

# Message-Passing: Computational Model, Programming Paradigm, and Experimental Studies\*

*Anand Sivasubramaniam  
Umakishore Ramachandran  
H. Venkateswaran*

Technical Report GIT-CC-91/11  
February 1991

College of Computing  
Georgia Institute of Technology  
Atlanta, GA 30332.  
Phone : (404) 894-5136.  
E-mail : *rama@cc.gatech.edu*

## Abstract

Since existing models of parallel computation do not capture the inherent asynchrony in message-passing architectures, a new model, called Communicating Random Access Machines (CRAM) is proposed. Even though message-passing is a viable alternative to shared memory for interprocess communication, it has received considerably less attention in terms of parallel algorithm development. Motivated by this observation, we propose a programming paradigm that would serve as a framework for algorithm development using the message-passing abstraction. It turns out that several asynchronous algorithms naturally fit this paradigm. An execution model for implementing this paradigm on parallel architectures is developed. Using this execution model, preliminary results from implementing this paradigm on the Intel iPSC/2 and the Sequent architectures are discussed. Such experimental studies are very important to understand the performance issues of parallel algorithms.

**Key words:** parallel architectures, message-passing architectures, models of computation, programming paradigms.

---

\*This work is supported in part by an NSF PYI Award MIP-9058430, an NSF Grant MIP-9200005, and by an NSF Grant CCR-8711749. The computational model part of this work has appeared in a condensed form in the Sixth International Parallel Processing Symposium [14].

# 1 Introduction

The programming of sequential machines has been dominated by the shared memory model. Since programming of parallel machines have tended to be the outgrowth of programming with sequential machines, parallel algorithm development as well as models of parallel computation have predominantly used a shared memory abstraction. A dual to shared memory, namely message passing, has drawn considerably lesser attention both from the programming point of view as well as the model point of view, which may be attributed to the general view that programming using the message-passing abstraction is hard. This view arises from the fact that this abstraction has not been explored as a viable alternative to shared memory for algorithm development. At best, this abstraction has been used to simulate shared memory leading to contrived implementations. This serves as a motivation for developing a paradigm for programming using the message-passing abstraction.

Another important motivation for exploring programming paradigms for message-passing stems from the architecture point of view. Parallel architectures may be broadly classified into two categories: shared memory, and message-passing based on the inherent communication and synchronization model underlying the basic architecture. Shared memory machines are characterized by a set of processors connected to a globally shared memory. All interprocessor communication and synchronization are effected via this shared memory. Message-passing machines are characterized by a set of processors each with its own private memories; there is a global interconnection network that allows processors to communicate and synchronize with one another via messages. In shared memory machines communication is implicit (e.g. updating shared locations); thus a set of mechanisms for synchronization is usually needed. On the other hand, in message-passing machines communication is explicit (via messages). Synchronization and communication are intertwined in that information exchange also serves as synchronization.

One may hypothesize that based on the above two machine classes, there ought to be two algorithm classes as well: shared memory and message passing. The justification for such a classification is that certain problems may map more naturally into one class than the other. Note that one class may be simulated with the other class using the standard principle of duality [10]. However, such simulations may be inefficient (see sections 4 and 5).

Models of parallel computation using the shared memory paradigm, such as the Parallel Random Access Machine (PRAM) [9], have implied scalability assumptions such as unlimited number of processors, unit cost for concurrent communication via the shared memory, and no-cost synchronization. In general terms a shared memory machine has three resources, namely, processors, interconnection network, and shared memory. The interconnection network serves dual purposes: switching between processors and memories; and interprocessor communication. When the number of processors scales up, while the interconnection network may be scaled up as well, the contention for the shared memory can only increase (even if the size of the shared memory is increased) thus limiting the scalability of such machines. A message-passing machine has only two resources: processors with private memories and interconnection network both of which scale well<sup>1</sup>. The interconnection network in such systems serves purely for interprocessor communication. The enhanced scope for scalability of message-passing architectures is another motivation for exploring paradigms and models for such machines.

---

<sup>1</sup>Note that there are several recent attempts to address the scalability of shared memory machines [11]; our point is merely to note that message-passing machines are intrinsically more scalable than shared memory machines [13].

The PRAM model which provides a framework for studying shared memory algorithms, captures only the computation aspect of parallel computation implicitly imposing a synchrony among the processors. However, this is not a realistic assumption since real parallel machines (such as the Sequent and the Butterfly) require programs to synchronize explicitly. Our earlier work also [15] reveals that dynamic scheduling which manifests itself as interprocessor synchronization is a limiting factor in achieving the theoretically predicted performance when such algorithms are implemented on these machines. Recently, there have been attempts at incorporating both communication and synchronization into models of parallel computation ([16], [6],[4]). For example, the asynchronous PRAM (APRAM) model introduced in [4, 6] captures the effect of asynchrony in the computation between processors. While the duality of shared memory and message-passing is well-known, a message-passing architecture has one key characteristic that distinguishes it from a shared memory architecture : the asynchronous notification associated with interprocessor communication via message-passing. Since this is an inherent property of a message-passing architecture, it is essential that any model of computation for such an architecture should capture this asynchrony. Thus we may identify two types of asynchrony in a parallel architecture : the first arises due to the relative speeds of the processors, while the second arises due to interprocessor communication. It may be noted that APRAMs capture only the first kind of asynchrony since they model shared memory architectures.

To better understand the performance issues in parallel computation it is important to undertake experimental studies involving the implementation of parallel algorithms on parallel architectures. In our earlier work [15], we implemented several shared memory algorithms on the Sequent and the Butterfly with a view to understanding the architectural impediments that limit their performance. To our knowledge, there are very few experimental studies that investigate the impact of architectural features on algorithmic performance. Anderson [1] reports results of an experimental and analytical study of parallel merge sort on the Sequent. Yew et al. [3], analyze specific parallel programs to identify the appropriate grain size of parallelism that exists in these programs. Lin and Snyder [12] compare message passing and shared memory paradigms for implementing specific parallel algorithms on shared memory multiprocessors.

This paper begins by developing a framework for message-passing architectures consisting of a machine model called *communicating random access machine (CRAM)* and a programming paradigm. The CRAM model would serve as a vehicle for the design and analysis of message-passing algorithms. The message-passing paradigm would make the mapping of algorithms that fit this paradigm onto message-passing architectures more natural. Preliminary experimental results from implementing this paradigm on parallel architectures are presented. Finally, the performance implications of implementing shared memory algorithms on message-passing architectures are discussed.

## 2 Message Passing Model

A *Communicating Random Access Machine (CRAM)* is a collection of processors each of which is a sequential random access machine (RAM) and an interconnection network for interprocessor communication. The interconnection network provides a physical connectivity between any pairs of RAMs. However, the set of processors that a RAM can communicate with (logical neighbor set) is determined dynamically as detailed below. The interconnection network is assumed to be failure free. The processors execute asynchronously with respect to one another. There is no global clock in

the system. A processor has private memory, a set of standard RAM instructions (such as reading and writing to private memory, and arithmetic and logical operations), and special instructions to model message passing. A *program* for the RAM consists of a set of labeled instructions. The special instructions that model message-passing are:

- *Non-deterministic Branch (NBR L1, L2)*: The RAM chooses non-deterministically one of the two execution paths identified by the labels L1 and L2.
- *Select Neighbor Set (SNS(< neighbor\_set >))*: The RAM creates a bit-vector that defines its logical neighbor set.
- *Send Message (SEND(< message >))*: The RAM sends the message to the processors identified by its neighbor set. The instruction is non-blocking, i.e., the processor executing this instruction does not wait for the message to be received by all the neighbors. There are two possible scenarios that may follow a SEND instruction in a program: In one case, the program may need to know that the neighbors have received the message. In the second case, the progress of the program may not depend on knowing that the message has been received. Both the scenarios may be modeled using standard RAM instructions.
- *Receive Message (RECEIVE(< message >, sender))*: The RAM receives a message (if there is any to be received). The identity of the sender is made available along with the message to the receiving RAM. The RECEIVE instruction is deemed as a non-blocking instruction (the processor executing this instruction returns immediately with either success or failure depending on whether a message is available or not).

One of the significant features of our model is its ability to capture asynchronous events associated with interprocessor communication, which is an inherent property of message-passing machines. None of the earlier models of parallel computation (such as the PRAM or the APRAM) address this asynchrony, since these models are for the shared memory paradigm. It may also be noted that asynchronous events such as interrupts can be captured using the NBR primitive of our model.

Using this model we can describe a real message-passing architecture such as the hypercube. Each processor in the hypercube has specialized instructions for communicating with one another in addition to the standard RAM instructions. The basic SEND primitive of the hypercube is non-blocking and is identical to our CRAM SEND. All other sophisticated forms of SEND (including blocking versions) can be implemented on top of this basic version using ordinary RAM instructions. The same is true of the RECEIVE primitive of the hypercube. The hypercube has a facility to interrupt a processor on message arrival. The RAM, PRAM, or the APRAM models do not provide primitives for efficiently capturing such asynchronous events. On the other hand, the non-deterministic branch primitive of the CRAM allows modeling such events. For example, interrupts may be modeled using a non-deterministic branch at the end of every instruction.

## 2.1 Computation Costs

The processors are assumed to be identical in functionality, and the clock speeds of all processors are assumed to be the same. We associate unit cost for all standard RAM instructions and the special instructions (NBR, SNS, SEND, and RECEIVE).

The NBR instruction is no different from a standard branch instruction from the point of view of execution cost. Although the model allows an unbounded number of neighbors, in reality this set is bounded by the total number of processors in the system. The SNS instruction is similar to a standard load instruction from the point of view of execution cost. For example, an implementation of the SNS instruction may be to load a processor register (interpreted as a bit-vector) from (possibly) multiple memory locations. Thus a unit cost for the SNS instruction is justified. While a logarithmic cost model proportional to the size of the neighbor set (to account for truly unbounded neighbor set) may easily be incorporated, we do not consider this approach in this paper for the sake of simplicity of analysis.

The RECEIVE instruction is non-blocking. If the message is in transit or is being formed, the instruction returns with failure. Only when the message is fully formed and available does the processor execute this instruction successfully. Thus the instruction execution time may be considered independent of communication delays such as the copying time of the message, and the transit time for the message through the interconnection network. Note that such delays may be accounted for in a program written in this model by standard instructions busy-waiting for a message arrival. Similarly the time for processing this message (which may be proportional to the size of the message) after reception may be accounted for in a program by standard RAM instructions. Similarly, the SEND instruction is also assumed to be non-blocking. The processor executing this instruction simply specifies the message to be sent.

## 2.2 Comparison with APRAM

The expressive power of the CRAM is illustrated in a simulation of an APRAM by a CRAM. The APRAM model used in the simulation is as defined in [6]. A formal instruction level simulation is given in Appendix A. The simulation uses the general notions of simulating shared memory using message-passing primitives. There are two approaches to simulating shared memory on message-passing architectures. The first approach is to view the distributed private memories of the processors as one big shared memory. Whenever a processor needs access to a remote memory location, it sends a message to the processor which owns that location. The remote processor gets interrupted, services the request, and continues with its computation. The second approach is to replicate the shared memory in the private memories of all the processors. All read and writes are now local to each processor. However, on writes each processor is responsible for updating the replicated copies of the location in all the remote processors. The simulation in Appendix A uses the first approach. It is unlikely that a CRAM can be efficiently simulated by an APRAM owing to the non-deterministic primitive in the CRAM. It is not clear whether simulating a non-deterministic branch between any two instructions on an APRAM can be accomplished in less than exponential time.

## 3 Message-Passing Paradigm and an Execution Model

A message-passing architecture is characterized by a loose coupling among the processing elements. Thus an algorithmic paradigm for such an architecture should provide for embodying this loose coupling.

### 3.1 A Message Passing Paradigm

A message-passing algorithm is a *dynamic graph* whose vertices are tasks, and the directed edges emanating from a vertex reflect the set of tasks with which the vertex needs to communicate at the end of the current *computation step*. A computation step is local to each vertex. At any instant, a vertex may non-deterministically choose to either do a computation step or respond to a communication event (Figure 1). In either case the vertex performs some local computation that may lead to changes in the neighbor set of this vertex.

External events are also modeled as vertices in the graph. Note that the current neighbor set of a vertex may change either as a result of computation step or as a result of responding to a communication event. See Figure 2 for an encoding of the message-passing paradigm using the instructions defined in our CRAM model.

This paradigm is general enough to capture formulation of message-passing style algorithms and asynchronous algorithms. In general, problems that map to the above paradigm exhibit the following properties: no global state, sporadic communication between tasks, and a regular communication pattern usually limited to a subset of the tasks. There are several problems that occur naturally in science and engineering that fit this paradigm. See for example Seitz [13], for a concurrent formulation of an N-body simulation problem.

There are problems for which concurrent formulations are possible requiring no explicit synchronization between concurrent threads. Such formulations are referred to as *asynchronous algorithms*. Solution of linear systems of equations, unconstrained optimization, dynamic programming algorithms, shortest path problem, network flow problems, solution of differential equations are some of the problems that fit this category [2]. It is interesting to note that algorithms with explicit synchronization between concurrent threads exist for many of these problems. Such asynchronous algorithms can be expressed as instances of our message-passing paradigm. We illustrate this fact by providing a CRAM program (Figure 4) for the Jacobi algorithm (Figure 3) for the solution of a system of simultaneous linear equations.

An iterative formulation of the Jacobi algorithm is : given an  $n \times n$  matrix  $A$ , an  $n$  vector  $b$ , find an  $n$  vector  $x^*$  such that  $x^* = Ax^* + b$ . A parallel version of the Jacobi algorithm on the PRAM using  $n$  processors is shown in Figure 3. During an iteration  $t$ , each processor  $i$  calculates the new value  $x_i[t + 1]$  using the values  $x[t]$  already calculated in the previous iteration. The solution  $x^*$  is obtained when  $x[t]$  converges to this limit (as  $t$  tends to infinity).

Figure 4 shows an implementation of the same algorithm on the CRAM. Owing to the absence of a globally shared memory, each processor maintains a local copy of the  $x$  vectors. During each iteration, each processor  $i$  updates the  $x_i$  vectors using the values in its local memory. The updated value is then sent to the other processors in the system (Note that the processor does not wait for the updated value to get reflected in other processors). The non-deterministic branch to the communication event during each iteration ensures that a CRAM processor updates its local state using the newly calculated values that may have been sent by other processors. The convergence criterion remains the same as in the PRAM implementation, and a formal proof of the condition under which the criterion holds may be found in [2]. Note that we do not assume that the communication channels between processors preserve the order of the messages transmitted nor do we assume that a processor can determine whether an update received from another processor is older than the corresponding value stored in its local memory.

## 3.2 An Execution Model

An execution model for a paradigm identifies the issues in implementing the paradigm onto a parallel architecture. In our earlier work [15], we developed an execution model for the shared memory paradigm. It is appropriate to re-visit this model to identify a corresponding model for the message-passing paradigm. Programming paradigms [5] for shared memory machines is well understood. An inherent property of shared memory machines is a tight coupling among the processing elements. Correspondingly, the tasks that constitute a parallel algorithm for such machines also reflect this tight coupling. The model of execution is single-program-multiple-data wherein each processor executes the same code on a different portion of the data (data partitioning). The input data is partitioned into chunks and each chunk of this partition is called a *task*. Each processor performs the same set of operations on the task assigned to it. There are four important issues to be considered in this execution model : *scheduling, task granularity, synchronization and communication*.

The assignment of a task to a processor is called scheduling. *Static* scheduling pre-assigns tasks to processors at compile time while *dynamic* scheduling does it at run-time. Dynamic scheduling requires maintaining a global queue of tasks. Task granularity has two dimensions : *computation granularity* and *data granularity*. The former deals with the amount of computation that a processor needs to do for the particular task while the latter involves the size of the data partition in the task. These are important input parameters that need to be considered to determine the effect of task granularity on performance. During the course of execution, a processor may need to synchronize with other processors. Synchronization is achieved through messages in a message-based architecture. A special form of synchronization is the *barrier synchronization* where all the processors participating in the problem need to arrive at a common point in the course of execution before any of them can proceed. Barrier synchronization is used in the model of execution considered here. Tasks in a shared memory algorithm communicate by reading and writing shared memory locations.

Shared memory is the fundamental abstraction of shared memory paradigm. Thus the primary issue that has to be addressed in the execution model for the shared memory paradigm is scheduling (i.e. dividing the problem into tasks and assigning them to processors). On the other hand, the message-passing paradigm is inherently process oriented. Thus the primary issue in an execution model for a message-passing paradigm is the assignment of processes to the available processors, referred to as *process assignment*. There are two approaches to address this issue: In the first approach an allocation of processes to processors is done at compile time and remains unchanged during the execution of the program (similar to static scheduling in shared memory multiprocessors). In the second approach the load at each processor is evaluated periodically during the execution of the program and processes may be migrated among processors to ensure an equitable distribution of load on all the processors (similar to dynamic scheduling in shared memory multiprocessors). Unlike the execution model for the shared memory paradigm, task granularity in the message-passing case has only one dimension, namely, computation granularity. The data granularity dimension is non-existent since there is no shared data. *Process connectivity* refers to the requirement posed by the communication interconnection among processes in the message-passing program. This is an important issue in the execution model since the underlying architecture has to support this requirement. Synchronization and communication are both achieved through the mechanism of message-passing. Another important issue to be considered in the execution model is the size of

the messages (or *message granularity*) exchanged between processors. Thus the execution model for the message-passing paradigm consists of the following issues: process assignment, computation granularity, process connectivity, and message granularity.

## 4 Implementation of the Message-Passing Paradigm

In this section, we discuss preliminary results of implementing the message-passing paradigm on both message-passing as well as shared memory architectures. In the former, the implementation is straightforward. To further simplify the implementation, we restrict the number of processes to be exactly equal to the available number of processors. In the latter case, message-passing has to be simulated using shared memory. This simulation entails two things: establishing a communication channel between processors, and a mechanism for notifying the processors of the arrival of messages. Non-determinism has been built into our message-passing paradigm to capture the asynchrony associated with message arrival. This asynchrony is typically implemented in message-passing architectures via interrupts. In our simulation a two-way communication channel between every pair of processors is established by designating a unique shared memory location. Using these channels, it is easy to model the send and receive primitives. The asynchrony has been simulated using polling. During each iteration, each processor polls all its channels for messages. Any pending messages are processed before continuing with its local computation. Although shared memory machines such as the Sequent usually have interprocessor interrupt capability, we chose not to use it for notification of message arrival since such a feature is really a message-passing feature.

In the shared memory paradigm, the amount of work involved in solving the problem grows as a function of the size of the shared data. Therefore, it is meaningful to discuss metrics such as speedup and completion time for shared memory algorithms since this work is being subdivided among a set of processors. On the other hand, the amount of work involved in a message-passing algorithm is the union of the work represented by the processes cooperating to solve the problem. The amount of work grows as a function of the number of processes in the message-passing algorithm. The notion of speedup would be meaningful in this context only if the number of processes is more than the available number of processors [13]. Since, the focus of our experiments is to study the effect of the message-passing overhead on the performance of the message-passing paradigm we restrict the number of processes to be equal to the available number of processors. The metric used for measuring the performance is the completion time of the algorithm. With respect to the paradigm presented in section 3, in the experiments we assume that the predicate becomes true after 1000 iterations of the repeat loop. The size of the messages considered in the implementation is zero bytes.

In the absence of any message-passing overhead, we would expect the completion time to remain the same for a given task granularity independent of the number of processors. Any deviation from this situation quantifies the effect of the message-passing overhead.

Figures 5 and 6 show the performance of this paradigm for different numbers of processors on the Hypercube and the Sequent. Figure 5 shows the performance of the message-passing algorithm on the hypercube for low computation granularities ranging from 0 to 50 microseconds. For low computation granularities, the message-passing overhead is significant (especially for larger number of processors). For example, with computation granularity zero, this overhead is 90.9% while it is 98.9% with 16 processors. At high computation granularities, this overhead becomes less dominant.

Figure 6 shows the performance of the message-passing paradigm on the Sequent. The com-

pletion times on the Sequent are better than those on the Hypercube for the following reasons: Message simulation on the Sequent (writing to a shared memory location) is much cheaper (order of microseconds) than true message-passing on the Hypercube (1 millisecond for a round-trip message). At such low computation granularities the message-passing overhead dominates the completion time. The size of the messages exchanged is zero bytes. Finally, since the number of processors used in the experiments is quite low (8 on the Sequent, and 16 on the Hypercube), the scalability advantages of the message-passing architecture is not evident. We hope to study the impact of large numbers of processors via simulation.

## 5 Implementation of the Shared Memory Paradigm

There has been considerable effort expended in developing parallel algorithms using the shared memory paradigm. It would be interesting to study the performance implications of implementing these algorithms on message-passing architectures. The fundamental abstraction of shared memory algorithms is shared memory. Thus implementation of such algorithms on message-passing architectures requires simulating the shared memory efficiently. This problem can be viewed as simulating the PRAM by the CRAM, which is presented in Appendix A. In this section, we present experimental results obtained by implementing two shared memory algorithms on the Intel iPSC/2 hypercube, a message passing architecture. The shared memory algorithms that have been taken up for this study - List Ranking and Parallel Prefix - are each representative of a class of problems. List ranking is a fundamental operation on lists wherein the position of each element in the list relative to the end of the list is to be calculated. The algorithm is data dependent and thus the memory access patterns of the algorithm are not known beforehand. The parallel prefix algorithm on the other hand is independent of the input data (data oblivious). Thus, since the access patterns are known in this case, the data could be partitioned so as to reduce the number of accesses.

The Intel iPSC/2 that is used in the implementation is a 16 node hypercube with all nodes being fully connected (when a node sends a message to another, the message goes directly to the receiving node without disturbing any of the other nodes). It uses a circuit-switched message transfer scheme with no store and forward. A round trip message of length zero takes around 1 millisecond.

In implementing the two algorithms on the hypercube, we have taken both approaches of simulating shared memory enumerated in Section 2. In both approaches each processor is allocated an equal chunk to work on. However, these two algorithms have the following characteristics: each processor has to write to only the chunk allocated to it; each processor may only need to read the remote chunks; the reads are only for values written prior to the most recent barrier synchronization point in the program. Given these characteristics it is possible to simplify the simulation of shared memory for the second approach: the locally allocated chunks are exchanged among the processors only at the barrier synchronization point. No messages are needed in the phases between barrier synchronization points. While the second approach limits message exchanges to synchronization points, the size of the messages exchanged may be arbitrarily large especially if the access pattern is not known *a priori*.

The barrier synchronization itself is implemented using a tree structure for communication [8] thus reducing the number of messages needed to achieve synchronization. For barriers implemented with message length zero, this approach takes less than 5 milliseconds for a cube of dimension four. In cases where there is no data exchanged at the barrier the system provided primitive [7] has been used.

The tree barrier works as follows: At the leaf level, the left child processors send their data to their respective siblings. The siblings combine this data with their own and send them as messages to their respective parents. Messages are propagated to the root processor in this fashion. When the root receives the entire data, it propagates down the tree to all the processors participating in the barrier.

Given the considerable overhead involved in maintaining a global queue of tasks in a message-passing architecture, static scheduling is more meaningful. To ensure equitable distribution of the workload at each processor, the data is partitioned into equal size chunks. This chunk of data is allocated locally on the processor. Since we consider only static scheduling in this study, the task granularity has only one dimension, namely computation granularity. The computation granularity is varied by introducing some artificial work in the place where the processor solves the problem for the chunk allotted to it. The artificial work is an idle loop and the loop count is used as a measure of the computation granularity. An advantage of implementing the barrier synchronization primitive using messages is that data could be exchanged among the processors at the synchronization point.

In the discussions that follow, the *completion time* of the program is used as a measure of the performance of the algorithm. It does not include the times taken for process creation and termination and any sequential part that deals with data allocation. Further, in the first approach for simulating shared memory, the time taken for processing interrupts is not included in the overall timings. The results for the uniprocessor case are obtained by running the multiprocessor parallel algorithm on a single processor. Hence the *speedup* using  $n$  processors refers to the ratio of the completion time of the parallel algorithm on 1 processor to that on  $n$  processors.

## 5.1 List Ranking

A parallel algorithm for the list ranking problem [9] is shown in Figure 7. The algorithm is data dependent and thus the access pattern of the list elements is not known *a priori*. The randomness of access of the list elements does not favor data partitioning.

A randomly generated input list of 32K elements is equally partitioned among the available number of processors. After a processor completes processing its chunk of data, it needs to barrier synchronize with the other processors before proceeding to the next iteration of the outermost loop.

### 5.1.1 First Implementation

Since the first approach captures the true semantics of shared memory, the implementation is straightforward. Figure 9 shows the performance of the first implementation on the hypercube, which is almost 20 times worse than that obtained when implementing the same algorithm on a shared memory machine like the Sequent (Figure 12). A roundtrip message is incurred for each external memory request which is far more expensive than a simple memory access on the Sequent. Further there is considerable queueing delay due to the network traffic. Hence the poor performance. By increasing the number of processors, the number of external requests made by a processor decreases. Thus with larger number of processors, the performance improves getting closer to linear speedup.

Similarly, the detrimental effect due to network latency and queueing may be compensated by increasing the computation granularity. However, as seen in Figure 9 this impact is not significant for the computation granularity experimented with.

### 5.1.2 Second Implementation

In this approach, all operations on the list are local to each processor during a phase of the computation delineated by barrier synchronization points. At the barrier synchronization point, the lists are exchanged using the tree barrier, so that they look identical in each processor for the start of the next phase.

Figure 10 shows the performance of the second implementation on the hypercube. The 4 processor performance seems to be the best with no added computation granularity. Increasing the number of processors, increases the available computing power but also increases the time spent at the barrier. Due to the tree barrier, the size and the number of messages exchanged at the barrier increases. Thus when we go from 1 to 4 processors, we get improvement in performance, but after that the synchronization overhead starts dominating. But this effect is compensated by increasing the computation granularity.

Comparing Figures 9 and 10, we note that the second approach fares much better than the first despite the increased size and number of messages exchanged at the barrier, showing that fewer longer messages are preferable to numerous shorter messages.

## 5.2 Parallel Prefix

The algorithm for the parallel prefix problem [9] used in this implementation is shown in Figure 8. The input data of 64K elements is equally allocated among the processors and each processor works on the chunk allocated to it.

Since the algorithm is data oblivious, the first approach to simulating shared memory does not incur any remote memory references. If the second approach to simulating shared memory is used, the amount of information that needs to be exchanged at the end of a phase delineated by barrier synchronization points is merely the prefix sum of the chunk allocated to each processor. Hence the cost for barrier synchronization in the second approach is roughly the same as a normal barrier synchronization without any data exchange. Hence the two approaches to simulating shared memory turn out to be equivalent in implementation cost for this algorithm.

The results for the parallel prefix algorithm in Figure 11 shows an almost linear improvement in performance with the number of processors. This behavior is more pronounced at higher computation granularity. The performance of this algorithm on the hypercube is comparable to its performance on a shared memory machine like the Sequent. See Figure 13 for the performance of parallel prefix on the Sequent.

## 5.3 Discussion

It is interesting to compare the results that we have obtained in the above experiments to those in our earlier study of these two algorithms on the Sequent which is a shared memory architecture. For the list ranking algorithm, we find the performance of both the implementations to be much worse than that on the Sequent. The data dependent nature of this algorithm results in increasing the message overhead. On the other hand, the performance of parallel prefix (which is a data oblivious algorithm) on the hypercube is comparable to that on the Sequent.

In summary, we note that shared memory algorithms are not naturally suited for implementation on message-passing machines.

## 6 Concluding Remarks

The main thrust of our work is to explore message-passing as a vehicle for parallel algorithm development. Since existing models of parallel computation do not capture the salient features of message-passing architectures, a new model (CRAM) was proposed. We presented a programming paradigm and showed its suitability to describing classes of problems that are not a natural fit for the shared memory paradigm. We also presented preliminary results of experimenting with this paradigm on both message-passing and shared memory architectures.

In conclusion we make general observations stemming from our study, and list avenues for further research.

As may be expected, shared memory algorithms perform much better on shared memory machines than on message-passing architectures. In simulating shared memory on message-passing architectures, it may be advantageous to replicate the data and customize its consistency maintenance in the algorithm rather than distributing the shared memory in the private memories of the processors. The overhead in simulating shared memory may be compensated by increasing the computation granularity. Data oblivious shared memory algorithms perform much better on message-passing architectures than data dependent algorithms. The performance of data oblivious algorithms on message-passing machines is comparable to their performance on shared memory machines.

Our preliminary results of experimenting with the message-passing paradigm suggests that it is suitable when (a) the task granularity is comparable to the message overhead, (b) the message-size is fairly high, and (c) when the number of processors solving the problem is fairly large. Our ongoing research includes investigating the effects of the above parameters through experimentation and simulation.

This study suggests several interesting research directions in parallel computation pertaining to the message-passing paradigm. There has been considerable amount of work done in investigating the mapping of processes to processors. Two important issues to be studied are load balancing, and process migration when there are more processes than available number of processors. These issues have been well explored in the context of distributed systems, but to our knowledge have not been addressed for multiprocessors to any great detail. Finally, it is hoped that the CRAM model would serve as a basis for the development and analysis of parallel algorithms using the message-passing paradigm, just as the PRAM model served as the basis for the shared memory paradigm.

## References

- [1] Richard J. Anderson. An Experimental Study of Parallel Merge Sort. Preliminary Version 0.1 . University of Washington, Seattle, 1990.
- [2] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation*. Prentice-Hall Inc., New Jersey, 1989.
- [3] D. Chen, H. Su, and P. Yew. The Impact of Synchronization and Granularity on Parallel Systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 239–248, 1990.

- [4] Richard Cole and Ofer Zajicek. The APRAM: Incorporating Asynchrony into the PRAM Model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.
- [5] D. Gelernter. Generative Communications in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
- [6] Phillip B. Gibbons. A More Practical PRAM Model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.
- [7] Intel Corporation, Oregon. *Intel iPSC/2 and iPSC/860 User's Guide*, 1989.
- [8] D. N. Jayasimha. Distributed Synchronizers. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 23–27, 1988.
- [9] Richard M. Karp and Vijaya Ramachandran. A Survey of Parallel Algorithms for Shared-Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 869–942. North Holland, Amsterdam, 1990.
- [10] Hugh C. Lauer and Roger M. Needham. On the Duality of Operating System Structures. In *Proceedings of the Second International Symposium on Operating Systems*, October 1978. Reprinted in *Operating Systems Review*, (13,2) April 1979.
- [11] D. Lenoski, J. Laudon, K. Gharachorloo, W-D Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.
- [12] Calvin Lin and Lawrence Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II 163–170, 1990.
- [13] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.
- [14] Anand Sivasubramaniam, Umakishore Ramachandran, and H. Venkateswaran. A Computational Model for Message-Passing. In *Proceedings of the Sixth International Parallel Processing Symposium*, pages 358–361, Beverly Hills, California, March 1992.
- [15] Anand Sivasubramaniam, Gautam Shah, Joonwon Lee, Umakishore Ramachandran, and H. Venkateswaran. Experimental Evaluation of Algorithmic Performance on Two Shared Memory Multiprocessors. In Norihisa Suzuki, editor, *Shared Memory Multiprocessing*, pages 81–107. MIT Press, 1992.
- [16] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

```

repeat
  computation step:
    compute locally      /* execute the task */
    communicate with the current neighbors
    reconfigure          /* change the neighbor set if need be */

  communication event:
    receive communication event      /* modify local state */
    reconfigure          /* change the neighbor set if need be */
until (<predicate>)

```

Figure 1: Message Passing Paradigm

```

SNS(Initial_config)
repeat {
  NBR Comp_Step, Comm_event;

  Comp_step :
    Computation();
    SEND(Message);
    Compute_new_config();
    SNS(New_config);
    BR Continue;

  Comm_event :
    if (RECEIVE(message,sender))
      Service_event();
      Compute_new_config();
      SNS(New_config);
    BR Continue;

  Continue :
} until (<predicate>);

```

Figure 2: Message-Passing Program using the CRAM Model

```

In parallel, for each processor  $1 \leq i \leq n$  do
   $x_i \leftarrow \langle \text{initial\_value} \rangle$ ;
   $t \leftarrow 0$ 
  repeat {
     $x_i[t+1] \leftarrow x_i[t]$ ;
     $x_i[t+1] \leftarrow \sum_{j=1}^n a_{ij} x_j[t] + b_i$ ;
     $t \leftarrow t+1$ ;
  } until  $|x[t] - x[t-1]| < \epsilon$ ;

```

Figure 3: A Parallel Implementation of the Jacobi Algorithm on the PRAM

```

In parallel, for each processor  $1 \leq i \leq n$  do
  SNS( $\langle \text{every\_one} \rangle$ );
   $x[0] \leftarrow \langle \text{initial\_value} \rangle$ ;
   $t \leftarrow 0$ ;
  repeat {
     $x[t+1] \leftarrow x[t]$ ;
    NBR Comp_Step, Comm_event;

    Comp_step :
       $x_i[t+1] \leftarrow \sum_{j=1}^n a_{ij} x_j[t] + b_i$ ;
      SEND( $x_i[t+1]$ );
      BR Continue;

    Comm_event :
      if (RECEIVE( $\langle \text{value} \rangle$ , sender))
         $x_{\text{sender}}[t+1] \leftarrow \text{value}$ ;
      BR Continue;

    Continue :
       $t \leftarrow t+1$ ;
  } until  $|x[t] - x[t-1]| < \epsilon$ ;

```

Figure 4: A Parallel Implementation of the Jacobi Algorithm on the CRAM

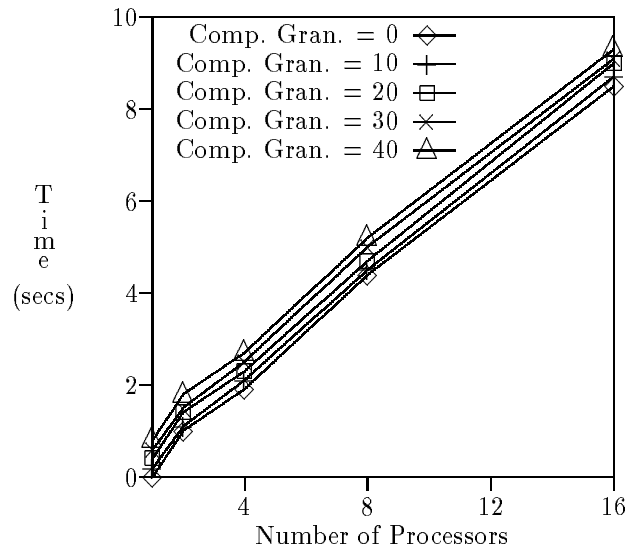


Figure 5: Message-Passing Paradigm - Hypercube

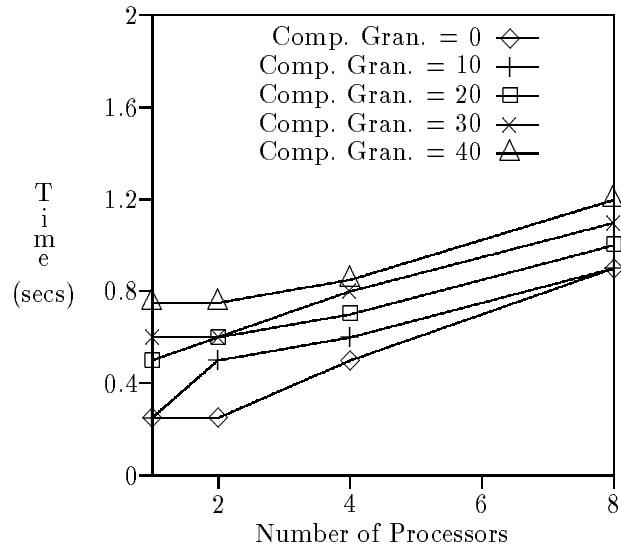


Figure 6: Message-Passing Paradigm - Sequent

```

For  $\log n$  iterations repeat
  In parallel, for  $i := 1, \dots, n$  do
    list[i].rank  $\leftarrow$  list[i].rank + list[list[i].successor].rank;
    list[i].successor  $\leftarrow$  list[list[i].successor].successor;

```

Figure 7: List Ranking Algorithm

```

Prefix( $x, n, s$ )
  If  $n=1$  then  $s_1 \leftarrow x_1$ 
  else
    in parallel, for  $i:=1$  to  $n/2$  do
       $y_i \leftarrow x_{2i-1} * x_{2i}$ 
    prefix( $y, n/2, ss$ )
     $ss_0 \leftarrow$  identity
    in parallel, for  $i:=1$  to  $n$  do
      if  $i$  even then  $s_i \leftarrow ss_{i/2}$ 
      else  $s_i \leftarrow ss_{(i-1)/2} * x_i$ 

```

Figure 8: Parallel Prefix Algorithm

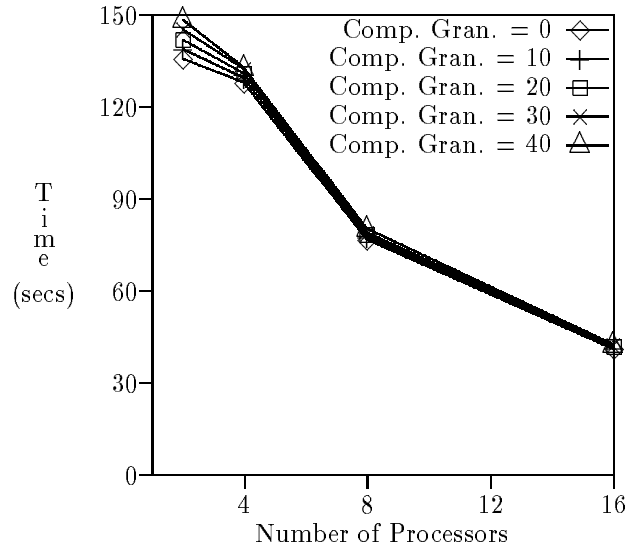


Figure 9: List Ranking - First Implementation, Data Size = 32K

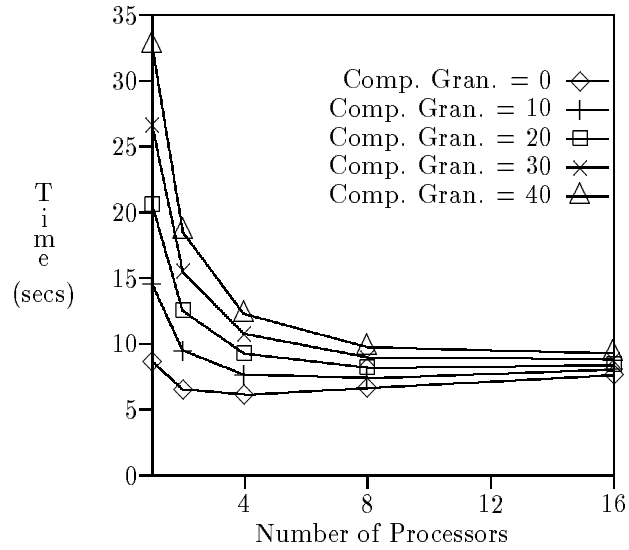


Figure 10: List Ranking - Second Implementation, Data Size = 32K

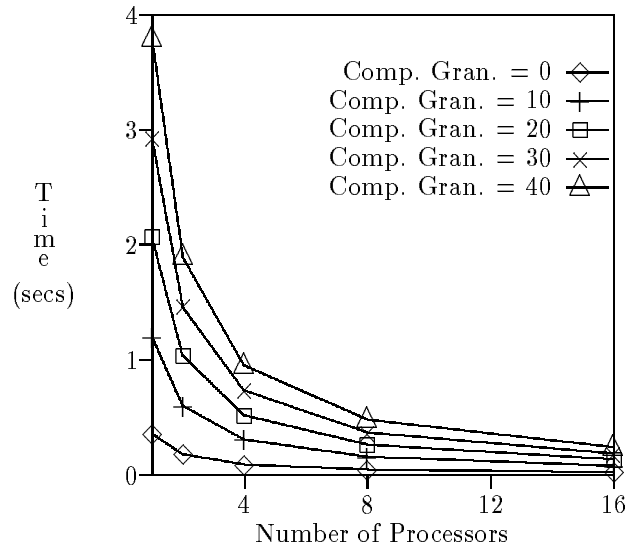


Figure 11: Parallel Prefix - Hypercube, Data Size = 32K

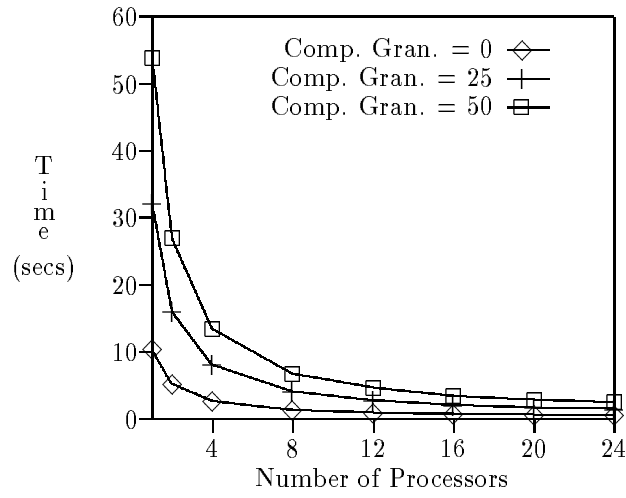


Figure 12: List Ranking on the Sequent, Data Size = 32K

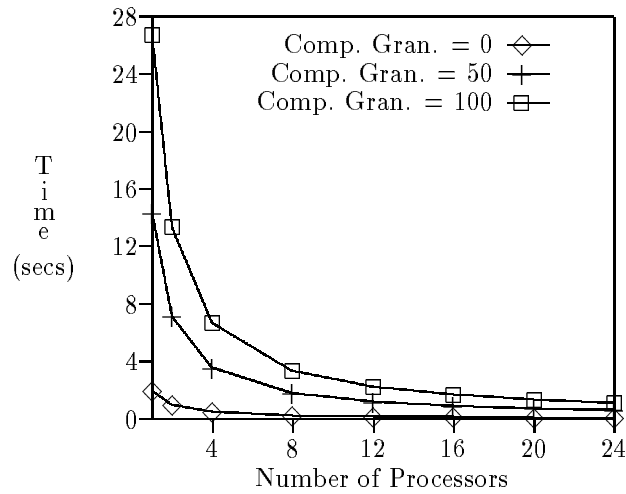


Figure 13: Parallel Prefix on the Sequent, Data Size = 32K

## Appendix

### A Simulation of an APRAM by a CRAM

An APRAM is a collection of loosely coupled processors (RAMs) that can each execute a set of instructions asynchronously with respect to each other. Each processor has a set of local registers  $R_0, R_1, R_2 \dots$  and can also access globally shared memory locations  $M[0] \dots M[max - 1]$ . The instruction set of an APRAM processor is given below. All instructions except 9 are standard RAM instructions. The BARRIER\_SYNC instruction is used to enforce synchronization among the processors during the course of execution.

1. START
2.  $R_{res} \leftarrow \langle constant \rangle$
3.  $R_{res} \leftarrow R_{op1}$
4.  $R_{res} \leftarrow R_{op1} @ R_{op2}$  where @ is a binary operator
5.  $R_{res} \leftarrow M[R_{op1}]$
6.  $M[R_{res}] \leftarrow R_{op1}$
7. BR  $\langle label \rangle$
8. BR  $\langle label \rangle$  IF  $\langle condition \rangle$
9. BARRIER\_SYNC
10. HALT

An instruction-by-instruction simulation of an APRAM on a CRAM is given below. For the simulating CRAM, each processor  $i$  has local memory registers  $R_0, R_1, R_2 \dots$  (which are identical to the APRAM registers), and local memory locations  $M[lo\_mem] \dots M[hi\_mem]$  where  $lo\_mem = (max \div no\_of\_procs) \times i$  and  $hi\_mem = (max \div no\_of\_procs) \times (i + 1) - 1$  (the APRAM's common memory is equally divided among the simulating CRAM processors). Any access by a CRAM processor outside these bounds will result in a memory fault. The instruction set of a CRAM processor includes the standard RAM instructions (all but instruction 9 in the set listed above) augmented by the specialized instructions enumerated in Section 2.

A frequently used macro in the simulation called CSEND is a variation of the original CRAM SEND in that the message is sent to a single destination. It can be easily implemented with the original primitive.

```
Macro CSEND( $\langle message \rangle$ ,  $\langle destination \rangle$ )
    SNS( $\langle destination \rangle$ )
    SEND( $\langle message \rangle$ )
    SNS( $\langle everybody \rangle$ )
End CSEND
```

Another frequently used macro is `Service_Msg` which is an interrupt service routine used for serving remote memory requests. When a CRAM processor needs to access a location corresponding to the APRAM's shared memory that is not local to the CRAM, it sends a message to the CRAM processor owning that location. The `Service_Msg` macro services such remote memory requests (which may be stores or retrieves to the memory locations).

```

Macro Service_Msg
    NBR Retrieve_msg, Store_Msg
Retrieve_msg :
    BR Continue IF !(RECEIVE(< RECEIVE, addr >, sender))
    CSEND(< REPLY, M[addr] >, sender)
    BR continue
Store_msg :
    BR Continue IF !(RECEIVE(< STORE, addr, val >, sender))
    M[addr] ← val
    CSEND(< ACK >, sender)
Continue :
End Service_Msg

```

The macro that implements the APRAM's `BARRIER_SYNC` primitive (instruction 9) is given below. The CRAM processor signals its arrival at the barrier to all the other processors in the system and waits for the others to arrive at the barrier. During the wait, it can go ahead and service any messages that it may receive (this feature prevents deadlocks).

```

Macro Barrier_sync
    < temp1 > ← R0
    < temp2 > ← R1
    R0 ← < no_of_procs >
    R1 ← 1
L1 : SEND(< BARRIER >)
L2 : R0 ← R0 - R1
    BR L6 IF (R0 < 1)
L3 : NBR L4, L5
L4 : Service_Msg
L5 : BR L3 IF !(RECEIVE(< BARRIER >))
    BR L2
L6 : R1 ← < temp2 >
    R0 ← < temp1 >
End Barrier_sync

```

The simulation of the instruction `START` of the APRAM on the CRAM processor can be accomplished as follows :

```

START
SNS(< every_body >)

```

Since the memory access patterns of the APRAM are not known beforehand, the simulating CRAM may need to potentially send a message to any processor in the system to access a memory location. Hence the neighbor set is set to include all the processors in the system.

For an instruction I (where I can be any of instructions 2, 3, 4, 7 or 8) that an APRAM executes, the corresponding CRAM executes :

```

        NBR L1, L2
L1 : Service_Msg
L2 : I

```

The CRAM processor can service a remote memory request (if there is one to be serviced) and then proceed with the actual instruction.

The simulation for the load instruction (instruction 5) is given below.

```

        NBR L1, L2
L1 : Service_Msg
L2 : BR L6 IF ( $R_{op1}$  IN  $\{lo\_mem...hi\_mem\}$ )
        CSEND( $\langle RETRIEVE, R_{op1} \rangle, (R_{op1}/(hi\_mem - lo\_mem + 1))$ )
L3 : NBR L4, L5
L4 : Service_Msg
L5 : BR next_instn IF (RECEIVE( $\langle REPLY, R_{res} \rangle, sender$ ))
        BR L3
L6 :  $R_{res} \leftarrow M[R_{op1}]$ 
next_instn :

```

If the memory address is local to the CRAM processor, it can look it up right away. If not, it needs to send a message to the processor owning that memory location and wait for the retrieved value. During this waiting period, it can go ahead and service other messages (thus avoiding any deadlocks that may otherwise occur).

A similar simulation for the store operation (instruction 6) is given below.

```

        NBR L1, L2
L1 : Service_msg
L2 : BR L6 IF ( $R_{res}$  IN  $\{lo\_mem...hi\_mem\}$ )
        CSEND( $\langle STORE, R_{op1} \rangle, (R_{res}/(hi\_mem - lo\_mem + 1))$ )
L3 : NBR L4, L5
L4 : Service_Msg
L5 : BR next_instn IF (RECEIVE( $\langle ACK \rangle, sender$ ))
        BR L3
L6 :  $M[R_{res}] \leftarrow R_{op1}$ 
next_instn :

```

The simulation for the HALT instruction on the CRAM is slightly tricky since we need to ensure that a processor service all memory requests that have been issued to it or will be issued in the future. Owing to the asynchrony among the processors, a CRAM processor may complete execution

before the others and may thus ignore any future memory requests that the executing processors may issue to it. Using a barrier synchronization before halting (as shown below) prevents such an execution.

```
Barrier_sync  
HALT
```

The above piece of code implements a barrier synchronization and is necessary to ensure that all memory requests are satisfied despite the differing processor speeds.

It is straightforward to present the correctness of this simulation. For each of instructions 2, 3, 4, 7 or 8 that an APRAM may execute, the corresponding CRAM executes the same instruction. The simulation for the HALT instruction ensures that every memory request in the system will be served despite the relative speeds of processors. And, for every memory access that is not local, a message is sent to the processor containing that location (simulations for instructions 5 and 6) which is eventually served. Assuming that message delivery takes zero time and memory requests are immediately serviced, this is an efficient simulation (within a constant running time of the original APRAM).

The simulation presented here is for an EREW APRAM. It can be modified to work for CRCW APRAMs also (changing the `Service_Msg` macro). To capture the concurrent read effect, we will have to check for presence of several messages, accumulate all responses into one long message, and send it all at once (using the `SEND` primitive) to the requesting processors. The concurrent write can be implemented by using some policy (priority, lowest numbered processor wins, etc.) on the incoming requests.