Simulation of Parallel Computer Systems on a Shared-Memory Multiprocessor

Rajat Mukherjee and John K. Bennett
Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77251-1892

Abstract

This paper describes an object oriented simulator model for parallel computer systems that is designed for a shared memory multiprocessor and uses the principle of execution driven simulation. We present and evaluate two alternatives for structuring the simulation. The first of these uses a single shared event list object and treats all processes uniformly. In the second, we represent the parallel program model and the target parallel architecture model each as a set of process objects that communicate with each other by means of messages. We take advantage of the available shared memory and examine only the *local* simulation times of the processes involved prior to removing an event from the event list. This method enforces timestamp ordering among all events, thus providing the advantages of global time and a global event list. An important contribution of our work is the integration of two independent concepts: parallel simulation on shared memory and execution driven simulation of concurrent programs.

1 Introduction

Parallel computer systems are becoming an increasingly popular alternative to sequential computing systems as the need for more computing power grows. The development and application of these computer systems require a broad knowledge of the close interactions between parallel programs and the actual parallel machine architectures on which these programs execute. To utilize the available parallelism effectively to realize significant performance improvement over sequential systems, it is necessary to match the structure of the parallel programs with that of the concurrent system on which they are to execute. Accurate and cost-effective performance evaluation techniques are necessary to evaluate how well this matching has been done.

Due to the relatively low accuracy and the difficulty of developing analytical models for the behavior of large parallel systems, performance evaluation techniques for parallel systems depend heavily on simulation. Traditional approaches to simulating sequential computers do not scale well to parallel systems and tend to be architecture-specific.

The Rice Parallel Processing Testbed (RPPT) is a simulation system that has been developed for simulating the execution of a parallel program running on a parallel computer. An RPPT simulation is *execution driven*. It uses a pseudo-concurrent execution of a real program to drive a simulation model of a parallel computer. Execution driven simulation is explained in more detail in Section 3.

The current version of the RPPT runs on a uniprocessor and uses global event list management. This limits the ability of RPPT to perform analysis of large scale parallel computer systems. We are developing a simulator for parallel computer systems that executes on a shared memory multiprocessor. We have found that, because of chronological dependencies between events, a parallel discrete event simulator cannot easily use a global clock or a global event list for synchronization, as is done in the uniprocessor case.

This paper describes an object oriented simulator model for parallel computer systems that is designed for a shared memory multiprocessor. It makes use of *local* simulation times and uses an abstraction of the global event list. It is based on execution driven simulation. We have explored two alternatives for structuring the simulation. The first of these uses a single shared event list object and treats all the processes uniformly. In the second, we represent the parallel program model and the parallel architecture model each as a set of pro-

cess objects that communicate with each other by means of messages. Each set maintains a separate event list. Each event list is defined as a synchronized object that supports mutually exclusive access by the processes it serves. By using separate event lists to represent an abstraction of a logical global event list, we are able to split the system simulation into two logical parts. These communicate through a single well-defined interface object.

In either case, our model requires no prior knowledge of the process interaction patterns, does not use a run-time kernel, and does not use null messages to avoid deadlock. Events are only taken off the event list after examining the local simulation times of the processes involved. This method enforces timestamp ordering among all events, thus providing the advantages of global time and a global event list.

Our approach is currently limited to the simulation of parallel programs using message passing for interprocess communication because the simulator model is based on the message passing paradigm. The simulation of shared memory programs is currently not directly supported.

2 The Rice Parallel Processing Testbed

The RPPT is a performance evaluation tool for studying the execution of parallel programs on concurrent systems. It uses execution driven simulation to accurately describe the data dependent flow of execution in a parallel program and to efficiently provide estimates of user code execution times. Detailed simulation models of the interconnection structure of the system being simulated account for delays in performing remote process synchronizations or remote data accesses. During simulation execution, the timing information from the architectural model is combined with the estimates of instruction execution times to obtain accurate estimates of the overall execution times of the parallel program on the chosen architecture.

There are four basic components in the RPPT software system:

- Concurrent C (CC): An extension of the C programming language obtained by providing a set of procedures that permit a user to create and control processes, and pass information between processes [14].
- C Simulation Package (CSIM): A process level, discrete event simulator implemented by extending Concurrent C with a set of procedures for event queue manipulation and statistics collection [10].
- Architecture Simulation Preprocessor (ASIMP): A software translator that modifies CC programs by inserting simulation primitives used to simulate architecture delays that are due to interprocess communication and synchronization between processes.
- Timing Profiler (TPROF): A program analyzer used to estimate the time required to execute those segments of a CC program that execute sequentially on a single processor, and to modify the compiled version of that program by inserting code used that is used to account for these delays during a simulation.

Figure 1 illustrates the relationship between the basic RPPT components, the user supplied program and architecture specification. First, the CC program is modified by ASIMP, which uses the assignment of procedures to processor modules and the definition of the architecture's interconnection network to insert code that simulates all remote procedure calls and data accesses. This modified program is then compiled by the standard C compiler. The resulting assembly language program is processed by the timing preprocessor TPROF, which counts the instructions between remote accesses and inserts delays corresponding to the time required to execute the intervening code. The modified assembly language program can now be used by CSIM to simulate the program's execution on the specified architecture.

3 Execution Driven Simulation

Three widely used approaches to modeling the workload of a computing system are distribution driven simulation, trace driven simulation and instruction driven simulation [1, 3]. *Distribution driven* simulations use a statistical model of the

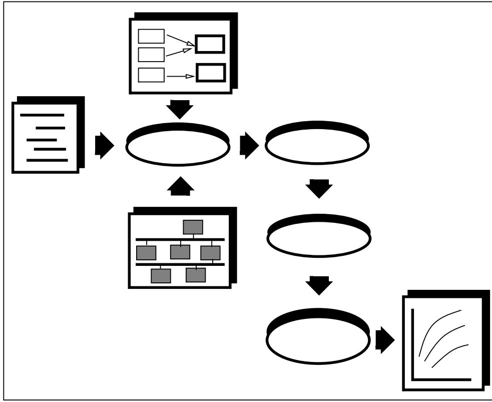

Figure 1: **The Rice Parallel Processing Testbed**

program to drive the simulation. A random process models the generation of data that move between the modules of a system during program execution. It statistically characterizes the way different data affect the resolution of conditional branches. The advantage of this approach is is that the simulation is very fast. The disadvantage is the potential inaccuracy of simulation results.

Trace driven simulations use a program's execution trace to drive a simulation model of an architecture. A trace is first obtained by actually executing the program on a computer that is similar to the one being simulated. As the program executes, the trace is generated by recording one or more distinguishing features of the execution (e.g., a list of the memory locations accessed). This trace is used as the workload for the architecture being simulated. There are two problems associated with this method. First, the simulated architecture must be very similar to that on which the trace was obtained. Second, traces derived from sequential systems are inappropriate for use in simulating a parallel system, since the order of execution is usually dependent on the order in which different parts of a parallel program complete execution. Since a sequential trace represents a fixed order of execution, it does not allow for possible interleaving of different parts of the program.

In *instruction driven* simulation, the execution of each instruction is usually simulated by the execution of several instructions on the simulation host computer. These instructions emulate the movement of data associated with the execution of an instruction. Instruction driven simulation can be quite accurate, but can be very slow, sometimes as much as 200 to 300 times slower than the actual execution of the parallel program [11].

Execution driven simulation is a technique developed at Rice University [7, 9] that speeds up the simulation without losing the accuracy of the results obtained. The simulations are driven by the actual execution of a program. A real program is executed and directly drives a model of the computing system under study i.e., the parallel program is run on the host machine and timings are adjusted by calls to *architecture model* routines so as to reflect the execution times and remote access times of the target machine. The execution of the program and the simulation of the architecture are therefore interleaved. Unlike instruction driven simulation where the execution of each individual machine language instruction is emulated by the execution of many host instructions (even when the host and target machines are nearly identical), execution driven simulation simply executes the machine language instructions of the program directly on the simulation host. Also, while instruction level simulation must update simulation time at least once during the emulation of each instruction, execution driven simulation will only update simulation time when a remote operation is encountered. The difference between execution driven and instruction driven simulations is illustrated in Figures 2 and 3.

Figure 2 shows the basic cycle of an instruction driven simulation. This cycle simulates the activity on a single processor, and each processor in the simulated parallel system will be simulated by a separate, but similar, cycle. The third block in the cycle, where the operation of an instruction is emulated, accounts for most of the large overhead associated with this kind of simulation, since several instructions must be executed by the host computer just to simulate the execution of a single instruction in the target processor. The second block, where the time for the execution of the instruction is accounted for, must be executed once for each simulated instruction. This operation will also require the execution of several instructions to manipulate an event queue on the sim-

ulation host computer.

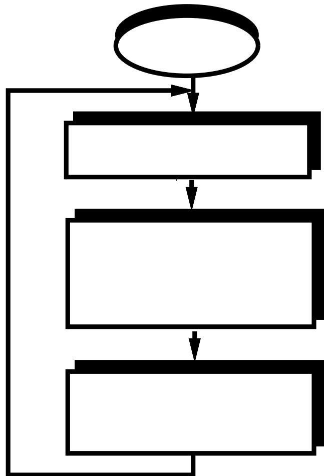

Figure 2: **Instruction Driven Execution Cycle**

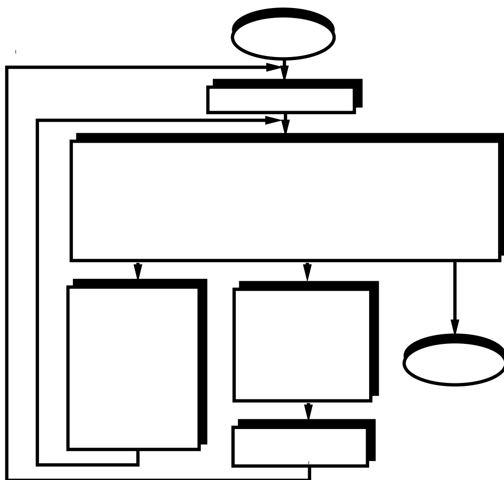

Figure 3: **Execution Driven Execution Cycle**

Figure 3 illustrates the basic execution driven simulation cycle for one of the processors in a parallel system. This is significantly faster than the corresponding cycle for an instruction driven simulation, because it executes the instructions of the target processor instead of simulating their operation, and the number of event queue operations that the cycle performs is much less. There is no overhead associated with an instruction that is not labeled, is not a branch instruction, or does not require interaction with another processor. Those instructions that branch or are labeled incur an overhead to increment a counter **N**. This counter represents the time elapsed in the actual execution of the code between two branch instructions or labeled instructions. Only when a remote operation is encountered is a significant overhead incurred. At that time, an event queue insertion must be processed and the user-supplied program that defines the interconnection network structure must be executed.

The way elapsed time is accumulated between process interaction points in the variable **N** accounts for the high accuracy of execution driven simulation. Since we have incrementing instructions at all possible interaction points, we are guaranteed that the conditional branches of the program are evaluated and executed, so that all the data dependent properties of the program are taken into account during simulation. This gives rise to a very accurate representation of the system's workload. Also, the overhead required to simulate data movement within a processor is avoided.

The sequences of instructions executed in the large box of Figure 3 are called *basic blocks*. Instructions that manipulate the variable **N** are inserted at the start of each basic block of a source program by a utility program called a *profiler*. The profiler detects the basic blocks and for each block, it examines its instructions and uses the information about instruction and memory access timings to compute an estimate of the amount of time required the entire block. This estimate is inserted into the source program in the form of a few instructions on each basic block boundary that increment the timing accumulator **N** by the amount of the estimate for that block.

Since it is not always possible to use a host processor that has the same instruction set as the target processor, a technique called *cross profiling* has been developed. Cross profiling attempts to match up the basic blocks on the host and the target machines and changes the estimates of the basic blocks on the host machine to reflect the estimates on the simulation target machine. Further details of cross profiling may be found in [6, 8].

The principle of execution driven simulation is appropriate for simulating parallel programs even if actual concurrency replaces pseudo-concurrency. The synchronization provided by the user for program correctness and the constraints en-

forced by the parallel simulator, (described in Section 5) are sufficient to make this an accurate and efficient approach to the simulation of parallel programs running on various parallel architectures, using a shared memory multiprocessor as the host.

4 Parallel Simulation

4.1 Sequential Discrete Event Simulation

Typically, the physical system to be simulated is partitioned into a set of component entities called *physical processes* that interact only at discrete times. This system of physical processes can then be simulated as a collection of *logical processes* that communicate via the sending and receiving of *timestamped* messages [2, 4]. Parallel programs (using message passing) are structured in a similar fashion. Thus, the transmission of messages between processes correspond to the events that occur in the simulation.

A global *clock* holds the time up to which the physical system has been simulated, and a global data structure, the *event list*, maintains a set of events, each with its associated time of transmission, that are scheduled for the future. At each step, the event with the smallest timestamp is removed from the event list, and the transmission of the corresponding message in the physical system is simulated, which may lead to changes in the event list. The clock is then advanced to the time of the message transmission that was just simulated.

Memory limitations and execution efficiency cause sequential discrete event simulation to exhibit disappointing performance on large simulations. Execution efficiency can be improved by partitioning the simulation task into several smaller pieces.

4.2 Parallel Discrete Event Simulation

Unfortunately, most simulations cannot be easily partitioned for parallel execution due to *chronological dependencies* between events. Especially in the distributed case, global event lists and global clocks must be eliminated because each host only has local information. Since the intention is to simulate the processes in parallel instead of one at a time, a global clock value has no meaning. Thus, it is necessary to take a new approach to parallel simulation.

In a parallel discrete event simulation, synchronization is required to ensure that no event is simulated until all the events that affect that event have been simulated. Two general approaches to synchronization in parallel discrete event simulation have been previously proposed: *optimistic* (with process rollback [13]) and *pessimistic* (with deadlock avoidance/recovery [15]).

In the optimistic approach, a message is processed as soon as it arrives; however, if a message with an earlier timestamp subsequently arrives, the logical process must *roll back* its state to the time of the earlier message and re-execute from that point. This may require the cancellation of messages that the logical process has sent, which can cause rollback at other logical processes (cascading rollback). This approach requires saving processor state and involves the sending of *anti-messages*[12]. Optimistic parallel simulation requires substantial low-level support in order to be efficient, and the use of general rollback forces a rethinking of many aspects of operating system design, including programming interface, scheduling, message routing and queuing, storage management, flow control, and commitment.

In the conservative approach, the simulation of each message (event) is blocked until it is verified that the event is *safe* i.e. no message with an earlier timestamp can ever arrive. Thus, a logical process can block even though it has pending input messages. Very often, much of this blocking is unnecessary. Worse, the simulation is susceptible to deadlock even if the physical system being simulated is deadlock-free.

One solution to the deadlock problem is to allow the simulation to deadlock, detect it and then recover. However, the simulation is making no progress during deadlock recovery, hence processing power is “lost” during these *phase interfaces* between phases of useful computation [5].

An alternative to deadlock detection and recovery is deadlock avoidance. A typical deadlock avoidance scheme has each logical process periodically sending *null messages* to inform “downstream” logical processes that it will not be send-

ing any real messages before a certain time. Unfortunately, this approach can be expensive, especially when the system under consideration is a parallel computing system, where the number of messages being sent between processes can be very large.

4.3 Shared Memory Parallel Simulation

Most early work on parallel simulation assumed a message-based distributed system host environment. While this may be useful in terms of generality and scalability, there are some drawbacks.

First, each logical process only has access to local information. Second, communication between logical processes that do not reside on the same host results in a physical message being sent on the network. There are conflicting constraints between load balancing and minimization of intra-host traffic by clustering more processes per host (*partitioning problem*).

The use of shared memory can alleviate this problem. One obvious benefit of shared memory is that it makes message passing very fast. This reduces the partitioning problem to a load balancing problem. By using global scheduling policies, such as having a pool of processes ready to run, and allocating any free processor to any process, we can maximize processor utilization.

Although the use of shared memory can make traditional synchronization algorithms run efficiently, completely different approaches can, and probably should, be considered in this environment. Algorithms that are used to synchronize the concurrently running processes must exploit the ability of any component in the system to access the state information of any other component i.e., each process has access to the state of every other process in the simulation. Explicit null messages can be avoided; their transmission can be implied by the modification of certain shared variables.

Synchronization algorithms have been developed for parallel simulation on shared memory. A technique making use of an event list per process has been described by Cota and Sargent [5]. The processes that can generate events that can influence the events of a given process are called its predecessors. Before an event can be taken off the event list of a process, the event lists of all its predecessors must be checked for the *safety* of the event. This method presupposes that the predecessors of each logical process are known at the start of the simulation. However, in a parallel computer program, any process is a potential predecessor of any other process, and the pattern cannot be known beforehand.

Wagner et al. have developed a technique for deadlock avoidance called *lazy blocking avoidance* [16]. Lazy blocking avoidance is executed by processors, which are physical resources, rather than by the logical processes. In contrast to the sending of *null messages*, which consumes processor resources on a regular basis, this technique, which is implemented as part of the run-time kernel, does no work unless some physical processor has nothing better to do.

In the following section, we describe a new approach to parallel discrete event simulation for shared memory. Our approach makes use of local simulation times for each logical process and uses an abstraction of the global event list. This model, combined with execution driven simulation, provides an efficient and accurate method for the parallel discrete event simulation of a parallel program on a specified parallel computer architecture.

5 Execution-driven Parallel Simulation On Shared Memory

Our aim is to simulate the running of a parallel program on a target parallel machine. The simulation is to be carried out on a given host machine. The parallel program that we are trying to simulate is structured as a set of processes that communicate by means of message passing. In our model, the host machine is a shared memory multiprocessor. We would like the simulation to be *execution driven* for reasons of accuracy and speed of simulation i.e., our simulation will involve the actual execution of the parallel program on the host rather than on the target.

In this case, the physical system and the logical system (see Section 4.1) both consist of a set of processes executing in parallel and communicating with each other by sending and

receiving messages. The only difference is that in the physical system, the parallel program is specified to run on the target machine, while the logical system uses the shared memory host machine to run the program. The parallel simulator on the host must account for the differences between the target and the host systems.

There are two issues to be considered when simulating by running the parallel program on the host machine rather than on the target:

1. program correctness, and
2. maintenance of simulation time.

These issues are discussed in the next two sections.

5.1 Program Correctness

The only "events" in the simulation of a parallel program under the execution driven approach are the explicit calls to the message passing routines of the parallel language, when delays must be introduced depending on whether the interacting processes are on the same physical node or on different physical nodes. These are the remote operations that were discussed in Section 3. It is these delays that must be computed by the architecture model of the target machine. In order to ensure program correctness, we must achieve the message ordering on the host machine as it would have occurred if we had executed the program on the target machine. Each process runs independently, except when message passing takes place. Therefore, if the ordering of the messages is correct, overall program correctness is ensured. In the remainder of the paper, the terms message and event will be used interchangeably. The only events on the event list are timestamped messages.

5.2 Maintenance Of Simulation Time

In our simulation model, each process maintains its *local simulation time*. The local simulation time is normally updated via the profiling step (described in Section 3) as the code for the process is executed. The only times when the processes interact are when messages are received. Each message on the event list has a timestamp associated with it. This timestamp is the *expected arrival time* at the intended receiver. This timestamp is calculated from the architecture model specifications. The simulator updates the local simulation time of the receiving process upon receipt of the message.

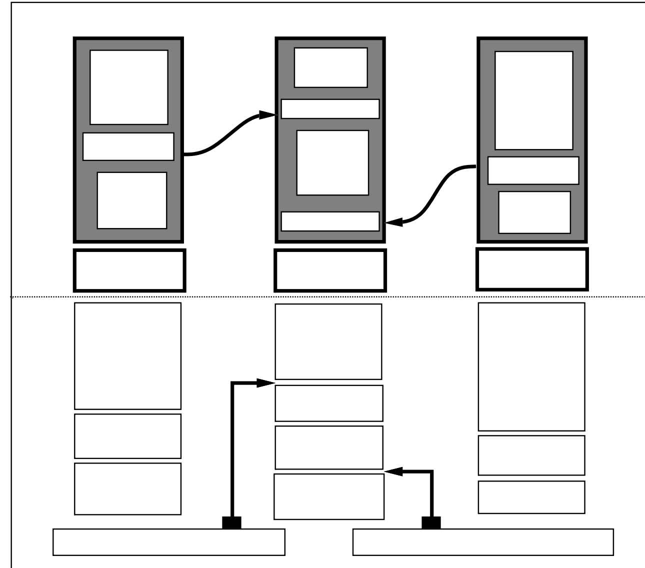

Figure 4: **Maintenance Of Simulation Time**

Figure 4 depicts an example of how simulation times are maintained by several processes. Each process executes sequential sections of code and updates its local simulation time. When a message is received, the local simulation time needs to be updated according to the timestamp of the event. In the example, the message M1 arrives at t = 31s, and Process 2 updates its local time to that value on receiving M1. However, the timestamp of the event corresponding to message M2 is 36, which implies that M2 had already arrived when Process 2 had reached that point in its execution. Thus, the time taken by M2 to traverse the network is not important as far as the local time of Process 2 is concerned. As long as the expected arrival times are generated accurately, our local simulation times are correct.

In order to remove an event from the event list, we must make sure that no other message with a lower timestamp can arrive. The traditional (Chandy-Misra) approach would mean waiting for messages (or null messages) from all other pro-

cesses. Since we are working with shared memory, however, we have access to the local simulation times of all other processes. We can easily check the local simulation times of the other processes to determine whether or not a message with a lower timestamp can arrive. Thus, the principle behind event list management in our simulator is described as follows:

> Given a message with a timestamp value *T* intended for receipt by some process, we can remove this message from the event list (deliver it to the intended receiver) if all other active processes have local simulation clock values greater than *T*.

Figure 5 shows how such a simulation would proceed. Each process maintains its local simulation time. A global event list contains a collection of messages, each marked with the intended receiver process and the expected arrival time. Processes run in parallel until they have to receive a message. At this point they block and wait for the proper message to be removed from the event list. An event list manager process examines the timestamps of the messages in the event list and the local simulation times of the active processes to decide if the messages can be delivered. Once a message is actually delivered, the receiver process unblocks and updates its local simulation time according to the rule displayed in the lowest block in Figure 5. As a practical implementation detail, the event list is maintained in timestamped order.

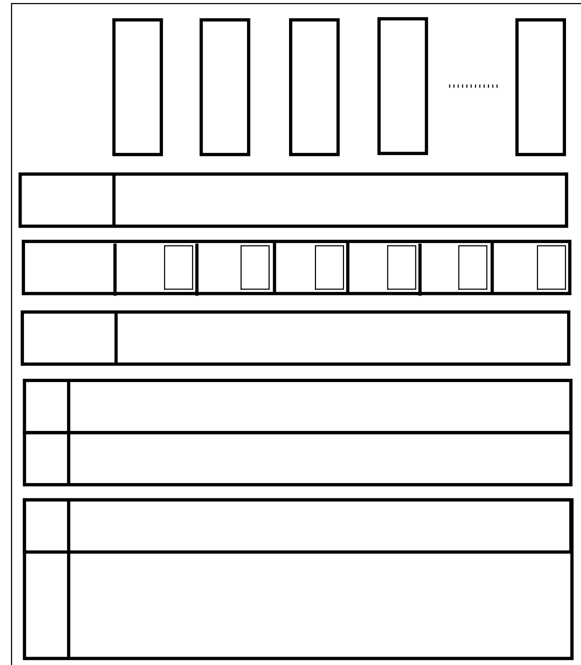

Figure 5: **Simulation Methodology**

6 Architecture Model

The basic structure of our simulator is shown in Figure 6. A single "manager process" removes events from the event list after inspecting the local simulation times of the involved processes. If the communication delays can be modeled by simple distributions, thereby allowing the expected arrival times to be calculated very simply, such a model would suffice. We could project an arrival time at the point when the message is sent and place the timestamped message on the event list to be delivered directly to the intended receiver. However, this may not be true for a complex architecture model where, for instance, the messages may need to be queued for some resource. In such a case, the architecture model itself is specified in terms of a set of communicating processes.

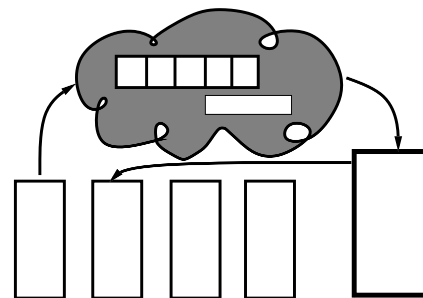

Figure 6: **General Simulator Structure**

Thus, in general, we have two sets of communicating processes, one representing the parallel program and the other representing the parallel target architecture. A preprocessing step changes the original parallel program so that all messages are now sent to an architecture process rather than the actual receiver processes. The architecture model can operate on a message (queue it, delay it, etc.) and compute its arrival time. After being processed by the architecture processes, a message is put back on the event list with the proper timestamp as

a message for the original receiver process.

To extend the simulation model to take into account a number of interacting processes that represent the architecture model of the target machine, we have two basic alternatives:

1. a single event list for all the processes, with all processes treated equally, or

2. one event list for the parallel program processes and one for the processes of the architecture model (and therefore two manager processes).

Simulator models representing these two alternatives are shown in Figure 7 and Figure 8, respectively.

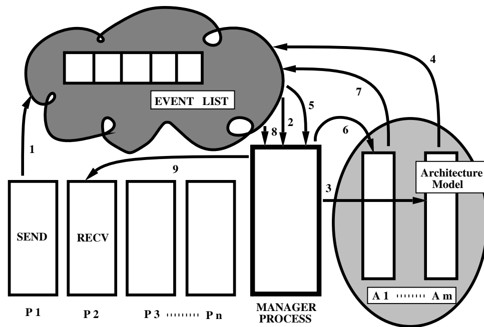

Figure 7: **Simulator With Architecture Processes And 1 Event List.**

Figure 7 shows that the program processes P_1 through P_n and the architecture processes A_1 through A_m are treated equally by the manager. A message sent from P_1 to P_2 is sent first to the architecture model by the manager process, and finally returns to the event list as a message for P_2 after being processed by the architecture. Figure 8 shows two separate event lists. The main manager handles the events for processes P_1 through P_n while the architecture manager handles the events for the architecture processes A_1 through A_m and is also allowed access to the first event list.

The first alternative treats all the processes uniformly. However, the manager must examine the local simulation times of all the processes before allowing an event to be removed from the event list. This makes the manager process a bottleneck.

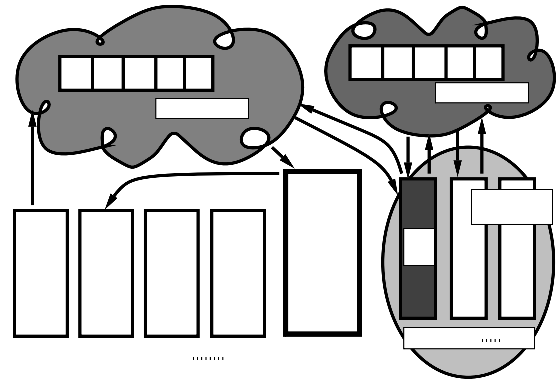

Figure 8: **Simulator With Architecture Processes And 2 Event Lists.**

In the second alternative (depicted by Figure 8), at least one process (called the *architecture manager*) in the architecture model must communicate with both the event lists. Since a message cannot be taken off an event list until it can be ascertained that there is no message being processed that has a lower timestamp, each manager process must maintain a time value that represents the timestamp of the earliest message being processed, and a flag representing the state of the processes it handles. Only these values are examined by the other manager process (besides the local timestamps of the processes it handles) to decide if an event can be taken off an event list.

6.1 Evaluation Of The Two Approaches

The first approach is easier to implement, since only a single global event list object has to be handled. If the expected number of processes in the simulation is small, this method is also efficient, since the manager process only has to examine a small number of local timestamp values. If the host machine has few processors, this approach is the better choice, since the processes of the simulation have a better chance to find an available processor (the manager process runs as an infinite loop). However, debugging such a system can be complicated, because all the processes are treated equally, and an error cannot easily be isolated in either part of the simulation.

In an object oriented environment, the second alternative is not cumbersome to implement. Both event lists are declared

as instances of the same object class. This approach alleviates the problem of the event list becoming a bottleneck in the simulation. If the number of processes in the simulation is very large, or the architecture very complex, the simulation runs faster with the two managers executing in parallel. This approach assumes that we have enough processors, so that the addition of another process is not detrimental to the simulation speed.

In our application, the second approach has some advantages over the first:

- Each event list can be described as a synchronized object, interacting with the other only through a well defined interface.

- Debugging is simplified since the two sets of processes can be monitored independently. This is important, since subtle synchronization errors can easily creep into such a simulation, especially if the architecture model is complex.

- There can be quite a large number of interacting processes in a parallel program. If the architecture specifications are complex, a single event list can easily become a bottleneck. If there are two manager processes running concurrently, each handling a separate event list, the simulation can run faster.

- The specification of the architecture model is more precise, since certain constraints have to be satisfied. Defining the interface precisely simplifies the implementation of an architecture model and also makes it less prone to error.

In a shared memory environment, both event lists are shared data structures. Lightweight processes that use shared memory to implement message passing can be executed in parallel, utilizing as much of the available processing power as possible.

7 Conclusions

The techniques for sequential discrete event simulation do not easily partition for parallel execution due to chronological dependencies between events. There are two major approaches to parallel discrete event simulation, both of which were originally designed for a distributed system setting, where the logical processes only have access to local information. However, with the advent of high speed shared memory multiprocessors, new alternatives for synchronization must be investigated; alternatives that utilize the potential of every process to access the state of any other process in the simulation system.

We have presented an object oriented approach to a parallel discrete event simulation on a shared memory multiprocessor that makes use of local simulation times, and that uses an abstraction of the global event list. Our model does not require prior knowledge of the process interaction patterns, does not use a run-time kernel, and does not use null messages to avoid deadlock. Instead, we take advantage of the available shared memory and examine the local simulation times of the processes involved.

An important contribution of our work is the integration of two independent concepts: parallel simulation on shared memory and execution driven simulation of concurrent programs. We have described how the integration of these concepts can be used to solve the problem of parallel simulation of a parallel program on a computer different from the target machine.

We have described a technique by which the processes that control the simulation can be split into two groups, one group for the processes of the parallel program, and the other for the processes that belong to the architecture model. Each set of processes owns an event list object, and each event list is controlled by a manager process. This simplifies debugging, and results in a clean program-architecture interface. Synchronization is enforced within the program itself, and the correct ordering of messages is enforced by the architecture model, which delays the messages appropriately before they are actually taken off the event list for the original receiver processes.

Our approach is currently limited to the simulation of parallel programs that employ message passing for interprocess communication. Shared memory parallel programs will run correctly on a shared memory host, but our current design will not correctly simulate concurrent programs with global (shared) variables. This is because the simulator model, which is based on the message passing paradigm, simulates the architecture by "delaying" the messages that are transmitted between processes.

We are currently modifying our simulator model to support shared memory. In the sequential simulation, accesses to shared memory can be implemented by simulating *locks* and updating simulation time via accounting routines. In our parallel model, delays due to shared variable access must be introduced either in the form of messages transmitted between architecture processes or by forcing the process that makes the shared access to suspend, and then be awakened by an architecture process that also updates the local simulation time based on the access.

For simulating shared memory parallel programs, we will also have a higher overhead associated with checking data references (to see if they are to shared data). These checks potentially reduce the speedup advantages of execution driven simulation. We are presently extending the execution driven approach to the simulation of shared memory programs on the available sequential simulator to see if the added overhead negates the time savings that are derived from execution driven simulation.

References

[1] T. Axelrod, P. Dubois, and P. Eltgroth. A Simulator for MIMD Performance Prediction: Application to the S-1 MkIIa Multiprocessor. *Parallel Computing*, 1:237–274, 1984.

[2] R. E. Bryant. Simulation of Packet Communication Architecture Computer Systems. Technical Report MIT,LCS,TR-188, Massachussetts Institute of Technology, Cambridge, Massachussetts, 1977.

[3] J.M. Butler and A.Y. Oruc. A Facility for Simulating Multiprocessors. *IEEE Micro*, 6(5):32–44, October 1986.

[4] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–206, November 1981.

[5] A.B. Cota and R.G. Sargent. An Algorithm for Parallel Discrete Event Simulation Using Common Memory. In *The Twenty Second Annual Simulation Symposium*, pages 23–31, Tampa, Florida, March 1989.

[6] R.C. Covington. *Validation of Rice Parallel Programming Testbed Applications*. PhD thesis, Rice University, Houston, Texas, December 1988.

[7] R.C. Covington, S. Dwarkadas, J.R. Jump, G. Lauderdale, S. Madala, and J.B. Sinclair. The Efficient Simulation of Parallel Computer Systems. Technical Report TR 8904, Rice University, March 1989.

[8] R.C. Covington, J.R. Jump, and J.B. Sinclair. Cross-Profiling as an Efficient Technique in Simulating Parallel Computer Systems. In *Proceedings of the IEEE 13th Annual International Computer Software and Applications Conference*, Orlando, Florida, September 1989.

[9] R.C. Covington, S. Madala, V. Mehta, J.R. Jump, and J.B. Sinclair. The Rice Parallel Processing Testbed. Technical Report TR 8610, Rice University, November 1986.

[10] R.G. Covington and J.R. Jump. CSIM User's Manual. Technical Report TR 8501, Rice University, April 1985. Revised February 1986.

[11] So. K. Darema-Rogers, F. George, D.A. Norton, and G.F. Pfister. PSIMUL:A System for Parallel Simulation of the Execution of Parallel Programs. Technical Report RC11674, IBM T.J. Watson Research Center, January 1986.

[12] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot. Time Warp Operating System. *Eleventh ACM Symposium on Operating Systems Principles*, pages 77–93, November 1987.

[13] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[14] S. Madala. Concurrent C User's Manual. Technical Report TR 8701, Rice University, January 1987.

[15] J. Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18(1):39–60, March 1986.

[16] D.B. Wagner, E.D. Lazowska, and B.N. Bershad. Techniques for Efficient Shared-Memory Parallel Simulation. Technical Report 88-04-05, University of Washington, Seattle, Washington, August 1988.