In *Proceedings of the 6th International Parallel Processing Symposium*, pages 358-361, March 1992.

A Computational Model for Message-Passing*

Anand Sivasubramaniam *Umakishore Ramachandran* *H. Venkateswaran*

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

(An Extended Abstract)

1 Introduction

An important motivation for developing a computational model and a programming paradigm 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. The justification for defining two algorithm classes corresponding to the two architecture classes is that certain problems may map more naturally into one class than the other.

The Parallel Random Access Machine (PRAM model) [1] which provides a framework for studying shared memory algorithms, captures only the communication 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. Recently, there have been attempts at incorporating both communication and synchronization into models of parallel computation ([4], [5],[6]). For example, the asynchronous PRAM (APRAM) model introduced in [6, 5] 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.

*This work has been funded in part by NSF grants MIPS-9058430 and CCR-8711749.

This extended abstract develops 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. Experimental results from implementing this paradigm on parallel architectures and the performance implications of implementing shared memory algorithms on message-passing architectures may be found in the full version of this paper [8].

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. 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 [5]. A formal instruction level simulation is given in [8]. 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.

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.

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 [2], 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 [7]. 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 [8].

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 [3], 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 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 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; its

suitability to describing classes of problems that are not a natural fit for the shared memory paradigm and results of experimenting with this paradigm on both message-passing and shared memory architectures may be found in [8].

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] R. M. Karp, V. Ramachandran. A Survey of Parallel Algorithms for Shared Memory Machines. *Technical Report UCB-CSD-88-408.* Computer Science Division, University of California, Berkeley. March 1988.

[2] C. L. Seitz. The Cosmic Cube. *Communications of the ACM.* Vol 28(1):22-33. January, 1985.

[3] A. Sivasubramaniam, G. Shah, J. Lee, U. Ramachandran, H. Venkateswaran. Experimental Evaluation of Algorithmic Performance on Two Shared Memory Multiprocessors. *First International Symposium on Shared Memory Multiprocessing.* Tokyo, Japan. pp. 424-433. April, 1991.

[4] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM.* Vol 33(8):103-111. August, 1990.

[5] P. B. Gibbons. A More Practical PRAM Model. *1st Annual ACM Symposium on Parallel Algorithms and Architectures.* pp. 158-168. 1989.

[6] R. Cole, O. Zajicek. The APRAM : Incorporating Asynchrony into the PRAM model. *1st Annual ACM Symposium on Parallel Algorithms and Architectures.* pp. 169-178. 1989.

[7] D. P. Bertsekas, J. N. Tsitsiklis. Parallel and Distributed Computation. Prentice-Hall Inc. 1989.

[8] A. Sivasubramaniam, U. Ramachandran, H. Venkateswaran. Message-Passing: Computational Model, Programming Paradigm, and Experimental Studies. Technical Report GIT-CC-91/11, College of Computing, Georgia Institute of Technology, 1991.


```
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