MIT/LCS/TR-317

AN ABSTRACT ARCHITECTURE FOR PARALLEL GRAPH REDUCTION

Kenneth R. Traub

This blank page was inserted to preserve pagination.

AN ABSTRACT ARCHITECTURE FOR PARALLEL GRAPH REDUCTION

by

Kenneth R. Traub

AN ABSTRACT ARCHITECTURE FOR PARALLEL GRAPH REDUCTION

by

Kenneth R. Traub

ABSTRACT

An implementation technique for functional languages that has received recent attention is *graph reduction*, which offers opportunity for the exploitation of parallelism by multiple processors. While several proposals for parallel graph reduction machines have been made, differing terminology and approaches make these proposals difficult to compare. This paper presents a systematic approach to the study of parallel graph reduction machines, and proposes an abstract architecture for such a machine that is independent of the base language and communication network chosen for an actual implementation. The abstract architecture, in addition to serving as a foundation for the design of real machines, lends quite a bit of insight into the essence of parallel graph reduction.

Keywords: Abstract Machines, Applicative Languages, Computer Architecture, Multiple Processor Architectures, Parallel Processing, Reduction.

ACKNOWLEDGEMENTS

It is quite possible that this thesis would never have existed were it not for the late nights Tim Chambers and I spent trying to complete the combinator reduction program for 6.847, so thank you, Tim, for helping me to understand reduction in the first place. The greatest thanks, of course, are owed my thesis advisor, Professor Arvind, who was a constant source of encouragement, and whose guidance improved the quality of this work immeasurably. And finally, a hearty thank you to Richard Soley, who provided timely insight into the intricacies of the Lisp Machine, and to Professors Donald Troxel and Steve Ward, who unknowingly supplied large amounts of computer facilities and resources.

This document was originally submitted to the Department of Electrical Engineering and Computer Science at M.I.T. on May 11, 1984 in partial fulfillment of the requirements for the Degree of Bachelor of Science in Electrical Engineering and Computer Science. Thesis advisor: Arvind, Associate Professor of Electrical Engineering and Computer Science.

1. Introduction

1.1. Background

An implementation technique for functional languages that has received recent attention is *reduction*. In reduction machines, the program is represented as a directed graph of operators and data, and is executed by the repeated application of identities, or *reduction rules*, that simplify portions of the graph until the original graph is transformed into the final result. Reduction machines can be divided into two broad categories: *string reduction* machines, in which there is no sharing of subgraphs, and *graph reduction* machines, in which there may be. The subgraph sharing in the latter can confer self-optimization properties upon its programs; the G-machine[3] and the SKIM machine[1] are uniprocessor machines that attempt to exploit this property.

Both graph reduction and string reduction approaches offer opportunities for parallel evaluation since several portions of the program graph may be reduced simultaneously. Mago[5] has described a parallel string reduction machine; Keller *et. al.*[4], Darlington and Reeve[2], and Sleep and Burton[6], have each made proposals for parallel graph reduction machines. The proposed graph reduction machines use different reduction languages, different communication networks, and different mechanisms for coordinating parallel execution, making it difficult to compare the machines to determine what aspects represent necessary features of all graph reduction machines and what aspects are features of the individual machines.

1.2. Parallel Graph Reduction Machines - A Systematic Approach

Figure 1 depicts the hierarchy of issues relating to the design of a parallel graph reduction machine. At the innermost level is the reduction base language itself; that is, the set of rules for transforming a graph into a printable answer, along with an algorithm for their systematic application. Since the design of a sequential reduction machine such as the G-machine encounters these issues alone, the issues at this level can be called the *sequential-semantic* issues.

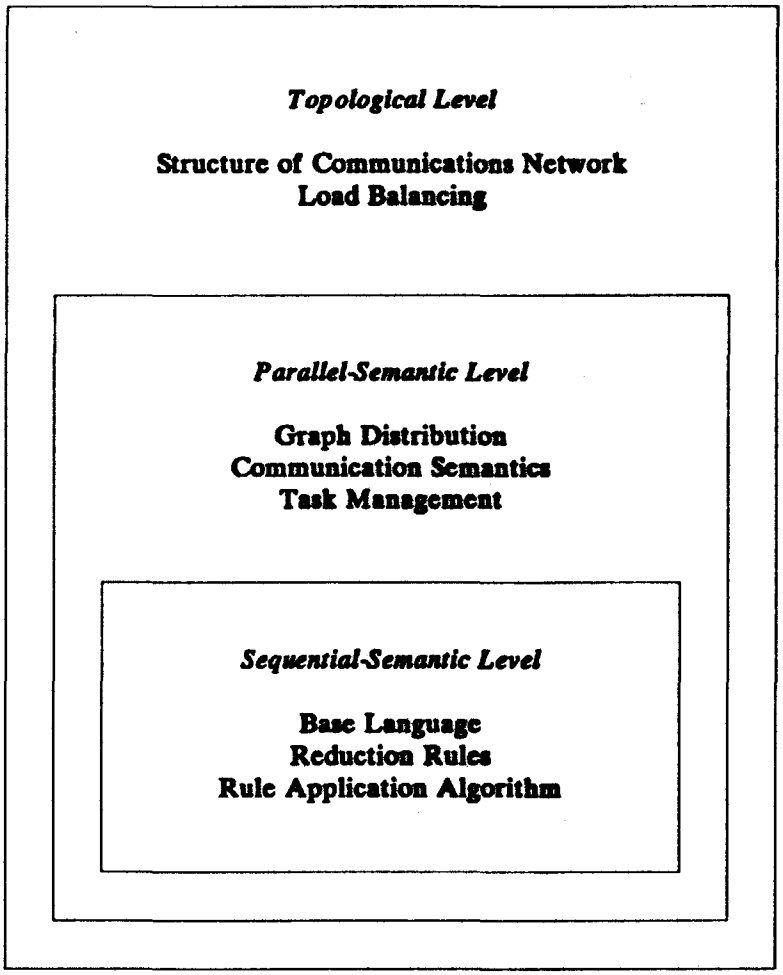

Figure 1. Hierarchy of Issues in the Design of a Parallel Graph Reduction Machine

One level out are the issues related to the "parallelization" of the reduction process. Any parallel reduction machine attempts to employ many individual processing elements (PEs) in the concurrent reduction of a single graph. This introduces problems of where to place the graph in relation to the PEs, of what information must be communicated by the PEs, and of what work must be done by each PE over and above the application of reduction rules. These can be called *parallel-semantic* issues.

Finally, at the outermost level, is the structure of the communications network that supports the intra-PE information flow proscribed by the parallel semantics; this level is called the *topological level*. As will be seen, the issues related to load balancing are most appropriately dealt with at this level.

Past proposals for parallel graph reduction machines have made no attempt to discuss the issues in each of the three layers separately. In particular, the boundary between the sequential-semantic and parallel-semantic layers is usually blurred, obscuring the distinction between language particulars and essential parallel reduction mechanism. No author has yet given a complete and detailed description of all issues embodied in the parallel-semantic layer, yet it is precisely these issues that are the essence of parallel graph reduction.

This paper attempts to concretely define and describe those aspects of a parallel graph reduction machine that fall into the parallel-semantic level of Figure 1 in a manner applicable to all languages and network topologies. What emerges can be thought of as an abstract parallel graph reduction machine, which when imbued with a particular reduction language and circumscribed by a particular communication network becomes a correct design for an actual machine. While a language based on Turner's combinators[7] will be used for illustrative purposes, it will be shown that the parallel-semantic layers of the existing proposals, to the extent that they are described at all, fit the model developed here. This in turn suggests that all parallel graph reduction machines *must* function as described here at the parallel-semantic level, regardless of their sequential-semantic and topological design.

2. The Sequential-Semantic Layer

In order to understand parallel reduction, it is first necessary to understand sequential reduction, and so a brief look will be taken at the sequential-semantic layer before proceeding on to the parallel-semantic layer. A subset of Turner's combinator language will be used to highlight the important points.

In all graph reduction machines, the program is expressed in a *constant applicative form* (CAF) language, in which there are no variables, only constants. These constants appear in a graph structure, and the reduction rules guide the machine in successively replacing substructures with simpler ones until all that remains is a single printable result. The program graph, then, is a collection of nodes, where each node contains one or more fields containing pointers to atomic constants or to other nodes. When a subgraph is to be reduced, a pointer to the root node of the subgraph is passed to a reduction algorithm procedure. This procedure examines the subgraph and applies the appropriate reduction rules, possibly causing the reduction of other subgraphs or the creation of new nodes. When reduction is complete, the reduction procedure returns the value that results, and replaces the original contents of the root node of the subgraph reduced with the result of reduction. The three important characteristics of the reduction algorithm are:

(1) It is a *procedure* that takes one argument: a pointer to the root node of the subgraph to be reduced.

(2) It *returns* one value: the result of reducing that subgraph. The result may be an atom or a more complex value.

(3) It has the *side-effect* of modifying the graph. The most important side-effect is that the root node of the subgraph reduced is replaced with the result of reduction.

Because the root node of a subgraph plays such an important role in that subgraph's reduction (its address is passed to the reduction procedure; its contents are replaced by the result), "reducing node N" is considered synonymous with "reducing the subgraph of which node N is the root".

To get a feel for what kind of operations are involved in the reduction of a node, a language based on a subset of Turner's combinator language will be presented. While Turner's combinator code is perhaps the least readable of all CAF languages, its semantics are quite simple and elegant, allowing the essential features of all CAF languages to be highlighted without getting too bogged down in language details.

The reduction rules for a subset of Turner's language is shown in Figure 2. In that figure, lowercase letters refer to any arbitrary graph, the notation <x> means "the result of reducing *x*", and the left arrow indicates both what is returned and what replaces the node being reduced[*]. Figure 3 shows in detail the reduction procedure to apply those rules. Here are some examples of reduction using this procedure; it will be helpful to refer to Figure 3 when reading these examples.

Example 1: E = I +.

Step 1: let T = Reduce(fn(E)) = I
An atom is already reduced, by definition.
Step 2: let Q = Reduce(op(E)) = +
Step 3: Write-op(E,Q)
The graph is left as I +_
Step 4: return Q
and the atom + is returned.

To compute <*f x*>,
use the following rules to compute <<*f*> *x*>:

<I *x*> → <*x*>

<K *x*> → K *x*

<K *x y*> → <*x*>

<+ *x*> → + *x*

<+ *x y*> → <*x*>+<*y*>

<S *f*> → S *f*

<S *f g*> → S *f g*

<S *f g x*> → <*f x* (*g x*)>

otherwise, → ERROR

Figure 2. A Small Reduction Language Based on Turner's Combinators

---

[*] If the result of reduction is an atom *a*, by convention the node reduced is replaced by I *a*. Such a node is called an *indirection node* by Turner.

The Reduction Procedure:
Given a pointer to a graph, *E*, reduce the graph and return the result.

```
procedure Reduce(E) {
    Start:
    let T = Reduce(fn(E));
    if T is an atom then {
        if T = I then {                    /* The rule <I x> → <x> */
            let Q = Reduce(op(E));
            Write-op(E,Q);
            return Q;}
        else if T = K then                 /* The rule <K x> → K x */
            Write-fn(E,T);
            return E;
        else if T = + then                 /* The rule <+ x> → + x */
            Write-fn(E,T);
            return E;
        else if T = S then                 /* The rule <S f> → S f */
            Write-fn(E,T);
            return E;
        else {                             /* The "error rule" */
            Write-fn(E,I);
            Write-op(E,ERROR);
            return ERROR;}}
    else if fn(T) is an atom then {
        if fn(T) = K then {                /* The rule <K x y> → <x> */
            let Q = Reduce(op(T));
            Write-fn(E,I);
            Write-op(E,Q);
            return Q;}
        else if fn(T) = + then {                    /* The rule <+ x y> → <x>+<y> */
            let Q = Reduce(op(T)) + Reduce(op(E));
            Write-fn(E,I);
            Write-op(E,Q);
            return Q;}
        else if fn(T) = S then             /* The rule <S f g> → S f g */
            Write-fn(E,T);
            return E;}
    else if fn(fn(T)) is an atom then
        if fn(fn(T)) = S then {            /* The rule <S f g x> → <f x (g x)> */
            let F = op(fn(T));
            let G = op(T);
            let X = op(E);
            Write-fn(E,Create(F,X));
            Write-op(E,Create(G,X));
            goto Start;}
} /* End of procedure Reduce */
```

Other Procedures Called by Reduce	
fn(*E*)	Returns the function field of the node pointed to by *E*.
op(*E*)	Returns the operand field of the node pointed to by *E*.
Write-fn(*E*,*X*)	Writes *X* in the function field of the node pointed to by *E*.
Write-op(*E*,*X*)	Writes *X* in the operand field of the node pointed to by *E*.
Create(*X*,*Y*)	Creates a new node, initializes its function field to *X* and its operand field to *Y*, and returns a pointer to it.

Figure 3. A Reduction Procedure for the Language in Figure 2.

Example 2: E = (I +) 3.

Step 1: let T = Reduce(fn(E)) = +
This reduction was illustrated in Example 1.
Step 2: Write-op(E,T)
The graph is left as + 3_.
Step 3: return E
and + 3 is returned.

Example 3: E = ((I +) 3) ((+ 4) 5)

Step 1: let T = Reduce(fn(E)) = + 3
This reduction was illustrated in Example 2.
Step 2: let Q = Reduce(op(T)) + Reduce(op(E)) = 3 + 9 = 12
op(T) = 3 (an atom), and op(T) = (+ 4) 5, which reduces to 9.
Step 3: Write-fn(E,I)
Step 4: Write-op(E,Q)
The graph is left as I 23_.
Step 5: return Q
and the atom 12 is returned.

Example 4: E = ((S +) (+ 3)) 4

Step 1: let T = Reduce(fn(E)) = (S +) (+ 3)
This reduction is similar to Example 2.
Step 2: let F = op(fn(T)) = +
fn(T) = S +, so op(fn(T)) = +.
Step 3: let G = op(T) = (+ 3)
Step 4: let X = op(E) = 4
Step 5: Write-fn(E,Create(F,X))
E's fn is now the new graph + 4
Step 6: Write-op(E,Create(G,X))
E's op is now the new graph (+ 3) 4
Hence, E is now the graph (+ 4) ((+ 3) 4)
Step 7: goto Start
The whole reduction procedure is started again on the new version of E.
This will eventually get reduced to 11.

These four examples are typical of the types of reduction rules encountered in most reduction languages. In Example 1 the node is unchanged; in Example 2 some descendents of the node are reduced and the results stored back into the node; in Example 3 descendents are reduced, a computation performed on the results, and the result of the computation returned and stored back into the graph; in Example 4 new nodes are created, the graph rearranged, and the reduction rules reapplied to the result. It should be noted that in Example 4 the node is is considered reduced not at Step 7 but only when a return statement is finally executed; the writing of a node does not necessarily take place only at the conclusion of its reduction. It should also be noted that in Step 2 of Example 3 the two reductions required could be performed simultaneously in a parallel machine; in general parallelism is obtained by "forking" demand across strict operators in this way.

While there are many CAF languages other than Turner's, the reduction procedures to implement those languages will be quite similar to the procedure in Figure 3. A careful examination of Figure 3 and the examples presented will reveal that there are only five kinds of operations performed on the graph during the reduction of a node N:

(1) Reading the fields of node N.

(2) Writing the fields of node N.

(3) Creating new nodes.

(4) Calling for the reduction of descendent nodes of node N.

(5) Reading the fields of those descendent nodes that have been reduced.

(The term "descendent node of node N" here denotes a node that is reached through the tracing of a chain of pointers of bounded length rooted at node N.) It is particularly important to note that the only node an instance of the reduction procedure writes is the node it is reducing. Stated another way, a node can only be altered by the instance of the reduction procedure that reduces it. This implies that once a node is reduced, *it is never written again*; nodes become constants after they are reduced.

The five kinds of operations listed above are the *only* ways in which the reduction procedure is permitted to interact with the program graph. Any other computation performed by the reduction procedure is limited to manipulation of its internal state. Such manipulation would include arithmetic operations on data obtained from the graph, comparisons in order to select a reduction rule, etc. Limiting the reduction procedure's access to the graph to the five operations above is not an arbitrary restriction but an observation that reflects the nature of graph reduction in general. This universal property of the sequential-semantic layer will be the guiding force in the development of the parallel-semantic layer.

3. The Parallel-Semantic Layer

3.1. Machine Organization

In a parallel reduction machine, there are many processing elements (PEs) all trying to reduce one graph. The first question to be resolved, then, is where the graph is to lie in relation to the PEs. An obvious approach is to place the graph in a memory that is *shared* among the PEs so that each PE has equal access to all nodes of the graph. While this approach is conceptually attractive, it introduces severe problems related to maintaining atomicity of operations performed upon the memory. Furthermore, it is clear that contention for the shared memory will swamp the benefits obtained from parallelism for even a modest number of PEs.

To eliminate the contention issue, each PE is given a certain amount of its own local graph memory, to which only it has access. This in turn requires that the program graph be distributed among the graph memories of the PEs, and so nodes of the graph must be able to point to other nodes that reside both in the local PE and in other PEs. A pointer to a node, therefore, must be a tuple of the form (*PE address*), where *PE* is the PE on which the node pointed to resides, and *address* is the address in that PE's local memory. Another way of viewing this scheme is as one large contiguous address space that is divided up among the PEs. A node residing in the memory of one PE can *refer* to a node residing in a different PE, but a node can be read or written only by that node's PE; i.e., by the PE in whose local memory that node resides.

Of course, there must be some sort of *communication network* between the PEs if they are to work in concert. In designing the parallel-semantic layer the only assumption made about the communications network is that a PE may send an arbitrary message to another PE; all other details of the network are properly dealt with in the topological layer. While the communication network is in some sense a shared resource, the design at the topological layer can be chosen to reduce any contention problems to a suitable level; the same cannot be said for a shared memory.

Distributing the nodes among the local memories of the PEs provides a natural way to divide the work of reducing the graph: the work of reducing any particular node - applying reduction

rules, etc. - is assigned to that node's PE. Node (2 45), therefore, will always be reduced by PE number 2, node (7 12) by PE number 7. This assignment of work is only natural, for the reduction of a node N is guaranteed to require reading and writing the fields of node N, and only node N's PE has the privilege of accessing node N. One effect of this assignment is that the distribution of nodes among the PE's memories is equivalent to distributing work among the PE's processors; if all nodes of a graph were placed in one PE's memory, only that PE's processor could take part in the reduction of that graph.

3.2. Inter-PE Communication Essentials

With the basic structure of the machine in hand, it is now necessary to make it function. In the previous section, the five kinds of operations performed on a graph during reduction were enumerated. It is the task of the parallel-semantic layer to insure that a method for accomplishing each of these operations exists in the parallel machine.

Implmenting the first two operations, reading and writing the node being reduced, are easy, since the node being reduced always resides in the graph memory of the PE performing the reduction. These operations are simple accesses to local memory.

The third and fourth kinds of operations, creating new nodes and calling for the reduction of existing nodes, require the assistance of other PEs; the former because new nodes will sometimes have to be created on other PEs to distribute the workload, and the latter because reduction of existing nodes is constrained to take place on each individual node's PE. In a sequential machine, the reduction procedure would accomplish these operations through procedure calls: a call to the "create" procedure creates a new node and returns a pointer, a call to the "reduce" procedure reduces a node and returns the result. In a sequential machine, of course, the latter is a recursive call. The reduction procedure in the parallel machine also can accomplish these operations through procedure calls, but in this case these procedures might require execution on a different PE. What is needed is a *remote procedure call* facility.

To implement remote procedure calls, we turn to the communications network. A remote procedure call in the parallel reduction machine is accomplished by a pair of messages: a *request* message, sent from caller to callee, communicating the arguments of the procedure, and an *acknowledgement* message, sent from callee to caller, communicating the results. Any side-effects caused by the remote procedure are restricted to the local memory of the callee. A request message takes the form:

reqest-id	*type*-REQ	*arguments*

while an acknowledgement looks like:

reqest-id	*type*-ACK	*results*

The *type* fields of the messages indicate in effect what procedure is being called, and the *request-id* field, copied by the called PE from request to acknowledgement, allows the acknowledgement message to be routed to the calling PE and identified there. Figure 4 lists the messages used in parallel reduction.

The first two messages in Figure 4 are used in the creation of new nodes. Suppose PE #1 wants to create a node and have it reside in the memory of PE #2. From a semantic point of view, PE #1 would like to call a procedure like Create(*initial-contents*), where *initial-contents* are the initial values for the fields of the new node, and have a pointer to the new node returned as a result. Note that PE #1 expects not only a returned result, but also the side effect of the creation of a new node. Using the remote procedure call mechanism, PE #1 prepares a **CREATE-REQ** message and sends it to PE #2. PE #1 then waits until it receives a **CREATE-ACK** message whose *request-id* field matches the request-id it created for the earlier request. When that message is received, PE #1 examines the *results* field to obtain a pointer to the new node.

From PE #2's point of view, PE #2 receives a **CREATE-REQ** message. It responds by allocating space for a node in its local memory, initializing the new node according to the *initial-contents* field of the message, and sending back a **CREATE-ACK** message containing a pointer to

(1) **Creation Request**

request-id	CREATE-REQ	*initial-contents*

Requests the creation of a new node initialized to *initial-contents*.

(2) **Creation Acknowledgement**

request-id	CREATE-ACK	*new-pointer*

Informs the sender of a CREATE-REQ message that the new node is pointed to by *new-pointer*.

(3) **Reduction Request**

request-id	REDUCE-REQ	*pointer*

Requests that the subgraph pointed to by *pointer* be reduced.

(4) **Reduction Acknowledgement**

request-id	REDUCE-ACK	*result*

Informs the sender of a REDUCE-REQ message that the result of reduction is *result*.

(5) **Increment Reference Count Request**

request-id	INCREF-REQ	*pointer*

Requests the reference count of the node pointed to by *pointer* be incremented.

(6) **Increment Reference Count Acknowledgement**

request-id	INCREF-ACK

Informs the sender of an INCREF-REQ message that the reference count has been incremented.

(7) **Decrement Reference Count Request**

request-id	DECREF-REQ	*pointer*

Requests the reference count of the node pointed to by *pointer* be decremented.

All messages carry a request identification in the field *request-id*. The request identification is created by the issuer of a request and copied from request message to acknowledgement message by the receiver of a request.

Figure 4. Inter-Processor Messages

the node. The pointer, of course, will be of the form (2 *address*). The *request-id* field of the request message contains the name of the sender, PE #1, so that PE #2 knows to whom to address the acknowledgement. PE #2 copies the entire request-id field from request message to acknowledgement. Thus with the aid of the first two messages in Figure 4, the third kind of operation required by reduction algorithms is accomodated.

The next two messages in the Figure implement the fourth kind of operation, the calling for of the reduction of another node. Here, the procedure call simulated is Reduce(*pointer*), where *pointer* is a pointer to the node to be reduced, which returns the result of reduction as well as having the side effect of altering the node reduced. The implementation of this procedure through message passing is analogous to the implementation of the "create" procedure: a **REDUCE-REQ** message carries a pointer to the node to be reduced to that node's PE, and that PE responds by reducing the node and sending back a **REDUCE-ACK** message that contains a copy of the result.

The subject of what exactly is returned in a **REDUCE-ACK** message requires some thought. If the result of a reduction is an atom, then the atom itself can simply be returned. If the result of reduction is a subgraph, however, it is not obvious what must be returned. Merely returning a pointer to the subgraph is not always sufficient, for the caller will generally need to access some of the nodes in this subgraph (i.e., the fifth kind of operation as listed in Section 2), which it cannot do if the subgraph remains on another PE. On the other hand, the entire subgraph should not be returned, not only because this is far more information than is needed, but also because the entire subgraph is not necessarily available to the PE preparing the acknowledgement, as it may be distributed across many machines.

The simplest policy is to return a copy of the root node of the subgraph to be returned; that is, to return a copy of the node reduced. The PE receiving the acknowledgement then takes the node from the acknowledgement and places it in its own local memory, and may then treat the new node in local memory as though it were the node on the foriegn machine. In doing this operation, two copies of the same node are created, raising the question of consistency. There is

no need to worry about consistency, however, for the node copied is a node that has already been reduced. As pointed out in Section 2, a node that has been reduced can never be altered again – it is effectively a constant until it is garbage collected. Thus, creating a copy of a reduced node is safe, since it amounts to creating a copy of a constant.

Before moving on, it is worthwhile to consider an example. Figure 5a shows the program + (* 3 4) 8 distributed across three PEs. The root node is at address 0 on PE #1, the two-node expression (* 3 4) is at addresses 0 and 1 on PE #3, and the remaining node is at address 0 on PE #2. The reduction of the program begins with the following message sent to PE #1:

request-id	REDUCE-REQ	(1 0)

PE #1 starts to apply the reduction procedure shown in Figure 3 to the node, whose first step is let T = Reduce(fn(E)). fn(E) is the node (2 0), so PE #1 sends the following message to PE #2:

request-id	REDUCE-REQ	(2 0)

PE #2 responds by applying the reduction procedure to node (2 0), and finds that since the function is the atom +, the node should be returned unaltered. So PE #2 sends a copy of node (2 0) back to PE #1 like so:

request-id	REDUCE-ACK	[(ATOM +) (3 0)]

When PE #1 receives this message, it creates a node in its own memory and puts the copy of (2 0) there. At this point, the PEs' memories appear as in Figure 5b (the function pointer of node (1 0) has not been changed from (2 0) to (1 1), as might be expected, but the pointer to (1 1) is kept in the temporary variable T of the reduction procedure executing on PE #1). The reduction procedure on PE #1 now resumes, and sees that the statement if fn(T) = + is satisfied, and proceeds to call for the reductions of the operands of nodes (1 0) and (1 1). Node (1 0)'s operand is an atom, but node (1 1)'s operand is the graph at (3 0), which is reduced by sending a reduction request to PE #3. PE #3 responds with a reduction acknowledgement containing the atom 12, and PE #1

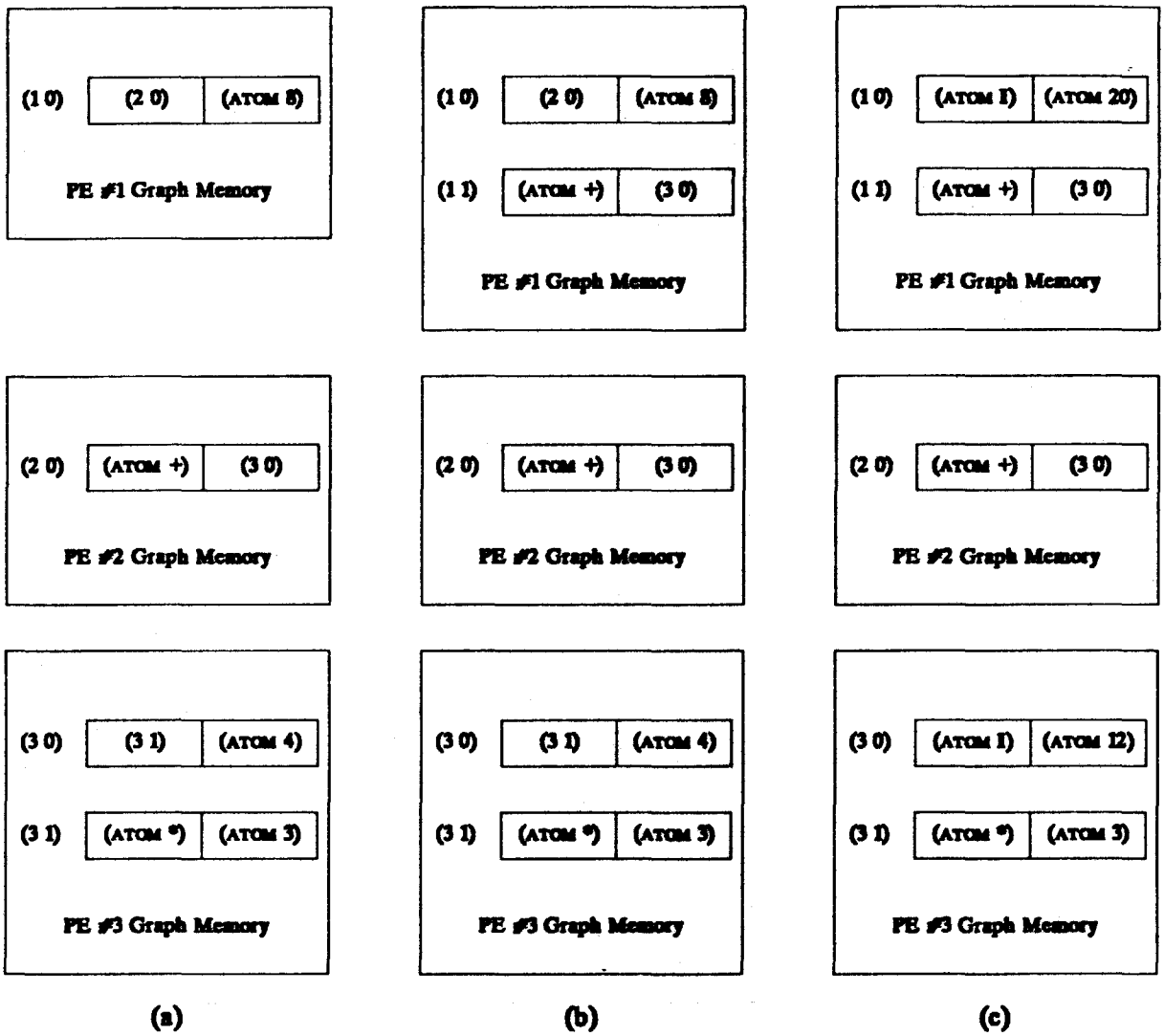

Figure 5. Steps in Parallel Reduction

reduces node (1 0) to I 20, sending a reduction acknowledgement containing the atom 20. Figure 5c shows the final appearence of the PEs' memories.

In the example above, the result of reducing node (2 0) was the three node subgraph + (* 3 4), but it was sufficient for PE #2 to return only the root node to PE #1 in the reduction acknowledgement, for the root node contained all information needed by PE #1. Consider now the reduction of S f g x, where each of the three nodes are on different PEs as shown in Figure 6a.

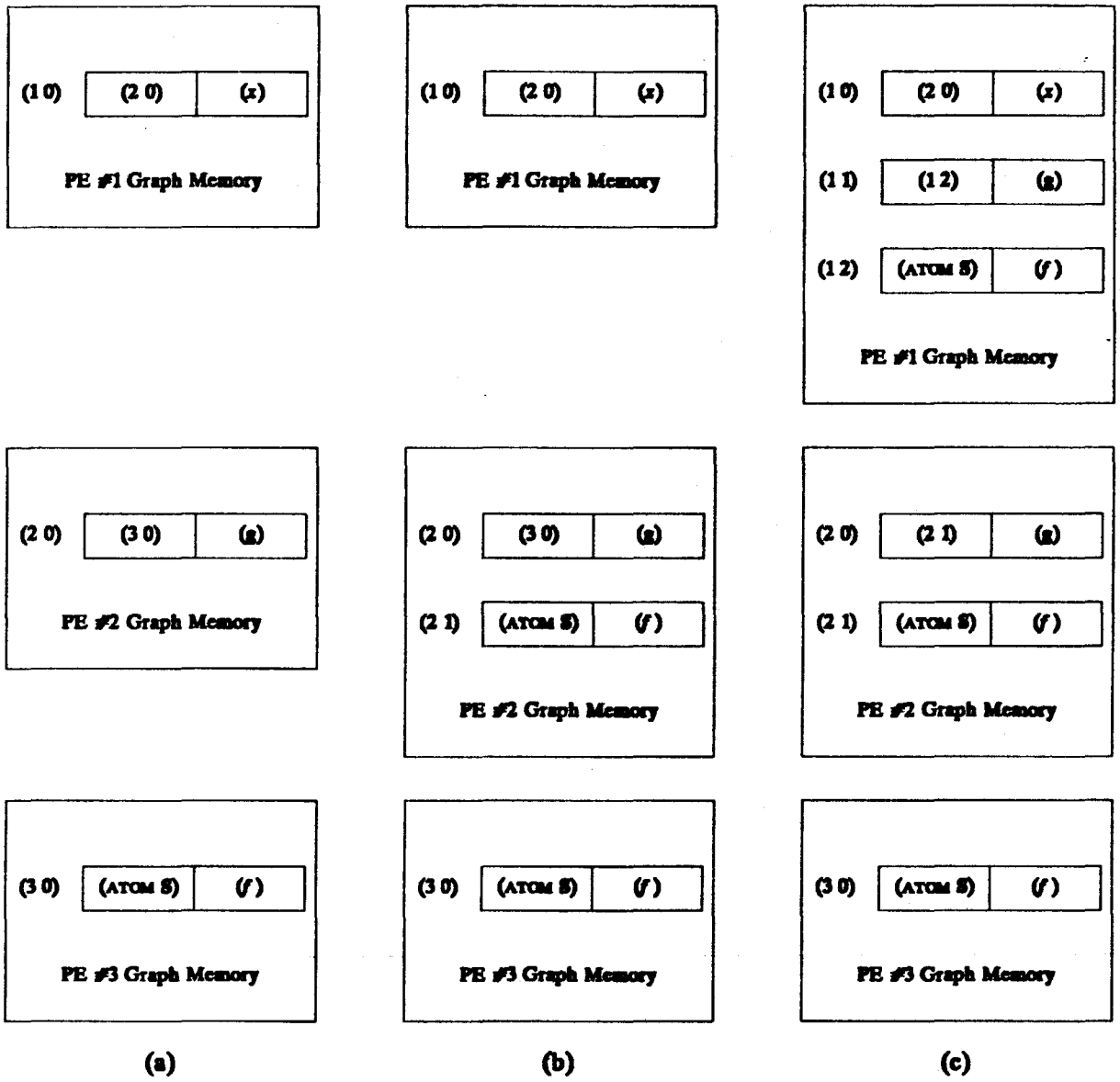

Figure 6. First Steps in Reducing S *f g x*

Reduction begins on PE #1, which sends a reduction request to PE #2, which in turn sends a reduction request to PE #3. PE #3, seeing that the function is the atom S, sends the following acknowledgement to PE #2:

request-id	REDUCE-ACK	[(ATOM S) (*f*)]

PE #2 copies this node into its own memory, and the memories are now as shown in Figure 6b. The reduction procedure on PE #2 sees that the statement **if** fn(*T*) = S succeeds, and so wants to return the two-node result (S *f*) *g*. If only the root node of a graph is returned, PE #2 sends this message to PE #1:

request-id	REDUCE-ACK	[(2 1) (*g*)]

When PE #1 receives this message, it will have two of the three nodes comprising the S expression, but to apply the reduction rule for S it needs all three, for it needs the pointers to *f*, *g*, and *x* (in fact, at this point it is missing the node that contains the S!). In this case, PE #2 must actually send two nodes back to PE #1, both of which will get copied into PE #1's local memory. This would be accomplished by a message like this:

request-id	REDUCE-ACK	{[(MSG 2) (*g*)] [(ATOM S)(*f*)]}

In this message, the pointer (MSG 2) points to the second node contained in the message; when PE #1 copies the contents of the message into its own graph memory, it will replace the (MSG 2) pointer with a pointer to the actual node created for the second node in the message. Figure 6c shows the state of the memories after PE #1 finishes this copying.

When a graph is to be returned from reduction, then, the rule for determining which nodes to include in the reduction acknowledgement is as follows. The root node of the graph to be returned is always included. In addition, any nodes pointed to by the root node that were returned from reductions requested during the reduction of the root node are also included. The nodes in this set are known to be reduced, making it safe to send them in a message, and are guaranteed to be accessible to the PE creating the acknowledgement.

3.3. The Need for Multi-Tasking

In the preceeding discussion, no mention was made of what a PE must do if it receives additional requests before dispensing with the one in progress. When a PE processes a reduction request, at several points it will send requests of its own and wait for the corresponding acknowledgements. It is unacceptable for the PE to suspend all activity when waiting for acknowledgements, because the requests it makes may cause other PEs to send additional requests back. If the PE ignores those requests, it will never receive the acknowledgements it is waiting for, and a deadlock occurs. Because the processing of a reduction request may be suspended while waiting for service from another machine, a PE must be capable of processing several reduction requests at once.

A single PE, therefore, can have several outstanding *reduction processes*, each one corresponding to a node currently undergoing reduction. Associated with each reduction process is a *process descriptor* (PD), which has enough information to allow the process to be suspended while waiting for acknowledgements and later resumed at the point of suspension. A process can be in one of two states: suspended or runnable. A suspended process is one that has sent requests but has not yet received all corresponding acknowledgements, and a runnable process is either one that has just been created or one that has received all acknowledgements. A runnable process will be selected by the PE for execution, at which point the reduction procedure will be resumed on that process until either one or more requests are issued, causing the process to become suspended, or until the algorithm finishes, causing a reduction acknowledgement to be sent. A suspended process becomes runnable again when it receives all acknowledgements for which it was waiting. Figure 7 illustrates the states a process can assume.

When a particular process's instance of the reduction procedure wants to make a request, it must do two things: it must send the appropriate request messages, and it must indicate in the process descriptor that it is waiting for acknowledgements. The PE may then pick another runnable process and work on it for a while. When acknowledgement messages are received, they must find

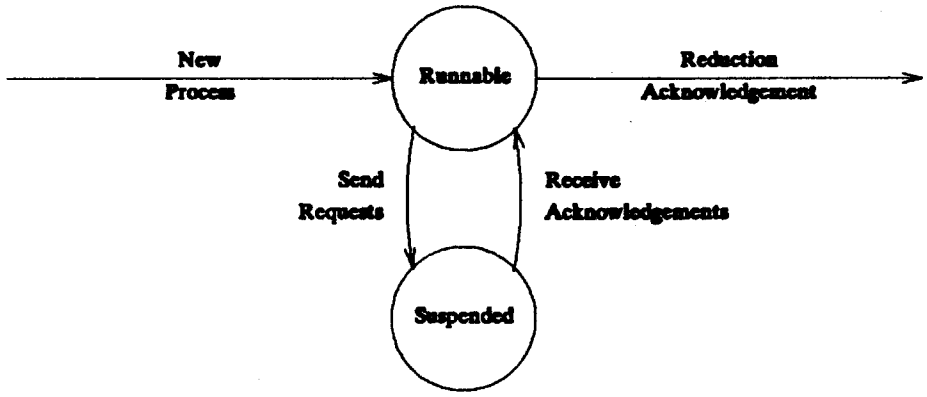

Figure 7. State Diagram for a Process.

their way to the correct process descriptor and return the process to the runnable state. To organize the flow of information, each process is assigned a unique process number, and several *request slots* are provided in each process descriptor. Recall that messages always contain a request indentifier. Whenever a process sends a request message, it includes a request identifier of the form (*PE process slot*), where *PE* is the number assigned to the requesting PE, *process* is the process number of the process making the request, and *slot* is the number of a request slot in that process descriptor. After sending the request message, the process stores the atom WAITING in request slot *slot* of the process descriptor; any process descriptor that has the atom WAITING in one or more of its request slots is considered suspended. Any acknowledgement arriving at the PE is stored in slot *slot* of process descriptor *process*, where *slot* and *process* are taken from the request identifier of the acknowledgement (remember that the request identifiers in acknowledgements are copies of the request identifiers contained in the correspondings requests). When a process receives the last acknowledgement it is waiting for, that acknowledgement replaces the last occurence of the atom WAITING in that process's request slots, and the process is considered runnable. When the reduction procedure is resumed on that process, it can find the results it requested in the request slots, for that is where the acknowledgement messages are stored. Note

that a process can make several requests at once by sending several request messages, each with a different value of *slot* in their request identifiers; this is how parallelism is achieved.

Another function of the process descriptor is to hold the request identifier of the reduction request message that created that process, for that information is necessary when preparing the reduction acknowledgement when the reduction procedure terminates. Because of subgraph sharing, it is possible for a second request to reduce a given node to arrive while the first request is still being processed. It is not safe for a second process to be started on that node, because the two processes will interfere with each other. Instead, only one process is allowed to reduce one node, but a process is allowed to send any number of reduction acknowledgements when it completes. To keep track of this, the process descriptor will contain a list of *notifiers*, one for each reduction request received for the node being reduced by that process. A notifier is merely the request identifier from a reduction request message; when the process completes, one reduction acknowledgement will be sent for every notifier in the notifier list, and the request-id fields of these acknowledgements will be created from the information in the notifiers.

Support for multiple processes also requires additional information to be stored with each node. Each node must have, in addition to the data fields proscribed by the sequential-semantic layer, a *status* field. A node can be in one of three states: unreduced, reducing, and reduced. When a node is created, either through the processing of a **CREATE-REQ** message or through the copying of nodes received in a **REDUCE-ACK** message, the status field is set to UNREDUCED. When the first reduction request to reduce that node arrives, a process descriptor is created and initialized, and the process descriptor number is stored in the status field of that node. Thus, the presence of a process descriptor number in the status field of a node indicates that the node is in the "reducing" state. If additional requests to reduce that node arrive while the node is in the "reducing" state, the status field of the node indicates which process descriptor should receive the additional notifier. When the process finally finishes reducing the node, the status field of the node is changed to REDUCED. Servicing any additional requests for the reduction of that node will

simply entail reading the node and preparing the appropriate reduction acknowledgement. As was noted earlier, once a node enters the REDUCED state it effectively becomes a constant.

3.4. Reference Count Garbage Collection

Because of the dynamic nature of reduction graphs, garbage collection is an important concern in the design of a graph reduction machine. It is doubly important in the parallel graph reduction machine because of the copying of nodes from one PE to another when reduction acknowledgements are sent. A useful propoerty of most reduction languages is that they can be defined in such a way so as never to create cyclic graphs. Turner's language, for example, can be made to either create cyclic graphs or not create cyclic graphs depending on the implementation of the Y combinator. In general, the avoidance of cyclic graphs entails a small amount of additional work during reduction, but there is a potentially great savings in the time required for garbage collection, for in the absence of cyclic graphs *reference count garbage collection* can be performed.

The mechanism necessary for reference count garbage collection is easily added to the system already described. Each node in graph memory is augmented with a *reference count* field, which is initialized to one when a node is created. When a reduction process creates an additional pointer to a node, it sends an Increment Reference Count Request (INCREF-REQ) message to that node's PE which contains a pointer to that node. The PE receiving an INCREF-REQ message responds by simply incrementing the reference count of that node. Similarly, when a node destroys a pointer to a node, it sends a Decrement Reference Count Request (DECREF-REQ) to the node's PE, which responds by decrementing the reference count of that node. If the reference count of a node is decremented to zero, DECREF-REQs are issued to the PEs of any nodes pointed to by that node, and the node is returned to the free list.

Since INCREF-REQs and DECREF-REQs can be issued for a given node by several PEs at once, precautions must be taken to make sure that these messages do not arrive out of order. If the reference count of a node is one, for example, and an INCREF-REQ followed by a DECREF-

REQ is issued for that node, if the messages arrive out of order the reference count will drop to zero before the INCREF-REQ message arrives, and the node will be garbage collected even though a pointer still exists to it. To prevent this occurence, it is noted that any time a process creates a new pointer to a node, it must already have a pointer to that node. Even if the INCREF-REQ message never arrives, the node will not be garbage collected as long as that process retains the original pointer it had to that node. Thus, the process issuing an INCREF-REQ can guarantee the correctness of the node's reference count by suspending its activity until it is sure the INCREF-REQ message has been received.

The obvious way to accomplish this synchronization is to have the issuer of an INCREF-REQ enter the suspended state until it receives an Increment Reference Count Acknowledgement (INCREF-ACK) message, which the receiver of an INCREF-REQ sends after incrementing the reference count. In this way, the process cannot accidentally issue a DECREF-REQ for that node until the INCREF-REQ has definitely been processed, and so the reference count will never be an underestimate. There is no need to have a Decrement Reference Count Acknowledgement, for there is no danger in overstating the reference count temporarily. The issuer of a DECREF-REQ can proceed immediately after issuing the message.

3.5. Summary

The essential design of the parallel-semantic layer is complete, and is now summarized. The overall appearence of the parallel reduction machine is as illustrated in Figure 8, with a number of identical Processing Elements connected by a communications network. The communications network is of arbitrary topology, but must support the reliable transmission of messages from one PE to another.

The flow of information within each PE is depicted in Figure 9. There are two types of data stored in the memory of a PE: nodes and process descriptors. Nodes, which are the objects comprising the program graph, are stored in Graph Memory (GM), and contain, in addition to the fields prescribed by the sequential semantic layer of the particular machine, a status field and a

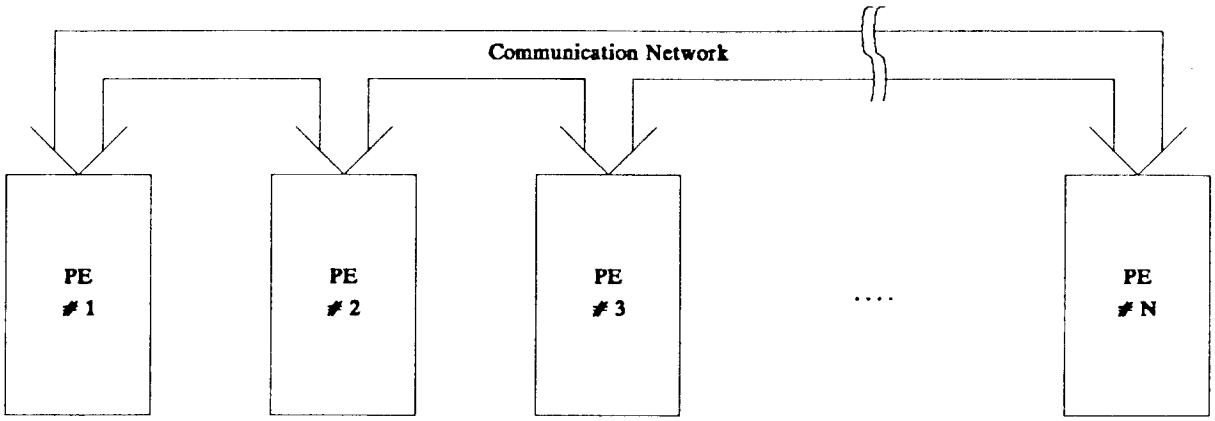

Figure 8. Organization of the parallel reduction machine.

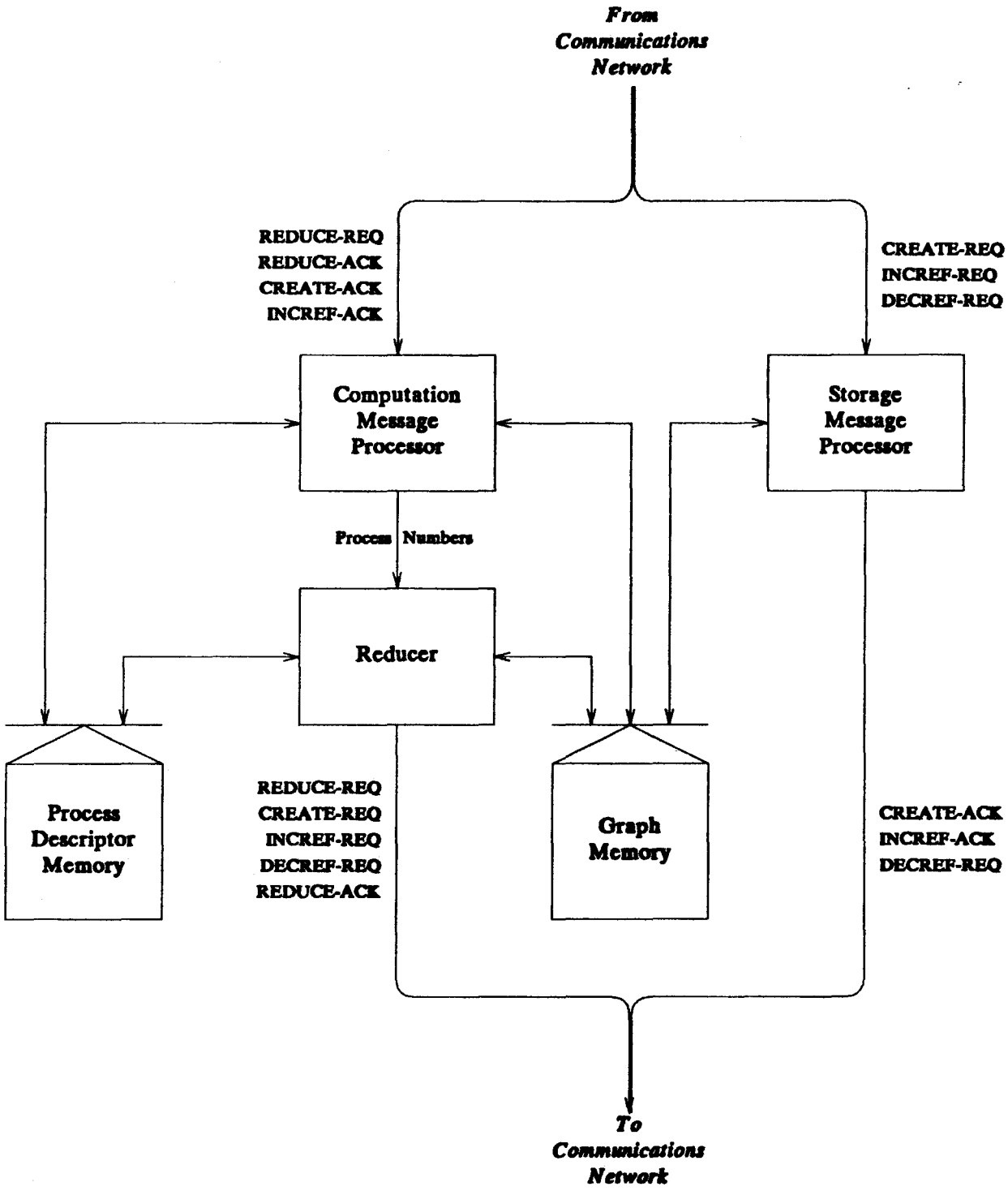

Figure 9. Summary of PE function.

reference count field. Process Descriptors keep track of the tasks in progress within a PE; there is one active process descriptor for every node in Graph Memory that is in the "reducing" state. The process decriptor contains a list of notifiers, one for every **REDUCE-ACK** message that will be sent upon the completion of that process, a set of request slots used both to indicate the status of the process and to hold acknowledgements after they are received, and enough state information to resume the reduction procedure after it becomes suspended through the issuing of requests.

There are logically three distinct computational entities within each PE. The Storage Message Processor handles the processing of incoming **CREATE-REQ**, **INCREF-REQ**, and **DECREF-REQ** messages. In processing these messages, the SMP requires access to the Graph Memory, and can issue **CREATE-ACK**, **INCREF-ACK**, and **DECREF-REQ** messages. The latter arise when nodes are garbage collected, and since **DECREF-REQ** messages have no corresponding acknowledgement, the SMP does not need to suspend its operations at any time.

The remaining messages, **REDUCE-REQ**, **REDUCE-ACK**, **CREATE-ACK**, and **INCREF-ACK**, are handled by the Computation Message Processor. The latter three messages cause the writing of request slots of process descriptors in the suspended state. The **REDUCE-REQ** message causes the status field of the node indicated in the message to be examined. If the status is "unreduced", an unused process descriptor is obtained and its number stored in the status field of the node to be reduced. The state information in the new process descriptor is initialized so that it points to the beginning of the reduction procedure with the node as argument. Finally, the notifier list of the process descriptor is initialized with the request-id of the **REDUCE-REQ** message. This results in a new runnable process. If the status field of the node in the **REDUCE-REQ** message was already the number of a process descriptor, the request-id is added to the notifier list of that process descriptor. If the status field of the node was "reduced", the operations performed are exactly the same as if the status field was "unreduced", except that the state information in the new process descriptor is initialized to begin at the end of the reduction procedure: at the beginning of the section that sends the reduction acknowledgements and removes the PD.

Processes move from the suspended state to the runnable state only upon the receipt of a message, so the Computation Message Processor is capable of providing a stream of process descriptor numbers of processes that have moved from the suspended state to the runnable state. A PD number is added to this stream in two cases: if a **REDUCE-ACK**, **CREATE-ACK**, or **INCREF-ACK** is received that overwrites the last occurence of the word WAITING in the request slots, or if a **REDUCE-REQ** is received that creates a new process descriptor. The stream of runnable process numbers is passed to the Reducer, which actually performs the reduction algorithm. When the Reducer resumes a process, it works on that process either until it issues one or more requests, whereupon the process enters the suspended state by virtue of the word WAITING in one or more of its request slots, or until it completes, causing one **REDUCE-ACK** message to be sent for every notifier in the notifier list, after which the PD is returned to the list of free PDs.

As Figure 9 illustrates, while the Storage Message Processor, the Computation Message Processor, and the Reducer are functionally independent, they share two data structures, Graph Memory and Process Descriptor Memory. Contention problems are avoided, however, because their use of these structures is disjoint. The Storage Message Processor, for example, is the only unit that uses the free node list or the reference count fields of the nodes. The data fields of nodes are only used by the reducer after the SMP creates them. The status fields of the nodes are used only by the Computation Message Processor. Similar divisions of usage occur between the Computation Message Processor's and the Reducer's use of process descriptors.

4. Optional Features

In the previous section, the minimum function of the parallel-semantic layer was described. There are many extensions to this basic system possible that will improve the performance.

4.1. Program Loading and I/O

While the capability for initial loading of program graphs is hardly an optional feature, it is of less importance than the actual execution of program graphs. Happily, providing this feature

requires no additional mechanism in the parallel-semantic layer.

Generally, the overall machine structure as shown in Figure 8 will also include a special Front-End Processor attached to the communication network, which can be addressed as if it were a regular PE. This special unit is in charge of all interaction with the user, including I/O and the loading of programs. The Front-End Processor loads a program into the machine by issuing CREATE-REQ messages, and begins its execution by issuing a REDUCE-REQ message. When it receives a REDUCE-ACK message, that message will contain the result to be printed for the user. The way in which I/O is handled is up to the base language, but it will usually be in the form of streams, whose operators interact with the Front-End Processor through REDUCE-REQ/REDUCE-ACK message pairs.

4.2. Time Sharing

Any parallel reduction machine built upon the principles set forth here is capable of performing time sharing, for each PE already has the facility for working on several tasks at once. To achieve the simultaneous execution of two unrelated programs, the Front-End Processor simply loads both programs onto the PEs and sends a REDUCE-REQ for each of the two root nodes. The two graphs will each get a more or less equal share of the PEs combined time, for the PEs have no way of knowing that the various nodes being reduced are part of unrelated graphs.

It is also relatively easy to provide this time sharing system with a crude priority mechanism. A *priority* field is added to the process descriptor and to the REDUCE-REQ message. When a PE receives a REDUCE-REQ message, it compares the priority field of the request with the priority field of the process descriptor that will process the request, and stores the greater back into the process descriptor. Whenever a process issues a REDUCE-REQ, it will take the priority field of the request from the priority field of the process's process descriptor. Thus, the priority is propagated to the descendant nodes of the original node reduced.

The priority comes into play when the PE chooses a runnable process for execution by the Reducer. When the PE selects a process from the stream of runnable processes, it always selects

the runnable process with the highest priority, thus assuring that higher priority processes are serviced first.

4.3. Reduced Idle Time Through Eager Evaluation

Up to now, the parallel reduction machine has been completely demand driven; a REDUCE-REQ is never issued for a node until some reduction process definitely needs the result. Some researchers have suggested that additional parallelism can be extracted from a program by reducing some nodes *before* they are needed, so that if their values *are* eventually needed they will have already been computed. This scheme can make use of any idle time that might otherwise exist in a system with a large number of PEs, but it is important that valuable time is not wasted reducing nodes whose values will never be needed.

The priority mechanism described in the previous section provides an elegant way of controlling eager evaluation. By assigning a higher priority to the REDUCE-REQ issued for the root node of the graph than for the REDUCE-REQs issued for other nodes of the graph, each PE will always work on nodes definitely needed for the computation of the final result if it has a choice. An additional problem introduced by eager evaluation is that nodes requiring garbage collection can have reduction processes active on them. The garbage collection mechanism must therefore collect processes as well as nodes.

4.4. Increased Throughput Through Multiple Reducers

Unlike many proposed parallel machines, the parallel reduction machine described here does not make use of shared memory at all. One consequence is that each PE must multi-task: a PE can have several runnable processes existing at once. The throughtput of a PE can be improved if the PE in Figure 9 is augmented to include several Reducers. These Reducers will have to share Graph Memory and Process Descriptor Memory, but to the degree that the Reducers can interleave memory cycles there will be more processes disposed of in any time interval. This system represents a very general type of multiprocessor where shared memory is used up to the point

where additional processors sharing the memory is no longer benificial, after which groups of processor/memory units are interconnected with a communications network.

4.5. Load Balancing

It was pointed out in Section 3 that because a node is always reduced by the PE in whose memory it resides, a policy for allocating new nodes to PEs is equivalent to a policy for distributing the workload. The distribution of workload is mainly an issue in the topological layer, for it is only the communications network that can "see" all the PEs and thereby have an indication of which PEs are lightly loaded and which are heavily loaded.

Load balancing is accomodated by changing the **CREATE-REQ** message so that is not directed at any particular PE. The communications network, upon obtaining a **CREATE-REQ** message, can route it to the PE that is the least loaded. Since the **CREATE-ACK** message contains a complete pointer, including PE number, no special support is required from the issuer of the **CREATE-REQ** message.

In general, two different types of **CREATE-REQ** messages will have to be provided: one for nodes that are to be allocated on a PE to be determined by the load balancer, and one for nodes where the PE is specified by the PE sending the request. An instance where the latter is required is when a PE must allocate a node in its own memory to copy a node received in a **REDUCE-ACK** message.

5. Comparison With Existing Proposals

In the introduction it was stated that the parallel-semantic layer as described here is essentially the same as the parallel-semantic layers of other parallel graph reduction machines that have been proposed, except that here it presented more systematically and thoroughly. The other proposals will now be compared to the system here.

5.1. Keller, Lindstrom, and Patil

Perhaps the most detailed description of a parallel graph reduction machine is given by Keller *et. al.*[4], and while their machine differs from the scheme here in minor ways, it fits the abstract architecture quite well.

The FGL language that their machine uses reflects their machine's load balancing policy: all nodes belonging to a single user procedure are allocated on the same PE. A code block in their system is a type of constant, and the *Invoke* operator executes by using the information in a code block to create a collection of nodes (all on one PE). Some of the nodes created by the *Invoke* will include information computed at run time in addition to the compile time information taken from the code block. This and many other issues discussed in the Keller paper actually pertain to the sequential-semantic layer rather than the parallel-semantic layer.

Other aspects of their machine are quite familiar. Their machine's "demand-list" and "result-list" are similar to the process descriptors of the abstract machine. In Keller's machine, however, notifiers are associated with each node, rather than with each process (task, in their terminology), and are preassigned in most cases. This is possible because they only attempt to exploit subgraph sharing within a user function definition, and so most notifiers are available at compile time. There is really no advantage in precomputing the notifiers, and leaving space in each node for a notifier is wasteful of space since only a fraction of the nodes at any time will be in the "reducing" state. Including the notifiers in the nodes also forces their system to use "forward chaining" to handle multiple global notifiers. While this technique has the advantage that the space for notifiers is not of variable size, it increases the amount of communication necessary, for in addition to the actual notification messages, their system requires additional messages to set up the forward chaining. No real memory space is saved, for the same number of notifiers must be stored in either system.

Keller's paper gives no detailed discussion of what messages are passed in his system, so no comparison of communication semantics is possible.

5.2. Darlington and Reeve

The ALICE multi-processor[2] is very interesting because at first glance it appears to be greatly different from the machine described here. As in Keller's machine, nodes of the graph contain notifiers in addition to the information contained in nodes of the abstract machine. In ALICE, however, the nodes are all put in a shared memory to which each of the PEs has access. Darlington recognizes that shared memory limits the number of PEs that can successfully be employed in this way, so he proposes connecting groups of memory/PE units with a communication network.

This, of course, is the scheme discussed in Section 4.4, wherein multiple Reducers are provided in each PE. In Section 4.4, the Reducers had to share common resources, including the memory itself, the Computation Message Processor, and the Storage Message Processor. These common services are also described in Darlington's paper; there, he visualizes the stream of runnable processes and the free node list as "constantly circulating slotted communications rings".

Darlington also points out that when PE groups are connected by a communication network, the network serves to "map the local memories onto the global address space of the system". This, of course, is reflected in the (*PE address*) form that pointers take in the system here. Darlington goes on to say that the communication network is used to share processable nodes and free space among the building blocks. While the latter is certainly true -- this is the load balancing function described in Section 4.5 -- the former contradicts his earlier statement, for the mapping of local memories into the global address space precludes the migration of nodes from one memory unit to another. Such migration is possible if forwarding addresses are left behind or if the communication network serves to translate "virtual addresses" appearing in nodes to "physical addresses" consisting of PE/address pairs, but the former entails communication overhead to perform the forwarding, and the latter turns the communication network into a huge bottleneck through which all memory references must pass. In particular, any benefit that might be obtained from grouping related nodes into the same memory segment is lost.

Abandoning the extremely inefficient feature of allowing the migration of unreduced or partially reduced nodes, then, brings ALICE on par with the abstract architecture presented here. The main difference is that in Darlington's paper, a shared memory system is the starting point from which a hybrid shared memory/message passing system is developed. Here, a message passing model is the starting point from which the hybrid is easily derived (in Section 4.4). Darlington's paper provides no details of what communication takes place in the hybrid version of ALICE.

The last major difference between the ALICE machine and the abstract machine presented here is that ALICE supports the accessing of nodes, for both reading and writing, that have not been reduced. This is in opposition to the principles set forth in Section 2, and reflects the fact that ALICE is capable of supporting base languages other than strictly constant applicative form languages. Whether this fact presents any special problems is a topic for future research.

5.3. Sleep and Burton

Sleep and Burton give a very brief description of a parallel reduction machine[6] that uses a form of combinator code as a base language. Most of their paper deals with the properties of base languages and with the details of their communication network, and so there is little to compare with the system here. What little they do discuss of the parallel-semantic layer is quite familiar; in particular, they describe the use of the status field of nodes.

6. Conclusions

Many parallel graph reduction machines have been proposed, but little has been done to establish the operating principles common to all such machines. The work here attempts to systemize the design of parallel reduction machines by dividing the topic into three layers: the sequential-semantic layer, the parallel-semantic layer, and the topological layer. The parallel-semantic layer, it turns out, embodies the fundamental essence of parallel reduction in the abstract; as such, the parallel-semantic layers of all parallel reduction machines will be similar, if

not identical.

The parallel-sematic layer has been described here to a sufficient level of detail that only the language and communication network would need to be designed to create a complete machine. In particular, the aspects covered in the parallel-semantic layer include the overall structure of the machine, the semantics of the messages that travel the communications network, the data structures maintained by the processing element, and the algorithms necessary to manage these data structures. The correctness of the scheme presented here was demonstrated by an emulation program written for a Symbolics 3600 Lisp Machine.

While other groups have proposed parallel reduction machines, no proposal has described the parallel-semantic layer of a machine to the degree of detail as with the abstract machine presented here. To the degree that these other machines are described, their parallel-semantic layers are consistent with the model here. But the architecture presented here is more than a hypothetical machine; by providing an abstract model for parallel graph reduction, it is hoped that insight into the parallel reduction process itself can be gained. Such insight will undoubtedly prove useful in the design and construction of actual high-performance graph reduction machines.

REFERENCES

(1) T.J.W. Clarke, P.J.S. Gladstone, C.D. Maclean, and A.C. Norman, "SKIM – The S, K, I Reduction Machine", *Proc. 1980 LISP Conference*, August 1980.

(2) J. Darlington and M. Reeve, "ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages", *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, 1981*, pp. 65-76.

(3) T. Johnsson, "The G-Machine: An Abstract Machine for Graph Reduction", Programming Methodology Group, Department of Computer Science, Chalmers University of Techonology, S-412 96 Goteborg, Sweden.

(4) R. M. Keller, G. Lindstrom, and S. Patil, "An Architecture for a Loosely-coupled Parallel Processor", Tech. Report UUCS-78-105, University of Utah, October 1978.

(5) G. A. Mago, "A Cellular Computer Architecture for Functional Programming", *COMPCON Spring 80*, February, 1980, pp. 179-187.

(6) M. R. Sleep and F. W. Burton, "Towards a Zero Assignment Parallel Processor", *Proceedings of the Second International Conference on Distributed Computing Systems*, 1980, pp. 80-84.

(7) D. A. Turner, "A New Implementation Technique for Applicative Languages", *Software – Practice and Experience*, 9(1979), pp. 31-49.