MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper No. 266 December 1984

Tradeoffs in Designing a Parallel Architecture for the Apiary

Carl R. Manning

ABSTRACT

The Apiary is an abstract architecture computer architecture designed for performing computation based on the idea of message passing between dynamic computational objects called actors. An apiary connotes a community of worker bees busily working together; similarily, the Apiary architecture is made of many *workers* (processing elements) computing together. The Apiary architecture is designed to exploit the concurrency inherent in the actor model of computation by processing the messages to many different actors in parallel. This paper explores the nature of actor computations and how the Apiary performs computation with actors to give the reader some background before looking at some of the tradeoffs which must be made to design special purpose hardware for the Apiary.

Table of Contents

Introduction

The Apiary is an abstract architecture computer architecture designed for performing computation based on the idea of message passing between dynamic computational objects called actors. An apiary connotes a community of worker bees busily working together; similarily, the Apiary architecture is made of many *workers* (processing elements) computing together. Thus the Apiary is an architecture designed to exploit the concurrency inherent in the actor model of computation by processing the messages to many different actors in parallel. This paper explores the nature of actor computations and how the Apiary performs computation with actors to give the reader some background before looking at some of the tradeoffs which must be made to design special purpose hardware for the Apiary.

The Actor Model of Computation

The actor model of computation is based on the idea that computation can be performed by sending messages between dynamic objects called actors. Actors are somewhat like objects in Simula and Smalltalk, but the behavior of an actor in response to receiving a communication is more limited. As a result of receiving a communication, an actor may make simple decisions, send new communications to actors it knows, create new actors, and specify a replacement behavior. In particular, actors may not loop or make assignments to variables. All control structures are implemented as patterns of passing messages, so for example iterative loops are tail recursive message passing. There is no assignment in actor languages; all change is accomplished through the mechanism of specifying a replacement behavior to handle the next message. (See [3]) Like other object oriented languages, everything in the actor model is an actor, and may be sent messages.

The structure of an actor

An actor has several parts which define its behavior. All actors have a *script*, the "program" which tells it what to do when it receives a communication. Many actors may share the same script; they will have the same general behavior. Actors may also have zero or more *acquaintances*, other actors they know about and may communicate with as the result of receiving a message.

The behavior of an actor

The behavior of an actor is how it responds to communications. An actor may only make simple decisions in deciding how to respond to a communication; then it may do any of three things. It may create new actors. It may send new communications to actors it knows either as acquaintances, from the communication received, or through creation. It may specify a replacement behavior to process the next communication.

An actor's behavior is determined by its script and its acquaintances. When a new actor is created, its script and all of its acquaintances must be specified. Similarily, when specifying the replacement behavior of an actor, the new script and acquaintances must again be specified.

Mail addresses

The message delivery system of the actor model is similar in character to a mail system. Communication between actors is asynchronous; when one actor sends another a message, the sender does not wait for the receiver to get the mail, but may move on to process its next message. To send a message to another actor, an actor must know the receiver's mail address, either because the receiver is an acquaintance or because the receiver was part of the message currently being processed.

Tasks and events

A *task* is the pairing of a communication with its target actor. It is analogous to an envelope -- it contains both the mail address of the target actor and the communication itself. An *event* is the execution of a task; it is the acceptance of a communication by an actor. Events are atomic. Recall that as the result of accepting a communication, an actor may only make simple decisions regarding the nature of the message, create new actors, send more messages, and specify a replacement behavior for itself. Sending more messages just creates new tasks for the mail system to deliver. All of these operations are done in the single atomic operation of executing the task.

Request + response = transaction

Communications are divided into two general categories. A *request* communication contains a *customer*, the actor to which the *reply* communication should be sent. It also contains a *complaint department*, the actor to which a *complaint* communication should be sent if the request cannot be fulfilled as expected. Either a reply or a complaint may be the *repsonse* to a request. A request and its response make a pair called a *transaction.*

Messages

Messages, the "information" part of a communication, may in theory take any form, but it is convenient to make them as descriptive as possible. For example, a message may take the form of an *instance description* such as **(a-message add-node (with parent self) (with value v))**. The *concept* ("add-node") serves to identify what kind of message this is, while the attributions identify and label arguments. Such descriptive messages can be decomposed by the receiving actor without worrying about argument order.

Continuations

In order to handle replies to requests, actor computations make wide use of short-lived actors called *continuations.* When an actor must make subtransactions before the response to a request can be determined, the actor creates a continuation actor to serve as the customer for the transaction. The continuation has as acquaintances any actors which will be needed to finish computing the response once it receives the reply; these may include acquaintances of the actor which received the request, and parts of the request communication including the customer to whom the reply should be sent. Since sequential transactions are performed by strings of continuations, intermediate results may also be acquaintances of later continuations.

If the actor which received the request can specify its replacement behavior without the intermediate results, (e.g. if the behavior stays the same), then it is free to accept its next communication immediately following the event; the continuations will take care of completing the first transaction. Since the continuations are created dynamically for each transaction, many transactions may be computed in parallel.

If the replacement behavior of the actor which received the request depends upon the intermediate results, the actor specifies the *insensitive behavior* as its replacement behvior. While the actor is insensitive, it accepts no messages except a *replace communication* sent by a continuation at the point in the computation when the replacement behavior can be determined. The replace communication specifies the replacement behavior, and after it is received the actor may accept further communications.

Sponsors

The use of resources by the many branches of a parallel actor computation may be controlled through actors called *sponsors.* Each task must have a sponsor who "pays for" executing the task. Whenever a task is executed, the sponsor's resource of "ticks" is decremented. The new tasks produced by the event may have the same sponsor, or they may have new sponsors, each with some fraction of the remaining resources of the parent sponsor.

When the sponsor runs out of resources, it may ask its parent sponsor for more resources. Depending on the state of the computation, the parent sponsor may or may not grant additional resources. If additional resources are granted, the computation may continue. If the parent sponsor does not reply immediately, then the computation is put on hold. The parent sponsor may also decide the computation is no longer needed, and tell the child sponsor to abort and "commit suicide".

For example, if a parallel search is being performed, the parent sponsor can give each of the branch sponsors a small number of ticks. Each branch sponsor must periodically report back asking for more ticks. If one branch finds an answer, then the other branches can be told to abort, conserving the computational resource.

The Apiary Abstract Architecture

The Apiary is an abstract architecture designed to exploit the concurrency of actor computations by executing many tasks in parallel. An Apiary is a network of processing elements called *workers*; each worker is a computer in itself, with its own local memory and processing power. Each worker executes tasks for the actors which reside in its memory; as tasks are generated for actors not local to a worker they are sent to the appropriate worker over the network. While the actor computation evolves, the load of tasks and actors is balanced across workers, thus sharing the burden and increasing parallelism.

Many details of implementation are left unspecified, allowing the implementor as much flexibility as possible to acheive performance.

Network

The precise network topology of an Apiary is left to the implementor, but it is suggested that it take the form of a hypertorus[4]. The members of the hypertorus family are isotropic, uniform, and easily extended. A uniform, isotropic grid without boundaries makes the load balancing problem easier. By making the network a grid rather than a tree, the network will be capable of handling message traffic congestion or node failure by rerouting. Since each worker has the same number of *neighbors* regardless of the size of the network, an Apiary can be easily expanded or contracted.

The primary feature separating workers is that neighboring workers do not have access to either other's memory. Neighbors communicate only through the network. This does not exclude from consideration the possibility that the communication link is implemented with a common memory, but workers do not execute tasks for actors which are not in their local memory.

Workers

Workers are the processing nodes in the Apiary network. Each worker has local memory holding the actors which reside there, and processing power to execute tasks for those actors. Each worker also has a communications link to each of its neighbors, and processing power to receive tasks and actors from neighboring workers and send them to other workers.

Conceptually, the worker can be divided into the work processor(s) and the communications processor(s). A work processor takes care of executing tasks, while a communications processor takes care of network traffic. The two cooperate to perform memory management, including garbage collection and removing inactive actors.

Work processor

The work processor is the computational heart of the worker, where the the actual work gets done. The work processor spends its time in a basic loop. With each iteration, it takes the next task off its queue of tasks, executes the task, and enqueues any resulting new tasks on its queue. In more detail: the work processor takes the next task off its queue, extracts the target actor and the communication, extracts the script from the actor, and calls the script with the actor and communication as arguments. The instructions in the script may, depending upon the contents of the communication, instruct the work processor to create new actors and/or create new tasks with new communications to actors. The script then returns the new tasks created, which the work processor then enqueues on its queue of tasks to execute. Once the created tasks are enqueued, the worker processor is ready to start the next iteration.

There are a few cases where the work processor must deviate from this behavior because the task cannot be executed. If the target actor of a task does not reside on this worker, the task is placed on the communications processor's queue to be forwarded to the current location of the actor.

The target actor may also be insensitive, i.e. locked, so the task must wait. If the script of an actor does not return a task with a replace comunication specifying a replacement behavior for the actor, the actor becomes locked. A continuation should later issue a task with a replace communication for this actor, but meanwhile the actor can not accept any other communications. Therefore if the worker encounters a target actor which is locked while trying to execute a task, it enqueues the task on the actor's own incoming queue to be processed after a replacement behavior is specified.

When the work processor executes a task with a replace instruction, it does not call the script but replaces the target actor with its new behavior and unlocks it. If the locked actor had any tasks on its incoming queue, then the first is immediately dequeued and executed just as if it had come off the worker's queue.

Notice that because executing a task only represents one short, atomic event, tasks are executed quickly. Thus there shouldn't be any need to interrupt the execution of a task to share the work processor among other tasks.

The work processor also takes care of charging sponsors for executing tasks. Each task has a sponsor and an allocation of ticks. Executing a task costs one tick, so in general whenever the work processor executes a task, it decrements its number of ticks and distributes the remaining ticks among the new tasks created as a result. If the number of ticks remaining falls below some threshold, the work processor does not execute the task but instead creates a task asking the sponsor for more ticks. The sponsor may then decide whether to allow the computation to continue by giving out more ticks.[1]

Communications processor

The communications processor is the locus of all communications activity with neighboring workers. It also spends its time in a basic loop. With each iteration, the communications processor looks to see if there are any tasks from the work processor to forward, or if any of its neighbors have sent it any tasks to forward. When it finds a task to process, it looks up the target actor in its unique-id (unique identifier) table to find where it should send the task. If the actor currently resides on this worker, the task is enqueued on the work processor's queue. Otherwise the entry in the unique-id table will indicate that the actor was last known to have migrated to one of this worker's neighbors, and the task is forwarded to that worker. When the communications processor forwards a task, it also forwards the communication and the message so the task may be processed when it gets to

[1]The implementation of sponsors is an area of current research, so not all the details are known.

the target actor.

The unique-id table is a hash table of actors which both this worker and other workers know about, either because a reference to an actor has been migrated, or the actor itself has been migrated between workers. Unique-id's are created on a local basis by the communications processor whenever it migrates an actor or a reference to a locally created actor. The unique-id table is used by the communications processor both for forwarding tasks and for recreating references to actors. References to actors are migrated as unique-id's, so multiple references to the same actor may be resolved by the receiving worker.

Load balancing

So far we have mentioned communication between workers only to transmit communications between actors on different workers. The Apiary also automatically manages the computational and storage loads between workers, so the work of executing tasks and memory management may be shared among neighboring workers.

The communications processors keep a watch on the lengths of the work processors' queues, and periodically pass the information along to each of their neighbors. If a communication processor notices that a neighbor's work queue is significantly larger than its own, it may send a reqeust to the neighbor asking it to share the load. Once a request to share the load is received, the work processor may stop to examine its queue, and decide to migrate some tasks and their target actors to the neighboring worker.

We would like the process of sharing the load to increase throughput of executing tasks, so we must try to preserve the locality between systems of actors as much as possible to avoid too much inter-worker communication traffic. In other words, we should try to migrate systems of actors which are likely to communicate with each other. The simplest scheme of load balancing, that of migrating some fraction of the tasks at the tail end of the queue, fails to take into account any of the relationships between actors, and is likely to disperse organizations of actors. A more promising method is based on the sponsors; by migrating a set of tasks with common sponsors, the actors associated with the branch of the computation controlled by those sponsors may be migrated together [1]. More complicated schemes may attempt to restore locality within separated organizations by noticing message traffic to individual actors or the location of acquaintances.

However, even if we manage to migrate actors by branches of the computation, there may still be much call for an actor on the old worker. One way to improve locality between an actor and organizations on several workers is to make copies of the actor on each worker. This works fine for as long as the actor doesn't want to change its behavior; each worker can execute tasks for the actor independently. When the actor needs to change its behavior, however, the copies must be eliminated so the change may be performed atomically. A scheme for keeping a tree of the copies of an actor has been proposed so that copies of actors which do not change often may be distributed among several workers; when the actor does need to change its behavior, the tree is first collapsed to eliminate the copies.

Memory management

Actor computation is a very creation intensive style of computation. New tasks, communications, actors, and references to actors are constantly being created. The memory management scheme of the Apiary must keep reclaiming to keep up with this non-stop allocation or the Apiary will soon run out of storage space.

Actors whose references are confined to one worker may be garbage collected using traditional garbage collection methods. An ephemeral garbage collector, which quickly reclaims short term storage, is highly desireable. However, once an actor has references on workers other than the one where it resides, the worker cannot check for those references, so the management scheme must become more complicated. A scheme which has been proposed keeps trees of workers with references to the actor for each actor. Garbage collection of references is gradually performed from the leaves inward; once a worker realizes it no longer needs a reference, it may delete itself from the tree. If an actor is no longer needed, its reference tree will eventually collapse to one worker, who can then garbage collect the actor normally. (See [2])

If memory is not uniformly distributed in an Apiary, for example if not all workers have virtual memory paging out to disk, then the memory manager will also be responsible for flushing out inactive organizations of actors. This may be coordinated with the garbage collection, so actors which haven't accepted any communications within the last few garbage collection cycles may be migrated to workers with larger, slower memory, freeing up fast worker memory for more active actors. Like in the case of load balancing, it is preferable to migrate the actors by organization, so if they do become active again, they will still be in close proximity with each other.

Tradeoffs in Designing a Worker

In an ideal Apiary there would be a very large number of workers, and communication between workers would be nearly instantaneous so that executing a task on another Worker is nearly as fast as executing it locally. In such an Apiary, in order that as many tasks be executing at once as possible, there should be as many work processors as possible.

However, there are complexity, cost, and physical limits to building an Apiary, so tradeoffs must be made regarding the number of workers, the storage capacity of each worker, and the computational power of an individual worker. By examining the extremes, we can perhaps get a better idea of the tradeoffs involved, and what information would be helpful to make decisions regarding those tradeoffs. For the purposes of simplifying the discussion, we will assume that an Apiary network is built according to the hypertorus design mentioned in the previous section; for Apiaries with few workers, there may be better designs based on shared busses or Omega networks.

In order to get an idea of how much memory each worker needs, we can look at the extremes. One extreme is the uniprocessor with a large amount of memory. The opposite extreme is to give each worker as little memory as possible, say enough to support one actor.

Few processors, large memory

An Apiary design with one or just a few processors is not very interesting from the point of massive parallel processing, but it does point out the advantages of keeping storage concentrated. With only a small number of large processors, a large number of actors may spend their lives without migrating or even references migrating, so those actors may be garbage collected without the extra overhead (both memory and processing) of keeping reference trees. Because the Apiary is small, the reference trees which are made don't grow very large and may be collapsed and collected more easily. In addition, organizations of actors cannot become widely dispersed, so delays due to long distance communications are not as acute.

Keeping storage in large pieces simplifies the storage management problem, but doesn't give us much parallism in processing. There may be long delays between when a task is created and put on a worker's queue and when it is executed. Since that parallelism is one of our principle aims, the other extreme may be more interesting.

How small can a worker be?

The absolute smallest memory worker would support one actor. We will see that the overhead of supporting the actor makes it unreasonable to support just one actor on a worker, but we can get an idea of what this overhead is.

Obviously, we need space to store the actor. If the actor is a long sequence (list) or a large array, it could require much memory. For the sake of this argument, lets assume that actors are limited to 25 acquaintances. Most actors have far fewer, say, 0 to 10 besides the script and descriptor (type). We can imagine that there is a larger worker somewhere else in the network which can hold and process requests to larger actors. Thus we need about 10 words for the actor, plus a word for each of its acquaintances. A copy of its script must also be local, so it can execute tasks, so another 60 or so words of script are needed, depending on many different messages the actor can recognize.

To buffer incoming tasks with communications and messages, we need about 20 words for each task, 10 words for the communication. If we limit messages to at most a concept with 5 attributions, then we can figure on about 20-25 words for the message. If we buffer three incoming tasks, then this adds up to another 165 words.

For each non-local actor whose mail address is known on this worker, we need a structure telling where (on what other worker) we think the actor is located, i.e. reference tree links. Since the acquaintances are non-local and the attributions of the incoming messages are non-local, this adds up to about 50 non-local actors we must be able to handle, or about 300 more words.

If we allow the actor to create one new actor in the processing the task, then we need about 85 words for the new actor. If we limit it to creating 5 new tasks at a time, then we need about 275 words for creating new tasks which may then be either sent to their targets on other workers, or kept on the incoming queue if it is a recursive communication.

This adds up to about 1k words for a worker which must garbage collect between every task execution, and only processes tasks for one actor. An Apiary made of such workers must have very efficient communication between workers, since all communication between actors is done over the network. It must also have some scheme for allocating workers for newly created actors. Since actor systems generate large numbers of actors, there must be large number of workers, so a complete network, where each worker is a neighbor of all other workers is infeasible. Therefore, attention must be paid to preserving locality between actors which frequently communicate.

One actor per worker is too small

A one actor worker is probably too small to be an effective tradeoff between locality and distribution of processing power. During an actor computation, most actors are dormant, not processing any tasks, so many actors can be time-shared on one worker without any significant loss in throughput.

There are also memory considerations which tend to make putting many actors on one worker attractive. In the Apiary with one actor workers, every reference to an actor from another worker requires the equivalent of a unique-id table entry on that worker; the unique-id table takes up a major fraction of the memory on each worker. If several actors which refer to an actor reside on the same worker, then the cost of just one entry can be shared among those actors. Similarily, a one actor per worker Apiary must have a separate script for each actor so that each worker has a script to follow, while in a worker with many instances of the same type of actor, a single copy of the script may be shared among those actors.

We noted before that storage allocation and garbage collection is much easier with large workers. With very small workers, nearly all actors have reference trees with their attendant costs in memory and processing time. A more efficient garbage collection scheme taking advantage of the parallel processing would need to be developed for such an Apiary.

So the question remains: If one large worker represents too little processing power, and many minimal workers represent too little locality, then how do we find a good compromise?

How much memory should each worker have?

We've seen that providing one worker for each actor is an ineffectual allocation of processing and memory resources; most actors are idle, and such a scheme requires that much memory be used for keeping track of references across workers. So the next question arises, how much memory should a worker have?

No answers or magic formulas are now known, the tradeoffs being as complex as they are. We can hope to get a hint from simulation and performing thought experiments. Here are two possible ways of attacking the problem.

Brute force simulation:

Decide upon a technology for processing, communication, and memory management (including garbage collection); then build a simulator of an Apiary which accurately models the relative speeds of the processor, communication, and memory management. Devise a "typical load" for the future Apiary which can be run on the simulator. Choose a total amount of memory which might be used in an Apiary, then iteratively run the typical load on the simulator, dividing the memory among more workers with each iteration, until throughput no longer increases due to the increased overhead of managing small partitions of memory.

Note that this simulation requires that each simulated processor manage its own memory, so it may be difficult to use any existing memory management technology of the simulation machine (for example, a lisp machine ephemeral garbage collector). An alternative is to build a preliminary network of simulation machines and scale the processing, communication, and memory management speeds with delays so their ratios match the technology you want to simulate. Then the above iteration may be performed by adding processors to the network, each time constraining each processor to less memory.

The advantanges of this method are that the experimenter can be pretty sure that the results will be correct if the simulation is accurate. However, the brute force method is not only computer intensive but also programmer intensive. Building such an accurate simulator is a very large project. Predicting what a typical load will be may also be difficult without experience in large actor systems. If the Apiary simulated is large, available computer resources may not be sufficient to run the simulation.

There are ways to simplify the simulator, such as dividing the simulation into two simulators. The first simulator just produces a characterization of the typical load, so we have an idea of what percentage of the tasks run result in multiple tasks, what percentage of the actors may be garbage collected quickly, etc. The second simulator can then use this information to study the congestion of messages and actors on a statistical basis. However, issues such as preservation of locality between communities of actors depend on accurate modeling of load balancing and migration, which cannot be done easily on a split simulator.

Theorizing:

We can look upon the problem of how much memory to allocate each worker from the opposite viewpoint: How many workers are needed to support the actors which fit into N words memory? We may be able to find some working ratio between the breadth of the task activation tree (forest) and the number of active actors, or even the total number of actors. Here, the task activation tree is the tree with a node for each task, and where the children of each node are the nodes representing the tasks which were activated by the parent task; i.e. the child tasks were created as a result of executing the parent task. We can define the active actors to be those actors which will soon receive communications at the corresponding level of the activation tree. Some definition of the typical load is also needed for this approach.

This method has the advantage that it doesn't require a large effort to build an accurate simulator, though some sort of simulator is needed to determine and study the "typical load". However, only experience through simulation can provide answers to the questions of how the proximity of related actors effects performance or how effective garbage collection will be. Answers to those questions are important for determining how much memory a worker will need above that sufficient just to store the actors.

How powerful should a worker be?

A few notes concerning the complexity of the processor part of the worker are in order as well. Workers must be able to perform certain functions, and need to perform some functions more than others. The resulting complexity and power of the processor will also have a bearing on the size memory allocated to it.

The instructions passed around in scripts should be either compact or interpreted from a compact form so scripts may be short, thus saving space and communication. In a first implementation, they should probably be interpreted so that we may experiment with their format, and make optimizations based on experience.

Special consideration of the nature of actor computation must be made when designing processor optimizations. In particular, it is unlikely that small caches will have much, if any benefit, since the processor is constantly switching between short, unrelated tasks with no loops. Almost always the target actor, message, and script will be different for consecutive tasks. However, the instruction interpreter itself may have some loops and data locality, so caches for the interpreter may be helpful.

Some special hardware may help speed up a few key functions of the worker. Actor computations are characterized by a large amount of storage allocation and garbage collection. A large portion of the objects have very short lives, so support for an ephemeral garbage collector, including a tagged memory, is strongly advised. There are also a few important queues in each worker; if the worker will be implemented as more than one process(or), provisions for interlocking the management of queues between them should be made. The communications processor will be frequently hashing on the unique identifiers of actors as they are received, so special support for hashing functions may be warranted.

Conclusion

The actor model of computation promises to be a strong basis for programming parallel systems. The Apiary is a abstract architecture which is well suited to performing actor computations. However, before an implementation of the Apiary in hardware can be designed, tradeoffs between the number of workers and the complexity and memory size of each worker must be studied. In particular, the character of a "typical load" we would like to support is not understood -- are there many more actors than tasks, or are most actors transient? The magnitude and consequences of the dispersal of actor communities during the running of the Apiary can make a significant difference on the load put upon the communications system and the amount of memory needed to store inter-worker references. The efficiency with which inter-worker garbage collection can be done is not known, yet it also has impact on design decisions about memory size and processing power. Simulation of an Apiary with many workers and with controls on memory size could provide many insights into these questions.

Bibliography

[1] Amsterdam, Jonathan; "Load Balancing Strategies for the Apiary"; B.A. Thesis, Harvard College, May 1984

[2] Halstead, Robert H., Jr.; "Reference Tree Networks: Virtual Machine and Implementation", Chap. 5; MIT LCS TR-222, July 1979

[3] Hewitt, Carl; "Viewing Control Structures as Patterns of Passing Messages", MIT Artificial Intelligence Memo 410, December 1976

[4] Hewitt, Carl; "The Apiary Network Architecture for Knowledgeable Systems", Proceedings of the First Lisp Converence, Stanford University, August, 1980