Parallel Discrete Event Simulation Using Shared Memory

DANIEL A. REED, MEMBER, IEEE, ALLEN D. MALONY, AND BRADLEY D. McCREDIE, STUDENT MEMBER, IEEE

Abstract—**With traditional event list techniques, evaluating a detailed discrete event simulation model can often require hours or even days of computation time. By eliminating the event list and maintaining only sufficient synchronization to ensure causality, parallel simulation can potentially provide speedups that are linear in the number of processors. We present a set of shared memory experiments using the Chandy-Misra distributed simulation algorithm to simulate networks of queues. Parameters of the study include queueing network topology and routing probabilities, number of processors, and assignment of network nodes to processors.**

Index Terms—**Chandy-Misra algorithm, deadlock recovery, discrete event simulation, distributed simulation, parallel processing.**

Introduction

HISTORICALLY, two of the major techniques for modeling systems have been queueing theory and discrete event simulation. When effective, queueing theoretic techniques can quickly provide mathematical insight into the behavior of systems over a broad range of parameter values. Their major limitation is the number of restrictive assumptions that must be satisfied to ensure accuracy. Conversely, simulation models can mimic a real-world system as closely as understanding permits and needs require. However, highly detailed simulation models can be computationally taxing. Computer systems simulations are particularly vexing because simulated events occur on a millisecond or microsecond time scale, often for many simulated minutes.

For example, simulating the behavior of a processor executing a user or system program may involve millions or even tens of millions of events. In one architecture performance study, we recently examined the performance of allocation strategies for register windows in reduced instruction set computers (RISC's) [17], [18] as a function of multiprogramming level [16]. This analysis required instruction-level simulation for many different program mixes and consumed many hours of processor time.

Manuscript received March 31, 1987; revised September 30, 1987. D. A. Reed was supported in part by the National Science Foundation under Grant DCR 84-17948 and NASA under Contract NAG-1-613. A. D. Malony was supported in part by the National Science Foundation under Grants DCR 84-10110 and DCR 84-06916, the Department of Energy under Grant DE-FG02-85ER25001, and a donation from IBM.

D. A. Reed is with the Department of Computer Science, University of Illinois, Urbana, IL 61801.

A. D. Malony is with the Center for Supercomputing Research and Development, University of Illinois, Urbana, IL 61801.

B. D. McCredie is with the Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL 61801.

IEEE Log Number 8819726.

Simulation of complex (VLSI) digital circuits for logic verification and fault analysis is another example of the computational constraints imposed on simulation of computing components. Although such simulations can consume *months* of machine time [20], [11], designers have little choice; an untested design is unacceptable. Moreover, simulation complexity continues to increase dramatically; technology advances are doubling the number of circuits per chip every 1–2 years.

At a much higher level than logic design, we recently encountered difficulties while studying multicomputer networks [21], [23], designed, ironically, to solve computationally intensive problems. Briefly, a multicomputer network is a large number of interconnected computing nodes that asynchronously cooperate via message passing to execute the tasks of parallel programs.[1] Many design issues must be resolved before constructing a multicomputer network (e.g., the relative speeds of computation processors and internode communication links, topology of connecting communication links, buffer requirements for messages, and memory sizes). Although some of these issues can be attacked analytically, most are analytically intractable and can only be resolved via simulation. A parametric simulation study, whose individual simulation runs cover several minutes of simulated time, typically requires several hundred hours of processor time.

Although each of the three preceding examples, processor simulation, logic simulation, and network simulation, is very different, they share a common need for faster simulation techniques. Processor simulation reflects the fetch/decode/execute cycle of instruction execution and is, by its nature, sequential; parallelizing this application is the subject of architectural research. Circuit simulation, although clearly amenable to parallel processing [20], typically involves sychronous activation of many entities. In contrast, network simulation is typically asynchronous.

Prior Work

It might initially appear that evaluating models of many complex systems is both analytically *and* computationally

[1]Hypercubes [25] are a special case of multicomputer networks.

intractable. However, recent developments have suggested that the computation time for *some* simulations can be reduced via either vector processing [6] or distributed simulation [20], [4], [15].

Vector Simulation

Chandak and Browne [6] recently proved an item of computing folklore—discrete event simulation models cannot always be vectorized. Specifically, they showed that any network of queues model containing feedback is not vectorized. This result is quite negative: most interesting simulation models contain some type of feedback.

Given this result, we recently investigated the level of vectorization *practically* achievable [22] by instrumenting a discrete event simulation of queueing network models on a Cray X-MP. Although we simulated a variety of workloads and queueing network models, the observed vectorization level never exceeded 5 percent. Even this fraction was primarily attributable to initialization code. Thus, the efficacy of vector simulation is in doubt.

Distributed Simulation

The inherently sequential nature of event list manipulation limits the potential parallelism of standard simulation models. The head of the event list must be removed, the simulation clock advanced, and the event performed (possibly causing new events to be added to the event list). Although techniques for performing event list manipulation and event simulation in parallel have been suggested [8], [9], large scale performance increases seem unlikely. Only by eliminating the event list, in its traditional form, can additional parallelism be obtained; this is the goal of distributed simulation.

If one views a simulation model as a network of interacting servers and queues, distributed simulation maps each server/queue pair onto a processor of a multicomputer network. Each processor operates with its own simulation clock, and there is no global event list. Event occurrence times are transmitted across communication links to appropriate recipients (e.g., a message departing one server for another would carry with it its time of departure).

Several distributed simulation techniques have been proposed, notably the Chandy—Misra algorithm [2]–[4] and the Time Warp algorithm [15]. The Chandy-Misra algorithm and Time Warp differ in their approach to time management. The former is pessimistic, advancing the processor simulation clocks only when conditions permit. In contrast, Time Warp assumes the simulation clocks can be advanced until conflicting information appears; the clocks are then *rolled back* to a consistent state, a so-called "time warp."

Both the Chandy-Misra algorithm and Time Warp have been simulated [25], [15], but few experimental results have yet been reported. In the remainder of this paper, we present the Chandy-Misra algorithm [4] and the results of an extensive study of its performance on a shared memory parallel processor when simulating queueing network models. Parameters of the study include queueing network topology and routing probabilities, number of processors, and assignment of queueing network servers to processors. We conclude with a summary of lessons learned and directions for future research.

The Chandy-Misra Distributed Simulation Algorithm

Consider some *physical* system composed of independent, interacting entities. A natural, distributed simulation of the physical system creates a topologically equivalent system of *logical* nodes. Interactions between two physical nodes are modeled by exchange of timestamped messages. The timestamp is the simulated message arrival time at the receiving node.

Each logical node is subject to some constraints. First, node interaction is *only* via message exchange; there are no shared variables. Second, each node must maintain a clock, representing the local simulated time. Finally, the timestamps of the messages generated by each node must be nondecreasing.

Intuitively, the distributed simulation has no single "correct" simulation time; each node operates independently subject only to those restrictions necessary to ensure that events happen in the correct simulated order (i.e., *causality* is maintained). Independent events can be simulated in parallel even if they occur at different simulated times.

Message timestamps and node clocks are a manifestation of the need for causality: the behavior of a node P at its simulated time T cannot be influenced by any information transmitted to it after time T. This constraint has rather dramatic ramifications. Consider a node P that receives messages from two other nodes A and B. When a message arrives from node A, one would expect node P to interpret the message, perhaps producing a message as a consequence. However, if the arrival time of the message from A is greater than the arrival time of the *last* message from B, the message from A *cannot* be processed. Why? A message might later arrive from B with a smaller timestamp. Thus, a node with multiple inputs must wait until it receives messages from all inputs before selecting a message to interpret.

Although appealing, distributed simulation poses several pragmatic problems:

- Optimal assignment of nodes to processors is expensive.
- Only a subset of all discrete event simulation models are amenable to distributed simulation. As noted above, shared variables are not permitted. Hence, no events depending on the global system state are possible.
- Deadlocks can occur in most simulation models. Recall that a node must insure that no information received later can affect its output; this may require waiting for additional inputs. A cycle of waiting nodes results in deadlock.

The assignment problem deserves additional comment. The natural hardware realization of the network of nodes is a multicomputer network. Pragmatics dictate, however, that the multicomputer network have a fixed interconnection topology. Thus, a node network must be mapped onto the multicomputer network. Unfortunately, finding an optimal mapping is known to be NP-complete [7]. In practice, scheduling heuristics must be used; suboptimal mappings are produced at considerably less computational expense. Even if an optimal mapping were found, the respective topologies of the multicomputer and node networks may be ill-suited, resulting in either large communication delays or processor load imbalances.[2]

Like node scheduling, deadlock resolution, although difficult, is solvable. Chandy and Misra have described two distributed deadlock resolution techniques, avoidance and recovery [4]. For specificity's sake, we describe these techniques in the context of our RESQ implementation [24] for simulating queueing networks, based on the distributed simulation implementation described in [25]. This implementation uses minimal knowledge of the structure of the simulated systems, including minimal lookahead [13].

In the RESQ scheme, there are five node types: *service*, *fork*, *merge*, *source*, and *sink*. Service nodes correspond to the interacting entities of a physical system (e.g., servers in a queueing network). In contrast, fork and merge nodes exist only to provide routing. Finally, source and sink nodes respectively create and destroy network messages. Thus, the central server model [1] of Fig. 1(a) would be represented, using the RESQ scheme, as shown in Fig. 1(b).

The RESQ notation for describing network models has been widely used as an input language for sequential simulations. Using RESQ for parallel simulation entails modifying the semantics of some node types. Specifically, distributed simulation with deadlock avoidance [3] requires fork, merge, and server nodes[3] to send *null* messages under certain conditions. These null messages are time stamped and tell the receiving node that no *real* message will be forthcoming before the specific time. This enables the receiver to process outstanding messages with the assurance that its actions will not be revoked at a later time.

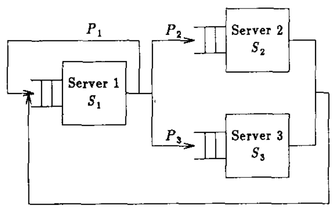
A *fork* node accepts a single stream of message inputs and distributes this stream across N outputs. Upon receiving a real or null input message, a fork node routes the message to the selected output and creates $N - 1$ null messages, each with the same timestamp as the message received. One null message is routed to each destination not selected.

A *merge* node accepts N streams of message inputs and routes them *in timestamp order* to a single output. As noted earlier, the time stamp ordering forces the node to wait for messages, perhaps null, on all inputs before producing an output.

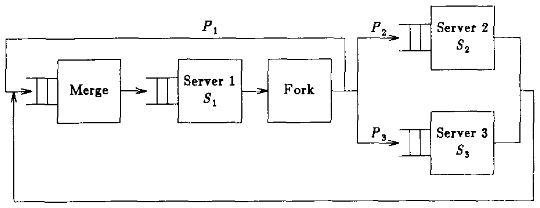
Finally, a *server* node accepts a single input stream and

[2]Scheduling difficulties can be ameliorated by a shared memory implementation of message passing. This approach is discussed later.

[3]By definition, source and sink nodes can never be members of a deadlock set.



(a)



(b)

Fig. 1. (a) Central server queueing model. (b) RESQ representation of central server model.

produces a single output stream. When the time of last message arrival is greater than the time of last message departure, and the server has no real messages to process, it produces a null message with a timestamp equal to the minimum time of next real message departure.

Although the null message technique provably avoids deadlocks, it does so at the price of potentially high overhead. In networks containing many fork/merge cycles, simulations have shown that the ratio of null to real messages can be very high [25], [21]. The alternative to deadlock avoidance is deadlock detection and recovery. In this approach, the distributed simulation alternates between computation and recovery phases. As proposed by Chandy and Misra [4], the simulation runs until a distributed deadlock detection algorithm verifies deadlock. The simulation then enters a deadlock recovery phase and finally returns to active computation.

Although deadlock detection and recovery avoids null messages, it does so by *diverting* computation resources to detection and recovery. The performance advantage of this approach versus deadlock avoidance depending on the relative costs of synchronization and message passing.

In light of the many potentially performance-limiting problems with distributed simulation, it seems important to analyze the performance of distributed simulation in a realistic environment. Many such performance studies of traditional simulation algorithms have been conducted, and, based on these studies, new event list algorithms have been proposed [12], [27]. Until recently, only limited *simulation* studies of distributed simulation were reported [25], [15], [22]; little were empirical data is available. In the remaining sections we discuss our experimental environment, implementation, and experimental results.

Experimental Environment

All simulation experiments were conducted on a Sequent Balance 21000 containing 20 processors and 16 Mbytes of memory. Each Balance 21000 processor is a National Semiconductor NS32032 microprocessor, and all

TABLE I
TYPICAL OPERATION TIMES FOR THE SEQUENT BALANCE 21000

Operation	Time (μsec)
Lock/unlock	60
Subroutine call/return	60
System call	400
Context switch	1000
Process creation	60000

processors are connected to shared memory by a shared bus with an 80 Mbyte/s (maximum) transfer rate. Each processor has an 8K byte, write-through cache and an 8K byte local memory; the latter contains a copy of selected real-only operating system data structures and code.

The Dynix™ operating system for the Balance 21000 is a variant of UC Berkeley's 4.2BSD UNIX® with extensions for processor scheduling. Because Dynix schedules all processes from a common pool, a process may execute on different processors during successive time slices. However, as long as the number of active processes is less than the number of processors, each process will execute on a separate processor. In this case, process and processor are equivalent notions. To the time-sharing user, the Balance 21000 appears as a standard UNIX system, albeit with better interactive response time.

Parallel programs consist of a group of UNIX processes that interact using a library of primitives for shared memory allocation and process synchronization. Shared memory is implemented by mapping a region of physical memory into the virtual address space of each process. Once mapped, shared memory can be allocated to specific variables as desired.

Access to the shared memory region is controlled by software spin *locks* and *barriers*. These locks, semantically equivalent to binary semaphores, provide mutual exclusion. Barriers are used to synchronize a group of processes; a process reaching a barrier is forced to wait until all processes in the specified group have reached the barrier.

In summary, the Balance 21000 is a "standard" UNIX system with minimal extensions for parallel programming. Consequently, many parallel operations are dominated by operating system overhead. For comparison with later discussion, Table I shows the elapsed times for typical operations.

SHARED MEMORY IMPLEMENTATION OF DISTRIBUTED SIMULATION

A shared memory multiprocessor, such as the Balance 21000, provides a flexible testbed for studying the performance of distributed simulation. The problems associated with mapping a node network onto a multicomputer network are removed; the shared memory processors are, effectively, completely connected. By implementing message passing using shared memory, communications costs are the same for all processors. However, a shared memory implementation of distributed simulation requires special consideration for synchronization of shared message queues, processor allocation, and deadlock management.

™Dynix is a trademark of Sequent Computer Systems.
®UNIX is a registered trademark of AT&T Bell Laboratories.

In a shared memory implementation of distributed simulation, all node state information, including input message queues, resides in shared memory. Message-based communication between nodes is implemented via shared access to the message queues of each node. Each message queue is protected by a synchronization lock to guarantee mutual exclusion. Synchronization is only necessary, however, if the communicating nodes execute on separate processors.

Before a node can send a timestamped message to another node, it must first acquire a free message from a shared free message list. A lock is necessary to prevent simultaneous access to the free message list. After retrieving a free message, the node timestamps it and writes it to the destination node's message queue, using synchronization primitives to lock and unlock the queue, if necessary. A message is returned to the free message list once it has been processed by the destination node. Because only messages are used for internode communication, the requirement that no simulated events depend on the global system state is still satisfied.

Processor Allocation

There are two basic approaches to processor allocation in a shared memory implementation of distributed simulation. The first approach, *static node assignment*, fixes the assignment of nodes to processors for the duration of the simulation. When the number of network nodes equals the number of allocated processors, each node is assigned to a separate processor. Otherwise, nodes must be clustered, and these clusters are assigned to individual processors. Several clusterings are possible when the number of nodes exceeds the number of processors; each such clustering exhibits different performance. One advantage of static node assignment is that communication between nodes in a cluster can be done "locally" without the overhead for locking message queues. However, intercluster message transmissions require queue locking.

The second approach, *dynamic node assignment*, assigns nodes to processors during the simulation. Idle processors obtain work from a shared queue of unassigned network nodes. This shared node work queue must be locked before a processor can be allocated an unassigned network node. When a processor obtains a node, it satisfies any outstanding work for the node before returning the node to the tail of the node work queue. Because processors are assigned only one node at any time, all communication between nodes must be synchronized to guarantee exclusive access to shared message queues.

With dynamic node assignment, nodes must wait on the work queue until assigned to a processor. The length of this delay depends on size of the node work queue and can be quite large for large networks. However, not all nodes on the work queue have outstanding work (i.e., there are input messages that will generate output messages when processed). In deadlock avoidance mode, for example, those nodes awaiting input can only generate null messages if processed. A natural strategy for improving performance places only those nodes with outstanding work on the work queue. This reduces the size of the node work queue and the waiting delay.

Our implementation of the above *node waiting* strategy is conservative. When a processor identifies a node with no outstanding work, it sets a ''waiting'' flag in the node's state and does not place the node on the work queue. When a message is sent to a waiting node, the processor sending the message will reset the waiting flag for the waiting node and place it on the work queue. The implementation is conservative because the new message may not actually instigate any new work for the node.

To investigate the effects of this node waiting strategy, we also implemented a *no node waiting* scheme. In this approach, a node is immediately placed at the tail of the work queue after it has been processed, even if it has no outstanding work.

Although static node assignment is efficient for nodes within a cluster, the node assignment cannot be changed to balance network load. Conversely, dynamic node assignment naturally adjusts to network load but incurs synchronization overhead not only for all messages but also for access to the node work queue. Which implementation is best for a particular simulation model depends on the relative costs of synchronization and the beneficial effects of load balancing.

Deadlock Avoidance and Recovery

Our deadlock avoidance approach is a straightforward implementation of the algorithm described earlier [24], [4]. In contrast, deadlock recovery merits further discussion.

As described by Chandy and Misra, distributed simulation with deadlock detection and recovery alternates between simulation and distributed deadlock detection and recovery phases. The presence of shared memory obviates the need for most of the protocol for distributed deadlock detection [5]. Instead, each processor sets a flag in global memory when it believes it is deadlocked. A *guardian processor* monitors the global system state and forces the processors to rendezvous at a synchronization barrier when they all report potential deadlock. The deadlock recovery algorithm is then invoked.

Notice, however, that all processors reporting an inability to progress is a necessary but not sufficient condition for deadlock. Between the time a processor P reports potential deadlock and the time the guardian processor sees this report, processor P may have received messages enabling it to progress. Consequently, the processors may appear deadlocked when they are not. To reduce the probability of detecting such false deadlocks, the guardian uses a *backoff* algorithm that must reverify a potential deadlock before invoking deadlock recovery. This algorithm, controlled by an input parameter, weighs the relative cost of forcing synchronization and deadlock recovery for a false deadlock against the lost time when detection of a real deadlock is delayed.

One may well ask why this deadlock detection technique was used, rather than a variation of graph reduction [10] or the distributed deadlock detection proposed by Chandy *et al.* [5]. Simply put, the number and frequency of deadlocks in a distributed simulation is potentially enormous. Hence, deadlock detection and recovery must be fast. To obtain a consistent state for graph reduction, the processors must either exchange messages or synchronize. The overhead of the first is near that for deadlock avoidance. The latter is as expensive as detecting false deadlocks. Thus, detecting some false deadlocks using a backoff mechanism seems a reasonable compromise.

Simulation Experiments

Experimental evaluation of distributed simulation requires not only an implementation but also a set of test cases. This is particularly important in light of earlier simulation studies [25], [22], which showed that the performance of distributed simulation is extremely sensitive to the topology of the simulated network. Simple tests (e.g., tandem queues) have easily interpretable results, but do not reflect typical simulations. Conversely, simulations of complex queueing networks, although realistic, make it difficult to interpret the sources of performance degradation in distributed simulation.[4]

As a compromise, we selected several simple queueing networks and a few complex ones.

- tandem networks (1, 2, 4, 8, and 16 server nodes)
- general, feed-forward networks (6, 10, and 14 nodes),
- cyclic networks (2, 4, and 8 nodes)
- central server networks (5 nodes), and
- cluster networks (10 and 18 nodes).

The tandem and feed-forward networks are open networks and contain no cycles. With potentially linear speedup, they represent the best-case performance of distributed simulation. The cyclic networks show the performance degradation of tandem networks when they are closed. As an often used model of computer systems, central server networks have pragmatic importance [1]. In addition, they have nested cycles, a more restrictive constraint than the simple cyclic networks. Finally, the cluster networks illustrate the effects of decomposability on simulation performance.

Each of these networks was simulated for a variety of workloads, (e.g., routing probabilities, arrival rates, and service times) using six variations of a Chandy–Misra implementation: static node assignment with deadlock avoidance, static node assignment with deadlock recovery, dynamic node assignment with deadlock avoidance,

[4]We distinguish between the performance of the Chandy–Misra simulation and the performance measures for the simulated network. The former are the subject of this study.

dynamic node assignment with deadlock recovery, dynamic node assignment with waiting and deadlock avoidance, and dynamic node assignment with waiting and deadlock recovery. In all cases, we varied the number of processors from one to the number of nodes in the simulated network. Together, these simulations represent approximately two weeks of computation time on the Sequent Balance 21000. Figs. 3, 4, 7-10, 12, 13 and Tables II-IV, discussed below, show the results of a portion of these experiments. All such figures and tables show 95 percent confidence intervals about mean values.

Speedup, defined as

$$S_p = \frac{T_1}{T_p},$$

where T_1 and T_p are the respective execution times using one and p processors, is the performance metric used to compare all experimental results. All speedups are shown relative to a one processor distributed simulation using static assignment with deadlock recovery. In this case, all simulated nodes execute on one processor. Consequently, no synchronization is needed during queue insertion and deletion. Deadlocks *can* occur with one processor. This increases the value of T_1 and, consequently, increases the apparent speedup. Although it might seem preferable to use an event list oriented simulation as the point of reference, this would color the results with the idiosyncrasies of *two* implementations. For comparison, we conducted equivalent event list simulations on the Balance 21000 using SMPL, a portable simulation package. These results show that a single processor distributed simulation always executes more slowly than the equivalent sequential simulations. Thus, the speedups presented can be viewed as upper bounds on the speedup achievable with distributed simulation.

In addition to speedup, we also use deadlock recovery and null message fractions as performance measures. These are defined as

$$F_D = \frac{\text{number of deadlock recoveries}}{\text{number of message transmissions}}$$

and

$$F_N = \frac{\text{number of null message transmissions}}{\text{number of message transmissions}},$$

respectively. The deadlock recovery and null message fractions measure the amount of *useful* computation performed by each simulation.

Tandem Networks

Tandem networks are a feasibility test of distributed simulation; see Fig. 2. If distributed simulation cannot achieve good pipelined speedups for tandem networks, there is little prospect for success for networks containing cycles.

Fig. 3 shows the speedups for both deadlock avoidance and recovery when nodes are statically assigned to pro-

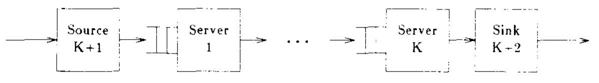


Fig. 2. Tandem queue.

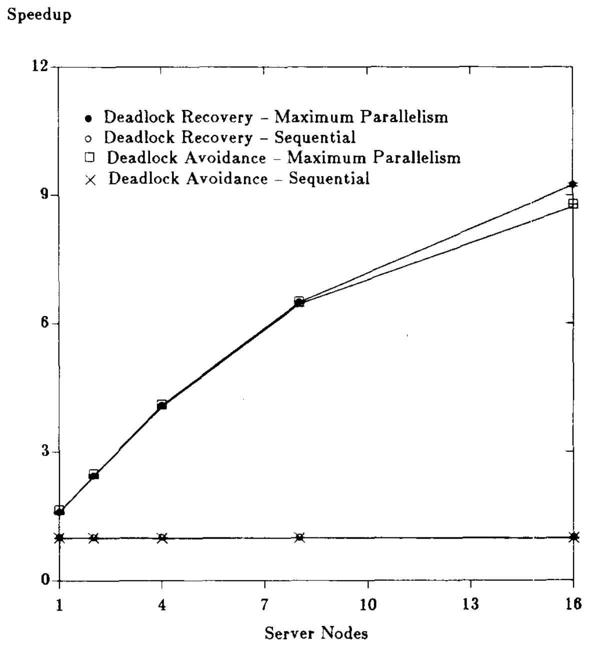


Fig. 3. Speedup for tandem queue (static node assignment).

cessors. Because there are no cycles, no deadlocks occur, and there is little distinction between deadlock recovery and avoidance. Recall that deadlock avoidance must continually verify that no null messages need be sent. Conversely, deadlock recovery does nothing until deadlock is detected. Thus deadlock avoidance incurs a small overhead even if no null messages need be sent. This difference is magnified as the number of nodes increases, leading to a small, but perceptible difference at 16 server nodes.

Fig. 3 shows a linear speedup for a small number of nodes, and a decrease in the slope of the speedup curve for additional nodes. This sublinear speedup for a larger number of nodes arises from memory and bus contention, as well as synchronization overhead.

By comparison, Fig. 4 shows speedups when deadlock recovery is used, and nodes are retrieved from a work queue.[5] Dynamic node assignment yields greater speedup than static assignment, but it too suffers from memory contention. Using half as many processors as nodes results in near linear speedup, albeit a smaller speedup than that obtained with maximal parallelism. This is the final confirmation of the effects of memory contention.

Waiting (i.e., placing nodes on the work queue only when they can profitably be evaluated), is ineffective be-

[5]In Fig. 4, "half parallelism" means that the number of processors used is equal to one half the number of network nodes.

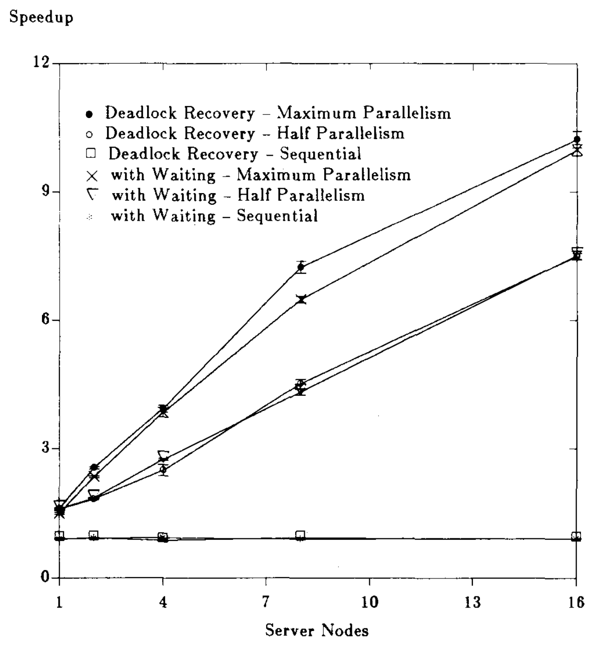


Fig. 4. Speedup for tandem queue with deadlock recovery (dynamic node assignment).

cause, in a tandem network, all nodes are always active. The additional overhead simply reduces the speedup, as shown when maximal parallelism is used.

The previous discussion, with one exception, assumed the number of processors equaled the number of nodes. When the number of nodes exceeds the number of processors, nodes must, with static assignment, be clustered onto processors. Table II shows the effects of this clustering for a tandem network containing 16 server nodes. For static assignment, speedup declines precipitously as the number of processors is reduced (e.g., reducing the number of processors from 18 to 12 reduces the speedup from approximately 9 to 4). In contrast, dynamic node assignment allocates processors to nodes based on their need for evaluation. When the number of nodes exceeds the number of processors, dynamic assignment is the method of choice.

Finally, we note that the sequential execution time, 113 seconds, compares favorably to the single processor distributed simulation. This suggests that the overhead for distributed simulation, other than that for deadlock avoidance or recovery, compares favorably to that for event-driven simulation.

General Feed-Forward Networks

Among the simplest generalizations of a tandem network are those containing forks and joins; see Fig. 5. Table III shows the corresponding speedups as a function of node clustering and deadlock technique.

Unlike the tandem networks, where deadlock avoidance and recovery are indistinguishable, feed-forward networks with forks necessarily distinguish between the two deadlock techniques. Because this network is open and contains no cycles, no deadlocks can occur, and deadlock detection detects none. In contrast, deadlock avoidance requires that null messages be sent at each fork node. This overhead is the reason for the difference in the performance of the two deadlock techniques.

Although speedups are *not* linear in the number processors, the fork and join nodes do not require as much processing time as the server nodes. Because of this, a linear speedup from a sequential simulation cannot be expected.

Cyclic Networks

The closed equivalent of a tandem network is the cyclic queue of Fig. 6. Unlike the tandem network, where the interarrival time at the source node does not affect the execution time of the simulation, the cyclic network depends on the simulated population. Fig. 7 shows the speedup obtained for a four node cyclic network; similar results also hold for larger cyclic networks. In contrast to the tandem networks, the cyclic network does not show linear speedup as a function of the number of processors.

Initially, one might suspect deadlock avoidance or recovery caused this decrease in performance. However, examination of the simulations shows that deadlock avoidance sent *no* null messages. Instead, messages circulate in large groups or *trains*; a node processes a train of messages and waits until they return on their next cycle. This suggests that fewer processors, dynamically assigned to the nodes, would achieve most of the potential speedup. Fig. 8 confirms this supposition: two, dynamically assigned processors, achieve nearly 80 percent of the speedup obtained with four processors. When dynamic assignment is augmented with waiting, the differenc grows even smaller.

The second important conclusion drawn from simulating cyclic queues is the extremely high cost for deadlock recovery. As noted above, deadlock avoidance sent no null messages. In contrast, deadlock recovery detected a small number of potential deadlocks. At population 40, the deadlock recovery fraction F_D was 0.0015. This corresponds to approximately 250 deadlock recoveries in 160 000 message transmissions. As discussed earlier, deadlock detection and recovery forces all processors to synchronize at a barrier before invoking the deadlock recovery algorithm. Execution profiling showed that the deadlock recovery routine and the barrier primitive comprised a negligible fraction of the simulation time. Synchronizing *is* expensive, but only because there is a significant interval between the arrival of the first processor at the barrier and the last. The parallelism declines as each processor reaches the barrier. Were the transition from computation to deadlock recovery abrupt, deadlock recovery would be inexpensive. As Fig. 8 shows, dynamic node assignment improves the performance of deadlock recovery, primarily because processors can more quickly rendezvous at the synchronization barrier. Finally, because fewer processors are actively evaluating nodes, dynamic node assignment with waiting further reduces the rendezvous delay.

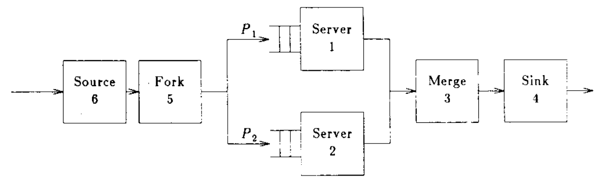


Fig. 5. Generalized feed forward network.

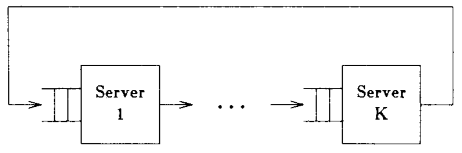


Fig. 6. Cyclic network.

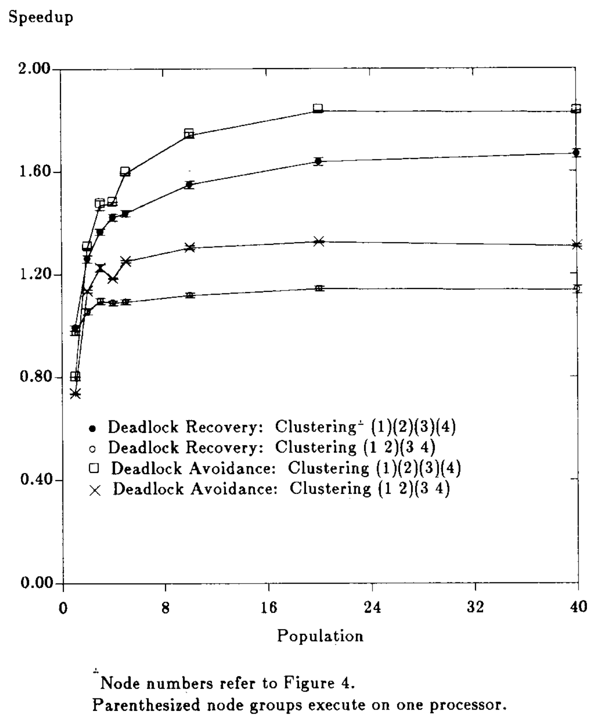


Fig. 7. Speedup for four node cyclic queue (static node allocation).

Central Server Networks

Central server networks have long been used as models of computer systems [1] and consequently have pragmatic importance. Because they contain nested cycles, central server networks are susceptible to deadlock in a distributed simulation. Hence, they are a more realistic test of distributed simulation. Figs. 9 and 10 show the speedup obtained for a central server network containing three servers; see Fig. 1 for the network topology. Even with five processors, the speedup barely exceeds unity. Moveover, this is using the single processor, static node assignment case as the basis for calculating speedup. As Table IV shows, *the parallel implementation rarely completes more quickly than the sequential implementation.* Indeed, static node assignment with deadlock avoidance runs 16 times more slowly than the sequential implementation. Consequently, the speedups over an event-driven simulation are much lower than Figs. 9 and 10 suggest.

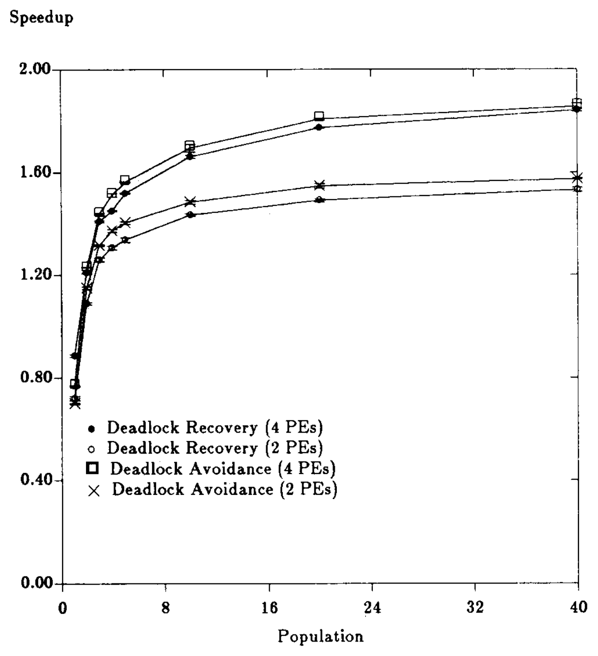


Fig. 8. Speedup for four node cyclic queue (dynamic node assignment).

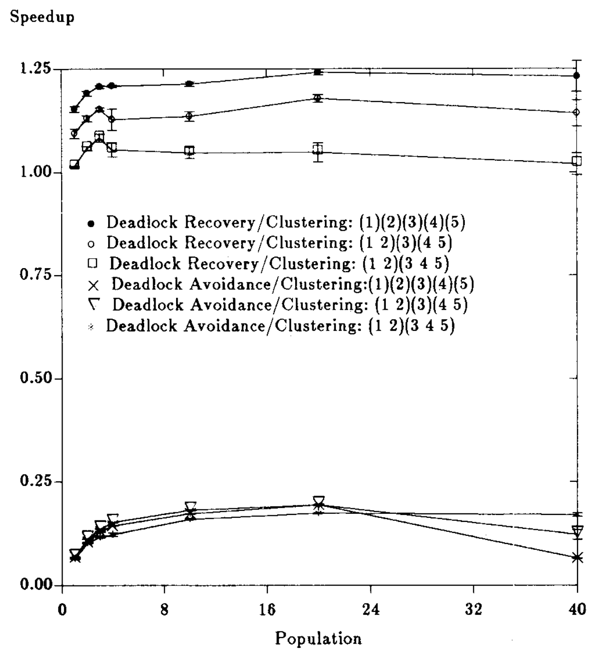


Fig. 9. Speedup for five node central server (static node assignment).

Unlike the simple cyclic network, where both deadlock avoidance and recovery were rare, the central server network frequently forces the simulation to either send null messages or attempt deadlock recovery. With only one

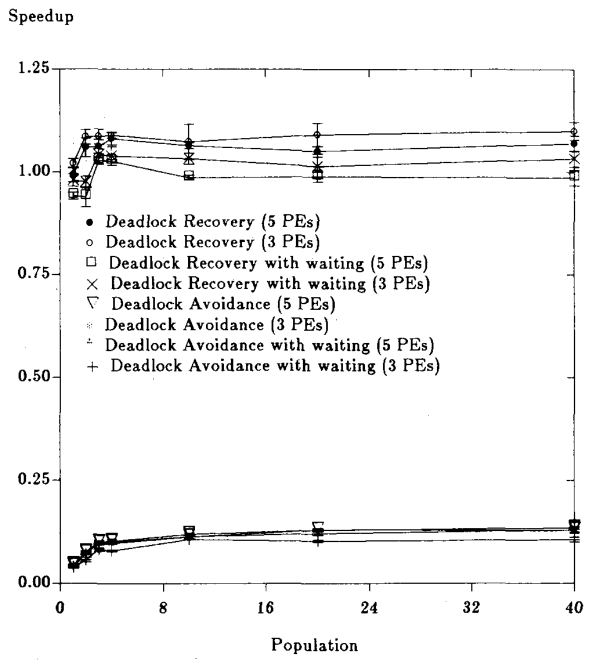


Fig. 10. Speedup for five node central server (dynamic node assignment).

TABLE IV
SEQUENTIAL AND PARALLEL MEAN EXECUTION TIME FOR FIVE NODE CENTRAL SERVER (TIME GIVEN IN SECONDS)

Population	SEQUENTIAL	STATIC PARALLEL		DYNAMIC PARALLEL			
		Recovery	*Avoidance*	*Recovery*	*Recovery w/ Waiting*	*Avoidance*	*Avoidance w/ Waiting*
1	26.32	28.97	491.85	33.62	35.47	569.00	662.30
2	42.80	44.67	510.71	50.19	56.70	619.56	655.15
3	51.44	52.73	490.48	59.89	61.95	599.29	632.47
4	56.96	56.92	477.92	63.64	67.02	580.51	623.12
10	67.22	67.20	471.20	76.66	82.69	601.95	602.09
20	74.42	70.58	450.91	83.47	88.70	628.46	620.91
40	87.78	74.74	1419.22	86.08	93.38	602.21	595.28

Parameter	Value
Routing Probability	(1) 0.10, (4) 0.45, (5) 0.45
Clustering case (5 PEs)	(1) (2) (3) (4) (5)[a]

[a]Node numbers refer to Figure 1. Parenthesized node groups execute on one processor.

circulating message, nearly fifty null messages are transmitted for each movement of the real message. Although the null fraction decreases as the number of circulating messages increases, it converges to approximately twenty null messages per real message transmission.[6] In contrast, the deadlock recovery fraction converges to 0.35. Although these deadlock recoveries are expensive, as the analysis of cyclic networks showed, their number is so small compared to the number of null messages sent during deadlock avoidance that deadlock recovery is significantly faster.

Finally, we must emphasize that these results are *significantly* more negative than earlier simulated results [22]. A sequential simulation of a network, but its nature, imposes some sequential ordering on the evaluation of network nodes. When those nodes are not being evaluated, they do not generate null messages, nor can they deadlock. In contrast, in a fully parallel implementation,

[6]This value, twenty, seems independent of the network routing probabilities. Removing the nested cycle from Fig. 1 neither increases the observed speedup nor decreases the null fraction. We hypothesize that the value is a function of the relative speeds of the processors and memory.

all nodes are always active. Thus, they continue to receive and generate null messages while awaiting receipt of real messages. Thus, the overhead is *higher* than suggested by a sequential simulation of distributed simulation.

Cluster Networks

Cluster networks were the most complex simulated during our experimental study. As Fig. 11 shows, a cluster network is composed of several tightly clustered subnetworks. This has two important ramifications. First, the network is nearly decomposed and should yield significant speedups with parallel simulation. Secondly, the clustering increases the expected execution time of the simulation. Why? The clocks of all nodes must reach the terminating value before the simulation completes. With only a few circulating messages, some nodes may be idle for long periods of time. Only when a message "escapes" from a subcluster will the clocks of other nodes advance. This asynchrony means that the clocks of some nodes may run far past the terminating simulation value.

Fig. 12 shows the speedup of the cluster network with static node assignment. For small populations, deadlock recovery is significantly faster than deadlock avoidance. As with the central server network, this is attributable to the large number of null messages sent. As the population increases, the null fraction decreases precipitously, and deadlock avoidance becomes the method of choice. Interestingly, the speedup obtained with deadlock recovery is relatively insensitive to the simulated population. With a large number of processors, the delay to synchronize at a barrier is prohibitive; this overhead is the reason for the poor performance of deadlock recovery.

When nodes are dynamically assigned to processors, Fig. 13, the performance of both deadlock avoidance and deadlock recovery increase significantly. As noted earlier, messages are often "trapped" in network subclusters and many nodes are often idle. With deadlock avoidance and static node assignment, many nodes continually generate null messages. These messages simply cause additional overhead and memory contention. With dynamic assignment, a node must migrate from the tail to the head of the node work queue before being evaluated. This additional delay between evaluations reduces the number of null messages and is the source of the additional speedup with dynamic node assignment.

Fig. 13 also shows that a smaller number of processors yields a marginally larger speedup than that obtained with maximal parallelism. Although the difference is not statistically significant in this case, the difference becomes large when node waiting is introduced. The reason is, as before, the presence of many idle nodes. By suspending nodes that cannot productively contribute to the simulation, contention for the node work queue is reduced, and those nodes with work can proceed without interference.

In summary, the cluster network shows that distributed simulation can produce significant speedups *if* the network is decoupled, and subclusters interact infrequently.

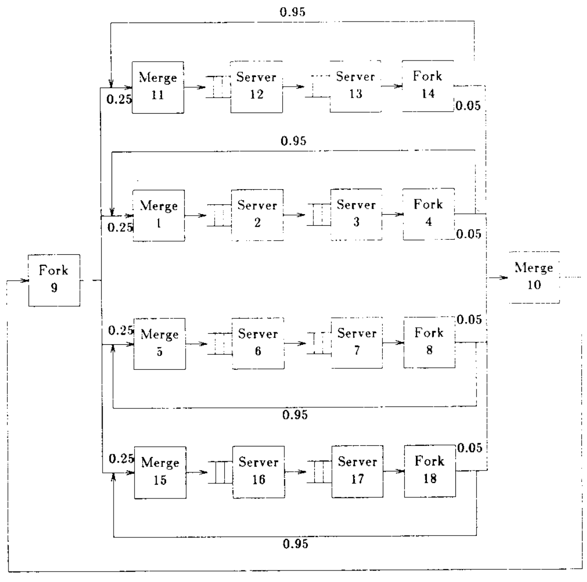
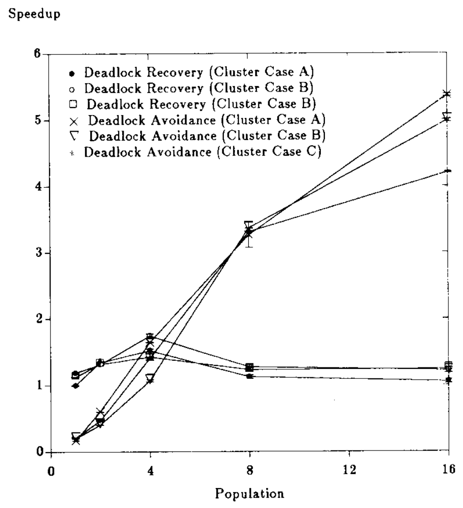


Fig. 11. Cluster network.



Parameter	Value
Node Service Time	1.0
Confidence Level	95%
Speedup Base	One processor,† deadlock recovery
Cluster case A	(1) (2) ... (18)
Cluster case B	(1 4) (2 3) (5 8) (6 7) (9 10) ...
Cluster case C	(1 2 3 4) (5 6 7 8) (9 10) ...

†Node numbers refer to Figure 5.
Parenthesized node groups execute on one processor.

Fig. 12. Speedup for four block cluster (static node assignment).

Summary

Distributed simulation has been the subject of several *simulated* performance studies; little or no experimental data have heretofore been available. Obtaining such data was the primary goal of this work. Using queueing networks as the simulation application, we simulated a variety of such networks with varying workloads using several variations of the Chandy–Misra algorithm on a shared memory machine.

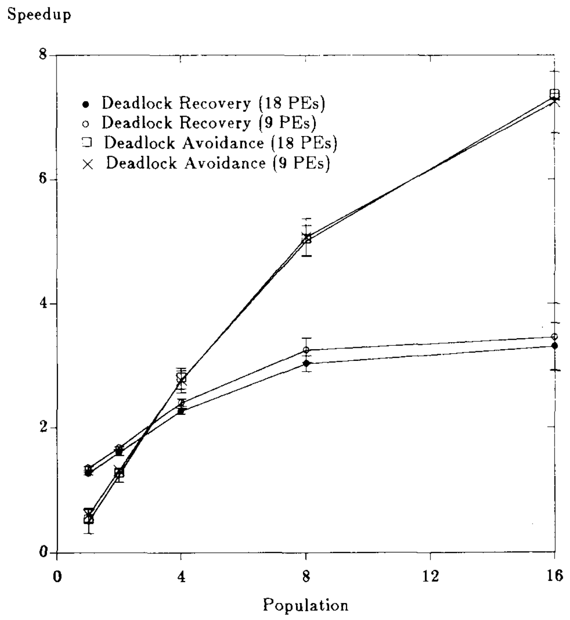


Fig. 13. Speedup for four block cluster (dynamic node assignment).

These experiments, based on an implementation described in [25], suggest that the Chandy-Misra distributed simulation technique may have limited performance when applied to queueing network models. There are three primary reasons for this. First, a single processor implementation of the Chandy-Misra algorithm is usually slower than the equivalent sequential, event-driven simulation. Hence, multiple processors are often needed to recoup the loss due to inefficiency. Second, networks with cycles require deadlock avoidance or recovery techniques. Third, the inability to lookahead [13], in the general case, limits parallelism.

Because queueing network simulation requires little processing by server nodes, nodes interact frequently in real time. Because of this, queueing networks are a stress test for distributed simulation. In simulations that require extensive computation between node interactions, distributed simulation is analogous to a group of decoupled processes. In such cases, distributed simulation should prove more attractive.

In conclusion, we caution that performance results are extremely sensitive to underlying assumptions and implementation techniques. Our implementation, based on [25], assumes limited lookahead [13] during deadlock avoidance and does not rely on knowledge of the underlying queueing network domain. We chose this approach because it is general and applicable to the widest variety of problems. However, recent results show that careful tuning of the implementation underlying the distributed simulation, based on knowledge of the queueing network domain, sometimes can result in significant performance improvements [13], albeit with consequent loss of generality. We believe the results presented in this paper typify the performance that would result when simulating systems whose structure and complexity preclude obtaining sufficient knowledge to tune the underlying parallel simulation mechanism. Additional implementation and experience are necessary, however, to determine the interdependence of generality and performance.

Acknowledgment

J. Dongarra and the Advanced Computing Research Facility of Argonne National Laboratory graciously provided both advice and access to the Sequent Balance 21000.

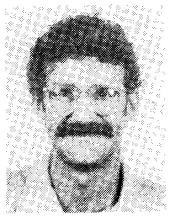
References

[1] J. P. Buzen, "Computational algorithms for closed queueing networks with exponential servers," *Commun. ACM*, vol. 16, no. 9, pp. 527-531, Sept. 1973.
[2] K. M. Chandy, V. Holmes, and J. Misra, "Distributed simulation of networks," *Comput. Networks*, vol. 3, no. 1, pp. 105-113, Feb. 1979.
[3] K.M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Trans. Software Eng.*, vol. SE-5,no. 5, pp. 440-452, Sept. 1979.
[4] ——, "Asynchronous distributed simulation via a sequence of parallel computations," *Commun. ACM.*, vol. 24, no. 4, pp. 198-206, Apr. 1981.
[5] K.M. Chandy, L. M. Haas, and J. Misra, "Distributed deadlock detection," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 144-156, May 1983.
[6] A. Chandak and J. C. Browne, "Vectorization of discrete event simulation," in *Proc. 1983 Int. Conf. Parallel Processing*, Aug. 1983, pp. 359-361.
[7] W. W. Chu *et al.*, "Task allocation in distributed data processing," *Computer*, vol. 13, no. 11, pp. 57-69, Nov. 1980.
[8] J. C. Comfort, "The design of a multi-microprocessor based simulation computer—I," in *Proc. Fifteenth Annu. Simulation Symp.*, Mar. 1982, pp. 45-53.
[9] J. C. Comfort, "The simulation of a master-slave event set processor," *Simulation*, vol. 42, pp. 117-124.
[10] R. A. Finkel, *An Operating Systems Vade Mecum.* Englewood Cliffs, NJ: Prentice-Hall, 1986.

[11] M. A. Franklin, D. F. Wann, and K. F. Wong, "Parallel machines and algorithms for discrete-event simulation," in *Proc. 1984 Int. Conf. Parallel Processing*, Aug. 1984, pp. 449–458.
[12] D. Franta and W. Maly, "An efficient data structure for the simulation event set," *Commun. ACM*, vol. 20, no. 8, pp. 596-602, Aug. 1977.
[13] R. M. Fujimoto, "Performance measurements of distributed simulation strategies," in *Distributed Simulation, 1988, Proc. SCS Multiconf. Distributed Simulation*, Feb. 3–5, 1988, pp. 14–20.
[14] P. Heidelberger, "Statistical analysis of parallel simulations," in *Proc. 1986 Winter Simulation Conf.*, to be published.
[15] D. Jefferson and H. Sowizral, "Fast concurrent simulation using the time warp mechanism," in *Distributed Simulation 1985, The 1985 Society for Computer Simulation Multiconf.*, San Diego, CA.
[16] M. B. Konsek, D. A. Reed, and W. Watcharawittayakul, "Context switching with multiple register windows: A RISC performance study," in preparation.
[17] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *Computer*, vol. 15, no. 9, pp. 8–21, Sept. 1982.
[18] D. A. Patterson, "Reduced instruction set computers," *Commun. ACM*, vol. 28, no. 1, pp. 8–21, Jan. 1985.
[19] J. K. Peacock, J. W. Wong, and E. G. Manning, "Distributed simulation using a network of processors," *Comput. Networks*, vol. 3, no. 1, pp. 44–56, Feb. 1979.
[20] G. F. Pfister, "The Yorktown Simulation Engine: Introduction," in *Proc. ACM IEEE 19th Design Automation Conf.*, June 1982, pp. 51–54.
[21] D. A. Reed, "A simulation study of multimicrocomputer networks," in *Proc. 1983 Int. Conf. Parallel Processing*, Aug. 1983, pp. 161–163.
[22] ——, "Parallel discrete event simulation: A case study" (Invited Paper), *Rec. Proc. 19th Annu. Simulation Symp.*, Mar. 1985, pp. 95–107.
[23] D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message Based Parallel Processing*. Cambridge, MA: M.I.T. Press, Nov. 1987.
[24] C. H. Sauer, E. A. MacNair, and S. Salza, "A language for extended queueing networks," *IBM J. Res. Develop.*, vol. 24, no. 6, pp. 747–755, Nov. 1980.
[25] M. Seethalakshmi, "Performance analysis of distributed simulation," M. S. thesis, Dep. Comput. Sci., Univ. Texas, Austin, 1978.
[26] C. L. Seitz, "The cosmic cube," *Commun. ACM*, vol. 28, no. 1, pp. 22–23, Jan. 1985.
[27] P. F. Wyman, "Improved event scanning mechanisms for discrete event simulation," *Commun. ACM*, vol. 18, no. 4, pp. 221-230, Apr. 1975.

Daniel A. Reed (S'80–M'82) received the B.S. (summa cum laude) degree in computer science from the University of Missouri, Rolla, in 1978 and the M.S. and Ph.D. degrees, also in computer science, from Purdue University, West Lafayette, IN, in 1983.

He is currently an Assistant Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign, where he was the recipient of an NSF Presidential Young Investigator Award. He also holds a joint appointment with the Center for Supercomputing Research and Development, where he is participating in the design of Faust, a programming environment for the Cedar shared memory multiprocessor. His current research interests include design of parallel algorithms and software for message-based parallel processors. He is currently directing the design of Picasso, a hypercube operating system. He is the author (with R. M. Fujimoto) of *Multicomputer Networks: Message-Based Parallel Processing* published by MIT Press.



Allen D. Malony received the B.S. and M.S. degrees in computer science from the University of California, Los Angeles, in 1980 and 1982, respectively.

From 1981 to 1985 he was a member of the Technical Staff at Hewlett Packard Laboratories in Palo Alto, CA. Presently, he is a Software Engineer with the Cedar project at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. He is also a Ph.D. student in the Department of Computer Science there. His research interests include parallel computation, multiprocessor architecture, and performance evaluation.

Bradley D. McCredie (S'87) was born in Cocoa Beach, FL, on February 27, 1964. He received the B.S. degree with highest honors in computer engineering from the University of Illinois at Urbana-Champaign in 1986.

He is currently a member of the Quantum Electronics Research Laboratory at the University of Illinois, where he is investigating models of high-speed optoelectronic semiconductor devices and high-speed interconnects.