s,d) > (C - C/\alpha)/2$. Thus

$$\begin{aligned}\hat{\rho}(s,d)/\rho(s,d) &< (\rho(s,d) + C/\alpha)/\rho(s,d) \\ &< 1 + (C/\alpha)/((C - C/\alpha)/2) \\ &= (\alpha+1)/(\alpha-1). \blacksquare\end{aligned}$$

From the proof it follows that the routing may not be shortest only when $|k' - k| = (\alpha-1)/2$ and $|k' - k| = (\alpha+1)/2$. We illustrate the routing strategy using Figure 6. Let s be 12 and d be 7; thus $k = 1$ and $k' = 2$. Since s precedes d counterclockwise around the exterior face from $v_1 = 1$ and $k' - k < (\alpha+1)/2 = 2$, a shortest counterclockwise routing is performed and the message reaches d on a shortest path in the network, via nodes 10 and 8. As another example, let s be 2 and d be 9; thus $k = 3$ and $k' = 2$. As d precedes s counterclockwise from v_1 and $k - k' < (\alpha+1)/2 = 2$, a shortest clockwise routing is performed. The message reaches 9 on a path of length 4, via nodes 4, 5, and 8. The shortest path is via nodes 1 and 10, and has length 3. Thus $\hat{\rho}(2,9)/\rho(2,9) = 4/3 < (\alpha+1)/(\alpha-1) = 2$.

3.2.2. Determining division points efficiently

The division points for each interior edge can be computed easily in $O(n)$ time, thus yielding an $O(n^2)$-time algorithm for finding the division points for all interior edges. However, one can do better. We now present an algorithm which runs in $O(\alpha n \log n)$ time. Our algorithm decomposes G into two smaller graphs, recursively solves the problem on these graphs, and then combines the two solutions into one for G. The decomposition is done by identifying a pair of vertices, called *separator vertices*, whose removal disconnects G into two subgraphs with vertex sets A and B, each of size at most $2n/3$. The separator vertices can be found in $O(n)$ time; see [5] for example, where a separator algorithm for the more general classes of series-parallel and k-outerplanar graphs is given. The division points algorithm is as follows.

Assume that G is biconnected. (Otherwise apply the algorithm to each biconnected component.) If G is a cycle then there are no division points to compute (as there are no interior edges) and the algorithm terminates. Otherwise, assign weight $1/n$ to each vertex of G and find separator vertices x and y of G. Infer the induced subgraphs G_1 and G_2 of G on the sets $A \cup \{x,y\}$ and $B \cup \{x,y\}$ respectively. If x and y are not adjacent in G, then include a path of length two between them in G_1 and in G_2, with cost as follows. Let F be the face boundary in G containing both x and y. Let F_1 (resp. F_2) be the portion of F contained in G_1 (resp. G_2). Then in G_1 (resp. G_2), assign cost $\| F_1/2 \|$ (resp. $\| F_2/2 \|$) to each edge of the path between x and y. Both G_1 and G_2 will be biconnected outerplanar graphs satisfying the generalized triangle inequality. Also, distances between nodes in G_1 (resp. G_2) will be unchanged from G.

Recursively solve the problem on G_1 and G_2. Complete the solution for G as follows. Let P be the path joining x and y in G_1 (resp. G_2). If x and y are adjacent in G, then P is e; otherwise it is the introduced path of length two. Let p be any division point for interior edge e' of G_1 (resp. G_2) that falls in the interior of P. Map p to a point on the portion of the exterior face boundary of G contained in G_2 (resp. G_1), such that the latter point is a division point for e' viewed as an interior edge of G. Furthermore, if P is e, then compute the division points for e as well.

We first show how to map each point p originating in G_1 and falling in the interior of P to the boundary of the exterior face of G contained in G_2 (the discussion for points originating in G_2 is similar). For convenience extend the distance functions $\rho_c(\cdot,\cdot)$ and $\rho_{cc}(\cdot,\cdot)$ to points. Corresponding to point p on P, there is a point p' on F_2 such that $\rho_c(p,v_1) - \rho_{cc}(p,v_1)$ in G_1 equals $\rho_c(p',v_1) - \rho_{cc}(p',v_1)$ in G. In particular, if p is at distance l from one of the endpoints, say x, of P, then p' is at distance $l' = l + (\| F_2 \| - \| P \|)/2$ from x along F_2. Thus, for each edge of F_2, the points p on P that map to points p' on that edge are consecutive on P. Determine the range of points on P falling on each edge of F_2. Repeat this process for each interior edge f of F_2, determining the ranges of points p' on f falling on the various edges of F_2', where F_2' is the remainder of the other face boundary to which f belongs. From these ranges, infer the ranges of points p on P falling on the edges of F_2'.

Proceeding in this fashion yields the range of points p on P that fall on each exterior edge of G contained in G_2. If p is the i^{th} division point for interior edge $e' = \{v_1, v_2\}$, and falls on an exterior edge $\{u,w\}$, then encode the 3-tuple (i,u,w) at v_1.

If P is e, then compute the division points for e as follows. In $G - e$ proceed around F counterclockwise from one of the endpoints of e, say x, and locate α points on F at intervals of $\| F \| / \alpha$. For each point determine the edge on the exterior face on which it falls, using the information now available about the ranges of points on each edge of F that fall on the various exterior edges of G. This yields the division points for e. Encode each division point as a 3-tuple.

Theorem 3. Let G be a biconnected n-node outerplanar graph satisfying the generalized triangle inequality. The above algorithm correctly computes division points for the interior edges of G. Furthermore, the running time of the algorithm is $O(\alpha n \log n)$, where α is the number of division points computed for each interior edge.

Proof (sketch). Correctness can be shown by induction on n, noting that the construction of G_1 and G_2 preserves distances and that the mapping of a point falling in the interior of P to the exterior face leaves the distance difference invariant. The running time analysis is straightforward. ∎

3.2.3 Computing division points distributively

The ideas of Section 3.2.2 yield a distributed algorithm to compute the division points for any given interior

edge using $O(\alpha n)$ messages. When an interior edge fails, one of its endpoints initiates the algorithm, determines the division points, and then propagates these to all nodes. The communication overhead in this approach compares favorably with the overhead in the approach where division points are precomputed, since in the latter case, $O(\alpha n)$ messages are needed simply to propagate division points to all the nodes. Furthermore, as we will see in the next section, with the distributed approach it is possible to efficiently handle the failure of more than one interior edge.

Assume that each each node has available the cost of each incident edge and the cost of each face containing the node. Let $e = \{v_1, u_1\}$ be the interior edge that fails, and let F be the boundary of the face resulting from the deletion of e. First, the nodes on F distributively locate α points $v_1 = p_0, p_1, \ldots, p_{\alpha-1}$ at intervals of $\| F \| / \alpha$ on F. Next, the points falling on each edge of F are distributively mapped to the portion of the exterior face of $G' = G - e$ contained in the corresponding F-component.

Let $v_1, v_2, \ldots, v_q$ be the nodes of F encountered going counterclockwise around F from v_1. Let e_i denote edge $\{v_i, v_{i(\mathrm{mod}\ q)+1}\}$, and let L_i denote the cost of the segment of F from v_1 counterclockwise to v_i, $1 \leq i \leq q$. Beginning with v_1, each node v_i in turn determines the range of the points $p_0, p_1, \ldots, p_{\alpha-1}$ falling on e_i as follows. It determines the largest integer j_i such that $(\| F \| / \alpha) j_i \leq L_i + \| e_i \|$. Thus, points $p_{j_{i-1}+1}, p_{j_{i-1}+2}, \ldots, p_{j_i}$ fall on e_i, where we take j_0 to be -1. If $j_i < \alpha - 1$, then v_i sends a message to v_{i+1}, with j_i, α, $\| F \|$, and $L_{i+1} = L_i + \| e_i \|$. Upon receipt of this message, v_{i+1} proceeds to determine the range of points falling on e_{i+1}.

Next, the points falling in the interior of each interior edge e_i are mapped to the exterior edges of G' in the corresponding F-component. These are the desired division points. We describe this process for e_1, assuming that it is an interior edge. The mapping is done in a succession of stages. In the first stage, the points in the interior of e_1 are mapped to the various edges of F_1, where F_1 is the remainder of the other face boundary to which e_1 belongs. The stage is initiated by v_1, which determines the farthest point on e_1 such that the distance of this point from v_1 plus $(\| F_1 \| - \| e_1 \|)/2$ is at most $\| f_1 \|$, where f_1 is the edge from F_1 incident with v_1. This gives the range of points from e_1 falling on f_1. If all points on e_1 do not fall on f_1, then v_1 sends a message over f_1 to its neighbor, which proceeds to determine the range of points from e_1 falling on f_2, where f_2 is the other edge of F_1 incident with it. In this manner the ranges of points on e_1 falling on the various edges of F_1 are determined. In the next stage this is repeated for each interior edge of F_1. Eventually, all points on e_1 get mapped to exterior edges. Similarly for points on $e_2, e_3, \ldots, e_q$.

Each division point is encoded in its 3-tuple form by one of the endpoints of the exterior edge it falls on. All the division points are then routed to v_1 (for instance, along the boundary of the exterior face of G').

The total number of messages used to do the mapping is $O(n)$, since at most one message is sent over any edge. The number of messages used to route the division points to v_1 is $O(\alpha n)$. If the edge costs are integers and are polynomial in n, then the length of any message is $O(\log n)$ bits. Similarly, the total additional storage needed to store face and edge costs is $O(n)$ items, where each item is $O(\log n)$ bits long.

3.2.4 Handling multiple interior edge faults

We now consider routing efficiently in an outerplanar network with t failed interior edges, $t > 1$. One problem in trying to generalize the approach of Section 3.2.1, by precomputing division points for each edge fault, is that the computation for any edge does not take into account changes in distances caused by the failure of the other edges. Furthermore, precomputing division points for every combination of t failed interior edges is expensive, both spacewise and timewise. However, it is possible to generate efficient routings if division points are computed distributively, as and when edges fail. We now present an approach that stores a total of $O(t\alpha n)$ items of additional routing information in the network and routes with a worst case bound of $((\alpha+1)/(\alpha-1))^t$, which is less than $(\alpha+t)/(\alpha-t)$, for $\alpha \geq t > 1$. The computation of the additional information uses a total of $O(t\alpha n)$ messages.

Let $e_i = \{v_i, u_i\}$, $1 \leq i \leq t$, be the failed interior edges. Let $G' = G - \{e_1, e_2, \ldots, e_t\}$. Let F_i be the face boundary that results when e_i is deleted from $G' \cup e_i$. Then graph G' is the union of a number of F_i-components. Each node v_i initiates the distributed algorithm of Section 3.2.3 on G' and computes α division points for e_i. These are then propagated to each node in the network. Each node also receives the name v_i and the names of the gateway nodes of the F_i-component containing the node. The total number of messages used to compute and propagate the information is $O(\alpha n)$ per failed edge, hence $O(t\alpha n)$ in total. The total space needed to hold this information is $O(t\alpha n)$.

The routing from source s to destination d is as follows. If s and d are in the same F_i-component for all i, $1 \leq i \leq t$, then the routing to d is performed using the interval routing information for G. Otherwise, without loss of generality, let s and d be in different F_i-components for $i = 1, 2, \ldots, l$, where $1 \leq l \leq t$. Let g_i be the edge from F_i such that s is in the F_i-component defined by g_i, $1 \leq i \leq l$. Then one of the endpoints of g_i will be in the interval (s, d) and the other in the interval (d, s). Without loss of generality assume that when going clockwise around the exterior face from s to d, the endpoints of $g_1, g_2, \ldots, g_l$ contained in interval (s, d) are encountered in order. Thus the g_i successively 'separate' s from d. Using the division points for e_1, s performs either a shortest clockwise or a shortest counterclockwise routing, as described in Section 3.2.1. Let a_i be the endpoint of h_i reached first, where h_i is the edge of F_i whose F_i-component contains d, $1 \leq i \leq l$. For $1 \leq i \leq l-1$, each a_i uses the division points for e_{i+1} to perform the routing to d. Once the message reaches a_l, interval routing information for G is used to route to d.

Theorem 4. Let $\rho(s, d)$ be the distance in $G' = G - \{e_1, e_2, \ldots, e_t\}$, between any nodes s and d, and let $\hat{\rho}(s, d)$

be the length of the routing generated by the above strategy. Then $\hat{\rho}(s,d)/\rho(s,d) < ((\alpha+1)/(\alpha-1))^t$, which is less than $(\alpha+t)/(\alpha-t)$, for $\alpha \geq t > 1$.

Proof (sketch). If s and d are in the same F_i-component for all i, $1 \leq i \leq t$, then they are both in the subgraph I of G' that is the intersection of the F_i-components. It can be shown that there is a shortest (s,d)-path from G in I. Thus the interval routing will be along this path.

If s and d are in different F_1-, F_2-, ..., F_l-components, $1 \leq l \leq t$, then we show by induction on l that $\hat{\rho}(s,d)/\rho(s,d) < ((\alpha+1)/(\alpha-1))^l$. The basis, $l = 1$, follows from Theorem 2.

For $l > 1$ we have $\hat{\rho}(s,d) = \hat{\rho}(s,a_1) + \hat{\rho}(a_1,d)$. Now, $\hat{\rho}(a_1,d)$ is the length of the routing from a_1 to d if G contains only the $l-1$ interior edge faults $e_2, e_3, \cdots, e_l$. Thus, by the induction hypothesis, $\hat{\rho}(a_1,d) < ((\alpha+1)/(\alpha-1))^{l-1}\rho(a_1,d)$. Also, $\hat{\rho}(s,a_1) + \rho(a_1,d)$ is the length of the routing from s to d if G contains only the fault e_1. Thus, by the induction hypothesis, $\hat{\rho}(s,a_1) + \rho(a_1,d) < ((\alpha+1)/(\alpha-1))\rho(s,d)$. Substituting these inequalities in $\hat{\rho}(s,d) = \hat{\rho}(s,a_1) + \hat{\rho}(a_1,d)$ yields $\hat{\rho}(s,d) < ((\alpha+1)/(\alpha-1))^l\rho(s,d) \leq ((\alpha+1)/(\alpha-1))^t\rho(s,d)$. An induction on t shows that $((\alpha+1)/(\alpha-1))^t < (\alpha+t)/(\alpha-t)$, for $\alpha \geq t > 1$. ∎

3.2.5 Handling both node and edge faults

If all t faults are nodes and exterior edges, then additional routing information is set up to handle these as described previously. The routings generated will be along shortest paths. However, suppose that t' of the t faults are interior edges. The $t - t'$ node and exterior edge faults cause the resulting network G' to be the union of S-components. Each interior edge fault will be confined to one of the S-components. Additional routing information is computed first for the node and exterior edge faults, and then for the interior edge faults, as before. The total number of items stored and the number of messages exchanged will both be $O((t-t')n + t'\alpha n)$. The routing from s to d is done exactly as it is for node and exterior edge failures, except that the portion of the routing which is within any S-component of G' that contains one or more failed interior edges is done using the division points.

In worst-case, all t' interior edge faults are in a single S-component and all, or nearly all, of the length of the shortest (s,d)-path is within this S-component. The routing realized within this S-component is at most $((\alpha+1)/(\alpha-1))^{t'}$ times longer than a shortest routing within the component. Thus the length of the (s,d)-routing is at most $((\alpha+1)/(\alpha-1))^{t'}$ times longer than a shortest (s,d)-path.

References

[1] G.N. Frederickson, "Implicit data structures for the dictionary problem", *Journal of the ACM*, Vol. 30, No. 1, January 1983, pp. 80–94.

[2] G.N. Frederickson, "A new approach to all pairs shortest paths in planar graphs", *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, New York City, May 1987, pp. 19–28.

[3] G.N. Frederickson and R. Janardan, "Designing networks with compact routing tables", *Algorithmica* – Special Issue on Parallel and Distributed Computing, Vol. 3, No. 1, 1988, pp. 171–190.

[4] G.N. Frederickson and R. Janardan, "Separator-based strategies for efficient message routing", *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, Toronto, October 1986, pp. 428–437.

[5] G.N. Frederickson and R. Janardan, "Space-efficient message routing in c-decomposable networks", submitted for journal publication. (Available as CSD-TR-615 (revised), Purdue University, 1986.)

[6] J.I. Munro and H. Suwanda, "Implicit data structures for fast search and update", *Journal of Computer and System Sciences*, Vol. 21, No. 2, October 1980, pp. 236–250.

[7] F. Harary, *Graph theory*, Addison-Wesley, Reading MA, 1969.

[8] N. Santoro and R. Khatib, "Labelling and implicit routing in networks", *The Computer Journal*, Vol. 28, No. 1, February 1985, pp. 5–8.

[9] J. van Leeuwen and R.B. Tan, "Computer networks with compact routing tables", in *The Book of L*, G. Rozenberg and A. Salomaa (eds.), Springer-Verlag, 1986, pp. 259–273.

[10] J. van Leeuwen and R.B. Tan, "Interval routing", *The Computer Journal*, Vol. 30, No. 4, August 1987, pp. 298–307.

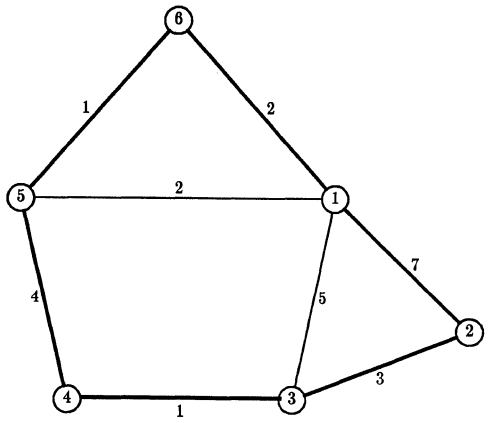


Figure 1. A clockwise naming of the nodes of a weighted outerplanar graph, with the boundary of the exterior face shown bold.

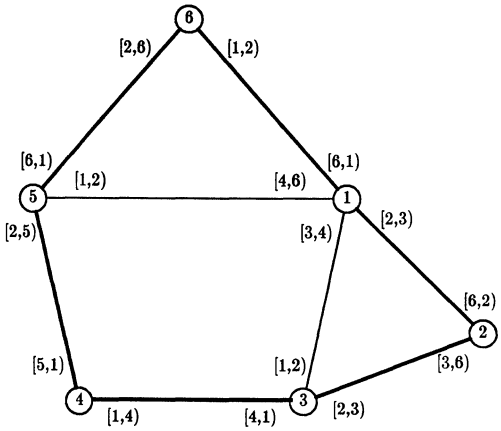


Figure 2. A labeling of the edges of the outerplanar graph of Figure 1 with intervals encoding shortest paths.

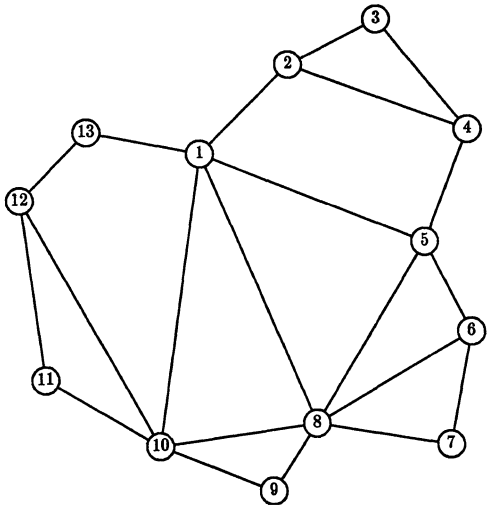


Figure 3. Outerplanar network G used to illustrate fault-tolerant routings.

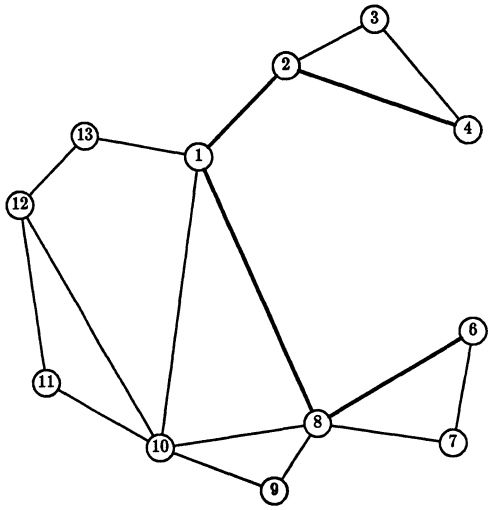


Figure 4. Network resulting from G after the failure of node 5.

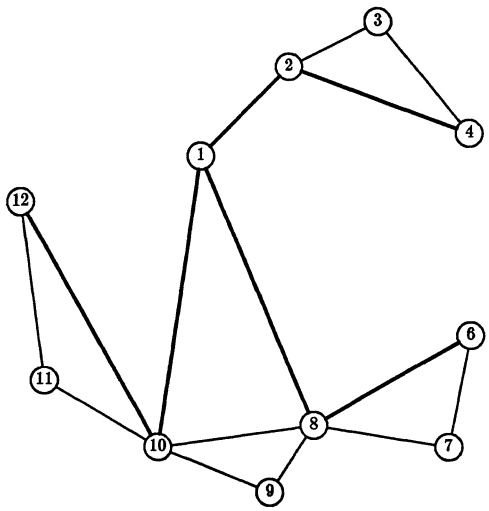


Figure 5. Network resulting from G after the failure of nodes 5 and 13.

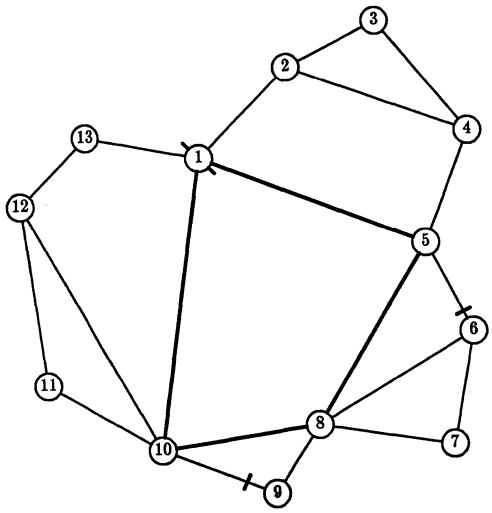


Figure 6. Network resulting from G after the failure of interior edge $\{1,8\}$.

A DECOMPOSITION APPROACH FOR BALANCING LARGE-SCALE ACYCLIC DATA FLOW GRAPHS

P. R. Chang and C. S. G. Lee

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Abstract

This paper presents an efficient decomposition technique which provides a more systematic approach in solving the optimal buffer assignment problem of an acyclic data flow graph (ADFG) with a large number of computational nodes. The buffer assignment problem is formulated as an integer linear optimization problem which can be solved in pseudo-polynomial time. However, if the size of an ADFG increases, then integer linear constraint equations may grow exponentially, making the optimization problem more intractable. The decomposition approach utilizes the critical path concept to decompose a directed ADFG into a set of connected subgraphs, and the integer linear optimization technique can be used to solve the buffer assignment problem in each subgraph. Thus, a large-scale integer linear optimization problem is divided into a number of smaller-scale subproblems, each of which can be easily solved in pseudo-polynomial time. Examples are given to illustrate the proposed decomposition technique.

I. Introduction

In designing VLSI systolic architectures for many complex computational tasks in pattern recognition and signal processing [6], and robotics [7],[8], the functional decomposition of the task into a set of computational modules can be represented as a directed task graph, and the inclusion of input data modifies the task graph to an acyclic data flow graph (ADFG). The nodes of an ADFG correspond to the computational modules, each of which can be realized by a linear pipelined functional unit for increasing the system throughput [5]. The operands or data move along the edges, each of which connects a pair of nodes. Due to a different computational time of the modules, data flow (both inputs and results from one module to another) in an ADFG may occur at different speeds in different directions. Thus, operands may arrive at multi-input modules at different arrival times, causing an unnecessary longer pipelined time in the ADFG. A conventional approach is to insert delay buffers (FIFO queues) at various paths to buffer the inputs or the output results from one module to another to achieve a balanced (or synchronous) ADFG. The problem of balancing a directed ADFG by inserting appropriate delay buffers along appropriate paths to achieve maximum pipelining has been solved previously by the cut-set theorem [5],[6], the local correctness criterion [6], and the graph-theoretic approach [2]. Furthermore, Hwang and Xu [4] showed that the balanced ADFG can be realized in a two-level pipeline network which is reconfigurable and provides the flexibility in various vector processing applications. The delay matching may be handled by programmable buffers so that proper non-compute delays can be inserted in each data flow path. An example is the design of the LINC chip [3] which is an 8-by-8 crossbar up to 32 units of programmable delays in each data flow path.

This paper presents an efficient decomposition technique which provides a more systematic approach in solving the optimal buffer assignment problem of an ADFG with a large number of computational nodes. Since it is of vital importance to minimize the number of buffers used in a systolic system to minimize the design cost, the optimal buffer assignment problem is formulated as an integer linear optimization problem, which can be easily solved in computers in pseudo-polynomial time [9]. However, if the number of computational nodes in an ADFG is quite large, then integer linear constraint equations may grow exponentially, making the optimization problem more difficult than it should be. The construction of integer linear constraint equations in a large-scale ADFG reveals the existence of many redundant integer linear constraint equations which come from the path overlapping between two paths of two different multi-input nodes. They can be easily removed by recognizing the overlapping path (or common path) traversed by different paths. In an effort to reduce the difficulty of optimizing a large number of integer linear constraint equations, an efficient and systematic decomposition technique is proposed to recognize all the decomposable subgraphs in an ADFG and generate their associated sets of integer linear constraint equations. The decomposition approach utilizes the critical path concept to decompose a directed ADFG into a set of connected subgraphs, and the integer linear optimization technique can be used to solve the buffer assignment problem in each subgraph. Thus, a large-scale integer linear optimization problem is divided into a number of smaller-scale subproblems, each of which can be easily solved in pseudo-polynomial time. Examples are given to illustrate the decomposition approach and finally, the proposed decomposition technique is used to balance an interconnection of CORDIC (COordinate Rotation DIgital Computer [11]) processors to achieve maximum pipelining for computing the robot inverse kinematic position solution [8].

This work was supported in part by the National Science Foundation under Grant CDR-8500022.

II. Formulation For Balancing Acyclic Data Flow Graphs

In formulating the optimal buffer assignment problem, we shall assume that the number of computing stages of any computational module of an ADFG is finite and that the execution time of any stage is a constant, called a basic time unit or stage latency. An ADFG is maximum pipelined if the minimum number of time units needed for obtaining two successive outputs from the pipeline is equal to one basic time unit. We shall concentrate our interests on single-input single-output (SISO) ADFG's and introduce some necessary definitions for formulation.

Definition 1: A weighted ADFG $GW=(V,E,W)$ corresponding to an ADFG $G=(V,E)$ is a weighted directed graph where W is a weight function from E to a set of non-negative real numbers. $V=(v_1,v_2,\cdots,v_n)$ is a finite set of computational nodes (or modules), and $E=(e_1,e_2,\cdots,e_n)$ is a finite set of edges. An edge connecting node v_i to node v_j is denoted by $e(i,j)$.

A logical way to convert an ADFG to a corresponding weighted ADFG is to assign weights to each output edge of a computational node such that the weight assigned to each edge is equal to the number of the computing stages of the computational node. For example, the weight $w(e(i,j))$ assigned to the edge $e(i,j)$ is equal to the number of computing stages of node v_i.

Definition 2: The cost (or weight) of any k th path $\phi_k(v_p,v_q)$ from node v_p to node v_q can be defined as the sum of the weights of all edges along the path. That is,

$$w(\phi_k)=\sum_{e(i,j)\in\phi_k(v_p,v_q)} w(e(i,j)).$$

Thus, the cost of a path from node v_p to node v_q is equal to the number of computing stages needed for an operand to travel along the corresponding path from node v_p to node v_q.

Definition 3: A weighted ADFG GW with an input node u_0 is said to be balanced if the cost for any two different paths from the input node u_0 to an arbitrary multi-input node u_k is equal.

This definition indicates that a balanced ADFG achieves maximum pipelining. Unfortunately, most ADFG's derived from given tasks are usually unbalanced. To balance an ADFG, appropriate delay buffers must be inserted along appropriate paths from the input node u_0 to any particular multi-input node of interest. Thus, any different paths from the input node u_0 to a multi-input node will have equal costs. The appropriate buffering graph in which delay buffers are inserted to balance an unbalanced ADFG can be defined as:

Definition 4: A buffering graph $GB=(V,E,WB)$ corresponding to a weighted ADFG $GW=(V,E,W)$ is a weighted graph where the weight WB corresponds to the buffering introduced on E. Then, GB is called a buffering graph of GW. Furthermore, an ADFG $GW'=(V,E,W')$ can be composed from GW and GB such that $w'(e(i,j))=w(e(i,j))+wb(e(i,j))$, for all $e(i,j)\in E$, where

$wb(e(i,j))$ is the weight of the buffers from node v_i to node v_j. If GW' is a balanced ADFG, then GB is a balanced buffering graph for GW.

It can be shown that a buffering graph GB for a corresponding GW always exists, though it may not be unique. In order to minimize the cost for implementing an ADFG in a VLSI device, it is desirable to obtain a balanced buffering graph with a minimum number of delay buffers. Since the cost for any two different paths from the input node u_0 to an arbitrary multi-input node u_k must be equal for a balanced ADFG, delay buffers can be inserted to balance the cost for all paths from the input node u_0 to a multi-input node u_k. Assume $U=\{u_0,u_1,u_2,\cdots,u_n\}$ is a finite set of all multi-input nodes and the input and end nodes in GW, and there are m_k paths from the input node u_0 to a multi-input node u_k, that is, $\phi_l(u_0,u_k)$ [a], $1\le l\le m_k$ and $1\le k\le n$. The *critical path* $\phi_{l_k^*}(u_k)$ of a multi-input node u_k in GW is the path from the input node u_0 to the node $u_k, 1\le k\le n$, having the "heaviest" path weight defined as:

$$w^c(u_k)\triangleq w^c(\phi_{l_k^*}(u_k))\triangleq \max_{1\le l\le m_k}\sum_{e(i,j)\in\phi_l(u_k)} w(e(i,j)). \quad (1)$$

No other path from the input node u_0 to the node u_k can have a path weight greater than the critical path weight $w^c(u_k)$. Thus, the cost of the critical path from the input node u_0 to the end node u_n constitutes the initial delay time of the pipeline. In order to balance an ADFG, buffers $B(e(i,j))$ are introduced to insert into appropriate paths $\phi_l(u_k)$, from the input node u_0 to a multi-input node u_k, $1\le k\le n$, to achieve all paths entering the node u_k to have the same cost. That is,

$$\sum_{e(i,j)\in\phi_l(u_k)} w(e(i,j))+\sum_{B(e(i,j))\in\phi_l(u_k)}|B(e(i,j))| = w^c(u_k)+\sum_{B(e(i,j))\in\phi_{l_k^*}(u_k)}|B(e(i,j))| \quad (2)$$

where $|B(e(i,j))|$ is the weight or the number of computing stages in the buffer $B(e(i,j))$, $1\le l\le m_k$ and $1\le k\le n$. The first term in Eq. (2) is a constant and can be easily computed. The problem of finding all critical paths of u_k, $1\le k\le n$, is known to be solvable by applying Bellman's equation with time complexity of $O(|N|^2)$ [1], where N is the number of computational nodes in the GW.

Since it is desirable to minimize the initial delay time of the pipeline so that it is equal to $w^c(u_n)$, no delay buffers $B(e(i,j))$ should be assigned to the critical path $\phi_{l_n^*}(u_n)$ of the end node u_n.

Lemma 1. The critical path $\phi_{l_n^*}(u_n)$ of the end node u_n is independent of the buffer stage variables.

Taking this into consideration and rewriting Eq. (2), we have

$$\sum_{B(e(i,j))\in\phi_l(u_k)\backslash\phi_{l_n^*}(u_n)}|B(e(i,j))|-\sum_{B(e(i,j))\in\phi_{l_k^*}(u_k)\backslash\phi_{l_n^*}(u_n)}|B(e(i,j))| = [w^c(u_k)-\sum_{e(i,j)\in\phi_l(u_k)} w(e(i,j))]=b(l,k) \quad (3)$$

where $b(l,k)$ is a computed integer constant, $|B(e(i,j))|$ are undetermined buffer stages, $1\le l\le m_k, 1\le k\le n$, and the notation $\phi_l(u_k)\backslash\phi_{l_n^*}(u_n)$ denotes set subtraction and is defined as $\phi_l(u_k)\backslash\phi_{l_n^*}(u_n)=\phi_l(u_k)-(\phi_l(u_k)\cap\phi_{l_n^*}(u_n))$. Equation (3) is a set of linear simultaneous equations and can be expressed in a matrix-vector form as $\mathbf{Ax}=\mathbf{b}$, where $\mathbf{A}$ is a matrix introduced from the paths, $\mathbf{x}$ and $\mathbf{b}$ are unknown buffer stage vector and computed integer constant vector, respectively. The solution $\mathbf{x}$ is usually not unique, however, we can impose some restrictions on the problem to become an integer linear optimization problem. That is, we would like to minimize the total number of buffer stages in a balanced buffering graph GB:

$$\text{Min}\sum_{B(e(i,j))\in GB\backslash\phi_{l_n^*}(u_n)}|B(e(i,j))| \quad (4)$$

subject to the equality constraints of Eq. (3) and $|B(e(i,j))|\ge 0$, integer. The above integer linear programming problem can be solved in pseudo-polynomial time [9].

In the above buffer assignment problem, the number of buffer stages are obtained from the solution of the integer linear programming problem, and the buffers are placed on the edges in the buffering graph GB corresponding to the GW, except the critical path $\phi_{l_n^*}(u_n)$ of the end node u_n. In order to reduce the total number of buffer stage *variables* in the optimal buffer assignment problem, an equivalent transformation is performed on a balanced buffering graph GB to transform it to a *normalized* buffering graph GB' which is still a balanced buffering graph (since a balanced buffering graph is not unique) with respect to the weighted ADFG GW [12]. With the equivalent transformation on a balanced buffering graph GB, the optimal buffer assignment problem can be reformulated for the normalized balanced buffering graph GB' instead of the balanced buffering graph. This, in effect, greatly reduces the total number of buffer stage variables because these variables are attached to multi-output (or multi-input) nodes.

While constructing the integer linear programming formulation for the normalized balanced buffering graph for a weighted ADFG GW, it can be shown that many redundant integer linear constraint equations (in Eq. (3)) exist, making the optimization problem more difficult than it should be. The redundant integer linear constraint equations come from the path overlapping between two paths of two different multi-input nodes. A path decomposition technique is utilized to remove redundant integer linear constraint equations. Let $\phi_l(u_k)$ denote an l th path from the input node u_0 to a multi-input node u_k which passes through some other multi-input nodes. Among these multi-input nodes, a multi-input node u^* which is *nearest* to the node u_k is selected to decompose the path $\phi_l(u_k)$ into two sub-paths, that is, $\phi_l(u_k)=\phi_l(u^*)+\phi_l(u^*,u_k)$. Thus, the integer linear constraint equations of the path $\phi_l(u_k)$ with respect to the node u_k can be written as:

$$\sum_{e(i,j)\in\phi_l(u_k)} w(e(i,j))+\sum_{B(e(i,j))\in\phi_l(u_k)}|B(e(i,j))| = \sum_{e(i,j)\in\phi_l(u^*)} w(e(i,j))+\sum_{B(e(i,j))\in\phi_l(u^*)}|B(e(i,j)| + \sum_{e(i,j)\in\phi_l(u^*,u_k)} w(e(i,j))+\sum_{B(e(i,j))\in\phi_l(u^*,u_k)}|B(e(i,j))| \quad (5)$$

where $1\le l\le m_k$. Using Eq. (2) for the path $\phi_l(u^*)$ to the node u^*, the result of Lemma 1, and Eq. (3), Eq. (5) becomes

$$\sum_{B(e(i,j))\in\phi_l(u^*,u_k)\backslash\phi_{l_n^*}(u_n)}|B(e(i,j))|+\sum_{B(e(i,j))\in\phi_{l^*}(u^*)\backslash\phi_{l_n^*}(u_n)}|B(e(i,j))|-\sum_{B(e(i,j))\in\phi_{l_k^*}(u_k)\backslash\phi_{l_n^*}(u_n)}|B(e(i,j))| = [w^c(u_k)-w^c(u^*)-\sum_{e(i,j)\in\phi_l(u^*,u_k)} w(e(i,j))]. \quad (6)$$

With the above procedure for reducing redundant equations, the integer linear constraint equations for the normalized balanced buffering graph with respect to a weighted ADFG GW can be constructed according to the Procedure ILEG (Integer Linear Equation Generator) listed below.

Procedure ILEG (GW, $ILCE(GB')$). This procedure generates a set of integer linear constraint equations $ILCE(GB')$ for a normalized balanced buffering graph GB' with respect to a given weighted ADFG GW with labeled nodes.

I1. [Determine all critical paths.] Find all the critical paths $\phi_{l_k^*}(u_k)$ and the cost of each critical path $w^c(u_k)$ with respect to a multi-input node $u_k, 1\le k\le n$, by applying the Bellman's equation [1].

I2. [Assign buffer stage variables.] Assign buffer stage variables to the output edges which are attached to multi-output nodes, except for the output edges belonging to the critical path of the end node u_n.

I3. [Generate integer linear constraint equations.] For any path $\phi_l(u_k)$ with respect to a multi-input node $u_k, 1\le l\le m_k, 1\le k\le n$, if $\phi_l(u_k)$ does not pass through any other multi-input nodes, then use Eq. (3) to generate integer linear constraint equations. Otherwise, use Eq. (6) to generate integer linear constraint equations where $u^*\in\phi_l(u_k)$ is a multi-input node nearest to the node u_k selected for path decomposition. Note that the paths and their costs between two multi-input nodes may be found with time complexity $O(n^3)$ by using the path-finding algorithm [1].

I4. [Output integer linear constraint equations.] Output the integer linear constraint equations from Eq. (3) or Eq. (6) and return.

END ILEG

[a] We use the notation $\phi_l(u_i,u_k)$ to indicate an lth path from node u_i to node u_k. If node u_i is the input node u_0, then $\phi_l(u_0,u_k)\equiv\phi_l(u_k)$.

Let us illustrate the above Procedure ILEG by an example. Fig. 1(a) shows a weighted ADFG GW. We would like to obtain an optimal normalized balanced buffering graph GB corresponding to the GW.

Step 1. Nodes G, J, K, and M are multi-input nodes. Then the critical path for

(a) Node G: $\phi_{l_G^*}(G) =$ Path $A-C-G$, $w^c(G) = 25$.

(b) Node J: $\phi_{l_J^*}(J) =$ Path $A-C-G-J$, $w^c(J) = 31$.

(c) Node K: $\phi_{l_k^*}(K) =$ Path $A-C-G-K$, $w^c(K) = 31$.

(d) Node M: $\phi_{l_M^*}(M) =$ Path $A-C-G-J-M$, $w^c(M) = 41$.

Step 2. Obtain the normalized buffering graph as shown in Fig. 1(b).

Step 3. The integer linear constraint equations are generated according to Eq. (3) or Eq. (6):

(a) For node G:

(i) Path $A-B-E-G$: $|B_1| + |B_4| = (25-5-6-2) = 12$.

(ii) Path $A-B-G$: $|B_1| + |B_5| = (25-5-6) = 14$.

(b) For node J:

(i) Path $A-B-F-J$: $|B_1| + |B_3| = (31-5-6-12) = 8$.

(c) For node K:

(i) Path $A-D-H-K$: $|B_2| + |B_6| - |B_8| = 13$.

(ii) Path $A-D-I-K$: $|B_2| + |B_7| - |B_8| = 9$.

The above integer linear constraint equations have been generated according to Eq. (3). The following case will show the integer linear constraint equations generated by using Eq. (6).

(d) For node M, we select $u^* \equiv K \in \phi_l(M)$ as the multi-input node nearest to the node M. The path $\phi_l(M)$ can be decomposed into two sub-paths, that is, $\phi_l(M) = \phi_l(u^*) + \phi_l(u^*, M)$. According to Eq. (6), we have $|B_8| + |B_9| = (41-31-8) = 2$.

Step 4. Minimize $\sum_{i=1}^{9} |B_i|$ subject to the constraints of the integer linear equations generated in Step 3. The optimization gives $|B_1| = 8$, $|B_2| = 9$, $|B_3| = 0$, $|B_4| = 4$, $|B_5| = 6$, $|B_6| = 4$, $|B_7| = 0$, $|B_8| = 0$, $|B_9| = 2$, and the total number of buffer stages is 33.

III. Formulation for Decomposition Approach

The previous section indicates that if the task graph is simple, then the buffer assignment problem can be easily solved as illustrated in the above example. However, if the number of computational nodes in an ADFG is quite large, then integer linear constraint equations may grow tremendously, making the optimization problem more intractable. Thus, a systematic approach in reducing the computational difficulty in a large-scale integer linear optimization for the buffer assignment problem must be devised. A decomposition approach, which utilizes the critical path concept to decompose the task graph into a set of connected subgraphs from which the integer linear optimization technique can be used to solve the buffer assignment problem in each subgraph, will be addressed in this section.

Lemma 2. If a multi-input node $u_k \in \phi_{l_n^*}(u_n)$ and its critical path is $\phi_{l_k^*}(u_k)$, then $\phi_{l_k^*}(u_k) \subseteq \phi_{l_n^*}(u_n)$. (Lemma 2 can be proved by contradiction [12].)

Definition 5: Let $\overline{GW} = (\overline{V}, \overline{E}, \overline{W})$ be an undirected graph with $N = |V|$ and $M = |E|$. A connected component π_m of $\overline{GW}$ is a maximal connected subgraph, which is a connected subgraph that is not contained in any larger connected subgraphs.

Definition 6: A directed block $\overline{\pi}_m$ of a directed graph GW^* is a directed subgraph, and its corresponding undirected subgraph π_m (i.e. $\pi_m = Undirect^{(b)}(\overline{\pi}_m)$) is a connected component of the corresponding undirected graph $\overline{GW}$ ($\overline{GW} = Undirect(GW^*)$).

The problem of finding all the connected components of an undirected graph $\overline{GW}$ may be solved with the time complexity of $O(N+M)$ by using the depth-first search algorithm SEARCH $(\overline{GW}, \pi_m)$ in [1], where $\overline{GW}$ is an input undirected graph and π_m, $1 \le m \le m_{cc}$, are output connected components, where m_{cc} is the number of the directed blocks in the corresponding directed graph GW^* of GW. The problem of finding the directed blocks $\overline{\pi}_m$ of a given directed graph GW^* may be solved by a modified depth-first search algorithm which is described in the Procedure DBS1 (Directed Blocks Searcher1) listed below.

(b) The notation $Undirect(\overline{\pi}_m)$ means taking the directed arrow of $\overline{\pi}_m$ out.

Procedure DBS1 $(GW^*, \overline{\pi}_m)$. This procedure finds all the directed blocks of a given directed graph GW^*.

D1. [Obtain the undirected graph of GW^*.] Let $\overline{GW} = Undirect(GW^*)$. That is, remove the directed arrow of GW^*.

D2. [Determine undirected connected components of $\overline{GW}$.] Find all the undirected connected components, π_m, $1 \le m \le m_{cc}$, of $\overline{GW}$ by the depth-first search algorithm SEARCH $(\overline{GW}, \pi_m)$.

D3. [Determine directed blocks.] Obtain all the directed blocks $\overline{\pi}_m$, $1 \le m \le m_{cc}$, by assigning the directed arrow back to π_m, $1 \le m \le m_{cc}$, according to the input directed graph GW^*.

D4. [Output the directed blocks.] Output all the directed blocks $\overline{\pi}_m$, $1 \le m \le m_{cc}$.

END DBS1

The connected components π_m from the depth-first search algorithm SEARCH $(\overline{GW}, \pi_m)$ and the directed blocks $\overline{\pi}_m$, $1 \le m \le m_{cc}$, from Procedure DBS1 will be used in our decomposition approach in obtaining a set of connected subgraphs from which the integer linear optimization technique can be applied to each subgraph to solve the buffer assignment problem. Our decomposition approach utilizes the critical path of the end node u_n, i.e. $\phi_{l_n^*}(u_n)$ as a cut set to partition an ADFG GW into several subgraphs. The procedure of graph partition and the determination of decomposed subgraphs (or directed blocks) is called graph decomposition [10]. The idea of the graph decomposition approach is to first take the critical path of the given directed graph out. This creates several edge disjoint subgraphs with some of the edges not connecting a pair of nodes because the nodes in the critical path are removed. In order to remedy this, nodes that are in the critical path $\phi_{l_n^*}(u_n)$ and are attached to two or more edges (incoming or outgoing) are called the *decomposed* nodes and denoted by $\tilde{u}_k$ (as the kth decomposed node); each of these decomposed nodes $\tilde{u}_k$ will be "split" into several independent pseudo-nodes $\tilde{u}_k^i$, $1 \le i \le d_k$, which are labeled according to the attached edges from left to right, and the last pseudo-node $\tilde{u}_k^{d_k}$ is always assigned to the kth decomposed node in the critical path $\phi_{l_n^*}(u_n)$, where d_k is the number of independent pseudo-nodes for the kth decomposed node. Thus, a new directed graph GW^* containing split directed subgraphs of the ADFG GW can be obtained by removing the critical path $\phi_{l_n^*}(u_n)$ and "splitting" the decomposed nodes. That is, $GW^* = (GW \backslash \phi_{l_n^*}(u_n)) \bigcup \{$labeled pseudo-nodes $\tilde{u}_k^i, 1 \le i \le (d_k - 1), 1 \le k \le k_{DN}\}$, where k_{DN} is the number of the decomposed nodes in $\overline{GW}$. The determination of the directed blocks $\overline{\pi}_m$ of an ADFG GW when the critical path $\phi_{l_n^*}(u_n)$ is taken out is very similar to the Procedure DBS1 for finding the directed blocks $\overline{\pi}_m$ of GW^*. The directed blocks $\overline{\pi}_m$ and $\overline{\pi}_m$ are always equivalent except for the existence of the pseudo-nodes, $\tilde{u}_k^i, 1 \le i \le d_k$. The procedure for determining the directed blocks $\overline{\pi}_m$ of an ADFG GW when the critical path $\phi_{l_n^*}(u_n)$ of the end node u_n is taken out can be described in the following Procedure DBS2 (Directed Blocks Searcher2).

Procedure DBS2 $(GW, \overline{\pi}_m)$. This procedure finds all the directed blocks of GW when its critical path $\phi_{l_n^*}(u_n)$ is taken out.

S1. [Remove critical path in GW and label decomposed nodes.]

(i) Obtain all the subgraphs from the ADFG GW by removing the critical path $\phi_{l_n^*}(u_n)$ of the end node u_n and splitting the decomposed nodes $\tilde{u}_k$, $1 \le k \le k_{DN}$.

(ii) Label the independent pseudo-nodes of the decomposed node $\tilde{u}_k$, that is $\tilde{u}_k^i$, $1 \le i \le d_k$, and $\tilde{u}_k = \tilde{u}_k^1 \oplus \tilde{u}_k^2 \oplus \cdots \oplus \tilde{u}_k^{d_k}$, where $\oplus$ is the direct sum of the pseudo-nodes coming from the same decomposed node.

S2. [Construct GW^*.] Construct a new directed graph GW^* which are the split directed subgraphs with labeled pseudo-nodes in step S1.

$$GW^* = \{GW \backslash \phi_{l_n^*}(u_n)\} \bigcup \{ \bigcup_{k=1}^{k_{DN}} \bigcup_{i=1}^{(d_k-1)} \{\tilde{u}_k^i\} \}.$$

S3. [Find the directed blocks of GW^*.] Use DBS1 $(GW^*, \overline{\pi}_m')$ to find the directed blocks $\overline{\pi}_m$ of GW^*.

S4. [Identify and merge pseudo-nodes in each directed block.] Determine the labeled pseudo-nodes which come from the same decomposed node and are in the same directed block $\overline{\pi}_m$. These labeled pseudo-nodes will be merged into a big labeled pseudo-node by the direct sum operator $\oplus$.

S5. [Determine and output the directed blocks $\overline{\pi}_m$.] Obtain $\overline{\pi}_m$ from $\overline{\pi}_m$ by applying the pseudo-nodes merging procedure in step S4 and output $\overline{\pi}_m$, $1 \le m \le m_{cc}$.

END DBS2

Using the Procedure DBS2 $(GW, \overline{\pi}_m)$, we can obtain all the directed blocks of GW, $\overline{\pi}_m$, $1 \le m \le m_{cc}$. Furthermore, new subgraphs can be constructed from $\overline{\pi}_m$ and defined as $\overline{\pi}_m^+ = \overline{\pi}_m \bigcup_{DN} \phi_{l_n^*}(u_n)$, for $1 \le m \le m_{cc}$, where the operator $\bigcup_{DN}$ means performing the set union of $\overline{\pi}_m$ and $\phi_{l_n^*}(u_n)$ (except the pseudo-nodes) and the direct sum on the pseudo-nodes coming from the same decomposed nodes in $\overline{\pi}_m$ and $\phi_{l_n^*}(u_n)$, simultaneously. These new subgraphs are called pseudo-connected components of the ADFG GW and will be used to decompose the buffer assignment problem into several small subproblems.

Let $\vec{\pi B}_m^+$ be a normalized balanced buffering graph for $\overline{\pi}_m^+$ and $ILCE(\vec{\pi B}_m^+)$ be the associated set of integer linear constraint equations which is obtained from the Procedure ILEG. Since an ADFG GW may have a large number of nodes, determining the buffer stage variables in GB from its large number of integer linear constraint equations may not be desirable. Since $GB = \bigcup_{DN}^{m_{cc}} \vec{\pi B}_m^+$, we would like to use this fact to see whether solving the buffer stage variables in each $\vec{\pi B}_m^+$, $1 \le m \le m_{cc}$, separately and independently is equivalent to solving the buffer stage variables in GB. If this is true, then we have divided a large-scale integer linear optimization problem into m_{cc} smaller-scale subproblems, each of which can be easily solved.

Theorem 1. Let GB and $\vec{\pi B}_m^+$, $1 \le m \le m_{cc}$, be, respectively, the normalized balanced buffering graphs of GW and its pseudo-connected components $\overline{\pi}_m^+$, $1 \le m \le m_{cc}$. The buffer stage variables in GB can be determined from their associated sets of integer linear constraint equations, $ILCE(\vec{\pi B}_m^+)$, $1 \le m \le m_{cc}$, separately and independently. Furthermore, the buffer stage variables determined from the set of integer linear constraint equations, $ILCE(\vec{\pi B}_{m1}^+)$, have no relations to the buffer stage variables determined from the set of equations, $ILCE(\vec{\pi B}_{m2}^+)$, where $m1 \ne m2$.

Proof: In order to prove the above theorem, we follow the procedure for constructing the associated sets of integer linear constraint equations for GB and show how they can be replaced by $ILCE(\vec{\pi B}_m^+)$, $1 \le m \le m_{cc}$. For convenience, we assume there is a multi-input node u_k in both GB (or the corresponding GW) and $\vec{\pi B}_m^+$ (or the corresponding $\overline{\pi}_m^+$). $\vec{\pi B}_m^+$ is the mth pseudo-connected component of GB. Assume that the associated paths from the input node u_0 to the node u_k in GB (or GW) are $\phi_l(u_k)$, $1 \le l \le m_{k\pi}$. Two cases are possible: (1) some of these paths pass through $\vec{\pi B}_m^+$ *only*, and (2) some of them pass through some other pseudo-connected components of GB. In case (1), because the paths in GB are also the paths in $\vec{\pi B}_m^+$, we will obtain the same resulting associated sets of integer linear equations for the paths in GB and the paths in $\vec{\pi B}_m^+$. In case (2), the paths from the input node u_0 to the node u_k may pass through some other pseudo-connected components, but they must intersect the critical path $\phi_{l_n^*}(u_n)$ of the end node u_n at some nodes, and finally end at the node u_k in $\vec{\pi B}_m^+$. It has been shown previously that a multi-input node u^*, which is on the critical path $\phi_{l_n^*}(u_n)$ and nearest to the node u_k, can be selected to decompose the path into two subpaths, that is, $\phi_l(u_k) = \phi_l(u^*) + \phi_l(u^*, u_k)$, where $\phi_l(u^*)$ is the path from the input node u_0 to the node u^* and passes through some other pseudo-connected components, and the entire traversal of the path $\phi_l(u^*, u_k)$ is in the $\vec{\pi B}_m^+$. Thus, the associated integer linear equation for the path $\phi_l(u_k)$ in GB can be rewritten as in Eq. (5). Using Lemma 1 and Eq. (2), the first two terms on the right hand side of Eq. (5) can be written as

$$\sum_{e(i,j) \in \phi_l(u^*)} w(e(i,j)) + \sum_{B(e(i,j)) \in \phi_l(u^*)} |B(e(i,j))| \qquad (7)$$

$$= w^c(u^*) + \sum_{B(e(i,j)) \in \phi_{l^*}(u^*)} |B(e(i,j))|.$$

Using the result of Lemma 2, the critical path to the node u^*, $\phi_{l^*}(u^*)$, is the path from the input node u_0 to the node u^* along the critical path $\phi_{l_n^*}(u_n)$, that is, $\phi_{l^*}(u^*) = \phi_{l_n^*}(u_0, u^*)$, which is independent of the buffer stage variables. Then Eq. (7) becomes

$$\sum_{e(i,j) \in \phi_l(u^*)} w(e(i,j)) + \sum_{B(e(i,j)) \in \phi_l(u^*)} |B(e(i,j))|$$

$$= w^c(u^*) = \text{a constant.} \qquad (8)$$

Substituting Eq. (8) into Eq. (5), we have:

$$\sum_{e(i,j) \in \phi_l(u_k)} w(e(i,j)) + \sum_{B(e(i,j)) \in \phi_l(u_k)} |B(e(i,j))| = w^c(u^*) + \qquad (9)$$

$$\sum_{e(i,j) \in \phi_l(u^*, u_k)} w(e(i,j)) + \sum_{B(e(i,j)) \in \phi_l(u^*, u_k)} |B(e(i,j))|.$$

Equation (9) indicates two things: First, the associated set of integer linear equations with respect to the node $u_k \in \vec{\pi B}_m^+$ depends only on the buffer stage variables in $\vec{\pi B}_m^+$ and are independent of the buffer stage variables in the other pseudo-connected components because $\phi_l(u^*, u_k) \in \vec{\pi B}_m^+$. Second, Eq. (9) can be generated and replaced by a path in $\vec{\pi B}_m^+$, that is, the path travels from the input node u_0 to the node u^* along the critical path $\phi_{l_n^*}(u_n)$, then from node u^* to node u_k along the path $\phi_l(u^*, u_k)$ in $\vec{\pi B}_m^+$. So, for any multi-input nodes u_k belonging to GB and $\vec{\pi B}_m^+$, it has been shown that the associated set of integer linear equation system for node u_k in GB can be replaced by the associated set of integer linear equation system for node u_k in $\vec{\pi B}_m^+$. In other words, the associated set of integer linear equation system for GB, i.e. $ILCE(GB)$, can be replaced by the associated sets of integer linear equation systems for $\vec{\pi B}_m^+$, i.e. $ILCE(\vec{\pi B}_m^+)$, $1 \le m \le m_{cc}$. □

Using the results from Theorem 1 and based on the fact that $GB = \bigcup_{DN}^{m_{cc}} \vec{\pi B}_m^+$, $\sum_{B(e(i,j)) \in GB} |B(e(i,j))|$ becomes $\sum_{m=1}^{m_{cc}} \sum_{B(e(i,j)) \in \vec{\pi B}_m^+} |B(e(i,j))|$, and the integer linear optimization problem in Eqs. (4) and (3) can be rewritten as follows:

$$\text{Min} \sum_{m=1}^{m_{cc}} \sum_{B(e(i,j)) \in \vec{\pi B}_m^+} |B(e(i,j))| \qquad (10)$$

subject to the associated sets of integer linear equation systems $ILCE(\vec{\pi B}_m^+)$, $1 \le m \le m_{cc}$. Because the buffer stage variables in different pseudo-connected components of GB are independent, Eq. (10) can be decomposed into the following subproblems:

For each $m = 1, 2, \cdots, m_{cc}$:

$$\text{Min} \sum_{B(e(i,j)) \in \vec{\pi B}_m^+} |B(e(i,j))|$$

subject to the associated set of integer linear equation system $ILCE(\vec{\pi B}_m^+)$.

This graph decomposition approach provides us with a technique to divide a large-scale integer linear optimization problem into a number of m_{cc} smaller-scale subproblems, each of which can be easily solved in pseudo-polynomial time. Let us apply the above decomposition approach to solve the same buffer assignment problem in section II.

Step 1. (a) Decompose the ADFG GW in Fig. 1(a) into subgraphs by removing the critical path $\phi_{l_M^*}(M)$ of the end node M.

(b) Label the pseudo-nodes of the decomposed nodes A, G, J, M, that is, $\{A_1, A_2, A_3\}$, $\{G_1, G_2, G_3, G_4\}$, $\{J_1, J_2\}$, and $\{M_1, M_2\}$.

(c) Construct
$GW^* = (GW \backslash \phi_{l_M^*}(M)) \bigcup \{A_1, A_2, G_1, G_2, G_3, J_1, M_1\}$.
Note that pseudo-nodes A_3, G_4, J_2, and M_2 are attached to the critical path $\phi_{l_M^*}(M)$. $GW^*, \phi_{l_M^*}(M)$, and the labeled pseudo-nodes are shown in Fig. 2. (2(a), 2(b), and 2(c)).

Step 2. This step is the same as the Procedure DBS2 $(GW^*, \overline{\pi}_m)$.

(a) Use Procedure DBS1 $(GW^*, \overline{\pi}_m)$ to find the $\overline{\pi}_m$, $1 \le m \le 2$, in GW^*. These directed blocks are shown in Figs. 2(a) and 2(b).

(b) Merge the labeled pseudo-nodes that come from the same decomposed node and are in the same directed block $\overline{\pi}_m$ into a big labeled pseudo-node by the direct sum operator. For example, G_1 and G_2 are the labeled pseudo-nodes coming from the decomposed node G in $\overline{\pi}_1$, and will be merged into $G_{1,2} = G_1 \oplus G_2$.

(c) Obtain $\overline{\pi}_m$ from $\overline{\pi}_m$, $m = 1, 2$, by applying the pseudo-nodes merging procedure. $\overline{\pi}_1$ is shown in Fig. 2(d).

Step 3. Let $\overline{\pi}_m^+ = \overline{\pi}_m \bigcup_{DN} \phi_{l_M^*}(M)$, $1 \le m \le 2$, which are the pseudo-connected components of GW (Figs. 2(e) and 2(f)).

Step 4. The corresponding normalized balanced buffering graphs GB and $\vec{\pi B}_m^+$ can be easily obtained by the buffer assignment rules and have the same graph structure as GW and $\overline{\pi}_m^+$, respectively. The buffer stage variables $B_1^1, B_2^1, B_3^1, B_4^1$ in $\vec{\pi B}_1$ as shown in Fig. 2(g), and $B_1^2, B_2^2, B_3^2, B_4^2, B_5^2$ in $\vec{\pi B}_2^+$ as shown in Fig. 2(h) correspond to the buffer stage variables B_1, B_3, B_4, B_5 and B_2, B_6, B_7, B_8, B_9 in GB as shown in Fig. 1(b), respectively.

Step 5. Generate $ILCE\ (\pi\vec{B}_1^+)$ and $ILCE\ (\pi\vec{B}_2^+)$ as follows:

$ILCE\ (\pi\vec{B}_1^+)$: $|B_1^1|+|B_2^1| \equiv |B_1|+|B_3| = 8$

$|B_1^1|+|B_3^1| \equiv |B_1|+|B_4| = 12$

$|B_1^1|+|B_4^1| \equiv |B_1|+|B_5| = 14.$

$ILCE\ (\pi\vec{B}_2^+)$: $|B_1^2|+|B_2^2|-|B_4^2| \equiv |B_2|+|B_6|-|B_8| = 13$

$|B_1^2|+|B_3^2|-|B_4^2| \equiv |B_2|+|B_7|-|B_8| = 9$

$|B_4^2|+|B_5^2| \equiv |B_8|+|B_9| = 2\,.$

Step 6. The integer linear programming problem for GB can be solved by two separated subproblems:

(1) $\text{Min} \sum_{B_i \in \pi\vec{B}_1^+} |B_i| \equiv \text{Min}\,[\,|B_1|+|B_3|+|B_4|+|B_5|\,]$
subject to the $ILCE\ (\pi\vec{B}_1^+)$ (found in Step 5).

(2) $\text{Min} \sum_{B_i \in \pi\vec{B}_2^+} |B_i| \equiv \text{Min}\,[\,|B_2|+|B_6|+|B_7|+|B_8|+|B_9|\,]$
subject to the $ILCE\ (\pi\vec{B}_2^+)$ (found in Step 5).
The optimization of subproblem (1) yields $|B_1| = 8$, $|B_3| = 0$, $|B_4| = 4$, $|B_5| = 6$, and the optimization of subproblem (2) gives $|B_2| = 9$, $|B_6| = 4$, $|B_7| = 0$, $|B_8| = 0$, $|B_9| = 2$. The results and solution are the same as given in the example in section II, but the optimization is much faster and simplier.

The above graph decomposition approach is applied to solve the buffer assignment problem of a larger problem — balancing the CORDIC-based pipelined architecture to achieve maximum pipelining for computing the joint solution of a PUMA robot manipulator [8]. Using Procedure DBS2 $(GW, \vec{\pi}_m)$, where GW is the directed task graph, 16 directed blocks, $\vec{\pi}_m, 1 \le m \le 16$, in GW are obtained. From these directed blocks, we can obtain the 16 pseudo-connected components, $\vec{\pi}_m^+$, $1 \le m \le 16$. The corresponding normalized balanced buffering graph $\overline{GB}$ for GW and the 16 pseudo-connected components in $\overline{GB}$, $\pi\vec{B}_m^+$, $1 \le m \le 16$, can be created. The associated sets of integer linear equation systems for $\pi\vec{B}_m^+$, $1 \le m \le 16$, can be obtained from the Procedure ILEG. The optimization solution for all the integer linear optimization subproblems yields a total of 159 buffer stages which agree with the solution given in [8].

IV. Conclusion

An efficient graph decomposition technique which provides a systematic approach in solving the optimal buffer assignment problem of a large-scale ADFG has been presented and discussed. The optimal buffer assignment problem is formulated as an integer linear programming problem. The construction of integer linear constraint equations in a large-scale ADFG reveals the existence of many redundant integer linear constraint equations, making the optimization more intractable. The proposed graph decomposition approach utilizes the critical path concept to decompose an ADFG into a set of connected subgraphs from which the integer linear optimization technique can be used to solve the buffer assignment problem in each subgraph. Thus, a large-scale integer linear optimization problem is divided into a number of smaller-scale subproblems which can be easily solved in computers in pseudo-polynomial time. The proposed graph decomposition technique is illustrated by two examples, and its efficiency and advantages can be seen in the example for balancing a CORDIC pipelined architecture to achieve maximum pipelining for computing the robot inverse kinematic position solution.

V. References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974, pp. 195-199.

[2] J. B. Dennis and R. G. Gao, "Maximum Pipelining of Array Operations on Static Data Flow Machine," *Proc. of 1983 Int'l. Conf. on Parallel Processing*, pp. 331-334, Aug. 1983.

[3] F. H. Hsu, H. T. Kung, T. Nishizawa, and A Sussman, "LINC: The Link and Interconnection Chip," Department of Computer Science, Carnegie-Mellon University, 1984.

[4] K. Hwang and Z. Xu, "Multipipeline Networking for Fast Evaluation of Vector Compound Functions," *Proc. of 1986 Int'l Conf. on Parallel Processing*, pp. 495-502, August 1986.

[5] H. T. Kung and M. Lam, "Wafer-Scale Integration and Two-level Pipelined Implementation of Systolic Arrays," *J. of Parallel and Distributed Computing*, vol. 1, no. 1, Sept. 1984, pp. 32-63.

[6] S. Y. Kung, H. J. Whitehouse, and T. Kailath, (editors), *VLSI and Modern Signal Processing*, Prentice-Hall, 1985.

[7] C. S. G. Lee and P. R. Chang, "Efficient Parallel Algorithm for Inverse Dynamics Computation," *IEEE Trans. Syst. Man, Cybern.*, vol. SMC-16, no. 4, pp. 532-542, July/August 1986.

[8] C. S. G. Lee and P. R. Chang, "A Maximum Pipelined CORDIC Architecture for Robot Inverse Kinematic Position Computation," *IEEE J. of Robotics and Automation*, vol. RA-3, no. 5, pp. 445-458, October 1987.

[9] C. H. Papadimitrious, "On the Complexity of Integer Programming," *J. of ACM*, vol. 28, no. 4, pp. 765-768, October 1981.

[10] J. A. Starzyk and A. Konczykowska, "Flowgraph Analysis of Large Electronic Networks," *IEEE Trans. on Circuits and Systems*, vol. CAS-33, pp. 302-315, March, 1986.

[11] J. E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electronic Computers*, vol. EC-8, no. 3, pp. 330-334, Sept. 1959.

[12] P. R. Chang and C. S. G. Lee, "A Decomposition Approach for Balancing Large-Scale Acyclic Data Flow Graphs," Technical Report TR-EE 87-23 School of Electrical Engineering, Purdue University, June 1987.
Also accepted for publication in *IEEE Trans. on Computers.*

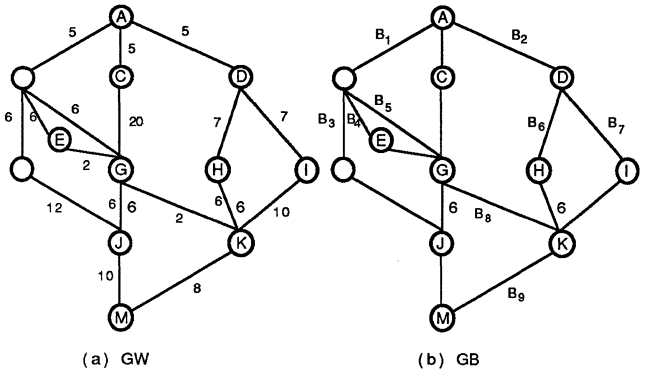


Figure 1. An Exampie for Buffer Assignment Problem

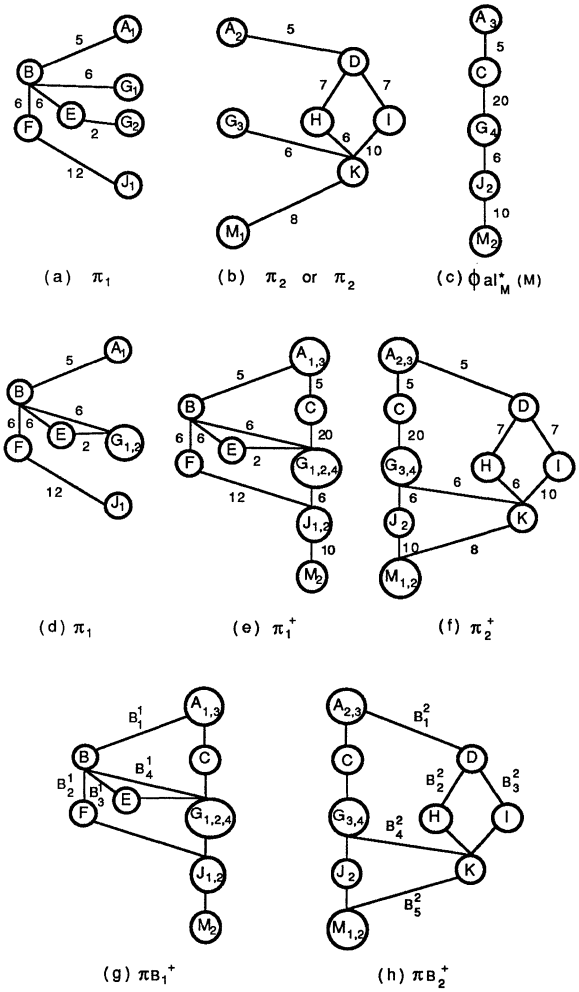


Figure 2. Graph Decomposition of the Example in Figure 1.

DILATION-2 EMBEDDINGS OF GRIDS INTO HYPERCUBES

Mee-Yee Chan
Computer Science Program
University of Texas at Dallas
Richardson, Texas 75083-0688

ABSTRACT

This paper addresses the following graph-embedding question: given a two-dimensional grid, and the smallest hypercube with at least as many nodes as grid points, how can we assign grid points to hypercube nodes (with at most one grid point per node) so as to keep grid-neighbors near each other as possible in the hypercube. We give a simple strategy which ensures that grid-neighbors are always mapped to hypercube nodes that are within a distance of two edges of each other.

1. INTRODUCTION

One of the key features of the hypercube is a rich interconnection structure which permits important network topologies, such as grids and trees, to be efficiently simulated. A **binary hypercube of dimension** n or **binary** n**-cube** can be thought of as an undirected graph of 2^n nodes labeled 0 to 2^n-1 in binary; two nodes are connected by an edge if and only if their labelings differ in exactly one bit position. To simulate a grid or a tree on the hypercube, nodes of the grid or tree must be mapped to hypercube nodes.

The question of interest here is: how can we map the nodes of any **two-dimensional grid** to the nodes of **its optimal hypercube** (the smallest hypercube with at least as many nodes as the grid), on a one-to-one basis, so that **dilation** (the worst case distance between grid-neighbors in the hypercube) is kept to a minimum.

A number of researchers have studied this problem [BMS, BS, CC, G, HJ, SS], with the following results. Over 61% of all two-dimensional grids can be embedded into their optimal hypercubes with a dilation of 1 (i.e. all grid-neighbors are also neighbors in the hypercube) by using binary-reflected Gray codes [SS]: Figure 1 shows how a 6x11 grid can be mapped into its optimal 7-cube. For the other over 38% of all two-dimensional grids, which have been proven to need at least dilation 2 [BS], we have the methods proposed in [BMS], [CC], [G] and [HJ]. [BMS], [HJ] and [CC] have shown that a substantial percentage of these grids (over 70% of the 38%) can be embedded with dilation 2, while [G] claims that all two-dimensional grids can be embedded with dilation 5.

The prevailing sentiment is that **all two-dimensional grids should be embeddable in their optimal hypercubes with at most dilation 2**, however, this has yet to be shown. This paper introduces a simple embedding strategy which does in fact confirm this conjecture.

2. THE EMBEDDING STRATEGY

Notation: Let N_k denote the sequence of k-bit binary-reflected Gray code, and let $N_k(p)$ denote the $(p+1)$st element in the sequence N_k. For example,

$N_1 \equiv (0,1)$,
$N_2 \equiv (00,01,11,10)$,
$N_3 \equiv (000,001,011,010,110,111,101,100)$
and $N_3(4) \equiv 110$.

Assume all logs are in base 2. Suppose we are given an x x y grid G.

CASE 1. $xy > 2^{\lfloor \log x \rfloor + \lfloor \log y \rfloor + 1}$ or $x = 2^{\lfloor \log x \rfloor}$ or $y = 2^{\lfloor \log y \rfloor}$

Then, embed G into its optimal hypercube using the binary-reflected Gray code strategy, and hence, with dilation 1.

CASE 2. otherwise

Assume, without loss of generality, $x \leq \frac{3}{2} 2^{\lfloor \log x \rfloor}$ (otherwise, we can rotate G by 90 degrees to assure this). Since $xy \leq 2^{\lfloor \log x \rfloor + \lfloor \log y \rfloor + 1}$, our objective is to label each node of the grid with a unique $(\lfloor \log x \rfloor + \lfloor \log y \rfloor + 1)$-bit binary number, which effectively names the node in the optimal $(\lfloor \log x \rfloor + \lfloor \log y \rfloor + 1)$-cube to which it is mapped. Since we have dilation 2 in mind, we allow the labels for grid-neighbors to differ in at most 2 bit positions.

Step 1. Determine the first $\lfloor \log x \rfloor$ bits of each node's label.

Create $2^{\lfloor \log x \rfloor}$ "chains", each of which is described by a y-vector of 1's and 2's. The vector for the first chain is

$$(a_{1,1}, a_{1,2}, \ldots, a_{1,y}) =$$

$$\left(\left\lceil \frac{x}{2^{\lfloor \log x \rfloor}} \right\rceil, \left\lceil \frac{2x}{2^{\lfloor \log x \rfloor}} \right\rceil - \left\lceil \frac{x}{2^{\lfloor \log x \rfloor}} \right\rceil, \ldots, \left\lceil \frac{yx}{2^{\lfloor \log x \rfloor}} \right\rceil - \left\lceil \frac{(y-1)x}{2^{\lfloor \log x \rfloor}} \right\rceil \right).$$

For the ith chain, $i = 2,3,\ldots,2^{\lfloor \log x \rfloor}$, the vector is

$$(a_{i,1}, a_{i,2}, \ldots, a_{i,y}) =$$

$$\left(\left\lfloor \frac{(i-1)x}{2^{\lfloor \log x \rfloor}} \right\rfloor - \left\lfloor \frac{(i-2)x}{2^{\lfloor \log x \rfloor}} \right\rfloor, a_{i-1,1}, a_{i-1,2}, \ldots, a_{i-1,y-1} \right)$$

The chains have the following properties:

$$\sum_{j=1}^{y} a_{i,j} \leq \left\lceil \frac{yx}{2^{\lfloor \log x \rfloor}} \right\rceil \leq 2^{\lfloor \log y \rfloor + 1} \quad \text{for } i=1,2,\ldots,2^{\lfloor \log x \rfloor}$$

$$\sum_{i=1}^{2^{\lfloor \log x \rfloor}} a_{i,j} = x \quad \text{for } j=1,2,\ldots,y$$

as well as

$$\sum_{j=1}^{k} a_{i,j} \in \left(\left\lfloor \frac{kx}{2^{\lfloor \log x \rfloor}} \right\rfloor, \left\lceil \frac{kx}{2^{\lfloor \log x \rfloor}} \right\rceil \right) \quad \text{for } i=1,2,\ldots,2^{\lfloor \log x \rfloor}$$

$$\sum_{i=1}^{k} a_{i,j} \in \left(\left\lfloor \frac{kx}{2^{\lfloor \log x \rfloor}} \right\rfloor, \left\lceil \frac{kx}{2^{\lfloor \log x \rfloor}} \right\rceil \right) \quad \text{for } j=1,2,\ldots,y$$

and since $x \leq \frac{3}{2} 2^{\lfloor \log x \rfloor}$, no consecutive 2's are possible within each chain vector.

The idea is that each chain vector represents a "chain": for example, (2,1,2,1,1,2,1,1,2,1,2) represents the chain depicted in Figure 2. Hence, for example, an 11 x 11 grid is associated with

(2,1,2,1,1,2,1,1,2,1,2)
(1,2,1,2,1,1,2,1,1,2,1)
(1,1,2,1,2,1,1,2,1,1,2)
(2,1,1,2,1,2,1,1,2,1,1)
(1,2,1,1,2,1,2,1,1,2,1)
(1,1,2,1,1,2,1,2,1,1,2)
(2,1,1,2,1,1,2,1,2,1,1)
(1,2,1,1,2,1,1,2,1,2,1),

and pictorially, we have Figure 3. Aligning the nodes of the above graph into 11 rows, or in general x rows, we get graph G_1 as shown in Figure 4. The $2^{\lfloor \log x \rfloor}$ chains will cover the x x y grid completely because of the properties stated above.

Each node of the x x y grid belonging to the ith chain is given $N_{\lfloor \log x \rfloor}(i-1)$ as the first $\lfloor \log x \rfloor$ bits of its $(\lfloor \log x \rfloor + \lfloor \log y \rfloor + 1)$-bit label. In this way, the first $\lfloor \log x \rfloor$ bits of adjacent nodes in the x x y grid differ in **at most one bit position**. Note that, since each chain has length $\leq 2^{\lfloor \log y \rfloor + 1}$ nodes, $\lfloor \log y \rfloor + 1$ bits are sufficient to distinguish the nodes of each chain.

Step 2. Determine the last $\lfloor \log y \rfloor + 1$ bits of each node's label.

Firstly, the jth node of the ith chain is marked with $(t_i + j) \bmod 2^{\lfloor \log y \rfloor + 1}$, where $t_1 = -1$ and $t_i = t_{i-1} - a_{i,1} + 1$. So, for the 11 x 11 grid, we have the situation shown in Figure 5. This marking, because of the nature of the G_1 graph, has the following property: the marking of adjacent nodes of the grid differ by at most 2 (in mod $2^{\lfloor \log y \rfloor + 1}$).

Figure 6 shows all possible marking scenarios for a node T and its grid-neighbor S. For any node T marked with t, the node S below T is marked $t-1$ if T and S are in the same chain, and t otherwise. For any node T marked with t, the node S left of T is marked $t-1$ or $t-2$.

Note: If we use $N_{\lfloor \log y \rfloor + 1}(t)$ as the last $\lfloor \log y \rfloor + 1$ bits of each node marked with t, then we effectively have an embedding of G into a $(\lfloor \log x \rfloor + \lfloor \log y \rfloor + 1)$-cube with **dilation 3**, i.e. the labels for grid-neighbors differ in at most 3 bit positions. Adjacent nodes in the grid will differ in at most 1 position of their first $\lfloor \log x \rfloor$ bits and at most 2 positions of their last $\lfloor \log y \rfloor + 1$ bits.

By changing each mark t into $\lfloor t/2 \rfloor$, we have a marking with the property that marks for adjacent nodes in the grid will differ by at most 1 (in mod $2^{\lfloor \log y \rfloor}$). So, for the 11 x 11 grid, we have Figure 7. The chains have been horizontally extended to have exactly $2^{\lfloor \log y \rfloor + 1}$ nodes each. Call such a marked graph G_2.

Our next objective is to color each node of the grid either red or black so that

(a) two nodes marked with the same number belonging to the same chain are colored differently, and

(b) two adjacent nodes marked with different numbers belonging to different chains are colored the same.

Condition (a) ensures that each node of the grid is indeed mapped to a unique node in the hypercube, and condition (b) ensures that dilation 2 is achieved for adjacent nodes of different chains.

Whether we can do this coloring hinges on whether the graph G_3, which has as its nodes the nodes of the extended graph G_2 but has as its edges the set {(S,T)| nodes S and T are marked the same and belong to the same chain in G_2} $\bigcup$ {(S,T)| there exists a node R such that nodes S and R are adjacent but belong to different chains and are marked so that S's mark is one less than R's, and nodes R and T are marked the same and belong to the same chain}, is bipartite or not. For the 11 x 11 grid, the graph G_3 would take the form shown in Figure 8. As it turns out, the G_3 graph for any grid, in general, can be shown to be acyclic, and hence, bipartite and colorable according to our objectives.

With such a coloring, we can do the following. A **red** node marked t is given $0N_{\lfloor \log y \rfloor}(t)$ as the last $\lfloor \log y \rfloor + 1$ bits of its label, while a **black** node marked t is is given $1N_{\lfloor \log y \rfloor}(t)$ as its last $\lfloor \log y \rfloor + 1$ bits. In this way, adjacent nodes of the **same** chain will differ in at most 2 bits position of their last $\lfloor \log y \rfloor + 1$ bits and share the same initial $\lfloor \log x \rfloor$ bits, making for a dilation of 2; adjacent nodes of different chains differ in at most 1 bit position of their last $\lfloor \log y \rfloor + 1$ bits and 1 bit position of their initial $\lfloor \log x \rfloor$ bits, again making for a dilation of 2. Hence, we finally do indeed have a dilation 2 embedding!

ACKNOWLEDGEMENTS

I would like to express my thanks to Hal Sudborough, Said Bettayeb, Joel Lee and Sheshu Madhavapeddy for their interest and help in looking over the first draft of this paper, and many inspiring discussions.

REFERENCES

[BMS] S. Bettayeb, Z. Miller and I.H. Sudborough, "Embedding Grids into Hypercubes", Computer Science Program, University of Texas at Dallas, August 1987.

[BS] J.E. Brandenburg and D.S. Scott, "Embeddings of Communication Trees and Grids into Hypercubes", Intel Scientific Computers Report #280182-001, 1985.

[CC] M.Y. Chan and F.Y.L. Chin, "On Embedding Rectangular Grids in Hypercubes", to appear in IEEE Trans. on Computers.

[G] D.S. Greenburg, "Optimal Expansion Embeddings of Meshes in Hypercubes", Technical Report YALEU/CSD/RR-535, Dept. of Computer Science, Yale University, August 1987.

[HJ] C.T. Ho and S.L. Johnsson, "On the Embedding of Arbitrary Meshes in Boolean Cubes with Expansion Two Dilation Two", Proceedings 1987 International Conference on Parallel Processing (August 1987) 188-191.

[SS] Y. Saad and M.H. Schultz, "Topological Properties of Hypercubes", Research Report 389, Dept. of Computer Science, Yale University, June 1985.

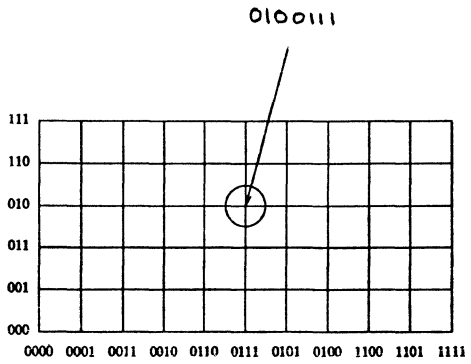


Figure 1. 6x11 grid

Figure 2. (2,1,2,1,1,2,1,1,2,1,2)

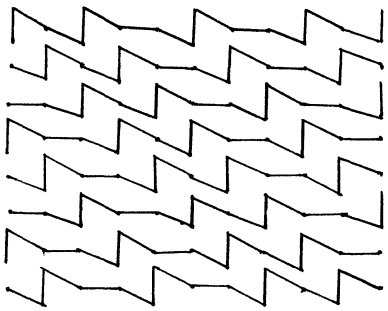


Figure 3. Chains for 11x11 grid

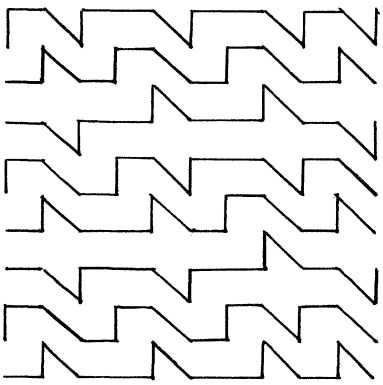


Figure 4. Chains redrawn : G_1

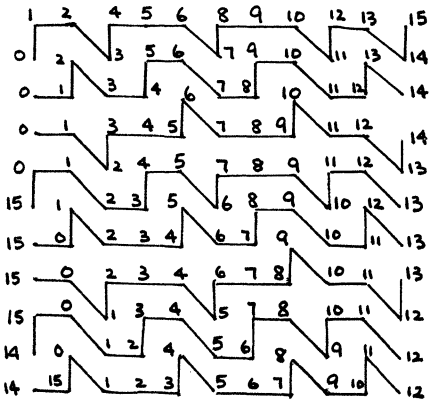


Figure 5. 1st marking

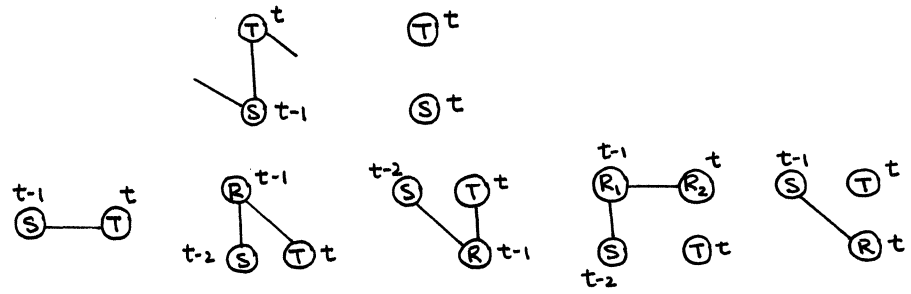


Figure 6. All possible scenarios

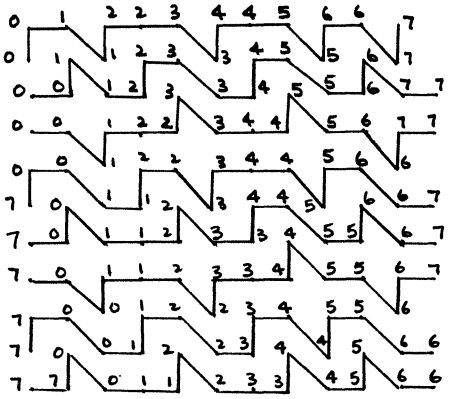


Figure 7. 2nd marking : G_2

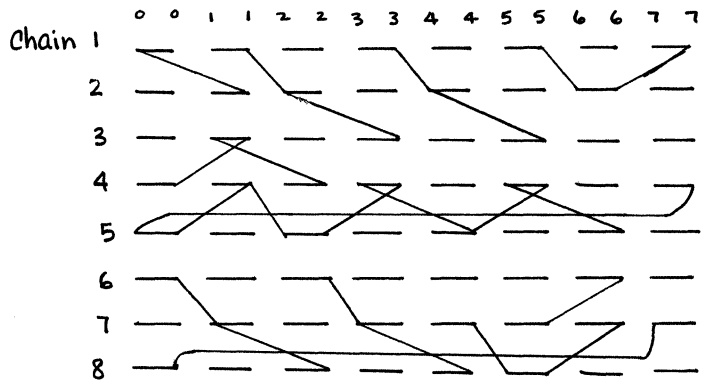


Figure 8. G_3 graph

Some results on graph coloring in parallel

S. Vishwanathan and M. A. Sridhar *
Department of Computer Science
University of South Carolina
Columbia, SC 29208

Abstract – The problem of constructing parallel graph-coloring algorithms is studied. It has been shown recently [13, 17, 21] that the problem of Brooks coloring of graphs is in $\mathcal{NC}$. In this paper, it is shown that the decision version of one of the sequential algorithms for coloring graphs, that typically uses fewer colors than the Brooks coloring algorithm, is logspace-complete for $\mathcal{P}$; therefore it is unlikely that this approach will yield an algorithm in $\mathcal{NC}$. An algorithm that colors some graphs with fewer colors than Brooks coloring is also shown.

1 Introduction

The *graph coloring problem* is the problem of assigning colors to the vertices of a graph in such a way that no two adjacent vertices receive the same color. Graph coloring has been studied extensively by researchers in the past. It is very easy to show that any graph G can be colored using no more than $\Delta(G)+1$ colors, where $\Delta(G)$ is the maximum degree of G. But it is also known that usually one requires fewer colors. A classic theorem on coloring is the theorem of Brooks [4], which shows that if G satisfies certain conditions, we can get away with using one less color:

Theorem 1 *A simple graph G of maximum degree Δ can be colored using at most Δ colors* iff *G is not an odd cycle and G does not contain a $K_{\Delta+1}$, i.e. a complete subgraph on $\Delta+1$ vertices.*

∎

Other bounds on the chromatic number are known [6], one of which we will use subsequently.

There has been considerable interest recently in constructing parallel algorithms for graph problems. Of particular interest are algorithms that use polynomially many processors and take polylog time. We call such parallel algorithms *good*. Problems having good parallel algorithms are said to be in the class $\mathcal{NC}$ [7]. It is obvious that $\mathcal{NC} \subseteq \mathcal{P}$, where $\mathcal{P}$ is the class of languages recognizable by a (sequential) deterministic Turing machine in polynomial time. It is not known whether the containment is proper. It is conjectured that the hardest problems in $\mathcal{P}$ are not in $\mathcal{NC}$.

Many problems exist for which there are simple sequential algorithms, but these algorithms seem hard to parallelize [1, 16]. The problem of finding a maximal independent set in a graph is one such. This problem was solved in three very different ways [10, 18, 20]. Another example is the problem of coloring graphs using the method implied by Brooks' theorem. Recently, Hajnal and Szemeredi [13] exhibited a good parallel algorithm to do this. There are also good parallel algorithms for five coloring planar graphs [9, 15].

In this paper we look at two different coloring methods. The first is a sequential coloring scheme for general graphs that does better than Brooks coloring in most cases, but we show that it is very unlikely that a *good* parallel algorithm exists for it. We then define a restricted class of graphs and exhibit a good parallel coloring algorithm for them.

2 Preliminaries

Our notation and terminology follows that of Chartrand and Lesniak [6]. A *(simple) graph* $G=(V,E)$ is a set V of *vertices* and a set E of *edges,* which are unordered pairs of vertices. A *multigraph* is a graph in which multiple edges are allowed between the same pair of vertices. For a graph G, we denote the degree of a vertex v by $d(v)$, and the degree of v in a subgraph H of G by $d_H(v)$. We denote the maximum degree of G by $\Delta(G)$, the miminum degree of G by $\delta(G)$. and the chromatic number of G (i.e. the least number of colors needed to color the vertices of G) by $\chi(G)$.

It was shown by P. Hajnal and E. Szemeredi [13] and independently by Howard Karloff [17] and Naor and Karchmer [21] that Brooks' coloring is in $\mathcal{NC}$. But as noted above, better colorings exist for most graphs. One way to color certain graphs using fewer colors is by using what we call the *c-index* of a graph.

Definition 1 *The c-index of a graph G, denoted $p(G)$, is the maximum, over all subgraphs H of G, of the minimum degree $\delta(H)$.*

It is known [23] that any graph G can be colored using at most $p(G)+1$ colors. Furthermore, this can be done in polynomial time, using a suitable ordering of the vertices.

Definition 2 *A minimum degree elimination sequence of a graph G is an ordering $v_1, v_2, \ldots, v_n$ of the vertices of G such that, for every i, the vertex v_i is of minimum degree in the subgraph G_i induced by the vertices $\{v_i, v_{i+1}, \ldots, v_n\}$.*

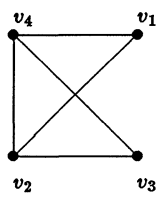


Figure 1: An example minimum degree elimination sequence for a graph

*This author's research is supported in part by NSF grant No. NCR 8706350.

Figure 1 illustrates a graph with four vertices, for which the order v_1, v_2, v_3, v_4 is a minimum degree elimination sequence. Note that a graph does not necessarily have a unique minimum degree elimination sequence; for example, v_3, v_4, v_1, v_2 is also a minimum degree elimination sequence for the graph of Figure 1.

Theorem 2 *If $v_1, \ldots, v_n$ is a minimum degree elimination sequence for a graph G then there exists i such that $p(G) = \delta(G_i)$.*

Proof Let H be a subgraph of G such that $\delta(H)$ is maximum. Given the minimum degree elimination sequence, choose the least i such that $H \preceq G_i$. Clearly, $\delta(G_i) \geq \delta(H)$, because H is a subgraph of G_i; and $\delta(H) \geq \delta(G_i)$, by maximality. The desired conclusion follows.

∎

Theorem 3 *For every graph G, the chromatic number $\chi(G) \leq p(G) + 1$.*

Proof. Let $v_1, \ldots, v_n$ be a minimum degree elimination sequence for G. Color the vertices in the order $v_n, v_{n-1}, \ldots, v_1$ in a "greedy" fashion, as follows: assign to v_n an arbitrary color; for the other vertices, assign to v_i a color that has not been assigned to any of its already-colored neighbors. It is clear from the definition of the minimum degree elimination sequence that when v_i is being colored, no more than $p(G)$ neighbors of v_i have been colored, and so the theorem follows.

∎

Next, we briefly review the notions concerning log-space completeness.

Definition 3 *Let A and B be decision problems. Problem A is said to be log-space reducible to problem B if there exists a function f, that can be computed by a deterministic Turing machine in logarithmic space, such that for every instance w of problem A, w has an affirmative answer whenever the instance $f(w)$ of B has an affirmative answer.*

Definition 4 *A decision problem B is log space complete for $\mathcal{P}$ if*

- *B is in $\mathcal{P}$, and*
- *for every $A \in \mathcal{P}$, A is log-space reducible to B.*

Lemma 4 *If B is log-space complete for $\mathcal{P}$ and B is log-space reducible to $A \in \mathcal{P}$ then A is log-space complete for $\mathcal{P}$.*

∎

Lemma 5 *If B is log space complete for $\mathcal{P}$ and B is in $\mathcal{NC}$ then $\mathcal{P} = \mathcal{NC}$.*

∎

For a more rigorous treatment of this material see [8, 14]. Other related results can be found in [5, 7].

3 The construction

In this section, we show that the problem of determining the order of vertices in any minimum degree elimination sequence of a graph and the problem of determining the c-index $p(G)$ of a graph are complete in $\mathcal{P}$ with respect to logspace reducibility. Our method is to transform the circuit value problem restricted to fanout two (CVP2), which is known to be log-space complete for $\mathcal{P}$ [11, 12, 19, 22], to these two problems. Howard Karloff pointed out recently that the results also follow using $\mathcal{NC}$ reductions from some results of Anderson and Mayr [2, 3].

Let us define formally the two problems of interest.

Problem Π_1:

Instance: Graph G,and two vertices u and $v \in V$; and the property that u appears before v in some minimum degree elimination sequence iff it appears before v in all minimum degree elimination sequences.

Question: Does u appear before v in some minimum degree elimination sequence ?

Problem CVP2:

Instance: A Boolean circuit represented as a sequence $B = (f_1, f_2, \ldots, f_l, B_1, \ldots, B_n)$ with the f_i's representing inputs with value false and the B_i's representing NOR gates. Each NOR gate has fanout at most 2. The circuit has no feedback, i.e. each gate B_i is of the form $\neg(B_j, B_k)$ where $j, k < i$. Each of the f_js are an input to exactly one B_i. The truth value of B is defined as the output truth value of B_n.

Question: Is the value of B false ?

Theorem 6 *CVP2 is complete in $\mathcal{P}$ with respect to logspace reducibility [12, 22].*

∎

We transform CVP2 to the minimum degree elimination sequence problem. Let B be an instance of CVP2. We construct a graph $G = (V, E)$ and pick out two vertices u and v from V such that u follows v in a minimum degree elimination sequence of G iff B_n is true. We construct a multigraph of maximum multiplicity 4 first, and then show how to convert this to a simple graph satisfying the same predicates. The construction is made up of several "components." (The term 'component' here does not mean 'connected component'; it is used to denote a subgraph corresponding to a gate or input.) Each input f_i has a component, each gate B_i has 2 components, and there is a garbage collecting component.

Notation. An edge (u, v) of multiplicity k is represented as $(u, v)_k$.

Every node of the multigraph is a tuple. The first element of the tuple determines the type of component the node is in, the second element determines which gate or input value the node represents, and the remaining

elements of the tuple identify particular nodes that make up the component.

For each input f we have a node $\langle \mathsf{INP}, i \rangle$. The constant INP at the first position tells us that this node corresponds to an input in B, and the second value i tells us the gate B_i to which it is input. The nodes in the multigraph corresponding to the inputs f of the circuit form the set V_{inp}.

Next, consider a gate B_i of the circuit, with inputs from gates B_j and B_k and outputs to gates B_s and B_t. Corresponding to B_i there are 2 components (see Figure 2):

(1) The "true" component, consisting of the subgraph $T_i = (V_{t_i}, E_{t_i})$, where

$$\begin{aligned} V_{t_i} &= \{\langle \mathsf{T}, i, \mathsf{IN}\rangle, \langle \mathsf{T}, i, \mathsf{OUT}, s\rangle, \langle \mathsf{T}, i, \mathsf{OUT}, t\rangle\}, \\ E_{t_i} &= \{(\langle \mathsf{T}, i, \mathsf{IN}\rangle, \langle \mathsf{T}, i, \mathsf{OUT}, s\rangle)_2, \quad (\langle \mathsf{T}, i, \mathsf{IN}\rangle, \\ &\qquad \langle \mathsf{T}, i, \mathsf{OUT}, t\rangle)_2\}. \end{aligned}$$

The element T in the first position indicates that this node is in the true part, and the element i in the second position indicates that this component corresponds to the gate B_i. The constants IN and OUT at the third place signify that these nodes refer to the inputs and the outputs of the gates. We will see later on that the OUT nodes of B_i will be connected to the IN nodes of B_s in some fashion.

(2) The "false" component, consisting of the subgraph $F_i = (V_{f_i}, E_{f_i})$, where

$$\begin{aligned} V_{f_i} &= \{\langle \mathsf{F}, i, \mathsf{IN}\rangle, \langle \mathsf{F}, i, \mathsf{G}\rangle, \\ &\qquad \langle \mathsf{F}, i, \mathsf{OUT}, s\rangle, \langle \mathsf{F}, i, \mathsf{OUT}, t\rangle\} \end{aligned}$$

$$\begin{aligned} E_{f_i} &= \{(\langle \mathsf{F}, i, \mathsf{IN}\rangle, \langle \mathsf{F}, i, \mathsf{G}\rangle)_3, \\ &\qquad (\langle \mathsf{F}, i, \mathsf{G}\rangle, \langle \mathsf{F}, i, \mathsf{OUT}, s\rangle), \\ &\qquad (\langle \mathsf{F}, i, \mathsf{G}\rangle, \langle \mathsf{F}, i, \mathsf{OUT}, t\rangle)\}. \end{aligned}$$

The part of the construction that depends on which gates have what inputs and outputs is the collection of communication edges. The true component of each gate has edges to the false components of its inputs and outputs. More precisely, suppose that the gate B_i has outputs connected to the inputs of gates B_s and B_t (there can be at most two, since we are considering the modified circuit value problem). Then the communication edges between the subgraphs representing B_s, B_t and B_i are:

$$\begin{aligned} E_{c_i} &= \{(\langle \mathsf{T}, i, \mathsf{OUT}, s\rangle, \langle \mathsf{F}, s, \mathsf{IN}\rangle), \\ &\qquad (\langle \mathsf{T}, i, \mathsf{OUT}, t\rangle, \langle \mathsf{F}, t, \mathsf{IN}\rangle), \\ &\qquad (\langle \mathsf{F}, i, \mathsf{OUT}, s\rangle, \langle \mathsf{T}, s, \mathsf{IN}\rangle)_4, \\ &\qquad (\langle \mathsf{F}, i, \mathsf{OUT}, t\rangle, \langle \mathsf{T}, t, \mathsf{IN}\rangle)_4\}. \end{aligned}$$

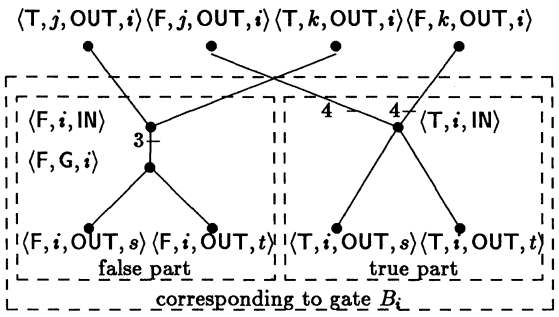


Figure 2: the subgraph corresponding to gate B_i

The idea is to force the nodes to get into the minimum degree elimination sequence in an order that models the flow of truth values through the Boolean circuit. We will force the node $\langle \mathsf{T}, i, \mathsf{IN}\rangle$ to get into the minimum degree elimination sequence iff the output value of gate B_i is true. We will see that nodes with degree 3 or 4 get picked to be put into the minimum degree elimination sequence at each stage and we will force the ordering by saying that when one node (say $\langle \mathsf{T}, i, \mathsf{IN}\rangle$) is picked, the other node ($\langle \mathsf{F}, i, \mathsf{IN}\rangle$) must have degree 5 or more.

To ensure that a node does not go into the minimum degree elimination sequence prematurely, we will need a garbage collection component, which we call GC. Corresponding to every node n in T_i and F_i whose degree needs adjusting there is a node n' in GC which is connected by some number of edges to its image n. The number of edges vary from 1 to 4 depending on the requirement. In addition each node in GC is connected to every other node in GC. This ensures that nodes in GC succeed all other nodes in any minimum degree elimination sequence. To understand the use of GC notice that for any i $\langle \mathsf{T}, i, \mathsf{OUT}, j\rangle$ has degree 3, but we want to ensure that $\langle \mathsf{T}, i, \mathsf{OUT}, j\rangle$ gets into the minimum degree elimination sequence only after $\langle \mathsf{T}, i, \mathsf{IN}\rangle$, so we need to increase its degree to atleast 5. This we do by connecting it to its image in GC. This business of connecting arcs emanating from a node to its image in GC is referred to as garbage collection. A node in the garbage collecting component is identified by gc in the first place.

$$\begin{aligned} E_{g_i} &= \{(\langle \mathsf{T}, i, \mathsf{OUT}, s\rangle, \langle gc, \mathsf{T}, i, \mathsf{OUT}, s\rangle)_3, \\ &\qquad (\langle \mathsf{T}, i, \mathsf{OUT}, t\rangle, \langle gc, \mathsf{T}, i, \mathsf{OUT}, t\rangle)_3, \\ &\qquad (\langle \mathsf{F}, i, \mathsf{G}\rangle, \langle gc, \mathsf{F}, i, \mathsf{G}\rangle)_2\}. \end{aligned}$$

There are just two more things we need consider.

1. When input to some B_i is false,then the communication links between the inputs to the circuit and the input gates to the circuit are as follows
   $E_{cf_i} = \{(\langle \mathsf{INP}, i\rangle, \langle \mathsf{T}, i, \mathsf{IN}\rangle)_4, (\langle \mathsf{F}, i, \mathsf{IN}\rangle, \langle gc, \mathsf{F}, i, \mathsf{IN}\rangle)_2\}$

   The nodes corresponding to inputs, i.e. nodes in V_{inp}, do not have a true part and hence the inputs are garbage collected.

2. If a gate has one or zero outputs then the arcs emanating from the out nodes are again garbage collected.

Now that we have described the connections, let us look at the intuitive ideas behind the constructions.

The elimination sequence for the false part should look like $\langle \mathsf{F}, i, \mathsf{IN} \rangle, \langle \mathsf{F}, i, \mathsf{G} \rangle, \langle \mathsf{F}, i, \mathsf{OUT}, s \rangle$ and $\langle \mathsf{F}, i, \mathsf{OUT}, t \rangle$. We note that when a node gets into the minimum degree elimination sequence the degree of its successor becomes 4 or less. Similarly the elimination sequence for the true part looks as $\langle \mathsf{T}, i, \mathsf{IN} \rangle, \langle \mathsf{T}, i, \mathsf{OUT}, s \rangle$ and $\langle \mathsf{T}, i, \mathsf{OUT}, t \rangle$.

We need $\langle \mathsf{T}, i, \mathsf{IN} \rangle$ to get into the minimum degree elimination sequence before $\langle \mathsf{F}, i, \mathsf{IN} \rangle$ iff the output of B_i is true. We see that if this does happen then $\langle \mathsf{T}, i, \mathsf{OUT}, j \rangle$ gets into the minimum degree elimination sequence before $\langle \mathsf{F}, i, \mathsf{OUT}, j \rangle$. We note the following two things :

1. $d(\langle \mathsf{F}, i, \mathsf{IN} \rangle)$ becomes less than 5 iff one of $\langle \mathsf{T}, j, \mathsf{OUT}, i \rangle$ and $\langle \mathsf{T}, k, \mathsf{OUT}, i \rangle$ get picked into the minimum degree elimination sequence or that atleast one of B_j and B_k is true.

2. $d(\langle \mathsf{T}, i, \mathsf{IN} \rangle)$ becomes less than 5 iff both $\langle \mathsf{T}, j, \mathsf{OUT}, i \rangle$ and $\langle \mathsf{T}, k, \mathsf{OUT}, i \rangle$ get picked into the minimum degree elimination sequence or that both B_j and B_k are false.

So we see the correspondence between the function of a NOR gate and the interconnection pattern.

The graph is the union of all the edges and vertices mentioned above:

$$\begin{aligned} G &= (V, E) \\ V &= V_{inp} \cup V_{t_i} \cup V_{f_i} \cup V_{GC} \\ E &= E_{t_i} \cup E_{f_i} \cup E_{c_i} \cup E_{g_i} \cup E_{cf_i} \cup E_{GC} \end{aligned}$$

A more formal proof of correctness of this construction is presented in [1].

We next show how to convert the multigraph into a simple graph without altering any of the previous results.

1. $(a, b)_2$ is replaced by adding 6 other nodes $c_1, c_2, c_3, d_1, d_2, d_3$ and the new edges We first form a K_6 minus the edges (c_1, c_3) and (d_1, d_3) and now connect a to c_1 and c_3 and b to d_1 and d_3. We note that for any of the new nodes the degree will become less than 5 *iff* one of a or b is removed. And it is seen that if one of a or b is removed the degree of the other can be decreased by 2 without increasing $p(G)$ to 5.

2. $(a, b)_4$ is converted to multiplicity 2 by adding 2 other nodes c and d and connecting using arcs of multiplicity 2, both a and b, to c and d and also connecting c and d.

3. $(a, b)_3$ is converted by a similar method only that the multiplicity of (a, c) and (b, d) is now one.

We now show that the following problem is complete in $\mathcal{P}$ with respect to logspace reducibility.

Problem Π_2 :

Instance: A graph G.

Question: Is $p(G) \leq 4$?

The problem with the previous construction is that p was determined by the GC. We see that B_n is true *iff* $\langle \mathsf{T}, n, \mathsf{IN} \rangle$ gets into the minimum degree elimination sequence before $\langle \mathsf{F}, n, \mathsf{IN} \rangle$. Infact before $\langle \mathsf{F}, n, \mathsf{IN} \rangle$ gets into the minimum degree elimination sequence we come across a subgraph with $\delta = 5$. We utilize this fact and change GC. For each arc going to a node in GC from the basic graph we will have a separate node. Call these set of nodes N_1 (the subscript will be apparent in a moment). All nodes in N_1 are connected to atleast 1 and atmost 2 other nodes in N_1 in any arbitrary fashion. Let number of nodes in N_1 be l. $V_{GC} = N_1 \cup N_2 \cup \cdots N_{\lceil \log_2 l \rceil + 2}$. We will call the subscript on N as levels. N_2 also contains l nodes as images of nodes in N_1 and are connected by arcs of multiplicity 4 to their images. Level 2 upto level $\lceil \log_2 l \rceil + 2$ looks like a binary tree, with nodes at level 2 forming the leaves except that

- if a node has only one child then the multiplicity of that arc is 3.
- all other arcs have multiplicity 2.
- every level has atmost one node having one child.

Figure 3 shows levels 1 and above of a garbage collecting component when the number of nodes at level 1 is five. Note that there are 5 levels. One other change is required. The arcs emanating from $\langle \mathsf{T}, n, \mathsf{OUT}, * \rangle$ are not garbage collected but connected to the root i.e. the node at the topmost level in GC.

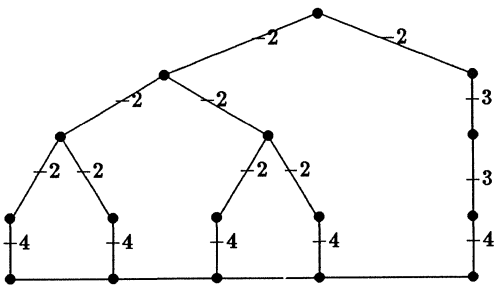


Figure 3: An example garbage collecting component

Theorem 7 *Output of B is true iff $p = 4$.*

Proof: If B is true then $\langle \mathsf{T}, n, \mathsf{IN} \rangle$ gets into the $MDES(l)$ and then $\langle \mathsf{T}, n, \mathsf{OUT}, * \rangle$. Now, the root in GC will have degree 4 and hence can get into the MDES. Easy to see that for a node at level i if its parent has been removed then its degree is less than 5. So all of GC can be removed keeping p 4, the rest of the graph follows suit. If on the other hand B were false then once $\langle \mathsf{F}, n, \mathsf{OUT}, * \rangle$ is put in the minimum degree elimination sequence the resultant graph has $\delta = 5$.

∎

4 An $\mathcal{NC}$ coloring algorithm

So we see that improving the bound of Brook's theorem may not be easy to do in $\mathcal{NC}$. However there is hope in the case of graphs where there is disparity in the degree of nodes in the graph. The following discussion illustrates our point. We consider the case when a node with large degree is connected to a large number of nodes of small degree.

Theorem 8 *Let $q(G)$ be the minimum number such that a graph G does not have a $K_{q(G)+1}$ and each vertex v with degree $\geq q(G)$ has at most $q(G)$ neighbours that have degree $\geq$ than $q(G)$. Then G can be colored in $\mathcal{NC}$ time using at most $q(G)$ colors.*

Proof The idea is to color the nodes with degree $\geq q(G)$ first. The other nodes can be colored without increasing the number of colors since each of them is adjacent to at most $q(G) - 1$ nodes. More formally let $I = \{v : d(v) \geq q(G)\}$. Consider the graph G_I induced by I. Since each node in I is is adjacent to atmost $q(G)$ nodes of degree $\geq q(G)$ the maximum degree of a node in G_I is $q(G)$. So G_I can be colored using $q(G)$ colors in $\mathcal{NC}$ time using Hajnal and Szemeredi's algorithm. The rest of the graph can now be colored using *procedure extend* mentioned in Luby's paper [20]. For details see [1].

5 Acknowledgements

The authors would like to thank Howard Karloff, Ernst Mayr and an annonymous referee for a lot of helpful suggestions.

6 References

[1] S. Vishwanathan, "Parallel graph algorithms ," M.S. thesis, University of South Carolina, Columbia, 1988.

[2] R. Anderson and E. Mayr, "A P-Complete problem and approximations to it ," Technical Report, Computer Science Dept. Stanford University., Sept. 1984.

[3] R. Anderson and E. Mayr, "Parallelism and Greedy Algorithms," Technical Report, Computer Science Dept. Stanford University., April 1984.

[4] R. L. Brooks, "On colouring the nodes of a network," *Proc. Cambridge Philos. Soc.* 37 (1941), pp. 194-197.

[5] A. Chandra, D. Kozen, and L. Stockmeyer, "Alternation," *Journal of the ACM* 28 (January 1981), pp. 114-133.

[6] G. Chartrand and L. Lesniak, *Graphs and digraphs*, Wadsworth, Inc., 1986.

[7] S. A. Cook, "A taxonomy of problems with fast parallel algorithms," *Information and Control* 64 (1985), pp. 2–22.

[8] M. R. Garey and D. S. Johnson, *Computers and intractibility: a guide to the theory of NP-completeness*, W. H. Freeman and Co., 1979.

[9] A. Goldberg, S. Plotkin, and G. Shannon, "Parallel symmetry-breaking in sparse graphs," *Proc. 19th annual ACM STOC.* 19 (1987), pp. 315-324.

[10] M. Goldberg and T. Spencer, "A new parallel algorithm for the maximal independent set problem," *Proc. 28th annual symp. on FOCS* 28 (1987), pp. 161-165.

[11] L. M. Goldschlager, "The monotone and planar circuit value problems are log space complete for P," *SIGACT News* 9 (2) (1977),

[12] L. M. Goldschlager, R. A. Shaw, and J. Staples, "The maximum flow problem is log space complete for P," *Theoretical Computer Science* 21, North-Holland (1982), pp. 105-111.

[13] P. Hajnal and E. Szemeredi, Brooks coloring in parallel, manuscript, 1987.

[14] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, MA, 1979.

[15] Boyar J. and Karloff H., Coloring planar graphs in parallel, manuscript, 1986.

[16] H. Karloff, "Fast parallel algorithms for graph-theoretic problems: matching, coloring and partitioning," Ph.D. thesis, University of California, Berkeley, 1985.

[17] H. Karloff, An NC algorithm for Brooks' theorem, manuscript, 1986.

[18] R. M. Karp and A. Wigderson, "A Fast Parallel Algorithm for the Maximal Independent Set Problem," *Proc. 16th ACM Symp. Theory of Computing*, 1984, pp. 266-272.

[19] R. E. Ladner, "The circuit value problem is log space complete for P," *SIGACT News* 7 (1975), pp. 18-20.

[20] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *Proc. 17th ACM STOC*, 1985, pp. 1-10.

[21] J. Naor and M. Karchmer, A fast parallel algorithm to color a graph with Δ colors, manuscript, 1986.

[22] J. H. Reif, "Depth-first search is inherently sequential," *Information Processing Letters* 20, North-Holland (1985), pp. 229-234.

[23] G. Szekeres and H. S. Wilf, "An inequality for the chromatic number of a graph," *Journal of Combinatorial Theory* 4 (1968), pp. 1-3.

SUBGRAPH ISOMORPHISM FOR CONNECTED GRAPHS OF BOUNDED VALENCE AND BOUNDED SEPARATOR IS IN NC

Andrzej Lingas
Department of Computer and Information Science
Linköping University
581 83 Linköping, Sweden

Abstract: We present a parallel algorithm for subgraph isomorphism restricted to a connected graph H of bounded valence and a connected graph G of bounded valence and bounded 0-1 weighted separator, i.e. a "$1/3 - 2/3$" separator for any assignment of 0-1 weights to vertices of G. Our algorithm runs in time $O(\log^3 n)$ using polynomial number of processors, i.e. it is an NC^3 algorithm.

1. *Introduction*

The *subgraph isomorphism* problem is to determine whether a graph can be *imbedded* in another graph, i.e. whether the former is isomorphic to a subgraph of the latter. It is a fundamental graph problem with a variety of applications in engineering sciences, organic chemistry, pattern recognition. For instance, if H is an n-vertex circuit and G is an n-vertex planar graph of valence 3, $n \in N$, then determining whether H can be imbedded in G is equivalent to the NP-complete problem of determining whether a planar graph of valence 3 has a Hamiltonian circuit [3]. Thus, the subgraph isomorphism problem is NP-complete even if G and H range only over connected planar graphs of valence ≤ 3. Subgraph isomorphism also remains NP-complete when the first input graph is a forest and the other input graph is a tree (see pp. 105 in [3]).
The only known polynomial-time algorithms for subgraph isomorphism are those for trees [14,15], two-connected outerplanar graphs [8], and two-connected series-parallel graphs [12]. Recently, it has been also shown that the problem of subgraph isomorphism for two-connected outerplanar graphs is in the class NC [11] (i.e. can be solved in poly-log time using polynomial number of processors [2]), and that the subgraph isomorphism problems for trees and two-connected series-parallel graphs respectively are in the random class NC (see [4,10] and [12] respectively). Are there other non-trivial classes of graphs for which subgraph isomorphism is in NC or at least can be solved in polynomial time?
In [7], it was shown that subgraph isomorphism restricted to connected graphs of bounded valence and bounded separator (the so called "$1/3 - 2/3$" separator) can be solved sequentially in time $n^{O(\log n)}$. In [9], the above result has been strengthened by showing that the above problem can be solved in parallel in poly-log time using $n^{O(\log n)}$ processors.
In this paper, we strengthen these results from [7,9] by presenting an NC^3 algorithm for subgraph isomorphism restricted to a connected graph H of bounded valence and a connected graph G of bounded valence and bounded 0-1 weighted separator, i.e. a "$1/3 - 2/3$" separator for any assignment of 0-1 weights to vertices of G. The algorithm is based on a non-trivial double use of the weighted separator, similar to that for subgraph homeomorphism in [11,16]. Seymour and Robertson have recently shown that any proper sub-family of planar graphs closed under the minor operation, as well as any family of graphs of bounded tree width have bounded 0-1 weighted separator [16,17]. Thus, our result applies to the case where G is any graph in the above families restricted to connected graphs of bounded valence. In particular G can be a connected series-parallel graph (see [6]) of bounded valence, or more generally, a partial k-tree. As for H, it should be a connected graph of bounded valence.

The remainder of the paper is divided into three sections. In Section 2 we introduce basic notions, definitions, and facts used in the succeeding sections. In Section 3 we present the algorithm and analyze its time complexity.

2. *Preliminaries*

We shall use standard set and graph theoretic notation and definitions (for instance, see [1,3]). Specifically, we assume the following set and graph conventions:

1) Given a partial mapping π of T into U, $dom(\pi)$ denotes the set of all elements of T on which π is defined. Next, given a subset T' of T, $\pi(T')$ denotes
$\{\pi(e) \mid e \in dom(\pi) \cap T'\}$.
2) For a graph G, $V(G)$ denotes its set of vertices.
3) Given a subset V' of $V(G)$, $G(V')$ denotes the the subgraph of G induced by V'.
4) Given graphs G_i, $i = 1, ..., k$, $\bigcup_{i=1}^{k} G_i$ (or, $G_1 \cup ... \cup G_k$ equivalently) denotes the graph G where
$V(G) = \bigcup_{i=1}^{k} V(G_i)$ and two vertices of G are adjacent if and only if there is i, $1 \leq i \leq k$, such that the vertices are adjacent in G_i.

For the definitions of the classes NC^k, NC, the reader is referred to [2].
We shall consider the following restriction of the subgraph isomorphism problem.

Definition 2.1: Let H, G be two graphs, and let π be a partial one-to-one mapping of $V(H)$ into $V(G)$. The π-*imbedding* problem for H and G is to decide whether there exists an isomorphism between H and a subgraph of G

that is an extension of π. Such an isomorphism is called a *π-imbedding* of H in G.

Note that the problem of subgraph isomorphism can be expressed as the π-imbedding problem where π is an empty mapping. We use dynamic programming to solve the π-imbedding problem for graphs H, G, where the cardinality of the domain of π is bounded and H, G are of bounded valence and bounded number of connected components, and G is of bounded weighted separator.
For technical reasons, we define the concept of bounded weighted separator of a graph with $0-1$ vertex weights (see [13]) through that of an m-separation of graph. It is left to the reader to verify that our definition is equivalent to a standard one.

Definition 2.2: Let m be a positive integer. Let G be a graph and let $W_1, W_2, ..., W_k$ be subsets of $V(G)$. Finally, let W be a subset of $V(G)$ of cardinality not greater than m. The sequence $(W_1, W_2, ..., W_k, W)$ is an *m-separation* of a graph G if the removal of W from G disconnects G into connected components C_i, $i = 1, ..., k$, where $V(C_i) = W_i$. A graph G is said to have an *m-separator* if for any subset U of $V(G)$ there is is an m-separation $(W_1, W_2, ...,$
$W_k,\ W)$ of G such that none of the intersections $W_i \cap U$, $i = 1, ..., k$, has more than $(2/3) \mid U \mid$ vertices.

Remark 2.1: If G has an m-separator then any subgraph of G has also an m-separator.
Hint: By the assignment of 0 weights to the vertices of G outside the subgraph one can extract the subgraph from G.

The correctness of our NC divide-and-conquer algorithm will in part follow from the following technical lemma. Its proof, as easy but lengthy, is left to the reader.

Lemma 2.1: Let H, G be graphs with m-separator. Next, let π be a partial mapping of $V(H)$ into $V(G)$. There is a π-imbedding of H in G if and only if for any m-separation $(W_1, ..., W_k,\ W)$ of G, there are an m-separation $(V_1, ..., V_l,\ V)$ of H, a partition P of $\{1, ..., l\}$, a one-to-one mapping f of P into $\{1, ..., k\}$, and one-to-one partial mappings π_S of $V(H)$ into $V(G)$, $S \in P$, such that for $S \in P$:
1) $H(\bigcup_{j \in S} V_j \cup V)$ can be π_S imbedded in
$G(W_{f(S)} \cup W)$,
2) $dom(\pi) \cap (\bigcup_{j \in S} V_j) \subseteq dom(\pi_S)$, and $V \subseteq dom(\pi_S)$,
3) $\pi_S(V) \subseteq W$,
4) π_S is consistent with π and with each of the other mappings $\pi_{S'}$, $S' \in P$.

3. *The NC algorithm*

Here, we consider a new approach to the subgraph isomorphism problem for connected graphs of bounded valence and bounded (weighted) separator. A naive method of guessing the vertices of the separator in the first graph and guessing their image in the second graph can lead to un-polynomial number of considered subgraphs, when applied recursively [7,9]. However, we will be able to keep the maximum number of vertices through which a recursive subgraph is connected to the rest of the graph constantly bounded using the weighted version of the separator, following the general idea of Robertson and Seymour for subgraph homeomorphism (see [11,16]).

Our parallel procedure for π imbedding uses as a subroutine the procedure $SEP(F, U, b, m)$ returning
m-separations of the input graph F whose components contain no more than b elements of the input subset U of $V(F)$. The procedure SEP is a straight-forward generalization of the procedure $2SEP$ from [12].

procedure $SEP(F, U, b, m)$
input: a graph F, a subset U of $V(F)$, and positive integers b, m.
output: the set of all m-separations $(W_1, W_2, ..., W_k, W)$ of F where for $i = 1, ..., k$, $\mid W_i \cap U \mid \leq b$.

for for all subsets W of $V(F)$ with at most m vertices of F **do in parallel**
begin
  $X \leftarrow TRUE$;
  find the connected components $D_1, ..., D_k$ of the graph resulting from deleting W from F;
  for $i = 1, ..., k$ **do in parallel**
  if $\mid (V(D_i) \cap U) \mid > b$ **then** $X \leftarrow FALSE$;
  if X **then return** $(W_1, W_2, ..., W_k,\ W)$
end

By a straight-forward generalization of Lemma 5.1 in [12], we have:

Lemma 3.1: For a fixed m, the procedure SEP can be realized by an NC^2 algorithm.
Sketch: It is sufficient to observe that the number of all subsets W of $V(F)$ with at most m elements is is $O(n^m)$ and that the connected components $D_1, ..., D_k$ can be constructed by a concurrent read concurrent write parallel RAM with a polynomial number of processors in time $O(\log n)$ [18], and hence by NC^2 algorithm by [19]. ∎

The main recursive procedure SI for π-imbedding is as follows.

procedure $SI(H, T, G, U, m, \pi)$
Input: graphs H, G, each of valence $\leq d$, G with m-separator, a subset T of $V(H)$, a subset U of $V(G)$, and a one-to-one mapping π of T into U which is
a sub-isomorphism between $H(T)$ and $G(U)$.
Output: If there is a π-imbedding of H in G then YES

$c \leftarrow \max\{\mid U \mid,\ m\}$
if $\mid T \mid > \mid U \mid$ or $\mid V(H) \mid > \mid V(G) \mid$ **then go to** E;
if $\mid V(H) \mid \leq \frac{10}{3}c$ **then**
begin
  decide whether there is a π-imbedding of H in G
  by brute force, if so return YES;
  go to E;
end
if $\mid U \mid > \frac{10}{3}c$ **then**

begin
Pick an m-separation $(W_1,...,W_k,\ W)$ returned by $SEP(G,U,(2/3) \mid U \mid)$
for $i = 1,...,k$ **do in parallel**
begin
$G_i \leftarrow G(W_i \cup W)$;
$U_i \leftarrow U \cap W_i \cup W$;
for all $(V_1,...,V_l,\ V)$ returned by $SEP(H,T,(2/3) \mid U \mid)$ **do in parallel**
for $j = 1,2,...,l$ **do in parallel**
begin
$H_j \leftarrow H(V_j \cup V)$;
$T_j \leftarrow T \cap V_j \cup V$;
end
for all subsets $\{T_{j_1},...,T_{j_r}\}$ of $\{T_1,T_2,...,T_l\}$ where $\mid \bigcup_{p=1}^{r} T_{j_p} \mid \leq \mid U_i \mid$ **do in parallel**
for all one-to-one mappings π' of $\bigcup_{p=1}^{r} T_{j_p}$ into U_i that are consistent with π and define a sub-isomorphism between $G(\bigcup_{p=1}^{r} T_{j_p})$ and $G(U)$ **do in parallel**
if $SI(\bigcup_{p=1}^{r} H_{j_p}, \bigcup_{p=1}^{r} T_{j_p}, G_i, \pi(\bigcup_{p=1}^{r} T_{j_p}), m, \pi')$ **then** $M((j_1,...,j_r),i) \leftarrow 1$;
Using the table M decide by Lemma 2.1 and brute force whether there exists a π-imbedding of H into G mapping V into W, if so, return YES;
end;
end
if $\mid U \mid \leq \frac{10}{3}c$ **then**
begin
Pick an m-separation $(W_1,...,W_l,\ W)$ of G returned by $SEP(G,V(G),(2/3) \mid V(G) \mid)$
for $i = 1,2,...,k$ **do**
begin
$G_i \leftarrow G(W_i \cup W)$;
$U_i \leftarrow U \cap W_i \cup W$;
for all $(V_1,...,V_l,\ V)$ returned by $SEP(H,V,(2/3) \mid V(G) \mid)$ **do in parallel**
for $j = 1,2,...,l$ **do in parallel**
begin
$H_j \leftarrow H(V_j \cup V)$;
$T_j \leftarrow T \cap V_j \cup V$;
end
for all subsets $\{T_{j_1},...,T_{j_r}\}$ of $\{T_1,T_2,...,T_l\}$ where $\mid \bigcup_{p=1}^{r} T_{j_p} \mid \leq \mid U_i \mid$ **do in parallel**
for all one-to-one mappings π' of $\bigcup_{p=1}^{r} T_{j_p}$ into U_i that are consistent with π and define a sub-isomorphism between $G(\bigcup_{p=1}^{r} T_{j_p})$ and $G(U)$ **do in parallel**
if $SI(\bigcup_{p=1}^{r} H_{j_p}, \bigcup_{p=1}^{r} T_{j_p}, G_i, \pi(\bigcup_{p=1}^{r} T_{j_p}), m, \pi')$ **then** $M((j_1,...,j_r),i) \leftarrow 1$;
Using the table M decide by Lemma 2.1 and brute force whether there exists a π-imbedding of H into G mapping V into W, if so, return YES;
end;
E:**end**

To outline the proof of the correctness of SI we shall use the following lemma.

Lemma 3.2: Procedure SI is correct.
Sketch: Suppose first that $\mid U \mid < \frac{10}{3}c$. Since G has an m-separator, $SEP(G,V(G),\ (2/3) \mid V(G) \mid)$ returns at least one m-separation. Further, suppose that there is a π-imbedding ϕ of H into G. Then, by Lemma 2.1, for the m-separation $(W_1,...,W_l,\ W)$ of G there is an m-separation $(V_1,...,V_k,\ V)$ of H returned by $SEP(H,V(H),\mid V(H) \mid)$, and a partition P of $\{1,...,l\}$ with a one-to-one mapping f of P into $\{1,...,k\}$ such that for $S \in P$, $\bigcup_{j \in S} H_j$ can be π_S-imbedded in $G_{f(S)}$ where π_S is a one-to-one mapping of $\bigcup_{j \in S} T_j$ into U_i consistent with π, and mapping V into W. Note that the graphs $H(\bigcup_{j \in S} H_j)$ and $G_{f(S)}$ have m-separators by Remark 2.1. Therefore, they are valid parameters for the recursive calls of SI. Conversely, if such an m-separation of H and a one-to-one mapping f exist then there is a π-imbedding of H in G by Lemma 2.1.
Suppose in turn that $\mid U \mid > \frac{10}{3}c$. The purpose of SI in this case is to reduce the original problem to imbedding subproblems where the subset U of $V(G)$ is split appropriately. Suppose that there is a π-imbedding ϕ of H into G. Since G has an m-separator, $SEP(G,U,(2/3) \mid U \mid)$ returns at least one m-separation. Further, suppose that there is a π-imbedding ϕ of H into G. Then, by Lemma 2.1, for the m-separation $(W_1,...,W_k,\ W)$ of G there is an m-separation $(V_1,...,V_l,\ V)$ of H returned by $SEP(H,T,(2/3) \mid U \mid)$ and a one-to-one mapping f of a partition P of $\{1,...,l\}$ into $\{1,...,k\}$ such that:
for $S \in P$, $\bigcup_{S \in P} H_S$ can be π_S-imbedded in $G_{f(S)}$ where π_S is a one-to-one mapping of $\bigcup_{S \in P} T_S$ into U_i consistent with π and the other mappings $\pi_{S'}$, $S' \in P$. Conversely, if at least one of the above conditions is satisfied then there is a π-imbedding of H in G. ∎

Lemma 3.3: If H, G are connected and the maximum c of the parameter m and the cardinality of the parameter U is constantly bounded then the procedure SI can be implemented by NC^3 circuits.
Sketch: We may assume without loss of generality that $\mid V(H) \mid \leq \mid V(G) \mid$ and $\mid V(H) \mid \geq \frac{10}{3}c$. The thesis of the lemma follows from the following propositions:

i) The subsets T and U have never more than $4c$ elements in any recursive call of SI.
ii) The recursion depth of SI is logarithmic.
iii) The body of SI can be implemented by NC^2 circuits if we do not count the recursive calls and the calls of SEP.
iv) SEP can be realized by an NC^2 algorithm.

Before proving the above propositions, let us show how they yield the thesis of the lemma. By the definition of SI, the propositions (ii), (iii) and (iv) ensure $O(\log^3 n)$ depth. By induction on the depth of a recursive call of SI, we easily observe that the subgraphs H', G' of H and G that are are parameters in the call are separated from the rest of the graph by vertices in the subsets parameters, say T', and U'. Since T', U' are of size $O(c)$ by (i), and H, G are of valence $\leq d$, the number of connected components of H' and G' is $O(cd)$, i.e. $O(1)$. Combining the above fact with (i), we conclude that the number of all possible parameters

in the direct recursive calls of $SI(H,T,G,U,m,\pi)$ is $O(1)$. Hence, by (ii), the number of recursive calls of SI on all recursion levels is polynomially bounded. This combined with (iii) and (iv) shows that we can compute all these calls using polynomial number of processors, keeping the $O(\log^3 n)$ depth.

The proof of (i) is by induction on the recursive depth of a call SI. At the zero depth, the subsets U and T are of cardinality bounded by c. Assume that (i) holds for the calls of SI at the recursive depth q. Let Q be a vertex set that is a parameter of one of the above calls of SI. If Q has at most $\frac{10}{3}c$ vertices than it can be expanded maximally by m vertices (W or V respectively) before further recursive calls. If Q has more than $\frac{10}{3}c$ and at most $3.5c$ (respectively, at most $4c$) vertices then it is split into parts of at most $3c$ (respectively, $\frac{10}{3}c$) vertices that can be augmented by at most m vertices before further recursive calls. This completes the proof of (i).

Let us prove (ii). If SI did not contain the case $\mid U \mid > \frac{10}{3}c$ then it would be of depth proportional to the partition tree of G induced by the "$1/3 - 2/3$" separator of G, i.e. logarithmic in the size of G. On the other hand, it takes at most two such calls to reduce the size of the subsets Q from $4c$ to at most $\frac{10}{3}c$ by the proof of (i). Hence, if we follow a path in the tree of recursive calls of SI then we never encounter more than two consecutive calls of SI dealing with the case $\mid U \mid > \frac{10}{3}c$, which does not increase the size of the graph parameters. By the above arguments, we can conclude that SI is of logarithmic depth.

The proposition (iii) follows from the fact that the number of all possible m-separations $(V_1,...,V_l,\ V)$, $(W_1,...,W_k,\ W)$ is $O(n^m)$, and the number of all possible mappings π' considered in the body of SI is constantly bounded by (i) and $l = O(1)$ (see the part of the proof preceding the proof of (i)). Note also that the set intersections in the body of SI are computed only for finite sets. Thus, the body of SI without the invoked procedure calls can be implemented by a concurrent read exclusive write parallel RAM with a polynomial number of processors in logarithmic time. Hence, it can be implemented by NC^2 circuits by [19]. Finally, the proposition (iv) follows from Lemma 3.1. ∎

Theorem 3.1: Let d and m be positive integer constants. Let H and G be connected graphs of valence $\leq d$. Assume that G has an m-separator. The problem of subgraph isomorphism for such graphs H, G is in NC^3.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, Massachusetts, 1974).

[2] S.A. Cook, *The taxonomy of problems with fast parallel algorithms*, Information and Control 64(1985),2-22.

[3] M.R. Garey, D.S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-completeness* (Freeman, San Francisco, 1979).

[4] P. Gibbons, R.M. Karp, G.L. Miller and D. Soroker, *Subtree isomorphism is in random NC*, manuscript, September 1987.

[5] F. Harary, *Graph Theory* (Addison-Wesley, Reading, Massachusetts, 1969).

[6] R. Hassim, A. Tamir, *Efficient algorithms for optimization and selection on series-parallel graphs*, SIAM J. Alg. Disc. Math. 7(1986), 379-389.

[7] A. Lingas, *Subgraph Isomorphism for Easily Separable Graphs of Bounded Valence*, in Proc. of 11th Workshop on Graph-theoretic Concepts in Computer Science, Castle Schwanberg (Trauner, 1985), 217-229.

[8] A. Lingas, *Subgraph Isomorphism for Biconnected Outerplanar Graphs in Cubic Time*, Proc. 3rd STACS's, Orsay (1986), LNCS 210, Springer.

[9] A. Lingas, *On Parallel Complexity of the Subgraph Isomorphism Problem*, technical report LiTH-IDA-R-87-10, Linköping University.

[10] A. Lingas and M. Karpinski, *Subtree isomorphism and bipartite perfect matching are mutually NC reducible, technical report*, LITH-IDA-R-87-09, Linköping University, May 1987.

[11] A. Lingas and A. Proskurowski, *Fast Parallel Algorithms for the Subgraph Homeomorphism and the Subgraph Isomorphism Problems for Classes of Planar Graphs*, in Proc. the 7th Conference on Foundations of Software Technology and Theoretical Computer Science, Pune, India 1987, LNCS, Springer.

[12] A. Lingas and M. Sysło, *A Polynomial Algorithm for Subgraph Isomorphism of Two-connected Series-Parallel Graphs*, in Proc. the 15th ICALP, Tampere, Finland 1988, LNCS, Springer.

[13] R.J. Lipton and R.E. Tarjan, *Applications of a planar separator theorem*, SIAM J. Computing 9 (1980) no. 3, 513-524.

[14] D. W. Matula, *Subtree isomorphism in* $O(n^{5/2})$, Annals of Discrete Mathematics 2 (1978) 91-406.

[15] S. W. Reyner, *An analysis of a good algorithm for the subtree problem* , SIAM J. Comput. 6 (1977), 730-732.

[16] N. Robertson and P.D. Seymour, *Disjoint paths - a survey*, SIAM J. Alg. Disc. Meth., Vol. 6 (1985), No. 2, 300-305.

[17] N. Robertson and P.D. Seymour, *Graph Minors 2. Algorithmic Aspects of Tree-Width*, J. Algorithms No. 7, 309-322.

[18] Y. Shiloach and U. Vishkin, *An* $O(\log n)$ *parallel connectivity algorithm*, J. Algorithms 3, 1, pp. 57-67.

[19] L. Stockmeyer and U. Vishkin, *Simulation of parallel random Access Machines by Circuits*, SIAM J. Compt. 13 (1984), pp. 409-422.

OPTIMAL SORTING ON REDUCED ARCHITECTURES

R. Cypher (*) J.L.C. Sanz (**)

ABSTRACT: This paper studies the problem of sorting N items on a P processor parallel machine, where $N \geq P$.. The central result of the paper is a new algorithm, called cubesort, that sorts $N = P^{1+1/k}$ items in $O(k\ P^{1/k} \log P)$ time using a P processor shuffle-exchange. Thus for any positive constant k, cubesort provides an asymptotically optimal speed-up over sequential sorting. Cubesort also sorts $N = P \log P$ items using a P processor shuffle-exchange in $O(\log^3 P/\log\log P)$ time. Both of these results are faster than any previously published algorithms for the given problems. Cubesort also provides asymptotically optimal sorting algorithms for a wide range of parallel computers, including the cube-connected cycles and the hypercube. An important extension of the central result is an algorithm that simulates a single step of a Priority-CRCW PRAM with N processors and N words of memory on a P processor shuffle-exchange machine in $O(k\ P^{1/k} \log P)$ time, where $N = P^{1+1/k}$.

1. Introduction

This paper presents a new parallel algorithm for sorting N items using P processors, where $N \geq P$. This new algorithm can be implemented efficiently on a wide range of parallel computers, including the hypercube, the shuffle-exchange and the cube-connected cycles. In particular, the algorithm runs in $O((N \log N)/P)$ time on any of the above architectures, provided $N = P^{1+1/k}$ for some positive constant k. This is the first sorting algorithm for any of the above architectures that obtains this performance. In addition, the sorting algorithm will be extended to obtain an efficient simulation of a Priority-CRCW PRAM using a hypercube, shuffle-exchange or cube-connected cycles. The remainder of this section reviews models of parallel computers and examines previous work in the field of parallel sorting.

The models of parallel computers that will be used in this paper are the PRAM [5], the hypercube [10], the shuffle-exchange [10] and the cube-connected cycles [11]. These models operate in an SIMD mode, with all of the processors performing the same instruction at any given time. The PRAM is a shared memory model in which all processors can access a common memory in unit time. The Priority-CRCW PRAM allows multiple processors to read from or write to a single memory location simultaneously. In the case of simultaneous writes to a single location, the lowest numbered processor attempting to write to that location succeeds.

The hypercube, the shuffle-exchange and the cube-connected cycles consist of a set of processors, each containing a local memory, that communicate with one another using a fixed interconnection network. In the hypercube, the P processors are numbered 0 .. P-1 and processors i and j are connected if the binary representations of i and j differ in exactly 1 bit position. In the shuffle-exchange, the P processors are numbered 0 .. P-1 and processors i and j are connected if $j = \text{Shuffle}(i, P)$, $j = \text{Unshuffle}(i, P)$ or $j = \text{Exchange}(i)$ where $\text{Shuffle}(i, P) = 2i \bmod (P-1)$, $\text{Unshuffle}(i, P) = j$ iff $\text{Shuffle}(j, P) = i$, and $\text{Exchange}(i) = i+1 - 2(i \bmod 2)$. The cube-connected cycles contains P processors, where $P = 2^K$ and $K = R + 2^R$. The processors are numbered with pairs (b, c) where b is a (K-R) bit number and c is an R bit number. Processor (b, c) is connected to processor (d, e) if $b = d$ and $c = e+1$, if $b = d$ and $c = e-1$, or if $c = e$ and the binary representations of b and d differ in only the e-th bit position. The shuffle-exchange and the cube-connected cycles are *feasible* models because each processor is connected to only a fixed number of other processors.

One of the earliest results in parallel sorting was obtained by Batcher. In [3], Batcher presented the bitonic sorting algorithm. In [12], Stone showed that the bitonic sort could be implemented on a shuffle-exchange. This yields an $O(\log^2 N)$ time sort for $N = P$ numbers on the shuffle-exchange.

In [4], Baudet and Stevenson show how any parallel algorithm for sorting N items with $P = N$ processors that is based on comparisons and exchanges can be used to obtain an algorithm for sorting N items with $P < N$ processors. By applying their technique to the bitonic sort on the shuffle-exchange, they obtained an $O((N/P) \log (N/P) + (N/P) \log^2 P)$ time sorting algorithm when $P \leq N$. Their algorithm provides an optimal speed-up over sequential comparison sorting only when $P = O(2^{\text{sqrt}(\log N)})$.

An algorithm for a special case of the sorting problem was given by Gottlieb and Kruskal [6]. They presented a shuffle-exchange algorithm for the permutation problem, where the N numbers to be sorted are in the range 1 through N and where each number appears exactly once. Their algorithm requires $O(P^{9/2} + (N/P) \log P)$ time and gives optimal speed-up over sequential comparison sorting when $P = O((N \log N)^{2/9})$. In their paper, Gottlieb and Kruskal state that they do not know of an optimal algorithm for the permutation problem when P is not in $O((N \log N)^{2/9})$. The current paper thus improves upon Gottlieb and Kruskal's result in two ways. First, the algorithm presented in this paper solves the general sorting problem rather than the permutation problem. Second, the algorithm presented here gives optimal speed-up when $N = P^{1+1/k}$ for any positive constant k.

A breakthrough in parallel sorting was obtained by Ajtai, Komlos and Szemeredi [2]. They created a network for sorting N items that consists of O(N log N) comparators and has O(log N) depth. This network was used by Leighton to create a feasible parallel machine that sorts in O(log N) time when $P = N$ [8].

(*) Computer Science Department University of Washington, Seattle, WA 98195
(**) Computer Science Department IBM Almaden Research Ctr, San Jose, CA 95120

Unfortunately, there are two serious difficulties with Leighton's technique. First, the technique performs poorly for $P < 10^{100}$. In contrast, the algorithm presented in this paper has a much smaller constant of proportionality and is much more likely to be useful in practice. Second, Leighton's network is not a standard network that has been shown to be useful for solving problems other than sorting. In contrast, the shuffle-exchange and the cube-connected cycles have been proven useful in solving a wide range of problems.

Another important related result was obtained by Leighton. Leighton has recently shown that his algorithm called columnsort [8] can be used to obtain an efficient algorithm for sorting $N = P^{1+1/k}$ items on a P processor shuffle-exchange [9]. He obtains an $O(k^T P^{1/k} \log P)$ time algorithm, where $T = 1/\log_4 1.5$ (T is approximately 3.419). The algorithm is based on calling columnsort in a nested manner so that the N items are sorted by repeatedly sorting groups of $P^{1/k}$ items each. Furthermore, there is a possibility that the value of the exponent T can be reduced to less than 1 by using Leighton's concept of closesorting [8],[9]. Finally, a similar result using columnsort was obtained by Aggarwal [1]. More research into the applications of columnsort is clearly needed.

The paper is divided as follows. Section 2 presents an abstract description of the new sorting algorithm and proves its correctness. Section 3 shows how this sorting algorithm can be implemented efficiently on a number of parallel computers and it presents an algorithm for simulating a Priority-CRCW PRAM with a shuffle-exchange computer. Throughout this paper, N will be the number of items to be sorted and P will be the number of processors available.

2. *Cubesort*

This section contains a description of a new parallel sorting algorithm that the authors call *cubesort*. The description of cubesort given in this section is independent of the architecture that is used to implement it. Cubesort works by repeatedly partitioning the N items to be sorted into small groups and sorting these groups separately and in parallel. In particular, let $N = M^D$ where M and D are integers. Each step of cubesort partitions the M^D items into either M^{D-1} groups of M items each or M^{D-2} groups of M^2 items each, and sorts the groups in parallel.

The M^D items to be sorted can be viewed as occupying a D-dimensional cube, where each side of the cube is of length M. Each location L in the cube has an address of the form $L = (L_D, L_{D-1}, ..., L_1)$, where $(L_D, L_{D-1}, ..., L_1)$ is a D-digit base-M number and L_i is the projection of location L along the i-th dimension. This numbering of the locations in the cube corresponds to an ordering of the locations that will be called *row-major order*. Cubesort will sort the items in the cube into row-major order.

In addition to viewing the items as forming a single D-dimensional cube, they can be viewed as forming a number of cubes of smaller dimension. A *j-cube*, where $0 \le j \le D$, is a set of M^j items with base-M addresses that differ only in the j least significant digits. That is, $A = (A_D, A_{D-1}, ..., A_1)$ and $B = (B_D, B_{D-1}, ..., B_1)$ are in the same j-cube if and only if $A_i = B_i$ for all i, $j+1 \le i \le D$. Each j-cube, where $0 \le j \le D$, is classified as being either *even* or *odd*. A j-cube, where $0 \le j \le D-1$, is even if it contains a location L where $L_{j+1} \bmod 2 = 0$, and it is odd otherwise. The D-cube that contains all N items is defined to be even.

There are D different partitions, represented as P_j where $1 \le j \le D$, that are used by cubesort. A group in partition P_j consists of a set of items with base-M addresses that differ only in digits j and j-1. Note that each group in P_1 contains M items, while each group in the remaining partitions contains M^2 items.

Finally, it is sometimes useful to view the items in a j-cube as forming a 2-dimensional array. A *j-array*, where $2 \le j \le D$, is an $M^2 \times M^{j-2}$ array of the items in a j-cube, where the items are placed in the array in row-major order. Thus each (j-2)-cube forms a row in a j-array, and each (j-1)-cube forms a band of M consecutive rows in a j-array. Also, each column in a j-array is a group in P_j.

In order for cubesort to work correctly, it is assumed that $D \ge 3$ and that $M \ge (D-1)(D-2)$. Cubesort makes use o two subroutines, Sort_Ascending and Sort_Mixed. The subroutine Sort_Ascending(i) sorts the groups in partition P_i in ascending row-major order. The subroutine Sort_Mixed(i, j) sorts the groups in partition P_j that are in even i-cubes in ascending order, while it sort the groups in P_j that are in odd i-cubes in descending order. Cubesort is called by first setting the global variables M and D and then calling Cubesort(D). The pseudo-code of cubesort is given below.

```
Cubesort(S)                          /* Abstract Descr
iption of Cubesort */
integer S;
{
   if S = 3 then
   {
                                     /* PHASE 1: */
      Limit_Dirty_Cubes(S);

                                     /* PHASE 2: */
      Sort_Mixed(S-1, S-1);

                                     /* PHASE 3: */
      Merge_Dirty_Cubes(S, S);
   }
   else
   {
                                     /* PHASE 1: */
      Limit_Dirty_Cubes(S);
      Limit_Dirty_Cubes(S);

                                     /* PHASE 2: */
      Cubesort(S-1);

                                     /* PHASE 3: */
      Merge_Dirty_Cubes(S, S);
   }
}


Limit_Dirty_Cubes(S)
integer S;
{
   if S > 2 then
      Limit_Dirty_Cubes(S-1);
   Sort_Ascending(S);
}


Merge_Dirty_Cubes(S, T)
integer S, T;
{
   Sort_Mixed(S, T);
   if T > 2 then
```

```
        Merge_Dirty_Cubes(S, T-2);
}
```

The call Cubesort(S) sorts each even S-cube in ascending row-major order and each odd S-cube in descending row-major order. In order to prove that cubesort works correctly, it is necessary to use the zero-one principle [7], which states that "if a network with n input lines sorts all 2^n sequences of 0's and 1's into nondecreasing order, it will sort any arbitrary sequence of n numbers into nondecreasing order". In keeping with the zero-one principle, the following discussion will assume that the input consists entirely of 0's and 1's.

The following definitions will be needed in the proof of correctness. A set of items is *dirty* if it contains both 0's and 1's, and it is *clean* otherwise. A sequence of 0's and 1's is *ascending* if it is of the form 0^a1^b, where a,b $\geq$ 0, and it is *descending* if it is of the form 1^a0^b, where a,b $\geq$ 0. A sequence is *monotonic* if it is either ascending or descending. A sequence of 0's and 1's is *bitonic* if it is of the form $0^a1^b0^c$ or of the form $1^a0^b1^c$, where a,b,c $\geq$ 0. A j-array is *cross-sorted* if all of its rows are monotonic and if it has at most 1 ascending dirty row and at most 1 descending dirty row. A j-array is *semi-sorted* if all of its rows are bitonic and if it has at most 1 dirty row. A j-array is *block-sorted in ascending (descending) order* if it consists of A rows containing only 0's (1's), followed by B dirty rows, followed by C rows containing only 1's (0's), where A,B,C $\geq$ 0. A j-cube is cross-sorted (or semi-sorted or block-sorted) if its corresponding j-array is cross-sorted (or semi-sorted or block-sorted).

The correctness of Cubesort(S) is established next. In order to save space, the proofs have been omitted.

LEMMA 1: If a j-array originally has B dirty rows, and if the columns of the j-array are then sorted in ascending (descending) order, the resulting j-array will be block-sorted in ascending (descending) order and will contain no more than B dirty rows.

LEMMA 2: After calling Limit_Dirty_Cubes(i), where i $\geq$ 2, there are at most i-1 dirty (i-1)-cubes in each i-cube, and the dirty (i-1)-cubes are consecutive within each i-array.

LEMMA 3: If a j-array is originally semi-sorted or cross-sorted, and if the columns of the j-array are then sorted in ascending (descending) order, the resulting j-array will be semi-sorted and it will be block-sorted in ascending (descending) order.

THEOREM 1: After calling Cubesort(3), each even 3-cube is sorted in ascending row-major order and each odd 3-cube is sorted in descending row-major order.

LEMMA 4: When S > 3, after Phase 1 there are at most 2 dirty (S-1)-cubes in each S-cube, and these dirty cubes are adjacent to one another in the S-array.

LEMMA 5: For any values of S and T, where 1 $\leq$ T $\leq$ S $\leq$ D, if originally each T-cube is either semi-sorted or cross-sorted, and if Merge_Dirty_Cubes(S, T) is then called, the resulting T-cubes will all be sorted. Furthermore, the T-cubes that are in even S-cubes will be sorted in ascending order and the T-cubes that are in odd S-cubes will be sorted in descending order.

THEOREM 2: Cubesort(S), where 3 $\leq$ S $\leq$ D, sorts each even S-cube in ascending order and each odd S-cube in descending order.

3. *Implementing Cubesort*

The cubesort algorithm given in the previous section sorts N = M^D numbers by performing $O(D^2)$ stages, where each stage consists of sorting, in parallel, groups containing $O(M^2)$ items. This section will show how cubesort can be implemented on a variety of parallel models.

First, the implementation of cubesort on a shuffle-exchange will be presented. It will be assumed that there are N items to be sorted and that P processors are available, where N = $P^{1+1/k}$. The items to be sorted are stored in an N item array A, where A_i is located in processor j = floor($i/P^{1/k}$), for 0 $\leq$ i $\leq$ N-1. In order to use the algorithm from the previous section, let D = 2k+2, let M = $P^{1/2k}$, and let location L = (L_D, L_{D-1}, ..., L_1) in the D-dimensional cube correspond to A_I.

Before the groups of a partition are sorted, the data are rearranged so that each group lies within a single processor. There are 2 permutations that are used to perform this rearrangement, namely the M-Shuffle and the M-Unshuffle. The definition of the M-Shuffle of N items is that M-Shuffle(X, N) = MX mod (N-1). The M-Unshuffle is the inverse of the M-Shuffle, so M-Unshuffle(Z, N) = X iff M-Shuffle(X, N) = Z.

The $P^{1/k}$ items that are local to each processor can be sorted in $O((1/k)\ P^{1/k}\ \log P)$ time. Also, the M-Shuffle and M-Unshuffle of the items to be sorted can each be accomplished in $O((1/k)\ P^{1/k}\ \log P)$ time. Because the shuffle-exchange implementation of cubesort consists of $O(D^2) = O(k^2)$ applications of sorts that are local to processors and $O(k^2)$ applications of the M-Shuffle and M-Unshuffle routines, the entire algorithm requires $O(kP^{1/k}\ \log P)$ time.

The implementations of cubesort on the hypercube and the cube-connected cycles are similar to the implementation on the shuffle-exchange. Because of space limitations, only the result will be stated. Cubesort can be implemented in $O(k^2P^{1/k}\ \log P)$ time on a hypercube or cube-connected cycles.

In the above discussion, it was assumed that N = $P^{1+1/k}$. However, cubesort can also be used when the number of items per processor grows more slowly. In particular, when N = P log P cubesort yields an $O(\log^3 P/\text{loglog } P)$ time sorting algorithm for a P processor shuffle-exchange. Again, space limitations prevent including that algorithm.

Finally, cubesort can be used to simulate a Priority-CRCW PRAM with a shuffle-exchange computer. Because of space limitations, only the result will be stated. A single operation of a Priority-CRCW PRAM with N processors and N memory locations can be implemented in $O(kP^{1/k}\ \log P)$ time on a P processor shuffle-exchange, where N = $P^{1+1/k}$.

Acknowledgments

The authors would like to express their gratitude to Prof. T. Leighton for bringing to their attention his extensions o columnsort for optimal sorting, and for many useful discussions. Also, the authors feel indebted to Prof. S. Hambrusch, Dr. M. Snir, Prof. L. Snyder and Dr. E. Upfal for their helpful comments on the contents of this paper. The work of R. Cypher was supported in part by an NSF graduate fellowship.

References and Notes

1. A. Aggarwal, *Unpublished manuscript, 1986.*
2. M. Ajtai, J. Komlos, E. Szemeredi, *"An O(n log n) Sorting Network", Proc. 15th Annual Symposium on Theory of Computing, 1983, pp. 1-9.*

3. K.E. Batcher, *"Sorting Networks and their Applications", 1968 AFIPS Conference Proceedings, pp. 307-314.*
4. G. Baudet, D. Stevenson, *"Optimal Sorting Algorithms for Parallel Computers", IEEE Transactions on Computers, vol. c-27, no. 1, January 1978, pp. 84-87.*
5. A. Borodin, J.E. Hopcroft, *"Routing, Merging and Sorting on Parallel Models of Computation", Proc. 14th Annual Symposium on Theory of Computing, 1982, pp. 338-344.*
6. A. Gottlieb, C.P. Kruskal, *"Complexity Results for Permuting Data and Other Computations on Parallel Processors", Journal of the ACM, vol. 31, no. 2, April 1984, pp. 193-209.*
7. D.E. Knuth, *"The Art of Computer Programming, Vol. 3: Sorting and Searching", Addison-Wesley, Reading, MA, 1973.*
8. T. Leighton, *"Tight Bounds on the Complexity of Parallel Sorting", IEEE Transactions on Computers, vol. c-34, no. 4, April 1985, pp. 344-354.*
9. T. Leighton, *Personal communication..*
10. D. Nassimi, S. Sahni, *"Data Broadcasting in SIMD Computers", IEEE Transactions on Computers, vol. c-30, no. 2, February 1981, pp. 101-107.*
11. F.P. Preparata, J. Vuillemin, *"The Cube-Connected Cycles: A Versatile Network for Parallel Computation", Communications of the ACM, vol. 24, no. 5, May 1981, pp. 300-309.*
12. H.S. Stone, *"Parallel Processing with the Perfect Shuffle", IEEE Transactions on Computers, vol. c-20, no. 2, February 1971, pp. 153-161.*