Massachusetts Institute of Technology
Artificial Intelligence Laboratory

A.I. Memo No. 626 May, 1981

Thinking About Lots Of Things At Once Without Getting Confused
Parallellism in Act 1

Henry Lieberman

Abstract

As advances in computer architecture and changing economics make feasible machines with large-scale parallelism, Artificial Intelligence will require new ways of thinking about computation that can exploit parallelism effectively. We present the *actor* model of computation as being appropriate for parallel systems, since it organizes knowledge as active objects acting independently, and communicating by *message passing*. We describe the parallel constructs in our experimental actor interpreter *Act 1*. *Futures* create concurrency, by dynamically allocating processing resources much as Lisp dynamically allocates passive storage. *Serializers* restrict concurrency by constraining the order in which events take place, and have changeable local state. Using the actor model allows parallelism and synchronization to be implemented transparently, so that parallel or synchronized resources can be used as easily as their serial counterparts.

Keywords: Actors, object-oriented programming languages, message passing, artificial intelligence, knowledge representation, data abstraction, parallelism, futures, serializers

Acknowledgements: This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Office of Naval Research under Office of Naval Research contract N00014-75-C-0522, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505.

Thinking About Lots Of Things At Once Without Getting Confused
Parallellism in Act 1

Henry Lieberman

Artificial Intelligence Laboratory
and Laboratory for Computer Science
Massachusetts Institute of Technology

Section 1. Introduction

This paper will try to accomplish several goals (in parallel):

We will argue that the *actor* model is an appropriate way to think about parallel computation, and has distinct advantages over other alternatives that have been proposed for constructing parallel systems. We propose that systems be organized completely of active objects called *actors*, who may communicate with each other only by *sending messages*. Since many actors may be actively sending or receiving messages at the same time, actors are inherently well suited to modelling parallel systems.

We will present some specific actors which we feel should be included in the programmer's tool kit for writing parallel programs. We will show examples illustrating the use of these primitives. *Futures* are actors which represent the values computed by parallel processes. They can be created dynamically and disappear when they are no longer needed. Other actors may use the value of a future without concern for the fact that it was computed in parallel. Synchronization is provided by *serializers*, which protect actors with internal state from timing errors caused by interacting processes.

We will show how these primitives have been implemented in our experimental actor system *Act 1*. Act 1 has been implemented on a serial PDP-10, but it simulates the kind of parallelism that would occur on a real multiprocessor machine. Act 1 provides a research tool for playing with applications programs and implementation techniques. Discussion of the implementation will give a more concrete picture of the mechanisms involved and will also show what would be needed for an implementation on a real network of parallel processors.

A more detailed discussion of the philosophy and implementation of Act 1 can be found in the companion paper [2].

1.1 Traditional techniques for parallelism have been inadeqate

Any language which allows parallelism must provide some way of *creating* and *destroying* parallel activities, and some means of *communicating* between them. Most of the traditional techniques for parallelism which have grown out of work in operating systems and simulation share these characteristics:

Usually, only a *fixed* number of parallel processes can be created, and processes cannot be created by programs as they are running. Processes usually must be *explicitly* destroyed when no longer needed. Communication between processes takes the form of assignment to memory cells shared between processes.

We propose that parallel processes be represented by actors called *futures* [7]. Futures can be created *dynamically* and disappear by *garbage collection* rather than explicit deletion when they're no longer needed. Communication between processes takes place using shared actors called *serializers*, which protect their internal state against timing errors.

Section 2. Creating parallelism with futures

2.1 Dynamic allocation of processes parallels dynamic allocation of storage

Act 1 solves the problem of allocating processes by extending Lisp's solution to the problem of allocating storage.

Languages like Fortran and machine languages take the position that storage is only allocated statically, *in advance* of the time when a program runs. Once allocation is made, it cannot be released, even though it may no longer be needed.

The inflexibility of static storage allocation led Lisp to a different view. With Lisp's CONS, storage magically appears whenever you need it, and the garbage collector magically recovers storage when it's no longer accessible. Even though the computer

only has a finite number of storage locations in reality, the user can *pretend* that memory is practically infinite.

We propose that parallel processes be allocated dynamically rather than statically. We will introduce actors called *futures* which represent parallel computations. There is a primitive which magically creates them whenever you need them. When they're no longer necessary, they get *garbage collected*, when they become inaccessible. The number of processes need not be bounded in advance, and if there are too many processes for the number of real physical processors you have on your computer system, they are automatically *time shared*. Thus the user can *pretend* that processor resources are practically infinite.

Machine languages communicated between procedures by using the *registers* of the machine, memory shared by the communicating procedures. Fortran procedures sometimes communicate through assignment to shared variables. This causes problems because a memory location shared between several users can be inadvertently smashed by one user, violating assumptions made by other users about the memory's contents.

Lisp uses the control structure of *function calls and returns*, procedures communicating by passing arguments and returning values. A process creating a future actor communicates with the future process by passing arguments, and the future process communicates with its creator by returning a value. We discourage explicit deletion of processes for the same reason we discourage explicit deletion of storage. If two users are both expecting results compted by a single process, then if one user is allowed to destory the process unexpectedly, it could wreak havoc for the other user.

2.2 Futures are actors representing the results of parallel computations

Act 1 provides a primitive HURRY which takes something to be computed, and which creates a *future* actor which represents the value of that computation. A future is like a *promise* or *I.O.U.* to deliver the value when it is needed.

HURRY always returns a future actor immediately, regardless of how long the computation will take. HURRY creates a *parallel process* to compute the value, which may still be running after the future is returned. The user may pass the future around, or perform other computations, and these actions will be overlapped with the computation of the future's value.

From the viewpoint of a user program, the future actor is indistinguishable from the value itself. The user of the future need not care that the value was computed by a parallel process. The only difference is that, on a parallel machine, it can be computed more quickly. The behaviour of a future actor is arranged so that if computation of the value has been completed, the future will act identically to the value. If the future is still running, it will delay the sender long enough for the computation to run to completion.

Futures are especially useful when a problem can be broken up into *independent subgoals*. If the main problem requires several subgoals for its solution, and each goal can be pursued without waiting for others to finish, the solution to the problem can be found much faster by allocating futures to compute each subgoal.

```
Define SOLVE-INDEPENDENT-SUBGOALS:
 Call COMBINE-RESULTS-OF-SUBGOALS on the result of
  Creating a FUTURE to solve INDEPENDENT-SUBGOAL-1, and
  A FUTURE trying to solve INDEPENDENT-SUBGOAL-2.
```

The futures for each subgoal will return immediately. Since the computation of each subgoal will presumably take a long time, the computation of the subgoals will overlap with each other and with the procedure combining their results.

How do we know when the value that the future returns will really be needed by someone else? In the actor model, that's easy - the only way another actor may do anything with the value is to send it a message. So, whenever any other actor sends a message to a future, we require that the future *finish* computing before a reply can be sent. If the value is requested before the future is ready, the caller must *wait* for the future to finish before getting the answer. When the future does finish, it stashes the answer away inside itself, and thereafter behaves identically to the answer, passing all incoming messages through to the answer.

A nice aspect of using futures is that the future construct packages up both the creation of parallel processes and synchronization of results computed by the processes. The parallelism and corresponding synchronization are always correctly *matched* with one another. This promotes more structured parallel programs than formalisms in which you describe the creation of parallel processes and their synchronization independently, opening the possibility that there may be some mismatch between parallelism and synchronization.

The property of being able to transparently substitute for any actor whatsoever a

future computing that actor is crucially dependent on the fact that in Act 1, everything is an object and all communication happens by message passing. This can't be done in less radical languages like Clu and Simula, which do have some provision for objects and message passing, but which don't treat everything that way. A future whose value is a built-in data type like numbers or vectors could not be used in a place where an ordinary number or vector would appear.

Futures can be used in Act 1 in conjunction with Lisp-like list structure, to represent *generator* processes. Suppose we have a procedure that produces a sequence of possibilities, and another procedure that consumes them, and we would like to overlap the production of new possibilities with testing of the ones already present.

We can represent this by having the producer come up with a list of possibilities, and the consumer may pick these off one by one and test them. This would work fine if there were a finite number of possibilities, and if the consumer is willing to wait until all possibilities are present before trying them out. But with futures, we can simply change the producer to create futures for the list of possibilities, creating a list which is *growing* in time, while the possibilities are being consumed.

```
Define PRODUCER:
 If there are no more POSSIBILITIES,
  return the EMPTY-LIST.
 If some possibilities remain,
  Create a list whose FIRST is:
   a FUTURE computing the FIRST-POSSIBILITY,
  and whose REST is:
   a FUTURE computing the rest of the possibilities by calling PRODUCER.

Define CONSUMER, consuming a list of POSSIBILITIES:
 Test the FIRST possibility on the POSSIBILITIES list,
 Then call the CONSUMER on the REST of the POSSIBILITIES list.
```

The consumer can use the list of possibilities as an ordinary list, as if the producer had produced the entire list of possibilities in advance. The consumer need not know the possibilities are being computed in parallel. We could get even more parallelism out of this by having the consumer create futures, testing all the possibilities in parallel.

On a machine with sufficiently many processors, the most radical way to introduce parallelism would be to change the interpreter to *evaluate arguments in parallel.*

Making this *eager beaver* evaluator would require just a simple change to create a future for the evaluation of each argument to a function. We haven't done this in our current implementation of Act 1 because processes are still a bit too expensive on our serial machine to make it the default to create them so frequently. Instead, we require that the parallelism be explicitly indicated.

Futures are more powerful than the alternative *data flow* model proposed by Dennis [33]. In the dataflow model, arguments to a function are computed in parallel, and a function is applied only when all the arguments have finished returning values. Let's say we're trying to compute the SUM of FACTORIAL of 10, FACTORIAL of 20 and FACTORIAL of 30, each of which is time consuming. In dataflow, the computations of the factorial function can all be done in parallel, but SUM can't start computing until *all* the factorial computations finish.

If futures are created for the arguments to a function, as can be done in Act 1, the evaluation of arguments returns immediately with future actors. The function is applied, with the future actors as arguments, without waiting for any of them to run to completion. It is only when the value of a particular argument is actually needed that the computation must wait for the future to finish.

In our sum-of-factorials example, imagine that the first two factorials have finished but the third has not yet returned. Act 1 allows SUM to begin adding the results of FACTORIAL of 10 and FACTORIAL of 20 as soon as they both return, in parallel with the computation of FACTORIAL of 30. So Act 1 allows overlap of application of a function with computation of arguments in a way that dataflow doesn't.

2.3 Explicit deletion of processes considered harmful

Notice that there are some operations on futures that we *don't* provide, although they sometimes appear in other parallel formalisms. There's no way to ask a future whether he has finished yet. Such a message would violate the property of futures that any incoming message forces him to finish, and that wrapping futures around values is completely transparent. It would encourage writing time and speed-dependent programs.

There's no way to *stop* a future before the future has returned a value. Continuing the analogy with list storage, we believe that explicitly stopping or destroying processes is bad for the the same reason that deleting pointers to lists would be bad

in Lisp. Deleting a process that somebody else has a pointer to is just as harmful as deleting a list pointer that is shared by somebody else. It's much safer to let deletion happen by garbage collection, where the system can automatically delete an object when it can be verified that it's no longer needed by anybody.

We don't exclude the possibility of providing such lower level operations as examining and destroying processes for the purposes of debugging and implementation, but they should not be routinely used in user programs. A safer way of being able to make a decision such as which of two processes finished first is to use Act 1's serializer primitives which assure proper synchronization.

2.4 How does Act 1 implement futures?

Futures are the actors by which Act 1 introduces parallel processes into a computation. What is a process, anyway? There are many conflicting definitions of the term *process* in the computer science literature, all somehow intended to capture the notion of parallel computation. We introduce a technical definition of *process* which is independent of physical processors, address spaces, or other implementation details.

In the actor model, a *process* consists of a *message* actor, a *target* actor, who is receiving the message, and a *reply continuation*, who expects to receive the result of the computation. The operation of an Act 1 interpreter is to repeatedly invoke the *script*, a program associated with the target, and produce a new message, target, and reply. (Details of Act 1's operation are elaborated in [2].) In an actor system, at any moment many targets may be receiving messages at the same time, hence many processes may be operating in parallel.

In a real parallel processor machine, creating a new process would mean finding a physical processor willing to perform the computation. In our single processor implementation, another process object is added to the list of processes which share the physical processor.

When the future receives a message intended for its value, there are two cases, depending on whether the computation is still running or not. The future needs a *flag* to distinguish these cases. If it is running, the sender must *wait* until the computation finishes. When the future finishes, it sets the flag, and remembers the answer to the computation in a memory cell. Any messages sent to the future are then relayed to the stored answer.

```
Define HURRY, creating a FUTURE evaluating a FORM:
 Create a CELL which initially says that the future is RUNNING.
 Create a CELL for the ANSWER to which the form will eventually evaluate.
 Create a PROCESS, and start it to work computing the value of the FORM,
  Then send the VALUE of the FORM to the continuation FINISH-FUTURE.

If I'm a FUTURE, and I get a request:
 I check my RUNNING cell.
  If it's TRUE, the sender waits until it becomes FALSE.
  Then, I pass the message along to my ANSWER cell.

Define FINISH-FUTURE, receiving a VALUE for the FORM:
 I receive the ANSWER to the computation started by HURRY.
 Update the RUNNING cell to FALSE, indicating the future has finished.
 Put the VALUE in the FUTURE's ANSWER cell.
 Cause the process to commit SUICIDE, since it is no longer needed.
```

2.5 Aren't futures going to be terribly inefficient?

Advocates of more conservative approaches to parallelism might criticize our proposals on the grounds that futures are much too *inefficient* to implement in practice. Allocating processes dynamically and garbage collecting them does have a cost over simpler schemes, at least on machines as they are presently designed. Again, we make the analogy with list structure, where experience has shown that the benefits of dynamic storage allocation are well worth the cost of garbage collection.

Trends in hardware design are moving towards designs for computers that have many, small processors rather than a single, large one. We think the challenge in designing languages for the machines of the near future will come in trying to make effective use of massive parallelism, rather than in being excessively clever to conserve processor resources.

One source of wasted processor time comes from processes that are still running using up processor time even though they are no longer needed, before they are reclaimed by the garbage collector. This is analogous to the fact that in Lisp, storage is sometimes still unavailable between the time that it becomes inaccessible and the time it is reclaimed by the garbage collector. This is a cost that can be minimized by a smart *incremental* real time garbage collector for processes like that of Henry Baker, or the one we are proposing for the Lisp Machine [25].

We intend that processes be cheap and easy to create, a basic operation of the system just like message passing. We have taken care to see that a process doesn't have a tremendous amount of state information or machinery associated with it. The state of a process is *completely* described by the *target* actor, the *message* actor, and the *continuation* actors. If this information is saved, the process can be safely interrupted or may wait for some condition to happen while other processes are running, and be resumed later.

This makes processes more *mobile*. It is easy to move a process from one processor to another, or time share many processes on a single processor. Multiple processor systems may need to do dynamic load balancing or time sharing when there are more conceptual processes than physical processors.

Section 3. Serializers are actors which restrict parallelism

3.1 Why is parallel programming different from all other programming?

There's a whole class of errors which arise in parallel programming which don't show up in sequential programming: *timing errors*. Timing errors occur when one process looks at the state of an actor, takes some action on the implicit assumption that the state remains unchanged, and meanwhile, another process modifies the state, invalidating the data.

Timing errors are possible when parallel programs use *changeable* actors incorrectly. An changeable actor has an internal state which he may modify, causing a *state change* (or *side effect*). Changeable actors have the ability to change who their acquaintances are. An actor is changeable if the same message can be sent to him on two different occasions and result in different answers.

Suppose we would like to implement a *global data base* for use in a parallel program, like the *blackboard* of the speech understanding system Hearsay [30]. In general, Act 1 discourages global data bases, since the knowledge should be distributed, each actor knowing only what's relevant to him. Global data bases are sometimes used when programs may gather unexpected information. If the structure of the information cannot be known in advance and there's no way of predicting which actors might be interested in knowing the information, a central place for new information can be helpful.

The blackboard actor will receive a message ASSERT which adds new information, and a CONTENTS message which reads out the contents of the blackboard. The ASSERT operation may include a step to COMBINE the new assertion with previous knowledge, either to check that the new assertion is consistent with what is already known, or to index the new assertion for speedy retrieval. Suppose we implement the blackboard as follows:

```
Define a BLACKBOARD:
 with an acquaintance named CONTENTS, containing the facts in the data base.

If I'm a BLACKBOARD actor and I get an ASSERT message with a NEW-FACT,
 I read out my current CONTENTS, call that CONTENTS-BEFORE-ASSERTION.
 I combine the NEW-FACT with the CONTENTS-BEFORE-ASSERTION,
  to get the CONTENTS-AFTER-ASSERTION.
 And I update my CONTENTS to be CONTENTS-AFTER-ASSERTION.
```

Looks simple, doesn't it? What could possibly go wrong? Nothing, if we're on a serial computer, but consider what can happen when two processes running in parallel each decides to try to ASSERT a new fact in the blackboard.

```
USER-1 sends ASSERT NEW-FACT-1 to a BLACKBOARD.
 The BLACKBOARD is initially EMPTY.
USER-1 combines NEW-FACT-1 with the EMPTY blackboard.
Now, USER-2 sends ASSERT NEW-FACT-2 to the BLACKBOARD.
 [USER-2 is running in parallel with USER-1.]
USER-2 combines NEW-FACT-2 with the EMPTY blackboard.
 [The BLACKBOARD is still EMPTY.]
USER-1 updates the CONTENTS of the BLACKBOARD to be:
 A data base containing only NEW-FACT-1.
USER-2 updates the CONTENTS of the BLACKBOARD to be:
 A data base containing only NEW-FACT-2.
```

and NEW-FACT-1 is lost!

3.2 Serializers are needed to protect the state of changeable actors

Why did a timing error occur when we attempted a straightforward implementation of the blackboard? The problem is that there's an implicit assumption that the

contents of the blackboard remains constant during the entire ASSERT operation. In a serial environment, that's no problem, but in a parallel one, that assumption gets violated by the other user, running in parallel, who smashes the memory while the first process isn't looking.

Another problem might arise if there had been several state variables instead of just one. We would like to keep the entire set of variables consistent, and might have the problem that different processes might be assigning different variables simultaneously.

Instead of letting the state of the blackboard be passively manipulated by its users, let's replace him with a more active object which responds to messages to perform the ASSERT operation. To insure that the state remains the same during the entire operation, we should be able to stipulate that processes should *take turns* performing the ASSERT operation. When one process sends a message, he should run until that message receives a reply that the operation is complete before another process gets a chance to get through. This facility is provided by an actor called ONE-AT-A-TIME.

ONE-AT-A-TIME is a kind of *serializer*, an actor which *restricts* parallelism by forcing certain events to happen serially. ONE-AT-A-TIME creates new actors which are protected so that only one process may use the actor at a time. A ONE-AT-A-TIME actor holds his state in a set of *state variables* which are defined locally to himself and are inaccessible from outside. He also has a script for receiving messages, and as a result of receiving a message, he may decide to change his state. When a message is received, he becomes *locked* until the message is handled and possibly, the state is changed, then he becomes *unlocked* to receive another message.

```
Define PROTECTED-BLACKBOARD:
 Create a ONE-AT-A-TIME actor,
  Protecting the unprotected BLACKBOARD.
```

If we send ASSERT messages now to the PROTECTED-BLACKBOARD, timing errors cannot arise. We could assure, through delegation, that actors which have internal state which should be protected against interference by competing processes like data bases or graphical objects always are created with their state surrounded by ONE-AT-A-TIME.

The ONE-AT-A-TIME actor embodies the same basic concept as Hoare's *monitor* idea [32]. ONE-AT-A-TIME has the advantage that it allows creating protected actors *dynamically* rather than protecting a lexically scoped block of procedures and variables. The actors created by ONE-AT-A-TIME are *first-class citizens*. They may be

created interactively at any time by a program, passed around as arguments, or returned as values, in a manner identical to that of any actor.

3.3 GUARDIANs can do more complex synchronization than ONE-AT-A-TIME

There are some kinds of synchronization which are not possible to achieve simply with ONE-AT-A-TIME. ONE-AT-A-TIME has the property that a reply must be given to a incoming message before possession is released and another message from a different process can be accepted. Sometimes a bit more control over *when* the reply is sent can be useful. The reply might sometimes have to be delayed until a message from another process gives it the go-ahead signal. In response to a message, it might be desired to cause a state change, release possession and await other messages, replying at some later time.

Imagine a *computer dating service*, which receives requests in parallel from many customers. Each customer sends a message to the dating service indicating what kind of person he or she is looking for, and should get a reply from the dating service with the name of his or her ideal mate. The dating service maintains a file of people, and matches up people according to their interests. Sometimes, the dating service will be able to fill a request immediately, matching the new request with one already on file. But if not, that request will join the file, perhaps to be filled by a subsequent customer.

The dating service is represented by an actor with the file of people as part of its state. If an incoming request can't be filled right away, the file of people is updated. The possession of the dating service actor must be released so that it can receive new customers, but we can't reply yet to the original customer because we don't know who his or her ideal mate is going to be!

The actor GUARDIAN provides this further dimension of control over how synchronization between processes takes place. When a message cannot be replied to immediately, the target actor will save away the means necessary to reply, continue receiving more messages, and perform the reply himself when conditions are right.

To do this, GUARDIAN makes use of the actor notion of *continuations*. Since the continuation encodes everything necessary to continue the computation after the reply, the GUARDIAN can remember the continuation, and reply to it later. GUARDIAN is like ONE-AT-A-TIME, but the continuation in messages sent to him is made explicit,

to give the GUARDIAN more control over when a reply to that continuation may occur. ONE-AT-A-TIME can be easily implemented in terms of GUARDIAN.

Here's the code for the computer dating service.

```
Define COMPUTER-DATING-SERVICE:
 Create a GUARDIAN actor,
  whose internal state variable is a FILE-OF-PEOPLE.

If I'm a COMPUTER-DATING-SERVICE and I get a message from a LONELY-HEART
 with a QUESTIONNAIRE to help find an IDEAL-MATE:
 Check to see if anyone in the FILE-OF-PEOPLE matches the QUESTIONNAIRE.
  If there is, reply to the LONELY-HEART the name of his or her IDEAL-MATE,
   and reply to the IDEAL-MATE the name of the LONELY-HEART.
  Otherwise, enter the LONELY-HEART in the FILE-OF-PEOPLE,
   and wait for a request from another LONELY-HEART.
```

What really happens in the implementation if more than one process attempts to use a GUARDIAN actor at once? Each such actor has a *waiting line* associated with him. Messages coming in must line up and wait their turn, in *first-come-first-served* order. If the GUARDIAN is not immediately available, the process that sent the message must wait, going to sleep until his turn in line comes up, and the GUARDIAN becomes unlocked. Then, the message is sent on through, and the sender has no way of knowing that his message was delayed. Each message may change the internal state of the actor encased by the guardian.

```
Define GUARDIAN, protecting a RESOURCE, which has its own internal state:
 Each GUARDIAN has a WAITING-LINE, and
 A memory cell saying whether the RESOURCE is LOCKED, initially FALSE.

If I'm a GUARDIAN, and I get a MESSAGE,
 If I'm LOCKED, the sender must wait on the WAITING-LINE until
  I'm not LOCKED and the sender is at the front of the WAITING-LINE.
 Set the LOCKED flag to TRUE.
 Send the RESOURCE the incoming MESSAGE,
  Along with the REPLY-TO continuation that came with the MESSAGE,
  so the RESOURCE can send a reply for the MESSAGE.
 The RESOURCE might update his internal STATE as a result of the MESSAGE.
 Then, set the LOCKED flag to FALSE,
  letting in the next process on the WAITING-LINE, if there is one.
```

3.4 Waiting rooms have advantages over busy waiting

Several situations in implementing the parallel facilities of Act 1 require a process to *wait* for some condition to become true. If a message is received by a future before it finishes computing, the sender must wait for the computation to finish. If a message is received by a guardian while it is locked, the sender must wait until the guardian becomes unlocked, and the sender's turn on the queue arrives. If a message is received by some actor which provides *input* from some device like a terminal or disk, the input requested may not be available, so the requesting process must *wait* until such time as the input appears.

One way to implement this behaviour is by *busy waiting*. When a process must wait for some condition, it goes into a loop, repeatedly checking the condition. The process can proceed only when the condition becomes true.

Busy waiting is a bad idea because it is subject to needless *deadlock*. The outcome of a condition is sensitive to the time at which the condition is checked. If the condition becomes true for a while, then false again, it's possible the condition won't be checked during the time it is true. The process will fail to leave the busy waiting loop when it should. Since one process which is waiting might depend upon another's action to release it, failing to detect the release condition for one process can cause a whole system containing many processes to grind to a halt. Another disadvantage of busy waiting is that repeated checking of conditions wastes time. It is inefficient to check conditions even when nothing has changed that might affect the conditions.

A preferable technique for implementing wait operations is to have *waiting rooms*. When too many people try to visit the dentist at the same time, they must amuse themselves sitting in a waiting room listening to muzak and reading magazines until the dentist is ready to see them, then they can proceed. Except for the time delay caused by the interlude in the waiting room, their interaction with the dentist is unaffected by the fact that the dentist was busy at first.

Our *waiting rooms* are lists of processes which are waiting for some condition to happen. When a message is sent to some actor who wants to cause the sender to wait, he places the sending process in a waiting room, including the sender's continuation, which contains all information necessary to reply to the sender. It is the responsibility of the actor which enters a sender in a waiting room to decide when the condition is true. When he signals the condition true, everybody waiting for that condition in a waiting room receives a reply.

With waiting rooms, conditions are not checked repeatedly, and there's no danger of overlooking an occurrence of the release condition. Act 1 does not provide a general WAIT primitive which accepts any boolean condition, as some other formalisms do. The only kinds of conditions that are safe from unwanted deadlock are *one-shot* conditions, which never become false once they become true. Act 1's primitives are implemented only using these kinds of conditions. Even if the implementation language's low level primitives use busy waiting for synchronization, we could still implement Act 1's primitives safely. If we had a WAIT primitive, there would be no way to enforce the constraint on waiting conditions. Any waiting in Act 1 happens as a result of sending messages to a future or serializer, which use waiting rooms.

Waiting rooms do introduce a problem with garbage collection, however. In order for an actor with waiting room to wake up a waiting process, it must *know about* (have a pointer to) that process. If the only reason a process is held onto is that it has requested something which isn't immediately available, that process is really no longer relevant. But the waiting room pointer protects the process from garbage collection. Waiting rooms should be implemented using *weak pointers*, a special kind of Lisp pointer which doesn't protect its contents from garbage collection.

3.5 RACE is a parallel generalization of Lisp's list structure

For creating an ordered sequence of objects, Lisp uses the elegant concept of *list structure*, created by the primitive CONS. For creating sequences of objects which are *computed in parallel*, and which are ordered by the time of their completion, we introduce an actor called RACE. RACE is a convenient way of collecting the results of several parallel computations so that each result is available as soon as possible.

CONS produces lists containing elements in the order in which they were given to CONS. RACE starts up futures computing all of the elements in parallel. It returns a list which contains the values of the elements *in the order in which they finished computing*. Since lists constructed by RACE respond to the same messages as those produced by CONS, they are *indistinguishable* from ordinary serial lists as far as any program which uses them is concerned.

If we ask for the FIRST or REST (CAR or CDR) of the list *before* it's been determined who has won that race, the process that sent the message *waits* until the outcome of the race becomes known, then gets the answer. This is just like what happens if we send a message to a future before the future has finished computing a value. (The RACE idea is similar to the *ferns* of Friedman and Wise [28].)

Using RACE, we can easily implement a parallel version of OR, helpful if we want to start up several heuristics solving a problem, and accept the result of the first heuristic to succeed in solving the problem. PARALLEL-OR starts evaluating each of a set of expressions given to it, in parallel, and returns the first one which evaluates TRUE, or returns FALSE if none of the expressions are TRUE.

We just create a RACE list, whose elements are the evaluated disjunct expressions. This searches all the disjuncts concurrently, and returns a list of the results. A simple, serial procedure runs down the list, returning the first TRUE result, or FALSE if we get to the end of the list without finding a true result. Since the elements of the list appear in the order in which they finish evaluating, as soon as a true element is found by any one of the disjuncts, it will be found by the procedure checking the results, and the PARALLEL-OR will return.

All evaluations which might still be going on after a TRUE result appears become inaccessible, and those processes can be garbage collected. We don't need to explicitly *stop* the rest of the disjuncts. If there are still processes running and we haven't found a TRUE value yet, the procedure checking the results will wait.

```
Define PARALLEL-OR of a set of DISJUNCTS:
 Start up a RACE list evaluating them in parallel using EVALUATE-DISJUNCTS.
 Examine the results which come back using EXAMINE-DISJUNCTS

Define EVALUATE-DISJUNCTS, of a set of DISJUNCTS:
 If the set is empty, return the empty list.
 Otherwise, break up the disjuncts into FIRST-DISJUNCTS and REST-DISJUNCTS.
  Start up a RACE between
   testing whether the evaluation of FIRST-DISJUNCT results in TRUE, and
   doing EVALUATE-DISJUNCTS on the REST-DISJUNCTS list.

Define EXAMINE-DISJUNCTS, of a list of clauses from EVALUATE-DISJUNCTS:
 If the list is empty, return FALSE as the value of the PARALLEL-OR.
 If not, break the list up into FIRST-RESULT and REST-RESULTS.
  If FIRST-RESULT is true, return it as the value of the PARALLEL-OR.
  Otherwise, do EXAMINE-DISJUNCTS on the REST-RESULTS list.
```

As another illustration of the use of RACE, consider the problem of *merging* a sequence of results computed by parallel processes. If we have two lists constructed by RACE whose elements appear in parallel, we can merge them to form a list containing the elements of both lists in their order of completion. The first element

of the merged list appears as the first of a RACE between the first elements of each list. This leads to a simple recursive program to merge the lists, which is similar to a program which interleaves the elements of two serial lists.

```
Define MERGE-LISTS of ONE-LIST and ANOTHER-LIST:
 Return a RACE between
 The first element of ONE-LIST, and
 the result of MERGE-LISTS on ANOTHER-LIST and the rest of ONE-LIST.
```

The implementation of RACE is a bit tricky, motivated by trying to keep RACE analogous to CONS. RACE immediately starts up two futures, for computing the FIRST and REST. The outcome of RACE depends upon which future finishes first. We use a serializer to receive the results of the two futures and deliver the one that crosses the finish line first.

First, let's look at the easy part, when the future for the FIRST element finishes before the future for the REST finishes. At that time, we can declare the race over, and the race actor can reply to FIRST and REST messages sent to him. When he gets a FIRST message, he should reply with the value of the first argument to RACE, the actor that won the race. When he gets a REST message, he can reply with *the future computing the second argument to* RACE, even though that future may *still be running.*

The more difficult case is when the REST future finishes first. We have to examine the value returned by the REST future to decide how to construct the list. The trivial case occurs when the REST future returns THE-EMPTY-LIST. A RACE whose REST is empty should produce a list of one element just like a CONS whose rest is empty.

If the value of the REST future isn't empty, then we can assume he's a list, and the trick is to get him to work in the case where he's a list produced by RACE. So we ask that list for *his* first element. If the list was produced by RACE, this delivers the fastest element among those that he contains. This element then becomes the first element of the RACE list node that's being constructed. The FIRST future, still running, must RACE against the remaining elements of the list to produce the REST of the final RACE list. When the computation of an element which appears farther down in the list outraces one which appears closer to the front of the list, he *percolates* to the front, by *changing places* with elements that are still being computed.

```
Define RACE, of a FIRST-FORM and REST-FORM:
 Create futures evaluating FIRST-FORM and REST-FORM.
  Return whichever of these FINISHES-FIRST:
  Either the FIRST-FUTURE-WINS, or
  the REST-FUTURE-WINS.

Define FIRST-FUTURE-WINS:
 If the FIRST-FUTURE finishes first,
  Return a list whose first is the value of the FIRST-FUTURE
  and whose rest is REST-FUTURE, which may still be running.

Define REST-FUTURE-WINS, helping RACE:
 If the REST-FUTURE finishes before the FIRST-FUTURE does,
  Then, look at the value of the REST-FUTURE.
   If it's empty, wait until the FIRST-FUTURE returns,
    and return a list of FIRST-FUTURE.
  If it's not empty,
   Return a list whose first is
    the first of the list returned by REST-FUTURE,
    and whose rest is
     a RACE between
     The value returned by FIRST-FUTURE, and
     The rest of the list returned by REST-FUTURE.
```

Section 4. We would like to implement Act 1 on a real parallel machine

Our experience with the present Act 1 implementation has taught us much about how to conceptualize parallel computations and realize implementations of parallel programs. We feel the technology will soon be ripe for implementing a language like Act 1 on a real network of parallel processors. Our group is proposing to construct such a machine, called the Apiary [10]. Implementation of Act 1 on an Apiary will require solving additional problems which we have not addressed in this paper or in the current implementation. These include transporting objects between processors, load balancing and garbage collection. A preliminary simulation of the Apiary has been constructed by Jeff Schiller, using several Lisp Machines, connected by a local packet switching network.

Acknowledgements

The notion of an actor, the future and serializer constructs for parallelism were invented by Carl Hewitt.

We would like to thank Luc Steels, Kenneth Kahn, Carl Hewitt, William Kornfeld, Daniel Friedman, David Wise, Dave Robson, Giuseppe Attardi, Maria Simi, Gerald Barber, for their helpful comments and suggestions on earlier drafts of this paper.

Bibliography

[1] Marvin Minsky and Seymour Papert, The Society Theory of Mind, MIT unpublished draft

[2] Henry Lieberman, A Preview of Act 1, AI memo 625, MIT AI Lab, 1980

[3] Carl Hewitt, Viewing Control Structures As Patterns Of Passing Messages, in "Artificial Intelligence, An MIT Perspective", Brown and Winston, eds.

[4] Carl Hewitt, et al., A Universal, Modular Actor Formalism For Artificial Intelligence, Third International Joint Conference on Artificial Intelligence, 1973

[5] Carl Hewitt, Giuseppe Attardi, and Henry Lieberman, Specifying And Proving Properties Of Guardians For Distributed Systems, Conference on Semantics of Concurrent Computing, Evian, France 1979

[6] Carl Hewitt, Giuseppe Attardi, and Henry Lieberman, Security And Modularity In Message Passing, First Conference on Distributed Computing, Huntsville, Ala. 1979

[7] The Incremental Garbage Collection Of Processes, Henry Baker and Carl Hewitt, Conference on AI and Programming Languages, Rochester, NY, Aug 1977

[8] Henry Baker, List Processing In Real Time On A Serial Computer, Communications of the ACM, April 1978

[9] Henry Baker, Actor Systems For Real Time Computation, MIT Lab for Computer Science report TR-197

[10] Carl Hewitt and Jeff Schiller, The Design of Apiary-0, 1980 Lisp Conference, Stanford University

[11] Kenneth Kahn, Director Guide, MIT AI lab memo

[12] Kenneth Kahn, How To Program A Society, Proceedings of the 1980 AISB Conference

[13] Kenneth Kahn, Creating Computer Animation From Story Descriptions, MIT AI lab report TR-540, 1980

[14] Kenneth Kahn, Dynamic Graphics Using Quasi-Parallelism, ACM SigGraph 78 Conference

[15] Alan Kay, Microelectronics and the Personal Computer, Scientific American, September 1977

[16] Adele Goldberg, Dave Robson and Xerox PARC Learning Research Group, Smalltalk: Dreams and Schemes, Forthcoming, 1981

[17] Daniel Ingalls, The SmallTalk 76 Programming System: Design and Implementation, Fifth ACM Conference on Principles of Programming Languages, 1978

[18] Alan Kay and Adele Goldberg, Personal Dynamic Media, IEEE Computer, March 1977

[19] Alan Borning, ThingLab: An Constraint Oriented Simulation Laboratory, Artificial Intelligence Journal, to appear 1981

[20] Jon Allen, Anatomy of Lisp, McGraw Hill 1979

[21] David Moon, MacLisp Reference Manual, MIT Lab for Computer Science report

[22] Daniel Weinreb, David Moon, Lisp Machine Manual, MIT Artificial Intelligence Lab report, 1978

[23] MIT Lisp Machine Group, Lisp Machine Progress Report, MIT Artificial Intelligence Lab memo

[24] Erik Sandewall, Programming In An Interactive Enviroment: The Lisp Experience, Computing Surveys, vol. 10 no. 1

[25] Henry Lieberman and Carl Hewitt, A Real Time Garbage Collector That Can Recover Temporary Storage Quickly, AI memo 569, MIT AI Lab, April 1980

[26] G. Birtwhistle, O. J. Dahl, B. Myrhaug, K. Nygaard, Simula Begin, Auerbach, 1973

[27] Daniel Friedman and David Wise, Cons Should Not Eval Its Arguments, Indiana U. technical report 44

[28] Daniel Friedman and David Wise, A Nondeterministic Constructor for Applicative Programming, 1980 Conference on Principles of Programming Languages, Las Vegas

[29] Barbara Liskov, Alan Snyder, Russell Atkinson, Craig Schaffert, Abstraction Mechanisms in CLU, Communications of the ACM, August 1977

[30] Victor Lesser and Lee Erman, A Retrospective View of the Hearsay-II Architecture, Fifth International Joint Conference on Artificial Intelligence, 1977

[31] Hans Moravec, Intelligent Machines: How To Get From Here To There And What To Do Afterward, Stanford AI Lab memo, 1977

[32] C. A. R. Hoare, Monitors: An Operating System Structuring Concept, Communications of the ACM October 1975

[33] Jack Dennis, The Varieties of Data Flow Computers, First International Conference on Distributed Computing, Huntsville, Ala., 1979

[34] Luc Steels, Representing Knowledge As A Society of Communicating Experts, AI memo TR-542, MIT AI Lab, 1980

[35] Carl Hewitt, Giuseppe Attardi, and Maria Simi, Knowledge Embedding In The Descripion System Omega, 1980 AAAI Conference, Stanford University