Parallel Network Simulation Techniques

by

Pearl Tsai

B.S., Yale University (1992)

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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author ....
Department of Electrical Engineering and Computer Science
May 12, 1995

Certified by .... [illegible] ....
William E. Weihl
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .... [illegible] ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Parallel Network Simulation Techniques

by

Pearl Tsai

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

The choice of network simulation techniques in parallel discrete event simulation of multiprocessor computers can affect overall performance by an order of magnitude. The choice can also affect the accuracy of the results of the simulation by a factor of two or more. Accordingly, it is important for users of parallel simulators to be aware of the available options and their implications. This thesis presents several techniques for parallel network simulation, evaluates their performance on different types of applications and network architectures, and provides the user with guidelines for trading off accuracy and performance when using hop-by-hop, analytical, topology-dependent, and combined network models. The hop-by-hop model accurately simulates the travel of messages hop by hop through a network. The analytical model uses a mathematical model to estimate the delay a message will encounter en route to its destination, and sends it directly to the destination. The topology-dependent model assumes that all messages take the same amount of time to arrive at their destinations for a particular network topology, and ignores the effects of network contention. The combined network model dynamically switches between the analytical and hop-by-hop models depending on the level of contention present in the network. This research was performed in the context of Parallel Proteus, a parallel simulator of message-passing multiprocessors.

Thesis Supervisor: William E. Weihl
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisor, Bill Weihl, for all his help, insight, and suggestions over the last three years. I am grateful for his guidance when I lacked focus, his careful pointing out of the lapses in my logic, and his patience when I was slow to comprehend. These past three years have been a very educational and enriching period of my life, and I'm sure my new knowledge will serve me well in the years to come.

I would like to thank my mentor at AT&T Bell Laboratories, Phillip Gibbons, for his advice and encouragement. Phil provided me with my introduction to research and started me on the path towards this work.

Thanks to Ulana Legedza for all the tips on Parallel Proteus, for helping me debug my code, and for many interesting conversations about parallel simulation. Thanks to Eric Brewer and Anthony Joseph for introducing me to Proteus and answering more questions than I can count. Thanks as well to all the members of the Parallel Software Group, including Kavita Bala, Patrick Sobalvarro, and Carl Waldspurger, for being there when I needed help and comments, and for providing a friendly group environment over the last few years.

I'd also like to thank Stephen Renaker for being a wonderful friend and typing for me when my wrists simply couldn't take it any more. Many thanks as well to Elizabeth Wilmer for providing reassurance and much-needed distraction from thesis angst.

Finally, I would like to thank my parents, Nien-Tszr and Elizabeth Tsai, for their love and support. Without them none of this work would have been possible.

Contents

List of Figures

Chapter 1

Introduction

Simulation of multiprocessor computers is a popular and effective research tool for computer scientists. Actual multiprocessors are expensive and it is rare that a researcher will have access to several different ones. Also, a researcher has to invest a lot of time and energy in learning how to use each new computer and in porting code between them. A general-purpose retargetable simulator that can simulate the behavior of many different types of multiprocessors is useful for cross-architectural comparison studies. For instance, the same code can be used and the parameters of the target machine tweaked slightly to study the performance of an algorithm on different architectures. Simulators, with their flexible development environments and ease of instrumentation, are also very useful for designing new architectures or developing application programs.

Sequential simulators that run on a workstation have been used by scientists for many years. However, workstations have limited processing power, memory, and disk space. Many simulations would take an unreasonably long time to run to completion on a workstation, or not be able to run at all because of the vast amounts of data required as inputs for the simulated problem. As people desire to investigate the possibilities of ever larger computer architectures, or run ever more detailed simulations of biological or physical systems, the limits of running on a single processor become increasingly problematic. Having a simulator that could run in parallel on an actual multiprocessor opens up wider areas for research.

Parallel simulators make use of powerful host machines and their large amounts of memory and storage capacity to simulate applications on multiprocessors. These multiprocessors can have large numbers of simulated processors, with complex networks connecting the processors. For accurate simulations of applications, it is necessary to simulate the interaction of the messages they send with the interprocessor connection networks. Traditionally, it has been thought that the only way to perform accurate simulations was through exact modeling of every step the messages traveled through the network. This can be a very time-consuming procedure which causes the entire simulation to run slowly.

This thesis describes several alternatives to exact network simulation that reduce the time required to perform network simulation without reducing its accuracy. Many of these concepts are not new, but a study of their relative performance and accuracy tradeoffs has not been done before. Analytical models, average delay models, topology-dependent models, and combined models have the potential to improve simulation performance over the default hop-by-hop model, but only under certain conditions. It is important to define what these conditions are and help users of parallel simulators decide when to use which network simulation technique.

The following chapter explains basic terminology and concepts of parallel discrete event simulation, and describes Proteus and Parallel Proteus, the simulators in the context of which this research was performed. Subsequent chapters present the main issues involved in parallel network simulation, describe the different network models that I examined, evaluate the performance and accuracy of the network models when used on a varied set of applications, and discuss related work.

Chapter 2

Parallel Simulation

2.1 Proteus

Proteus is a sequential discrete-event simulator of multiprocessors that was created at MIT as the masters' thesis work of Brewer and Dellarocas[BDC+91],[Bre92],[Del91]. It allows users to select a parallel machine architecture, run a C program that calls special library routines to simulate interaction between processors, and collect detailed timing information about the program and its behavior on that particular architecture. It currently runs on a DECstation or SPARCstation.

The discrete-event simulation model used in Proteus assumes that the system being simulated only changes state at discrete points in simulated time. These state changes are characterized as timestamped events, which are executed in non-decreasing timestamp order. This is in contrast to cycle-by-cycle simulators, which simulate every component of a machine at every clock cycle, and typically run one or two orders of magnitude slower than event-driven simulators.

Proteus is highly modular and easily configurable. Users can select from a wide range of parameters for multiple-input multiple-data (MIMD) machine architectures. It is possible to imitate most commercially available parallel computers, if the user knows their specifications. If Proteus does not support the desired configuration, its modular design allows users to write their own modules which will provide the correct behavior.

2.2 Parallel Proteus

A major limitation of Proteus is its workstation host. Due to memory and processor speed restrictions, it is generally not feasible to run large simulations with more than 256 virtual processors, or realistic applications such as those that one would want to run on an actual parallel machine. Therefore, Legedza developed Parallel Proteus, a parallelization of Proteus that runs on the CM-5, as part of her master's thesis work[Leg95].

In parallel discrete event simulation, the system being simulated is divided into entities. The simulator consists of many logical processes (LPs), each of which simulates one or more entities. For example, in Parallel Proteus, each processor of the simulated multiprocessor, also known as a virtual processor, corresponds to an entity. In addition, there must exist some sort of interconnection network between the processors in the simulated system, composed of switches and wires connecting the switches. Pieces of the network can also constitute entities. Each physical processor of the CM-5 host runs a LP that simulates one or more of these entities.

When communication occurs between entities in the simulated machine, LPs create events that need to be executed by other LPs. The LPs do not share any global data structures, but rather communicate via timestamped messages. Each LP maintains its own local clock. Saying that physical processor P is at time t means that the LP on P has already executed all events on all its entities with timestamps smaller than t. Saying that virtual processor p_i is at time t means that the LP on its host physical processor P is about to execute an event for p_i with timestamp t.

For any given simulated architecture, the wire delay quantum refers to the number of cycles it takes for a message to traverse the shortest wire in the network. The message quantum refers to the minimum number of simulated cycles it takes for a message to travel between any two processors, including all wire and switch delays.

Chapter 3

Network Simulation Issues

Simulation of the interconnection network is a difficult problem in parallel discrete event simulation. Depending on the simulated machine and application characteristics, it is possible to spend over 90% of the running time on network simulation alone. Correctness, speed, accuracy, and relevance to the user's simulation needs are all important issues to consider when developing network modules.

3.1 Correctness

A difficult problem in parallel network simulation is correctness: ensuring that the simulated messages arrive at the receiving entity at the correct simulated time. In sequential simulation, this is easy, because all events take place on one physical processor and thus can be ordered exactly. On a parallel machine, the message may have to travel across many physical processors and interact with other messages in the network at the appropriate simulated times. In general, the more events are serialized, the easier it is to achieve correctness, but the slower the simulation runs, as the parallel speedup is lost. This is closely linked to the problem of synchronizing the events on the different physical processors.

In the host machine, there is no guarantee that a processor will receive a message by a particular point in time unless the sender and the recipient synchronize at that point. One way to assure that all the processors have reached the same point, and therefore received all messages, is to execute a global barrier. When a global barrier is executed every time a fixed number of simulated cycles has elapsed, that fixed time period is known as the synchronization quantum. Periodic global barriers are very useful, and are the most common synchronization method used in parallel simulators. They are also the simplest known method of synchronization; other methods are more complicated to implement and involve more overhead.

Global barriers are the default synchronization method in Parallel Proteus, and will be the only method considered in this thesis. However, barriers are also expensive. The time required for the communication between processors that establishes the barrier can be substantial, depending on how well global operations are supported in the host machine. Also, if there is significant skew in the arrival times of the processors at the barrier, those that arrive early must waste time sitting idle while waiting for the others. The high cost of global barriers creates a strong incentive for network simulation to find a way to keep the synchronization quantum as long as possible and thus execute as few barriers as possible.

3.2 Speed and Accuracy

In the world of network simulation, speed and accuracy are the two holy grails. Users want simulators that run the most complicated programs in the shortest possible time, and want the results to be exactly the same as if they had been run directly on the simulated machine. Unfortunately, detailed simulations generally take longer to run than approximations, so tradeoffs are necessary. Parallel Proteus is designed to be a truly general-purpose simulator, so it is important to support the user who wants speed as well as the user who wants accuracy. This is especially true if both those users are the same person. In many instances, it is the applications with irregular communication patterns that are most interesting to examine under different architectures, so it should be possible to simulate all types of applications equally accurately.

Contention is a measure of how congested a network is. If there are many messages all trying to travel across the same wires at the same time, some will get to use the wires and others will have to wait. Messages arrive at their destinations later than they would have in the absence of contention, and all these delays add to the total simulated running time of the program. Hot spots occur when many messages contend for one link in the network. This spot has much more traffic than the surrounding areas, and long delays occur at this one spot even though the rest of the network may be free of contention.

Measuring the delay due to contention for each message is the crux of the speed-accuracy tradeoff. It is far easier to simulate a network quickly and accurately in the absence of contention. If a program only sent one message, it would be a simple matter to calculate the message's arrival time by counting the number of wires and switches it must travel across to reach its destination and multiplying by the minimum fall-through latency due to wire and switch delays. Since the message would not have to interact with other messages, it could be sent directly to the receiving processor. Unfortunately, the real world is rarely that simple; many schemes have been proposed to track messages and their delays due to interaction, with varying amounts of overhead.

3.3 Trends in Parallel Networks

Another goal of a good simulator is to be a useful tool. For a general-purpose simulator like Proteus, this means being able to simulate the types of parallel machines that most users are interested in. Even in the few years that simulators of multiprocessors have been around, the target machines have changed substantially. New machines tend to have ever-faster networks in which wire and switch delays are only a few clock cycles. Machines from the 1980s like the BBN Butterfly or the Thinking Machines CM-2 took the equivalent of tens of cycles in switch and wire delays. The CM-5, which was first sold in 1992, has a switch delay of 8 cycles and a wire delay of 1 cycle[Eic93]. The IBM SP2, first commercially available in 1994, has a switch delay of 5 cycles and a wire delay of 1 cycle[SSA+94]. The Cray T3D, also introduced in 1994, has a switch delay of 2 cycles and a wire delay of 1 cycle; in addition, changing between network dimensions counts as an extra hop[ST94]. Current research machines which emphasize low message overheads do better yet. The J-Machine[ND91] and Alewife[Joh94] have combined switch and wire delays of only one clock cycle. A simulator that was written under one set of assumptions about network behavior may perform poorly when those assumptions change.

Chapter 4

Network Models

This chapter describes the network and routing models that are most relevant to sequential Proteus and Parallel Proteus. Elements common to all models, as in actual multiprocessors, are the simulated processor, network interface/routing chip, and the wires linking routers and processors. The links are all assumed to be bidirectional, with one channel in each direction.

4.1 Routing Models

There are three principal routing techniques used in actual multiprocessors: store-and-forward, wormhole, and virtual cut-through routing. Any of these may be used in conjunction with algorithms that implement multiple virtual channels per physical wire. Using multiple virtual channels reduces the possibility of deadlock, but does not change the fundamental behavior of the routing techniques. Any of these techniques may also be used with any network topology. The main differences between the routing algorithms lie in the amount of buffer space present at each routing chip and when messages block in the network. Each message packet is typically composed of multiple flow control units, or flits.

Store-and-forward and wormhole routing are based on older techniques like the circuit-switching used in telephone networks and the packet-switching pioneered by ARPA for early data networks connecting computers. Virtual cut-through routing, a more recent technique, was designed to combine the advantages of circuit and packet-switching. There is a large literature on the subject of routing models. Interesting reading on the subject includes [KK79], [Tan81], [Sei85], [FRU92], [RS76], [DS87], and [RS94].

In a store-and-forward network, the entire message packet must arrive at a node and be buffered there before any of it can continue to the next node. Buffer size must be large, usually a small multiple of the maximum message size. Average latencies for messages are high, since the head flit of a message will arrive at a routing chip before the tail flit, but must wait for the tail flit to also arrive before it can continue through the network. Many early multiprocessors used this type of network because it was easy to build. The design of the routing switch can be fairly simple, since it is not necessary to keep track of the different flits of a message across multiple routers.

In wormhole routing, the head flit no longer has to wait for the rest of a message to arrive before continuing to the next router. The head flit of a message packet establishes a path that the flits in its body also take, but the flits will be spread out in a continuous stream along that path. As a result, at any given time, a message may occupy space on several routing switches and wires. Buffer size is a constant, and typically small, number of flits. Since buffer size is not proportional to message size, when the head of a packet is blocked, the body has to remain in its current location. The blocked packet in turn stalls any other message traffic that wishes to use those channels that it occupies. Very little buffer space is needed, so this sort of routing chip is relatively simple to build. Message latency is very low initially, but as packet traffic increases, this type of network saturates rapidly and performance drops.

In virtual cut-through routing, buffer space is constant in the size of the messages, like in store-and-forward routing. However, like in wormhole routing, the head flit of a message may continue ahead of its body. The difference is that there must be enough buffer space on the next routing switch to contain the entire message, or else the head will not be allowed to advance. If the head is blocked, its body will collapse into a buffer on the same routing switch as the head, freeing the wires behind it for other messages to pass. Because it minimizes network congestion and also gets good message latency, virtual cut-through routing is now a popular choice with computer architects. Router complexity has increased to the point that a little bit more buffer space is not a serious obstacle.

4.2 Implementation

The default routing model in Parallel Proteus simulates a virtual cut-through router with infinite buffers. On a real multiprocessor, the buffer would be sized to hold a constant number of message packets. This abstraction has two main effects. First, it simplifies the task of simulating the network, thus improving overall performance. Parallel Proteus does not have to simulate a handshake protocol between the routing switches to determine if there is enough buffer space free to hold a potential incoming message. Second, it creates the possibility that delays from congestion in the network may be underreported. This effect is probably slight in most cases, as modern networks tend to have large buffers on routing switches. On a real machine, if the network buffers did fill up on one switch, delays would spread to its neighboring switches. In Parallel Proteus, delays are confined to the switches on which they originate. Also, sending processors are always able to inject messages into the network, albeit with long delays; in the real world, they might have to resend the message later, taking a few extra cycles which would not be accounted for in Parallel Proteus.

In the routing implementation on Parallel Proteus, the sending processor starts by packaging the message, including its thread ID number, the current timestamp, the sending processor's ID number, and the destination processor's ID number. The sending processor calls a message-handling routine to actually send the packet. The send handler blocks the sending processor for a number of cycles equivalent to the length of the packet in flits, representing the amount of time it takes the processor to send the packet to the router. It then sends the message to the router, scheduling its arrival for the timestamp plus the packet length plus the switch delay. This accounts for the amount of time it takes for the packet to arrive at the router and be ready to go out on a wire.

Each processor has an associated network routing switch that in a k-ary n-cube serves as a NxN+1 crossbar, dispatching messages between the processor and all its neighbors. Messages from the processor and incoming from the network share the outgoing wires equally, neither having priority over the other. The routing handler simulates the router's behavior, selecting which wire the packet will go out on, with the default being dimension-order routing. It then sends the packet to the router on the other end of that wire, scheduling its arrival time for $max(current\_time, wire\_next\_free)+ wire\_delay + switch\_delay$. This accounts for the amount of time it takes for the head flit of the packet to arrive at the next router and again be ready to go out on a wire. Effectively, this creates infinite buffers on incoming wires. The routing handler also updates the time that the outgoing wire is next free. A further assumption is that the switch can feed all its outputs simultaneously. If a user wanted to simulate a more restricted switch, it would be possible to write a more detailed switch simulation module to delay outgoing messages appropriately.

If the router at the other end of the wire is the one associated with the destination processor, the message is sent to a receive handler instead of a routing handler, with the same arrival time. The receive handler immediately dispatches the packet to the recipient processor.

4.3 Sequential Proteus

In sequential Proteus, there are two options for simulating a message-passing network. There is an analytical modeled network and an exact hop-by-hop network.

The analytical model is an approximation based on Agarwal's network model[Aga91], which is fast but can be inaccurate. A sending processor multiplies the number of hops to a message's destination by the minimum per-hop latency, adds an estimated figure for network contention, and sends the message directly to its destination. Every time a message is sent, the model also updates its global view of contention, so it can account for some slowdown due to congestion. However, since this is a network-wide average, it does not capture the effects of hot spots or other uneven communications patterns in the network.

The exact network model maintains global data structures representing the states of all of the elements of the interconnection network. As each message arrives at a switch, an outgoing wire is selected for it, the data structures are updated, and an event is scheduled for the time that it will arrive at the next switch in the network. The message follows the path it would take in the simulated machine every step of the way; thus, the name hop-by-hop. This simulates contention in a network with high accuracy, but is on average 2-4 times slower than the modeled network[BDC+91].

4.4 Parallel Proteus

Parallelizing the network simulation adds the additional complexity of synchronizing the physical processors, dealing with separate address spaces, and the need to have virtual processors actually communicate through messages rather than instantaneously scheduling events for each other. I present several different mechanisms to consider for trading off speed and accuracy when simulating the network in Parallel Proteus.

4.4.1 Analytical Model

It is no longer possible to maintain a global measure of contention when the entities do not share a single global address space. One way to implement a modeled network is to assign each physical processor copies of the contention state variables and then update them every time the processors perform a global barrier. This method should be fast and a reasonable approximation for rough simulations. Depending on how long the synchronization quantums are, this will lead to some local skew, as in between barriers each physical processor will only be receiving contention information from the virtual processors that it is simulating. Compared to the sequential version, this will also tend to underestimate contention. It may be possible to remedy that by extrapolating global contention from the last period between barriers forward into the next period.

4.4.2 Hop-by-Hop Model

The sequential hop-by-hop model parallelizes in a straightforward manner. In a direct network, each physical processor simulates one or more virtual processors and their associated network interface switches. The state for the outgoing wires is kept along with the switches; if there is a queue for a wire, packets are buffered on the sending processor until they reach its head. Then the simulated message is packaged in a CM-5 message and sent across the physical network to the physical processor that is responsible for the network switch at the other end of the wire.

The main drawback of this method is that in order to guarantee correctness under all conditions, it is necessary to set the synchronization quantum to one switch delay plus one wire delay. Especially if one is simulating a multiprocessor with a very fast network, this can cause overall Parallel Proteus performance to be relatively slow. In the worst case, there will be no congestion in the network when a virtual processor a (on physical processor A) sends a minimum-length message to a destination c (which happens to be on physical processor C). The topology of the network is such that the message must pass through an intermediate network routing switch b on processor B. Now assume that the switch delay is 5 cycles and the wire delay is 1 cycle. Therefore, there will be a barrier every 6 cycles, just before $t = 0, 6, 12...$

As an example, message m, which is only 1 flit long, leaves a's network interface at $t = 5$. It will take 1 cycle to cross the wire to b, and 5 cycles to pass through the network circuitry on b and be ready to leave b on an outgoing wire. So at time 5 A sends a CM-5 message to B instructing it to schedule an event to handle m at time 11. Just before $t = 6$ there is a barrier, so the message is guaranteed to arrive at B by time 6, so it can be scheduled in a timely manner. At $t = 11$, m leaves b's network interface. It will take 1 cycle to cross the wire to c, and 5 cycles to pass through the network switch circuitry on c and be available for use by processor c. So at time 11 B sends a CM-5 message to C instructing it to schedule an event to handle m at time 17. Just before $t = 12$ there is a barrier, so the message is guaranteed to arrive at C by time 12, so it can be scheduled in a timely manner.

4.4.3 Compacted Network Simulation

When simulating indirect networks like BBN's butterfly or the CM-5's fat-tree, it is not obvious how to map intermediate network switches onto physical processors. One possibility is to assign one or a few physical processors the exclusive responsibility for simulating the network. If there is only one network processor, the rest of the processors send all their messages to it, it simulates all the intermediate hops, and then sends messages directly to their vdestinations. If there are a few network processors, each one handles a subset of the other processors, grouped by location. The network processors receive all the messages, pass them among each other for the intermediate hops, and then send messages directly to their destinations.

This method could also be used for direct networks. A possible advantage is that this method would generate fewer actual messages overall. Network processors would be more likely to be able to process intermediate hops themselves, rather than needing to send the message to another location. A disadvantage is that this might cause a bottleneck as all the messages funnel into and out of one processor, but one could experiment with varying the fraction of processors dedicated to the network.

4.4.4 Contention-Free Models

If an application does not generate much congestion in the network, one way to attempt to improve performance is by ignoring the effects of congestion.

A variable delay model calculates the number of hops a message must travel, multiplies by the message quantum to produce arrival time, and sends the message directly to its destination. The processors must synchronize once every message quantum, because a message from a processor to its nearest neighbor will get there in only one hop.

An average delay model uses information from a previous simulation of the application to calculate network delay. It takes the actual average of all the message transit times in a specific application on a specific network topology, and uses that as the delivery delay for all messages in subsequent simulations. Messages are delivered directly to their destinations.

A purely topology-dependent model takes the average number of hops that a message will travel for a given network topology, multiplies by the message quantum to produce arrival time, and sends the message directly to its destination. For example, in an 8-ary 2-cube (a two-dimensional mesh of 64 processors), the delivery delay for all messages will be 8 message quantums, assuming a perfectly random communication pattern.

A constant delay model assumes that all messages take a constant amount of time to reach their destinations, typically 100 cycles. This approach is popular with other parallel simulators. However, a drawback is that it completely ignores the effects that different network architectures can have on simulated performance, which is a common reason for using a parallel simulator in the first place.

An interesting side effect of the uniform delay network models, which assume that all messages take the same amount of time to reach their destinations, is that the synchronization quantum can grow much larger than one wire delay plus one switch delay. For these models, a small savings in simulation time will come from reductions in overhead, such as not having to calculate how long a message will take to reach its destination. However, a much larger performance improvement is expected from not needing to synchronize as often, eliminating many global barriers and their attendant cost and waiting times.

4.4.5 Dynamically Combining Models

A combination of different simulation techniques may also prove useful. Some applications that alternate between communication and computation phases exhibit only sporadic contention. To improve performance, the network simulator switches between the analytical model and the hop-by-hop model, depending on the level of contention. Each physical processor monitors the network traffic in its local area. When using the analytical model, a global OR of all the processors is taken at every barrier, and if any one signals that its messages are experiencing delays of more than a fixed percentage of their base transit time, all processors switch to the hop-by-hop simulator. When using the hop-by-hop model, the reverse is true; at every barrier, a global AND of all the processors is taken, and if none signal that their messages are experiencing delays of less than that fixed percentage of their base transit time, all processors switch to the modeled simulator. Any messages in transit are simply sent to their destinations.

The exact percentages used for the threshold values should be determined through experimentation. It may be best to leave them as user-settable parameters. Setting them too low may result in the simulation running unnecessarily slowly; setting them too high may result in underreporting of congestion delays. The threshold for switching from the analytical to the hop-by-hop network should be set higher than the threshold for switching from the hop-by-hop to the analytical network. The difference introduces some hysteresis into the simulation and prevents thrashing between models if the congestion levels happen to be around the threshold value.

4.4.6 Other Models

There are other techniques that relax the correctness requirements of the hop-by-hop network but still use information gained from going through all the hops to estimate contention. These involve more overhead than the preceding techniques, and it is not clear if their improvements in accuracy would be worth the cost.

In one such method, the sending processor would use a modeled lookahead mechanism to deliver the contents of a message to the receiving processor while the message header works its way through the network hop by hop. The information in the message can be used at close to the correct simulated time, even though the simulation of congestion delays may lag behind.

The Wisconsin Wind Tunnel[BW95] has the sending processor deliver timing messages to all the intermediate network switches along the path of the message. The intermediate points separately estimate the local delay due to contention for the message and send their estimates to the receiving processor, which sums all the estimates to arrive at the total delay for the message. This method has the advantage of parallelizing the delay computation, but it also produces much more work for the sending and receiving processors, and doubles the total actual message traffic in the network.

Chapter 5

Experiments

This chapter presents and discusses performance and accuracy results for the parallel network simulation techniques described in the previous chapter.

The overall performance of Parallel Proteus as well as the results of the simulation depends strongly on the interaction of the tested application's communications characteristics with the simulated machine parameters. What may be a light workload for one target architecture may be crippling for another. In general, the higher the performance of the simulated network, the slower Parallel Proteus will run. A network that can deliver messages every simulated clock cycle will be harder to simulate than a network that takes a minimum of 20 cycles to deliver a message between neighboring processors. Obviously, however, the amount of data that the application sends over the network also matters. If an application rarely sends messages, Parallel Proteus will not have to spend much time delivering messages, no matter whether the network is slow or fast.

Before discussing results, I provide some background to help interpret what these results mean to the user of Parallel Proteus. I define general categories of applications, and describe the specific applications used to test the network models.

5.1 Program Categorization

To simplify discussion, I classify simulation runs on Parallel Proteus into three broadly defined groups by their communications demands on the network: low, moderate, and heavy contention. These groups reflect how the communication patterns of a program can affect its overall simulation time. Since Parallel Proteus currently only supports message-passing programs, communication means explicit message exchange between processors.

The groupings are also relative to machine architecture. The same application can run on a multiprocessor with a fast network and experience no contention, or on one with a slow network and take double the amount of time to run, because of contention in the network.

There are two key indicators for determining into which group an application running on a particular machine architecture will fall. One is the percentage of the total simulated running time that is caused by network congestion delays. For the low contention group, that number should be below 5%, and for moderate contention, below 20%. The second is the average length of time a message is delayed at any given hop in its path, if it is delayed at all. For the low and moderate contention groups, this number as a multiple of the base per-hop delay should be under 10. The boundaries between groups are by nature imprecise, but these are useful approximations to make.

5.1.1 Low Contention

Application and machine architecture combinations that do not result in much network contention fall into this category. Messages may be short or infrequent, or the machine may have a very fast network. It does not matter if the communication occurs in phases, is evenly distributed, or falls into some other random pattern, because overall it is still too light to significantly affect the application's total running time. Recent multiprocessor networks tend to be overengineered relative to processor speeds, and thus simulations using modern network characteristics combined with most applications will fall into this category. This category yields the best opportunities for performance improvement.

5.1.2 Moderate Contention

Applications in this category exhibit moderate to heavy communication, but it is evenly distributed. The application may also be programmed in such a way as to hide the effects of communication latency, e.g., a communication phase, followed by a computation phase that does not require the results of the communication (so as to hide latency), and then finally another computation phase that does use the results of the communication. Even though contention exists, it is evenly balanced among the processors, which send about the same number of messages at the same times. There are no hot spots that significantly affect overall running time. This category presents some possibilities for performance improvement, but it may be difficult to correctly discern which network simulation technique will be the optimal one to use.

5.1.3 Heavy Contention

Applications in this category characteristically have heavy and/or highly variable communication. Network contention is high, as evidenced by the long delays that messages face before being able to acquire network links. Hot spots are a significant problem. If communication is very heavy, unless the application is very well matched to its underlying physical network, it is highly unlikely that it can avoid having some hot spots. This category is the most difficult to simulate accurately. It offers little potential for performance improvement through changing network simulation techniques.

5.2 Test Applications

I used three applications to test the different network models: *radix*, *SOR*, and *test* (a synthetic application). The applications have varying computation and communication requirements, and attempt to provide a range of realistic demands on Parallel

Proteus that will match those of other users.

5.2.1 Radix Sort

The algorithm for *radix* uses four distinct phases to sort numbers with d digits and a radix of r. The phases are each repeated d times. First, each processor locally performs a counting sort on its values. Second, the processors perform r staggered scans up and down a binary tree to combine the proper counts for each digit. Third, the processors send their values to the appropriate destinations, each one in a separate message. Fourth, the processors locally update their arrays of numbers. The messages in the second and third phases tend to be short and their destinations randomly distributed.

5.2.2 SOR

Successive overrelaxation of matrices, or *SOR*, is an iterative method of computing an approximate solution to a system of linear equations. Each processor goes through three phases to update each number in its assigned grid, whose value only depends on the values of its neighbors. First, it sends a message to each of its four neighbors containing the values on its borders. Second, while it waits to receive the messages from its neighbors, it updates the values in the middle of its grid. Finally, after it receives the four messages from its neighbors, it updates the numbers on its borders. The algorithm has no explicit global synchronization. Only a small number of long messages are sent, and those go only to each processor's nearest neighbors. The computation phases are also very long.

5.2.3 Test Workload

It was difficult to find actual applications to test all the interesting combinations of computation and communication, so I created a synthetic test program to measure the performance of the network techniques under different conditions. It has two variants.

*Test*1 is designed to test the effects of moderate contention in the network and moderate hot spot activity. It consists of 10 iterations of 2 phases, alternating computation and communication. The computation phase lasts for 3000 cycles, and the total number of messages sent during each iteration varies from 4,000 to 16,000. To generate some hot spots, as might be the case in a producer-consumer application, only a few processors send messages. The messages are sent to random destinations.

*Test*2 is designed to test the effects of switching between network simulation models when levels of contention in the network vary between low, heavy, and moderate. The overall level of contention would be considered moderate, but there are distinct variations in contention over time. *Test*2 consists of 10 iterations of 3 phases. The first phase is pure computation, and lasts for 30,000 cycles. The second is an intense communication phase, where each virtual processor sends 500 to 2,000 messages to random destinations. The third phase intersperses computation and communication; each virtual processor sends one-tenth as many messages as in the previous phase to nearby destinations, waiting 300 cycles in between sending each message.

5.3 Results

This section presents and discusses results for the various network simulation techniques. I ran experiments on 32-node CM-5 partitions. Each CM-5 node is a 33-MHz SPARC processor with 32 MB of RAM, but no virtual memory. I used the hop-by-hop network model, which accurately measures network congestion, as the benchmark against which the less accurate models are compared.

The first three subsections compare the accuracy and performance of the hop-by-hop, analytical, average and topology-dependent network models in scenarios with low, moderate, and heavy levels of network contention. The fourth subsection builds on the knowledge learned by conducting the first three sets of experiments. It presents the results of dynamically combining the hop-by-hop and analytical models for an application with varying levels of contention. The fifth subsection discusses three techniques that offer little benefit under any scenario: the variable delay model, the constant delay model, and compacting the network simulation.

5.3.1 Low Contention Scenarios

The fast networks found on modern multiprocessors tend to minimize the problem of network contention, when lightly loaded. To illustrate this, I ran simulations of the applications *SOR* and *radix* on a 16-ary 2-cube (a total of 256 virtual processors), with a switch delay of 5 cycles and a wire delay of 1 cycle. For both applications, delays due to network congestion accounted for less than one percent of the total simulated running time.

During runs with the exact hop-by-hop and analytical modeled networks, the synchronization quantum was set to 5 cycles, or $switch\_delay + wire\_delay - 1$. The average number of hops that a message must travel from a random starting point to a random destination in a 16-ary 2-cube is 16. So, for the topology-dependent model, I set the synchronization quantum to 96 cycles, or $16 * (switch\_delay + wire\_delay)$. In these applications the average number of hops that messages actually travelled was lower: 11 for radix and 1 for *SOR*. I set the synchronization quantum for the average delay model for *radix* to 66, and for *SOR*, to 5.

Accuracy

For both *radix* and *SOR*, as shown in figures 5-1 and 5-2, the total simulated running time for the simulations that used approximate networks was within 2% of the results for the simulations that used the exact network. There was little difference between average delay network simulations and topology-dependent modeled simulations that did not take the application characteristics into consideration. The results show slightly more variance for *radix* than for *SOR*, reflecting the larger number of messages sent by *radix*.

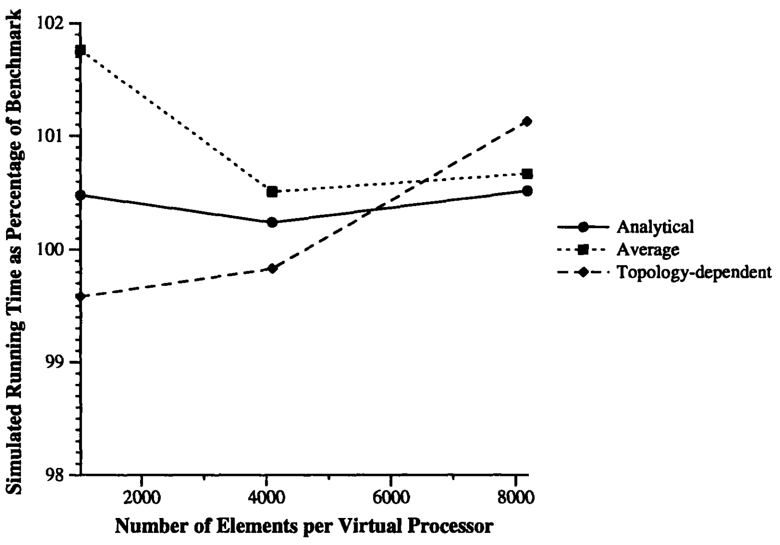

Figure 5-1: **Radix.** Low contention network accuracy results.

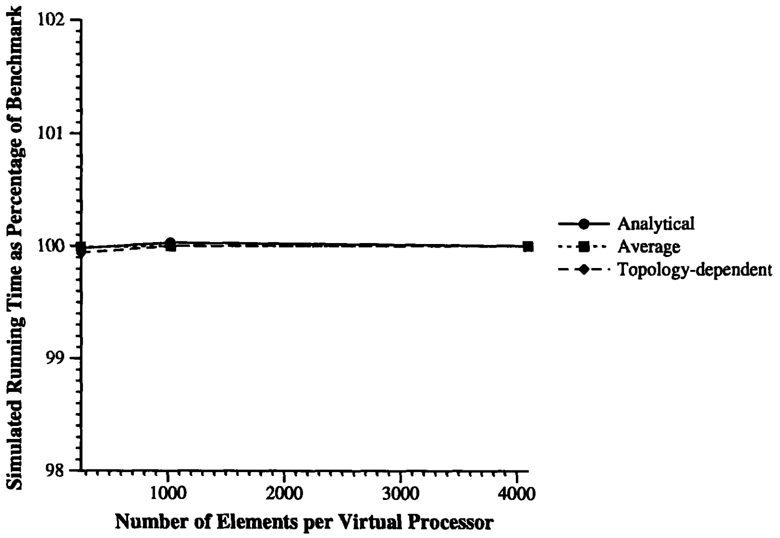

Figure 5-2: **SOR.** Low contention network accuracy results.

Performance

For *radix*, as shown in figure 5-3, the analytical modeled network took 64-79% as much time to run as the hop-by-hop network, reflecting the reduced amount of time devoted to network simulation. For the largest granularity, when each virtual processor operates on 8192 elements, each virtual processor sends an average of 10,000 short messages over the course of a simulation, so the savings can be considerable. The performance results for uniform delay simulations were equally striking. Even though all three took almost the same number of simulated cycles to run, the average delay and topology-dependent simulators took only 13-28% as long to run on the CM-5 as did the hop-by-hop simulator.

The performance improvements for *SOR*, shown in figure 5-4, were not nearly as dramatic as those for *radix*. This is because *SOR* uses the network far less, creating less room for improvement. In the case when each virtual processor operates on 4096 elements, each processor only sends 10 long messages over the course of the entire simulation, and those go to its nearest neighbors, so the network simulator only needs to process one hop per message. The analytical model and average delay models both took 90-93% as long to run as the hop-by-hop network. The topology-dependent model was the only one to achieve significant savings, running in 46-65% of the time of the hop-by-hop network.

Discussion

Most of the time savings for the uniform delay models comes from reducing the total number of global barriers required over the course of the simulation. For the average delay model on *radix*, increasing the synchronization quantum from 5 cycles on the hop-by-hop simulator to 66 cycles means that Parallel Proteus only needs to execute 7.6% as many global barriers. For the topology-dependent models, increasing the synchronization quantum from 5 cycles on the hop-by-hop simulator to 96 cycles means that Parallel Proteus only needs to execute 5.2% as many global barriers.

It initially seems surprising that the uniform delay models are so accurate, but in low contention scenarios, very little of the simulated running time is due to conges-

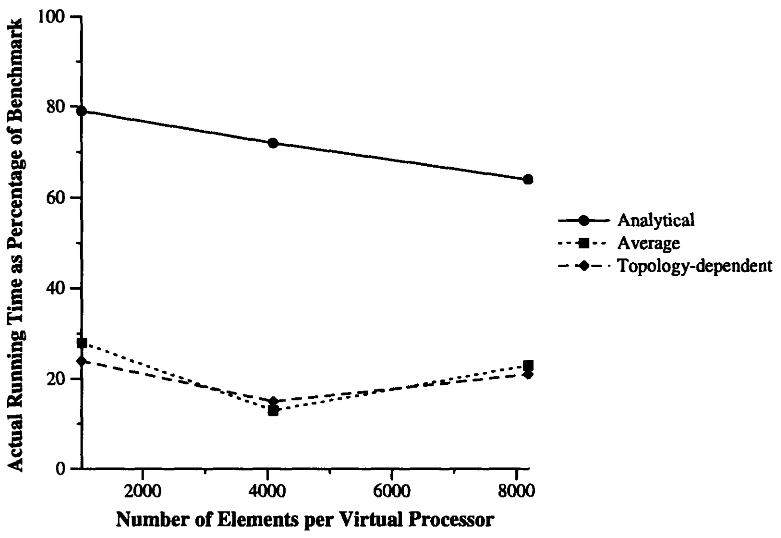

Figure 5-3: **Radix.** Low contention network performance results.

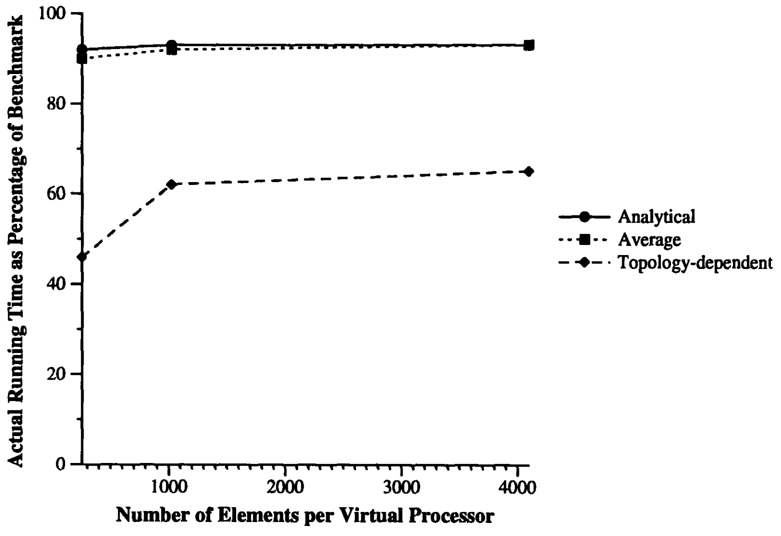

Figure 5-4: **SOR.** Low contention network performance results.

tion delays. Application granularity and the existence of separate communication and computation phases seem to be relatively unimportant, as long as there are no significant congestion delays. In these scenarios, it is clear that using uniform delay network simulations can significantly reduce the time spent on simulating the network, and improve overall Parallel Proteus performance without sacrificing accuracy.

5.3.2 Moderate Contention Scenarios

There exist categories of applications that exhibit moderate network congestion for which using the analytical modeled network offers the optimal balance between speed and accuracy. Neither *SOR* nor *radix* fell into this category, so I ran simulations using *test*1. Results are shown in figures 5-5 and 5-6. In the case where 16,000 messages per iteration are sent, delays due to network congestion account for 20% of the total simulated running time of *test*1.

The analytical model is almost as accurate as the hop-by-hop network simulator, and takes 92-94% as long to run. The topology-dependent model only comes within 79-85% of the simulated running time of the hop-by-hop network. Although its relative performance is quite good, it is not nearly accurate enough for most cross-architectural studies, in which differences of a few percent can be significant. The average model would have produced results very similar to the topology-dependent model, so I did not show experiments using it.

There undoubtedly are many other types of communications patterns in applications which will produce similar results, and it would be difficult to catalog them all. This is an interesting area for further research. For the purposes of this work, it is sufficient to note the existence of moderate contention scenarios.

5.3.3 Heavy Contention Scenarios

For applications that cause heavy contention in the network, using the hop-by-hop simulator is the only way to get accurate results. The network tends to become congested when either it is relatively slow, or the application sends messages that are

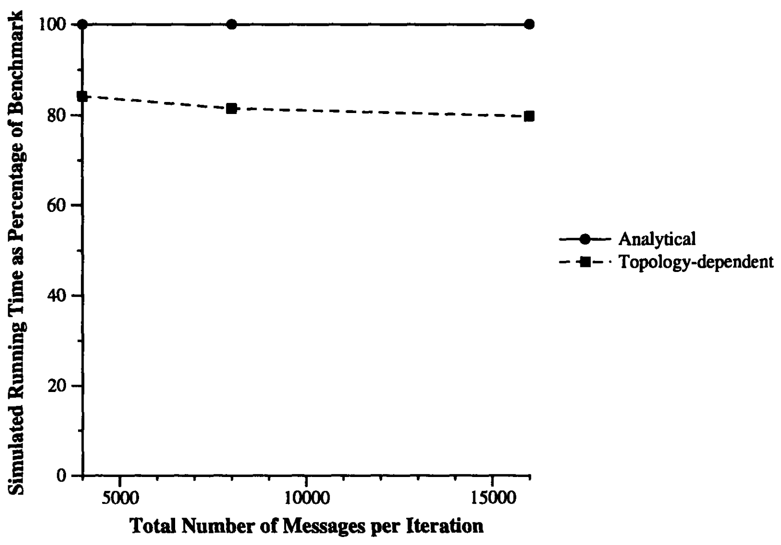

Figure 5-5: **Test1.** Moderate contention network accuracy results.

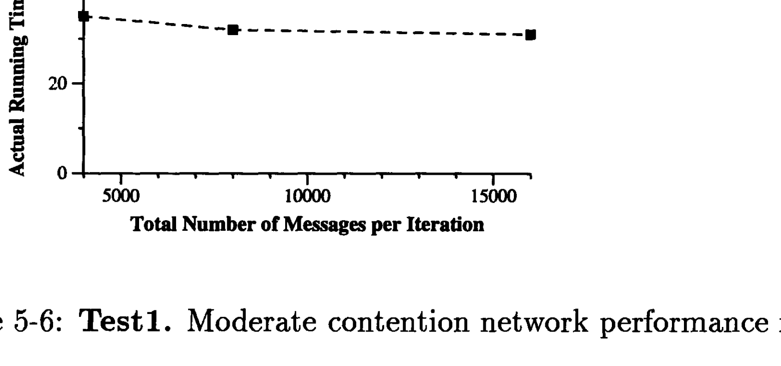

Figure 5-6: **Test1.** Moderate contention network performance results.

numerous and/or long. I ran experiments contrasting the analytical model with the hop-by-hop network; preliminary studies showed that the accuracy of the uniform delay models were even worse than the analytical model, so they are not presented here.

In one experiment, I took *radix* and artificially quadrupled the length of the messages it sends, from one word to four words, while holding the switch delay fixed at 5 cycles. In another experiment, I quadrupled the network latency to a switch delay of 20 cycles instead of 5 cycles, while keeping message length stable at one word. Results are shown in figures 5-7 and 5-8.

When message length was quadrupled, hot spots caused by the longer messages slowed network throughput. The effect was most dramatic with higher granularity, as more messages per virtual processor were sent. For the case of 8192 elements per virtual processor, the simulated running time on the exact network nearly doubled from a base of 3.8 to 6.0 million cycles. By comparison, the analytical network model only recorded an increase in running time from 3.8 to 3.9 million cycles.

Quadrupling the network latency had a similar effect on *radix*. For the case of 8192 elements per virtual processor, simulated running time again jumped from 3.8 to 6.0 million cycles, while the analytical model only posted an increase to 3.9 million cycles.

The performance of the analytical modeled network on *radix* varied significantly with the granularity, doing best when the processors sent the fewest messages.

Similar experiments that quadrupled the network latency and message length in *SOR* had very little effect on either the accuracy or performance of the analytical modeled network, as compared to the base case. *SOR* simply does not have enough message traffic to create problems with network congestion. Those experiments fell into the low contention category.

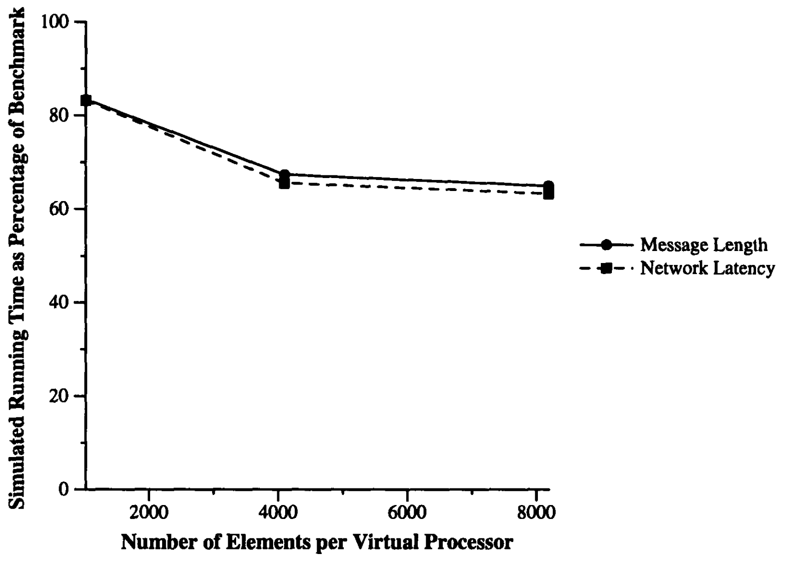

Figure 5-7: **Radix.** Analytical network model accuracy results compared to hop-by-hop model, when either message length or network latency is quadrupled.

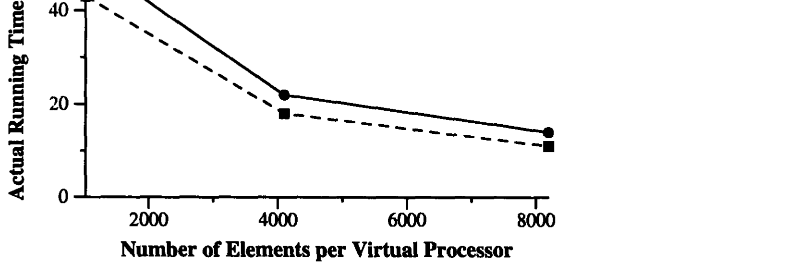

Figure 5-8: **Radix.** Analytical network model performance results compared to hop-by-hop model, when either message length or network latency is quadrupled.

5.3.4 Combining the Models

Not all applications are as dichotomous as *radix*, with its constant barrage of communication, and *SOR*, with its almost complete lack of communication. Applications which alternate between periods of intense communication and low communication can benefit from dynamically switching between the hop-by-hop and analytical network modules. The profile of *test2* fits this combination; the percentage of its running time due to congestion ranged between 10-20%, and the average message delay (for delayed messages) was over 1000 times the base delay.

Results of experiments on *test2* using the combined hop-by-hop and analytical model are shown in figures 5-9 and 5-10. I set the threshold value of per-hop contention delay for switching from the analytical to the hop-by-hop model at 50% of the base delay, and the threshold for switching in the other direction at 250%. These thresholds keep thrashing to a minimum while still permitting dynamic switching when the application's communication characteristics change. As expected, the combined model's accuracy and performance results fall in between those of the analytical model and the hop-by-hop model. Accuracy is quite good, ranging between 96-98%. Performance is closer to that of the hop-by-hop model than that of the analytical model, since during the heavy contention periods, which take the longest to simulate, the hop-by-hop model is in use.

Using the analytical model for the low contention periods does not produce as much of a time savings as using a uniform delay model might. It would be interesting to look at combining the hop-by-hop with the topology-dependent model. However, a metric for determining when to switch to the hop-by-hop model would have to be created, since the topology-dependent model does not calculate contention in the network.

5.3.5 Other Models

The variable delay model does not have significantly less overhead than the analytical model, and its accuracy is consistently lower. Although its accuracy is better than the

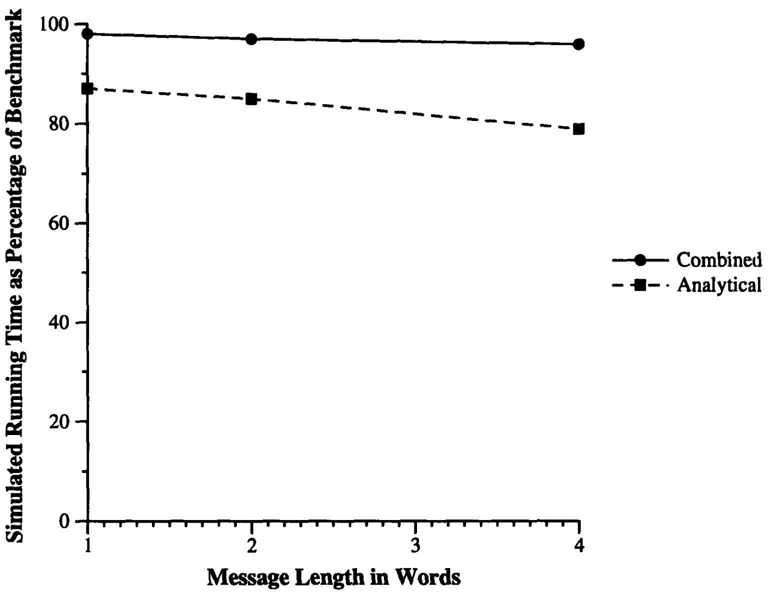

Figure 5-9: **Test2.** Analytical network model and combined analytical/hop-by-hop accuracy results compared to hop-by-hop model.

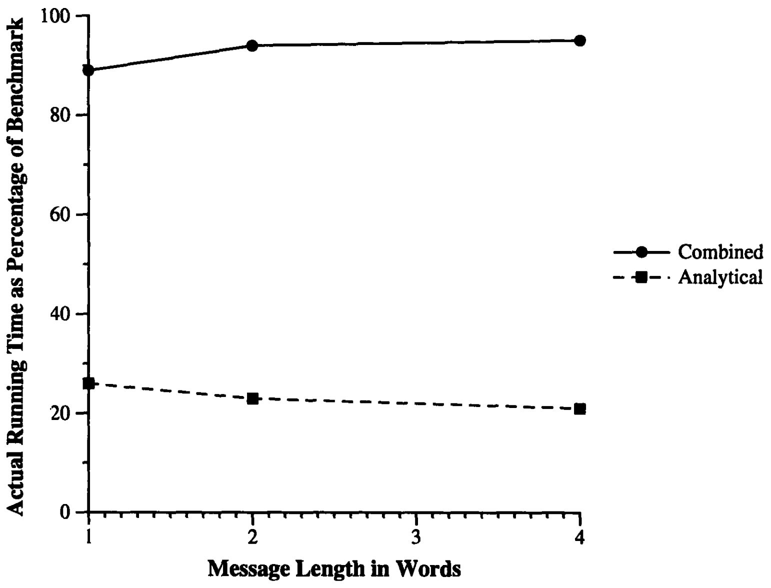

Figure 5-10: **Test2.** Analytical network model and combined analytical/hop-by-hop performance results compared to hop-by-hop model.

uniform delay models, that advantage is very slight, and it is unable to offer any of the performance benefits of eliminating global barriers. Therefore, I did not study it in great detail. Similarly, the constant delay model does not offer any advantages over the topology-dependent model, unless the interconnection network has a very small diameter. In addition, it has the downside of not accounting for the interconnection network at all.

Compacting the network simulation onto one or a few physical processors did not improve performance for either *radix* or *SOR*. For *radix*, even though the total number of messages sent on the CM-5 network decreased, the performance of Parallel Proteus worsened as the number of simulated messages increased. *Radix* sends many messages, but has relatively little computation in between. The processors dedicated to network simulation spend a lot of time on the overhead costs of processing messages, while the processors dedicated to simulating the virtual processors sit idle. For *SOR*, it is not surprising that there was no performance benefit, since the total number of messages sent is so small, and dedicating processors to network simulation only means that there are fewer processors available to perform the rest of the simulation. Changing the load distribution so that some physical processors had exclusive responsibility for simulating the network, but also simulated a few virtual processors as well, had little effect on performance. The performance of the CM-5 network does not appear to be a limiting factor for Parallel Proteus, but this might be different on another host machine.

5.4 Discussion

Experiments have shown that for programs that produce little congestion in the network, surprisingly simple network models can produce simulation results within 1% percent of those of exact hop-by-hop simulations, while running up to 700% faster. These same models, under different conditions, can also produce extremely inaccurate results. The challenge is therefore for a user of Parallel Proteus to be able to correctly choose between the available models based on her needs for accuracy and performance.

Performing one or two simulations of an application using the hop-by-hop network produces information that can help determine which network model to use in subsequent runs of the application. In order to gain insight into an issue, users of parallel simulators typically perform many runs of the same application under slightly differing conditions, so this capability can prove very useful in the long term.

One key indicator is the percentage of the total simulated running time that is caused by network congestion delays. The time due to congestion delays can be measured by subtracting the results of a hop-by-hop simulation with the contention measurements disabled from a standard run that produces the correct running time. In the low contention scenarios examined here, that number was consistently below 5%. In the moderate contention scenarios, it was about 10-20%, and in the heavy contention scenarios, it was about 30-40%. It is certainly possible to have applications for which an even higher percentage of the simulated time is due to congestion. The contention-free models become consistently less accurate as this percentage rises. This is as expected, since they do not try to account for contention. Since the topology-dependent model offers the best performance, the user can decide whether to use it depending on how much inaccuracy she is willing to tolerate, which may in turn depend on the magnitude of the effects she is trying to measure.

Trying to determine when to use the analytical model is more complex. It will certainly be at least as accurate as the topology-dependent model, but in low contention scenarios the topology-dependent model performs much better. Much more exhaustive experimentation needs to be done to establish the boundaries of moderate contention scenarios within which the analytical model does well; for example, will it work well for most or all applications in which the percentage of running time due to congestion delays is under 20%? In the meantime, another important indicator is the average length of time a message is delayed due to congestion at any given hop in its path, if it is delayed at all. In the low and moderate contention scenarios, this number as a multiple of the base per-hop delay without congestion was under 100, while in the heavy contention scenarios the number was over 1000. Due to the nature of the analytical model used to calculate network delays, it does not handle heavy congestion well.

For applications that lie in the area between moderate and heavy contention, having periods of heavy and light network traffic, using a dynamic combination of the analytical and hop-by-hop networks is the best solution. Performance will be better than the hop-by-hop if the periods of light network traffic are longer, while accuracy will be higher than the analytical model if the periods of heavy network traffic are longer. There is extra overhead associated with determining when to switch between models in this technique, which is why it is better to use either the analytical or the hop-by-hop models in isolation if the choice is clear.

As for the exact hop-by-hop network, it is the only choice for accurate simulation when there is heavy contention in the network. According to the guidelines above, it should definitely be used if the percentage of running time due to congestion delays is above 30% or if the per-hop message delay is more than 1000 times the base delay. Below those thresholds is a grayer area, where it may be desirable to use the hop-by-hop network to assure accuracy, but to the possible detriment of overall performance.

It is always possible for the user of Parallel Proteus to just take stabs in the dark and compare all the different models against the benchmark. It may even be desirable, if a user wishes to use a specific model repeatedly, to compare it against the benchmark and ensure its accuracy. This discussion is intended to forestall some of that testing and provide a framework within which it can take place. The potential time savings are definitely worth a little bit of preliminary comparison, since the typical user plans to run multiple simulations. This is especially true with the amazing speed of networks in modern multiprocessors, which permits a wide range of application and machine architecture combinations to be considered low contention scenarios.

Chapter 6

Related Work

Parallel and distributed simulation is an active field that has been in existence since the late 1970s. Much of the activity involves military wargame simulation or specialized circuit or scientific simulators. The theoretical papers have tended to focus on different synchronization protocols. Fujimoto presents an excellent survey of the field in [Fuj90]. However, little of this work is directly relevant to improving network simulation techniques, because it does not involve simulating interconnection networks. The effects of hot spots in a parallel network have been investigated in [Dal90] and [PN85].

Very few general-purpose multiprocessor program and architecture simulators have been developed that run on actual parallel machines. This chapter discusses the simulators most closely related to Parallel Proteus and how they handle network simulation. Legedza's modifications to Parallel Proteus' barrier scheduling mechanisms have similar speed and accuracy goals, and I will discuss how my work complements hers to improve overall simulator performance.

6.1 LAPSE

The Large Application Parallel Simulation Environment (LAPSE)[DHN94], developed at ICASE by Dickens, Heidelberger, and Nicol, is a parallel simulator of message-passing programs that runs on an Intel Paragon. Its performance relies on the assumption that many message-passing numerical codes have long intervals of computation followed by short periods of communication, so that lookahead is high. Its application code runs ahead of the simulation process and generates a timeline of message events, which are used to schedule periodic global barriers. In the "windows" between barriers, entities perform pairwise synchronization through a system of appointments. Each appointment represents a lower bound on the arrival time of a message, and is updated as the simulation progresses and more accurate timing information becomes available.

There are a number of issues that limit the applicability of LAPSE's results. First, its primary goal is to support analysis of Paragon codes, so its network simulation/synchronization protocol takes advantage of the fact that the Paragon's primary method of interprocessor communication is explicit send/receive messaging. Therefore, it is usually possible to predict when the effects of a message will first be noticed, as opposed to in the CM-5, where active messages can be received at any point. If LAPSE was extended to handle a general-purpose multiprocessor, it would need to send far more messages in the average case and therefore experience a significant slowdown, in order to ensure that messages were received before they could affect the results of another processor's computation. Second, if the simulated programs communicate frequently, lowering the lookahead, performance also drops. Third, LAPSE uses a contention-free network model, so its results will be inaccurate for high-contention programs.

6.2 Wisconsin Wind Tunnel

The Wisconsin Wind Tunnel (WWT) was developed at the University of Wisconsin by Reinhardt, Hill, Larus, Lebeck, Lewis, and Wood[RHL+93]. It is a multiprocessor simulator that, like Parallel Proteus, runs on the CM-5. In its original design, it only simulated shared-memory architectures, and assumed that all interprocessor communication took 100 cycles, making no attempt to simulate different interconnection network topologies or network contention.

Burger[BW95] later implemented an exact network simulator for the WWT that ran entirely on one physical node of the CM-5. This solved the problem of synchronizing network interactions by centralizing them on one node. The drawback to this is that it created a serialized bottleneck as well, since he synchronized at the end of every message quantum, ran the network processor while all the others sat idle, then synchronized again to ensure message delivery. On a 32-processor run the exact simulator was an average of 10 times slower than the original version, and this slowdown factor would only increase with the size of the simulation.

Burger also implemented four distributed approximations: first, one that assigned each message a constant delay based on the result of an earlier run on the exact simulator; second, a variable-delay simulator that took into account network topology but not contention; third, a variable-delay simulator that added a contention delay based on an earlier run of the exact simulator; and fourth, one that estimated contention separately for each wire, based on an average of past global information. Some of these did well on average, but when simulating applications with irregular patterns of contention, their performance degraded severely. There was a conscious decision to emphasize speed over accuracy, under the assumption that most users of parallel simulators would not require exact interconnection network simulation for their work.

6.3 Tango

Tango Lite[Gol93] is a discrete-event simulator developed at Stanford that runs on a workstation and is very similar to Proteus. Goldschmidt and Herrod worked on parallelizing Tango Lite, porting it to the DASH shared-memory multiprocessor. They tried using two different synchronization methods: one that relaxed the ordering of memory references, and one that imitated the original WWT and assumed a constant-delay communication latency of 100 cycles. However, they had a difficult time obtaining speedup and abandoned the project[Her94]. Their ability to test parallel Tango Lite was limited by the small size of DASH, which is an experimental machine. The largest simulations they could run were 32 simulated processors on 8 physical processors. Using all 8 physical processors only cut in half the time it took to run the simulation using only one processor.

6.4 Synchronization in Proteus

Legedza examined two synchronization alternatives to periodic global barriers for Parallel Proteus, local barriers and predictive barrier scheduling[Leg95]. These methods improve speedup without sacrificing accuracy, and complement the techniques outlined in this thesis.

Local barrier synchronization exploits the fact that it is only crucial for a processor's simulated time to stay within one message quantum of its immediate neighbors in the simulated network. Therefore, any given host processor only needs to participate in barriers with its neighbors, and once it has done so, it can go ahead and simulate through the next synchronization quantum, although one of its neighbors may still be waiting for another barrier to complete. This looser synchronization of the processors improves performance when work is unevenly divided among the processors in each quantum, yet averages out overall.

Predictive barrier scheduling takes advantage of the fact that sometimes there are long periods of time during which processors do not actually communicate with each other. Thus, it is not necessary to actually perform barriers for every synchronization quantum. This improves performance by eliminating many of the barriers and thus the time spent idle while waiting for them.

Any of the network simulation techniques could run at the same time as local barrier synchronization or predictive barrier scheduling. The combined performance improvements might not be as dramatic as the separate results, however. For instance, network techniques that involve lengthening the synchronization quantum increase the chances that processors will communicate during any quantum, and therefore decrease the likelihood that predictive barrier scheduling will find any unnecessary barriers.

Chapter 7

Conclusions

The choice of techniques used for parallel network simulation can have a dramatic effect on overall simulator accuracy and performance. It is possible for a user to run a simulation in a tenth the time and still maintain 100% accuracy, if the conditions are right. It is also possible for a simulation to return completely incorrect timing information if the wrong network simulation technique is chosen under the wrong conditions. Users of multiprocessor simulators have typically had little control over this important decision. They may have faced a choice between a "fast, inaccurate" or "slow, accurate" network simulation, but without any information about the actual speed and accuracy tradeoffs.

In this thesis, I have presented a variety of network simulation techniques for Parallel Proteus, and provided guidelines to help the user choose between hop-by-hop, analytical, topology-dependent, and a combination of those network models. If the percentage of the total simulated running time that is caused by network congestion delays is under 5%, the topology-dependent model should be used. If that percentage is under 20% and the average per-hop delay due to congestion is under 100 times the base delay, the analytical model should be used. If the total contention delay is under 20% and the per-hop delay is over 100 times, or the total delay is between 20 and 30%, the combination of the analytical and hop-by-hop models should be used. If the total contention delay is over 30% of the running time, then the hop-by-hop model should be used.

There are many opportunities for refinements or extensions to this work. Much more thorough experimentation should be done to further specify the guidelines for choosing between the models, and provide a graph of accuracy and performance versus contention for each network model. All of this work was also done using virtual cut-through routing. If a user wishes to use store-and-forward or wormhole routing, conditions would change slightly and possibly alter the optimal guidelines. For any given application/architecture combination, store-and-forward routing would tend to lower the congestion seen in the network, and speed up the overall simulation. Wormhole routing would tend to increase network congestion, and slow down the simulation.

Bibliography

[Aga91] Anant Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.

[Bre92] Eric A. Brewer. Aspects of a Parallel-Architecture Simulator. Technical Report MIT-LCS-TR-527, S.M. Thesis, Massachusetts Institute of Technology, February, 1992.

[BDC+91] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, William E. Weihl. Proteus: a high-performance parallel architecture simulator. MIT-LCS-TR-516, Massachusetts Institute of Technology, September, 1991.

[BW95] Douglas Burger, David Wood. Accuracy vs. Performance in Parallel Simulation of Interconnection Networks. In *Proceedings of the Ninth International Parallel Processing Symposium*, April 1995.

[DS87] William J. Dally, Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, pp. 547-553, May 1987.

[Dal90] William J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, pp. 775-785, June 1990.

[Del91] Chrysanthos N. Dellarocas. A High-Performance Retargetable Simulator for Parallel Architectures. Technical Report MIT-LCS-TR-505, S.M. Thesis, Massachusetts Institute of Technology, June, 1991.

[DHN94] Phillip M. Dickens, Philip Heidelberger, David M. Nicol. Parallelized direct execution simulation of message-passing parallel programs. ICASE Report No. 94-50, June 1994.

[Eic93] Thorsten von Eicken. Private communication, April 1993.

[FRU92] Sergio Felperin, Prabhakar Raghavan, Eli Upfal. An Experimental Study of Wormhole Routing in Parallel Computers. IBM Technical Report RJ 9073, November 1992.

[Fuj90] Richard Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, Vol. 33, No. 10, pp. 30-53, October 1990.

[Gol93] Stephen R. Goldschmidt. Simulation of multiprocessors: accuracy and performance. Ph.D. thesis, Stanford University, June 1993.

[Her94] Steve Herrod. Private communication, August 1994.

[Joh94] Kirk Johnson. Private communication, November 1994.

[KK79] Parviz Kermani, Leonard Kleinrock. Virtual Cut-Through: A New Computer Communication Switching Technique. *Computer Networks*, vol. 3, pp. 267-286, October 1979.

[Leg95] Ulana Legedza, Synchronization Techniques for Parallel Simulation, MIT master's thesis, May 1995.

[ND91] Peter R. Nuth, William J. Dally. The J-Machine Network. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1992.

[PN85] G.F. Pfister, V.A. Norton. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. IEEE, 1985.

[RHL+93] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, David A. Wood. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, May 1993.

[RS94] Jennifer Rexford, Kang G. Shin. Support for Multiple Classes of Traffic in Multicomputer Routers. In *Proceedings of the Parallel Computer Routing and Communications Workshop*, May 1994, Springer-Verlag Lecture Notes in Computer Science, pp. 116-129.

[RS76] Roy Rosner, Ben Springer. Circuit and Packet Switching. *Computer Networks*, vol. 1, pp. 7-26, June 1976.

[Sei85] Charles Seitz et al. *Wormhole Chip Project Report*, Winter 1985.

[SSA+94] Craig Stunkel, Dennis Shea, Bulent Abali et al. The SP2 Communication Subsystem. Unpublished manuscript, 1994.

[ST94] Steve Scott, Greg Thorson. Optimized Routing in the Cray T3D. *Proceedings of the Parallel Computer Routing and Communications Workshop*, May 1994, Springer-Verlag Lecture Notes in Computer Science, pp. 281-294.

[Tan81] Andrew Tanenbaum. *Computer Networks.* Englewood Cliffs, NJ: Prentice-Hall, 1981.