From: AAAI-83 Proceedings. Copyright ©1983, AAAI (www.aaai.org). All rights reserved.

An Object-Oriented Simulator For The Apiary

Henry Lieberman

Artificial Intelligence Laboratory
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Mass. 02139 USA
Arpa Network Address: HENRY@MIT-AI

Abstract

This paper describes a simulator for the proposed *Apiary*, an object-oriented, message passing parallel machine for artificial intelligence applications, using the *actor* model of computation. The simulator implements an interpreter for the lowest level "virtual machine language" of the Apiary, specifying computations in terms of creating objects and sending messages rather than loading and storing registers. The simulator is itself programmed in the object-oriented style advocated by the actor philosophy, allowing experimentation with alternative implementation mechanisms without disturbing the behavior of the simulation. Technical details in the paper assume some familiarity with object-oriented programming and the actor formalism.

Paper category: Support Software and Hardware

1. Should a parallel machine for AI be like a parallel machine for physics?

What does it mean to build a machine optimized for artificial intelligence? Let's look at the process of building specialized machines in other domains. Mathematics and physics, like AI, are areas which have important problems where solutions are limited by constraints on computing power. In these areas, an accepted methodology for optimizing machines involves identifying the *inner loop* of some interesting problem, a small piece of code that takes a large percentage of computing resources. Then, this inner loop is implemented at as low a level as is feasible, preferably in microcode or directly in hardware. If what's taking the time in your problem is doing FFT's, build an FFT machine. Can AI use this approach?

Probably not. We conjecture that AI doesn't have a simple "inner loop", that an AI machine will have to be a fast "general purpose" problem solver, just as people are. The difference is that in physics problems the patterns of computation tend to be static and *predictable*, whereas in AI the patterns of computation are likely to be dynamic and therefore *unpredictable*. An AI program attempting to solve a problem may have no idea which one of a number of heuristics will be useful before it starts to work on the problem. It may even have to learn or invent new solution methods as it goes along. Some specialized algorithms will undoubtedly be useful, such as pattern matching, set intersection and searching, but probably no one algorithm will be so dominant as to warrant tuning an AI machine to just that algorithm.

So what can you do to optimize a machine for unpredictable computations? First, you optimize the machine to take advantage of large amounts of parallelism. It will soon be more important to take advantage of the potential parallelism in a computation than to minimize the number of machine cycles used by a computation.

It is important to optimize for *flexibility*, avoiding any sort of centralized control which might become a bottleneck. A consequence is that all resources in the machine should be allocated dynamically, including both memory and processor resources. Work should be distributed among parts of the machine as evenly as possible, to take maximum advantage of parallelism. Rather than dedicating special purpose hardware to particular algorithms, it is preferable to have many general purpose processors able to run parts of algorithms as the need arises. Computations should be able to move from processor to processor, even while they are running. Stored objects should be able to move from the memory of one processor to the memory of another processor without affecting programs that use the objects.

The programmer should be able to program the machine pretending that an "infinite" number of processors are available, just as garbage collection and virtual memory let the programmer pretend an "infinite" number of memory cells are available. The system should time-share available physical processors, just as virtual memory systems time-share the use of physical memory. Simple allocation strategies with good average behavior [like the least-recently-used paging algorithm] should be used to manage resource allocation.

These are the design principles that serve as our criteria for a parallel machine for AI. The actor model of computation, described in [1], [3], [4], [5] provides a basis for designing a machine which will meet these criteria.

2. A simulator helps us gain experience with unconventional machine architectures

The basic von Neumann machine architecture has been around for over thirty years. A tremendous amount of experience in hardware design, systems programming, debugging, and programming style has been built up over the years. Some of this experience will carry over to the new generation of parallel machines, but some of it will not. The construction of a simulator has gained us considerable experience in discovering how a parallel object-oriented machine will differ from the machines of today.

The fundamental components of a von Neumann machine are *registers*, the basic data structures *bit strings*, and the basic actions *load* and *store* of registers. An Apiary will be built of *actors* as the fundamental units, with *message passing* as the sole action and means of communication between actors. Accordingly, we have outfitted the simulator with an object-oriented instruction set, where instructions specify the creation of actors and sending of messages. We have arranged that even primitive operations like addition of numbers will obey the message passing protocol, so that new implementations of system data types can always be added to an existing system.

The introduction of parallelism requires conceptual changes and provides challenges to our implementation. We have implemented mechanisms for migration of actors and load balancing. We have begun to explore the special problems of debugging programs in a parallel environment, an area long neglected. Details on these issues will follow later in the paper.

3. The simulator itself is an experiment in object-oriented programming

If we believe that object oriented programming is a good general-purpose programming methodology, then it should be good for putting together a simulator for an object-oriented machine! We are fortunate, indeed, that the Lisp Machine, on which the simulator is implemented, has many features which support a kind of object-oriented programming style; it extends conventional Lisp by adding a new *flavor* data type. Regrettably, we cannot use the flavor **SEND** operation to model message passing between actors. Primitive Lisp functions like + do not operate on flavors, and the flavor implementation relies on Lisp stacks, making it unusable in the presence of parallelism.

There is, of course, a performance penalty in using objects down to a very low level in our machine, but the advantages are numerous. Foremost among them is the ability to experiment with implementation alternatives. As an example, transmission of actors across physical machines is done by sending messages to objects representing the connections between the machines to **TRANSMIT** and **RECEIVE** actors. We have two completely different implementations of this; one uses the Chaosnet, a packet-switched local network, the other a dedicated hardware bus coupler. Switching between alternatives does not affect any other code in the simulator.

The object-oriented philosophy also facilitates instrumentation of the simulator. Any object can be replaced by a new version with the same message passing behavior, but which also records activities for later display or analysis, without affecting the simulator's operation. The simulator can record the number of events, number of actors created, average size of actors, and other information for metering performance.

4. Parallel processing is simulated on a serial machine

The Apiary simulator runs on one or more Lisp Machines. An simulated Apiary with any number of processors can be run on a single Lisp Machines, or several machines, each physical machine simulating a subset of the Apiary processors.

Since a single Lisp Machine is a sequential computer, we must simulate the effect of running several processors concurrently in software. We have opted *not* to use the Lisp Machine's **PROCESS** objects to implement parallelism among Apiary processors, primarily because Lisp Machine process switching is inefficient and because of the lack of debugging tools for parallel programs. Instead, parallelism is simulated by a **TICK** mechanism.

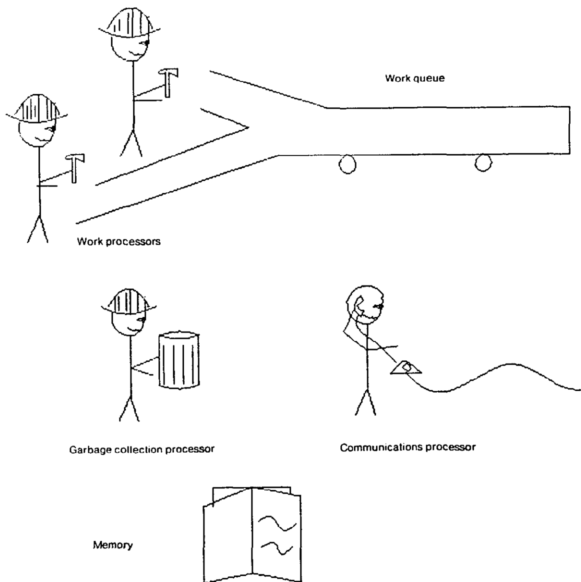
A **TICK** is the smallest quantum of time in the Apiary, the "cycle time" of a processor. When the object representing a physical processor receives a **TICK** message, it performs one primitive *event*, causing an actor to receive a message. The Apiary distributes tick messages among simulated processors. We do not rely on the presence of a global clock, or synchronization between ticks on different processors.

5. The architecture of an Apiary worker

Each individual processor in the Apiary is called a **WORKER**, and the simulator contains a worker object to represent each processor. Each worker is connected to a list of **NEIGHBORS**, a small number of other workers in the Apiary. The internal structure of each worker involves several subprocessors; a **COMMUNICATIONS PROCESSOR**, which sends and receives messages between workers, and one or more **WORK PROCESSORs** which run programs. **GARBAGE COLLECTION** processors may perform steps of an incremental, real-time garbage collection in parallel with the work processors [6].

Instead of having registers as in a conventional machine, the "machine state" of a worker is represented by an object called a **TASK**. A task is the most fundamental unit of "work to be done" in the Apiary, representing the reception of a single message by a target actor. It is very important for a parallel machine that the machine state be encoded in objects of small size. Process switching time can be slowed if the machine must switch between states comprising large numbers of registers.

Each worker has a **WORK-QUEUE**, a list of tasks representing all the computations that the worker may perform concurrently at a given moment. Work processors may take tasks from the queue for execution. The work queue must be synchronized to allow access from more than one work processor.



6. The Apiary instruction interpreter is based on objects rather than bit strings

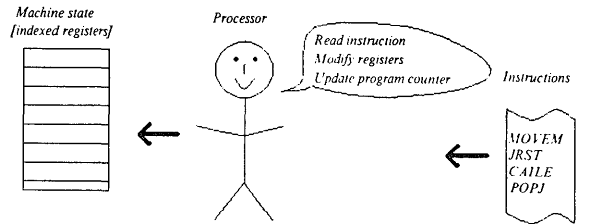
Consider a conventional von Neumann machine. The behavior of the machine is usually defined in terms of an *instruction interpreter*, or *virtual machine*. This is an algorithm that takes a machine state, defined in terms of the contents of the relevant machine registers, and an instruction in the binary machine language, and yields a new state of the machine, perhaps by changing registers, the program counter, etc. The state of the machine is represented by an array of indexed memory locations, each one containing a fixed-length bit string. The instructions are represented by fixed- or variable-length bit strings, and cause the contents of various memory locations to be altered to obtain the next machine state.

The heart of the Apiary consists of an *instruction interpreter* or "virtual machine" for each work processor. In contrast to von Neumann machines, the memory of Apiary workers is considered to consist, at the virtual machine level, of *objects* rather than bit strings. Although at the lowest level, objects must be encoded as bit strings. Apiary instructions do not treat them as such. For example, there are no instructions which load and store registers. The instructions themselves are also represented as objects, and the components of instruction objects replace "addressing modes" in conventional instructions.

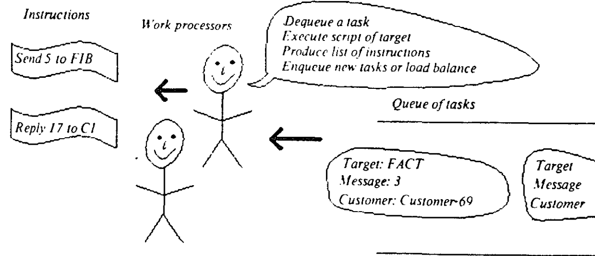
The execution of each instruction object is expected to produce zero or more new instruction objects. An instruction producing only one new instruction corresponds to the case of a traditional machine sequentially executing instructions. More than one new instruction indicates concurrency or "forking". Finally, an instruction generating no new instructions indicates the termination of a process. This method of implementing the instruction interpreter eliminates the troublesome program counters and side effects to internal registers of conventional machines.

Each instruction object specifies a *state transition* function, from the task which represents the "old state" to a task representing the "new state" of the work processor. The instruction may also result in creating new actors for components of the new task. The new tasks produced by an instruction may either be placed back on the work queue of the worker from which they came, or sent to the work queues of neighboring workers for load balancing.

The lowest-level programs which control what happens when actors receive messages, are called *scripts*, written in terms of these instruction objects. Scripts are the Apiary's "microcode", and are used as the target language for compiling very low level software. Given the "current state" of the work processor, as embodied in a task object, the script produces a list of one or more instructions, which then produce new tasks, and so on.



A conventional instruction interpreter



The instruction interpreter of the Apiary

7. The data architecture of the Apiary

In some sense, there is only one kind of data object in the Apiary, an *actor*. The implementation, however, distinguishes between *rock-bottom* actors and *scripted* actors. A scripted actor is made up a two parts: a *procedural* part and a **data** part. The procedural part is a program, the **script**, that tells the actor how to behave when it receives a message. Scripted actors have their scripts stored explicitly as the first component of the actor data structure.

What language are scripts written in? In the simulator, primitive scripts are written in the implementation language, Lisp. In a hardware Apiary, the most primitive scripts are written in the machine's "microcode", directly accessing hardware primitives. The user may create new objects to serve as scripts, executed by an interpreter written in the implementation language.

The *acquaintances* are the "data part" of an actor. These are a list of actors which are remembered as the actor's local state. The script may access these actors in forming new tasks.

Actors also can migrate from worker to worker. To accomplish this, each actor has a `FORWARD-TO` component. If non-`NIL`, all messages intended for that actor are passed along instead to the actor named in the `FORWARD-TO` part. This may result in sending the message across workers.

But not all actors can be represented as a structure containing script and acquaintances. At some point, we must have *rock-bottom* objects like numbers that can be operated on directly by the hardware without going through the message passing protocol. If a rock-bottom actor appears in the target component of a task, the script for that kind of primitive object is retrieved from a table of such scripts, indexed by type. The "acquaintances" in this case are simply the underlying machine representation of the object. In the simulator, Lisp objects such as numbers, symbols, and lists are rock-bottom actors. In a hardware Apiary, these would be a set of data types distinguished by type codes.

For example, the script for a rock-bottom number can receive a message asking it to add itself to another number. It checks its operand to see if it, too, is a rock-bottom number. If so, the machine operation for adding two numbers can be safely used. If the other number is a scripted actor, then it is given the responsibility of figuring out how to perform the add operation by sending the add message to it, passing in the original target number as an operand.

8. Scripter is a high-level "microcode compiler" for writing scripts of actors

What corresponds to the "microcode" on an Apiary machine are programs for a set of scripts for primitive actors. These perform the lowest level operations of the machine, like adding two numbers, constructing lists, extracting elements from lists, or changing the bits of the display screen.

Programming directly in the language which drives the virtual machine of the simulator is, unfortunately, not very convenient. Because the object-oriented philosophy is pressed to such a low level of the machine, even small programs require code for creating large numbers of objects. A simple `FACTORIAL` takes about three pages, which is probably the world's record for the longest `FACTORIAL` program!

A higher level "microcode compiler" called *Scripter* uses Lisp macros to compile more concise programs to code which drives the simulator [or eventually, hardware] directly. Unlike most current microcode compilers, the code looks more like an object-oriented variant of Lisp than an assembly language. Most function calls are replaced by the message sending primitive `ASK`. Here is the Scripter code for `FACTORIAL`, in its entirety:

```
(DEFSCRIPT FACT (N)
          (IF (ASK N (A ZEROP))
              1
              (ASK (ASK FACT
                        (ASK N (A 1-)))
                   (A *
                      (WITH MULTIPLIER
                            N)))))
```

Scripter still isn't a "user-level" language, however, since it doesn't have an interpreter, and is allowed to "cheat" and call Lisp [eventually, hardware] primitives directly without going through message passing protocol. However, it is the responsibility of any Scripter-written scripts to make any "cheating" completely transparent to user-written code. Scripter scripts must always check before performing any primitive operations on objects, and revert to message sending if they encounter user-defined objects. This check will be performed by macros supplied by Scripter.

9. Scripter provides several services which aid the script writer

One of the services performed by Scripter is to convert code written in the usual functional style of Lisp to continuation style, automatically creating continuation actors as necessary.

Ordinary function call/return control structure is *bidirectional*. Whenever a function is called, a return address is pushed on a stack, and popped upon return from the function. Message passing, by contrast, is *unidirectional*, and there are no stacks in a message passing machine. The functional style is achieved by *continuation passing*, where a *request* event, which corresponds to a function call, contains a *customer*. The customer is an actor which will receive the returned value of the function as a message in a *reply* event and will "continue" the computation. Scripter automatically figures out which actors need to be saved as acquaintances of customers, and provides syntax for accessing acquaintances of actors using simple variable references.

An expression

```
(ASK (ASK TARGET-1 MESSAGE-1)
     (ASK TARGET-2 MESSAGE-2))
```

would be translated as

```
Create a REQUEST-INSTRUCTION object,
  Sending MESSAGE-1 to TARGET-1
  with a new customer CUSTOMER-1.

CUSTOMER-1 receives a message ANSWER-1,
  [ANSWER-1 is the result of
   (ASK TARGET-1 MESSAGE-1)]
```

```
And sends MESSAGE-2 to TARGET-2,
With a new customer CUSTOMER-2,
  which has an acquaintance ANSWER-1.

CUSTOMER-2 receives the result of
  (ASK TARGET-2 MESSAGE-2),
   And sends it to ANSWER-1.
```

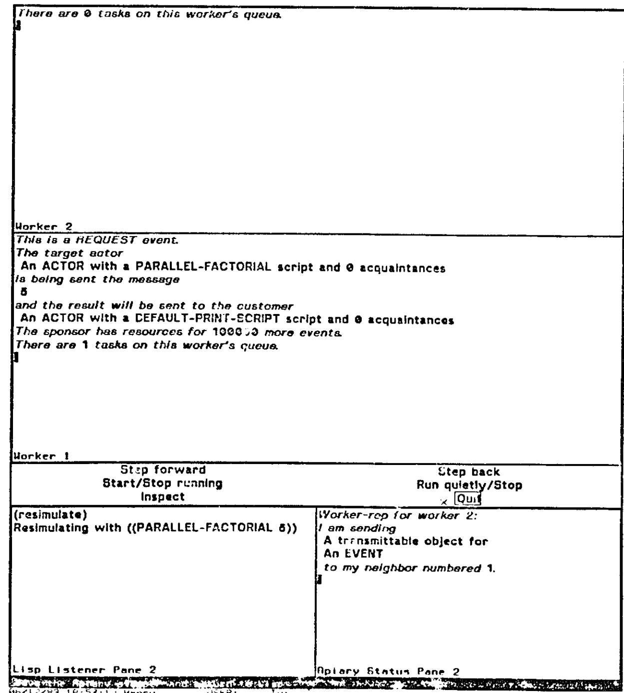
Scripter provides macros which abstract out common patterns of message passing. For example, to stick to our uniform actor protocol, conditionals must be done by message passing. Scripter provides an `IF` macro which replaces the traditional `T`-or-`NIL` test with sending an `IF` message containing the two alternatives to the result of the predicate part of the conditional.

Scripter tries to make the translation between source code and simulator code *reversible*, to aid debugging. Each piece of translated code has a component which stores the source code which produced that target code. Customers created by Scripter remember the source code which produced the value which they receive. The correspondence between source and target code is not one-to-one, since some Scripter constructs may produce more than one instruction for the simulator. The ability to even partially reverse the transformation performed by Scripter has proven valuable in debugging Scripter's output.

10. The simulator incorporates a window-oriented "machine language" debugger

How do we tell if the simulator is performing correctly? One tool is a "machine language" stepper for the Apiary virtual machine, a parallel generalization of traditional machine language steppers such as the classic `DDT` for the PDP-10/20 machines. It is *not* intended to replace tools for debugging user programs, but rather to test whether the simulator works, test the output of higher level language compilers, and act as a debugger of last resort for particularly hard-to-find bugs. It gives a "worker's-eye view" of the Apiary, using a separate Lisp Machine window to display the state of each worker. For tasks created by Scripter code, the stepper may also display the source code which corresponds to that event.

The next illustration shows a simulated Apiary with two workers about to work on `(PARALLEL-FACTORIAL 5)`. Only one worker is actually busy at the moment.



The first line in each worker window indicates the kind of event taking place -- usually a `REQUEST`, `REPLY` or `COMPLAINT` event. The next few lines contain descriptions of the event. A window at the bottom right displays the communications traffic between workers. Each object in the Apiary simulator accepts messages to produce an English-like description of itself for display in the window.

The following illustration shows the `FACTORIAL` computation at a later stage. The program has broken up the task of computing factorial of 5 into the tasks of computing the product of numbers from 1 to 3 and the product of numbers from 4 to 5, in parallel. Next, we will decompose the product from 1 to 3 into multiplying 3 by the product of 1 to 2. The load balancing algorithms have spread work from one worker to another, so now both workers are busy.

We can step the whole Apiary or individual workers, one event at a time. Each worker keeps a history of its states, so that workers can be stepped backward as well as forward. The Apiary can be run continuously, either with or without displaying events after each step. It is also useful to be able to specify a description of a certain event, and tell the stepper to run until an event satisfying the condition is encountered.

```
This is a REQUEST event.
The target actor
 An ACTOR with a RANGEPRODUCT script and 0 acquaintances
is being sent the message
 (A DOIT (WITH LOW 1) (WITH HIGH 2))
and the result will be sent to the customer
 An ACTOR with a FORWARDING-SCRIPT script and 2 acquaintances
The sponsor has resources for 100000 more events.
This is a REPLY event.
The value
 T
is being returned to the customer
 An ACTOR with a FORWARDING-SCRIPT script and 2 acquaintances
The sponsor has resources for 99999 more events.
There are 2 tasks on this worker's queue.

Worker 2
This is a REPLY event.
The value
 An ACTOR with a FUTURE script and 1 acquaintances
is being returned to the customer
 A CUSTOMER actor whose script is RANGEPRODUCT-SCRIPT-154
The sponsor has resources for 99958 more events.
This is a REQUEST event.
The target actor
 An ACTOR with a RANGEPRODUCT-SCRIPT-152 script and 2 acquaintances
is being sent the message
 T
and the result will be sent to the customer
 A CUSTOMER actor whose script is RANGEPRODUCT-SCRIPT-157
The sponsor has resources for 99858 more events.
There are 2 tasks on this worker's queue.

Worker 1
          Step forward        x                    Step back
       Start/Stop running                      Run quietly/Stop
            Inspect                                  Quit
(resimulate)                                  Worker-rep for worker 1:
Resimulating with ((PARALLEL-FACTORIAL 6))    I am sending
Computing the product of numbers from 1 to 5   A transmittable object for
Computing the product of numbers from 1 to 3   A TASK
Computing the product of numbers from 4 to 5   to my neighbor numbered 2.

Lisp Listener Pane 2                          Apiary Status Pane 2
06/12/83 18:44:13 Henry        USER:      Tyi
```

11. Acknowledgments

Major support for this research was provided by the System Development Foundation. Additional support was provided in part by ARPA under ONR contract N00014-80-C-0505.

I would like to thank Carl Hewitt for originating the actor and Apiary concepts, and support in their implementation. I would like to thank Charles Smith of SDF for his help in obtaining support for this research. Jon Amsterdam, Dan Theriault, and Roy Nordblom contributed code to the Apiary simulator and Scripter language. I am grateful to Gene Ciccarelli, Al Davis, Peter deJong, Mike Farmwald, Peter Fiekowsky, Peter Hart, Kenneth Kahn, William Kornfeld, Carl Mikkelsen, Henry Sowizral, and Daniel Weld, for Apiary-related discussions.

References

1. Carl Hewitt. Viewing Control Structures As Patterns of Passing Messages. In R. Brown and P. H. Winston, Ed., *Artificial Intelligence, an MIT Perspective*, MIT Press, 1979.

2. Carl Hewitt. The Apiary Network Architecture for Knowledgeable Systems. Proceedings of the First Lisp Conference, Stanford University, August, 1980.

3. Henry Lieberman. A Preview of Act 1. AI Memo 625, MIT Artificial Intelligence Laboratory, April, 1980.

4. Henry Lieberman. Thinking About Lots of Things At Once Without Getting Confused. AI Memo 626, MIT Artificial Intelligence Laboratory, April, 1980.

5. Henry Lieberman. Machine Tongues IX: Object Oriented Programming. *Computer Music Journal* 6, 3 (Fall 1982).

6. Henry Lieberman and Carl Hewitt. A Real Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM* (June 1983).