ORSA Journal on Computing

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Feature Article—Parallel Discrete Event Simulation: Will the Field Survive?

Richard M. Fujimoto,

To cite this article:
Richard M. Fujimoto, (1993) Feature Article—Parallel Discrete Event Simulation: Will the Field Survive?. ORSA Journal on Computing 5(3):213-230. https://doi.org/10.1287/ijoc.5.3.213

ORSA Journal on Computing
Vol. 5, No. 3, Summer 1993

0899-1499/93/0502-0213 $01.25

FEATURE ARTICLE

Parallel Discrete Event Simulation: Will the Field Survive?

RICHARD M. FUJIMOTO / *College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280;*
Email: fujimoto@cc.gatech.edu

(Received: September 1992; revised November 1992, March 1993; accepted: April 1993)

Despite over a decade and a half of research and several successes, technologies to use parallel computers to speed up the execution of discrete event simulation programs have *not* had a significant impact in the general simulation community. Unless new inroads are made in reducing the effort and expertise required to develop efficient parallel simulation models, the field will continue to have limited application, and will remain a specialized technique used by only a handful of researchers. The future success, or failure, of the parallel discrete event simulation field hinges on the extent to which this problem can be addressed. Moreover, failure to meet this challenge will ultimately limit the effectiveness of discrete event simulation, in general, as a tool for analyzing and understanding large-scale systems. Basic underlying principles and techniques that are used in parallel discrete event simulation are briefly reviewed. Taking a retrospective look at the field, several successes and failures in utilizing this technology are discussed. It is noted that past research has not paid adequate attention to the problem of developing simulation models for efficient parallel execution, highlighting the need for future research to pay more attention to this problem. A variety of approaches to make parallel discrete event simulation an effective tool are discussed.

As long as computers have been used to model large, complex systems, simulationists have been devising models that require far more computing power than could be provided by the fastest machines of the day. Indeed, every simulationist is well aware just how easy it is to construct a program that will take an inordinate amount of time to execute. Today, simulation applications abound that require days or weeks of CPU time on the fastest available uniprocessor machines.

While the speed of sequential computers continues to increase every year, these increments have been far outstripped by our ability to find new applications that utilize all of the CPU cycles that are available, and then some. This can partly be attributed to the increasing complexity of the world in which we live. The systems we envision, and desire to simulate, are always one step bigger, grander, and more sophisticated than the systems that are currently in place. New simulation models become increasingly more complex, and experiments with existing models become more elaborate. Just as demands for memory continue to grow at a seemingly unending rate, I believe the same will remain true with respect to computing power for the foreseeable future.

For example, simulation is considered a vital tool in the computer design community to validate and optimize computer hardware designs. But, it is also extremely time consuming. Soulé[65] reports that final verification of a supercomputer developed by the Ardent Computer Company required two weeks using two workstations. As technology advances, the problem becomes only worse. The number of logic circuits that can be placed on a single chip approximately *doubles* every two years.[34] Today, simulations of massively parallel multiprocessors are required that contain hundreds or thousands of CPUs, and even larger systems are just around the corner. The computational requirements for such large-scale simulations are staggering.

Over the last decade, improvements in computing speed have taken on a new twist: the most significant improvements in performance now come from widespread exploitation of parallelism, and only secondarily from faster circuit technologies. Increasingly, the "fastest machines of the day" are highly parallel multiprocessor systems containing thousands of high performance microprocessors. Machines such as Thinking Machine's CM-5 and Kendall Square Research's KSR-1 are recent examples of this trend. This phenomenon has a broad impact on all areas utilizing high performance computing, e.g., optimization, expert systems, and simulation, among others, highlighting the growing importance of parallel algorithms and software design.

The emergence of parallel computation has an especially important impact in the simulation community. Execution

Subject classifications: Simulation: efficiency, languages. Computers software.
Other key words Parallel computation.

of discrete event simulation programs on a parallel computer (referred to as *parallel discrete event simulation* or *PDES*) is no trivial task. Even though the system being simulated often contains much intrinsic parallelism, translating this into concurrent execution of the simulator has proven to be challenging.

Of course, parallel discrete event simulation is not a new field, with seminal work dating back nearly 15 years. Hundreds of papers have since been published, the majority appearing in the last five years. Substantial speedups have been reported across a wide variety of applications: 1900 for molecular spin models,[47] 57 for queueing network simulations,[25] and 24 for combat models,[75] to mention a few. For digital logic circuits, Briner[12] reports speedups as high as 25 using 32 processors of a BBN Butterfly. Now that a substantial body of knowledge has developed, it is appropriate to step back and examine what has been learned, and more importantly, where future research in the field should be directed. This is the question that I wish to pursue here. In particular, I would argue that it is time to shift some of the emphases in parallel discrete event simulation research to new directions.

To be specific, it is my contention that despite several successes, parallel discrete event simulation has not significantly affected the general simulation community, and will not do so until new inroads are made in making the technology more accessible to simulation practitioners. In particular, much research is required to aid in the development of simulation models that are suitable for efficient concurrent execution. Until this problem is addressed, parallel simulation will continue to have little impact in solving real-world problems.

Throughout this article, I am concerned with the problem of speeding up the execution of a *single* discrete event simulation program by decomposing the model into units of computation that can be executed concurrently. I am especially concerned with event-driven simulations where the simulation model changes state at discrete, usually irregular, points in simulated time, as opposed to continuous time simulations. To be sure, there are other approaches to using multiprocessors to speed up execution, e.g., replicating independent simulation runs on different processors to explore various parameter settings or reduce variance, and parallel timestepped simulators that advance simulated time one clock tick at a time. These methods have their place, and should be used if the situation warrants it. However, my concern is with large simulation problems that cannot satisfactorily be solved using these techniques alone. Throughout the remainder of this article, I use the term *parallel simulation* to refer to techniques that execute a single discrete event simulation program on a parallel or distributed computer.

1. Is Parallel Simulation a Viable Technology?

There is ample evidence, both empirical and analytic, that suggests that parallel simulation is indeed a viable technology. Later, I will provide a survey of results produced by the field. In this section, I will focus attention on one specific application area: telecommunication networks.

For many years, simulation has been heavily used in the telecommunication industry for network design and management. It is often the tool of choice when the complexity of the network makes mathematical analysis intractable, or when studying transient conditions that are difficult to express mathematically. The computational requirements of network simulators increase linearly with the number of events that must be processed which, in turn, increases with traffic density, network size, and complexity. It comes as no surprise that the computation requirements of network simulators continue to increase each year.

Telecommunication network technology has undergone substantial changes in recent years because of two trends: (1) the availability of high bandwidth connections, due to advances in electrical and optical technologies, and (2) the increased need for users to transmit a variety of different types of traffic, e.g., voice, data, video, and fax images. So-called *broadband integrated services digital networks* (*B-ISDN*) are being developed to support these changes, with *asynchronous transfer mode* (*ATM*) switches leading the way.

Components used in such a telecommunication system include *statistical multiplexers* that combine multiple incoming streams of traffic (e.g., phone connections from individual subscribers) into a single output stream, and switches that route traffic to selected destinations (see Figure 1a). Each multiplexer includes one or more finite capacity buffers to hold traffic that cannot be immediately transmitted. The switches may logically be viewed as being configured in a fully connected interconnection (the physical topology is usually different, however), as shown in Figure 1b, and are connected via trunk lines. A typical routing strategy might first attempt to utilize a direct connection to the destination switch. If that fails (because the trunk line cannot accept any additional connections), the switches will utilize some routing policy in an attempt to form a connection through an intermediate via switch. Should that fail, the network may try to build a more complex path using two or more intermediate switches, or it may simply return a busy signal to the caller, depending on the strategy that is in place.

Simulations of these networks are often extremely time consuming. For example, target data loss rates in modern telecommunication networks may be on the order of 10^{-9} or lower, meaning one in every 10^9 cells (the unit of data transmitted in these networks) is lost.[63, 73] Cell losses typically occur because of overflows in finite capacity buffers such as those depicted in Figure 1a. Approximately 10^{11} cell arrivals must be simulated to achieve adequate confidence intervals for cell loss statistics, so a simulator for only a single multiplexer may require 10^{11} or more events. Such a computation will require more than a day of CPU time on existing machines, even if each event computation requires only a microsecond. Simulations of practical interest may include dozens of multiplexers and switches, further increasing computational requirements. Such simulations are often driven by traces of actual measured network traffic loads.

A number of issues must be addressed before parallel computation can be applied to this problem. The model

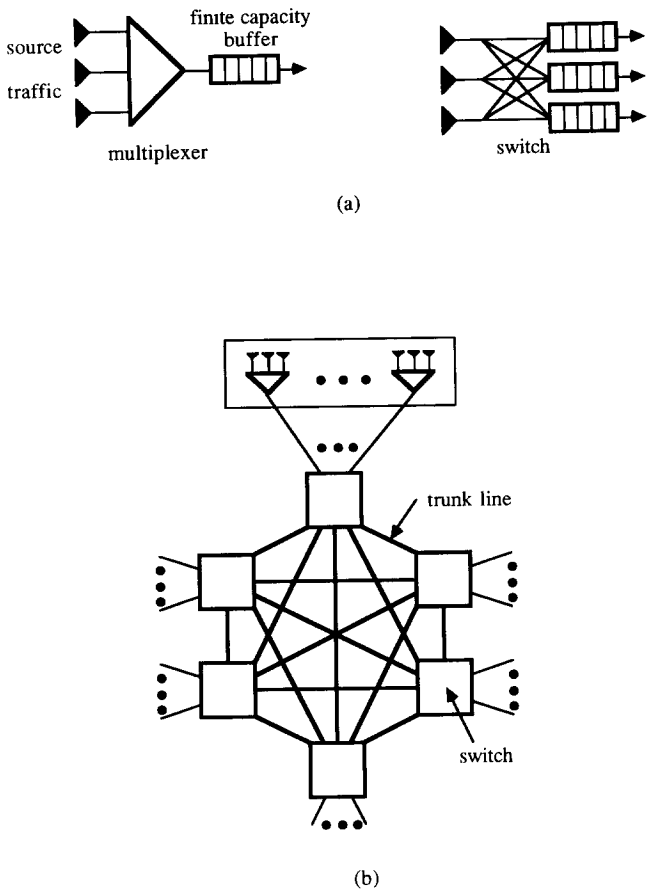


Figure 1. Telecommunication networks. (a) Components include multiplexers that combine traffic from multiple sources into a single stream, and switches that route traffic to appropriate destinations. (b) Simplified diagram indicating the logical connections of a switching system.

must be decomposed into suitable components, and the component simulators must be distributed across the parallel computer to balance the workload. Interactions among programs executing on different processors must not be too frequent, or too much time will be spent passing information rather than executing simulator computations. A suitable approach to synchronizing the parallel simulator must be devised (more will be said about this later). These tasks are not straightforward, and a significant amount of time must usually be devoted to tune the model to achieve acceptable performance.

Despite these obstacles, a handful of researchers have obtained significant speedups using parallel simulation techniques. Earnshaw and Hind report that up to an order of magnitude speedup was obtained in simulating B-ISDN network.[20] Gaujal, Greenberg, and Nicol[31] also report an order of magnitude speedup on their parallel simulator, and indicate that simulator performance exceeds 3,000,000 calls per minute. Lin describes an algorithm for simulating a finite capacity queue (a central component of the aforementioned switches) that yields speedup which is linear in the number of processors.[43] Mouftah and Sturgeon[54] report that a parallel simulation executing on two workstations provides a modest speedup (up to 20%) for certain test cases over a single processor simulation. Turner and Xu[72] also report success in speeding up telephone switching network simulations, though specific speedup figures are not provided (performance relative to an alternative parallel simulation algorithm is reported).

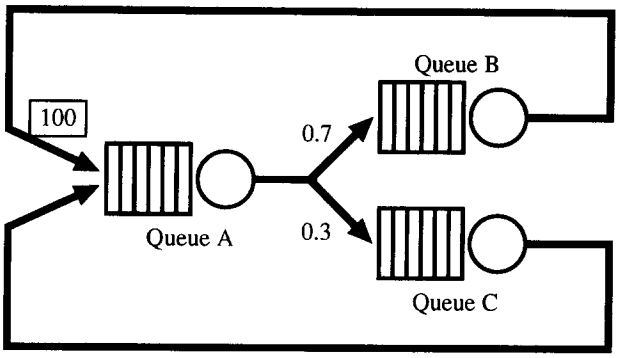
I report these results to provide evidence that parallel simulation merits serious consideration by researchers and practitioners. This is not to say that there are no serious hurdles to be overcome. Indeed, in much of the remainder of this article, I argue that the field has a long way to go before parallel simulation is widely accepted as a useful technology. Before arguing this point, however, I will briefly review the basic synchronization problem that has drawn much of the attention of the parallel simulation community in recent years, and survey results spanning the last 15 years.

2. Review of the Synchronization Problem

Synchronization is nontrivial, and is one aspect of PDES that distinguishes it from other forms of parallel computation. I will use the simulation of the closed queueing network depicted in Figure 2a to illustrate the key concepts. This "central-server" network contains three queues and some fixed number of jobs circulating indefinitely among them. Assume that the jobs are processed in each queue using a first-come-first-serve discipline, and completed jobs are routed along the outgoing links according to the probabilities assigned to each arc (if no such value is indicated, it is assumed to be 1.0). Thus, a job leaving the "central server" in this network (queue A) is routed to queue B with probability 0.7, or to queue C with probability 0.3.

The simulator of this network can be defined as a collection of sequential simulators, one for each queue. Each of the sequential simulators is referred to as as a *logical process* (*LP*). Let LP_A, LP_B, and LP_C denote the logical processes that simulate queues A, B, and C, respectively. The computation performed by each logical process is a sequence of event computations. Each event denotes some "interesting" activity in the simulated system, and contains a timestamp that indicates when that event occurs. In this example, there are two types of events in the simulation of each queue: a job *arrival* event denoting the arrival of a new job at the queue, and a job *departure* event denoting the fact that a job has completed service, and is being forwarded to another queue. There will be one arrival and one departure event for each job as it passes through a queue.

As shown in Figure 2b, the computation associated with an *arrival* event at (say) LP_B is to perform one of two actions: (1) if the server is busy, the job is added to the queue of waiting jobs, or (2) if the server is idle, it is marked as busy (by setting the `QBusy` variable in Figure 2b to TRUE), and LP_B schedules a departure event for itself to denote the job's subsequent departure. Later, when the departure event at LP_B is processed, a new arrival event will be scheduled at LP_A. While LP_B and LP_C always



(a)

```
State Variables for LP.
  QBusy: BOOLEAN,    /* TRUE if server is busy, else FALSE */
  QLength: INTEGER;  /* number of jobs waiting for service */
/* QBusy is initialized to FALSE, QLength is initialized to 0 */


Arrival Event @ time T
if (QBusy) then
  QLength  = QLength + 1, /* add job to queue */
else  /* server is idle */
  QBusy := TRUE; /* server becomes busy */
  schedule departure event for self  @ T+ServiceTime
end-if


Departure Event @  time T
schedule arrival event for LP A @ T,
if (QLength > 0) then
  QLength := QLength - 1, /* new job begins service */
  schedule departure event for LP B @ T+ServiceTime
else  /* no jobs waiting for service */
  QBusy := FALSE;  /* server becomes idle */
end-if
```

(b)

Figure 2. Parallel simulation example. (a) Central server queueing network. (b) Simulation code for LP_B for processing arrival and departure events.

schedule this event for LP_A, LP_A must select either LP_B or LP_C by flipping a biased coin. After scheduling the arrival event, the server in queue B will either begin servicing the next job, or become idle.

In general, a PDES program consists of a collection of logical processes that interact by exchanging timestamped event messages. Each *LP* simply processes incoming messages. Incoming messages that have not yet been processed are stored in a "pending event" list data structure for the process.

Now comes the hard part: *each LP must process events in nondecreasing timestamp order*. This requirement is sometimes called the *local causality constraint*. Failing to adhere to this constraint could cause events in the future to affect events in the past, leading to an incorrect simulation. It is the synchronization mechanism's responsibility to ensure that this does not happen, i.e., it must ensure that each *LP* processes events in timestamp order.

To understand the ramifications of this constraint, consider the following situation (see Figure 2a): LP_A has an unprocessed arrival event from LP_B with timestamp 100, but no unprocessed messages from LP_C. Further, suppose the last time LP_A received a message from LP_C, that message contained timestamp 50, and that messages are transmitted from one process to another in timestamp order. If LP_A were to process the timestamp 100 event, it might later receive a message from LP_C with a smaller timestamp, thereby violating the local causality constraint. Therefore, a reasonable strategy might be for LP_A to wait until it receives an event from LP_C. However, suppose no such message is forthcoming! This would happen if there were only one job circulating in the network, or if all of the jobs were last routed through LP_B. In this case, the simulator is *deadlocked* because LP_A is waiting to receive a message from LP_C, and LP_C is waiting for a message from LP_A. Uncertainty as to when an *LP* can process incoming event messages is at the heart of the synchronization problem.

Broadly speaking, there are two general approaches to attacking the synchronization problem (see Reynolds[60] for a more detailed taxonomy). *Conservative* approaches *avoid* the possibility of executing events out of timestamp order by using a protocol to determine when it is "safe" to process each event. *Optimistic* approaches allow errors (out of order execution) to occur, but provide a mechanism to *detect* and *erase* them.

Seminal work in conservative algorithms appeared in the late 1970s and early 1980s, with the "null message" algorithm developed independently by Chandy and Misra,[14] and Bryant.[13] In this algorithm, each process sends a "null message" with timestamp T_N to its neighboring processes after processing each event. The null message is a "promise" that the process will not later send a message with timestamp smaller than T. In the example depicted in Figure 2, suppose LP_A's last departure event contained timestamp 80, and the minimum service time of any job is 15 units of simulated time. LP_A could then send a null message to LP_C (as well as LP_B) with timestamp 95, because consecutive departures must be at least 15 time units apart. Assume the server in queue C is idle, and no jobs are waiting for service, as would be the case if the system were deadlocked. Upon receiving this null message, LP_C could then send a null message to LP_A with timestamp 110, assuming LP_C's minimum service time is also 15. This allows LP_A to process the arrival event containing timestamp 100, thereby eliminating the deadlock. One can show that the null message approach *avoids* deadlock situations so long as there is no cycle such that a message could traverse the cycle with zero timestamp increment. This approach is thus sometimes referred to as the *deadlock avoidance* or *null message* algorithm.

Chandy and Misra[15] later proposed an algorithm based on deadlock detection and recovery. Here, no null messages are used, but instead, the system is allowed to deadlock. Separate mechanisms are then used to (1) detect the deadlock, and (2) recover from it. The deadlock can be broken by processing the smallest timestamped event, which is guaranteed to be "safe" (i.e., can be executed without leading to a violation in the local causality constraint).

Seminal work in optimistic methods is due to Jefferson and Sowizral,[37] whose Time Warp mechanism first appeared in 1982. Here, processes are allowed to process events, risking later violations of the local causality constraint. Should such a violation occur, the process is rolled back, and the events are reprocessed in timestamp order. The rollback may necessitate "unsending" one or more previously sent messages. An elegant mechanism called anti-messages is provided for this purpose.

An anti-message is a copy of a previously sent message that differs from the original in only a sign bit field. Whenever an anti-message and its matching (positive) message are both stored in the same queue, the two annihilate each other, leaving no trace behind. To "unsend" a previously sent message, a process need only send the corresponding anti-message (see Figure 3a). Upon receipt of an anti-message, a process will check to see if the corresponding (positive) message has already been processed. If it has not, the message and anti-message pair are simply annihilated (Figure 3b). On the other hand, if it has been processed (see Figure 3c), the process is first rolled back to a point in time before the message was processed, effectively making it an unprocessed message (this rollback may, in turn, cause additional anti-messages to be sent), and then annihilation takes place. In systems where messages are not necessarily delivered in the order that they were sent, the positive message may not have been received when the anti-message arrives. In this case, the anti-message is buffered in the input queue of the process, and annihilation takes place when the positive copy is received. Recursively applying this "roll back and send anti-message" cycle will eventually erase all effects of the "unsent" message.

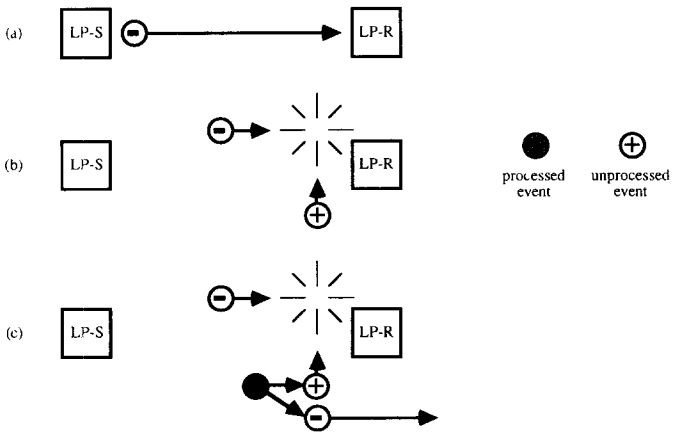


Figure 3. The anti-message mechanism. (a) "Unsending" a previously sent message by sending the corresponding anti-message. (b) Case when receiving an anti-message and the corresponding positive message has not yet been processed. The message and anti-message pair are annihilated. (c) Case when receiving an anti-message, and the corresponding positive message has been processed. The process is rolled back, causing the processed message to become marked as unprocessed and (in this example) an anti-message to be generated. The message and anti-message pair are then annihilated.

Numerous other approaches have been proposed for attacking the synchronization problem. Several surveys have appeared that describe and compare different approaches.[27, 28, 52, 62]

Before leaving this section, one point is worth mentioning because it will resurface later. Most PDES algorithms, including Time Warp and the conservative algorithms described earlier, assume that *all* interactions between logical processes occur through timestamped event messages. In particular, it is mandated that processes do not access common (shared) state variables. For example, in Figure 2, suppose the routing of jobs by queue A were dependent on the state of the other queues, e.g., one might decide to route messages to the queue with the fewest jobs waiting for service. In a sequential simulator, this could be programmed by simply having LP_A examine the `QLength` variables (see Figure 2b) in LP_B and LP_C, and scheduling the arrival event at the process with the shorter queue. This is not allowed in the parallel simulator because the processes are usually at different points in simulated time. LP_A may wish to know the length of the queue at time 100, but the current value of the `QLength` variable in LP_B may correspond to that at simulated time 80, because LP_B has only advanced that far through the simulation. Thus, some other mechanism must be defined to extract this information. This is commonly accomplished by *query* event messages that do nothing but read a state variable, and return its contents to the process that initiated the query. Query events return the proper value because they are treated as ordinary events by the simulation protocol, and are thus processed in timestamp order along with the other events.

3. Where Do We Stand Today?

Much of the early work in the field was focused on solving the aforementioned synchronization problem. Even today, nearly 15 years after the initial algorithms were proposed, protocol research continues to be the focus of much of the work in the field with new proposals appearing every year (e.g., see [56, 72]). These protocols vary from optimizations of existing schemes, e.g., to reduce the number of null messages in Chandy/Misra/Bryant's original proposal[18] or rollbacks in Time Warp,[64, 72] to new approaches, e.g., relying on periodic global resynchronizations (for example, see Lubachevsky[47]).

Hand-in-hand with protocol development, another body of research concerns itself with evaluating the performance of parallel simulation mechanisms. For example, an analytic model for Time Warp based on a renewal process appeared as early as 1983.[41] A large body of knowledge utilizing empirical and analytic methods has evolved to characterize the behavior of both conservative and optimistic protocols. Although there is clearly still more work to be done, much more is now known concerning the performance of parallel simulation mechanisms than was known only a few years ago. A few, selected results regarding protocols and performance evaluations of parallel simulation algorithms are listed in Table I. Other "hot" topics include memory management, load balancing, and hard-

Table I. Parallel Simulation Performance: Selected Results

Year	Result
1977	Bryant[13] proposes null message algorithm; Chandy/Misra[14] algorithm appears shortly thereafter.
1982	Jefferson and Sowizral[37] invent Time Warp.
1983	Lavenberg, Muntz, and Samadi[41] develop first analytic model for Time Warp (based on a renewal process). An approximate model assuming two processors was developed. Mitra and Mitrani[53] and Felderman and Kleinrock[23] later develop exact models for two-processor Time Warp.
1987	Reed, Malony, and McCredie[59] report some of the first experimental measurements of parallel simulation protocols on a multiprocessor system. They report discouraging results using the null message and deadlock detection and recovery algorithms for queueing network simulations.
1988	Fujimoto[24] demonstrates that the performance problems Reed, Malony, and McCredie observed can be overcome for first-come-first-serve queues by exploiting "lookahead" information, but notes that these techniques are not so easily applied to other types of queues.
1988	Window based protocols begin to appear. Lubachevsky[46] proposes a conservative protocol, and later shows that performance scales as the problem size and the size of the multiprocessor are increased in proportion. Sokol, Briscoe, and Wieland[64] propose adding time windows to Time Warp. Numerous other proposals for hybrid protocols are developed in subsequent years.
1989	Researchers at JPL[21, 35, 58, 75] report that significant speedups were obtained using Time Warp for simulations of combat models and other applications. This is perhaps the first experimental evidence demonstrating that Time Warp can achieve good performance. Soon after, Fujimoto[25] reports speedup as high as 57 on 64 processors using Time Warp for queueing network simulations.
1990	Madisetti, Walrand, and Messerschmitt[49] develop a simple two-processor Markov model for Time Warp, and suggest extensions to multiple processors. Later, Gupta, Akyildiz, and Fujimoto[33] develop a validated Markov model for N-processor Time Warp with homogeneous workloads, and Dickens and Reynolds[19] develop a model for window based protocols. These are among the first analytic models of Time Warp performance for the case of more than two processors.
1990	Lin and Lazowska[44] show that Time Warp yields optimal performance if incorrect computation (computation that will later be rolled back) never rolls back correct computation, assuming zero overheads. Lipton and Mizell[45] show that one can construct simulations where Time Warp outperforms Chandy/Misra by arbitrary amounts, but the opposite is not true; Chandy/Misra can only outperform Time Warp by a constant factor, assuming constant rollback overhead.
1990	Lin and Lazowska,[44] Nicol,[55] and Felderman and Kleinrock[22] develop models to compare Time Warp performance with conservative protocols.
1990	Jefferson[36] invents the Cancelback protocol to allow Time Warp to execute using no more memory than that required by the sequential execution. Akyildiz et al.[3] later develop and validate an analytic model for this protocol, showing that a modest amount of memory is required to achieve efficient execution.
1990	Lubachevsky, Shwartz, and Weiss[48] develop conditions under which Time Warp will become unstable.

ware support, as well as more classical simulation topics such as random number generation, and statistical analysis of results, applied in the context of parallel computation. See Fujimoto and Nicol[28] for a recent survey.

Can the field claim success? A few selected speedup results are shown in Table II. Here, speedup is defined as the execution time of a sequential implementation of the simulation program divided by the execution time of the parallel version. The table indicates the speedup, the number of processors, the hardware platform, and the synchronization mechanism used in the study.

If success is measured in speedup, then parallel simulation research appears to be largely successful. No doubt, this has contributed to the increasing interest in parallel simulation over the last few years. Although a few negative results have appeared,[59, 61, 66] as will be elaborated upon in a moment, many published results report one to three orders of magnitude speedup across numerous applications, ranging from combat models to telecommunication networks to vehicular traffic. Speedups are reported on both SIMD (single-instruction stream, multiple-data stream) and MIMD (multiple-instruction stream, multiple-data stream) parallel computers. These results demonstrate that parallel simulation techniques can achieve significant performance improvement, at least for certain benchmark applications.

4. Just a Matter of Time?

The successes described in Table II might lead one to conclude that it is simply a matter of time before parallel

Table II. Selected Speedup Measurements

Application	SU[a]	Platform	Protocol
Ant foraging. A collection of ants move over a terrain, foraging for food. Logical processes are used to model ants, nests, and grid sectors that define the terrain. This and the colliding pucks benchmark were designed to be simplified benchmarks that capture behaviors (specifically, a set of entities moving over a physical space) arising in applications such as combat models.[21]	13	BBN Butterfly 32 processors	Time Warp
Colliding pucks. A set of pucks slide over a frictionless surface. Deflections are simulated as pucks collide with each other or the edge of the surface.[35]	12	BBN Butterfly 32 processors	Time Warp
Combat models. Two opposing armies (an aggressor and a defender), each consisting of a collection of tank divisions, attack each other. The simulation consists of three major phases: the advance phase where the armies approach each other, the conflict phase where combat takes place across the battlefield, and the "clean up" phase where the outcome of the battle has been determined, but minor pockets of fighting remain.[75]	24	JPL Mark III 48 processors	Time Warp
Digital logic circuits. Gate- and transistor-level simulations of adders, multipliers, microprocessors,	25	BBN Butterfly 32 processors	Time Warp
direct-memory-access (DMA) circuits that transfer blocks of data, and various special purpose circuits were studied. Briner reports speedups ranged from 10 to 25.[12] Soulé and Gupta[66] report speedup relative to a sequential execution of Chandy/Misra ranging from 7 to 9, but then go on to report that performance relative to an efficient event list algorithm (as opposed to the parallel algorithm executing on a single processor) is much worse, and achieves only 5-fold speedup.	5	Simulated Stanford Dash Multiprocessor 64 processors	Chandy/Misra
Health care system. A hierarchical network of hospitals is simulated where hospitals at higher levels have a higher degree of capability. Patients move up the hierarchy if their illness cannot be handled by the local facility. A 20-millisecond delay was added to each event to reduce the effect of overhead computations, and expose the amount of parallelism that the simulator is able to exploit.[7]	18	Transputers 32 processors	Time Warp
Ising spin. This model consists of atoms assigned to fixed positions in a two dimensional grid. Atoms attempt to change the direction of their spin at randomly selected points in time (attempted spin changes follow a Poisson process). At each attempt, a new spin direction is computed based on the direction of spin of neighboring atoms as well as the spin value of the atom attempting to change.[47]	1900	CM-1 16,384 processors	Bounded lag, a conservative protocol based on periodic, global synchronizations and simulated time windows[47]
Multistage interconnection networks. Networks such as those found in multiprocessors are modeled. Each stage contains a collection of feedforward crossbar switches.[4]	14	Sequent 15 processors	Distance between objects, similar to bounded lag, but without time windows[4]

Table II. (*Continued*)

Application	SU[a]	Platform	Protocol
Petri network. A set of "tokens" move through a network of "places." When a token has arrived at each incoming arc of a place, the place "fires," generating new tokens which are forwarded to other places.[68] Petri networks are often used for performance evaluation studies.	10	Sequent 12 processors	Selective receive, a specialized conservative protocol for simulating Petri nets[68]
Telephone switching networks.[31, 72] Telephone traffic in a network of switching offices is simulated. The offices are fully connected, i.e., each has a trunk line to each other office. If a direct connection from one office to another is not possible because no lines are available in the direct trunk line, calls are routed through an intermediate switch.	12 6	Transputers 36 processors iPSC/860 16 processors	Time Warp Sweep, an algorithm designed specifically for parallel simulation of telephone switching networks[31]
Queueing networks. Many performance studies use queueing networks as a benchmark for evaluating the performance of the parallel simulator. It is useful because certain real-world applications, e.g., telecommunication systems, may be modeled as networks of queues. The results reported in the other columns of this table indicate speedup of simulations of a closed queueing network consisting of a set of jobs moving among a set of first-come-first-serve queues. While Reed, Malony, and McCredie[59] report discouraging results, Fujimoto[25] reports significant speedups.	57 < 1	BBN Butterfly 64 processors Sequent 5 processors	Time Warp Chandy/Misra
Road traffic. Traffic for 292 vehicles and 292 road junctions is simulated in this study. Individual vehicles are programmed to attempt to make progress in a certain direction, or to make turns at intersections.[51]	19	Transputers 33 processors	Chandy/Misra

[a]SU denotes speedup.

simulation is embraced by the general simulation community. I believe that this is *not* the case. Though highly encouraging, these results are somewhat misleading. First, one tends to see a biased picture because positive results usually find their way to publication, but negative results often do not. Second, research results report speedup, but seldom report the effort that went into developing the code to achieve acceptable performance. This effort is often quite substantial, as will be discussed momentarily. Third, most of these results were produced by highly skilled experts in the field with a high degree of sophistication or coaching with respect to developing high performance software for parallel computers. In many cases, the application programmer is the same person who developed the underlying parallel simulation mechanism, and is thus intimately familiar with the algorithm being used. This is quite different (fortunately) from the situation that currently prevails for sequential simulation software. Many sequential software packages exist, and can usually be successfully applied to the problem at hand without in-depth understanding of the simulator's internal operation.

There is some evidence to indicate that when "ordinary" programmers are used to develop parallel simulation software, the results are not so favorable. For instance, a group of programmers were assigned the task of developing a parallel simulation model using the Jet Propulsion Laboratory's TWOS (Time Warp Operating System) software. The programmers were experienced in coding simulation applications for sequential machines, but had little experience in parallel computation, let alone Time Warp. The programs that were developed performed very poorly, and had to be discarded. It was only after a long redevelopment effort that programs achieving acceptable performance were obtained. A key reason that the original efforts failed was that the simulation model was most naturally programmed by making extensive use of *query events* to allow processes to access state variables defined in other processes. This inevitably leads to poor performance, as will be elaborated upon later.

Similarly, researchers at Los Alamos National Laboratories reported difficulties in parallelizing their software, although the problems they encountered were due, at least

in part, to the specific software they were using rather than to parallel simulation per se.[61] This work involved converting a large-scale combat model developed in Modsim, an object-oriented simulation language (discussed later), to a parallel implementation utilizing the TWOS software. A large part of the problem stemmed from the fact that the simulation model was programmed using tens of thousands of "tiny" objects, many of which contain few state variables. Operations on these objects often contained very little computation. However, the TWOS Time Warp executive was designed to support a modest number (e.g., hundreds) of objects, where operations on the objects were assumed to require significant amounts of computation (e.g., tens of milliseconds per event). Such large amounts of computation are required because each operation entails a certain amount of "system" computation, e.g., state saving to enable one to recover the state of the process should a rollback occur later. Unless the simulation computation is much larger than these overheads, performance will be poor. Because of this gross mismatch between the problem decomposition and the parallel simulation environment, the Los Alamos model never successfully ran on the Time Warp system.

At Stanford, Soulé and Gupta[66] report difficulties in a failed effort to use Chandy and Misra's deadlock detection and recovery algorithm to speed up the simulation of logic circuits. Like the Los Alamos experience, one difficulty they encountered was the amount of computation per event was very small, causing overhead computations to dominate.

Another difficulty that must be overcome is bottlenecks may lead to poor performance. Encouraged by their earlier successes in parallelizing combat models, Time Warp researchers at JPL developed models to simulate the strategy used by the allied land forces to attack the Iraqi army in the Persian Gulf conflict. These models produced disappointing speedup results (less than 10 using 82 processors). The military strategy called for using a narrow corridor through which the allied forces "punched through" the Iraqi lines. This leads to intense simulation activity along a small portion of the simulated battlefield. The resulting bottleneck limited the speedup that could be obtained.

The experiences at JPL, Los Alamos, and Stanford highlight a seemingly inescapable fact: *a simulation model that is developed according to what is most "natural" or "elegant" to the simulation programmer is often one that is poorly suited for execution on a parallel computer*, even if the system being simulated contains substantial amounts of concurrent activity. Here, at Georgia Tech, I have observed similar problems in student projects involving the development of parallel simulation models of cellular telephone networks and memory systems for multiprocessor computers.

5. Some Technical Problems

There are important technical issues that make development of software for parallel simulators substantially more difficult than developing software for sequential machines. One is that programming styles that are widely used in sequential programming are either unsupported or inefficient in existing parallel simulation mechanisms. For example, as mentioned earlier, shared state variables that are visible to concurrently executing simulator processes are not supported by most parallel simulation mechanisms. Notable exceptions that do support shared state variables are described by Jones et al.[39] and Fujimoto.[26] Of course, one can emulate shared variables by simply defining a logical process to hold each one, and replacing each "read" or "write" operation with a message send to request that the operation be performed. Query events were provided in TWOS for this purpose. However, a query event entails orders of magnitude more overhead in parallel simulation as compared to sequential simulation, where only a memory reference is required. In parallel simulation, a query requires that a message be sent, and a reply message be created and returned. Furthermore, in Time Warp, these messages could also cause rollbacks, and incur still other overheads. Even in implementations that are optimized to avoid rollback for query events, the overhead is still substantial, so poor performance will result if queries are used extensively.

This is just one of many factors that make developing parallel simulation code very complex and time consuming. There are many others. Partitioning the model into logical processes (required by most parallel simulation mechanisms) becomes complex because performance implications must be considered. These performance considerations may conflict with what is convenient and natural from a modeling standpoint. Also, statistics collection and initialization may have to be parallelized. Although some of these issues (e.g., initialization) do not represent major technical obstacles, they do nevertheless contribute to the complexity of the application code.

The difficulties just described are not limited to optimistic synchronization methods. Indeed, they are usually even more severe for conservative approaches because these protocols often require an even greater level of sophistication on the part of the programmer to achieve acceptable performance. This is because conservative mechanisms must determine when it is "safe" to process an event, i.e., when it can be guaranteed that the process will not receive any events containing a smaller timestamp than the one about to be processed. Recall that in the queueing network example described earlier (see Figure 2), LP_A had to determine that LP_C would *not* send it a message with timestamp smaller than 100 before it could process the timestamp 100 message it had received from LP_B. Determining this information is highly application dependent. Earlier, I had to rely on a minimum service time in each server to resolve this issue. Thus, conservative algorithms such as the null message scheme require knowledge concerning what messages *might* be generated in the future. Currently, this information must be provided explicitly by the simulation programmer, and programmed directly into the simulation model itself (work by Bagrodia and Liao[10] on this problem is described later). Failure to write the application in such a way as to fully extract and exploit information such as minimum timestamp increments will usually lead to very poor performance.[24] Again, existing parallel simu-

lation mechanisms demand that the programmer be intimately familiar with these facts if good performance is to be obtained. Needless to say, details such as these significantly complicate model development.

Despite growing overall interest in the field, only a modest amount of work has been focused on easing the amount of effort and expertise required to parallelize simulation applications. I believe that until this problem receives more attention, parallel simulation will fail to make a significant impact in the general simulation community.

6. General Purpose Parallel Computation

One could argue that the problems that have just been described are no different than those of parallel computation in general. After all, why should developing parallel simulation software be any different from developing parallel programs to perform matrix operations or to solve the traveling salesman problem? It is well known that our ability to produce highly parallel machines has far surpassed our ability to effectively program them, and there continues to be an enormous amount of work aimed at attacking this problem. One might therefore argue that once this problem has been solved for general purpose parallel computation, it will automatically be solved for parallel simulation.

While there is some truth to this statement, I believe the programmability problem in parallel simulation is in one sense easier, and, in another sense, harder than for general parallel computations. In any case, it is quite different, and will not be automatically solved by research in other fields. It is easier in the sense that, with simulation applications, one is attempting to capitalize on concurrency in the system being modeled. Indeed, cause-and-effect relationships translate directly to dependencies between the corresponding simulator computations. Thus, we have the advantage that there is a reasonably concrete conceptual model of the simulation application that can be used as a starting point to construct the parallel program. General parallel applications often do not share, or benefit, from this characteristic. Conceptualizations of the problem, which in turn give rise to specific parallel algorithms, must usually arise from the designer's own ingenuity.

At the same time, the parallelization problem is also harder for simulation applications. Clearly, whenever parallel computation comes into the picture, performance becomes paramount (otherwise, why bother with parallel computers?). But discrete event simulation programs tend to be very dynamic, irregular, and data dependent. These characteristics make the costs of "doing business" in parallel simulation programs higher than in other, more structured applications (e.g., data parallel computations such as matrix operations on large sets of data). Runtime mechanisms must often be used because there is insufficient information at compile time to make intelligent decisions. This affects programmability because the cost of essential programming primitives and mechanisms is higher, so great care, and, in some cases, restraint must be exercised in using them.

For instance, consider synchronization in parallel discrete event simulation. Where simple blocking primitives for synchronization will suffice in many general parallel applications, deadlock avoidance algorithms (i.e., null messages) or rollback mechanisms are accepted practice in discrete event simulation. Where a message suffices in object-oriented systems, rollback and state saving are also needed in Time Warp. Shared variables can be simply accessed using the hardware primitives in a shared memory multiprocessor, but the issue of shared state becomes much more complex in discrete event simulation. The costs associated with simple mechanisms are sometimes orders of magnitude higher in parallel simulation than in general purpose parallel computation. Programmers must account for these costs in designing their programs, or poor performance will result. Increasing overheads require increasing computation per event to hide these costs. This may be difficult to accomplish in practice. These aspects make the already challenging problem of parallelization all the more difficult for simulation applications. To be sure, advances in new programming languages and paradigms for general purpose parallel computation will help simplify the task of developing parallel simulation software, but they are not by themselves enough.

What are the "silver bullets" that offer hope in making parallel simulation more accessible to the simulation community? A few approaches that have been proposed are outlined below:

- application specific libraries,
- new languages for parallel simulation,
- support for shared state, and
- automatic parallelization.

In the following sections, I elaborate on these approaches.

7. Application Specific Libraries

One solution to the programmability question is to avoid programming altogether, or at least as much as possible. This is possible in certain, established, application domains where a set of well-defined primitive components is known. One can envision a parallel simulation package consisting of a collection of pre-defined simulation modules. For instance, a queueing network simulation package might include different types of servers and queues (see Figure 4). Simulation users would then specify their model by instantiating and interconnecting these modules, e.g., using a graphical interface to "draw a picture" of the network. Using this approach, one can tolerate the difficulties associated with developing parallel simulation software. This is because programming the models is the responsibility of a few individuals for whom it is reasonable to expect a high degree of sophistication. This is not a new idea, of course, but it has not been widely exploited in the context of parallel simulation.

Two application areas that appear to be particularly promising candidates for this approach include telecommu-

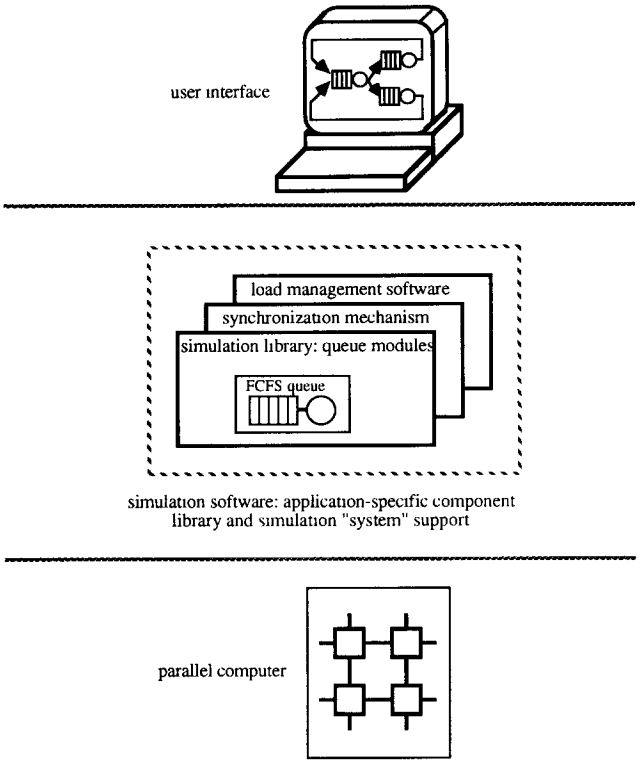


Figure 4. Simulation environment. Users construct a simulator by selecting component modules from a parallel simulation library using a graphical interface. The library effectively isolates the user from the complexities of parallel computation.

nication networks and digital logic circuits. Manufacturing applications may represent a third. Tsai, Das, and Fujimoto[70] describe a prototype that allows users to specify simulations of communication networks using a graphical interface called Arcsim, a program developed by Optimization Technologies.[40] Arcsim produces a textual specification of the network in CACI's Network II.5 language.[30] Another program then reads the Network II.5 specification and builds a Time Warp-based parallel simulation program that can then be executed on a parallel computer. Simulation results can then be viewed graphically using the Arcsim program.

Another parallel simulation package for simulating communication networks is described by Phillips and Cuthbert.[57] Components of the simulator include queue processes and traffic generators, among others. The simulation package is reported to have been implemented using two different conservative synchronization mechanisms, the null message algorithm and a synchronous approach. However, disappointing performance is reported; in most cases, the parallel simulator executing on a network of transputers ran more slowly than the sequential simulator.

A second area where this approach may be viable is in the parallel simulation of digital logic circuits. Standard components and building blocks have evolved, including logic gates, memories, arithmetic circuits, and even complete CPUs. Several studies have been reported that target gate level logic simulation.[12, 16, 66, 67]

Library-based simulation tools provide a means by which research in domain specific simulation methods might affect the general simulation community. Work along these lines has been in progress for some time, but has seen only modest application in the simulation community.

8. New Languages

In sequential simulation, much work has been devoted to devising new languages to simplify the task of developing complex simulation models. Languages such as Simscript, Simula, and GPSS, among others, have flourished. A new, challenging aspect of this problem is to define languages that are not only suitable for modeling, but for concurrent execution as well.

Parallel simulation languages differ in at least two ways from sequential simulation languages:

- the language should avoid constructs that are not easily supported by the underlying synchronization mechanism, e.g., shared state variables or use of any primitive that relies on a global event list, and
- the language should include primitives that are natural from a modeling standpoint and that enhance concurrency in the underlying synchronization mechanism.

An important question that must be answered is "just how transparent should the parallel execution be?" Ideally, the parallel simulator should be completely transparent to the application program so that the modeler can concentrate on the model itself rather than details of the underlying synchronization mechanism. By analogy, programmers for sequential simulations usually need not be concerned with the data structure used to implement the underlying event list. However, a language that hides too much may deceive the user. Simulation primitives that are convenient, but expensive (in terms of computation time) will lead to very poor performance if they are invoked too frequently. As discussed earlier, excessive usage of the query event for reading another process's state variables can lead to disaster! Such primitives should somehow be flagged so that they are used in an appropriate way, or eliminated altogether. The latter alternative was taken in the TWOS implementation of Time Warp. Neither approach is very satisfying, however, because in both cases, the burden is laid to rest on the modeler's shoulders.

8.1. Three Language Proposals

Perhaps the three most well-known languages that were developed for parallel simulation are Modsim,[11, 74] a Module-2-based language developed by CACI Products Co., sim++,[6] a library of primitives for simulations written in C++ developed by Jade Simulations, and Maisie,[8, 10] a C-based language developed by Bagrodia and Laio at

UCLA. A good discussion and comparison of these languages and others is given by Baezner.[5]

Modsim and sim++ are based on the object-oriented programming paradigm. Briefly, an object-oriented program consists of a collection of objects (actually, object instances; here, I will use the terms object and object instance synonymously), where each object contains state information, and a set of routines or *methods* for performing operations on the object. Parallel simulation strategies and object-oriented programming have obvious similarities in that a logical process is similar to an object, and sending a message is similar to invoking a method. Thus, it is not surprising that many parallel simulation language proposals are based on this paradigm.

Common characteristics of Modsim, sim++, and Maisie include the following:

- The simulation program is composed of a collection of logical processes that interact by exchanging time-stamped event messages. Modsim and sim++ provide a "process object" type to represent a logical process.
- No shared state between concurrently executing processes is allowed. Sim++ does allow shared read-only variables, as well as shared state between processes that are guaranteed to always reside on the same processor.
- All three languages provide various forms of a "wait" primitive to advance simulated time. For instance, a process modeling a server in a queueing network simulation might execute "wait 20" to denote the fact that a customer is being served for the next 20 units of simulated time. These primitives are similar to those that are commonly found in sequential simulation languages.

Modsim and sim++ provide somewhat different approaches to representing concurrency within the object-oriented paradigm. In Modsim, a simulation process can contain a collection of autonomous *activities*, one for each invoked method. Each activity can be viewed as a concurrent "thread of execution." For instance, one might model the operations of a tank as several activities (methods): one to rotate the turret to a new position, another to turn the tank to a new direction, and a third to receive an incoming message from headquarters. By contrast, a sim++ object is viewed as a *single* thread of control. Concurrency is modeled by using separate objects, one for each concurrent activity. The difference between these two different approaches is largely one of convenience for the programmer, rather than concurrency in the simulator, however. Existing implementations of Modsim do not provide concurrent execution of threads within the same object. This is because the process's state is shared among its activities, and as mentioned earlier, no support is provided for shared state.

Modsim and sim++ were designed with implementation on a Time Warp system in mind. Maisie targets both conservative and optimistic synchronization mechanisms, a much more ambitious goal. The inclusion of conservative mechanisms is significant because, as mentioned earlier, conservative approaches often require much more application specific information than do optimistic approaches.

Each language offers certain insights into language design for parallel simulation, including the following:

- In addition to the aforementioned problem with shared state, the semantics of certain Modsim primitives also make concurrent execution of methods within a single process difficult. One can "interrupt" activities within a process. If several activities are eligible for interruption, the interrupt primitive specifies that *the most imminent activity* is interrupted. The "most imminent activity" is easily determined in a sequential environment by scanning the list of pending events. It is much more difficult to determine in a parallel environment, however, if activities execute concurrently on different processors, because no centralized event list is available. This highlights the fact that simulation primitives which implicitly assume an event list are not well suited for parallel simulation.
- Sim++ provides a primitive called *sim_write_lock* that is used by an entity to "lock" certain state variables so that they cannot be modified by the process until it is first unlocked. The purpose of this primitive is to inform the Time Warp executive that it need not take checkpoints on that portion of the process's state vector (checkpoints are required in case a rollback later occurs), since it is not being changed.
- In Maisie, a wait statement automatically buffers incoming messages that arrive while a process is waiting (sim++ also provides a similar primitive). For instance, if a process at simulated time 100 executes "wait 20," then it can be guaranteed that no new messages will be generated until simulated time 120. This information can be used for generating null messages, which guarantee that no new messages will be sent with timestamp smaller than 120.
- The wait primitive can also provide information that can be used to optimize the performance of rollback-based mechanisms such as Time Warp. For instance, in a queueing network simulation, a process might wait from simulated time 100 to 120 because it is simulating the service of a job. Assuming that this service cannot be interrupted, messages arriving between time 100 and 120 can be received out of timestamp order without any deleterious effects, so no rollback is necessary should this occur. The wait primitives in Maisie provide sufficient information to the simulation kernel to detect this situation, and avoid certain unnecessary rollbacks.

8.2. Future Directions in Language Design

Ideally, one should be able to develop a simulation model without being overly concerned with the execution environment, and achieve "acceptable" (though not necessarily optimal) performance should it be executed on a parallel machine. At present, the simulation community is very far from this situation. Today, even experienced simulation programmers will have difficulty achieving acceptable performance on a parallel computer without a relatively high degree of sophistication in parallel computation and paral-

lel simulation techniques. I see two general approaches to attacking this problem, one evolutionary, and the second revolutionary.

The evolutionary approach builds upon the rich experiences that have accumulated with simulation languages on sequential machines. The language proposals just described are of this nature. Another approach, using hierarchical decomposition combined with a specialized parallel simulation protocol, is also in a similar spirit.[77] Evolutionary approaches attempt to build a synergistic co-existence of variations of traditional simulation primitives (e.g., wait and interrupt primitives) with the particular constraints associated with implementation on a parallel computer.

In these approaches, I would argue that the fundamental problem that needs to be addressed boils down to one thing: achieving good performance for those constructs that are frequently used in sequential simulation models. While some constructs can probably be safely avoided, others cannot. For instance, I believe the lack of shared state is one issue that cannot simply be sidestepped. Without a suitable alternative, simulation programmers are left to their own devices to realize this aspect of the model. This, in turn, requires great sophistication in parallel processing, as well as simulation modeling. It is my belief that approaches to simplify model development need to confront these performance issues head on. Until the performance issue is satisfactorily resolved, language designers will have their hands tied, and will not be able to provide constructs that both satisfy the modeler's needs and achieve efficient execution.

Along the same lines, parallel simulation mechanisms have not yet fully addressed the problems associated with dynamic creation and management of components making up the simulation. Although techniques for dynamically creating processes and allocating memory in optimistic protocols are known (e.g., see Tinker and Agre[69]), their performance properties, and techniques for maximizing performance, have not been widely studied. Specifically, one needs to be able to roll back the creation of any entity. Dynamic process creation has similarly not been widely studied for conservative protocols, and is expressly forbidden by many approaches. Furthermore, state saving overhead, already a problem in optimistic synchronization, becomes even more of a problem when copies of complex, dynamically created data structures must be made. Packaging complex data structures into messages for transmission to other processes represents yet another time-consuming operation.

Other, more radical, revolutionary approaches to simplifying the development of parallel simulation software have also been proposed. One approach along these lines is to use logic programming languages for simulation modeling, and concentrate parallelization efforts in efficiently executing these languages.[17, 29] Time Warp has been proposed as the execution mechanism for Prolog programs, for instance. Another approach uses the Unity paradigm as the basis for model development.[1, 2] Unity has also been used as the basis for developing parallel simulation protocols, e.g., see Bagrodia, Chandy, and Liao.[9] This provides not only a means for concurrent execution, but a framework for proving properties of the simulation program as well. It remains to be seen, however, to what extent the simulation community will embrace such approaches. By analogy, it is perhaps noteworthy that a great deal of effort has gone into studying functional programming languages for general purpose parallel computation, yet the languages that are most widely used today for parallel computation are imperative languages such as C or Fortran.

It is clear that the simulation community is only beginning to scratch the surface with respect to developing new languages for parallel simulation. While some important work has been completed in this direction, the field is still relatively wide open.

9. Shared State

Many simulation models naturally consist of two types of components: *passive* entities that do not directly affect the state of the system, and *active* entities that do. For instance, in a combat model, the passive entities include terrain data and information indicating the location of various elements in the simulation (roads, mine fields, combat units, etc.). Active entities include the combat units themselves, command stations that issue orders, etc. It is simple and natural to define variables to model the passive entities, and simulation processes to model the active ones. While it is possible that new modeling methodologies or world views might be developed to eliminate this conceptualization, it is not immediately clear what form such methodologies would take.

One approach to efficiently implement shared state is to transmit required information to processes in advance of when it is needed. This is sometimes called "push processing," while the more direct approach of reading the data on demand as they are needed is called "pull processing."[76] Push processing has been demonstrated to overcome the performance problems (recall the earlier discussion regarding query events) associated with pull processing, and can lead to good performance. It does, however, greatly complicate the coding of the simulation, and thereby inflate development time and detract from the software's maintainability. Because multiple copies of modifiable data may be distributed across several processes, the application program must, in effect, implement a memory management protocol to maintain consistency among the multiple copies.

Another approach is to support shared state in the simulation mechanism itself. Recall that the problem with shared state is the following: Consider two concurrently executing simulation processes both examining a common state variable, e.g., LP_A and LP_B in the queueing network example discussed earlier accessing the `QLength` variable (see Figure 2b) in LP_B. In general, these processes will be at different points in simulated time. LP_A at simulated time 100 expects to see the state of the variable as it existed at simulated time 100. On the other hand, LP_B at time 80 expects to see the state as of time 80. Which value should be maintained in the variable?

One approach to resolving this dilemma is an abstraction called space-time memory (not to be confused with the space-time simulation protocol[9] While conventional mem-

ory is viewed as a one-dimensional array of values that is addressed with a *spatial* coordinate (i.e., a word address), space-time memory is two-dimensional, and addressed with both a *spatial* and a *temporal* coordinate. In the queueing network example, LP_A would specify that it is reading the value of `QLength` at simulated time 100, and the memory system would guarantee that it returns the version that existed at 100. Similarly, LP_B would request the version that existed at time 80.

Space-time memory is used in conjunction with a Time Warp like rollback mechanism to recover from synchronization errors.[26] To enable rollback, a history of the execution is required. Assume that the shared state is represented as a collection of object instances (or simply *objects*) $O_0, O_1, \ldots, O_{n-1}$ that model the state of the system being simulated. Each object represents a collection of state variables that the simulation mechanism treats as an indivisible unit. Here, objects simply denote collections of state variables. Methods are not a priori associated with objects (as in object-oriented programming).

A point (T, O) in Figure 5 denotes the state of object O at simulated time T. The evolving state of an object is represented by a horizontal *object history* line. The short vertical bars intersecting the object history denote *state changes*. For example, object O_1 changes state at simulated times 0 (where it was initialized), 10, and 25; i.e., events with timestamps of 10 and 25, respectively, modified some variable within the object. A segment of the object history line that is terminated at each end by state change bars, and is not broken by any other change bars, is called a *version* of the object; it denotes the object's value over some simulated time interval. In Figure 5, there are three versions of object O_1 that cover the time intervals $[0, 10)$, $[10, 25)$, and $[25, \infty)$.

The underlying runtime system maintains an *execution history* that is superimposed on top of the state history. Three event computations are shown in Figure 5, C_1, C_2, and C_3 at simulated times 10, 25, and 18, respectively. The arc extending from O_1 to C_1 indicates that C_1 read data from this object. The arc from C_1 to O_1 indicates that C_1 also modified O_1. The arc from C_1 to C_2 indicates that C_1 scheduled the event C_2.

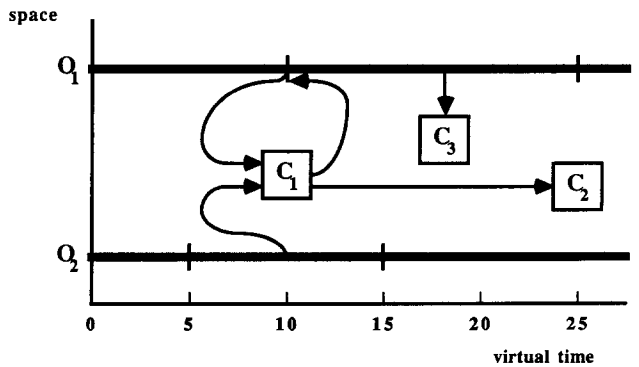


Figure 5. Space-time memory showing the evolution of state variables over simulated time and event computations accessing the state variables.

These arcs provide the necessary information to enable rollback operations. For instance, if C_1 is rolled back, the write it performed on O_1 must be retracted, implying that C_3, which read the erroneous value, must also be rolled back. Similarly, event C_2 that was erroneously scheduled must be cancelled.

Space-time memory can be implemented using either push or pull processing. An implementation on a shared memory multiprocessor is described by Ghosh and Fujimoto[32] that uses pull processing. The implementation is optimized to perform well on a shared memory architecture. A push-based approach amounts to using a "prefetch" mechanism to fetch data before they are needed. This could be facilitated by compiler support.

The principal objective of the space-time memory mechanism is to provide the abstraction of shared state, thus simplifying code development, but at the same time hiding the complexities of memory management from the user. In this respect, it is not unlike *distributed shared memory* systems that allow shared memory constructs to be used to program message-based multicomputers (e.g., see Li and Hudak[42]). Space-time memory allows the simulation to be programmed in a natural way using shared state variables where appropriate, but (hopefully) without sacrificing performance. More research still needs to be completed, however, to fully realize this goal. One interesting application of space-time memory is in the automatic parallelization of sequential simulation programs. I will describe this application next.

10. Automatic Parallelization

Existing parallel simulation methods require that the application program be rewritten for parallel execution. If a larger base of software has already been developed, this is a serious drawback. Another approach is to use compilers and runtime system support to *automatically* parallelize the sequential simulation. Much work has been devoted to automatically parallelizing regular numeric computations. However, these approaches cannot be directly applied to discrete event simulation because data dependence relationships cannot be easily predicted at compile time. They depend on computed values such as timestamps, defeating existing compiler-based approaches. Furthermore, because each event requires an access to the event list, execution may be serialized.

The space-time memory construct described previously suggests an alternative approach. A sequential discrete event simulation program contains two principal data structures: the variables that describe the state of the system, and an *event list* that contains pending events that have been scheduled, but not yet processed (see Figure 6). The sequential simulator contains a loop that repeatedly removes the smallest timestamped event from the event list, and then executes the code to process that event. This code will read and (possibly) modify the state variables, and schedule new events into the event list. Abstractions based on process-oriented and object-oriented simulation

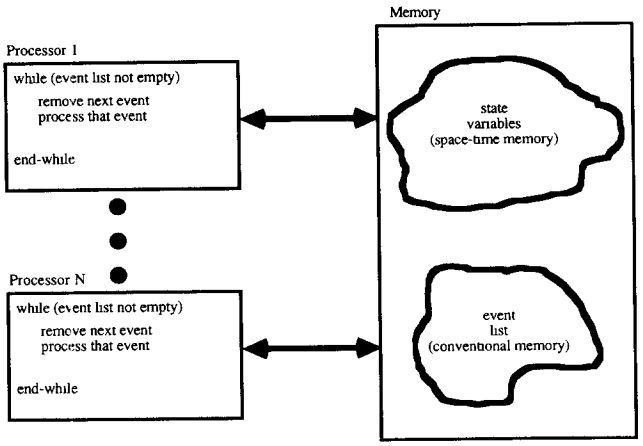


Figure 6. Automatic parallelization of sequential event list simulator. One loop executes on each processor.

methodologies can be built on top of this paradigm, but the basic underlying mechanism remains the same.

Assume for the moment that event procedures may *only* (1) read or modify state variables and (possibly) (2) schedule new events. Furthermore, assume each event is assigned a unique timestamp to reflect the sequential execution order (this requires appending "hidden" fields to the timestamp of each event; details of how to accomplish this are beyond the scope of the present discussion, but one such scheme is described by Mehl[50]). The goal is to execute the sequential simulation program "as is" on a shared memory multiprocessor. This can be accomplished if (see Figure 6):

1. the state variables are stored in space-time memory, and
2. the scheduler loop is executed in each processor of the multiprocessor system.

This is because the space-time memory mechanism will automatically ensure that each read operation returns the same value that is returned in the sequential execution. Because the read operations provide all of the inputs to the event computation, each event computation will produce exactly the same results as the corresponding sequential execution. Anytime this is not the case, the computation is rolled back, transparent to the application program.[26]

Using space-time memory and rollback-based synchronization, the parallel simulator achieves the same results as the sequential simulator, and, at least in principle, does not require any modification to the sequential simulation code for processing events. A compiler for automatically parallelizing SIMSCRIPT II.5 programs using this approach has been developed, and is currently being evaluated.[71] In the previous discussion, a global event list containing the set of pending events was assumed. A concurrent priority queue data structure[38] or multiple event lists could be used to prevent this from becoming a bottleneck.

Sequential simulation programs often utilize other simulation primitives in addition to event scheduling and accessing state variables. Libraries are usually provided for statistics collection, queue management, and random number generation, among others. In principle, these can be simply treated as part of the application program, and do not affect the correctness of the parallelization scheme. They do affect efficiency, however. For instance, each call to the random number generator modifies the generator's seeds. If each event requires one or more random numbers, and the seeds are placed in space-time memory, execution will be serialized and poor performance will result. Similarly, statistics collection may serialize execution if no special precautions are taken. Thus, parallelized libraries or automatic code restructuring techniques are required if this approach is to be viable.

Other sequential simulation primitives are more problematic in that they rely on the internal structure of the sequential simulator, which is not preserved in the parallel execution. Certain simulation primitives may directly interrogate the event list, e.g., a primitive that returns the number of pending events. In principle, these primitives can be accommodated by building the necessary data structure (e.g., a mirror image of the event list) in space-time memory. Alternatively, one can observe that the event list at simulated time T includes those events that were scheduled prior to simulated time T and contain a timestamp greater than T. One can then scan the parallel event list to collect this information. It remains to be seen how efficient this approach is, and what techniques can be developed to optimize performance.

It is reasonable to ask if automatic parallelization is a viable approach for discrete event simulations, particularly since even *manual* parallelization does not appear to be so straightforward. At this point in time, I can only speculate. The approach outlined previously relies on two key premises: the application has a sufficient amount of concurrency to offset the overheads associated with parallel execution, and the rollback-based synchronization mechanism (or some variation) has the ability to effectively exploit this parallelism. Published studies on Time Warp performance (e.g., see [25, 75]) offer some hope that the latter premise can be met, though additional work is required to further reduce Time Warp overheads (particularly state saving).

Do sequential simulations contain sufficient parallelism? Simulations contain two types of dependencies that limit the amount of available parallelism. First, there are dependencies that arise from the simulation model itself. The cause-and-effect relationships of the system being modeled map directly onto dependencies between events in the simulator. Therefore, one can conclude that the simulator will only have parallelism if the system being simulated also contains parallelism. The envisioned applications for many large scale simulations, e.g., simulations of combat scenarios and communication networks, appear to have this property. The second type of dependencies are artifacts of the simulation itself, and do *not* correspond to any behavior in the system being simulated. As mentioned earlier, global variables that are used for random number generator seeds and statistics collection are two examples. Ultimately, the viability of automatic parallelization may

depend on the development of automatic techniques to restructure the simulation to avoid these latter dependencies. However, even if automatic restructuring proves to be infeasible, semi-automatic approaches may still be acceptable.

11. Conclusion

In this article, I have argued that although much has been learned about concurrent execution of discrete event simulation programs and many successes have been reported over the last 15 years, to date, this technology has failed to make a significant impact in the general simulation community. While work in "traditional" areas of parallel simulation research will (and should) continue, the issue of simplifying the development of efficient parallel simulation software has not received adequate attention. Several approaches that attempt to address this problem, including application specific library packages, new simulation languages, support for shared state, and automatic parallelization offer some hope in making parallel simulation a general tool with widespread application, but much more research remains to be done. I believe the success of parallel simulation in the next decade hinges critically on the success of these or other approaches to simplifying model development.

In the past, simulation modelers could rely on faster uniprocessor computers to achieve reduced model execution times. It is unlikely, however, that this will continue. As computing speeds begin to approach fundamental limits such as the speed of light, performance improvements of uniprocessor computers will come at a much slower pace. Simulation users will have to rely more and more on parallelism to derive the benefits of high performance computing. If efforts to exploit parallel simulation technology fail, simulation users will be forced to make more and more compromises in accuracy and problem size, as systems become larger and more complex. Thus, in the long term, the success or failure of parallel simulation methods could ultimately determine the extent to which simulation can remain a viable, effective approach to analyzing large, complex systems.

Acknowledgments

This article is based in part on a keynote address presented by the author at the 1991 Workshop on Parallel and Distributed Simulation (PADS) on January 24, 1991. Performance data for telecommunication network simulations reported in Table II were provided by Greg Lomow. Anecdotes concerning difficulties encountered in parallelizing combat models at JPL were provided by Peter Reiher. The work described in this article concerning space-time memory was supported by Innovative Science and Technology contract number DASG60-90-C-0147 provided by the Strategic Defense Initiative Office and managed through the Strategic Defense Command Advanced Technology Directorate Processing Division. The work describing automatic parallelization of sequential simulation programs was supported by NSF grant CCR-8902362. Finally, the author is grateful to Yannis Nikolaidis, Rajive Bagrodia, and the editors and anonymous referees for their numerous, thoughtful, and detailed comments on earlier versions of this document. Their suggestions lead to significant improvements in both the presentation and content of this article.

References

1. M. Abrams, E.H. Page and R.E. Nance, 1991. Linking Simulation Model Specification and Parallel Execution through Unity, in *1991 Winter Simulation Conference Proceedings* (December), pp. 223–232.
2. M. Abrams, E.H. Page and R.E. Nance, 1991. Simulation Program Development by Stepwise Refinement in Unity, in *1991 Winter Simulation Conference Proceedings* (December), pp. 233–242.
3. I.F. Akyildiz, L. Chen, S.R. Das, R.M. Fujimoto and R. Serfozo, 1992. Performance Analysis of Time Warp with Limited Memory, in *Proceedings of the 1992 ACM SIGMETRICS Conference on Measuring and Modeling Computer Systems*, Vol. 20 (May), pp. 213–224.
4. R. Ayani and H. Rajaei, 1990. Parallel Simulation of a Generalized Cube Multistage Interconnection Network, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 22 (SCS Simulation Series, January) pp. 60–63.
5. D. Baezner, 1991. Language Design for Parallel Simulation, M.S. Thesis, Computer Science Department, University of Calgary, Calgary, Alberta, Canada.
6. D. Baezner, G. Lomow and B. Unger, 1990. Sim++: The Transition to Distributed Simulation, in *Distributed Simulation*, Vol. 22 (SCS Simulation Series, January), pp. 211–218.
7. D. Baezner, G. Lomow and B. Unger, 1993. A Parallel Simulation Environment Based on Time Warp (to appear in *International Journal of Computer Simulation*).
8. R. Bagrodia, 1991. Iterative Design of Efficient Simulations Using Maisie, in *1991 Winter Simulation Conference Proceedings* (December), pp. 243–247.
9. R. L. Bagrodia, K. M. Chandy, and W-T, Liao, 1991. A Unifying Framework for Distributed Simulation, *ACM Transactions on Modeling and Computer Simulation 1:4*, 348–385.
10. R.L. Bagrodia and W-T. Liao, 1990. Maisie: A Language and Optimizing Environment for Distributed Simulation, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 22 (SCS Simulation Series, January), pp. 205–210.
11. R. Belanger, B. Donovan, K. Morse, S. Rice and D. Rockower, 1989. *Modsim: A Language for Object-Oriented Simulation.* CACI Products Co., Los Angeles, CA.
12. J. Briner, Jr., 1991. Fast Parallel Simulation of Digital Systems, in *Advances in Parallel and Distributed Simulation*, Vol. 23 (SCS Simulation Series, January), pp. 71–77.
13. R.E. Bryant, 1977. Simulation of Packet Communication Architecture Computer Systems. MIT-LCS-TR-188, Massachusetts Institute of Technology, Cambridge, MA.
14. K.M. Chandy and J. Misra, 1979. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs, *IEEE Transactions on Software Engineering SE-5:5*, 440–452.
15. K.M. Chandy and J. Misra, 1981. Asynchronous Distributed Simulation via a Sequence of Parallel Computations, *Communications of the ACM 24:4*, 198–205.
16. M. Chung and Y. Chung, 1991. An Experimental Analysis of Simulation Clock Advancement in Parallel Logic Simulation on an SIMD Machine, in *Advances in Parallel and Distributed Simulation*, Vol. 23 (SCS Simulation Series, January), pp. 125–132.
17. J. Cleary, 1990. Colliding Pucks Solved Using a Temporal Logic, in *Distributed Simulation*, Vol. 22 (SCS Simulation Series, January), pp. 219–226.

18. R.C. De Vries, 1990. Reducing Null Messages in Misra's Distributed Discrete Event Simulation Method, *IEEE Transactions on Software Engineering 16:1*, 82–91.
19. P.M. Dickens and P.F. Reynolds, Jr., 1991. A Performance Model for Parallel Simulation, in *1991 Winter Simulation Conference Proceedings*, (December), pp. 618–626.
20. R.W. Earnshaw and A. Hind, 1992. A Parallel Simulator for Performance Modelling of Broadband Telecommunication Networks, in *1992 Winter Simulation Conference Proceedings* (December), pp. 1365–1373.
21. M. Ebling, M. DiLorento, M. Presley, F. Wieland and D.R. Jefferson, 1991. An Ant Foraging Model Implemented on the Time Warp Operating System, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 21 (SCS Simulation Series, March), pp. 21–26.
22. R. Felderman and L. Kleinrock, 1991. Bounds and Approximations for Self-initiating Distributed Simulation without Lookahead, *ACM Transactions on Modeling and Computer Simulation 1:4*, 386–406.
23. R. Felderman and L. Kleinrock, 1991. Two Processor Time Warp Analysis: Some Results on a Unifying Approach, in *Advances in Parallel and Distributed Simulation*, Vol. 23 (SCS Simulation Series, January), pp. 3–10.
24. R.M. Fujimoto, 1989. Performance Measurements of Distributed Simulation Strategies, *Transactions of the Society for Computer Simulation 6:2*, 89–132.
25. R.M. Fujimoto, 1989. Time Warp on a Shared Memory Multiprocessor, *Transactions of the Society for Computer Simulation 6:3*, 211–239.
26. R.M. Fujimoto, 1989. The Virtual Time Machine, in *Proceedings of the International Symposium on Parallel Algorithms and Architectures* (June), pp. 199–208.
27. R.M. Fujimoto, 1990. Parallel Discrete Event Simulation, *Communications of the ACM 33:10*, 30–53.
28. R.M. Fujimoto and D.M. Nicol, 1992. State of the Art in Parallel Simulation, in *1992 Winter Simulation Conference Proceedings*, pp. 246–254.
29. I. Futo, 1988. Distributed Simulation on PROLOG Basis, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 19 (SCS Simulation Series, July), pp. 160–165.
30. W.J. Garrison, 1991. *Network II.5 User's Manual*, CACI Products Company, La Jolla, CA.
31. B Gaujal, A.G. Greenberg and D.M. Nicol, 1993. A Sweep Algorithm for Massively Parallel Simulation of Circuit-switched Networks (to appear in *Journal of Parallel and Distributed Computing*).
32. K. Ghosh and R.M. Fujimoto, 1991. Parallel Discrete Event Simulation Using Space-Time Memory, in *Proceedings of the 1991 International Conference on Parallel Processing*, Vol. 3 (August), pp. 201–208.
33. A. Gupta, I.F. Akyildiz, and R.M. Fujimoto, 1991. Performance Analysis of Time Warp with Homogeneous Processors and Exponential Task Times, in *Proceedings of the 1991 ACM SIGMETRICS Conference on Measuring and Modeling Computer Systems*, Vol. 19 (May), pp. 101–110.
34. J.L. Hennessy and D.A. Patterson, 1990. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, CA.
35. P. Hontalas, B. Beckman, M. DiLorento, L. Blume, P. Reiher, K. Sturdevant, L. Van Warren, J. Wedel, F. Wieland and D.R. Jefferson, 1989. Performance of the Colliding Pucks Simulation on the Time Warp Operating System, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 21 (SCS Simulation Series, March), pp. 3–7.
36. D.R. Jefferson, 1990. Virtual Time II: Storage Management in Distributed Simulation, in *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing* (August), pp. 75–89.
37. D.R. Jefferson and H. Sowizral, 1982. Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control, Technical Report N-1906-AF, RAND Corporation, Santa Monica, CA.
38. D.W. Jones, 1989. Concurrent Operations on Priority-Queues, *Communications of the ACM 32:1*, 132–137.
39. D.W. Jones, C-C. Chou, D. Renk and S.C. Bruell, 1989. Experience with Concurrent Simulation, in *1989 Winter Simulation Conference Proceedings* (December), pp. 756–764.
40. T.M. Langlois, 1991. *ARCSIM Versions 3.0 User's Manual*, Optimization Technologies Inc., Huntsville, AL.
41. S. Lavenberg, R. Muntz and B. Samadi, 1983. Performance Analysis of a Rollback Method for Distributed Simulation, in *Performance '83*, Elsevier Science Pub., North Holland, pp. 117–132.
42. K. Li and P. Hudak, 1989. Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems 7:4*, 321–359.
43. Y-B. Lin, Parallel Trace-Driven Simulation for Packet Loss in Finite-Buffered Voice Multiplexers (to appear in *Parallel Computing*).
44. Y-B. Lin and E.D. Lazowska, 1990. Optimality Considerations of "Time Warp" Parallel Simulation, in *Proceedings of the SCS Multiconference on Distributed Simulation* Vol. 22 (SCS Simulation Series, January), pp. 29–34.
45. R. Lipton and D. Mizell, 1990. Time Warp vs. Chandy-Misra: A Worst-case Comparison, in *Distributed Simulation*, Vol. 22 (SCS Simulation Series, January), pp. 137–143.
46. B.D. Lubachevsky, 1988. Bounded Lag Distributed Discrete Event Simulation, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 19 (SCS Simulation Series, July), pp. 183–191.
47. B.D. Lubachevsky, 1989. Efficient Distributed Event-Driven Simulations of Multiple-loop Networks, *Communications of the ACM 32:1*, 111–123.
48. B.D. Lubachevsky, A. Shwartz and A. Weiss, 1991. An Analysis of Rollback-Based Simulation, *ACM Transactions on Modeling and Computer Simulation 1:2*, 154–193.
49. V. Madisetti, J. Walrand and D. Messerschmitt, 1990. Synchronization in Message-Passing Computers-Models, Algorithms, and Analysis, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 22 (SCS Simulation Series, January), pp. 35–48.
50. H. Mehl, 1992. A Deterministic Tie-Breaking Scheme for Sequential and Distributed Simulation, in *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, Vol. 24 (SCS Simulation Series, January), pp. 199–200.
51. B.C. Merrifield, S.B. Richardson and J.B.G. Roberts, 1990. Quantitative Studies of Discrete Event Simulation Modelling of Road Traffic, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 22 (SCS Simulation Series, January), pp. 188–193.
52. J. Misra. Distributed-Discrete Event Simulation, 1986, *ACM Computing Surveys 18:1*, 39–65.
53. D. Mitra and I. Mitrani, 1984. Analysis and Optimum Performance of Two Message Passing Parallel Processors Synchronized by Rollback, in *Performance '84*, Elsevier Science Publishers, North Holland, pp. 35–50.
54. H.T. Mouftah and R.P. Sturgeon, 1990. Distributed Discrete Event Simulation for Communication Networks, *IEEE Journal on Selected Areas in Communications 8:9*, 1723–1734.

55. D. NICOL, 1991. Performance Bounds on Parallel Self-initiating Discrete-Event Simulations, *ACM Transactions on Modeling and Computer Simulation 1:1*, 24–50.
56. D. NICOL, A. GREENBERG, B. LUBACHEVSKY and S. ROY, 1992. Massively Parallel Algorithms for Trace-Driven Cache Simulation, in *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, Vol. 24 (SCS Simulation Series, January), pp. 3–11.
57. C.I. PHILLIPS and L.G. Cuthbert, 1991. Concurrent Discrete-Event Simulation Tools, *IEEE Journal on Selected Areas in Communications 9:3*, 477–485.
58. M. PRESLEY, M. EBLING, F. WIELAND and D.R. JEFFERSON, 1989. Benchmarking the Time Warp Operating System with a Computer Network Simulation, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 21 (SCS Simulation Series, March), pp. 8–13.
59. D.A. REED, A.D. MALONY and B.D. MCCREDIE, 1988. Parallel Discrete Event Simulation Using Shared Memory, *IEEE Transactions on Software Engineering 14:4*, 541–553.
60. P.F. REYNOLDS, JR., 1988. A Spectrum of Options for Parallel Simulation, in *1988 Winter Simulation Conference Proceedings* (December), pp. 325–332.
61. D.O. RICH and R.E. MICHELSEN, 1991. An Assessment of the Modsim/TWOS Parallel Simulation Environment, in *1991 Winter Simulation Conference Proceedings* (December), pp. 509–518.
62. R. RICHTER and J.C. WALRAND, 1989. Distributed Simulation of Discrete Event Systems, *Proceedings of the IEEE 77:1*, 99–113.
63. J.W. ROBERTS (ed.), 1992. *Performance Evaluation and Design of Multiservice Networks*, Commission of the European Communities, Luxembourg.
64. L.M. SOKOL, D.P. BRISCOE and A.P. WIELAND, 1988. MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 19 (SCS Simulation Series, July), pp. 34–42.
65. L. SOULÉ, 1992. Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms, Technical Report No. CSL-TR-92-527, Computer Systems Laboratory, Stanford University, Palo Alto, CA.
66. L. SOULÉ and A. GUPTA, 1991. An Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation, *ACM Transactions on Modeling and Computer Simulation 1:4*, 308–347.
67. W.K. SU and C.L. SEITZ, 1989. Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 21 (SCS Simulation Series, March), pp. 38–43.
68. G.S. THOMAS and J. ZAHORJAN, 1991. Parallel Simulation of Performance Petri Nets: Extending the Domain of Parallel Simulation, in *1991 Winter Simulation Conference Proceedings* (December), pp. 564–573.
69. P.A. TINKER and J.R. AGRE, 1989. Object Creation, Messaging, and State Manipulation in an Object Oriented Time Warp System, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 21 (SCS Simulation Series, March), pp. 79–84.
70. J.J. TSAI, S.R. DAS and R.M. FUJIMOTO, 1991. Parallel Execution of Communication Network Simulators, Technical Report GIT-CC-91/48, College of Computing, Georgia Institute of Technology, Atlanta, GA.
71. J.J. TSAI and R.M. FUJIMOTO, 1993. Automatic Parallelization of Discrete Event Simulation Programs, Technical Report GIT-CC-93/19, College of Computing, Georgia Institute of Technology, Atlanta, GA.
72. S. TURNER and M. XU, 1992. Performance Evaluation of the Bounded Time Warp Algorithm, in *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, Vol. 24 (SCS Simulation Series, January), pp. 117–128.
73. W. VERBIEST, L. PINNOO and B. VOETEN, 1988. The Impact of the ATM Concept on Video Coders, *IEEE Journal on Selected Areas in Communications 6:9*, 1623–1631.
74. J. WEST and A. MULLARNEY, 1988. Modsim: A Language for Distributed Simulation, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 19 (SCS Simulation Series, July), pp. 155–159.
75. F. WIELAND, L. HAWLEY, A. FEINBERG, M. DILORENTO, L. BLUME, P. REIHER, B. BECKMAN, P. HONTALAS, S. BELLENOT and D.R. JEFFERSON. 1989. Distributed Combat Simulation and Time Warp: The Model and Its Performance, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 21 (SCS Simulation Series, March), pp. 14–20.
76. F. WIELAND and D.R. JEFFERSON, 1989. Case Studies in Serial and Parallel Simulation, in *Proceedings of the 1989 International Conference on Parallel Processing*, Vol. 3 (August), pp. 255–258.
77. G. ZHANG and B.P. ZEIGLER, 1989. DEVS-Scheme Supported Mapping of Hierarchical Models onto Multiple Processor Systems, in *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 21 (SCS Simulation Series, March), pp. 64–69.