PERFORMANCE MEASUREMENTS OF DISTRIBUTED SIMULATION STRATEGIES

Richard M. Fujimoto[1]
Computer Science Department
University of Utah
Salt Lake City, UT 84112

Technical Report No. UUCS-87-026a
November, 1987

Key Words: Discrete simulation, event-oriented, queueing models, performance analysis, distributed simulation.

[1]This work was supported by ONR contract number N00014-87-K-0184 and NSF grant number DCR-8504826.

PERFORMANCE MEASUREMENTS OF DISTRIBUTED SIMULATION STRATEGIES

ABSTRACT

A multiprocessor-based, distributed simulation testbed is described that facilitates controlled experimentation with distributed simulation algorithms. The performance of simulation strategies using deadlock avoidance and deadlock detection and recovery techniques are examined using various synthetic and actual workloads. The distributed simulators are compared with a uniprocessor-based event list implementation. Results of a series of experiments demonstrate that message population and the degree to which processes can look ahead in simulated time play critical roles in the performance of distributed simulators using these algorithms. An "avalanche" phenomenon was observed in the deadlock detection and recovery simulator, and was found to be a necessary condition for achieving good performance.

The central server queueing model was also examined. The poor behavior of this test case that has been observed by others is reproduced in the testbed, and explained in terms of message population and lookahead. Based on these observations, a modification to the server process program is suggested that improves performance by as much as an order of magnitude when first-come-first-serve (FCFS) servers are used.

These results demonstrate that conservative distributed simulation algorithms using deadlock avoidance or detection and recovery techniques can provide significant speedups over sequential event list implementations for some workloads, even in the presence of only a moderate amount of parallelism and many feedback loops. However, a moderate to high degree of parallelism is not sufficient to guarantee good performance.

1. INTRODUCTION

Discrete event simulation has long been a task with computation requirements that challenge the fastest available computers. For example, simulations of communication networks, parallel computer architectures, and logic networks often require hours, days, or even weeks of CPU time using traditional, single processor techniques. Simulator performance may be improved using vectorizing techniques (Chandak and Browne, 1983), processors dedicated to specific simulation functions (Comfort 1984), execution of independent trials on separate processors (Biles et al., 1985), or the execution of a single instance of a simulation program on a parallel computer. The last technique, referred to as distributed simulation, is the subject of this paper.

Simulation would initially appear to be a natural candidate for parallel processing because many of the aforementioned applications contain a high degree of parallelism. However, the exploitation of this parallelism is elusive because the global notion of simulated time does not easily map to a distributed computer. This property distinguishes distributed simulation from other forms of parallel computation.

Several schemes have been proposed to solve this problem. A survey of the literature has been reported by Kaudel (1987). One important class of distributed simulation algorithms are the so-called "conservative" mechanisms that were first developed by Chandy and Misra. One mechanism is based on a deadlock avoidance technique in which null messages are used to distribute clock information among the processes taking part in the simulation (Chandy and Misra, 1979). Another mechanism is based on a deadlock detection and recovery paradigm — the simulator runs until deadlock, the deadlock is detected, and an algorithm is executed to break the deadlock (Chandy and Misra, 1981). Enhancements to these algorithms are described in (Misra, 1986). Other approaches to distributed simulation have been proposed, notably the Time Warp approach proposed by Jefferson (1985), but the work proposed here will be confined to deadlock avoidance and deadlock detection and recovery techniques.

To date, little empirical data measuring the performance of specific implementations of these algorithms is available, although numerous simulation studies have been reported (for example, see Seethalakshmi 1979 or Reed 1985). The relationship between the system being simulated and performance of a distributed simulator is not well understood. A notable exception is the recent work of Reed, Malony, and McCredie (1987). This work is restricted to queueing network simulations, however, and does not attempt a comprehensive study of distributed simulator performance across a wide variety of workloads. The goal of the present study is to collect empirical data in order to identify aspects of the workload that have a critical impact on performance. In particular, one important goal of this work is to evaluate the effectiveness of distributed simulation strategies in achieving speedups over sequential event list implementations when the workload contains a moderate to high degree of parallelism.

A set of experiments are described that were designed to evaluate the effectiveness of conservative distributed simulation strategies. Empirical data is collected for the deadlock avoidance and the deadlock detection and recovery schemes developed by Chandy and Misra. A distributed simulation testbed has been developed using a shared memory multiprocessor, the BBN Butterfly.™ A wide range of synthetic and actual distributed simulation workloads can be easily created in the testbed facilitating flexible experimentation.

2. THE DISTRIBUTED SIMULATION TESTBED

The testbed consists of four important components:

- the application program that simulates some physical system; initially, a synthetic workload generator was used to facilitate the creation of parameterized workloads. Later, queueing network simulations were added to examine performance for a specific application.
- the distributed simulator responsible for correctly executing the application program;

- software to implement important "system" functions such as scheduling and interprocess communication; and
- the multiprocessor hardware on which the programs execute.

Each of these will be described next.

2.1. The Application Program

The distributed simulator application consists of a collection of n *logical processes* LP_0, LP_1, $\cdots$, LP_{n-1}. Each simulates a portion of the *physical system*, i.e., the system being modeled. All interactions among LPs are through timestamped messages. A *link* from LP_i to LP_j indicates that LP_i may send messages to LP_j. The sequence of timestamps on messages sent over any given link in the simulator is always a non-decreasing sequence of values.

Each logical process maintains a set of input and output message queues, one associated with each incoming and outgoing link respectively. Output queues buffer messages until (1) the LP can guarantee no new message will be sent on the link with a smaller timestamp, and (2) there is room in the corresponding input queue in the neighboring process to receive the message. Input queues hold messages that are ready to be processed by the LP if it can be guaranteed that no smaller timestamped message will be received on another input port sometime in the future.

Messages in each input and output queue are sorted in non-decreasing timestamp order. While a first-in-first-out (FIFO) discipline is sufficient to ensure this ordering for input queues, explicit insertion procedures are used to preserve the sorted order in output queues.

The *LinkClock* value of an incoming link is defined as the timestamp of the first message in the input queue associated with the link. If the queue is empty, then it is the timestamp of the last message placed into (or removed from) the queue. This measures the extent to which interactions between the corresponding processes in the physical system have been simulated.

Synthetic workloads are created using busy wait loops and random number generators to emulate program behavior. Queueing network simulations were also performed to relate the synthetic workload results to a specific application, as will be described later. Use of synthetic workloads facilitates controlled experimentation and evaluation of the influence of specific workload parameters on performance. The testbed contains a complete implementation of the distributed simulation algorithms, and allows application specific simulations to be inserted in place of the synthetic workload without significant modification of other portions of the testbed.

For illustrative purposes, *LP* behavior in the synthetic workload is described in the context of a simulator for an air traffic system in which each logical process models an airport, and interactions between *LP* s correspond to airplanes traveling between airports. Logical process LP_i executes the following steps:

(1) Determine the incoming link to LP_i that has the smallest *LinkClock* value.

(2) If the corresponding input queue is empty, then LP_i blocks. When a message is placed into that queue, resume execution and perform the steps that follow.

(3) Remove the next message from the input queue and execute a busy wait loop to emulate simulation activity. In the air traffic simulation, this corresponds to the time the simulator spends simulating the activities of an airplane from its arrival at the airport until its departure for a new destination.

(4) The timestamp of the message is incremented. This increment corresponds to the amount of time the airplane spends in the airport (e.g., waiting for gates, new flight crews, etc.) as well as its transit time to the airport that it will visit next.

(5) Select an output link and forward the message on this link. The simulator will buffer the message in the corresponding output queue until it can be sure no message with smaller timestamp will be sent on this link, and space is available in the remote input queue. If this input queue is full, and the output link has a *LinkClock* value smaller than that of any

input link, the process is blocked. In the airport simulator, the output port selection may be specified by predetermined flight schedules.

(6) Return to step (1).

2.2. Lookahead

One aspect of the workload model described above merits special attention. In any simulation application, a logical process LP_i cannot transmit a message to another process LP_j until:

- LP_i can determine the new timestamp and direction in which the message is to be sent; for example, in the airport simulation, the airport process may have to first make sure an "airplane crash" event does not occur that would close the airport before it can determine the departure time of a flight.
- LP_i has determined that no other message will be sent in the future to LP_j that carries a smaller timestamp. This requirement is necessary to ensure correctness of the distributed simulation algorithms.

In either case, some amount of simulated time must usually elapse before LP_i is able to safely transmit the message. This amount of (simulated) time depends on the ability of the process to "look ahead" into the future. The greater the lookahead ability of a process, the sooner it will be able to safely transmit the message. In general, lookahead is a complex function that is time variant and highly dependent on the details of the simulation that is being performed.

Lookahead is a fundamental aspect of *all* discrete event simulation programs. Event driven simulations progress by scheduling new events in the simulated time future based on knowledge of events that have occurred in the past. Without this lookahead ability, no future events could be scheduled, and the simulation program will quickly grind to a halt.

In the synthetic workload model implemented by the testbed, the lookahead function is the minimum timestamp increment a message will encounter in traveling through the process. If all

messages with timestamp T or less have been received, and t_{min} is the minimum timestamp increment (i.e., the lookahead), then the process can determine all messages it will send with timestamp $T + t_{min}$ or less. In the testbed, processes treat messages independently of one another. It will be seen later that lookahead plays a critical role in determining the effectiveness of conservative distributed simulation strategies in achieving good speedup.

2.3. Synthetic Workload Parameters

In the testbed, several parameters characterize a specific simulation workload:

- **Time Stamp Increment.** The amount of simulated time by which the timestamp of a message is increased as it travels through an *LP* may be selected from one of the following distributions (see table 1): deterministic, biased, shifted exponential, shifted uniform, or bimodal. The *shifted* exponential and uniform distributions are used to ensure the lookahead value is strictly greater than zero, a necessary condition for the deadlock avoidance simulation method. These distributions are commonly used in performance evaluations of sequential event list implementations of simulation programs and represent distributions often observed in practice (e.g., see Jones 1986).

- **Message Population.** The number of messages that exist in the simulator remains constant until the end of the simulation approaches and messages are deleted. The experimenter may select an arbitrary number of messages to reside in the simulator (subject to memory constraints) to control the amount of available parallelism.

- **Topology of Logical Processes.** Arbitrary network topologies may be selected by specifying a connectivity matrix to the workload generator. Logical processes configured as 16 (4 by 4) and 64 (8 by 8) node toroid networks were examined in the experiments discussed here (see figure 1). The toroid topology was selected because (1) it is a topology of practical interest for simulation of parallel computer architectures and communication networks; (2) it does not contain

any inherent bottlenecks that could color the results; (3) it is rich in cycles, providing a good test case for the distributed simulation algorithms; and (4) it contains a moderate node degree (of four), again providing a reasonably challenging test case for the algorithms.

- **Routing Probability.** A pseudo random number generator is used to select the output link on which each message is to be forwarded. The workload model programmed into the testbed allows arbitrary routing probabilities to be selected. Except where indicated otherwise, the experiments described here assume messages are uniformly distributed among the available output links. This avoids bottlenecks that could color performance measurements. This assumption is relaxed in some of the experiments described later.

- **Computation Granularity and Distribution.** The amount of time spent in the busy wait loop to emulate simulation activity is selected stochastically (at present, either an exponential of normal distribution may be selected) or deterministically. If the selected time is less than that required to execute random number generators to select the timestamp increment, output link, and computation time, the busy wait is skipped. This imposes a lower bound on the computation granularity in the testbed configuration of a few hundred microseconds depending on the distributions that are selected.

The initial experiments performed in this study assume homogeneous simulation systems, i.e., each logical process is parameterized in the same way. This reflects the situation found in many simulators, e.g., simulations of communication networks, and simplifies the analysis of measurement data. Some experiments were also performed in which non-uniformity was introduced into the workload. Also, the queueing network simulations described later represent another class of non-uniform workloads.

2.4. The Distributed Simulator

One level lower than the application workload described above is the simulator software that must correctly execute the simulation program. Important parameters of the testbed simulators are:

- **Simulation Strategy.** At present, deadlock avoidance and deadlock detection and recovery algorithms similar to those developed by Chandy and Misra have been implemented, as well as sequential event list implementations.

- **Process Mapping.** An arbitrary process to processor mapping function can be specified. The mapping of processes to processors is greatly simplified by the hardware configuration, a shared memory multiprocessor, that effectively provides full connectivity among the processors. The mapping problem then becomes one of clustering communicating processes onto the same processor and balancing workloads to avoid bottlenecks. A simple partitioning strategy that clusters communicating processes was used that satisfies both of these goals. The toroid was simply divided into a coarser grid, and processes at the same grid position are mapped to the same processor. Experiments were designed so that the same number of processes could be mapped to each processor in each trial. The homogeneous nature of the workload does not justify any other mapping. Therefore, these experiments make the optimistic assumption that a good mapping can be found for the distributed simulation program.

- **Process Scheduling.** A static scheduling policy may be used in which processors are restricted to only execute processes that have been mapped to it. Alternatively, a dynamic load balancing strategy may be selected in which idle processors examine process queues in other processors in order to find work. This will be described in greater detail later. Unless indicated otherwise, we will assume static scheduling is used.

Other aspects of the testbed software include the scheduler and software to implement message passing primitives. We will defer descriptions of these aspects of the testbed until after the

testbed hardware has been described.

2.5. Testbed Hardware

A BBN Butterfly™ multiprocessor was used for the distributed simulation testbed. The multiprocessor consists of a collection of processor nodes and a high performance interconnection switch. As shown in figure 2, each processor node contains a 16 MHz MC68020 with MC68881 floating point coprocessor, up to 4 MBytes of memory, and a *processor node controller* (PNC), a microcoded engine implemented with 2900 series AMD parts. The interconnection switch is configured as an Omega network. Although the system may be expanded to contain up to 256 processors, the testbed on which the experiments were performed contained only 17.

All memory references made by the 68020 are passed to the PNC. Local memory references are forwarded to the local memory, while remote references are passed to the appropriate processor node through the switch. The PNC also handles memory requests made by other processors to this node. Atomic test-and-set like memory operations are also implemented in the PNC.

Execution times of various instructions and operations are shown in table 2. As can be seen, a local memory reference requires 600 nanoseconds and a remote memory reference 4 microseconds assuming no switch contention. Experimental data indicates that switch contention, and hot spot congestion in particular, is unlikely (Thomas 1986a). Atomic memory operations such as the atomic-or on which the locking primitives are based require approximately 20 microseconds, and a parameterless function call requires 6.9 microseconds. Although portions of Chrysallis, the Butterfly's operating system, are implemented in the PNC, only the atomic operations are used in the distributed simulation testbed.

Each processor executes a *single* Chrysallis process that contains:

- code for the logical processes,
- a scheduler that controls the execution of logical processes,
- code for message passing primitives, and
- code for the distributed simulator.

Implementation of logical processes *within* a single Chrysallis process greatly reduces the cost of context switches. The scheduler can initiate and resume execution of a logical process with a simple procedure call. A copy of all of the testbed's code exists on each processor, allowing any processor to execute any logical process; however, the state variables (mailboxes, etc.) do not migrate from the processor to which the process was originally mapped, so remote execution incurs some performance degradation because more remote memory references are required.

The simulation testbed was implemented using the Uniform System programming environment provided by BBN (Thomas 1986b). The Uniform System is used primarily for initialization purposes, e.g., to initially map the Butterfly's memory into a common, global address space. A single Uniform System task is created on each processor that executes the scheduler, and does not terminate until the simulation is complete. After initialization, only the locking and low-level atomic operations provided by the Butterfly are used.

2.6. The Scheduler

A scheduler executes on each processor that selects and then resumes execution of logical processes. Each scheduler maintains a queue of runable processes. Only processes mapped to the local processor may reside in the process queue of a given scheduler. When a logical process causes a blocked process to become unblocked, it simply adds that process to the end of the scheduling queue on which that process resides. Locks ensure that queue accesses are properly synchronized.

The scheduler is simply a loop that repeatedly removes the next process from its process queue and executes it. If the queue is empty and static scheduling is used, the scheduler keeps interrogating the queue until a process mapped to that processor is added to the queue.

If the dynamic scheduling policy is used, each scheduler is allowed to execute any runable logical process in the system. Idle schedulers poll the process queues in other processors searching for a runable process. The local process queue is given highest priority because execution of processes mapped to the local processor is more efficient — fewer remote memory references are required. Processors are numbered $0, 1, 2, \cdots, N-1$. If the scheduler on processor i finds its local queue is empty, it checks the queue on processor $i+1$, then $i+2$, $i+3$ etc. (all arithmetic is modulo N), until a nonempty queue is found. A single, global scheduling queue was avoided to eliminate the possibility of it becoming a bottleneck. Moreover, this scheduling policy is well suited to the symmetric, homogeneous nature of the distributed simulation testbed and the workload used in the experiments.

2.7. Message Passing Primitives

Message passing primitives were implemented using the lock and unlock primitives. The *send* operation first places the message into the appropriate output queue of the sending process, making sure that the timestamp order of the queue is preserved. If the sender can guarantee no smaller timestamped messages will follow, and there is space in the corresponding input queue, the message is copied (by the sender) into the input queue of the receiver using remote memory references if the receiver is on a different processor. Otherwise, the message remains buffered in the output queue. Locking is required on *input* queue operations, but none is needed on output queue operations. If the send operation causes an input queue to become full, *and* the *LinkClock* value of this link is smaller than that on any incoming link, the sending process is blocked. The latter condition regarding the *LinkClock* value is necessary to ensure correct operation of the deadlock avoidance simulator. If it is not met, the process is allowed to continue processing

incoming messages until the *LinkClock* value of the incoming links exceed that on the blocked output link. Input queues may contain up to 16 messages. Simulation experiments by Seethalakshmi (1979) indicate that additional buffers provide little performance improvement.

The *receive* primitive inspects the process's input queues to determine which has the smallest *LinkClock* value. If the selected queue is empty, the process is blocked and a flag on the queue is set indicating the process is blocked on this queue. Otherwise, the message at the front of this queue is returned.

Sending or receiving a message may unblock a neighboring process. The sending process checks the blocked flag on the receiving input queue. If the blocked flag is set, the sender schedules the receiver and resets the flag. A receiver can detect when it must unblock a sender by checking a second flag on the input queue that indicates if the sender is blocked. Because it is the responsibility of the sending or receiving process to unblock its neighbors *before* it itself is blocked, at least one process will always be running or scheduled in some processor's scheduling queue except when the system is deadlocked or terminated. This simplifies deadlock detection.

3. THE SIMULATION ALGORITHMS

Two distributed simulation algorithms were implemented in the testbed: one based on deadlock avoidance and another based on deadlock detection and recovery. The shared memory architecture of the Butterfly was used to facilitate deadlock detection. A single processor, event list implementation was also developed in order to compute speedup.

3.1. Deadlock Avoidance Strategy

The deadlock avoidance scheme developed by Chandy and Misra was implemented first. Each logical process sends a null message to each of its neighbors whenever it blocks. The timestamp on this message is equal to the local clock value of the process plus the lookahead

value. Lookahead values for the different timestamp increment functions are shown in table 1. This timestamp indicates a lower bound on the timestamp of the next forthcoming message. Chandy and Misra have shown that this approach is sufficient to avoid deadlock (Chandy and Misra, 1979).

One optimization was performed to streamline the processing of null messages. Rather than enqueueing each null message sent to another processor, a single variable is associated with each input link that contains the timestamp of the last null message that was received. This avoids unnecessary enqueue and dequeue operations and leads to more efficient memory utilization.

3.2. Deadlock Detection and Recovery Strategy

The second simulation approach is based on deadlock detection and recovery. The simulation runs until deadlock, the deadlock is detected, and an algorithm is initiated to break the deadlock (Chandy and Misra, 1981). A central controller is used to coordinate the deadlock recovery procedure.

Deadlock in the testbed is easily detected by maintaining a global counter indicating the number of processes that are either scheduled or running. The counter is incremented whenever a process unblocks another, and decremented whenever a process becomes blocked. Once a process decrements the counter, it will not increment it again until it has been rescheduled and resumes execution. The system is deadlocked whenever the counter reaches zero and there is at least one process that has not yet terminated (otherwise, the computation has terminated). Each scheduler checks the deadlock counter whenever it fails to find a process to run. If the counter is zero, the scheduler initiates a computation to break the deadlock.

The deadlock recovery algorithm locates the message in the system with the smallest timestamp and arranges for it to be processed next. A distributed algorithm is used to perform this computation. Each scheduler finds the message with minimum timestamp buffered in the

processes mapped to that scheduler's processor. It then reports this local minimum to the central controller which computes a global minimum. By convention, the scheduler executing on PE 0 acts as the central controller.

An alternative deadlock recovery algorithm was also implemented in which messages are propagated throughout the system in order to restart as many processes as possible. This algorithm is described by Chandy and Misra (1981). It was found, however, that the additional time required to execute this algorithm yielded a net loss in performance. The performance figures reported here are based on the former deadlock recovery approach.

3.3. Uniprocessor Simulation Algorithm

Finally, a single processor, event list simulator was developed to allow comparison of distributed simulation programs with sequential event list implementations. In order to obtain a fair comparison, the uniprocessor simulator was constructed by modifying the distributed simulator. Both implementations maintain the same overall structure, organization, programming style, and conventions.

The principal modifications to the multiprocessor code to generate the event list implementation were:

- Code was added to insert and remove elements from a global timestamp sorted event list.
- Code for synchronization overhead, e.g., locks, was eliminated.
- Code for message queues and message passing primitives was eliminated. The procedure to send messages in the multiprocessor program was replaced by a procedure to insert an event into the event list. The procedure to receive messages was eliminated entirely.
- The scheduler program was replaced with a loop that repeatedly removes the first element in the event list and executes the code for the appropriate logical process to process that event.

The event list was implemented as a splay tree (Sleator and Tarjan, 1985). Empirical evidence suggests that splay trees are among the fastest methods for implementing an event list (Jones 1986). An alternative implementation using a singly linked linear list was also developed. It was found that this implementation yielded performance comparable to the splay tree for small simulations, but as expected, ran much more slowly for many of the larger simulations. The splay tree implementation is used in all comparisons with uniprocessor simulations reported here.

3.4. Performance Metrics

Three metrics are defined to evaluate the performance of the distributed simulation programs:

- **Speedup.** $SU(n)$, the speedup using n processors, is defined as the execution time of the single processor, event list implementation using a splay tree divided by the execution time of the distributed simulation program when n processors are used.
- **Null Message Ratio.** NMR is defined as the number of null messages processed by the simulator using deadlock avoidance divided by the number of real (non-null) messages processed. This measures the overhead of the deadlock avoidance approach.
- **Deadlock Ratio.** DR is the number of messages processed by the distributed simulator using deadlock detection and recovery divided by the number of deadlocks that occur. This figure measures the efficiency of the deadlock detection and recovery algorithm.

The single processor execution times were obtained by running the splay tree simulator on a single node of the Butterfly. The same compiler as that used by the distributed simulator was used. Therefore, compiler and processor speed dependencies are factored out of the speedup figures.

4. EXPERIMENTS USING SYNTHETIC WORKLOADS

The discussion that follows describes the experimental methodology that was used and provides an overview of the results that were obtained using synthetic workloads. Some statistical aspects of the measurements are then described, followed by the measurements themselves.

4.1. Experimental Methodology

The large number of testbed parameters complicates the performance evaluation. Complete, factorial testing of all combinations of parameters was not feasible. Therefore, it is appropriate to describe the methodology that was used. One important goal of these experiments was to determine the effectiveness of distributed simulation strategies in achieving good speedups when moderate or high degrees of parallelism are available in the application.

An initial set of experiments was performed using a set of parameters that are ''reasonable'' for a typical simulation application, e.g., a communication network simulator. These parameters are summarized in table 3. The workload is a reasonable model for communication network simulations that have been performed by the author to evaluate the performance of multicomputer networks (Fujimoto 1983).

After initial experimentation, it quickly became apparent that timestamp distribution had a rather profound impact on performance. The distributed simulators provided significant speedups over the sequential, event list implementation for many, though not all, workloads. In order to test the robustness of this dependence, parameters of the initial workload were varied. Several parameters were found to have only a secondary effect on performance. In particular, the computation grain, distribution, and variances were varied, but yielded only modest changes in performance. A dynamic scheduling strategy was measured to evaluate the effectiveness of the static scheduling policy. It was found that the static scheduling policy performed reasonably well, and therefore could not be blamed for poor performance. Attention could now be focused on parame-

ters believed to be of primary importance.

Closer examination of the internal operation of the simulators revealed that the aspect of the timestamp distribution that appeared important is the lookahead function. To verify and document this observation, a quantity called the *lookahead ratio* was defined, and varied across different timestamp increment distributions. It was found that, indeed, different timestamp distributions with the same lookahead ratio yielded similar performances.

A second parameter of great importance is the message population. Experiments varying message population revealed an ''avalanche'' phenomenon in the deadlock detection and recovery simulator where performance remains poor at relatively low and moderate message populations, but then increases dramatically once message population reaches a certain level. The ''knee'' in this performance curve is referred to as the ''avalanche point.'' Experiments were performed in which both of these critical parameters, lookahead and message population, were varied. It was found that the avalanche point was highly dependent on the lookahead ratio.

To test the robustness of these results, other aspects of the workload were varied. In particular, asymmetry was introduced into the workload, and found *not* to invalidate the previously observed results. Some final experiments were performed to further test the results that had previously been derived.

4.2. Confidence Intervals

The experimental data that follows was obtained by averaging several trial runs on the distributed simulation testbed. Several different seeds for the random number generator were used to test the stability of the results. For the most part, measurement data was well behaved. The 95% confidence intervals for the data points was less than ± 5% of the reported values for most of the experimental data, and was typically less than 2%. One notable exception is measurements of deadlock ratio near, and especially beyond the avalanche point. Here, variances were substan-

tially larger, and confidence intervals expanded to much higher levels (e.g., 30% or 40%). This does not change the qualitative conclusions that were drawn from these measurements, however.

4.3. Varying the Timestamp Increment Distribution

Speedup curves for the 16 (4 by 4) and 64 (8 by 8) node toroid network simulator using the deadlock avoidance strategy are shown in figure 3. The parameters used in these runs are shown in table 3; notably, the message population is set at four per logical process, or one per link in the toroid. The *total* message population remains constant throughout each execution of the distributed simulator, however, the number of messages residing in a particular link or logical process will vary during the run.

As can be seen, the deadlock avoidance strategy yields good speedup figures for the deterministic and biased distribution, but disappointing performance for the others. When 64 processes are used, as much as 75% of ideal speedup is obtained for the biased and deterministic distributions using 64 processes, but only 20 to 45% of ideal for others. Speedup figures for the 16 process toroid are somewhat less. These curves demonstrate conclusively that distributed simulation methods can provide significant speedups over sequential, event list implementations for some interesting workloads exhibiting a moderate degree of parallelism, even in the presence of feedback loops.

The corresponding speedup curves for the deadlock detection and recovery strategy are shown in figure 4. Speedup is rather disappointing, however, except for the deterministic timestamp increment distribution. All of the speedup curves yield performances well below those for the simulator using deadlock avoidance. As we shall soon see, this can be attributed to a relatively low message population.

The deterministic distribution arises in physical systems that are synchronous to a global clock. It is a somewhat specialized distribution from the standpoint of distributed simulation

strategies because the simulator behaves very similarly to a time driven simulator, i.e., one in which processes advance in lock step from one time step to the next. If the message population is low, the processes execute all of the messages in the current time step and then deadlock. The recovery algorithm will then break the deadlock and, in effect, advance the simulation to the next time step. The number of deadlocks that occur is simply the number of timesteps in which the simulation is run. The deadlock ratio is the same as the message population. As will be demonstrated later, significantly higher performance can be obtained if the message population is increased; the frequency of deadlock can be reduced significantly if each incoming link contains at least one message most of the time.

Performance of the 16 node toroid is somewhat less than the 64 node toroid because the simulation does not contain sufficient parallelism to keep all of the processors busy. In addition, as the number of processes per processor is decreased, each process is afforded less time to collect messages before it is executed by the scheduler. As a result, a process may be scheduled more often than if there were more processes mapped to the processor. The additional scheduling overhead and increased idle time lead to poorer performance in the 16 node simulator, particularly as the number of processors is increased.

The null message ratio overhead for the deadlock avoidance strategy and number of real messages per deadlock in the recovery scheme are plotted in figures 5 and 6 respectively. These demonstrate that the poor (or good) speedup figures are caused by high (or low) overhead in the simulation strategy. Ten to twenty null messages per real message were often observed when speedup was poor, in contrast to null message ratios that were typically less than one when good speedup was observed. Null message ratios tend to increase as the number of processors increases because, as described earlier, the time between resumptions of tasks is shorter so blocking occurs more frequently. The generally lower null message ratios in the 64 node toroid, when compared to the 16 node toroid, is also a consequence of this phenomenon.

The real messages per deadlock ratios in the detection and recovery simulator indicate that except for the deterministic timestamp increment, only one or two messages are processed between deadlocks. This accounts for the poor speedup figures.

4.4. Varying the CPU Grain and Distribution

The original performance measurements assumed the CPU execution time to process each message was selected from an exponential distribution with mean of 1 millisecond (actually, a truncated distribution because, as noted earlier, the time required to invoke random number generators places a lower bound on the CPU time that is used). Experiments were run in which a larger grain of computation was used. In particular, these experiments used the following distributions to select a computation time for each message:

- an *exponential distribution* with mean of 50 milliseconds.
- a *normal distribution* with 50 millisecond mean, and standard deviation of 1 millisecond.
- a *normal distribution* with 50 millisecond mean, and standard deviation of 40 milliseconds.

The results of these experiments are shown in figures 7 and 8 for the deadlock avoidance and recovery strategies respectively. As can be seen, increasing the CPU grain does improve performance somewhat, however, this is an effect of secondary importance. This is, to some extent, a consequence of using a shared memory multiprocessor as the testbed. An implementation with a slower communication switch can be expected to show a significant degradation in speedup if the grain of computation is too small because the communication overhead will dominate.

4.5. Dynamic Scheduling

The experiments described thus far assumed a static scheduling policy. Simulation runs using the dynamic scheduling policy described earlier were also performed to ensure that the scheduling policy was not leading to poor performance. Speedup curves for these runs are shown

in figures 9 and 10 for the avoidance and recovery strategies respectively. The dynamic scheduling policy performs more *poorly* than the static scheduler in most instances. This is because the symmetric nature of the workload and hardware organization allows a good static mapping of tasks to processors to be obtained. When this is the case, it is better for schedulers to wait until some task that is mapped to it is ready to execute rather than locate and execute a task from another processor's queue; the latter incurs a performance degradation because remote memory references are required to access the process's state. These speedup curves verify that the static scheduler obtains reasonably good performance, and thus does not contribute appreciably to poor speedup figures.

4.6. Lookahead and Lookahead Ratio

The data presented thus far has demonstrated that the distribution of the timestamp increment has an important effect on performance. The reason for this dependence is related to the notion of lookahead that is relevant to *all* distributed simulators no matter what strategy is used. The lookahead notion was introduced by Chandy and Misra where it is essential to the operation of the deadlock avoidance algorithm. Though not *explicitly* required in the deadlock detection and recovery algorithm, it is nevertheless essential as discussed earlier. We shall soon see that lookahead plays an important role in the performance of both of the distributed simulation strategies that were examined.

As described earlier, lookahead characterizes the ability of a process to predict future messages that it will send based on knowledge of messages it has already received. In particular, if a process has received all messages with timestamp t or less, and can predict all future messages with timestamp $t + \bar{t}$ or less, then we say the lookahead of the process is $\bar{t}$. Here, we assume the lookahead ability of a process is fixed throughout the simulation, and is the same on each output link.

Lookahead is important in the distributed simulation algorithms discussed here because a process with poor lookahead ability will be forced to delay forwarding messages, effectively decreasing the message population and the available parallelism. In addition, the conservative distributed simulation strategies discussed here require that the sequence of timestamps of messages sent over a link must be non-decreasing, further delaying message transmission in certain situations. These two factors tend to reduce the efficiency of the distributed simulator.

In the deadlock avoidance strategy, lookahead plays a critical role because if the message population is not sufficiently large, a cycle will develop carrying a null message that, in effect, increases in timestamp at each hop by an amount equal to the lookahead value. This cycle will persist until some process's clock becomes sufficiently large to allow the next real message to be processed. The smaller the lookahead, the longer the cycle will persist, leading to a corresponding degradation in performance.

To test this hypothesis and quantitatively measure this phenomenon, we define the *lookahead ratio (LAR)* as:

$$LAR = \frac{mean\ \ timestamp\ \ increase}{lookahead}.$$

A *low* (e.g., 1.0) *LAR* corresponds to a *high* degree of lookahead. In the testbed, the lookahead function is the minimum timestamp increase a message will encounter in traveling through a process, so *LAR* is the mean of the function divided by its minimum. The deterministic distribution is characterized by the lowest possible lookahead ratio (in the testbed), 1.0. Lookahead ratios for the other distributions used for these experiments are shown in table 1.

Figure 11 shows the null message ratio for the biased and exponential distributions, two distributions that earlier displayed significant differences in performance, as the lookahead ratio is increased. *LAR* is increased by increasing the mean of the timestamp increment function while holding the minimum constant. Eight processors were used in these, and the experiments that follow. These curves demonstrate that the two timestamp increment distributions that earlier

yielded strikingly different performances now perform almost identically once they are modified to have the same lookahead ratio. This confirms the hypothesis that the dependence of performance on the timestamp increment function which was earlier observed is due to differences in the lookahead ratio.

The null message ratio increases approximately linearly with the lookahead ratio. This is consistent with the above description of performance degradation in deadlock avoidance simulators where null messages circulate in loops.

Similar experiments varying the lookahead ratio for the deadlock detection and recovery strategy were not as enlightening. Deadlock ratio as a function of lookahead ratio is shown in figure 12. Deadlock ratios remained from one to three messages per deadlock, independent of the lookahead ratio, that is, except when the lookahead ratio was 1.0, and the timestamp increment is deterministic; in this case, the simulator runs much more efficiently, as described earlier. Higher message populations are required to gain efficient operation for non-deterministic timestamp increments, as will be described next.

4.7. Message Population and the Avalanche Effect

The message population, as well as the number of logical processes, determines the amount of parallel activity that can occur in the simulation. Simulations of distributed simulators have indicated that the message population is a critical factor in determining the efficiency of distributed simulation strategies.

The null message ratio is shown as a function of message population in figure 13 for the biased timestamp increment distribution with LAR values of 1.11 and 11.0. The corresponding curve for the exponential timestamp increment distribution ($LAR = 11.0$) is also shown for comparison. Highly efficient execution is obtained for both distributions as the message population is increased, though overhead remains continually lower for smaller lookahead ratios, as described

earlier. It is also seen that the exponential and biased distributions with identical *LAR* values yield nearly identical performance, consistent with the analyses described earlier.

Up to this point, the deadlock detection and recovery strategy has not yielded satisfactory performance except in the special case when timestamp increases were deterministic. Figure 14, where deadlock ratio (messages per deadlock) is plotted as a function of message population, provides an explanation. The deadlock ratio remains approximately constant at one or two messages per deadlock until a critical point at which the ratio increases dramatically, leading to an equally dramatic improvement in performance.

We refer to the phenomenon that leads to this behavior as *message avalanche*, borrowing the name from an analogous phenomenon in a reverse biased diode. As one increases (makes more negative) the voltage across a diode, little current flows until a certain voltage at which current increases dramatically. The reason for this dramatic change in behavior is that once a critical voltage is reached, charge carriers (electrons) obtain sufficient energy to collide, and jar free other electrons at lattice positions in the silicon crystal. These newly created charge carriers collide at other lattice positions and jar free a second set of charge carriers, and so on. This causes a multiplicative effect whereby one charge carrier causes the ''release'' of several others, leading to a significant flow of current. A similar behavior was observed in the distributed simulator whereby once the message population reached a certain level, a new, incoming message caused the ''release'' of several others, which in turn, triggered the ''release'' of still others, and so on. An avalanche of message traffic results from the initial message. This avalanche effect proved to be a necessary condition for the deadlock detection and recovery simulator to achieve good performance.

Viewed from another perspective, the deadlock detection and recovery scheme will perform well if message population is sufficiently high to ''sustain'' each logical process, i.e., the simulator will operate efficiently if there is at least one message on each of its input queues whenever it

interrogates them, looking for messages to process. Conversely, performance will be poor when this is not the case. The avalanche behavior suggests that the transition between these two modes of operation is a very sharp one.

Near the point where the avalanche effect begins to take hold, it was found that the time to execute the simulator actually decreased as the message population, and thus the amount of work to be performed, was *increased.* This suggests that deadlock detection and recovery simulators may, when near avalanche, benefit from the addition of ''dummy'' message traffic.

A second, important observation is that the message population necessary to induce avalanche depends on the lookahead ability of the processes. Figure 14 shows that avalanche begins with from 8 to 16 messages per *LP* when *LAR* was 1.11, but from 64 to 128 messages per *LP* were required when *LAR* was increased to 11.0. The reason for this behavior is similar to that described earlier — a larger portion of the message population will not be ''processable'' as the lookahead ratio is increased.

4.8. Asymmetric Workloads

The workloads presented above were highly symmetric and uniform, leading one to ask if these results are applicable to asymmetric workloads as well. Further, it is possible that the avalanche effect described earlier is a consequence of the symmetric nature of the workload — because all processes behave in the same fashion, one might argue that avalanche is really a magnification of the behavior of a *single* process. It will be demonstrated below that the results described so far, and avalanche in particular, remain valid even if asymmetry is introduced.

The previous experiments assumed that incoming message traffic was uniformly distributed among the outgoing links. This assumption was tested by having processes randomly select one link at the beginning of each simulation run as a ''favorite'' that received twice as many messages as the others. Experiments were performed in which one, one fourth, one half, and finally

all processes have a favorite link. The resulting curves are shown in figures 15 and 16 for the deadlock avoidance and recovery algorithms. The corresponding curve when no processes have a favorite port is also shown for comparison. As can be seen, this modification in the workload had only a minor impact on performance, and the avalanche effect still persisted in the detection and recovery simulators.

Secondly, because lookahead had an important influence on performance, some number of processes were selected to have different timestamp increment behavior, and correspondingly different lookahead functions, than others. Figures 17 and 18 show the results of these experiments when zero, one, one fourth, one half, and then all of the processes have a lookahead ratio of 11.0, while the others have an *LAR* value of 1.11. The processes with high lookahead were selected at random. Again, the avalanche effect persists in the deadlock recovery simulator. Further, it is seen that the presence of only one ''asymmetric'' process causes a noticeable increase in the number of messages required to reach avalanche, and that a significant fraction of asymmetric processes yields roughly the same avalanche point as the case when *all* processes have the higher *LAR* value. This is a negative result because it implies that the performance of the distributed simulator is largely controlled by the most poorly parameterized processes. This phenomenon is analogous to bottleneck phenomena where the slowest, ''bottleneck'' devices dominate. The deadlock avoidance strategy was found *not* to be as susceptible to this behavior.

4.9. A Final Experiment

As another test of the observations collected thus far, the initial set of experiments (figures 3 and 4) were repeated with a higher message population. The message population was increased from 1 to 32 per *LP*. According to figure 13, this would reduce the null message overhead in the deadlock avoidance simulator by more than an order in magnitude. According to figure 14, this population should be sufficient to induce message avalanche in the biased and deterministic timestamp increment distributions, but not the others. The speedups of the distributed simulators

over the sequential, splay tree simulator are shown in figures 19 and 20. The speedup curves are consistent with the analyses presented earlier.

5. SIMULATION OF QUEUEING NETWORKS

The results reported thus far were based on experiments using synthetic workloads. To test the validity of these observations in a real application, queueing network simulations were performed on the distributed simulation testbed. These experiments were *not* intended to be a comprehensive study of queueing network simulation, but rather were intended as an exercise in using the results developed above to explain and improve performance in a specific application.

5.1. The Central Server Queueing Network

A five process, central server queueing network was selected for these experiments. The central server model is an important, widely studied network in the performance evaluation field.

Simulation and empirical studies by Seethalakshmi (1979) and Reed (1987) respectively concluded that the central server network is ill suited for conservative distributed simulation algorithms. We reproduce and explain the poor results that these researchers observed in terms of message population and lookahead. Finally, we demonstrate that an order of magnitude improvement in performance is possible for a wide range of queueing networks when the distributed simulator is written to exploit all available lookahead. On the other hand, we discuss other queueing networks that fundamentally contain poor lookahead properties, and hypothesize that they are not well suited for distributed simulation algorithms using deadlock avoidance or detection and recovery techniques.

The central server network that was modeled is depicted in figure 21. This network contains three types of logical processes:

- *first-come-first-serve server (FCFS)* processes. Each contains a queue of jobs waiting to be processed and a server that serially processes them. Two different ways of programming the server process are described below. The process used in these experiments computes the average queue length of the server in addition to processing events.

- a *fork* process. This process receives incoming jobs on its input port, and routes each job to a randomly selected output port. In these experiments, either of the two output ports was equally likely to be selected. The program for the fork process immediately forwards incoming messages on a randomly selected output port. If deadlock avoidance is used, a null message is only generated when the process is forced to block.

- a *merge* process. This process receives streams of messages on its incoming links, and combines them into a single output stream. Because the messages in the output stream must have non-decreasing timestamps, the merge process cannot forward a message that has arrived on one port until it has received a message on the other with a timestamp at least as large as the first. Only then can it guarantee that no smaller timestamped messages will be received in the future. Like the fork process, the timestamps of messages passing through the merge process are not modified.

5.2. Lazy Servers

Initially, the server process was programmed using two events:

- An *arrival event* occurs whenever a message is received, representing a job arriving at the server. This causes the queue length in the server to be increased by one. If the server was previously idle (i.e., the queue length was zero), a service time is selected and a departure event (described next) is scheduled by having the process send a message to itself.

- A *departure event* indicates service for a job is completing, and results in a message being sent on the server's output port with timestamp equal to the departure time. The length of the job

queue is also reduced by one. If there are one or more jobs remaining in the queue, a service time is selected and a new departure event is scheduled.

We refer to the above algorithm for the server process as the *lazy server* strategy because the server does not attempt to forward a message until all events preceding the departure time have been processed. In particular, the lazy server will not send a message with timestamp *TS* until it has received a message with timestamp at least this large. An *eager* message forwarding server will be described later in which this condition is relaxed, and messages are sent as soon as their departure time can be determined.

5.3. Performance Using Lazy Servers

The five node central server queueing network using lazy servers was executed on the Butterfly distributed simulation testbed. In all of the experiments described below, each logical process was mapped to a separate processor, and static scheduling was used. Service times for server processes were selected from either the deterministic or shifted exponential distribution. Experiments were repeated for a wide range of message populations.

Null message ratios for the deadlock avoidance strategy are shown in figure 22, and speedup curves in figure 24. As can be seen, performance is poor. Null message ratios converge to two or three null messages per real message as the message population is increased, and execution times were two or more times *longer* than that of the uniprocessor, event list implementation.

Deadlock ratios and speedup for the deadlock detection and recovery strategy are shown in figures 23 and 25. Performance of this lazy server simulation is also poor. Even at high message populations, only two or three messages are processed between deadlocks, and execution times are two to three times *longer* than the event list execution time. No avalanche effect was observed, even at relatively high message populations.

These results can be explained by examining the lookahead ratio for the server processes. Because the server will not forward a (non-null) message with departure time *TS* until it has received a message denoting a job arrival with timestamp at least *TS*, the lookahead of each lazy server is essentially zero. The lookahead ratio of the lazy server is the job service time plus the job waiting time (the time it remains in the server queue) divided by the process's lookahead. It is unbounded in size, reflecting the poor lookahead ability of the lazy server process.

Message avalanche did not occur in the queueing network simulators because the lookahead ratio is very poor (recall poorer lookahead ratios result in high message populations to achieve avalanche), and in fact becomes ''worse'' as the message population is increased. The lookahead ratio becomes worse in the sense that the numerator (the message waiting time in particular) becomes larger as message population increases while the denominator remains the same.

Therefore, the poor performance of the lazy server program is a consequence of the poor lookahead properties of the servers. Because lookahead is poor, increasing the message population only increases the number of messages internally ''buffered'' by each server process. Such messages cannot spawn any new simulation activity until they are forwarded to other processes, so they do not lead to any significant improvement in performance.

These results are qualitatively similar to those reported by Reed and Seethalakshmi. The servers used in those studies are a variation of the lazy server described above, and share the same (poor) lookahead properties — a message will not be forwarded until another message is first received with a timestamp at least as large as the departure time of the first. Therefore, the results described here provide an explanation for the poor performance that they observed.

Somewhat lower null message ratios are reported here than that reported by Reed and Seethalakshmi. This is because the distributed simulator described here only generates null messages when the program blocks. In contrast, the fork process in Reed's and Seethalakshmi's programs generate a null message on *every* outgoing link each time a message is sent on one. This is

necessary whenever the message I/O interface is such that processes cannot determine when they will block on a message I/O before actually doing so.

5.4. Eager Servers

Based on the above analysis, better performance ought to be obtainable by reprogramming the simulator, and the FCFS server process in particular, to have better lookahead characteristics. This is possible because the poor lookahead properties in the lazy server were an artifact of the *implementation* of the server rather than a fundamental characteristic of the server itself.

A second server process was developed that forwards (real) messages as soon as the outgoing message can be constructed. The message can be constructed as soon as the departure timestamp can be determined. Because a FCFS queueing discipline is used, the departure time can be determined as soon as the message is received. The server process need only keep track of the time the server will become idle assuming no further jobs are received. We shall refer to this time as $FINISH$. Whenever a new message is received with timestamp TS:

(1) A service time $SERVICE$ is selected;

(2) If $FINISH > TS$, the server is busy when the job arrives, so the message is *immediately* forwarded with timestamp $FINISH + SERVICE$.

(3) Otherwise ($FINISH \leq TS$) the server is idle when the job arrives, so the message is forwarded with timestamp $TS + SERVICE$.

(4) In either of the above cases, $FINISH$ is updated to the departure time of the job that was just forwarded.

We call processes using this algorithm *eager* servers because they forward incoming messages as soon as they arrive. Eager server processes have a lookahead ratio of 1.0 because they can look ahead as far as the departure time of the next incoming message.

5.5. Performance Using Eager Servers

The five node central server model using eager server processes was executed on the testbed, yielding overhead and performance curves shown in figures 22 through 25. As predicted, dramatic improvements in performance result. Null message ratios are reduced by one to three orders of magnitude for moderate and high message populations. Message avalanche appeared in the deadlock detection and recovery simulator at relatively modest message populations, leading to deadlock ratios that improved by several orders of magnitude over the lazy server implementation. Execution times for moderate to high message populations were two to three times *shorter* than that of the event list simulator, nearly an order of magnitude better than the lazy approach.

Although the above results are very encouraging, we hasten to add that reprogramming the application to utilize greater lookahead is not always possible. For example, reprogramming the server process in this way was possible because FCFS queues were used. If the network contained *prioritized* queues, one could not forward a message as soon as it was received. This is because the departure timestamp cannot be determined until it can be determined that a higher priority message (job) will not preempt the original. Simulations such as these *inherently* contain poor lookahead properties. Our empirical data suggests that there is little hope in achieving good speedup for such applications using deadlock avoidance or deadlock detection and recovery techniques, except perhaps in special circumstances such as feedforward networks that contain no feedback loops.

Finally, we note that at first glance, reprogramming logical processes to maximize lookahead may complicate other aspects of the simulation, e.g., statistics collection. For example, the eager server does not pause for departure events, so statistics that are most easily collected at job departure must be collected at other points in simulated time. This problem is easily reconciled by scheduling local departure events (as was done before) that are only used for statistics collection purposes. To ensure eager message forwarding, one need only see that incoming messages

are forwarded as soon as possible, i.e., as soon as they are received by the FCFS server. Some care must be taken in programming the server, however, to ensure that it does not delay processing messages arriving from *other* processes because it is waiting to be sure it does not receive a message with smaller timestamp from itself!

6. A PERSPECTIVE ON LOOKAHEAD: NON-EVENTS

The influence of lookahead on performance can be viewed from the following perspective: Processes with very good lookahead ability are able to act in a largely autonomous fashion. Their behavior is not heavily influenced by the activities of other processes, so they can perform simulation work at "full speed," limited only by the rate at which they can be fed work, and the number of CPU cycles (or other resources) that they can obtain. The eager queueing network process is a good example of such autonomous behavior.

On the other hand, processes with poor lookahead ability must frequently obtain additional information from other processes before they can safely proceed. This is unfortunate because not only must such processes wait for real events to be generated by other processes, but often they must also wait to be sure other events will *not* occur. The fact that an airplane will *not* crash and close the airport in the next moment of simulated time must be discovered before the airport process can go about its business of deciding what *will* happen next. We call these "phantom" events that never materialize *non-events.*

In the deadlock avoidance simulator, knowledge of non-events is passed explicitly through the use of null messages. In the deadlock detection and recovery simulator, this information is obtained by system deadlock — processes with messages waiting to be processed must wait until they can be certain that specific events will *not* occur. Certainty as to the eventuality of non-events comes about when the deadlock is broken, and the deadlock resolution protocol is invoked. Because sequential, event list simulators incur no overhead for non-events, speedup in

the distributed simulator is correspondingly poor.

7. CONCLUSIONS

Extensive empirical performance evaluations of distributed simulation programs were performed using the deadlock avoidance and deadlock detection and recovery algorithms developed by Chandy and Misra. The principal results of these studies are:

- The lookahead ability of logical processes plays a critical role in determining the efficiency of the deadlock avoidance and deadlock detection and recovery algorithms. This is attributed to the fact that processes must spend an excessive amount of time waiting to be sure that certain events will *not* occur if their lookahead ability is poor.
- Message avalanche was observed in the deadlock detection and recovery simulator for moderate to high message populations, and was necessary to achieve efficient execution. The poorer the lookahead ability of a process, the larger the message population necessary to achieve avalanche. If lookahead is sufficiently poor, avalanche may never be observed for workloads of practical interest.
- Deadlock detection and recovery simulators containing different types of logical processes can be adversely affected by a small number of processes that exhibit poor lookahead ability. The existence of a few such processes can greatly increase the message population necessary to achieve avalanche, even if many other processes contain very good lookahead properties.
- Queueing networks that contain cycles, previously thought to be ill suited for conservative distributed simulation algorithms, can achieve good performance if servers are reprogrammed to take advantage of all available lookahead. Other networks that inherently contain poor lookahead properties, e.g., prioritized queues, appear ill suited for these algorithms.
- Distributed simulation using deadlock avoidance or detection and recovery algorithms is a viable approach to speeding up many simulation workloads containing moderate to high

degrees of parallelism, even if there are many feedback loops in the logical process topology. However, abundant parallelism does not guarantee good performance.

The appropriate speedup technique that should be used for a specific simulation problem is highly dependent on the system being simulated, arguing for simulation systems that are sufficiently flexible to support a wide range of approaches. It is becoming more and more clear that programmers must be intimately familiar with the intended application in order to judge which performance enhancement approach will be most effective.

REFERENCES

Biles, W., "Statistical Considerations in Simulation on a Network of Microcomputers," *1985 Winter Simulation Conference Proceedings*, pp. 388-393 (December 1985).

Chandak, A. and Browne, J. C., "Vectorization of Discrete Event Simulation," *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 359-361 (August 1983).

Chandy, K. M. and Misra, J., "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering* **SE-5**(5) pp. 440-452 (Sept. 1979).

Chandy, K. M. and Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM* **24**(4) pp. 198-206 (April 1981).

Comfort, J. C., "The Simulation of a Master-Slave Event Set Processor," *Simulation* **42**(3) pp. 117-124 (March, 1984).

Fujimoto, R. M., "VLSI Communication Components for Multicomputer Networks," Ph. D. dissertation, Electronics Research Laboratory Report No. UCB/CSD 83/136, University of California, Berkeley, CA (1983).

Jefferson, D. R., "Virtual Time," *ACM Transactions on Programming Languages and Systems* **7**(3) pp. 404-425 (July 1985).

Jones, D. W., "An Empirical Comparison of Priority-Queue and Event-Set Implementations," *Communications of the ACM* **29**(4) pp. 300-311 (April 1986).

Kaudel, F. J., "A Literature Survey on Distributed Discrete Event Simulation," *Simuletter* **18**(2) pp. 11-21 (June 1987).

Misra, J., "Distributed-Discrete Event Simulation," *ACM Computing Surveys* **18**(1) pp. 39-65 (March 1986).

Reed, D. A., "Parallel Discrete Event Simulation: A Case Study," *18th Annual Simulation Symposium*, pp. 95-107 (March 1985).

Reed, D. A., Malony, A. D., and McCredie, B. D., ''Parallel Discrete Event Simulation: A Shared Memory Approach,'' *IEEE Transactions on Software Engineering*, (to appear).

Seethalakshmi, M., ''A Study and Analysis of Performance of Distributed Simulation,'' MS Report, University of Texas, Austin, Texas (May 1979).

Sleator, D. D. and Tarjan, R. E., ''Self-Adjusting Binary Search Trees,'' *Journal of the ACM* **32**(3) pp. 652-686 (July 1985).

Thomas, R. H., ''Behavior of the Butterfly Parallel Processor in the Presence pf Memory Hot Sopts,'' *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 46-50 (August 1986).

Thomas, R. H., ''The Uniform System Approach to Programming the Butterfly Parallel Processor,'' BBN Report No. 6149, BBN Laboratories Inc. (June 1986).

Table 1.
Timestamp Increment Distributions

Distribution	Expression to compute random values[a]	Lookahead[b]	LAR[c]
Deterministic	1.0	1.0	1.0
Biased 0.9-1.0	0.9 + 0.2 rand	0.9	1.11
Exponential	0.1-*ln*(rand)	0.1	11.0
Uniform	0.1+rand	0.1	5.5
Bimodal	0.95238 rand + if rand < 0.1 then 9.5238 else 0	0.1	10.0

[a] rand returns a random value uniformly distributed between 0 and 1.
[b] Lookahead is defined as the minimum value for the distribution.
[c] Lookahead Ratio (*LAR*) is defined as the mean divided by the lookahead.

Table 2.
Hardware Parameters

Operation	Execution Time (microseconds)
Local memory reference	0.60
Remote memory reference	4.0
Register-to-register instruction	0.71
16 bit Load (Local Memory)	1.3
16 bit Load (Remote Memory)	6.3
Parameterless function call	6.9
Atomic inclusive OR	20

Table 3.
Parameters used in experiments.

Parameter	Value
topology	toroid
routing probability	uniform
CPU time distribution	exponential
mean CPU time per message	1 millisecond
message population	4 per *LP*
scheduling	static

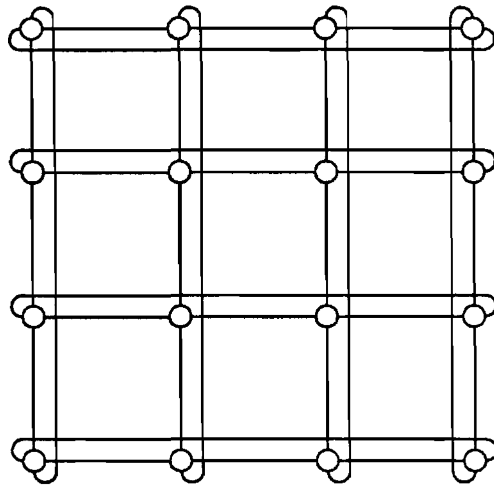

Figure 1. 16 node toroid network.

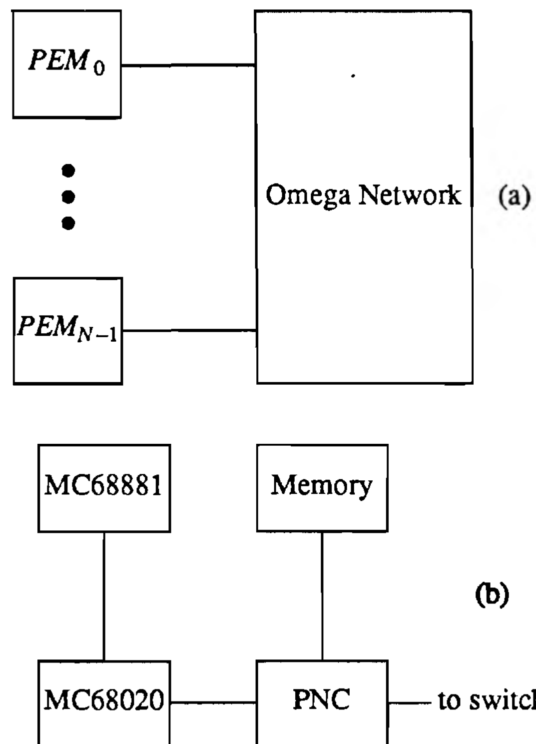

Figure 2. (a) Butterfly Multiprocessor. (b) Single PEM Node.

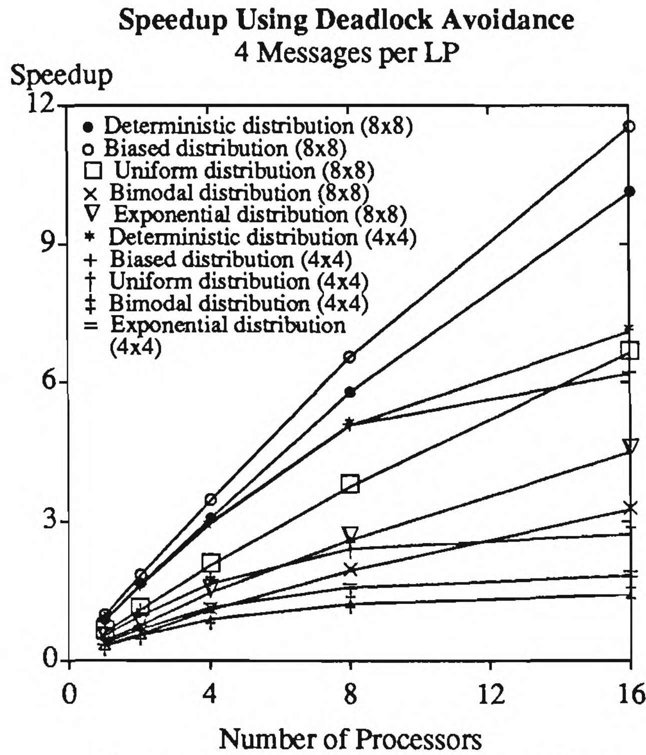

Figure 3. Speedup using deadlock avoidance.

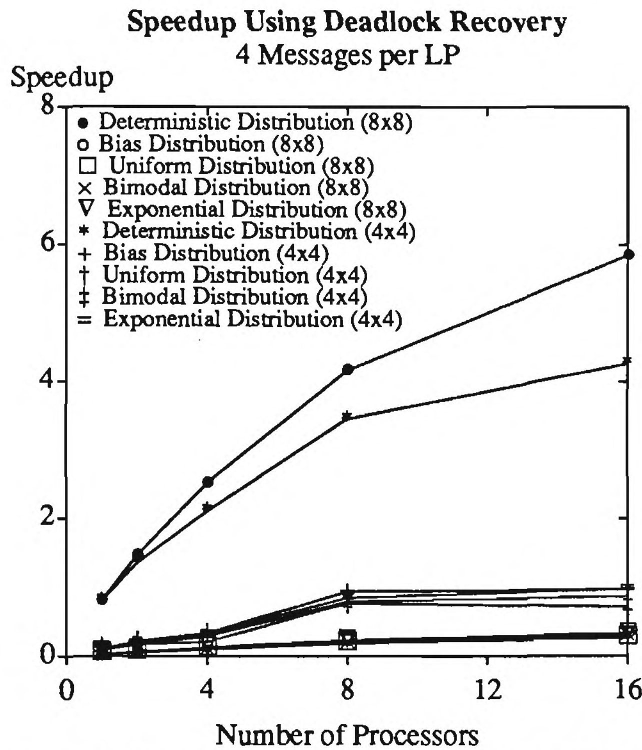

Figure 4. Speedup using deadlock detection and recovery.

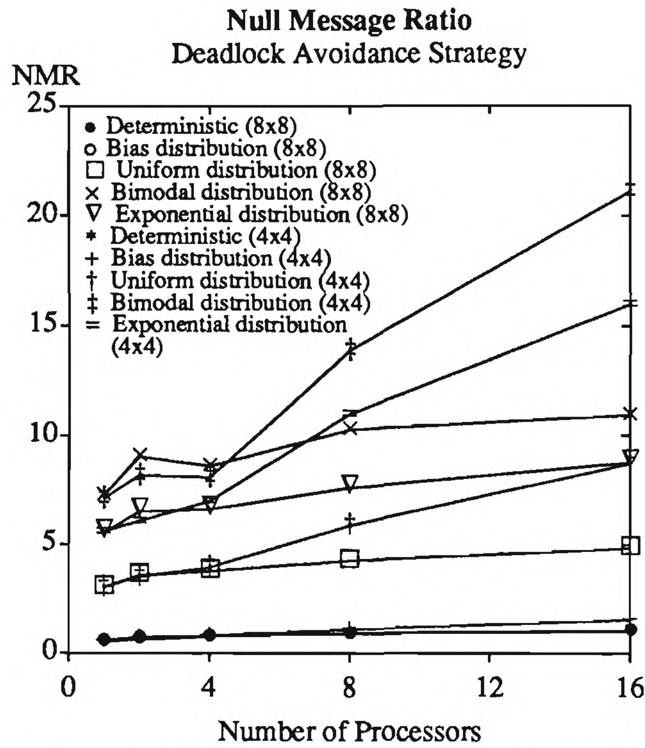

Figure 5. Overhead in deadlock avoidance simulator.

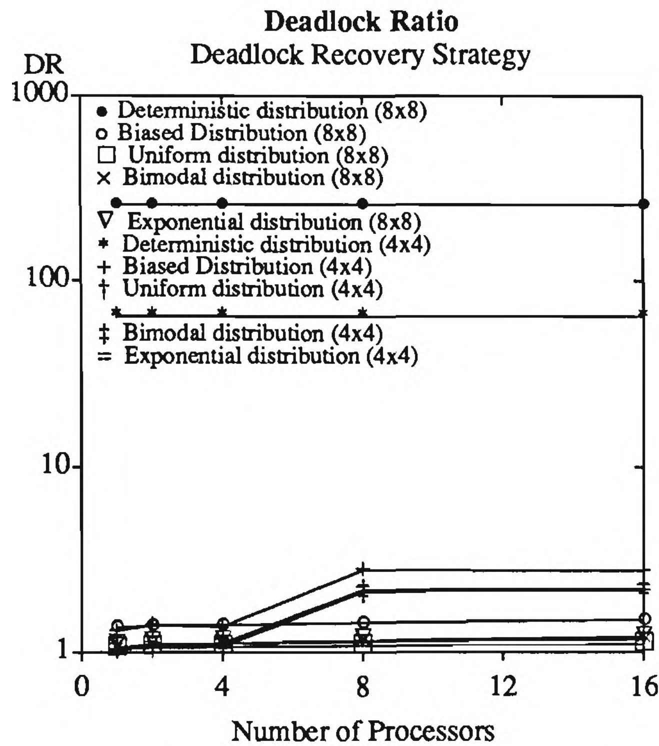

Figure 6. Overhead in deadlock recovery simulator.

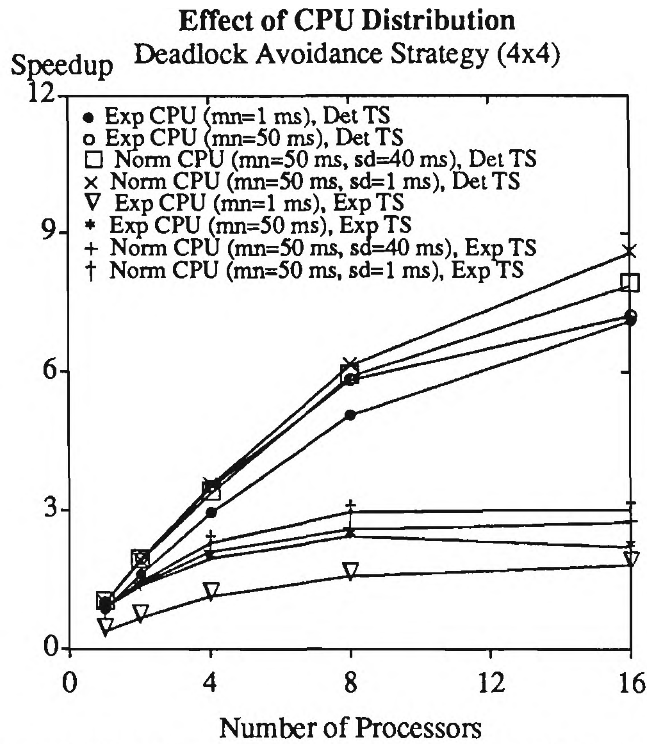

Figure 7. Speedup as CPU grain varied — deadlock avoidance.

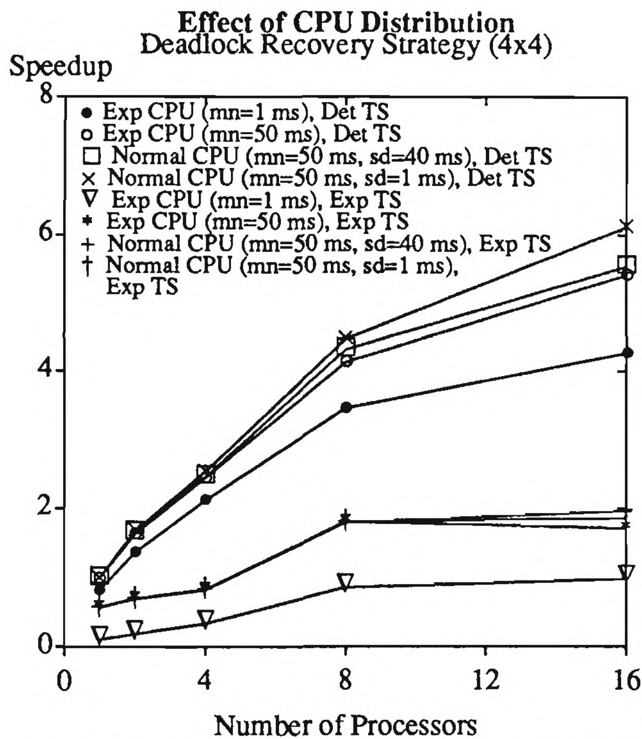

Figure 8. Speedup as CPU grain varied — deadlock recovery.

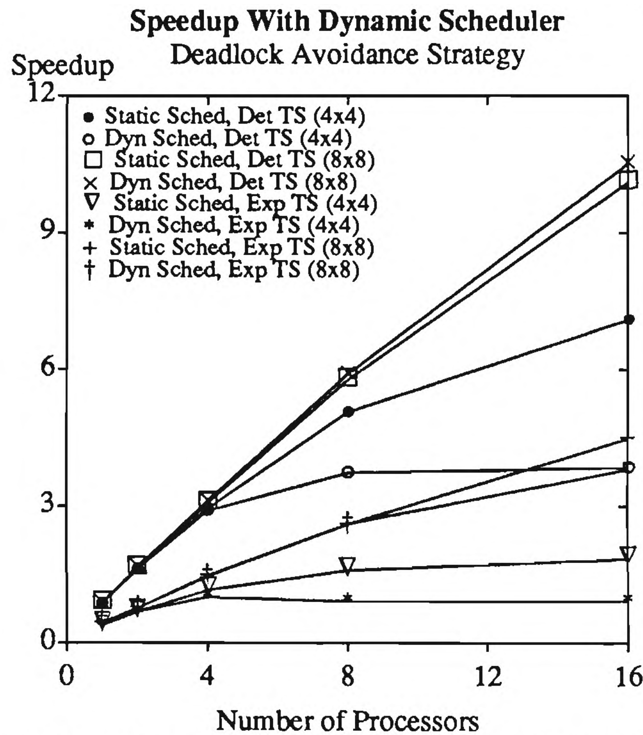

Figure 9. Speedup using dynamic scheduler — deadlock avoidance.

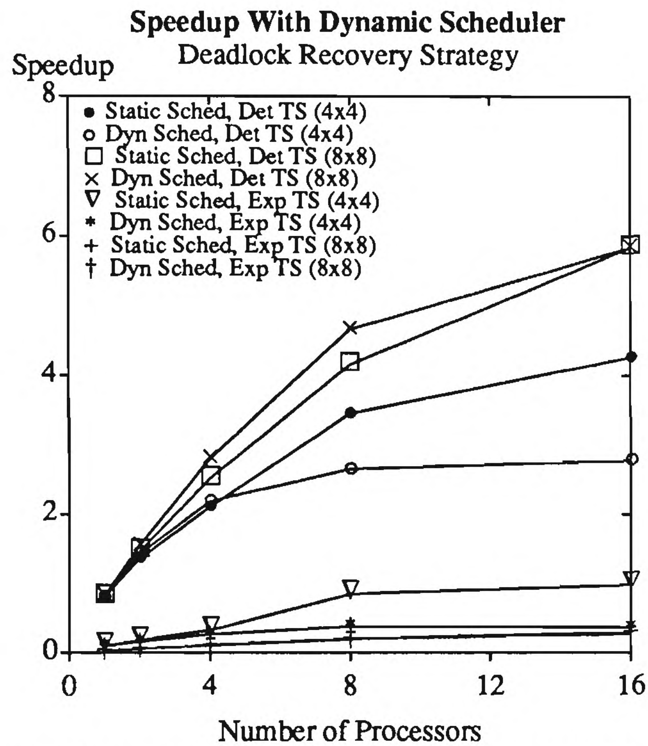

Figure 10. Speedup using dynamic scheduler — deadlock recovery.

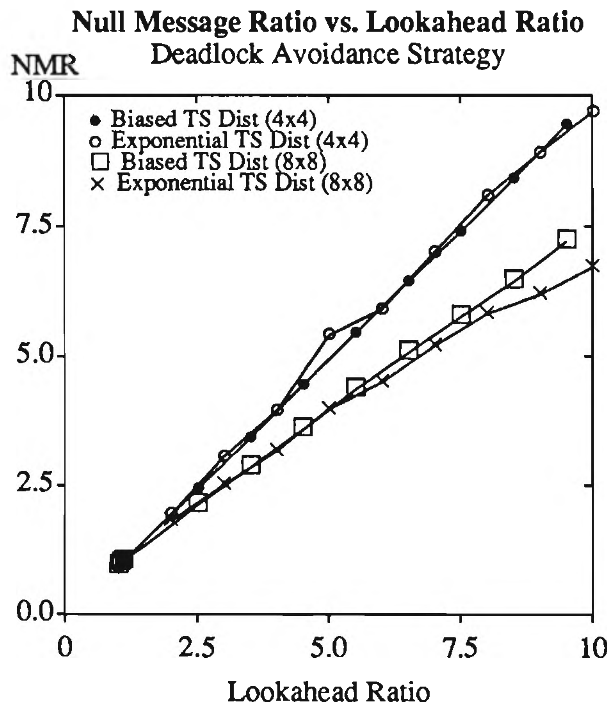

Figure 11. Overhead as lookahead ratio varied — deadlock avoidance.

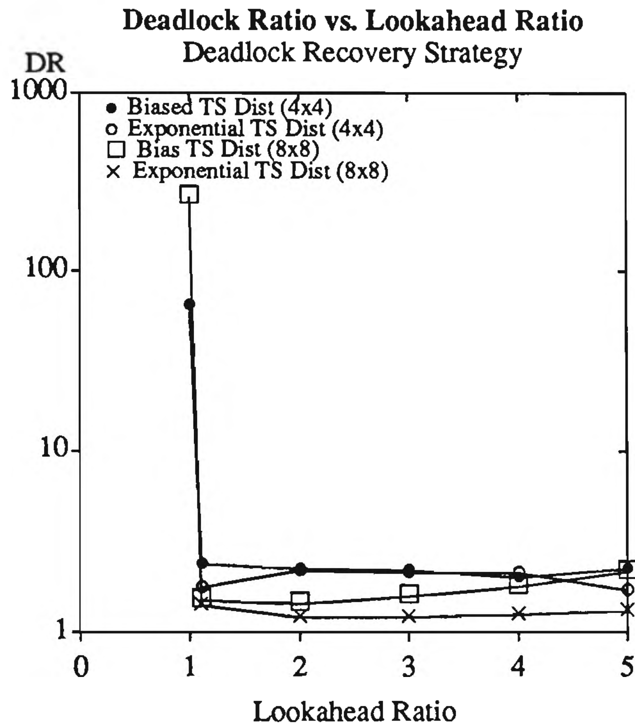

Figure 12. Overhead as lookahead ratio varied — deadlock recovery.

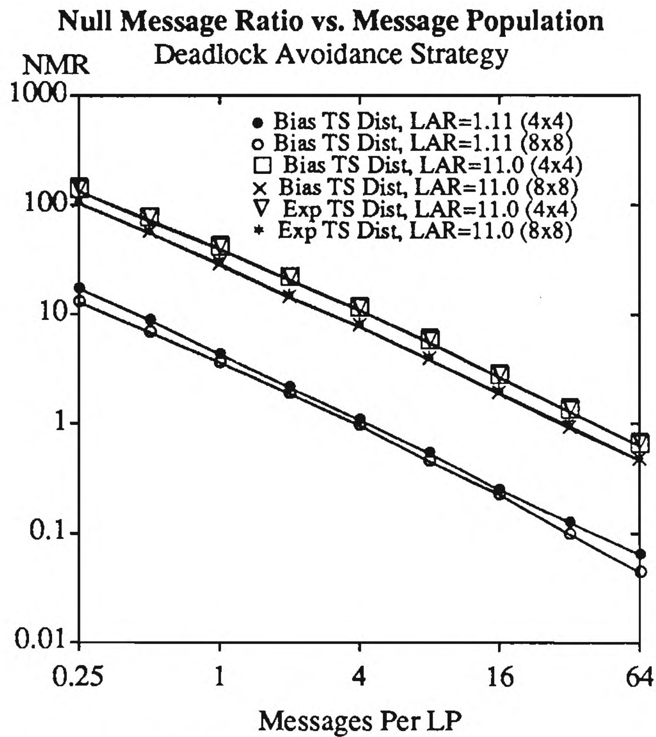

Figure 13. Overhead as message population varied — deadlock avoidance.

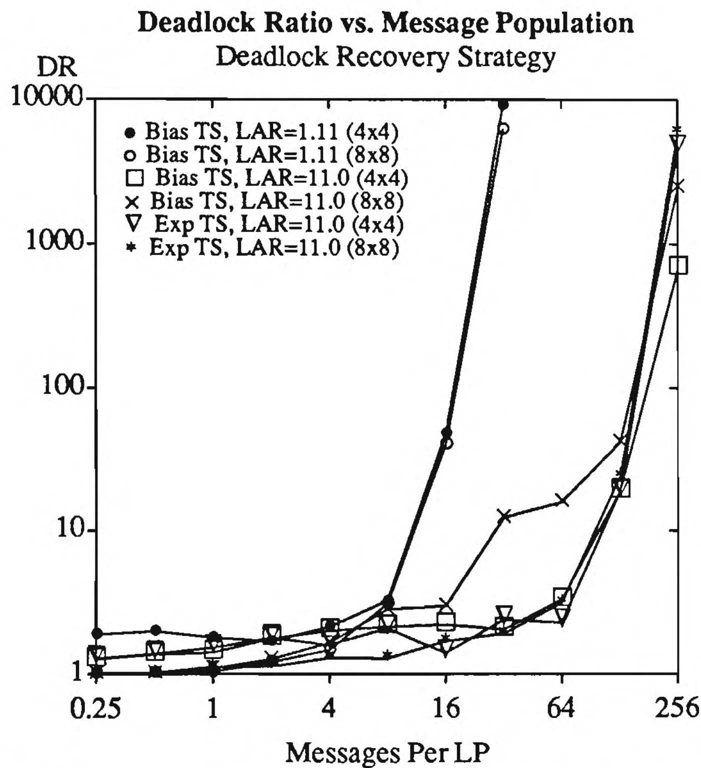

Figure 14. Overhead as message population varied — deadlock recovery.

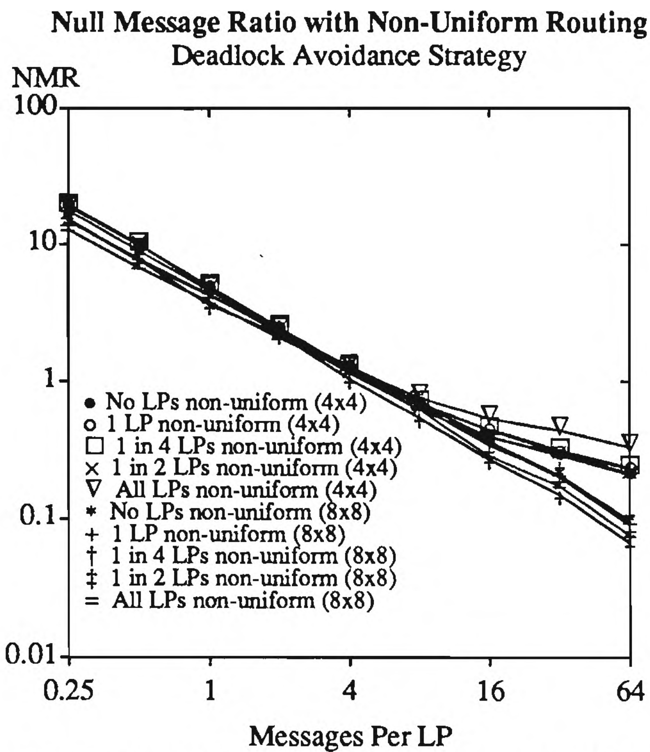

Figure 15. Overhead with non-uniform routing — deadlock avoidance.

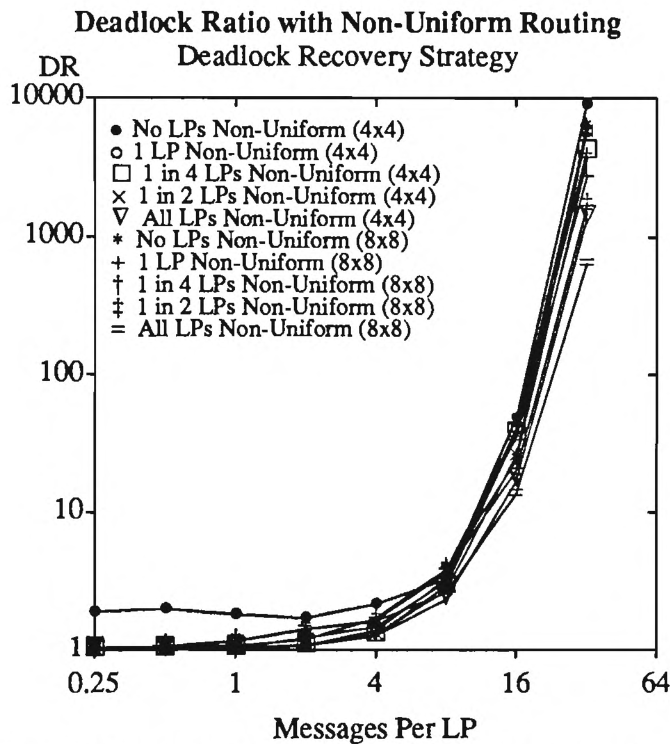

Figure 16. Overhead with non-uniform routing — deadlock recovery.

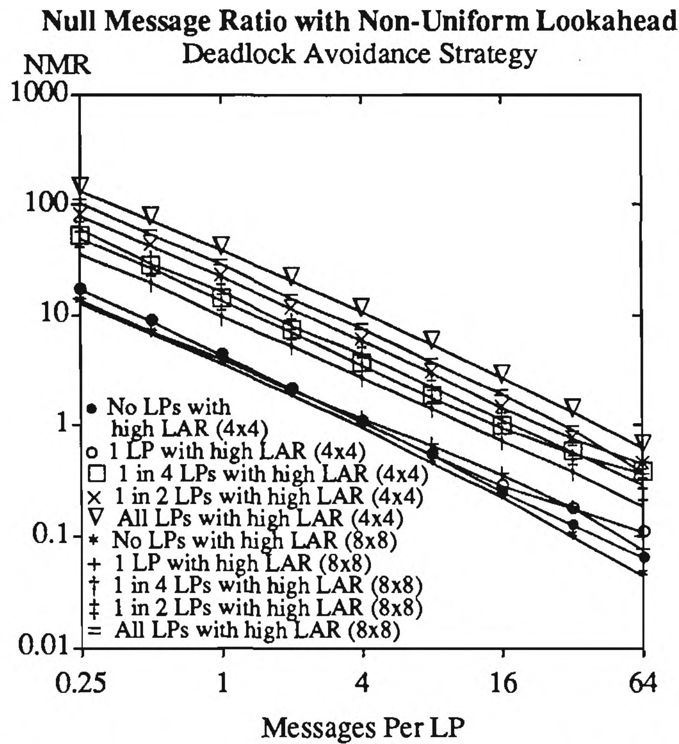

Figure 17. Overhead with non-uniform lookahead — deadlock avoidance.

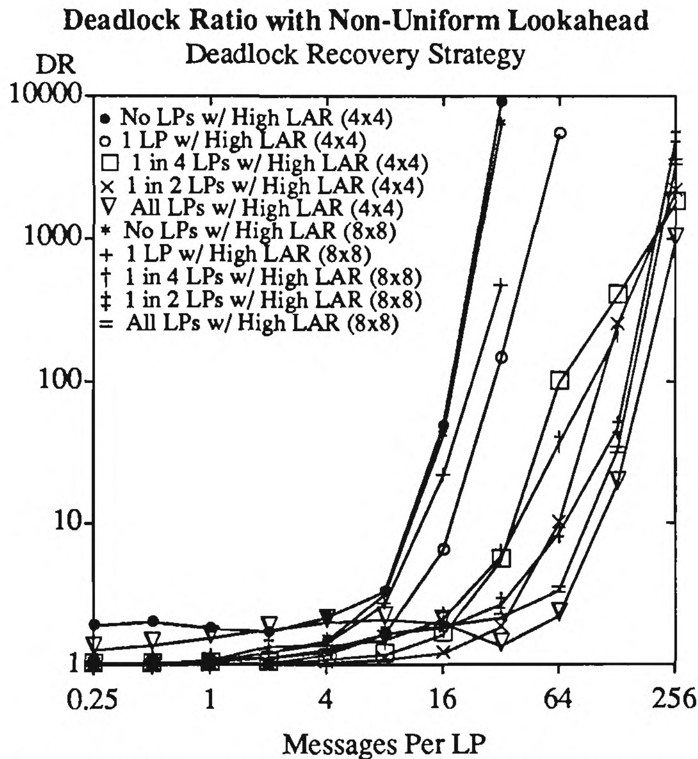

Figure 18. Overhead with non-uniform lookahead — deadlock recovery.

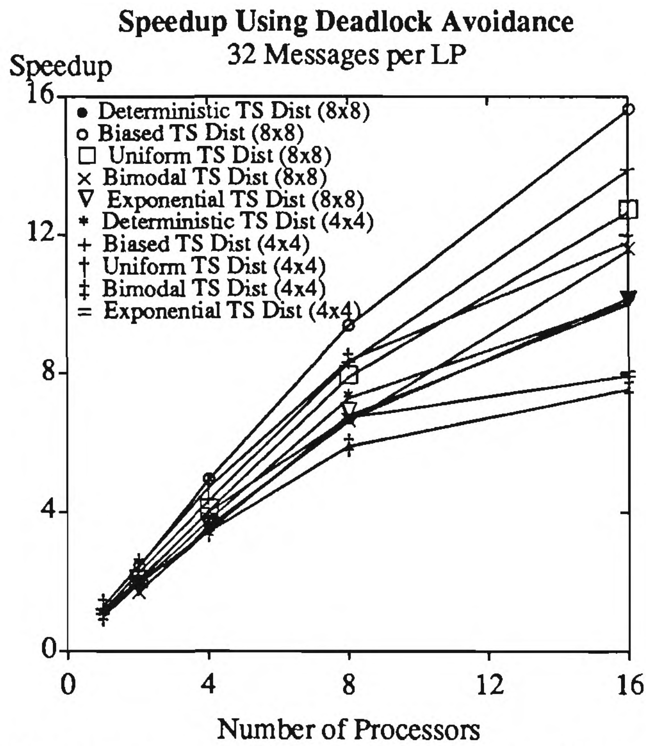

Figure 19. Speedup with high message population — deadlock avoidance.

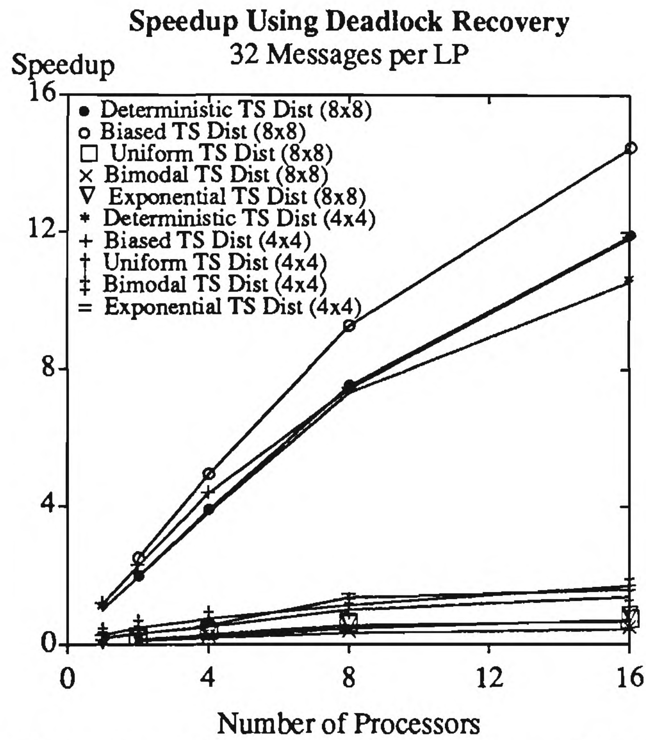

Figure 20. Speedup with high message population — deadlock recovery.

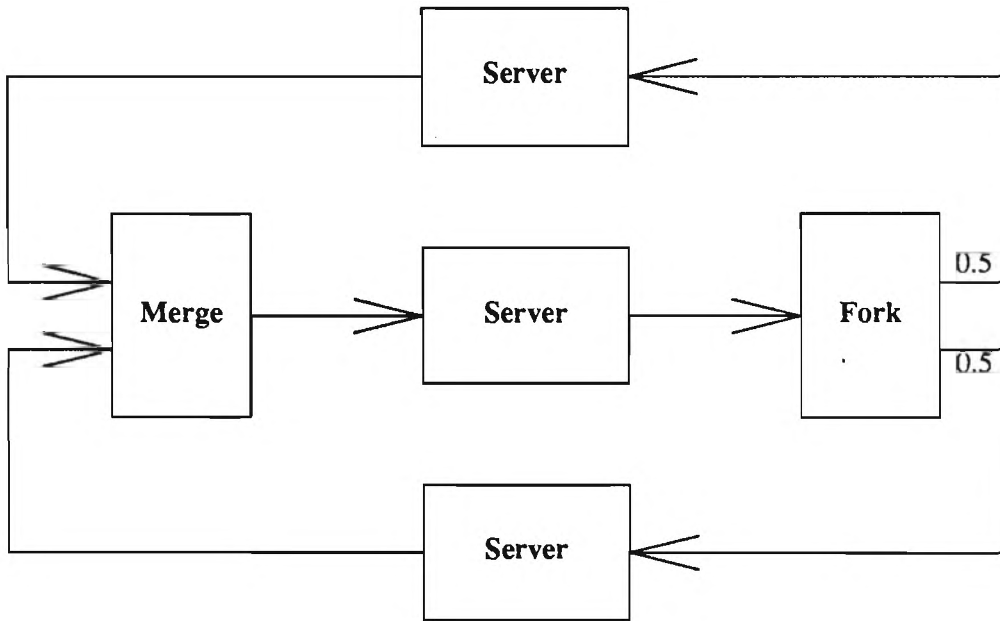

Figure 21. Central server queueing model.

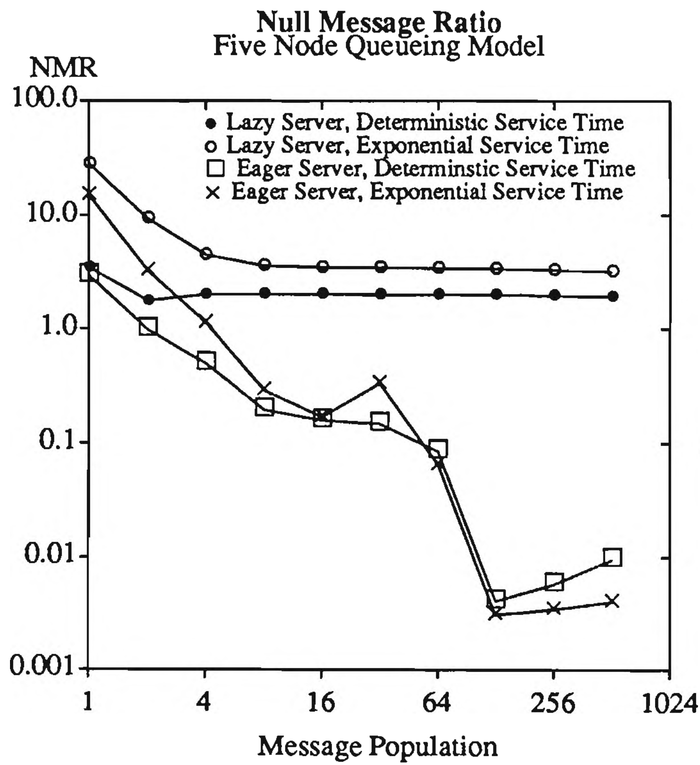

Figure 22. Null message ratio for central server queueing model.

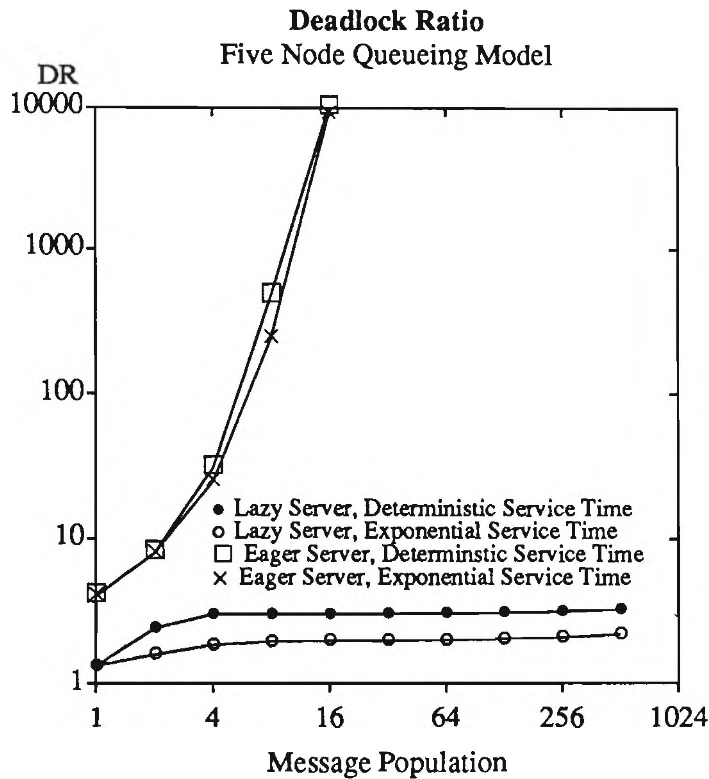

Figure 23. Deadlock ratio for central server queueing model.

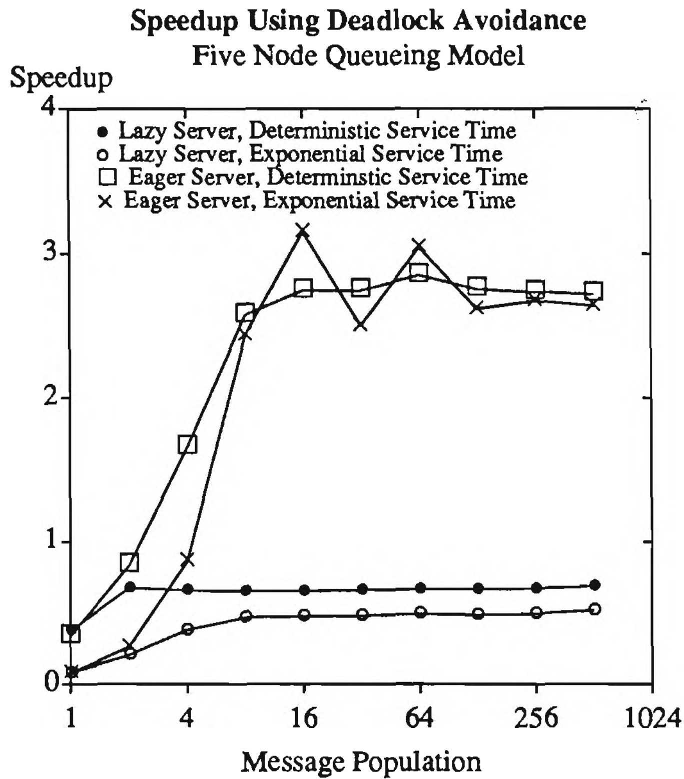

Figure 24. Speedup of queueing model using deadlock avoidance.

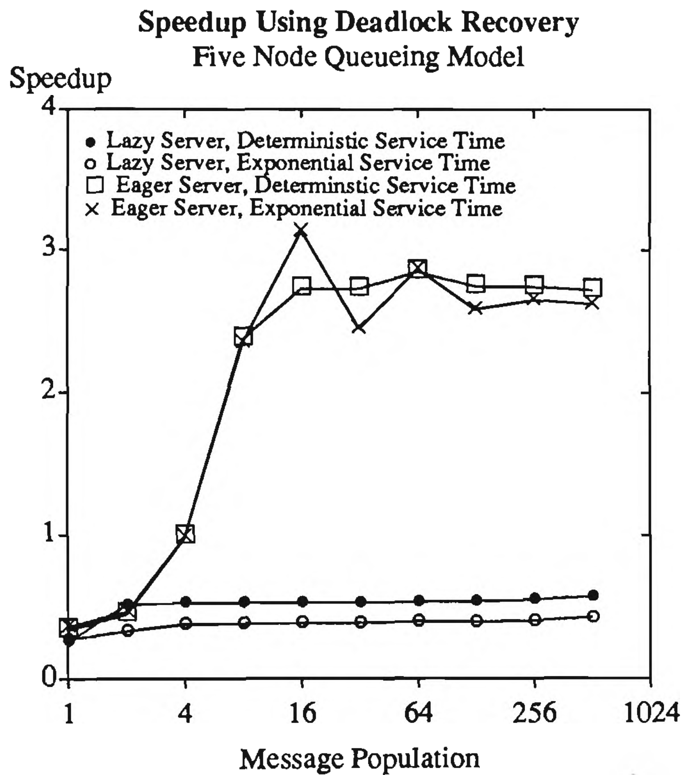

Figure 25. Speedup of queueing model using deadlock recovery.