Proceedings of the 1992 Winter Simulation Conference
ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson

MIMD PARALLEL SIMULATION OF CIRCUIT-SWITCHED COMMUNICATION NETWORKS

David Nicol

Dept. of Computer Science
College of William & Mary
PO Box 8795
Williamsburg, VA 23187-8795

Albert Greenberg
Boris Lubachevsky

AT&T Bell Labs
600 Mountain Ave
Murray Hill, NJ 07974

ABSTRACT

This paper describes techniques for the parallelized simulation of circuit-switched communication networks. We discuss implementations based both on conservative synchronization, and optimistic synchronization. Performance results are presented from experiments on 256 processors of the Intel Touchstone Delta, wherein we observe call simulation rates of nearly eight million calls per minute on a realistic network.

1 INTRODUCTION

We are interested in using parallel computers to simulate realistic models of large circuit-switched networks. A motivating example is the the AT&T long distance telephone network, composed of nearly 120 switching centers distributed throughout the US, with fiber-optic links between centers. A link $\{u, v\}$ is a connection between two centers u and v, and is comprised of a number of *trunks*, each of which can carry one call. The link capacities vary by as much as three orders of magnitude throughout the network. A call is placed between switching centers by identifying a path between them, and allocating one trunk on each link of the path. A trunk is allocated for the entire duration of the call. The network (say with N nodes) is nearly fully connected (there are $\approx N(N-1)/2$ links), so that most calls can be directly routed over a single link. However, an arriving call may find that its direct connection is loaded to capacity, in which case an alternative two-link route (called a *via*) is sought. Should either link on the alternative path also be saturated, the call is *blocked*.

In our simulations we use a via selection policy known as ALBA, for Aggregated Least Busy Alternative (Mitra *et al.*, 1991). This policy is a close approximation of that used in the AT&T network. Under the ALBA-K policy, the state of a link $\{u, v\}$ is encoded as in integer $S(u, v)$ between 0 and K. Thresholds $r_0, r_2, \ldots r_K$ specific to the trunk govern the state assignment. If the link is using c trunks, the ALBA state is the smallest index j such that $c \leq r_j$. For a given link, the smaller the state, the more free trunks are available. If a call arrives and cannot be placed on its direct link $\{u, v\}$, a node w is sought that minimizes $\max\{S(u, w), S(w, v)\} < K$. A link in state K may not accept a rerouted call, as $n - r_{K-1}$ trunks are reserved for direct calls ($n = r_K$ being the capacity of the link). A call that cannot be placed directly or find an acceptable via is said to be *blocked*.

Simulation plays a critical role in the design and analysis of such networks. Network engineers frequently use simulation to predict network performance under stress conditions such as severed links, and overloads. The simulation events are individual call arrivals and departures. Production simulations of a network the size of AT&T's involve simulating call-by-call activity over the course of at least twenty million calls. Owing to the size of the simulation, serial simulations require a dedicated workstation with a large memory (128 Mbytes) if they are to avoid memory thrashing. Without such a memory, a serial simulation of twenty million calls on an ordinary workstation takes many hours. It is clear that parallelization has much to offer this application, both in terms of exploiting the larger memories of parallel machines, and execution speed.

In this paper we describe two parallel techniques used to simulate large networks similar to AT&T's. Our methods are suitable for MIMD multiprocessors, and are implemented on the Intel Touchstone Delta (Lillevik, 1991) multiprocessor, using up to 256 processors. On a network similar to AT&T's, our algorithms achieve a simulation execution rate of over 7.9 million calls per minute, reducing the time required for a production run to a small number of minutes.

Previous treatments of this simulation problem include an application of synchronous relaxation (Eick

et al., 1991), and a "sweep" algorithm (Gaujal *et al.*, 1992). Both methods are geared primarily for SIMD architectures. The present treatment is specifically MIMD; furthermore, the call simulation rates we report are larger than those reported before.

2 A CONSERVATIVE METHOD

The first algorithm we describe is *conservative*, being based on the notion of *appointments* (Nicol and Reynolds, 1984, Nicol, 1988). To appreciate the synchronization problem we face, imagine a simulation decomposed naturally by assigning links to processors. Problems arise when a call arrives on a saturated link $\{u, v\}$, say at simulation time t. Under ALBA, the processor responsible for that call needs to find a via whose cost *at simulation time* t is least, and then request that the two associated links—say $\{u, w\}$ and $\{w, v\}$—carry the call. Under a conservative philosophy, we ought not permit a processor to simulate past simulation time t if there is any possibility of later receiving simulation work for time $s < t$. In particular, the processors responsible for $\{u, w\}$ and $\{w, v\}$ should not simulate them past time t.

If processors responsible for $\{u, w\}$ and $\{w, v\}$ could be forewarned of the via request, they could simply synchronize (at time t) with the processor responsible for $\{u, v\}$. The principle difficulties with such an approach are that

- Link $\{u, v\}$ must simulate up to the time of the call arrival at t before it can determine that the call cannot be directly placed.
- Once $\{u, v\}$ realizes a via is needed, the selection of the via route is dependent on instantaneous state information from all links of the form $\{u, w\}$ or $\{w, v\}$.

The most difficult of these issues is that of via selection based on instantaneous data. Our solution sacrifices a small amount of model fidelity in order to finesse the problem: via decisions are based on slightly old state information. This is justified, as a link's state changes slowly, and experiments have shown the insensitivity of the resulting statistics to slightly stale state information.

Our method is synchronous, and window-based. Supposing all processors have simulated up to and synchronized at time s (initially $s = 0$), they cooperatively define a simulation time t, simulate all workload in $[s, t)$, and globally synchronize again at t. Time t is carefully chosen so that no call arriving in $[s, t)$ can also be completed in $[s, t)$. Window construction is not difficult, for call arrival and duration times are randomly sampled from distributions that are independent of the state of the system. One issue is whether enough events can be found in the window to make this approach viable. In (Gaujal *et al.*, 1992) we have shown that if the aggregate call arrival rate to the network is λ, and if the call duration is comprised of a constant C ($0 \leq C \leq 1$) plus an exponential with mean $1 - C$, then on average $\lambda C + \sqrt{\lambda(C-1)\pi/2} + O(1/\sqrt{\lambda})$ calls are processed in each window. We have observed that $C \approx 0.025$ is large enough to ensure good performance on networks modeled after AT&T's, owing to tremendously large aggregate arrival rates.

Link state information is globally disseminated as part of the synchronization at time s, as follows. Each processor initially clears a table containing $N(N-1)/2$ link state codes. Then, for each of its assigned links i (in global link coordinates) it records the ALBA code for link i, in position i of the table. The processors then perform a global vector OR reduction (supported on Intel multiprocessors with the `gior()` system call), which has the effect of providing every processor with a copy of every link's state code. This state information is assumed to be constant over the interval $[s, t)$. Upon receiving the completed table, a processor categorizes every possible via for every one of its assigned links $\{u, v\}$. To select a via for $\{u, v\}$ the processor is able to select uniformly at random from the set of $\{u, v\}$ vias with lowest ALBA classification.

Given window $[s, t)$, each processor builds a linearly linked event list for each of its assigned links. A link's event list spans $[s, t)$. Then, for every call arrival event, a via for that event is chosen, *even though we don't yet know whether the via is needed.* Messages reporting the via selection are sent to the processors holding the via links. Upon receipt of such, additional via arrival events are merged into the link event lists. For the purposes of simulation, these events play the role of *appointments*—a processor will not simulate past the time of a via arrival event until it knows whether the via is needed. By preselecting vias we establish the synchronization points needed by a conservative approach.

Following these steps, every link's event list contains arrival events, via arrival events, and call completion events. The completions are associated with calls accepted in a previous window, and so are unconditional events. It is not yet known which of the via arrival events might actually be needed, nor is it yet known which of the new call arrivals can be accepted. Simulation will provide those answers.

Consider the processing of a link's event list. Call departure events are simple—increment the link's number of free trunks. To process a call arrival we see

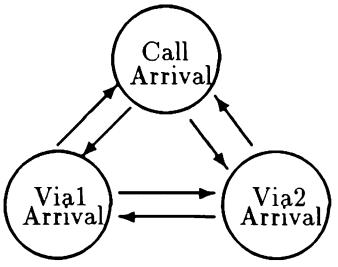

Figure 1: Three-Way Synchronization For a Call Arrival Event.

if any free trunks are available on the associated link, and send a message (either *yes* or *no*) to the links carrying the preselected via arrivals. A *yes* message is interpreted to mean that the call can be routed directly, *no* means it cannot, and so is viewed as a request to carry the call. Similarly, upon reaching a via arrival event, the processor sends a *yes* (a via can be placed here) or *no* message to both the other via link, and to the link originating the call.

For every call arrival there are three events, in three different links' event lists. We need to effect a three-way synchronization between these events, as illustrated in Figure 1. The action a processor takes upon reaching an event depends on the *event state*, a 3-tuple (x, y, z) where each component is either *yes*, *no*, or ? describing respectively the processor's knowledge of the original link's ability to accept the call, and the two vias' abilities to accept the call.

A brute force means of synchronizing three links at an event is to have each event dispatch a state message to the others, and then wait for them to send their own states. Once a link knows the states of all three events involved, it can determine for itself whether the call is routed directly, routed using the via, or is blocked. This approach clearly has high overhead, as it involves the exchange of six messages for every single call arrival. Such a voluminous exchange is not always necessary, for any one of the links can preempt the possibility of the via call being accepted—e.g., the original call may be acceptable, or one of the via links is unable to accept the call, if requested. Consequently, if a processor scans an event and can discern from the event state that the via has already been preempted, there is no reason for it to send or receive further information.

The table below gives the actions associated with a call arrival event, with optimizations to reduce message passing. x signifies "don't care"; CLP means "Continue Link Processing", i.e., advance to the next event in the link's list. Finally, messages are sent only the first time the event encountered; a link returning from suspension doesn't need to resend its state (even if message transmission is indicated by the table).

Table 1: Event Actions for Call Arrival Event

Event State $(call, via1, via2)$	Action
$(n, ?, ?)$	Send "n" msg. Suspend link.
(n, n, x)	Record call as blocked. CLP.
(n, x, n)	Record call as blocked. CLP.
$(n, ?, y)$	Send "n" msg. Suspend link.
$(n, y, ?)$	Send "n" msg. Suspend link.
(n, y, y)	Send "n" msg. Record call as rerouted. CLP.
$(y, ?, ?)$	Send "y" msg. Accept call. CLP.
(y, y, y)	Send "y" msg. Accept call. CLP.
(y, n, x)	CLP.
(y, x, n)	CLP.

In Table 1, to *accept* a call is to decrement the number of free trunks on its link, record it as accepted, and to schedule the call's completion. To *suspend* a link is to remove it from a list of active links. At the point of suspension, the event upon which the link suspends is recorded. The link rejoins the active list when the missing message(s) for that event are received. These messages alter the state of the event, which is then processed according to Table 1. For example, if $\{u, v\}$ suspends at some event e in state $(n, ?, ?)$, the link is reactivated only by a message that changes e's state to $(n, n, ?)$, $(n, ?, n)$, or (n, y, y) (our message handler does not reinstate the link in intermediate states $(n, ?, y)$ or $(n, y, ?)$).

Corresponding tables describe processing for via arrival events. Table 2 below is tailored for *via1*, (arbitrarily) the via link with lowest index.

Table 2: Event Actions for Via Arrival Event

Event State $(call, via1, via2)$	Action
$(?, n, ?)$	Send "n" msg. CLP.
(x, x, n)	CLP.
(y, x, x)	CLP.
$(n, y, ?)$	Send "y" msg. Suspend link.
(n, y, y)	Accept via call. Send "y" msg. CLP.

Much of the extra work performed by the conservative method supports the pre-generation and handling of via arrival events that are never used. This presents a prime opportunity to employ optimism, in order to avoid generating via arrival events until actually needed. This issue is taken up in the next section.

3 AN OPTIMISTIC METHOD

It is relatively straightforward to add optimism to our conservative method. Furthermore, within its basic framework we can minimize traditionally expensive overheads such as state-saving and rollback. As we will see, distinct performance advantages are gained by using optimism.

Windows are defined exactly as before, event-lists for the window are constructed, and ALBA state information is exchanged exactly as before. One key difference between the conservative and optimistic methods is that we do not pre-generate via arrival events for every call arrival. Another difference is that the optimistic method will retain a link's window event list throughout the processing of the window, because an event can be scanned any number of times, as dictated by rollback activity. However, by construction, the list itself changes little over the course of processing a window (the only changes being the insertion of via arrivals) a pointer to the current position in the list is maintained.

We associate an event state (x, y) with every call arrival event. The first component $x \in \{y, n\}$ signifies whether a via request *has ever* been generated for the arrival; the second component $y \in \{y, n, ?\}$ signifies whether the call arrival was accepted or rejected the last time it was scanned. The initial state of every call arrival event is $(n, ?)$.

Table 3 below specifies the state-dependent behavior for a call arrival event, when it is determined that the call cannot be placed given the current link state.

Table 3 : Event Processing for an Non-acceptable Call Arrival

Event State $(sent, call)$	Action	New State
$(n, ?)$	Select vias. Send **MakeVia** msgs. CLP.	(y, n)
(n, y)	Select vias. Send **MakeVia** msgs. CLP.	(y, n)
(y, y)	Send **NeedVia** msgs. CLP.	(y, n)
(y, n)	CLP.	(y, n)

Table 3 illustrates that upon the first recognition that a via is needed on link $\{u, v\}$ (i.e., when $sent = n$) a via node w is selected and special **MakeVia** messages are sent to links $\{u, w\}$ and $\{w, v\}$. (Upon receipt, via arrival messages are inserted into the via link's event list. Rollback occurs if the link has already simulated past the insertion point.) The event is found in state (y, y) if the call arrival to $\{u, v\}$ already selected vias once, and if on the previous scan it appeared that they weren't needed. Since they appear to be needed again, $\{u, v\}$ sends **NeedVia** messages to both via links. If the event is found in state (y, n) there is no need to notify the vias, as the last acceptability transition will have done that.

Table 4 gives the appropriate event behavior when the call arrival **can** be accepted.

Table 4: Event Processing for an Acceptable Call Arrival

Event State $(sent, call)$	Action	New State
$(n, ?)$	Use 1 trunk. CLP.	(n, y)
(n, y)	Use 1 trunk. CLP.	(n, y)
(y, y)	Use 1 trunk. CLP.	(y, y)
(y, n)	Use 1 trunk. Send **RemoveVia** msgs. CLP.	(y, y)

Table 4 shows that messages need be sent only to to cancel a previous via request.

Now consider the handling of via arrival events, which are inserted only by receipt of **MakeVia** messages. Tables 3 and 4 show that the originating call arrival event may later toggle its via request (by sending **NeedVia** and **RemoveVia** messages) as its own apparent state shifts between rejecting and accepting the call. Clearly then, one component of a via's arrival event state must indicate whether the call can be carried directly, or not. This component is updated immediately upon the receipt of any **MakeVia**, **NeedVia**, or **RemoveVia** message. Next, since the via arrival event may be scanned several times, a second component records whether on the last scan where the via appeared to be needed, it was determined that the via link could carry the call. Finally, if the via is needed, then the link needs to know whether the other via link can carry the call. A third component records present knowledge of that link's ability to carry the call. Thus a via arrival's event state is a 3-tuple, with each component being y or n.

Table 5 gives the event actions when the via arrival cannot be accepted. The table is given with respect to *via*1. Any messages generated are sent only to

*via*2. Observe that a message is sent to *via*2 even if we presently believe that *via*2 is unable to carry the call. This ensures that whenever the original link needs a via, both via links have an up-to-date copy of the other link's state.

Table 5 : Event Processing for an Non-acceptable Via Arrival

Event State $(call, prev, via2)$	Action	New State
(y, x, x)	CLP.	(y, x, x)
(n, y, x)	Send "n" msg. CLP.	(n, n, x)
(n, n, x)	CLP.	(n, n, x)

Again we see that the only communication occurs to notify the other link via that the state has changed.

Table 6 gives the event actions when the via arrival at link *via*1 can be accepted. All messages are sent only to *via*1. Here we see that if the via is needed and can be carried locally, the via link optimistically assumes that the other via can also carry it, unless it has information to the contrary.

Table 6 : Event Processing for an Acceptable Via Arrival

Event State $(call, prev, via2)$	Action	New State
(y, x, x)	CLP.	(y, x, x)
(n, n, n)	Send "y" msg. CLP.	(n, y, n)
(n, n, y)	Send "y" msg. Use 1 trunk. CLP.	(n, y, y)
$(n, n, ?)$	Send "y" msg. Use 1 trunk. CLP.	(n, y, y)
(n, y, n)	CLP.	(n, y, n)
(n, y, y)	Use 1 trunk. CLP.	(n, y, y)

Only two pieces of data state need to be saved to support rollback. After we scan an event we save two integers : the current link capacity, and the state of the link's random number stream. State-saving costs are thus negligible. Now a link is rolled back if it receives a message at a time less than that of the event at the list's current position pointer. After the pointer is moved back, the link's state is restored using state saved in the event just prior to the new current position. Lazy message cancellation (Reiher *et al.*, 1990) is already built into our state transition tables.

When implementing our optimistic code we encountered a fundamental problem. We desire that processors synchronize globally at the upper edge of the window, yet a processor cannot be sure that it won't be rolled back after entering barrier synchronization logic. We have addressed this problem elsewhere (Nicol, 1992). The solution we propose allows processors to be rolled back out of the barrier logic; at the same time, it ensures that no processor leaves the barrier until all processors have received all messages destined for them prior to synchronization. Following such a synchronization, a processor knows that the current states of all events in all links are final. At this point each processor goes through the lists, accepting call and via arrivals as indicated, and releasing the space used to store events. It should be noted that a link does not know whether a call it cannot accept itself was accepted by the vias. For the purposes of statistics, we have the via links record that the call was carried, or blocked. In a final step at termination we are able to reconstruct each link's blocking statistics.

Scheduling is always a critical issue for optimistic simulations. We experimented with a scheduler that allocates a quanta of simulation time to each link, with a quanta allocated to the link whose next event time is least. A link that exhausts its quanta but has remaining workload is returned to the priority heap. Since the cost of probing for possible messages is relatively high on Intel multiprocessors, our intuition was that as the quanta grows (from allowing one event per link to simulating a link all the way to the window's edge), different overheads would dominate. Each quanta slice involves priority heap management overheads, as well as message probing overheads. A system using tiny quanta receives messages almost as soon as they arrive, and is careful to execute local events in nearly monotone increasing order (between rollbacks). A system using larger quanta reduces these overheads, but increases the risk of erroneous computation. As we see in the next section, the balance turned in favor of large quanta.

4 EXPERIMENTS

Both the conservative and optimistic methods have been implemented the Intel Touchstone Delta (Lillevik, 1991), an architecture based on the Intel i860 CPU (Intel Corporation, 1990). The system has 560 processors, connected in a mesh. Our test problem is based on measured network capabilities and demand following the accidental severing of several links at a switching center on the east coast. As a result, the network suffers significant blocking on several hundred links.

All these experiments use an ALBA-4 via selection policy. The link state tables are updated every

unit of simulation time. The experiments also use a small call duration constant portion of $C = 0.025$. The link-to-processor mapping is the same in both methods, being based on the longest processing time first list scheduling heuristic analyzed in (Graham, 1969) (we use link capacities to estimate "processing time"). This heuristic makes no attempt to optimize communication patterns, but does balance the workload quite effectively.

Table 7 below compares the performance of the two methods on various numbers of processors. Each experiment measures the average number of calls simulated per minute, on simulation runs of approximately ten million calls. For the optimistic version we also present the percentage of "extra" events simulated due to rollbacks. For these experiments, the quanta used by the optimistic version's scheduler is the entire window width.

Table 7: Call Processing Rates on Intel Touchstone Delta

#	Conservative	Optimistic	
procs	Calls/Min.	Calls/Min.	% extra events
16	810,300	1,410,300	14%
32	1,447,080	2,433,780	15%
64	2,339,400	3,959,760	16%
128	2,700,480	5,351,640	17%
256	2,496,180	7,920,840	17%

Consider first the conservative performance. While a peak rate of over 2.7M calls/minute is observed, we also see that performance improvement drops off sharply for more than 64 processors. While the 64 processor run achieves a rate nearly three times faster than the 16 processor run, the 128 processor run is only 1.15 times faster than the 64 processor run; the 256 processor run actually runs slower than the 128 processor run. This occurs because the relative (and increasing) cost of frequent synchronization eventually dominates performance.

The optimistic version fares much better in general, especially at high processor counts. Even though the relative fraction of time a processor spends in synchronization related activities must also increase with the processor count, that fraction is lower than that of the conservative method. However, further improvements in performance are hindered by load balancing considerations; in our 256 processor runs several processors are assigned only one "fat" link. So long as we constrain ourselves to assigning at most one processor to a link, the addition of more processors will not reduce the total finishing time. We believe this might be ameliorated using an optimistic version of our sweep algorithm (Gaujal *et al.*, 1992), which supports splitting a single link's workload across processors.

In another set of experiments (on 32 processors) we examined the sensitivity of the optimistic version's performance to the scheduler quanta. Somewhat to our surprise, we discovered that it is less expensive to suffer the re-execution costs associated with large quanta, than it is to suffer the overhead costs of small quanta. Table 8 below presents measured call simulation rates, as a function of quanta q. Quanta units are "fraction of window width"; e.g., $q = 0.01$ means the quanta is 1/100 of the width of the synchronization window. The data clearly indicates that in this simulation it is cheaper to re-execute events than it is to probe for messages and manage priority heaps.

Table 8: Sensitivity to Scheduling Quanta

Quanta	Calls/Min	% extra events
0.001	1,737,720	7%
0.01	1,873,080	7%
0.1	2,045,820	8%
0.2	2,197,680	9%
0.5	2,222,000	12%
1.0	2,433,780	15%

Table 7's peak measured performance is nearly 8M calls/minute, a rate that will simulate a production run of twenty million calls in approximately two and a half minutes. How good is this? The memory of one Delta processor is too small to contain an optimized serial implementation, so that true speedup can't be measured. However, we have measured an optimized version's performance on a smaller network, and found it to be 0.2M calls/minute. Taking this as an upper bound, we observe a performance acceleration of nearly 40, using 256 processors. While a "speedup" of 40 in 256 seems modest, one should keep in mind that the hypothetical serial version requires an enormous memory—at least 128 Mbytes—to contain the simulation. Without such a large memory, we estimate the serial running time would be on the order of ten hours for twenty million calls (a figure arrived at by measuring serial execution time of a twenty million call run on a Sparc, then accounting for the speed differential between the workstation and single Delta node). Our methods ought to be viewed as a way of exploiting multiprocessor's larger memory capacity. Secondly, one ought to bear in mind that our simulation is doing more work than does an optimized serial implementation. An important advantage our optimistic version has over the conservative one is that generation and processing of via events is greatly reduced, an advantage clearly borne out in the performance numbers. Even so, the optimistic version generates messages, computes the link

state table, and may execute events more than once. None of these activities are mirrored in the serial version; they are simply the price one pays to exploit parallelism.

As a final note we remark that we have observed entirely similar call processing rates on the more commonly available Intel iPSC/860 multiprocessors. We report measurements on Touchstone Delta primarily to show simulation rates on large numbers of processors.

5 SUMMARY

This paper describes two techniques for the parallelized simulation of large, nearly fully connected circuit-switched networks. Notable characteristics of our methods are (i) they are window based, (ii) they map network links to processors, and (iii) on a network similar to AT&T's long-distance phone network they achieve a call simulation rate of nearly eight million calls per minute, using 256 processors of the Intel Touchstone Delta. Our methods offer the promise of reducing production simulation execution times from several hours, to several minutes. The first characteristic is the key to our success. By choosing small enough windows, we are able to craft very efficient inter-link synchronization mechanisms used while processing the window. While small windows might be faulted for allowing in too few events, networks of the size of AT&T's are so large that thousands of calls are simulated within each window.

Two issues invite yet further study. Our methods only approximate the actual network's use of instantaneous state information when making a via selection. We believe we can incorporate instantaneous state changes into our optimistic version, by the inclusion of state-change broadcasts. The severity of the performance penalty one pays for this accuracy is an open question. A second issue is to examine a simulation based on the seemingly more natural method of mapping network *nodes* to processors, rather than links. It is quite possible that a node-mapped simulation will better handle instantaneous link state changes, as well as being more "self-balancing". We intend to pursue these issues in the near future.

ACKNOWLEDGEMENTS

The contribution of David Nicol was supported in part by NASA grants NAG-1-1060 and NAG-1-995, NSF grants ASC 8819373 and CCR-9201195. This work was initiated during a supported visit by him to AT&T Bell Laboratories. The authors are also grateful for the support of Sandia National Labs in providing access to the Intel Touchstone Delta at Caltech.

REFERENCES

Eick *et al.*, 1991. Synchronous relaxation for parallel simulations with applications to circuit-switched networks. In *Proceedings of the 1991 Workshop on Parallel and Distributed Simulation*, pages 151–162.

Gaujal *et al.*, 1992. A sweep algorithm for massively parallel simulation of circuit-switched networks. ICASE Report 92-30 Available from ICASE, MS 132C, NASA Langley Research Center, Hampton, VA 23668.

Graham, 1969. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math*, 17(2):416–419.

Intel Corporation, 1990. *i860 64-bit microprocessor programmer's reference manual.* Intel Literature Sales, P.O. Box 7641, Mt. Prospect, IL 60056.

Lillevik, 1991. The Touchstone 30 gigaflop DELTA prototype. In *Distributed Memory Computer Conference 91*, pages 671–677. IEEEPRESS.

Mitra *et al.*, 1991. Analysis and optimal design of Aggregated-Least-Busy-Alternative Routing on symmetric loss networks with trunk reservations. circuit-switched networks. In *13th International Teletraffic Congress*, North Holland, Copenhagen, Denmark.

Nicol and Reynolds, 1984. Problem oriented protocol design. In *Proceedings of the 1984 Winter Simulation Conference*, pages 471–474, Dallas.

Nicol, 1988. Parallel discrete-event simulation of FCFS stochastic queueing networks. *SIGPLAN Notices*, 23(9):124–137.

Nicol, 1992. Optimistic barrier synchronization. ICASE Report 92-34. Available from ICASE, MS 132C, NASA Langley Research Center, Hampton, VA 23668.

Reiher *et al.*, 1990. Cancellation strategies in optimistic execution systems. In *Distributed Simulation 1990*, pages 112–121. Society for Computer Simulation.

AUTHOR BIOGRAPHIES

DAVID M. NICOL received a Ph.D. in Computer Science from the University of Virginia in 1985, and is presently an Associate Professor in the Department of Computer Science, at the College of William and Mary, Williamsburg, Virginia. He is an associate editor for the ACM's *Transactions on Modeling and Computer Simulation* and for the *ORSA Journal on Computing*, and has served as the 1989 Program Chairman and the 1990 General Chairman of the Workshop on Parallel and Distributed Simulation (PADS). His interests are in parallel simulation, performance analysis, and algorithms for mapping parallel workload.

ALBERT G. GREENBERG received a Ph.D. in Computer Science from the University of Washington in 1983. Since 1983 he has been a Member of the Technical Staff at AT&T Bell Laboratories, in Murray Hill, New Jersey.

BORIS D. LUBACHEVSKY is a Member of the Technical Staff at AT&T Bell Laboratories, in Murray Hill, New Jersey. He received the candidate degree (equivalent Ph.D.) in Computer Science in 1977 from Tomsk Polytechnical Institute (USSR). From 1980 to 1984 he was a member of the Ultracomputer project team in New York University. His research interests include parallel programming and simulation techniques.