Reducing Synchronization Overhead in Parallel Simulation

by

Ulana Legedza

Technical Report MIT/LCS/TR-655

May 1995

This work was supported in part by the Advanced Research Projects Agency under Contract N00014-94-1-0985, by grants from IBM and AT&T, and by an equipment grant from DEC. Ulana Legedza was supported by a National Science Foundation Graduate Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Reducing Synchronization Overhead in Parallel Simulation

by

Ulana Legedza

Technical Report MIT/LCS/TR-655

Abstract

Synchronization is often the dominant cost in conservative parallel simulation, particularly in simulations of parallel computers, in which low-latency simulated communication requires frequent synchronization. This thesis presents **local barriers** and **predictive barrier scheduling**, two techniques for reducing synchronization overhead in the simulation of message-passing multicomputers. Local barriers use nearest-neighbor synchronization to reduce waiting time at synchronization points. Predictive barrier scheduling, a novel technique which schedules synchronizations using both compile-time and runtime analysis, reduces the frequency of synchronization operations. These techniques were evaluated by comparing their performance to that of periodic global synchronization. Experiments show that local barriers improve performance by up to 24% for communication-bound applications, while predictive barrier scheduling improves performance by up to 65% for applications with long local computation phases. Because the two techniques are complementary, I advocate a combined approach. This work was done in the context of **Parallel Proteus**, a new parallel simulator of message-passing multicomputers.

Keywords: parallel simulation, multiprocessor simulation, synchronization overhead.

This thesis was submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science on May 1, 1995. This thesis was supervised by Professor William E. Weihl.

Acknowledgments

I thank my advisor, Bill Weihl, for supervising my research. His enthusiasm for my work was evident in the invaluable suggestions and insights he gave me. I also appreciate his careful and thorough reading of my thesis drafts.

Special thanks to Andrew Myers, with whom I spent countless hours discussing synchronization and parallel simulation. His questions and comments forced me to think harder about my work, and improved the quality of my research as a result. His picky comments on my thesis drafts were also very helpful.

I thank my sister, Anna Legedza, for taking time out from her own graduate studies to type parts of my thesis. She also provided helpful comments on my writing style.

A number of other people deserve thanks for their help with this thesis: Eric Brewer's technical expertise, numerous ideas, and encouragement helped my thesis work progress smoothly. He also provided Proteus, the simulator on which my work is based. Wilson Hsieh and Anthony Joseph were very helpful in the early stages of my project. They patiently answered all of my questions, and helped me sort through the confusion of running Proteus on itself. Pearl Tsai put up with Parallel Proteus bugs, and discussed various parallel simulation issues with me. Steve Keckler provided useful feedback on computer architecture and allowed me to use his computer for experiments. Jose Robles helped me come up with an analytical performance model and helped me understand and eliminate unsightly spikes in experimental speedup graphs.

I am grateful to all the members of the Parallel Software Group, including Kavita Bala, Carl Waldspurger, Patrick Sobalvarro, Pearl Tsai, Anthony Joseph, and Wilson Hsieh, for creating an enjoyable and intellectually stimulating environment in which to work.

Greg Klanderman has been a true friend to me throughout the last three years. I thank him for his sense of humor, his willingness to listen, and lots of delicious dinners.

I thank my mother, Wira Legedza, for the love, support, and encouragement she has given me throughout my academic career. Her wisdom and strength are a continual source of inspiration for me.

Most of all, I am thankful for the abundant blessings God has given me.

Contents

Chapter 1

Introduction

Simulation technology is useful to both scientists and engineers. Natural scientists use simulators to study the behavior of biological and physical systems. Engineers use simulators to evaluate programs and designs. Traditionally, sequential simulators have been used for this purpose. While they have the advantage of running on an individual's personal workstation, sequential simulators tend to be slow and inadequate for detailed simulation of large systems. First of all, many simulations are compute-intensive, either because the simulated system is being modeled in a very detailed way or because the simulated system is very large. Second, memory limitations of sequential computers limit the size of systems that can be simulated by sequential simulators: a sequential computer can hold only a limited amount of low-latency main memory.

Parallelism can be used to overcome the limitations of sequential simulation. The larger computational resources and memory capacity it provides makes fast, detailed simulation of large systems possible. However, despite its benefits, parallel simulation introduces some new costs. The foremost of these is the overhead cost of synchronization. Synchronization is necessary to ensure correctness in parallel simulation and must be performed at regular intervals. It can often dominate simulator execution time. The problem of synchronization overhead is the subject of this thesis.

Parallel simulation is particularly useful in the field of parallel computing. Because hardware prototypes of parallel computer architectures are time-consuming to build and difficult to modify, software simulators are useful tools both in architecture design and in application development. Parallelism is needed in simulation of multiprocessors in order to accommodate large data requirements of simulated applications, large numbers of simulated processors, and detailed simulation of the processors, memory system and network. Synchronization is a particular problem in such simulators because accurate simulation of the fast networks found in parallel systems requires frequent synchronization. This requirement can be relaxed somewhat by decreasing the accuracy of the simulation, but this is not always desirable or appropriate. For example, accuracy is needed when modeling network contention, which, in turn, is useful in algorithm development. Also, accuracy in network simulation is needed when using simulation to test and evaluate the network or network interface.

This thesis describes local barriers and predictive barrier scheduling, two techniques for reducing synchronization overhead in the simulation of message-passing multicomputers while maintaining high accuracy in network simulation. Local barriers use nearest-neighbor synchronization to reduce waiting time at synchronization points. Predictive barrier scheduling, a novel technique which schedules synchronizations using both compile-time and runtime analysis, reduces the frequency of synchronization operations.

This thesis also reports on the design and implementation of Parallel Proteus, a parallel simulator of message-passing multicomputers. It is based on the fast sequential simulator, Proteus [BDCW91]. Like all sequential simulators, sequential Proteus is time- and space-limited. It is adequate for simulating small and short applications running on machines not larger than 512 nodes (more memory would help capacity but not speed). Large simulations either take too long or run out of memory. Parallel Proteus overcomes these limitations but encounters the problem of synchronization overhead mentioned above. This thesis describes how local barriers and predictive barrier scheduling were incorporated into Parallel Proteus and how effective they are in reducing synchronization overhead.

The following chapter overviews parallel simulation concepts and terminology, and discusses the problem of synchronization overhead in more detail. Later chapters describe the design and performance of Parallel Proteus, present and evaluate the local barrier and predictive barrier scheduling techniques, and discuss related work.

Chapter 2

Parallel simulation overview

This chapter overviews parallel simulation concepts, explains the need for synchronization, and discusses the factors that affect the performance of parallel simulators.

2.1 Definitions

The research presented in this thesis focuses on *discrete event simulation.* This model assumes that the system being simulated changes state only as a result of discrete, timestamped *events.* An event's timestamp corresponds to the time at which it occurs in the simulated system (*simulated time*). Sequential simulators hold pending events in a queue and execute them in non-decreasing timestamp order.

In parallel discrete event simulation (PDES), the simulated system is divided into *entities.* The simulator consists of many *logical processes* (*LP*'s), each of which simulates one or more entities using the discrete event model. The LP's are distributed among the *host processors* executing the simulator. Each LP processes the events relating to the entities it is simulating in non-decreasing timestamp order. An LP is said to be at simulated time `t` when it is about to process an event with timestamp `t`. The LP's do not share any global data structures, but communicate via timestamped *event transfer messages.* Because of communication or connections between components of the simulated system, LP's can generate events that need to be executed by other LP's. These events are transferred from one LP to another via event transfer messages.

In Parallel Proteus, each processor of the simulated or *target* computer is an entity. Each processor of the host machine (CM5) runs an LP that simulates one or more of these target processors. When one target processor sends a message to another target processor, the LP simulating the first processor sends an event to the LP simulating the second processor.

2.2 Synchronization

The main problem in PDES is handling causality errors. A causality error occurs when one entity tries to affect another entity's past. For example, one LP can simulate a processor that sends a message at simulated time `t` to another target processor (simulated by a second LP). In the simulated system, this message is scheduled to arrive at the destination target processor at time `t + q`. This message causes an event with timestamp `t + q` to be sent from the first LP to the second. A causality error occurs if simulated time on the second LP has exceeded `t + q` when it receives this event.

In optimistic PDES, the problem of causality errors is ignored until one occurs. These errors are detected, and rollback is used to correct them. In conservative PDES, causality errors are avoided. Synchronization is used to ensure that when an LP processes an event with timestamp `t`, it will never receive an event with timestamp smaller than `t` from another LP. The overhead cost of synchronization is one reason for the development of optimistic simulation; however, optimistic systems incur large overhead costs for checkpointing and rollbacks. This work focuses on conservative PDES.

How often synchronization is needed depends on the simulated system. If the minimum time for a message to travel between two simulated entities A and B is equal to `q` cycles, then the two LP's simulating them on host processors X and Y, respectively, must synchronize at least every `q` cycles of simulated time. Thus, the simulated time on X will always stay within `q` cycles of simulated time on Y and no causality errors will occur. The minimum value of `q` for the entire system is called the *synchronization time quantum* `Q`. The term *quantum* also signifies the period of time between synchronizations during which simulation work is done.

Periodic Global Barriers

The simplest way to perform the necessary synchronization is with a global barrier executed every `Q` cycles. This technique requires that, upon reaching a synchronization point, each host processor wait for all others to reach the synchronization point and for all event transfer messages to arrive at their destinations before continuing with the simulation. Some parallel machines (such as the Thinking Machines' CM5 and Cray's T3D) provide hardware support for global barrier synchronization. This makes the periodic global barrier approach fairly efficient and easy to implement. However, this method suffers from a potential load imbalance problem. At the end of each quantum, all host processors are forced to wait for the processor doing the most work to simulate that quantum. This waiting time may add non-negligible overhead to each synchronization.

2.3 Performance Considerations

The goal of using parallelism in simulation is twofold. First, the larger amount of memory on a parallel machine makes it possible to simulate very large systems with large memory requirements. Second, the numerous processors of a parallel machine promise a substantial improvement in performance over sequential simulation. However, as in all parallel programs, there are several types of overhead that can limit the speedup. The following equation for speedup illustrates this fact (P_h equals the number of host processors):

$$\begin{aligned} \mathit{Speedup} &= \frac{\mathit{runtime\ of\ sequential\ simulator}\ (= R_s)}{\mathit{runtime\ of\ parallel\ simulator}} \\ &= \frac{R_s}{\frac{R_s}{P_h} + \mathit{event\ transfer\ time} + \mathit{synchronization\ time}}. \end{aligned}$$

One such type of overhead is event transfer overhead, the time required to transfer events from one host processor to another. In general, event transfer overhead is proportional to the amount of communication in the simulated system. It cannot be avoided because event transfer messages are needed to simulate communication in the simulated system. Systems which perform an inordinate amount of communication generate a large amount of event transfer overhead which may outweigh the benefits of parallelism. Clearly, such systems are better suited for sequential simulation.

The second type is synchronization overhead. The percentage of simulation runtime spent on synchronization depends on two synchronization-related factors (frequency and duration) and on two simulation-related factors (amount of detail and number of simulated entities). The frequency of synchronizations is controlled by the synchronization time quantum `Q`. The duration of a synchronization depends on the time it takes to execute the synchronization operation and on the time spent waiting at the synchronization point. Clearly, the lengthier and/or more frequent the synchronizations, the larger the synchronization overhead. However, if the simulation is very detailed, then the time spent synchronizing may be very small compared to the amount of simulation work done, even if the synchronizations are very frequent. In this case, synchronization overhead is not much of a problem. Another situation in which simulation work outweighs synchronization overhead occurs when the simulated system consists of a large number of entities. This happens because each host processor (LP) is responsible for a larger number of entities, and so must do more work between synchronizations. In general, given a fixed number of host processors, speedup increases with the number of simulated entities.

Synchronization overhead is often a problem in simulators of parallel computers with fast networks. Accurate network simulation usually requires the value of `Q` to be very small and, consequently, synchronization to be very frequent. Low overhead techniques for simulating the behavior of each target processor cause synchronization overhead to outweigh the time spent doing simulation work. Large memory requirements for each target processor prevent the simulation of a large number of target processors by each host processor. Consequently, synchronization overhead dominates simulation time. Parallel Proteus is such a simulator. It simulates message-passing multicomputers in parallel. Its level of detail and `Q` are small enough so that synchronization overhead becomes the dominant cost in simulations. `Q` could be increased (frequency decreased) by decreasing accuracy, but this is undesirable. Instead, this work focuses on how to reduce the synchronization overhead while maintaining a high degree of accuracy.

The following chapter describes Parallel Proteus in greater detail and documents its synchronization overhead. Later chapters present and evaluate techniques which I have developed to improve performance both by reducing the frequency of synchronization and by speeding up each synchronization operation.

Chapter 3

Parallel Proteus Overview

This research focuses on simulation of large-scale multiprocessors. Such simulation technology is useful to hardware designers, algorithm developers, and network interface designers. Parallelism allows accurate simulation of realistic data sizes, long computations, fast networks, and machines with thousands of processors.

This chapter describes the parallel simulator I developed, Parallel Proteus. The following sections outline the characteristics of the machines it simulates, techniques used by the simulator, applications used in experiments, and initial performance results.

3.1 Characteristics of the Simulated System

Parallel Proteus simulates message-passing MIMD multiprocessors which are made up of independent processor nodes connected by an interconnection network. Each processor node consists of a processor, memory, and a network chip. The network chip provides the processor with access to the network, and acts as a router in the network. Each processor runs one or more application threads which communicate with threads on other processors via active messages [vECGS92] (or inter-processor interrupts). An active message is a packet of data which travels through the network and causes an asynchronous interrupt upon reception at the target processor. The interrupt triggers the execution of a message handler routine in the context of the thread that was executing on the target processor when the message arrived. The effects of incoming messages can be delayed by temporarily disabling interrupt/message handling on a processor. Applications, which are written in C, call library routines in order to send messages, check for interrupts, turn interrupts on and off, spin on synchronization variables, and manage threads.

3.2 Simulation Techniques

Parallel Proteus is a conservative parallel discrete event simulator. As its name suggests, it is a parallel version of Proteus, the sequential simulator developed at MIT by Brewer and Dellarocas [BDCW91]. Parallel Proteus runs on a Thinking Machines' CM5. This section describes some of the techniques used in Parallel Proteus to provide accurate simulation. Some of these are very similar to or the same as those in sequential Proteus, while others are unique to the parallel version. The similarities and differences will be noted.

3.2.1 Event-driven approach

While sequential Proteus runs on a single processor and has a single event queue, Parallel Proteus runs on an arbitrary number of host processors. Each of these processors manages an event queue for the target processors assigned to it. Each event is marked with a timestamp indicating the time in the simulated system (simulated time) at which the event takes place. The events on each queue are sorted according to timestamp and are processed in non-decreasing timestamp order.

Events are of the following types: thread execution, network, synchronization, and completion checking. Thread execution events involve simulating the execution of application code on the simulated machine. Network events are created when application code sends messages. Processing a network event usually involves scheduling another network event on a different host processor, one that is responsible for a part of the simulated network through which the message must travel. When this happens, this network event is sent over to the other host processor via an event transfer message (implemented using the CMMD/CMAML block transfer mechanism). There, it is inserted into that processor's event queue. Synchronization events ensure that such network events arrive on time.

The simulation ends when all of the event queues on all host processors are empty at the same time and no event transfer messages are in transit. Completion checking events are scheduled in order to help each individual host processor detect this condition. Completion checks synchronize all of the LP's (with a Strata CM5 global OR reduction) to determine the status of each event queue: empty or not empty. If all LP's have run out of events to process, the simulation ends. Otherwise, the simulation continues. Completion checking events are scheduled every 2000 cycles of simulated time. This is not often enough to cause a performance problem.

3.2.2 Direct Execution

Like Proteus, Parallel Proteus uses direct execution to provide low overhead simulation of most application instructions. Application instructions are of two types: *local* and *non-local*. Local instructions are those that affect only data local to the simulated processor on which they are executed. Non-local operations are those that potentially interact with other parts of the simulated system. Examples include message sends, checks for interrupts, and access to data shared by application threads and message handlers. Synchronization variables are an example of such shared data.

Special routines simulate the functionality and concomitant cost of non-local operations. Local instructions, which usually account for most of the instructions in application code, are executed directly. In order to correctly account for execution time of local instructions, application code is augmented with cycle-counting instructions. This is done with the Proteus utility `augment` [Bre92]. `Augment` reads the assembly language version of the application, divides it into basic blocks, and generates augmented assembly language code. Then, the augmented code is compiled into the simulator and run when thread events are processed.

3.2.3 Thread Simulation

Like Proteus, Parallel Proteus simulates a thread executing on a simulated processor at time `t` by allowing the thread to execute up to 1000 local instructions at once without interruption. This 1000 cycle limit is called the *simulator quantum*. While this technique allows target processors to drift apart in time by more than the value of the synchronization time quantum `Q`, it does not necessarily introduce any inaccuracy into the simulation.[1] The reason is that local instructions can be performed at any time and in any order as long as non-local operations are performed at the correct time and in the correct order. To ensure the correct timing and ordering of non-local operations, a thread is interrupted as soon as it encounters a non-local operation. Control then returns to the simulator and the non-local operation is delayed until

[1]Actually, the 1000 cycle simulator quantum *does* produce inaccuracy in sequential Proteus, but this has been fixed in Parallel Proteus.

the simulator has processed all earlier non-local events. This technique allows the simulator to achieve good performance without sacrificing accuracy. (See appendix A for a detailed discussion of the accuracy of thread simulation in Parallel Proteus).

3.2.4 Network Simulation

Parallel Proteus supports both accurate and less accurate network simulation. The accurate version simulates the progress of each packet through the interconnection network (a k-ary n-cube network in this work) hop by hop. At each hop on its path through the simulated network, a packet incurs a minimum delay of two cycles of simulated time (1 cycle for wire delay, 1 cycle for switch delay). When necessary, additional delay is incurred for channel contention. Packets are routed using virtual cut-through routing with infinite buffers. This technique allows simulation of network contention, including hot spots.

When packets move from node to node in the simulated network, the network state of each node in the packet's path must be checked. Because these nodes are likely to be simulated on other host processors, network events must travel from node to node in the host machine as well. In accurate simulation, a network event on host processor A with timestamp `t` can generate a network event for host processor B with timestamp `t + 2` (due to the two cycle minimum network delay). Therefore, the synchronization time quantum `Q` is equal to two cycles of simulated time. Between synchronizations, then, the simulator does only two simulated cycles of work per target processor – a certain performance hit. My research aims to reduce the overhead cost of this synchronization while maintaining the same degree of accuracy.

As an alternative to accurate simulation, Parallel Proteus also provides very fast, but less accurate, network simulation using the constant delay model. All packets, regardless of source, destination, or network traffic on the simulated network, incur a total network delay of 100 cycles. This model, therefore, does not account for network contention. It avoids checking the network state of all intermediate nodes and increases the synchronization time quantum `Q` up to 100 cycles of simulated time. As a result, less accurate simulation in Parallel Proteus requires 50 times fewer synchronizations than the hop by hop model. This technique is also used by the Wisconsin Wind Tunnel parallel simulator [RHL+93].

3.2.5 Synchronization

To avoid causality errors in Parallel Proteus, host processors are synchronized with periodic global barriers. Each global barrier ensures that all host processors have reached the same point in simulated time and that all event transfer messages generated in the last quantum have been received by their destination processors. To this end, global barrier synchronization events are scheduled on each processor at identical points in simulated time. Upon reaching a synchronization event, a processor enters into a global reduction operation, which returns only after all other host processors have entered into it as well. However, all of the host processors may reach the barrier before all of the event transfer messages have arrived at their destinations. To ensure that these messages have been received by the end of a barrier, each host processor keeps track of the total messages it has sent and received during the last quantum. Each processor then computes the difference between these two values, and contributes this difference to a global addition. If this global addition returns any value other than zero, then the network is not yet empty. Each processor repeatedly polls the network and enters global additions until a value of zero is returned. At that point, the network is empty, all host processors have reached the barrier, all event transfer messages have arrived. Then, all processors can safely move on to simulate the next quantum. The following pseudocode illustrates how periodic global barriers are implemented in Parallel Proteus:

```
DO
    Poll_Network;
    difference = msgs_sent - msgs_received;
    result = Strata_Global_Add(difference);
WHILE (result != 0);
```

Implemented using Strata CM5 control network reduction functions [BB94], each global synchronization takes 150 cycles (1 cycle = 30 nanoseconds) for each global reduction performed (usually 1 or 2), plus waiting time.

As in described in the previous section, the synchronization time quantum `Q` for the networks Parallel Proteus simulates is equal to two cycles of simulated time. Therefore, synchronization events are scheduled every two cycles. When the less accurate network simulation module is used, synchronizations are scheduled only every one hundred cycles.

3.3 Applications

The applications considered in this study are *SOR* (successive over-relaxation), parallel *radix* sort, and *simple* (a synthetic application).

3.3.1 SOR

SOR is a classic iterative method for solving systems of linear equations of the form `Ax = b`. For systems of linear equations where `A` is a dense matrix, this algorithm is inherently sequential and therefore does not parallelize well.

However, when the matrix `A` is obtained from the discretization of a partial differential equation, it is sparse in such a way as to make SOR parallelize quite well. In particular, each iteration of the algorithm involves updating each point in a grid of points by performing a relaxation function on each point. The relaxation function requires only the value of the point to be updated and the values of the points adjacent to it in the grid. Subblocks of this grid are distributed among the processors of the parallel machine. During each iteration each processor performs the relaxation function on the grid points assigned to it. For points on the boundaries of subblocks, this requires a processor to obtain the values of some grid points assigned to other processors. Communication is needed for this. Each iteration of the algorithm requires each processor to perform the following steps:

1. send border values to processors that need them
2. compute relaxation function on interior grid points
3. wait until all necessary border values have been received from other processors
4. compute relaxation function on border points.

A more detailed description can be found in [BT].

I used grid sizes of 2^{11} entries to 2^{20} entries, and up to 625 target processors. I set the algorithm to perform 10 iterations. This application alternates between long communication phases and long computation phases.

3.3.2 Radix Sort

The second application I used is a parallel radix sort that uses a counting sort as the intermediate stable sort. This algorithm sorts d-digit radix r numbers in d passes of the counting sort. Pass i sorts the numbers on digit i. The numbers to be sorted are distributed evenly among the processors. Each pass (on digit i) consists of 3 phases: a counting phase, a scanning phase, and a routing phase.

The counting phase involves local computation only. Each processor executes the following algorithm (for pass i):

```
FOR j = 0 to r-1
    count[j] = 0
END
FOREACH number n in processor's data set
    x = ith digit of n
    count[x] = count[x] + 1
ENDFOR
```

This algorithm generates counts for every possible value of a digit (x ranges from 0 to r - 1).

Next, in the scanning phase, each processor enters into r parallel prefix addition scans, contributing count[j] to the jth scan. Scan j returns to processor k the position in the global array of data of the first number in processor k's array of data whose ith digit equals j. From this information, processor k can figure out the position of its other numbers whose ith digit equals j; they come in order right after the first one (because it's a stable sort). From a position in the global array a processor can easily determine which processor and in which position on that processor each data point belongs.

In the routing phase, each processor sends its data points to their new locations (just determined by the scanning phase).

This application has a short local computation phase in the count phase, and then does a great deal of communication in the scanning and routing phases. I used 2-digit radix-64 numbers, requiring 2 passes and 64 scans per pass. 8192 numbers were assigned to each target processor and 64 to 1024 processors were simulated.

3.3.3 Simple

For some experiments, I used a synthetic application called *simple*. This application has characteristics similar to those of parallel iterative algorithms like SOR. *Simple* computes for 10 iterations, each involving a computation phase and a communication phase. In the computation phase, each target processor computes for 30000 cycles. In the communication phase, each target processor sends and receives 10 messages.

Because of its minimal memory requirements and desirable scaling properties, this application is invaluable for demonstrating the performance of Parallel Proteus. In order to observe a full range of performance data, it is necessary to measure the speedup of Parallel Proteus in the case of a large number of target processors. This requires sequential Proteus performance data. However, the memory requirements of *radix* and *SOR* limit the number of processors that can be simulated on sequential Proteus. It was possible to simulate *simple* running on many target processors because it takes up very little memory. Therefore, speedup could be measured for the case of many target processors.

3.4 Performance

In order to evaluate the performance of Parallel Proteus, I ran experiments on 32 and 128-node CM5 partitions. Each CM5 node is a 33 MHz Sparc processor with 32 megabytes of RAM and no virtual memory. Speedup was measured by comparing the performance of Parallel Proteus with that of sequential Proteus running on a 36 MHz Sparc10-30 processor with 64 megabytes of RAM. By running a series of different sequential programs both on the Sparc10 and on individual CM5 nodes, I determined that programs ran approximately twice as fast on the Sparc10 as on a CM5 node. Therefore, in order to make a fair comparison of the performance of Parallel Proteus with sequential Proteus, I scaled the runtimes obtained on the Sparc10 by a factor of two.

The memory limitations of the Sparc10 prevented me from obtaining measurements of the sequential runtimes of large simulations (those involving a large number of target processors and/or a large data set). Therefore, some of the sequential runtimes used for speedup computations were obtained by extrapolating from the sequential runtimes of smaller simulations of the same application. The extrapolated values are very conservative estimates of the actual sequential runtimes of these large simulations because they do not account for the amount of time that would be spent in paging activity.

In order to get accurate timings and decrease variability, CM5 experiments were run in "dedicated" mode. This means that timesharing is turned off, and each application runs to completion without being interrupted to run other applications. The only source of variability, therefore, is in the routing of messages through the CM5 host network. In practice, this variability was very small. Each CM5 experiment was run 3 times. The data presented in this chapter (and in chapter 6) are averages ($\overline{X}$) of three timings obtained in these runs. The standard deviations (σ) were also computed. For all of the experiments, $\sigma/\overline{X}$ was $\leq 4\%$.

The following sections describe three aspects of Parallel Proteus performance:

1. effective simulation of realistic problem sizes;

2. performance under the less accurate network model; and

3. performance under the accurate network model.

Because the question of interest is how to provide fast, accurate simulation, emphasis will be placed on the performance of the accurate model.

3.4.1 Large simulations

The purpose of using parallelism in simulation is not only to speed up simulation but also to simulate larger systems and data sizes than can be done on uniprocessors. A uniprocessor, after all, can hold only a limited amount of main memory. While this limit can be large and is growing, there will always be problem sizes which require more memory than is available. Also, in most general-purpose computing environments, it is often more cost-effective to spread a large amount of memory over many processors than to put a very large amount of memory on one processor. In both these cases of uniprocessor memory limitations, parallelism can be used to accommodate large memory requirements; a parallel system with P processors can have P times more memory than a uniprocessor. In this section, I compare the capacity of the present CM5 implementation of Parallel Proteus with that of sequential Proteus running on a Sparc 10. While both the Sparc 10 uniprocessor and the CM5 processors were equipped with only relatively small amounts of main memory, the comparison demonstrates the benefit of scalability that parallel simulation provides.

The following comparison is not completely fair to Parallel Proteus because the uniprocessor running sequential Proteus has 64 megabytes of RAM and virtual memory support, while each

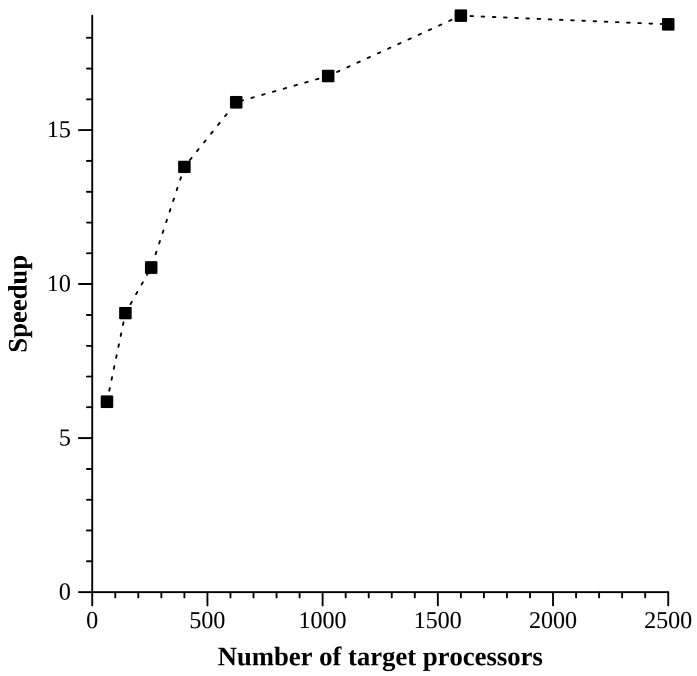

Figure 3-1: **Less accurate network simulation.** Speedup under constant delay network model (delay = 100 cycles) for *simple*, 32 host nodes.

processor of the CM5 has but 32 megabytes of RAM and no virtual memory support. Despite these limitations, however, Parallel Proteus is indeed able to simulate larger systems (without running out of memory) than can sequential Proteus. (While sequential Proteus runs extremely slowly when simulating large systems and data sets because of virtual memory thrashing, I still consider it "able" to simulate these sizes as long as it does not completely run out of memory).

The Sparc 10 processor used for sequential experiments can simulate up to 1600 target processors running *simple*, and up to 512 target processors running *SOR* on a 1024 x 1024 grid (but with a lot of paging activity). Parallel Proteus does much better. Running on 32 host processors, Parallel Proteus can simulate up to 8192 target processors running *simple* and up to 625 target processors running *SOR* on a 1024 x 1024 grid. Also, Parallel Proteus running on 32 host processors can comfortably simulate 64 target processors running *SOR* on a 2048 x 2048 grid, a configuration too large for sequential Proteus. Virtual memory support and more RAM on each node would increase the capacity of Parallel Proteus.

3.4.2 Less accurate network simulation

This section presents performance results for Parallel Proteus using the constant delay

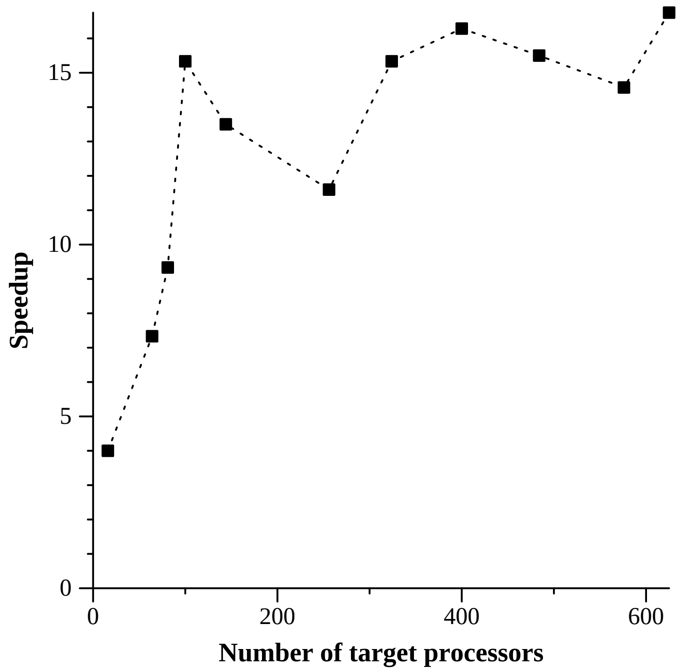

Figure 3-2: **Less accurate network simulation.** Speedup under constant delay network model (delay = 100 cycles) for *SOR*, 64 x 64 grid size, 32 host nodes.

network model and periodic global barrier synchronization. The performance metric used is speedup, which indicates how much better Parallel Proteus performs compared to sequential Proteus. The graphs in figures 3-1 and 3-2 illustrate that speedup increases very quickly with the number of processors being simulated. The speedup is greater than 15 for 625 target processors in *simple* and for 100 target processors in *SOR*. (*SOR*'s speedup curve is not as smooth as *simple*'s because the simulation work required for *SOR* does not scale as well with the number of target processors). In these applications, synchronization overhead accounted for 5% to 65% of total simulator runtime. As will be shown in the next section, this is small compared to the synchronization overhead (70% to 90%) observed under accurate network simulation, where synchronization is more frequent. This difference in synchronization overhead is the reason why the simulator performs better under the less accurate, constant delay network model.

3.4.3 Accurate network simulation

This section describes the performance of Parallel Proteus using the exact network model and periodic global barrier synchronization. Speedup, simulation slowdown, and synchronization

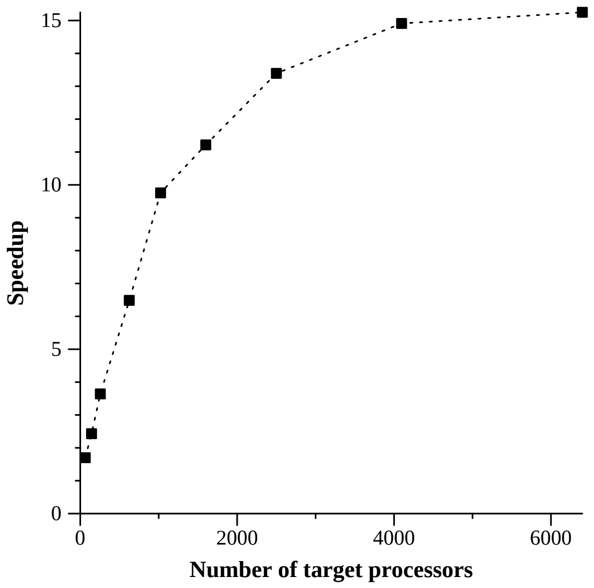

Figure 3-3: **Accurate network simulation.** Speedup vs. number of target processors for *simple*, 32 host nodes.

overhead were measured. The data show that performance is much worse than that for inaccurate simulation due to the very frequent synchronization required for accurate simulation. Although the synchronization overhead can be amortized over many target processors, realistic data sizes often preclude the simulation of a large number of target processors.

Speedup

The data show Parallel Proteus achieves greater speedups when simulating a large number of target processors, as expected. With a large number of simulated processors, more work is done between synchronization operations and synchronization thus accounts for a smaller percentage of overall runtime. This results in better performance. Overall, however, speedup under accurate network simulation is much worse than the speedup achieved under less accurate network simulation.

Figures 3-3 and 3-4 show speedup curves for *simple* for 32 and 128 host processors, respectively. Both graphs show speedups increase with the number of target processors being simulated, with the highest speedup achieved when each host processor simulates about 128 target processors. In both graphs, the highest achieved speedup is about 50% of the maximum

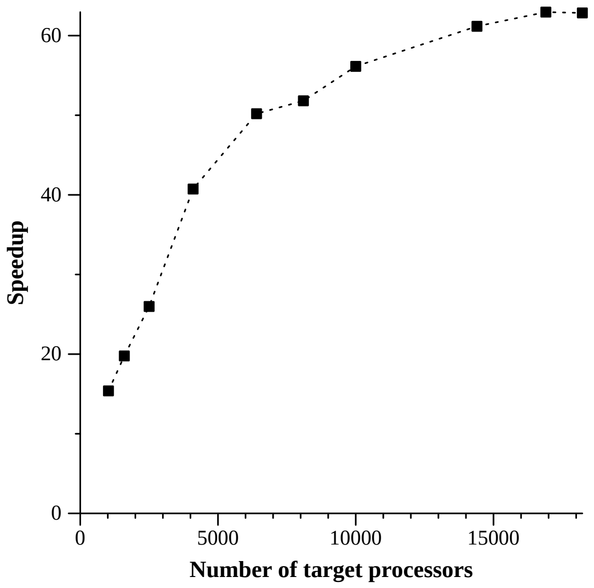

Figure 3-4: **Accurate network simulation.** Speedup vs. number of target processors for *simple*, 128 host nodes.

---

possible speedup (maximum speedup = number of host processors).

The failure to achieve more than 50% of the maximum speedup is primarily due to the cost of enqueueing events. This cost is higher in Parallel Proteus than in sequential Proteus because the parallel simulator must do more work to achieve determinism and repeatability. Parallel Proteus carefully orders events with identical timestamps in order to overcome the non-determinism introduced by the host machine's network. The time it takes to do this depends partly on the number of events in the queue. When there are more target processors, the number of events in each host processor's queue increases. Consequently, the time to enqueue an event increases, as does the percentage of total runtime spent enqueueing events. This increase in enqueueing cost limits the speedup that can be achieved with a large number of target processors. However, this speedup remains high compared to the speedup for a small number of target processors.

The difference between the performance achieved under accurate network simulation and that achieved under less accurate network simulation can be observed by comparing figures 3-1 and 3-3. Figure 3-3 shows that in the case of 32 host processors *simple* running on 625 target processors only achieves a speedup of 6. This is small compared to 16, the speedup achieved in the same case under the constant delay model. In accurate network simulation, speedup does

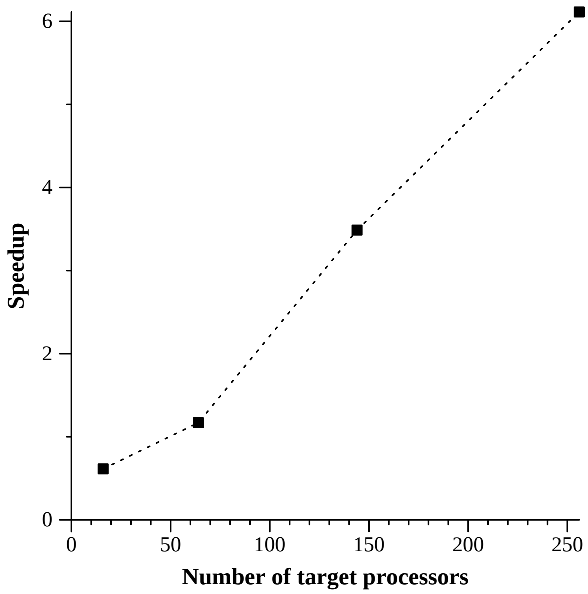

Figure 3-5: **Accurate network simulation.** Speedup vs. number of target processors for *SOR*, 1024 x 1024 grid size, 32 host nodes.

---

not reach 15 until 6400 target processors are simulated.

The speedup graphs for *SOR* and *radix* can be seen in figures 3-5 and 3-6. These graphs also show speedup increasing with the number of target processors, but the speedups achieved are not nearly as high as those in *simple*. This is due to two factors. The first factor is large memory requirements. Both *SOR* and *radix* consume a significant amount of memory in order to store their large data sets. *Radix* also requires a large amount of memory in order to store the state associated with its numerous threads. Because these large memory requirements preclude the simulation of a large number of target processors, synchronization overhead still dominates overall runtime. The second factor is the larger percentage of total runtime spent transporting the event transfer messages caused by the large amounts of communication in *radix* (in particular) and *SOR*.

Again, as in *simple*, speedup under accurate network simulation is not as high as it is under the constant delay model. Although the data are for different grid sizes, comparing figures 3-5 and 3-2 shows that under less accurate network simulation, *SOR* on 256 target processors achieves a speedup of 15, while under accurate network simulation, it achieves a speedup of only 6.

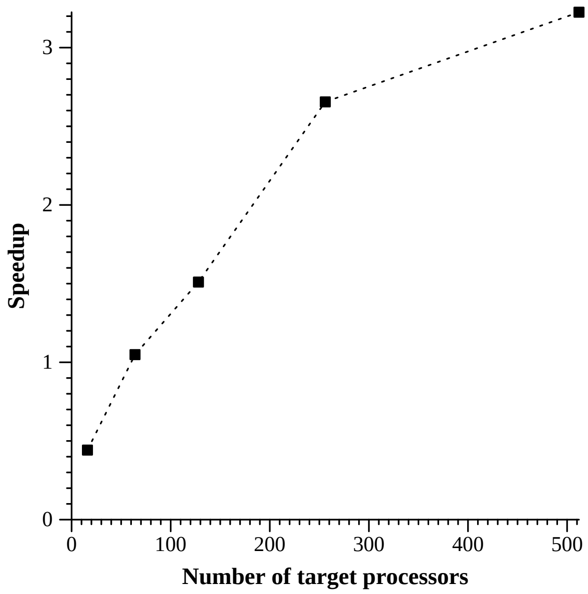

Figure 3-6: **Accurate network simulation.** Speedup vs. number of target processors for *radix*, 8192 numbers per target processor, 32 host nodes.

Slowdown

Simulation slowdown measures how much slower the simulator is than the simulated system. It quantifies the efficiency of the simulator - the lower the slowdown, the higher the efficiency. There are several ways to measure slowdown.

The first measure is *absolute slowdown*, the ratio of the time it takes to run a program on the simulator to the time it takes to run on a real machine:

$$absolute\ slowdown = \frac{runtime\ of\ Parallel\ Proteus\ in\ cycles}{number\ of\ simulated\ cycles}.$$

While absolute slowdown indicates how much slower the simulator is than the simulated system, it does not take into account the discrepancy between the number of host and target processors. Therefore, it is not a fair measure of the efficiency of the simulator.

Host node slowdown per target processor is a better measure of parallel simulator efficiency, taking into account both the number of simulated entities and the number of host processors. It measures the efficiency of each individual host processor in a parallel simulator. It depends not only on the number of simulated processors or on the number of host processors, but on the load assigned to a host processor (the number of target processors per host processor). As seen in the previous section, Parallel Proteus performs better with larger loads. The slowdown

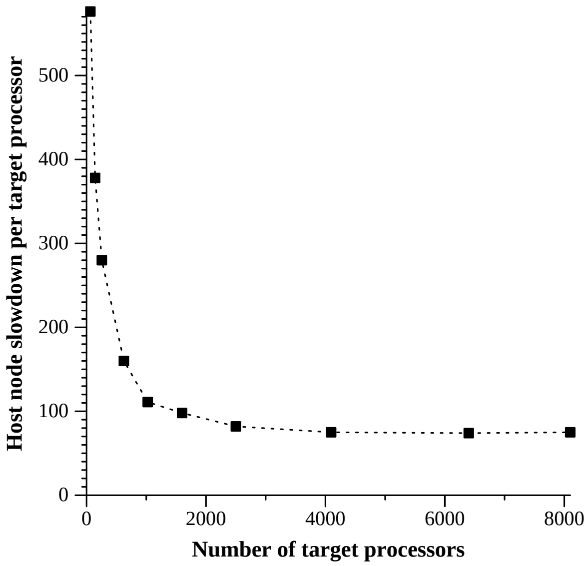

Figure 3-7: **Slowdown.** Host node slowdown per target processor vs. number of target processors for *simple*, 32 host nodes.

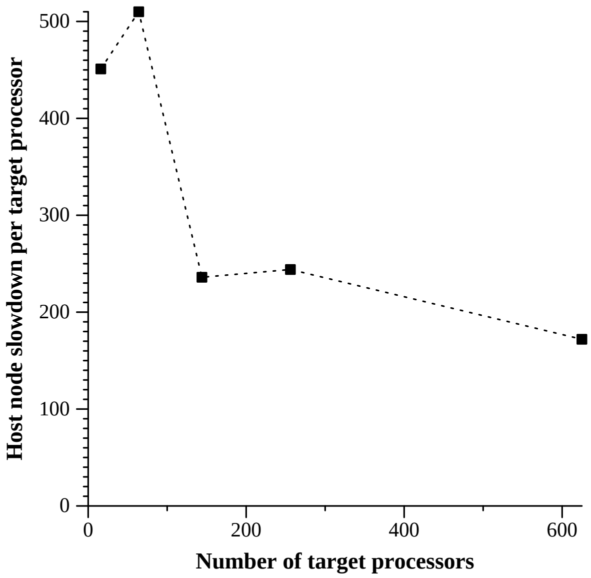

Figure 3-8: **Slowdown.** Host node slowdown per target processor vs. number of target processors for *SOR*, 1024 x 1024 grid size, 32 host nodes.

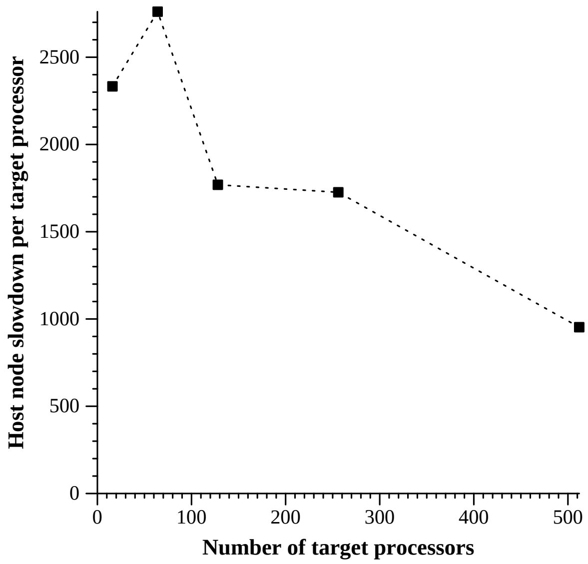

Figure 3-9: **Slowdown.** Host node slowdown per target processor vs. number of target processors for *radix*, 8192 numbers per target processor, 32 host nodes.

is expected, therefore, to be lower for larger loads. *Host node slowdown per target processor* is calculated as follows:

$$\textit{host node slowdown per sim processor} = \frac{\textit{runtime of Parallel Proteus in cycles}}{\textit{number of simulated cycles} * \frac{\textit{number of simulated processors}}{\textit{number of host processors}}}.$$

Figure 3-7 shows the host node slowdown per target processor for *simple*. As expected, the slowdown decreases as the number of target processors increases. Again, this happens because the synchronization overhead is amortized away. *SOR* and *radix* exhibit similar behavior (see figures 3-8 and 3-9). Since the necessary simulation work does not scale as well with the number of target processors as it does in *simple*, these graphs are less smooth. *Radix* slowdown remains large even for 512 target processors. This is due to the large overhead of transporting event transfer messages which are needed to simulate heavy communication traffic in *radix*.

Synchronization overhead

This section presents measurements of synchronization overhead as a percentage of total simulator runtime. Figures 3-10, 3-11, and 3-12 graph synchronization overhead (as a percentage of total runtime) for *simple*, *SOR*, and *radix*. Since *simple* has minimal memory requirements, it allows many target processors to be simulated on each host node, which causes synchronization overhead to become only a small percentage of overall runtime. *SOR* and *radix*, on the other

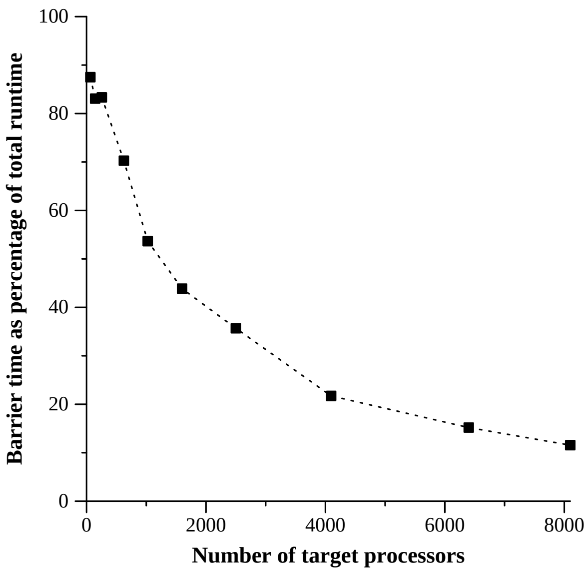

Figure 3-10: **Synchronization overhead** vs. number of target processors for *simple*, 32 host nodes.

---

hand, have large memory requirements. Therefore, fewer target processors can be simulated on each host node and synchronization overhead remains a very large percentage (80%) of total runtime.

3.4.4 Discussion

In general, synchronization overhead is the dominant cost in accurate simulation of multiprocessors due to the high frequency with which it must be performed. Data presented in the previous sections show this is particularly evident in Parallel Proteus. As seen in *simple*, this synchronization overhead can be amortized away when simulating a large number of target processors. However, it is not always possible to do so in an application with realistic data sizes. For example, in *radix* and *SOR*, a large number of target processors cannot fit on each host processor due to memory shortage. Synchronization overhead, therefore, remains a large percentage of total runtime. While memory shortage can be alleviated by equipping host nodes with virtual memory and more RAM, there will always be applications which use up this memory for more data rather than for more target processors.

In the following chapters, I will present techniques for reducing this synchronization overhead in accurate simulations. The first technique will speed up synchronizations by reducing

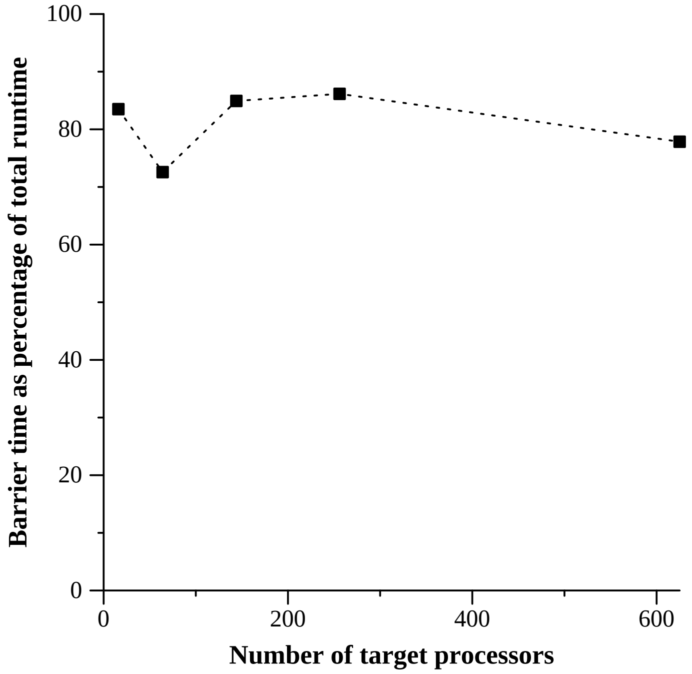

Figure 3-11: **Synchronization overhead** vs. number of target processors for *SOR*, 1024 x 1024 grid size, 32 host nodes.

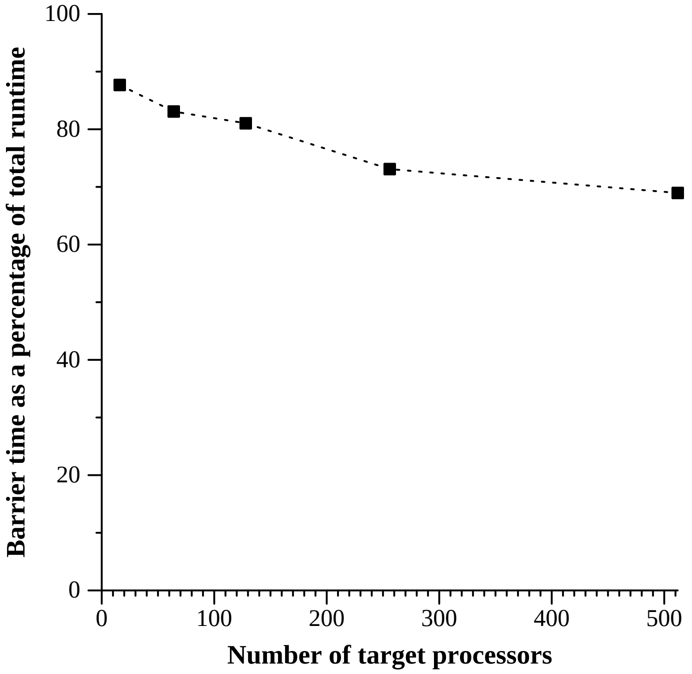

Figure 3-12: **Synchronization overhead** vs. number of target processors for *radix*, 8192 numbers per target processor, 32 host nodes.

the waiting time at each synchronization operation. The second will improve simulation performance by reducing the number of synchronizations performed.

Chapter 4

Description of synchronization techniques

This chapter describes several alternatives to periodic global barriers: local barriers, fine-grained local barriers, predictive barrier scheduling, and barrier collapsing. Local barriers use nearest-neighbor synchronization to reduce waiting time at synchronization points, while predictive barrier scheduling and barrier collapsing reduce the frequency of synchronization operations.

4.1 Nearest Neighbor Synchronization

In many simulations, a host processor must keep within `Q` cycles of the simulated time of only several other processors - its nearest neighbors. The following two techniques use nearest-neighbor synchronization in an attempt to reduce synchronization overhead by reducing waiting times at barriers.

4.1.1 Local Barrier

In a local barrier, the nearest neighbors of host processor X are those host processors that simulate entities that, in the simulated system, are directly connected to any of the entities simulated on processor X. Each host processor participates in a barrier with these nearest neighbors. Once a host processor and its nearest neighbors have reached the barrier, the host processor can move on to simulate the next quantum. [1] However, its nearest neighbors may still

[1] A processor's nearest neighbors are not necessarily the processors closest to it on the host machine; they depend on the system being simulated and its layout on the host machine.

be left waiting for their own nearest neighbors to reach the synchronization point. Therefore, some host processors may be done with a barrier while others are still waiting.

Loosely synchronizing the processors in this way (as opposed to keeping them tightly synchronized as in periodic global barriers) will shorten barrier waiting times in the situation where the simulation work is not equally distributed among the host processors during a quantum. Only a few processors will have to wait for each slow (heavily loaded) processor. Therefore, unless the same processor is heavily loaded during every quantum, the simulation will not be forced to be as slow as the most heavily loaded processor in each quantum. Instead, some lightly loaded processors will be allowed to go ahead to the next quantum while heavily loaded processors are still working on the previous quantum; the heavily loaded processors will catch up in a future quantum when they have less simulation work to do.

While this technique may alleviate the problem of waiting, it introduces a new problem of software and communication overhead involved in performing local barriers in the absence of hardware support.

4.1.2 Fine-grained Local Barrier

When a large number of entities of a simulated system are mapped to one host processor, the processor may end up having a large number of nearest neighbors, again raising the problem of processors idling while waiting for others to arrive at a barrier. A fine-grained local barrier, in which a host processor continues processing events of some entities while waiting on synchronizations for others, may alleviate this problem. A fine-grained local barrier involves separately synchronizing each entity. For example, one host processor simulating 20 entities can be viewed as having 20 LP's, each simulating one entity. Each LP participates in its own local barriers, which now involve a smaller number of neighbors (some or all of these neighbors may be other LP's on the same host processor). Since some of a host processor's LP's can do work while others are waiting on a barrier, it is less likely that a host processor will ever be idle. However, this technique may result in prohibitive simulation overhead for scheduling and synchronizing the virtual LP's.

4.2 Improving Lookahead

With a small synchronization time quantum `Q`, synchronizations must be performed frequently. Therefore, the total time to perform synchronization operations will be a large percentage of total simulation time no matter what synchronization scheme is used. However, during periods of the simulation when the simulated entities do not communicate, these synchronization operations are unnecessary. It is only necessary to synchronize when communication is taking place, to make sure it happens correctly. Therefore, it may be useful to eliminate synchronization altogether during local computation periods. This requires that all host processors be able to predict when communication is going to occur in the simulated system and then agree on a time at which to perform the synchronizations.

The ability to predict the future behavior of the simulated system is called *lookahead.* A simulator is said to have lookahead `L` if at simulated time `t` it can predict all events it will generate up to simulated time `t + L` [Fuj90]. This ability to predict future events and event transfer messages makes lookahead useful in determining how long simulator LP's can run between synchronizations. A large amount of lookahead enables a simulator to avoid synchronizing during long periods of simulated time during which the simulated entities are not communicating. Therefore, a simulation with a large amount of lookahead is more likely to achieve good performance than one with poor lookahead.

In Parallel Proteus, the lookahead `L` is equal to `Q`. I have developed two synchronization techniques which improve the lookahead in Parallel Proteus and, consequently, reduce the number of unnecessary synchronization operations. These techniques are barrier collapsing and predictive barrier scheduling. They attempt to improve performance by reducing the frequency of synchronization operations rather than by making each synchronization operation faster.

4.2.1 Barrier Collapsing

Barrier collapsing schedules global barrier synchronizations at runtime rather than statically at compile-time as in periodic global barriers. After each barrier, the host processors decide when to execute the next barrier. This scheduling decision is made based on information about the timestamps of the events in the queue. If, during a barrier synchronization at time `t`, none of the host processors has any event in its queue with timestamp less than `t + x`, then no event transfer messages will be generated before time `t + x`, and the timestamps of these remote events will be at least $\texttt{t + x}$. Therefore, no synchronization is needed until time MAX($\texttt{t + Q}$, $\texttt{t + x}$). In order to determine the value of $\texttt{t + x}$, the minimum timestamp of all events in the simulator at time $\texttt{t}$, each host processor first determines the minimum timestamp of all the events in its queue locally. Then, a global minimum reduction is performed to find the global minimum timestamp, $\texttt{t + x}$. The next global synchronization is scheduled for MAX($\texttt{t + Q}$, $\texttt{t + x}$). If $\texttt{x} > \texttt{Q}$, this technique eliminates $\frac{\texttt{x}-\texttt{Q}}{\texttt{Q}}$ synchronizations.

4.2.2 Predictive Barrier Scheduling

Like barrier collapsing, predictive barrier scheduling schedules barriers at runtime based on information about the timestamps of events in the queue. However, predictive barrier scheduling also takes into account the communication behavior of each event in the queue – i.e., how long until each event generates a remote event, one that needs to be processed on another host processor. For example, say that one host processor has two thread execution events in its queue: thread A, at time $\texttt{t}$, and thread B, at time $\texttt{t + 20}$. Thread A will execute only local operations until time $\texttt{t + 100}$, at which point it will send out a message to a processor simulated on another host processor, and B will execute local instructions until time $\texttt{t + 50}$, at which time it also sends a message. In this situation, barrier collapsing determines that the earliest timestamp in the queue is $\texttt{t}$, and keeps that as an estimate of the minimum time until an event on a particular host processor generates a non-local event. Predictive Barrier Scheduling, on the other hand, generates a better estimate of the earliest time an event on the queue will generate a remote event: no non-local operations will be executed before $\texttt{t + 50}$. As in barrier collapsing, each processor computes this estimate locally, and then enters a global minimum operation with the other host processors to compute the global estimate of the minimum time until next non-local event, and the next barrier is scheduled for this time. Since each individual processor is able to make a better estimate, the global estimate is likely to be higher, and more barriers will be eliminated.

Predictive barrier scheduling seems especially promising for applications that have long phases of local computation, alternating with communication phases. The synchronization during the communication phases would remain frequent, while the synchronization during computation phases would be eliminated.

Chapter 5

Implementation of synchronization techniques

This chapter describes how the synchronization techniques presented in chapter 4 were implemented and incorporated into Parallel Proteus. The fine-grained local barrier approach has not yet been implemented.

5.1 Local Barrier

In the local barrier approach, each host processor synchronizes only with its nearest neighbors. In Parallel Proteus, the nearest neighbors of each CM5 host processor are determined at the beginning of each simulation. Each host processor, using information about the simulated network, figures out which target processors are adjacent (one hop away on the simulated network) to the target processors it is responsible for simulating. The host processors that are responsible for simulating these adjacent target processors are the nearest neighbors.

As in the periodic global barrier technique, each host processor schedules a synchronization every `Q` cycles. Between synchronizations, each host processor keeps track of how many event transfer messages it has sent to (and how many it has received from) each of its nearest neighbors during the current quantum. Upon reaching a barrier, a host processor X sends to each of its nearest neighbors Y_i a count of the event messages that were sent from X to Y_i in the previous quantum. Processor X then waits until it receives similar information from each of its nearest neighbors. It polls the network until it receives and enqueues all of the incoming events it was told to expect, then goes on to simulate the next quantum.

In this approach, the host processors are only loosely synchronized (i.e., some processors may finish synchronizing and move on to the next quantum while others are still waiting). Consequently, a processor may receive event transfer messages generated in the next quantum while still waiting on synchronization ending the last quantum. In order to distinguish event messages from different quanta, successive quanta are labeled RED and BLACK. Each event transfer message is sent with a quantum identifier (RED or BLACK). In this way, when a processor is still waiting to receive some event messages from the last quantum (RED), it will not be confused by event messages one of its neighbors has sent from the next quantum (BLACK). Only two colors are needed because nearest neighbors will always be within one quantum of each other (and a processor only receives event messages from its nearest neighbors).

The local barrier technique requires each host processor to do more work during each synchronization than is needed to do hardware-supported global synchronization. Instead of issuing just one synchronization operation as in a global barrier, each processor sends, waits for, and receives a number of messages. Therefore, the benefits of this scheme may be outweighed by the overhead of sending and receiving messages and waiting for the messages to travel through the host network.

5.2 Barrier Collapsing

In the barrier collapsing approach, each global barrier involves three steps: synchronizing all of the processors, determining the timestamp of the earliest event in each host processor's queue, and then finding the global minimum timestamp using the local values. The next barrier is scheduled for this time.

The first step, synchronization, is performed exactly as in periodic global barriers, with a Strata CM5 control network reduction function [BB94]. This step includes synchronizing all host processors as well as waiting for the host network to drain.

The second step involves searching the event queue. In Parallel Proteus, as in sequential Proteus, the event queue is implemented as an array. Each array entry i is a "bucket" which contains events whose timestamps have least significant bits equal to i. When the events in each bucket are kept unsorted, insertions into the queue are very fast.[1] An event is inserted into the

[1] In Parallel Proteus, the events in each bucket can be maintained either in sorted order or unsorted order. (In sequential Proteus, they are always unsorted). Keeping them unsorted produces a correct, but not necessarily repeatable, simulation run. In order to provide repeatability of simulations, the events in each bucket must be

queue by first determining j, the value of the least significant bits of its timestamp, and then placing the event into the jth bucket. To dequeue an event, the simulator first examines bucket k, where k equals the value of the least significant bits of the current simulated time. If an event whose timestamp is equal to the current simulated time is found in bucket k, then it is removed from the bucket and then processed. If no such event is found, the simulator increments the current simulated time and examines bucket $k + 1$. Successive buckets are examined until the next event is found. When the events in the queue are close together in time, this procedure is fast because it involves examining only very few buckets.

The event queue can be searched for its earliest timestamp in two ways. First, a host processor can find the timestamp of its earliest event in the same way that it does when it actually wishes to dequeue an event – by starting at the bucket corresponding to the current simulated time and searching sequentially through the array until the next event is found. The advantage of this technique is that not all of the events in the queue have to be examined in order to find the minimum. However, if there are not very many events in the queue and they are far apart in the array, this search takes a long time because it involves examining many empty buckets. An alternative is to maintain a doubly linked, unsorted list of all the events in the queue (in addition to the array) and simply examine all of them to determine the minimum timestamp. This technique is very slow when there are many events in the queue.

Parallel Proteus combines the advantages of both techniques and avoids the disadvantages by using an adaptive technique. A doubly linked, unsorted list of pending events is maintained as events enter and leave the queue. When there are many events in the queue (above a certain threshold), the sequential search through the array is used to find the minimum timestamp. When there are few events in the queue (below the threshold), the linked list is traversed.

After all of the local minima are computed using the adaptive method, the global minimum timestamp is computed (the third step) with a Strata global minimum reduction operation.

Barrier collapsing takes advantage of lookahead already exposed by Parallel Proteus. The simulator allows local computations to run ahead of simulated time for up to 1000 cycles at a time without interruption. This exposes the short-term future communication behavior of the

sorted according to timestamp and unique identification number. This ensures that events on a particular host node with identical timestamps are processed in the same order in every run of the simulation, even though the host network may deliver them in different orders in different runs. Sorting the events in each bucket in this way makes insertion into the queue slower, but dequeuing faster (because the earliest events in each bucket are easily accessed).

application. If the application is in a local computation phase, no communication events will be generated and thread execution events will be scheduled at 1000-cycle intervals with no other events in between. In the periodic global barriers technique, such a situation results in the execution of many useless barriers. In situations where $\mathtt{Q} = 2$, five hundred consecutive barriers are performed in the 1000-cycle interval during which the simulator catches up with the thread. All of these barriers are useless because no work is done between them. Barrier collapsing collapses all such useless consecutive barriers into one, thereby decreasing the total number of barriers executed. This technique will clearly be most useful for simulating applications that have local computation phases of considerable length.

5.3 Predictive Barrier Scheduling

Predictive barrier scheduling is similar to barrier collapsing in that it also involves a global barrier followed by a computation of a local minimum on each processor followed by scheduling the next barrier at the global minimum of these local values. The difference is that the values being minimized in predictive barrier scheduling are not the timestamps of the events in the queue, but rather lower bound estimates on the time until these events generate remote events (those that need to be sent over to other host processors).

Two extensions to the barrier collapsing implementation are required in order to implement predictive barrier scheduling. The first one is a method of determining and associating with each event a lower bound estimate on the time until it generates a remote event. The second is a way of examining several of the earliest events in the event queue (as opposed to just the one earliest event) in order to compute a lower bound estimate of the earliest time *any* event in the queue will generate a remote event. (Note: it is not necessarily the earliest event in the queue which will generate the earliest remote event). This section describes how Parallel Proteus accomplishes these two tasks.

5.3.1 Computing Estimates

In order to understand how Parallel Proteus determines when events will generate remote events, let us examine the different types of simulator events that can be in a host processor's event queue during a synchronization operation: thread execution events, network events, and completion checking events. Completion checking events never generate any remote events, so they can be assigned arbitrarily large estimates of minimum time until remote event.

Network events involve either sending, receiving, or routing packets in the simulated network. Send events launch packets into the simulated network. A receive event occurs at each node in a packet's path through the network. A receive event at an intermediate node generates a route event that will send the packet through its next hop. A receive event at the packet's final destination node spawns a message handling routine. Using the architectural parameters of the simulated network specified at compile-time, it is straightforward to figure out the earliest time each network event may generate a remote event. Route packet events with timestamp `t` generate receive events to be processed on remote processors at `t + Q`; send events and receive events with timestamp `t` generate route packet events with timestamp `t` which then generate receive events to be processed on remote processors at `t + Q`. A receive event with timestamp `t` can also generate an interrupt which causes a handler to be run at simulated time `t`. If the first thing the handler does is to send a message, a send event at time `t` results, and this leads to another receive event to be processed on a remote processor at `t + Q`. Therefore, the lower bound estimate on the time until a network event generates a remote event is `Q` simulated cycles.[2] When a network event with timestamp `t` is enqueued, the simulator knows that it will not generate a remote event until at least time `t + Q`, and associates this value with the event.

Thread events are more complicated. They involve resuming an application thread at the location in the application code where the thread was last interrupted. It is not straightforward to determine how long a thread will run before generating a remote event.

Parallel Proteus computes lower bound estimates of the time until threads generate remote events in the following way. First, dataflow analysis of the application code is done at compile-time to determine the minimum distance (in cycles) to a non-local operation from each point in the application code (remote events result only from the execution of non-local operations). Then, the code is instrumented with instructions which make these minimum distances accessible to the simulator at runtime. When the application thread is executing, the code added in the instrumentation phase updates a variable which holds the value of the current minimum time (in simulated cycles) until the thread executes a non-local instruction. Every time control passes from the thread back to the simulator, the simulator examines this variable and

[2]Actually, if a message passes through several hops of the simulated network which are simulated on the same host node before traveling to those simulated on other host nodes, an estimate larger than `Q` can be computed. This technique has not been implemented in Parallel Proteus, however.

associates its value with the thread execution event that gets enqueued in order to continue execution of the thread at a later time.

Compile-time Analysis

At compile-time, dataflow analysis is performed on the basic block graph of the application code that is constructed by the Proteus utility `augment`. Each basic block consists of sequence of local, non-branching instructions followed by a branch instruction or a procedure call. Each basic block is labeled with its execution length in cycles, and has one or two pointers leaving it which indicate where control flows after leaving the block. This graph is used to calculate a conservative estimate of the minimum distance (in cycles) to a non-local operation from the beginning of each basic block. Because of the way `augment` instruments the code, a thread never gets interrupted in the middle of a basic block. Therefore, estimates are only needed for the beginning of basic blocks. I chose not to implement inter-procedural analysis; instead, every procedure call is assumed to lead to a non-local operation.

Because no non-local operations occur in the beginning or middle of a basic block, a first estimate of the minimum distance from the beginning of a basic block to a non-local operation is the size of the basic block. If the block ends with a procedure call (which is either a non-local operation or assumed to immediately lead to one), then the estimate is done. However, if the block ends with a branch statement (a *local* operation), the following is added to its current estimate: the estimate for the basic block following it or, if it ends in a conditional branch, the minimum of the two estimates for the two blocks following the first block. In order to compute the estimates for the blocks following the first one, the same operations have to be performed on them. In order to compute all the minimum distances to non-local operations from all the basic blocks in the graph, the following iterative dataflow algorithm (based on algorithm 10.2 in [ASU86]) is used:

```
FOREACH basic block B in graph
  estimate(B) = sizeof(B)
ENDFOR
DO
    change = false
    FOREACH basic block B in graph
```

```
        old_estimate = estimate(B)
        if (B ends in procedure call) then estimate(B) = sizeof(B)
        else if (B has 1 successor) then estimate(B) = sizeof(B) + estimate(successor(B))
        else estimate(B) = sizeof(B) + MIN ( estimate (left_succ(B)), estimate(right_succ(B)))
        if (estimate(B) != old_estimate) then change = TRUE
    ENDFOR
UNTIL change == false
```

Code Instrumentation

After the minimum distance to remote operation has been determined for each block, each block is instrumented not only with code that increments a counter with the number of cycles spent in that block every time the block is executed (which was the original purpose of augment), but also with code that records away the value of the minimum distance to remote operation for the beginning and end of the block. In this way, when the thread gets interrupted or makes a procedure call which causes control to go back to the simulator and the thread to be enqueued as a thread event to be continued later, the thread's minimum time until remote operation can be accessed in the variable where it was stored immediately before the thread was suspended. This value is then associated with the new thread event in the queue.

5.3.2 Searching the Event Queue

This section describes how Parallel Proteus calculates the minimum time until remote event of all the events in the queue, given the minimum times until remote event for each event in its queue. As in barrier collapsing, there are two ways in which this minimum can be computed. First, the event queue can be searched sequentially, in increasing timestamp order, starting with the earliest event and continuing until the timestamps exceed the minimum value already computed (the minimum value until remote event for each event is greater than or equal to its timestamp). As in barrier collapsing, this sequential search through the queue may take a long time if the events are few and far between. Its performance is likely to be better when there are many events in the queue, when it is more likely that the events are close together, because fewer events will be examined before the minimum is found. Fortunately, in the case where there are few events in the queue, a linked list containing all of the events in the queue can be searched quickly. So, as in barrier collapsing, an adaptive scheme is used. When there are many events in the queue, the first, sequential search algorithm is used; if there are few events, a search through the linked list of all the events is used.

5.3.3 Summary

Predictive barrier scheduling consists of three parts. First, compile-time analysis is used to instrument code with instructions which store information about each thread's future communication behavior. Second, when being enqueued, each event has associated to it a minimum time until it generates a remote event (for which synchronization would be needed). For network events, this value is determined by the network parameters of the simulated network. For thread events, this value is determined by the place in the application code where the thread left off, and the instrumentation code makes that value available at the time the thread event is enqueued. Third, during each barrier synchronization, each host processor inspects its own event queue to figure out the earliest time any one of its events will generate a remote event. It does this by calculating the minimum time until remote event of all the events in the queue. Each processor contributes its local minimum to a global minimum operation to determine the global minimum of the time of the next remote event, and the next barrier synchronization is scheduled for this time.

This technique adds quite a bit of overhead to each synchronization operation. The hope is that it will eliminate so many of the synchronization operations that it will improve performance over periodic global barriers even with a higher overhead cost per synchronization.

Chapter 6

Experiments and evaluation of synchronization techniques

This chapter presents and discusses performance results for the local barrier and barrier scheduling synchronization techniques. As in chapter 3, the data presented here are averages of 3 runs, with $\sigma/\overline{X} \leq 4\%$.

6.1 Barrier Scheduling

Barrier collapsing and predictive barrier scheduling are two barrier scheduling techniques whose purpose is to improve the performance of Parallel Proteus by reducing the number of synchronization operations. Barrier collapsing is a runtime technique which allows application threads to run ahead of simulated time. An improved approach, predictive barrier scheduling couples barrier collapsing with compile-time analysis. These techniques will be evaluated against the baseline periodic global barrier approach by examining three criteria: overhead, percentage of barriers eliminated, and overall performance improvement.

As expected, experiments show that predictive barrier scheduling always performs better than barrier collapsing alone since the former incorporates the latter. Although predictive barrier scheduling eliminates more barriers, barrier collapsing is in fact responsible for the overwhelming majority of that amount. Predictive barrier scheduling improves upon barrier collapsing alone by eliminating an additional 4% to 26% of the barriers executed under the baseline periodic global barrier approach. I will focus on the results for predictive barrier scheduling because of its superior performance.

6.1.1 Overhead

Predictive barrier scheduling adds overhead in four areas:

- compilation,
- application code execution,
- barrier scheduling, and
- barrier waiting time.

At compile time, application code is analyzed to determine its communication behavior. This analysis adds little noticeable overhead to compilation because it operates on data structures previously constructed by `augment` (the program that adds cycle-counting instructions to the application code). However, more overhead would be incurred by simulators that are not direct-execution-based, because more work would have to be done to generate the basic block graph necessary for this analysis.

During code instrumentation, predictive barrier scheduling adds 6 instructions to each basic block of the application. These instructions store the estimates of the time until a thread executes a non-local operation that were generated by the compile-time analysis. These are in addition to the 17 instructions already added in order to implement direct execution (for cycle counting and stack overflow checking). These 6 instructions do not significantly increase simulation time because only a small percent of total runtime is spent executing them. The overhead due to these added instructions is clearly negligible compared to synchronization overhead, which accounts for a full 80% of simulation time (see chapter 3).

During each barrier synchronization, host processors schedule the next barrier. This entails computation of local estimates of the time until a host processor generates an event transfer message, and an additional global reduction operation in order to obtain the global minimum of these values. The extra global reduction takes 150 cycles (4.5 microseconds), and the extra computation takes anywhere from 5 to 50 cycles, depending on the number of events in the queue. The more target processors, the more events in the queue, the longer the computation.

Predictive barrier scheduling reduces the frequency of synchronizations and, therefore, increases the amount of simulation work between synchronizations. This, in turn, increases the waiting time at each barrier, because there is a larger difference in the amounts of simulation work done by each processor in each quantum. Thus, each barrier time increases. Experiments

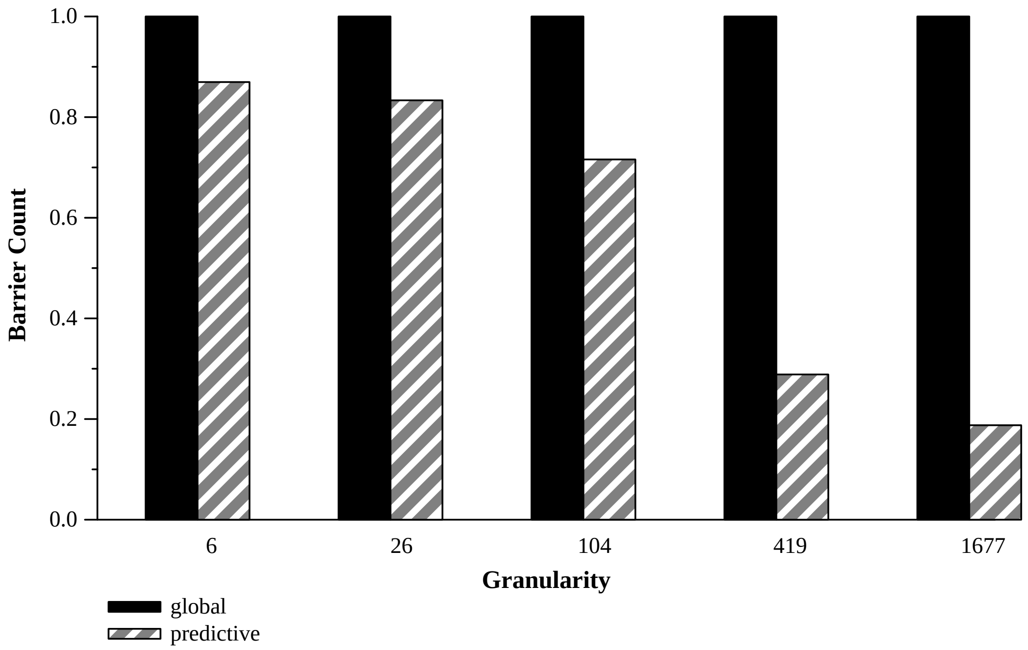

Figure 6-1: **Predictive Barrier Scheduling vs. Periodic Global Barriers.** Normalized barrier count vs. granularity for *SOR*, 625 target processors, 32 host nodes, varying grid size.

---

show that in my applications, each barrier (which is typically 20 to 600 microseconds long) becomes 20% to 1400% longer. The less frequent the synchronizations, the longer they are.

6.1.2 Barriers eliminated

The purpose of predictive barrier scheduling is to eliminate unnecessary barriers which occur during long computation phases. The data presented in figures 6-1, 6-2, and 6-3 show that this method does, in fact, reduce the number of barriers executed by 12% to 95%. As expected, the number of barriers eliminated is greater in the simulation of applications with long periods of local computation between communication phases. To illustrate this feature, I graph the number of barriers executed versus *granularity*. Granularity is a measure of the length of local phases of computation relative to the length of communication phases. Applications with large granularity have long local computation phases. Those with small granularity are dominated by communication and short computation phases.

Granularity is measured differently in each application because the length of local computation phases depends on application-specific parameters. In *SOR*, for example, local computation phases lengthen when more data are assigned to each target processor. Therefore, I define gran-

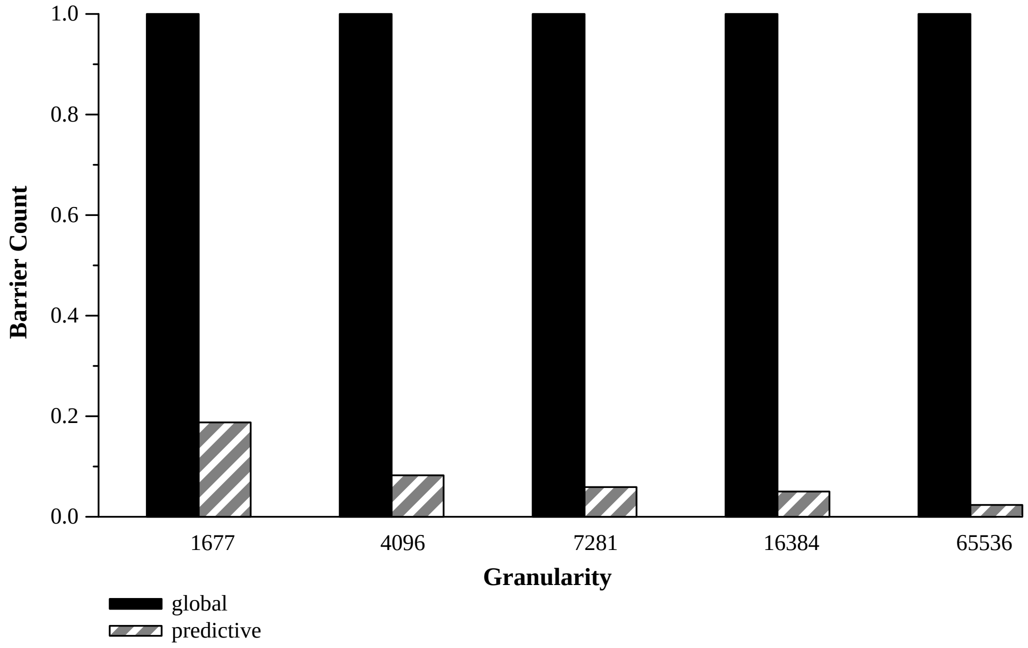

Figure 6-2: **Predictive Barrier Scheduling vs. Periodic Global Barriers.** Normalized barrier count vs. granularity for *SOR*, 1024 x 1024 grid size, 32 host nodes, varying number of target processors.

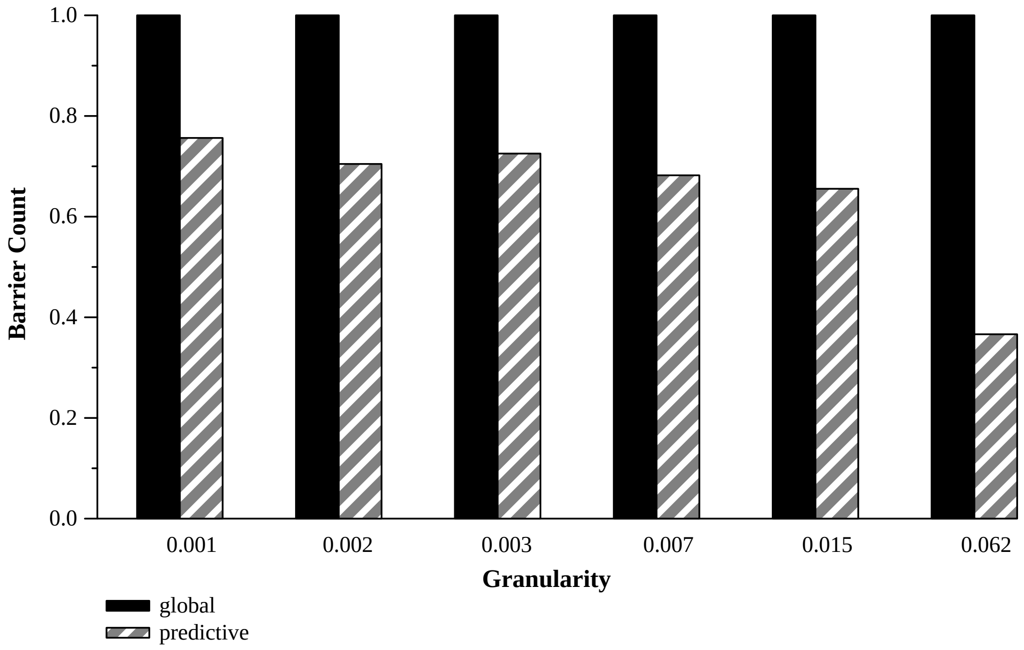

Figure 6-3: **Predictive Barrier Scheduling vs. Periodic Global Barriers.** Normalized barrier count vs. granularity for *radix*, 8192 numbers per target processor, 32 host nodes, varying number of target processors.

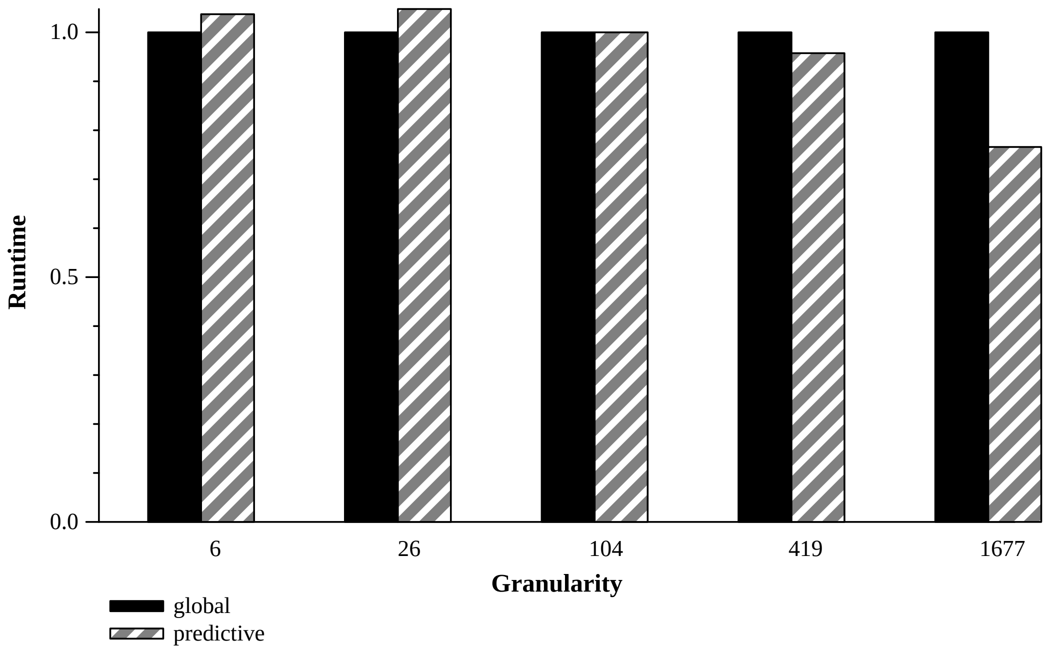

Figure 6-4: **Predictive Barrier Scheduling vs. Periodic Global Barriers.** Normalized simulator runtime vs. granularity for *SOR*, 625 target processors, 32 host nodes, varying grid size.

ularity for *SOR* to be equal to the number of data points assigned to each target processor. In *radix*, on the other hand, the length of communication phases relative to the length of computation phases increases with the number of target processors, even if the target processors are assigned the same amount of data in all cases. Therefore, I define granularity for *radix* to be equal to the inverse of the number of target processors, making granularity decrease when the number of target processors increases. (In *radix*, the number of data points on each target processor is constant in all experiments). Figures 6-1, 6-2, and 6-3 show how the number of barriers eliminated increases with granularity in *SOR* and *radix*.

6.1.3 Performance Improvement

This section describes the overall performance improvement achieved by predictive barrier scheduling. Experiments indicate that performance improvement over the periodic global barrier approach ranges from -5% to +65%. Because predictive barrier scheduling lengthens each barrier, no performance improvement is achieved when few barriers are eliminated. Results show that, in general, about 30-40% of the barriers (otherwise executed under the periodic global barrier approach) have to be eliminated in order to achieve any performance improve-

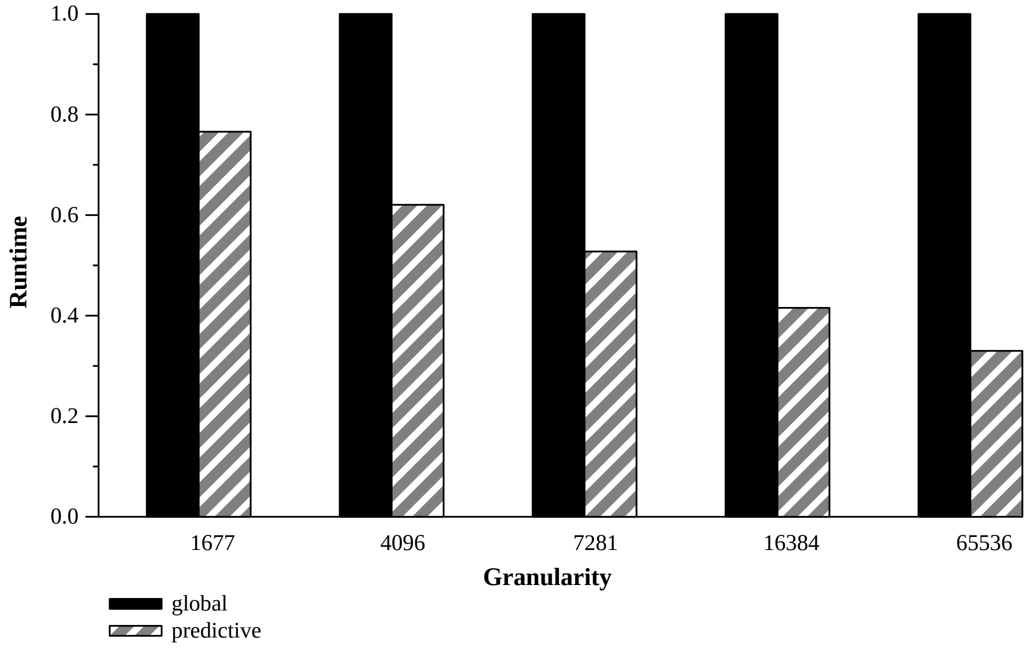

Figure 6-5: **Predictive Barrier Scheduling vs. Periodic Global Barriers.** Normalized simulator runtime vs. granularity for *SOR*, 1024 x 1024 grid size, 32 host nodes, varying number of target processors.

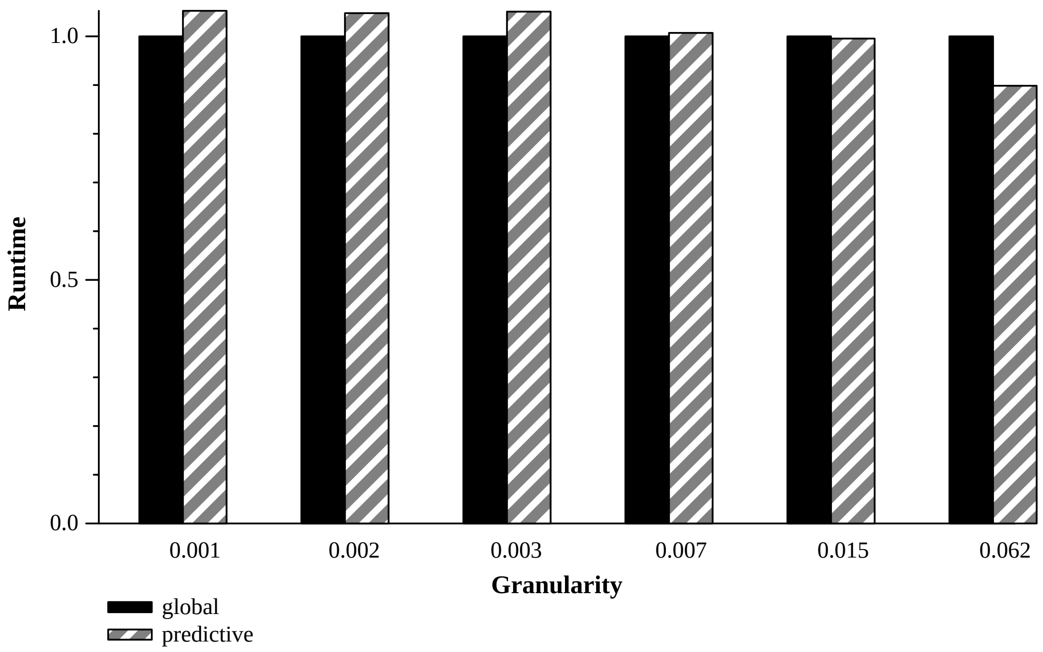

Figure 6-6: **Predictive Barrier Scheduling vs. Periodic Global Barriers.** Normalized simulator runtime vs. granularity for *radix*, 8192 numbers per target processors, 32 host nodes, varying number of target processors.

---

ment. As expected, performance improvement increases with granularity, just as demonstrated in the previous section. This is evident in the graphs of normalized simulator runtime for *SOR* and *radix*, figures 6-4, 6-5, and 6-6. In general, the computation phases in *radix* are much shorter than those in *SOR*. Therefore, little improvement (only up to 10%) is seen in *radix*. *SOR* with large data sizes has large granularity and, therefore, achieves large performance improvements (up to 65%). Clearly, the predictive barrier scheduling method is only useful for applications with reasonably long local computation phases.

6.2 Local Barriers

The purpose of the local barrier approach is to improve the performance of simulators like Parallel Proteus by reducing the waiting time at synchronization points. Results show that this approach is effective in many cases. However, local barriers cannot always improve upon the global barrier approach. When waiting time at periodic global barriers is short, local barriers do not perform well because they cannot improve waiting time. In fact, they add time to the non-waiting period at the synchronization point.

6.2.1 Overhead

The local barrier approach adds overhead to each synchronization because it requires each host processor to send, receive, and await the individual messages of each of its nearest neighbors. The greater the number of nearest neighbors, the greater the overhead. In contrast to the single hardware-supported synchronization operation in the periodic global barrier approach, there is no hardware support on the CM5 for the pairwise synchronizations required for local barriers. In the best case, when all processors are already synchronized, a global barrier (on the CM5) takes 150 cycles (4.5 microseconds). Under the same conditions, a local barrier takes anywhere from 800 to 1300 cycles (24 to 39 microseconds), depending on the number of nearest neighbors each host processor has. Consequently, the local barrier method is expected to do better only when the periodic global barrier approach has long waiting times at each synchronization. In that case, local barriers can improve performance by decreasing the waiting times, even though they add some overhead to the execution of each synchronization. In the case where the periodic global barrier approach has very short waiting times, the local barrier approach will be unable to improve performance.

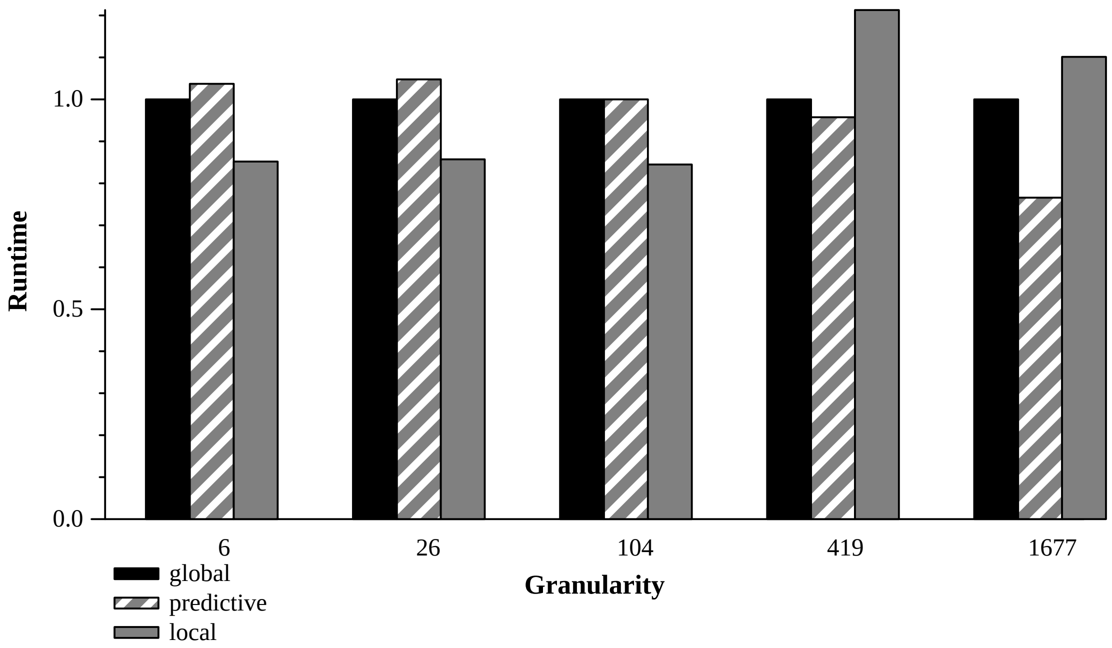

Figure 6-7: **Local Barriers.** Normalized simulator runtime vs. granularity for *SOR*, 625 target processors, 32 host nodes, varying grid size.

6.2.2 Performance Improvement

As expected, experiments show that the local barrier approach improves performance by decreasing barrier waiting time in situations where the periodic global barrier approach suffers from long waiting times. This occurs in applications of small granularity, where communication is frequent and computation phases are short. In large granularity applications, on the other hand, Parallel Proteus (using periodic global barriers) executes the long computation phases 1000 cycles at a time (see chapter 3 and appendix A). This leads to the execution of many (unnecessary) barriers in a row, with no simulation work in between. The waiting time at most of the barriers, then, is very short.

Local barriers shorten synchronizations in small granularity applications by up to 25%. In large granularity applications where barriers are already short, the local barrier approach lengthens barrier time by up to 192%.

Graphs of normalized simulator runtime (figures 6-8, 6-7, and 6-9) show that the behavior of the local barrier approach complements that of predictive barrier scheduling. Predictive barrier scheduling improves performance for applications with large granularity, while local barriers improve performance for applications with small granularity. Local barriers improve

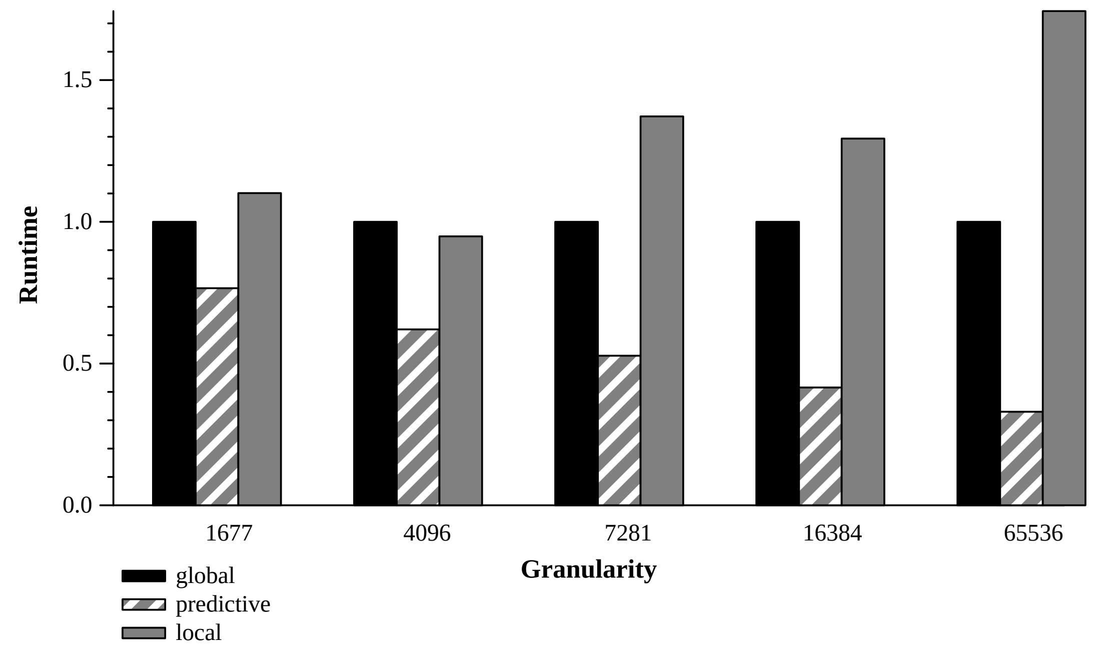

Figure 6-8: **Local Barriers.** Normalized simulator runtime vs. granularity for *SOR*, 1024 x 1024 grid size, 32 host nodes, varying number of target processors.

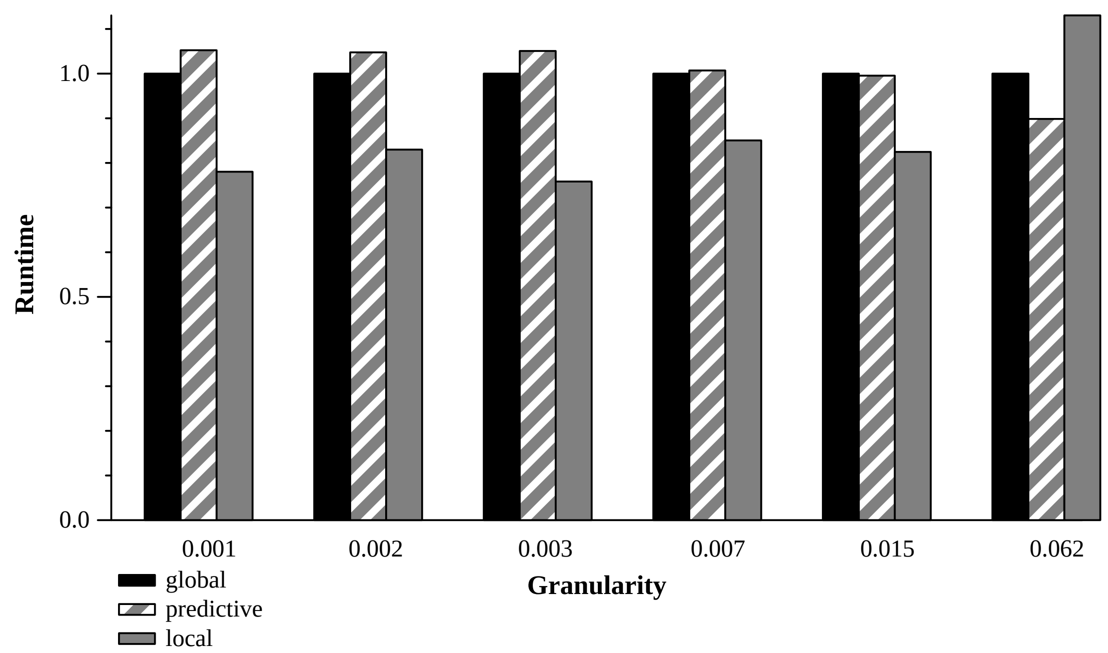

Figure 6-9: **Local Barriers.** Normalized simulator runtime vs. granularity for *radix*, 8192 numbers per target processor, 32 host nodes, varying number of target processors.

performance (reduce runtime) most in *radix* (up to 24%) and in small granularity *SOR*.

6.3 Discussion

The previous sections showed that both nearest-neighbor synchronization and lookahead-increasing techniques are effective ways of reducing synchronization overhead in the parallel simulation of message-passing multiprocessors. While neither technique is successful all of the time, each complements the other by improving performance in situations where the other fails to do so.

This complementary relationship between predictive barrier scheduling and the local barrier approach suggests that one might combine the two techniques in the hope of achieving more consistent performance improvement. One way to do this is to use each technique in the situation where it does better: the local barrier approach during communication phases and predictive barrier scheduling during computation phases. In this way, unnecessary barriers in the computation phases will be eliminated by predictive barrier scheduling. However, the extra overhead required will not be incurred in the communication phases. For its part, the local barrier approach will shorten the waiting times in communication phases.

The difficulty lies in determining when to switch from one technique to the other. This could be determined at compile time by examining the program to see when it calls communication operations. However, a better approach may be to switch adaptively at runtime. Under predictive barrier scheduling, the simulator can monitor application communication by counting event transfer messages at runtime. If the simulator detects frequent communication, it will switch to the local barrier approach. However, switching from local barriers to predictive barrier scheduling is more complicated, because a global decision must be made to switch, but processors are doing *local* synchronization. To remedy this, each processor can continue to monitor event transfer traffic while synchronizing locally every `Q` cycles, but global synchronizations will be performed every 100*`Q` cycles to facilitate a global decision. If there has been little communication traffic, signifying a computation phase, the simulator will switch back to predictive barrier scheduling. I have not experimented yet with this combined technique, but it is clearly worth doing so.

While the local barrier approach and predictive barrier scheduling have been evaluated in the context of a direct-execution simulator, they are certainly also applicable to simulators which simulate application code execution in a different way (e.g., Bedichek's threaded code simulator

[Bed95]). First, local barriers do not depend on direct execution in any way. Second, while my implementation of predictive barrier scheduling does rely on Parallel Proteus's support for direct execution, this is not essential to the technique. All that is needed to implement predictive barrier scheduling is a way of associating with each value of an application's program counter, an estimate of the time until the application will next communicate. The necessary analysis can be done at compile-time as in Parallel Proteus, but instead of instrumenting code, the information generated at compile-time can be stored in a table. As each instruction is simulated, information about future communication behavior can be looked up in the table. Therefore, the applicability of local barriers and predictive barrier scheduling extends to more simulators than just the one described here.

Chapter 7

Related Work

This chapter discusses the evolution of parallel simulation techniques, the use of nearest neighbor synchronization and lookahead in other parallel simulators, and the techniques used in other parallel simulators of parallel computers.

7.1 Parallel Simulation Techniques

Bryant, Chandy, and Misra developed the first conservative PDES algorithms [Bry77] [CM79] [CM81] . In both the Bryant and Chandy-Misra approaches to conservative PDES, LP's do not synchronize regularly, but only during the transmission of event transfer messages called for by communication in the simulated system. LP's are required to wait on their neighbors, not knowing if or when a message will be sent. This can lead to deadlock. The Chandy-Misra approach involves detecting and recovering from deadlock. Bryant's method involves sending additional timing messages in order to avoid deadlock. It is similar to the idea of local barriers, but in some cases it involves the transmission of many more messages than a local barrier requires. Chandy and Misra consider a similar deadlock avoidance technique (NULL messages), but dismiss it as being too expensive in the number of messages produced.

Synchronous simulation algorithms, such as the one used in Parallel Proteus, evolved out of the Bryant and Chandy-Misra (BCM) approaches. These involve iteratively determining which events are safe to process before executing them. Barrier synchronizations are used to separate iterations, while operation is asynchronous between barriers. These algorithms avoid deadlock and guarantee the same semantics and correctness as the BCM approaches. Examples of synchronous algorithms can be found in [Lub88] [Aya89] [AR92] [CS89] [SBW88] [Ste91] (plus one more reference: yawns).

7.2 Nearest Neighbor Synchronization

Synchronous techniques often suffer in performance because of long waiting times at global synchronizations. As described previously, nearest neighbor synchronization can be useful in alleviating this problem. This section describes the use of nearest neighbor synchronization (in effect, a move back towards the asynchronous nature of the original BCM algorithms) in two simulators and one other parallel application.

Cheriton et al. have looked into the problem of long barrier waiting times in their particle-based wind tunnel simulation which runs on a shared memory multiprocessor [CGHM93]. They observed better performance with nearest-neighbor (local barrier) synchronization for this application.

Lubachevsky improved the performance of an Ising spin model simulation with a technique similar to local barriers [Lub89a]. His algorithm involves several steps. First, the minimum simulation time over all the nodes is computed and broadcast. Second, each node examines the simulated time of a group of neighboring nodes in order to compute an estimate of the minimum time at which it could be affected by other nodes. Events with timestamps less than this minimum time are then processed. Simulated time on all nodes is kept within a certain number `B` units of each other. The value of `B` determines how many neighboring nodes have to be examined in the checking phase described above. A small value for `B` can greatly reduce the number of nodes that have to be checked, which improves the simulation performance. However, there are explicit global synchronizations within the algorithm which seem to counteract any gains made. The algorithm runs well on the SIMD CM-2, where global synchronization is almost free because the nodes run in lock-step. However, on most other computers, where nodes run asynchronously, this synchronization will be very expensive. A local barrier is very similar to this algorithm (when `B` is chosen to equal time quantum `Q`), except that it involves no global synchronization.

Nearest neighbor synchronization can be viewed as a form of fine grain synchronization. Yeung and Agarwal report a significant performance improvement for preconditioned conjugate gradient when using fine grain synchronization instead of global synchronization. They attribute this improvement in part to the elimination of unnecessary waiting [YA93].

7.3 Improving Lookahead

Lookahead, the ability to predict the future communication behavior of a simulated system, can be quite useful in reducing synchronization overhead in conservative parallel simulators. This has been documented by [Nic91] and [LL89]. Fujimoto also demonstrated the importance of lookahead in achieving good performance with experiments on synthetic workloads using the Chandy-Misra deadlock avoidance algorithm [Fuj88].

The following are two examples of conservative parallel simulators whose good performance is due to improved lookahead.

Lubachevsky's *opaque periods* improve the performance of his Ising spin model simulator by increasing the number of safe events found at each iteration of his algorithm [Lub89a]. Because all events are non-local in this simulation, each host processor determines the length of each opaque period by calculating the minimum of the times of the next events on each of its neighbors.

LAPSE is a conservative parallel simulator of the message-passing Intel Paragon [DHN94]. It achieves good performance by exploiting two sources of lookahead. First, like barrier collapsing, LAPSE lets some application code execute in advance of the simulation of its timing. LAPSE also exploits the lookahead provided by simulating the large software overhead required to send a message on the Paragon. (If the software overhead to send a message on the simulated architecture is always $\geq x$ cycles, then there will always be at least an x-cycle gap between communication operations on each simulated node). LAPSE is different from Parallel Proteus in that it does not model network contention while Parallel Proteus does. It is not clear whether the techniques used in LAPSE will work as well when the network is simulated more accurately, since synchronization will be needed much more frequently.

7.4 Other Parallel Simulators of Parallel Computers

This section describes several other parallel simulators of multicomputers (in addition to LAPSE).

7.4.1 Wisconsin Wind Tunnel

The Wisconsin Wind Tunnel is a parallel simulator of shared memory architectures which runs on the CM5 [RHL+93]. It uses a synchronous conservative PDES algorithm with periodic global barriers and a synchronization time quantum of 100 cycles. WWT achieves good performance in part because of this large time quantum (a consequence of a very simple network simulation model).

In more recent work on the Wisconsin Wind Tunnel, several more accurate network simulation modules have been added [BW95]. The most accurate ones (*Approximate* and *Baseline*) exploit lookahead proportional to the minimum message size in the simulated system. The first scheme (Approximate) distributes network simulation among the processors of the host machine (as does the hop-by-hop model in Parallel Proteus), and models contention and message delivery times approximately. The synchronization time quantum takes on a value of 15 cycles. In the second scheme (Baseline), network simulation is accurate but centralized; it takes place on only one node of the host machine. (This scheme is slightly more accurate than the distributed scheme used in Parallel Proteus because it models finite buffers at routers while Parallel Proteus assumes infinite buffering). They report on the increased slowdowns seen with the more accurate network simulation techniques, as well as on the magnitude of the inaccuracies observed with the less accurate techniques. They contend that fast approximate simulators are more valuable than slow accurate ones.

7.4.2 Parallel Tango Lite

Parallel Tango Lite is a parallel simulator of shared memory architectures which runs on the DASH machine [Gol93]. It uses the same simple network model and conservative PDES algorithm as the Wisconsin Wind Tunnel, also synchronizing every 100 timesteps. Experiments were done using small workloads (24 target processors) on small host machine configurations (up to 12 host processors). It achieved a maximum speedup of 4.4 on 8 host processors. The developers report that the performance of the simulator is limited by barrier synchronization time (because of their choice of host machine, they were not able to take advantage of the fast global synchronization operations available on the CM5, as were the designers of the WWT). Load imbalance contributed to the large cost of the barrier synchronizations. Another obstacle to speedup is the global data structure needed to simulate the single thread queue of the target machines. The performance of the simulator improved slightly when multiple target processors were simulated on each host processor.

7.4.3 PSIMUL

PSIMUL is a parallel simulator of shared memory architectures which runs on the 2-CPU IBM System 3081 [SDRG+86]. It is used to collect memory traces of parallel programs. It achieves good speedup because no synchronization is performed during the simulation process; instead, synchronization markers are inserted into the trace files and later used to correctly order the memory references listed in the traces.

7.4.4 NWO-P

NWO-P is the parallel version of NWO, a detailed sequential simulator of the MIT Alewife distributed-memory multiprocessor [JCM93]. NWO-P operates in lock-step, synchronizing every cycle of simulated time. Because NWO-P simulates each cycle of Alewife in a great amount of detail, the overhead for simulating each cycle is very high and far outweighs the synchronization overhead [Joh95].

7.5 Discussion

Of all the work described in this chapter, the Wisconsin Wind Tunnel (WWT), Parallel Tango Lite (PTL), and LAPSE are the most closely related to Parallel Proteus. In this section, I discuss how simulating shared memory (as WWT and PTL do) limits the lookahead that can be exploited. Then I compare network simulation in Parallel Proteus and WWT, and conclude with a discussion of the lookahead-improving techniques in Parallel Proteus and LAPSE.

WWT, PTL, and LAPSE are similar to Parallel Proteus in that they are also parallel, direct execution simulators of parallel architectures. The fact that direct execution is a common feature indicates that all four simulators incur similar overhead for simulating operations local to each target processor. However, they differ in the simulation of remote operations. WWT and PTL simulate shared memory architectures, while Parallel Proteus and LAPSE simulate message-passing architectures. Consequently, WWT and PTL cannot exploit lookahead in the same way as Parallel Proteus and LAPSE can. Communication can potentially happen on every memory reference in shared memory systems, so it is difficult to identify ahead of time periods of computation during which it is certain that no communication will take place. Also, all accesses to shared data (whether requiring communication or not) potentially affect other target processors, so it is not correct to let application threads run ahead until a communication operation is encountered. Instead, WWT and PTL achieve good performance by sacrificing some accuracy in the simulation of the network.

WWT currently supports a range of network simulation modules (from very accurate to not very accurate), all of which could be easily incorporated into Parallel Proteus and used along with the synchronization techniques described in this work. (Parallel Proteus now supports a fairly accurate hop-by-hop model which simulates contention, as well as a much less accurate 100-cycle delay model (exactly like the one in WWT)).

LAPSE and Parallel Proteus both improve lookahead by letting application threads run ahead of simulated time. However, Parallel Proteus improves on this technique by also doing compile-time analysis of the application code in order to increase lookahead (predictive barrier scheduling). Unlike LAPSE, Parallel Proteus does not rely on less accurate network simulation nor on any particular simulated system's large software overhead for sending messages in order to obtain lookahead.

Chapter 8

Conclusions

Parallelism is necessary for fast, detailed simulation of large multicomputers. Synchronization overhead, however, often severely limits the performance of conservative parallel simulators. This occurs because low-latency communication in simulated networks requires frequent synchronization of simulator processes. One way to reduce synchronization overhead is to sacrifice accuracy in the simulation of the network. Focusing on local barriers and predictive barrier scheduling, this thesis demonstrates that for simulations of message-passing parallel computers, nearest neighbor synchronization and application-specific optimization can improve performance by reducing synchronization overhead *without* sacrificing accuracy in network simulation.

Local barriers use nearest-neighbor synchronization to reduce waiting time at synchronization points. Under local barriers, the host processors are only loosely synchronized; each host processor only waits for several others at each synchronization point. While this technique does add some overhead to the execution of each synchronization, it often produces an overall performance improvement (compared to periodic global barriers) because it significantly shortens the time spent waiting at each synchronization point. The largest performance improvements (up to 24% in my experiments) are achieved in simulations of communication-bound applications, where waiting times under periodic global barriers are high. In simulations of computation-bound applications, local barriers slow down simulations by up to 75%.

Predictive barrier scheduling uses both compile-time and runtime analysis to reduce the frequency of synchronization operations. Applications to be run on the simulated machine are analyzed at compile-time in order to determine their future communication behavior. More information about communication behavior is produced at runtime, when application threads are allowed to run ahead of simulated time. Using the information about communication behavior generated at compile-time and runtime, the simulator tries to schedule synchronizations only when they are absolutely necessary – during communication phases. Experiments show that under predictive barrier scheduling, simulations perform 10% to 95% fewer synchronizations than under periodic global barriers, achieving overall performance improvements in the range of -5% to +65%. The largest performance improvements occur in computation-bound applications (complementing the behavior of the local barrier approach).

The local barrier approach is a general-purpose technique. It can be (and has been) used in many different types of parallel simulators because it does not depend on the specific characteristics of multicomputer simulation. Predictive barrier scheduling, on the other hand, is very specific to the simulation of message-passing parallel computers. Because it depends on the ability to easily identify communication operations, it cannot be easily adapted for simulators of shared-memory parallel computers. Predictive barrier scheduling does not, however, depend on any specific simulation technique (such as direct execution). It can be used in any type of simulator of message-passing multicomputers.

It may be possible to improve the performance of the two techniques. The following are several suggestions on how to accomplish this. First, a fine-grained implementation of local barriers (as described in 4.1.2) may increase the benefits of loose synchronization already observed in my experiments. Second, in predictive barrier scheduling, simple inter-procedural analysis (perhaps identifying procedures where no communication occurs at all) may improve estimates of computation phase lengths, and eliminate more synchronizations as a result. A third way of improving performance is to combine both techniques into an adaptive approach where each technique is used in the situation where it can improve performance the most. Synchronization during communication phases would be performed using local barriers, but during long computation phases, predictive barrier scheduling would be used. In this way, the benefits of both techniques can be observed in every simulation.

Appendix A

Accurate thread simulation in Parallel Proteus

Like Proteus, Parallel Proteus simulates an application thread executing on a target processor at time `t` by letting the thread execute for a maximum of 1000 simulated cycles without interruption. In effect, threads are allowed to run ahead of simulated time. While this technique allows the simulator to avoid numerous context switches and facilitates the barrier collapsing and predictive barrier scheduling techniques, there is a danger of executing local operations[1] too early and message handlers too late. This appendix explains how thread simulation remains accurate in the Parallel Proteus experiments described in this thesis, and describes a design for maintaining thread simulation accuracy in all situations.

Threads are allowed to execute up to 1000 local instructions without interruption. If a thread encounters (but does not yet execute) a non-local operation (such as a message send or synchronization variable access) after `x` cycles of local instructions, control returns to the simulator immediately. The simulator notes that the thread performed `x` cycles of work and then goes back to processing events from simulated time `t` forward. This includes handling all messages that, in the simulated system, arrive before the non-local operation occurs. After all of these have been handled, and this handling took `x'` cycles of simulated time, the non-local operation is performed at the correct time, `t + x + x'`. Although in the simulated system, the

[1]As described in section 3.2.2, local operations are those that affect only the data local to the simulated processor on which they are executed. Non-local operations are those that potentially interact with other parts of the simulated system. Examples of non-local operations are message sends, checks for interrupts, and access to data shared by application threads and message handlers.

`x` cycles of local operations and `x'` cycles of message handling would have been interleaved and here they are not, the simulation is still accurate. The local instructions do not access any of the same data accessed by the message handlers (and vice versa) (by definition, any access of data common to the thread and the message handler is a non-local instruction), so the two sets of instructions can be interleaved in an arbitrary way, as long as both sets execute before the non-local instruction. (This is actually more accurate than sequential Proteus, where non-local operations are executed early).

If, on the other hand, the application thread issues no non-local operations during its allotted 1000 cycles, then after executing the 1000 cycles of local instructions, control goes back to the simulator. The simulator notes the 1000 cycles of work done by the thread, handles any outstanding messages (which take a total of `x''` simulated cycles of work), and schedules a thread execution event to continue the thread at simulated time `t + 1000 + x''`. When an application executes thousands of local instructions at a time, message handling gets caught up every 1000 simulated cycles. Message handling gets caught up completely when a non-local operation is issued. Because message handlers do not share any data with local instructions, it does not matter that message handlers and local instructions are not always executed in the correct order with respect to each other.

The technique as described so far ensures that non-local operations issued by a particular simulated processor occur at the correct time (it takes into account time spent in message handlers) and that message handlers and non-local operations on every simulated processor are ordered as they would be in the simulated system. However, message handlers are not always executed at the correct time. For example, because an application in a long local computation phase only executes message handlers every 1000 cycles, a message handler scheduled to run at time `t` may not execute until time `t + 1000`.

Let's explore the effects of this inaccuracy. There are several different types of operations a message handler can perform that could potentially be affected by this inaccuracy: accessing synchronization variables that other threads will also access, creating and starting a thread, sending a message. If a "late" message handler writes synchronization variables that other threads on the processor will read, then no inaccuracy exists. The reading of these synchronization variables by threads is by definition a non-local event and involves handling all pending messages before being performed. Even though the message handler will not execute at exactly the correct simulated time, it will execute in the correct order with respect to the read of the synchronization variable by the thread and no inaccuracy will be seen.

If a message handler executed at the wrong time starts a thread on the processor, the thread will be late in being submitted to the processor's thread scheduler. However, since the thread scheduling quantum is 10,000 cycles, and the message handler execution is only off by ≤ 1000 cycles, it will not have much effect on when the new thread is run.

However, if a message handler which is executed late sends a message to the network, this message will be sent out to the network late, and this lateness could have cascaded effects at the other processors. For example, consider a message handler scheduled to execute on target processor A at time `t`, which sends a synchronization message to target processor B, and this synchronization message would have arrived at B at time `t + 10`. If the message handler on A is instead executed at time `t + 1000`, the synchronization message will not arrive at B until `t + 1010`, 1000 cycles late. If B was spin-waiting for this synchronization message, the simulated time computed for this application will be off by 1000 cycles. This inaccuracy is not observed in the work reported here because none of the applications I use have message handlers that send out reply messages. Nevertheless, this inaccuracy could be corrected by saving the amount of extra local computation a thread performed without advancing the simulation clock by the entire amount, and then executing the message handlers at the exact simulated time they should be run, and adding in the local computation time later. This would eliminate the inaccuracy. However, this correction has not yet been implemented.

Bibliography

[AR92] Rassul Ayani and Hassan Rajaei. Parallel simulation using conservative time windows. In *Proceedings of the 1992 Winter Simulation Conference*, December 1992.

[ASU86] Alfred A. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Aya89] Rassul Ayani. A parallel simulation scheme based on distances between objects. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, January 1989.

[BB94] Eric A. Brewer and Robert Blumofe. Strata: A multi-layer communications library, version 2.0 beta, February 1994.

[BD92] Eric A. Brewer and Chrysanthos N. Dellarocas. Proteus user documentation, version 0.5, December 1992.

[BDCW91] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel architecture simulator. MIT/LCS 516, Massachusetts Institute of Technology, September 1991.

[Bed95] Robert C. Bedichek. Talisman: fast and accurate multicomputer simulation. In *Proceedings of SIGMETRICS '95*, 1995.

[Bre92] Eric A. Brewer. Aspects of a parallel-architecture simulator. MIT/LCS 527, Massachusetts Institute of Technology, February 1992.

[Bry77] R. E. Bryant. Simulation of packet communication architecture computer systems. MIT/LCS 188, Massachusetts Institute of Technology, November 1977.

[BT] Bertsekas and Tsitsikus. *Parallel and Distributed Computing Numerical Methods.*

[BW95] Douglas C. Burger and David A. Wood. Accuracy vs. performance in parallel simulation of interconnection networks. In *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[CGHM93] David R. Cheriton, Hendrik A. Goosen, Hugh Holbrook, and Philip Machanick. Restructuring a parallel simulation to improve behavior in a shared-memory multiprocessor: The value of distributed synchronization. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, pages 159–162, May 1993.

[CM79] K. Mani Chandy and Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[CM81] K. Mani Chandy and Jayadev Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[Cor92] Thinking Machines Corporation. *CM-5 Technical Summary.* 1992.

[CS89] K. M. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, January 1989.

[DHN94] Philip M. Dickens, Philip Heidelberger, and David M. Nicol. Parallelized direct execution simulation of message-passing parallel programs. Technical Report 94-50, ICASE, June 1994.

[Fuj88] Richard M. Fujimoto. Performance measurements of distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 14–20, February 1988.

[Fuj90] Richard M Fujimoto. Parallel discrete event simulation. *Communiations of the ACM*, 33(10):30–53, October 1990.

[Gol93] Stephen R. Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance.* PhD thesis, Stanford University, June 1993.

[JCM93] Kirk Johnson, David Chaiken, and Alan Mainwaring. Nwo-p: Parallel simulation of the alewife machine. In *Proceedings of the 1993 MIT Student Workshop on Supercomputing Technologies*, August 1993.

[Joh95] Kirk Johnson. Personal communication. March 1995.

[LL89] Yi-Bing Lin and Edward D. Lazowska. Exploiting lookahead in parallel simulation. Technical Report 89-10-06, University of Washington Department of Computer Science and Engineering, October 1989.

[Lub88] Boris D. Lubachevsky. Bounded lag distributed discrete event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 183–191, January 1988.

[Lub89a] Boris D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–123, January 1989.

[Lub89b] Boris D. Lubachevsky. Scalability of the bounded lag distributed discrete event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 100–107, March 1989.

[Nic91] David M. Nicol. Performance bounds on parallel self-initiating discrete-event simulations. *ACM Transactions on Modeling and Computer Simulations*, 1(1):24–50, 1991.

[RHL+93] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C.Lewis, and David A. Wood. The wisconsin wind tunnel: Virtual prototyping of parallel compuers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, May 1993.

[SBW88] Lisa M. Sokol, Duke P. Briscoe, and Alexis P. Wieland. Mtw: A strategy for scheduling discrete simulation events for concurrent execution. In *Proceedings of the 1988 SCS multiconference on distributed simulation*, January 1988.

[SDRG+86] K. So, F. Darema-Rogers, D.A. George, V.A. Norton, and G.F. Pfister. Psimul - a system for parallel simulation of the execution of parallel programs. RC 11674, IBM T.J. Watson Research Center, January 1986.

[Ste91] J. Steinman. Speedes: synchronous parallel environment for emulation and discrete event simulation. *Proceedings of the SCS western multiconference on advances in parallel and distributed simulation*, 23(1):95–103, 1991.

[vECGS92] T. von Eiken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[YA93] Donald Yeung and Anant Agarwal. Experience with fine-grain synchronization in mimd machines for preconditioned conjugate gradient. In *Proceedings of the 4th Symposium on principles and practice of parallel programming*, May 1993.