*Proceedings of the 1989 Winter Simulation Conference*
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

# PARALLEL DISCRETE EVENT SIMULATION

Richard M. Fujimoto
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

## ABSTRACT

This tutorial surveys the state of the art in executing discrete event simulation programs on a parallel computer. Specifically, we will focus attention on *asynchronous* simulation programs where few events occur at any single point in simulated time, necessitating the concurrent execution of events occurring at different points in time.

We first describe the parallel discrete event simulation problem, and examine why it so difficult. We review several simulation strategies that have been proposed, and discuss the underlying ideas on which they are based. We critique existing approaches in order to clarify their respective strengths and weaknesses.

## 1 INTRODUCTION

Parallel discrete event simulation (PDES) refers to the execution of a single discrete-event simulation program on a parallel computer. PDES has attracted a considerable amount of interest in recent years. From a pragmatic standpoint, this interest arises from the fact that large simulations in engineering, computer science, economics, and military applications (to mention a few) consume enormous amounts of time on sequential machines. From an academic point of view, parallel simulation is interesting because it represents a problem domain that contains substantial amounts of parallelism, yet paradoxically, is one of the most difficult to parallelize on existing machines. A sufficiently general solution to the PDES problem may lead to new insights in parallel computation as a whole. Historically, parallel discrete event simulation has been identified as an application where vectorization techniques using supercomputer hardware provide little benefit [7].

The simulations of particular interest here are those of *asynchronous* systems. For these systems, simulation techniques based on lock-step execution using a global clock perform poorly because few simulator events occur at any single point in simulated time. As we shall soon see, concurrent execution of events at different points in simulated time introduces interesting synchronization problems that are at the heart of the PDES problem.

This paper deals with the execution of a *single* simulation program on a parallel computer by decomposing the simulation application into a set of concurrently executing processes. For completeness, we close this section by mentioning other approaches to exploiting parallelism in simulation problems. Comfort has proposed using dedicated functional units to implement specific *sequential* simulation functions (e.g., event list manipulation) [12]. This method can provide only a very limited amount of speedup, however.

If the simulation is largely stochastic and one is performing long simulation runs to reduce variance, or if one is attempting to simulate a specific simulation problem across a large number of different parameter settings, an alternative (and probably preferred) approach is to execute independent, *sequential* simulation programs on different processors [5, 24]. This *replicated trials* approach can be very effective, though it is only useful if each processor contains sufficient memory to hold the entire simulation. This approach is less suitable in a design environment, however, where results of one experiment are used to determine the experiment that should be performed next, since one must wait for a *sequential* execution to be completed before results are obtained.

## 2 WHY IS PDES HARD?

Discrete event simulation would initially appear to be an ideal candidate for parallel processing. Not only do many large simulation problems consume enormous amounts of time on sequential machines, but the systems that are typically modeled often contain substantial amounts of intrinsic parallelism. Yet, paradoxically, effectively utilizing parallel computers to speed up large discrete event simulation problems continues to be one of the most challenging problems in parallel computation today.

The reason PDES is so hard becomes evident if one examines the operation of a *sequential* discrete event simulator. Sequential simulators typically utilize two (often global) data structures: the *state variables* that describe the state of the system, and an *event list* containing all pending events that have been scheduled for the simulated time future, but have not yet taken effect. Each event contains a timestamp, and usually denotes some change in the state of the system being simulated (with the timestamp denoting when this change occurs in the actual system). The "main loop" of the simulator repeatedly removes the smallest timestamped event from the event list, and processes that event. Processing an event involves executing some simulator code to effect the appropriate change in state, and *scheduling* zero or more new events to model causality relationships in the system under investigation.

In the above execution paradigm, it is crucial that one always select the smallest timestamped event ($E_{min}$) from the event list as the one to be processed next. If one were to select some other event $E_X$, it would be possible for $E_X$ to modify state variables used by $E_{min}$. This would amount to simulating a system in which the future could affect the past! This is clearly unacceptable; we call errors of this nature *causality errors*.

Let us now consider parallelization of a simulation program that is based on the above paradigm. The greatest opportunity for parallelism arises from processing events

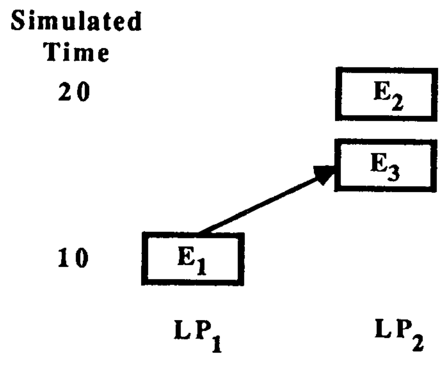

Figure 1: Event $E_1$ affects $E_2$ by scheduling a third event $E_3$ which modifies a state variable used by $E_2$. This necessitates sequential execution of all three events.

concurrently on different processors. However, a direct mapping of this paradigm onto (say) a shared memory multiprocessor quickly runs into difficulty. Consider two events $E_1$ and $E_2$ with timestamps $T_1$ and $T_2$ respectively, such that $T_1 < T_2$. If $E_1$ and $E_2$ both access a common state variable (say $E_1$ writes into the variable and $E_2$ reads it), then $E_1$ and $E_2$ must be executed sequentially ($E_1$ before $E_2$) to be sure no causality error occurs.

Most existing PDES strategies avoid scenarios such as these by mandating that a process-oriented methodology is used which strictly forbids processes to have direct access to shared state variables (an exception is the mechanism described in [17]). The system being modeled, usually referred to as the *physical system*, is viewed as being composed of some number of *physical processes* that interact at various points in simulated time. The simulator is constructed as a set of *logical processes* $LP_0, LP_1, ...LP_{N-1}$, one per physical process. Interactions between physical processes are modeled by time-stamped event messages sent between the corresponding logical processes. Each logical process contains a portion of the state corresponding to the physical process it models. We will later examine the implications of requiring this world view of the simulation.

This view of the simulation as a set of logical processes that communicate by exchanging timestamped messages is used by all of the simulation methods discussed here. Using the logical process paradigm, one can ensure that no causality errors occur if *each* LP processes events in non-decreasing timestamp order. We call this requirement the *local causality constraint*; adherence to this constraint is sufficient, though not necessary, to guarantee that no causality errors occur. It is not necessary because two events within a single LP may be independent of each other, in which case processing them out of timestamp sequence does not lead to causality errors.

Although the logical process paradigm avoids many types of causality error, it does not prevent others. Consider two events, $E_1$ at logical process $LP_1$ with timestamp 10, and $E_2$ at $LP_2$ with timestamp 20 (see figure 1; it is convenient to depict these situations using a space-time graph). If $E_1$ schedules a new event $E_3$ for $LP_2$ which contains a timestamp less than 20, then $E_3$ could affect $E_2$, necessitating sequential execution of all three events. If one had no information regarding what events could be scheduled by what other events, one would be forced to conclude that the only event that is "safe" to process is the one containing the smallest timestamp, leading to a purely sequential execution.

Consider this situation from the perspective of the physical system. If the activity in the physical system corresponding to $E_1$ has any effect, either direct or indirect, on the activity that is modeled by $E_2$, then there is a cause-and-effect relationship between these activities. The laws that govern the way our universe works dictate that $E_1$'s activity must occur before that of $E_2$'s. In the simulator, these cause-and-effect relationships correspond to data dependence relationships (i.e., one computation modifies one or more state variables that are used by another) that dictate that one computation must precede the other for the computation to be correct. In other words, the constraints that dictate which events must be processed before which other events are dictated by the behavior of the physical system itself, and no amount of "massaging" of the simulation code (short of reformulation of the model) can change this fact.

Now comes the hard part. Operationally, we must decide whether or not $E_1$ can be executed concurrently with $E_2$. But, how do we know whether or not $E_1$ affects $E_2$ without actually performing the simulation for $E_1$? This is the fundamental dilemma PDES strategies must address. The scenario in which $E_1$ affects $E_2$ can be very complex, and is critically dependent on the timestamp of events affected by $E_1$.

PDES is hard because the precedence constraints that dictate which computations must be executed before which others is, in general, quite complex and highly data dependent. This contrasts sharply with other areas in which parallel computation has had a great deal of success, e.g., vector operations on large matrices of data, where much is known about the structure of the computation at compile time. Thus it is not too surprising that a general solution to the parallel simulation problem has been elusive.

PDES mechanisms broadly fall into two categories: *conservative* and *optimistic*. Conservative approaches strictly *avoid* the possibility of any causality error ever occurring. These approaches rely on some strategy to determine when it is "safe" to process an event, i.e., they must determine when all events that could affect the event in question have been processed. On the other hand, optimistic approaches use a *detection and recovery* approach: causality errors are detected, and a *rollback* mechanism is invoked to recover. We will now describe some of the details and underlying concepts behind several conservative and optimistic simulation mechanisms that have been proposed. First, however, we will make a brief digression to discuss the implications of forcing the computation to be written without the use of shared variables.

Throughout this paper, we assume that the simulation consists of $N$ logical processes, $LP_0...LP_{N-1}$. $Clock_i$ refers to the simulated time up to which $LP_i$ has progressed: when an event is processed, the process's clock is automatically advanced to the timestamp of that event. If $LP_i$ may send a message to $LP_j$ during the simulation, we say a *link* exists from $LP_i$ to $LP_j$.

## 3 LOGICAL PROCESSES, REVISITED

The logical process methodology requires application programmers to statically partition the simulator's state variables into a set of disjoint *states*, and ensure that no simulator event accesses more than one state. It is appropriate to ask if this is a natural way to program simulations.

The exclusion of shared variables may or may not be burdensome, depending on the application. For example, it is usually *not* a severe restriction for a queueing network simulation. Here, it is natural to create on logical process for each server. Because the behavior of one server is independent of the state of the others, exclusion of shared variables does not create any problem.

On the other hand, consider the simulation of a pool table with some number of pool balls randomly moving across its surface, and occasionally colliding with each other and/or the sides of the table [3]. This simulation is similar to that found in many military applications [20, 47]. To avoid expensive global searches, the pool table is usually partitioned into a two-dimensional grid with each "grid sector process" containing state information indicating which pool balls currently reside in that grid sector, as well as their velocities, directions, etc. Because pool balls may simultaneously reside in several grids, processes modeling these entities must access state that resides in several distinct grid processes. Where a sequential simulation would simply use memory references to global state variables, the distributed simulator must invoke expensive message passing mechanisms to duplicate the necessary state information in the appropriate processes. This can lead to substantial performance degradations, and may significantly complicate the coding of the simulation model. The situation becomes even worse in combat simulations where some simulation entities, e.g., aircraft, can "see" into many grid sectors at a single point in simulated time. It is noteworthy that such performance degradations will not be apparent in speedup curves if the parallel simulation model executing on a sequential simulator is used as the basis for computing speedup.

## 4 CONSERVATIVE MECHANISMS

Historically, the first distributed simulation mechanisms were based on conservative approaches. As discussed earlier, the basic problem conservative mechanisms must solve is to determine when it is "safe" to process an event. More precisely, if a process contains an unprocessed event $E_1$ with timestamp $T_1$ (and no other with smaller timestamp), and that process can determine that it is impossible for it to later receive another event with timestamp smaller than $T_1$, then it can guarantee that local causality will be preserved, so it may safely process $E_1$. Processes containing no "safe" events must block; this can lead to deadlock situations if appropriate precautions are not taken.

### 4.1 Deadlock Avoidance

Independently, Chandy and Misra[10], and Bryant[6] developed some of the first PDES algorithms. In order to determine when it is safe to process a message, it is required that the sequence of timestamps on messages sent from one process to another is non-decreasing. This guarantees that the timestamp of the last message received on an incoming link is a lower bound on the timestamp of any subsequent message that will be later received.

Messages arriving on each incoming link are stored in FIFO order, which is also timestamp order because of the above restriction. Each link has a clock associated with it that is equal to the timestamp of the message at the front of that link's queue, if the queue contains a message, or the timestamp of the last received message, if the queue is empty. The process repeatedly selects the link with the smallest clock and, if there is a message in that queue's link, processes it. If the selected queue is empty, the process blocks. This protocol guarantees that each process will only process events in non-decreasing timestamp order, thereby ensuring adherence to the local causality constraint.

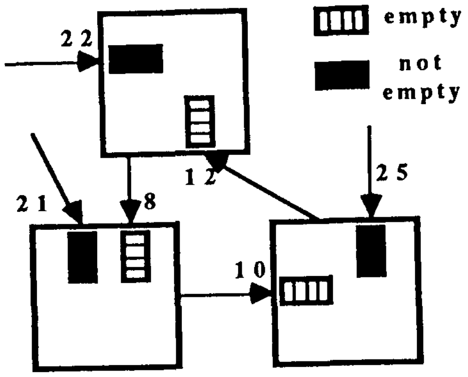

Figure 2: Deadlock situation.

If a cycle of empty queues arises that has sufficiently small clock values, each process in that cycle must block, and the simulation deadlocks. Figure 2 shows one such deadlock situation. In general, if there are relatively few unprocessed event messages compared to the number of links in the network, or if the unprocessed events become clustered in one portion of the network, deadlock may occur very frequently.

*Null* messages are used to avoid deadlock situations. Null messages are used only for synchronization purposes, and do not correspond to any simulation activity. The clock value of each incoming link provides a lower bound on the timestamp of the next unprocessed event that will be removed from that link's buffer. When coupled with knowledge of the simulation performed by the process, this incoming bound can be used to determine a lower bound on the timestamp of the next *outgoing* message on each output link. When a process blocks, it sends a null message on each of its output ports indicating this bound; the receiver of the null message can then compute new bounds on its outgoing links, send this information on to its neighbors, and so on. It can be shown that this mechanism avoids deadlock as long as one does not have any cycles in which the collective timestamp increment around this cycle is zero. A necessary and sufficient condition for deadlock using this scheme is that a cycle of links must exist with the same link clock time [37].

An alternative to sending null messages whenever a process blocks is to have processes query other processes when they need to receive a better link clock value [2, 33]. This helps to reduce the amount of null message traffic.

### 4.2 Deadlock Detection and Recovery

Chandy and Misra also developed an alternative approach to parallel simulation that eliminates the use of null messages. The mechanism is similar to that described above, except no null messages are created when a process blocks. A separate mechanism is used to detect deadlock situations, and still another mechanism is used to break the deadlock. Deadlock detection mechanisms are described in [13, 21, 33]. The deadlock can be broken by observing that the smallest timestamped message in the entire simulation is always safe to process. Alternatively, one may use a distributed computation to compute lower bound information (not unlike the distributed computation using null messages described above) to enlarge the set of safe messages. Unlike the deadlock avoidance approach, this mechanism does not prohibit cycles of zero timestamp increment, though performance may be poor if many such cycles exist.

Several other conservative approaches to parallel simulation have been developed [1, 8, 22, 23, 30, 37, 42]. The key ideas used by these mechanisms are described next.

### 4.3 Synchronous Operation

Several researchers have proposed synchronous algorithms in which one iteratively determines which events are safe to process, and then processes them. Barrier synchronizations are used to keep one iteration (or components of a single iteration) from interfering with each other. Because barrier synchronizations are necessary, these algorithms are best suited for shared memory machines in order to keep the associated overheads to a minimum.

It is instructive to compare the synchronous style of execution with the deadlock detection and recovery approach described earlier. Both share the characteristic that the simulation moves through phases of (1) processing events, and (2) performing some global synchronization function. Deadlock recovery is similar to the overhead function of the synchronous methods in that one attempts to determine which events are safe to process.

In the best case, the detection and recovery strategy will never deadlock, virtually eliminating clock synchronization overhead completely. In contrast, synchronous methods will continually block and restart throughout the simulation. On the other hand, the synchronous methods do not require a deadlock detection mechanism, though deadlock detection is trivial on shared-memory machines (a counter indicating the number of non-blocked processes is sufficient). One disadvantage of the detection and recovery method is that the period leading up to a deadlock may contain very little parallelism. Such behavior can lead to limited speedup in accordance with Amdahl's law which states that no more than $k$ fold speedup is possible if $1/kth$ of the computation is sequential. Synchronous methods have some control over the amount of computation that is performed during each iteration, so, at least in principle, they offer some advantage here.

The feature that separates different synchronous approaches is principally the method used to determine which events are safe to process. We discuss ideas that have been introduced to streamline this process below. A common thread that runs through many techniques is the minimum timestamp increment function used in the original deadlock avoidance approach. A simple extension of this concept leads to the *distance* between processes; distance provides a lower bound on the amount of simulated time that must expire for an unprocessed event on one process to propagate (and possibly affect) another event.

### 4.4 Conservative Time Windows

Lubachevsky uses a moving simulated time window to reduce the overhead associated with determining when it is safe to process an event [30]. The lower edge of the window is defined as the minimum timestamp of any unprocessed event. Only those unprocessed events whose timestamp resides within the window are eligible for processing.

The purpose of the window is to reduce the "distance" one must search in determining if an event with smaller timestamp will later be received. For example, if the window extends from simulated time 10 to time 20, and the application is such that each event processed by an LP generates a new event with a minimum timestamp increment of 8 units of simulated time, then each LP need only examine the unprocessed events in neighboring LPs to determine which events are safe to proceed. No unprocessed event two or more hops away can affect one in the 10 to 20 time window because such an event would have to have a timestamp earlier than the start of the window.

Of course, one must not make the window so small that it contains few unprocessed events. Similarly, if the window is too large, it does little to improve efficiency. Setting the window to an appropriate size requires application specific information that must be obtained either from the programmer or from monitoring the simulation.

### 4.5 Improving Lookahead

Lookahead refers to the ability to predict what will happen, or equally important for conservative methods, what will not happen, in the simulated time future based on knowledge of the application, events that have already been processed, and pending events waiting to be processed. Non-zero minimum timestamp increments are the most obvious form of lookahead and were essential for the deadlock avoidance approach to make progress. Because lookahead enhances one's ability to identify events that are safe to process, it is reasonable to expect that improving a process's lookahead ability is bound to improve performance.

Nicol proposes improving the lookahead ability of processes by precomputing portions of the computation for future events [36]. For example, in a queueing network simulation without preemption, one can precompute the service time of jobs that have not yet been received. If the server process is idle and its clock has a value of 100, and the service time of the next job has been precomputed to be 50, then the lower bound on the timestamp of the next message it will send is 150 rather than 100. If the average service time is much larger than the minimum, then this will provide a better lower bound on the timestamp of the next message.

Interestingly, the ability to use precomputation to improve lookahead *itself* requires lookahead ability. Precomputation is possible if one can predict aspects of future event computations without knowledge of the event message that causes that computation, or the state of the process when the event constructs the message timestamp. For example, if the service time depends on a parameter in the message (e.g., a message length for a communication network simulation), precomputation would not be possible. Nevertheless, precomputation appears to be a useful technique when it can be applied.

## 4.6 Conditional Knowledge

Chandy and Sherman propose a paradigm that combines mechanisms used in sequential simulations with conservative mechanisms [8]. In a sequential simulation, one often schedules an event (e.g., a job departure from a queueing network server) under the premise that this event will take place if no disruptive event (e.g., a job preemption) occurs first. Such events are referred to as *conditional events.*

All conservative approaches convert conditional events to definite events (events that are guaranteed to occur) before they can be processed. This is accomplished in sequential simulations by virtue of the fact that no events exist in the event list with smaller timestamp than the conditional event when that event is processed. Like other conservative mechanisms, a protocol is required to determine when it is "safe" to process conditional events. Chandy and Sherman propose both synchronous and asynchronous protocols to perform this task; these protocols use broadcasts to distribute "time of next event" information in order to avoid deadlock situations [8]. The conditional knowledge approach to simulation arises from the Unity theory of parallel programming [11]. An alternative approach, also based on Unity, is described in [9].

## 4.7 Conservative Performance

A substantial amount of work has been completed to evaluate the performance of the deadlock avoidance and deadlock detection and recovery algorithms. Reed, Malony, and McCredie performed simulations of queueing networks, and report disappointing performance [39]. Fujimoto demonstrated that performance of these algorithms is critically related to the degree to which logical processes can look ahead into the simulated time future [15]; processes must exhibit good lookahead characteristics, and be programmed to exploit this lookahead, or else performance will be poor. Fujimoto reproduced the poor performance reported by Reed, and showed that reprogramming processes to exploit lookahead yielded dramatic improvements in performance for networks using first-come-first-serve queues. These techniques are not generally applicable to other disciplines, however. Su and Seitz report some success in using variations of these algorithms to speed up logic simulations [44]. Reed et al., Fujimoto, and Wagner, Lazowska and Bershad[46] exploit techniques using shared memory to improve the efficiency of these algorithms.

Wagner and Lazowska[45], Lin and Lazowska[28], and Nicol[35] examined lookahead analytically. Lubachevsky has examined the performance and scalability of the bounded lag approach that uses synchronous execution, lookahead, and time windows to improve performance [30, 31]. Specifically, he uses two forms of lookahead: minimum timestamp increments to allow *idle* logical processes to lookahead, and "opaque" periods that allow certain *busy* processes to do the same. The latter requires the exclusion of (for example) preemptive behavior. Lubachevsky argues that performance of this approach scales as the problem and machine size increase in proportion to within a factor of $O(logN)$ of ideal, assuming adequate lookahead is available. Ayani [1] and Chandy and Sherman [8] also report some success in speeding up queueing network simulations using their approches.

Much has been learned concerning the performance of conservative mechanisms. As noted in our introduction to the PDES problem, all conservative mechanisms rely on the ability to predict the future in order to ascertain which events may be safely processed. If no such capability existed, one could not guarantee the safety of any event other than the one containing the (globally) smallest timestamp, forcing sequential execution (ignoring the case of two events on distinct processes having identical timestamps).

In order to achieve good performance, conservative mechanisms must be adept at predicting what will *not* happen because it is the fact that "*no* smaller timestamped event will later be received" that is the firing condition that allows an event to be safely processed. Various forms of information are used to predict what will not occur, including:

1. the structure of the network of logical processes, i.e., which processes can send messages to which other processes. This structure restricts the paths "dangerous" events may use to reach others; Sparsely connected networks present the best case for conservative mechanisms.
2. received event messages. Assuming messages are transmitted in timestamp order, each received message excludes the possibility of later messages containing a smaller timestamp. Conservative mechanisms usually work best when there are many unprocessed events relative to the connectivity of the network (i.e., the number of processes), and these events are uniformly distributed among the links.
3. knowledge of logical process behavior. Here, the characteristic that one looks for is lookahead, i.e., a *guaranteed invariance in behavior* in the physical system, regardless of any new events that might later occur. For example, new jobs arriving at a non-preemptable queue do not affect the behavior of the job that is currently receiving service. Similarly, a minimum timestamp increment for some logical process is derived from the observation that the corresponding physical process will not perturb the system in any way up to some guaranteed time in the future, regardless of what happens next. We shall return to this property later.

Any given simulation application may exhibit favorable, or unfavorable characteristics for each of these properties. It appears that the inability to effectively exploit *any* of these aspects of behavior is fatal to existing conservative approaches. Conversely, most approaches can obtain good speedup if all of these properties appear in a favorable manner. Depending on the specifics of the strategy in question, the inability to exploit one or more of these aspects may or may not be fatal. It remains to be seen to what extent applications that arise in practice exhibit these properties.

## 4.8 Critique of Conservative Mechanisms

Perhaps the most obvious drawback of conservative approaches is that they cannot fully exploit the parallelism available in the simulation application. If it is possible that event $E_A$ *might* affect $E_B$ either directly or indirectly, conservative approaches must execute $E_A$ and $E_B$ sequentially. If the simulation is such that $E_A$ seldom affects $E_B$, these events could have been processed concurrently most of the time. In general, if the worst case scenario for determining when it is safe to proceed is far from the typical scenario that arises in practice, the conservative approach will usually be overly pessimistic, and force sequentiality when it is not necessary.

A related problem faced by conservative methods concerns the question of robustness; it has been observed that seemingly minor changes to the application may have a catastrophic effect on performance. For example, adding short, high priority messages that interrupt "normal" processing in a computer network simulation may lead to severe performance degradations. This is problematic because experimenters often do not have advance knowledge of the full range of experiments that will be required, so it behooves them to invest substantial amounts of time parallelizing the application if an unforeseen addition to the model at some future date could invalidate all of this work.

Critics of conservative methods are also quick to point out that many existing conservative techniques (the deadlock avoidance and deadlock detection and recovery mechanisms in particular) require static configurations: one cannot dynamically create new processes, and the interconnection among logical processes must also be statically defined. Techniques to circumvent this problem, e.g. to create "spare" processes at the start of the simulation and to define a fully connected network, usually lead to excessive overheads, e.g., broadcast communications may be required to determine when it is safe to proceed.

Many conservative schemes require knowledge concerning logical process behavior to achieve good performance. Information such as minimum timestamp increments or the guarantee that an event occurring at time $T$ really has no effect on the behavior of certain other events may be difficult to derive for complex simulations. Users would be ill-advised to give overly conservative estimates (e.g., a minimum timestamp increment of zero) because very poor performance may result.

Proponents of optimistic approaches argue that the user should not have to be concerned with the details of the synchronization mechanism in order to achieve good performance. Sequential simulation programs need not be concerned with the details of the implementation of the event list (Actually, some sequential simulation programs rely on the fact that events with the same timestamp are processed in the order in which they are inserted into the event list; many regard this as an unwise design practice, however.). Of course, certain guidelines that apply to *all* parallel programs must be followed when developing parallel simulation code, e.g., selecting an appropriate granularity and maximizing parallelism, but requiring the programmer to also be intimately familiar with the synchronization mechanism and program the application to maximize its effectiveness will often lead to "fragile" code that is difficult to modify and maintain.

## 5 OPTIMISTIC MECHANISMS

Optimistic methods detect and recover from causality errors rather than strictly avoid them. In contrast to conservative mechanisms, optimistic strategies need not determine when it is safe to proceed; instead they need to determine when an error has occurred, and how to recover. One advantage of this approach is that it allows the simulator to exploit parallelism in situations where it is possible causality errors *might* occur, but in fact don't.

The Time Warp mechanism, based on the Virtual Time paradigm, is the most well known optimistic protocol [26]. Here, virtual time is synonymous with simulated time. In Time Warp, a causality error is detected whenever an event message is received that contains a timestamp smaller than that of the process's clock (i.e., the timestamp of the last processed message). The event causing rollback is called a *straggler*. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the straggler (more precisely, those processed events that have timestamps larger than that of the straggler).

An event may do two things that have to be rolled back: it may modify the state of the logical process, and/or it may send event messages to other processes. Rolling back the state is accomplished by periodically saving the process's state, and restoring an old state vector on rollback. "Unsending" a previously sent message is accomplished by sending an *anti-message* that annihilates the original when it reaches its destination. If the annihilated positive message has already been processed, then that process must also be rolled back to undo the effect of processing the message. Recursively repeating this procedure allows all of the effects of the erroneous computation to eventually be cancelled. It can be shown that this mechanism always makes progress under some mild constraints.

As noted earlier, the smallest timestamped, unprocessed event in the simulation will always be safe to process. In Time Warp, the timestamp on this event is called global virtual time (GVT). No event with timestamp smaller than GVT will ever be rolled back, so storage used by such events (e.g., saved states) can be discarded. Also, irrevokable operations (such as I/O) cannot be committed until GVT sweeps past the simulated time at which the operation occurs. The process of reclaiming memory and committing irrevokable operations is referred to as *fossil collection*.

### 5.1 Lazy Cancellation

Some optimizations have been proposed to "repair" the damage caused by an incorrect computation rather than completely repeat it. For instance, it may be the case that a straggler event does not sufficiently alter the computation of rolled back events to change the (positive) messages generated by these events. The Time Warp mechanism described above uses *aggressive* cancellation, i.e., whenever a process rolls back to time $T$, anti-messages are immediately sent for all positive messages sent after simulated time $T$. In lazy cancellation [19], processes do not immediately send the anti-messages for any rolled back computation. Instead, they wait to see if the reexecution of the computation regenerates the *same* messages; if the same message is recreated, there is no need to cancel the message. An anti-message created at simulated time $T$ is only sent after the process's clock sweeps past time $T$ without regenerating the same message.

Depending on the application, lazy cancellation may improve or degrade performance. It requires some additional overhead whenever an event is executed to determine if a matching anti-message already exists; one or more message comparisons will be required if one is reexecuting previously rolled back events. Also, lazy cancellation may allow erroneous computations to spread further than they would under aggressive cancellation, so performance may be degraded if the simulator is forced to execute many incorrect computations. One can construct cases where lazy cancellation executes a computation with $N$ fold parallelism $N$ times *slower* than aggressive when $N$ processors are used [40].

On the other hand, lazy cancellation has the interesting property that it can allow the computations to be executed in less time than the critical path execution time [4]. The explanation for this phenomenon is that computations with incorrect input may still generate correct results! Therefore, one may execute some computations prematurely, yet still generate the correct answer. This is not possible using aggressive cancellation because rolled back computations are immediately discarded, even if they did generate the correct result. One can construct a case where lazy cancellation can execute a sequential computation with $N$ fold speedup using $N$ processors, while aggressive cancellation requires the critical path length execution time [40].

Although it is instructive to construct best and worst case behaviors for lazy and aggressive cancellation, it is not clear that such extreme behaviors arise in practice. Some evidence exists that lazy cancellation tends to perform as well as, or better than, aggressive cancellation in situations that commonly arise in practice.

## 5.2 Jump Forward

The *jump forward* optimization is somewhat similar to lazy cancellation, but deals with state vectors rather than messages. Consider the case where the state of the process is the same after processing a straggler event message as it was before it executed. If no new messages arrived, then it is clear that the reexecution of rolled back events will be identical to the original execution, so one need not reexecute them, but instead, jump forward over these events. This requires a comparison of state vectors to see if the state has changed.

The utility of the jump forward optimization is not clear. The place where one could derive significant benefit from jump forward is "read-only" or query events (Sokol also calls these non-side affecting events [43]). However, it has been argued that query events should not be provided anyway because they encourage users to construct programs as if shared state variables were permitted. This inevitably leads to poor performance because query events entail a significant overhead to ensure correct synchronization, even if the hardware supports shared memory. Finally, it is worth mentioning that jump forward may significantly complicate the Time Warp code, detracting from its maintainability.

## 5.3 Relationship to Lookahead

As mentioned earlier, the fundamental aspect of the computation that is exploited by lazy cancellation and jump forward is an invariance in the behavior of events in the simulated future to straggler events. This is closely related to the lookahead property that is used extensively by conservative approaches.

In any discrete event simulation, if a process at simulated time $T$ can predict with absolute certainty that some event will occur at simulated time $T+L$ (where $L$ is less than or equal to the process's lookahead), then it can immediately schedule that event. "Absolute certainty" means the computation that produces the event at $T+L$ is invariant to any new events that occur in the interval $[T, T+L]$. Thus, we see that the invariance property that lazy cancellation and jump forward attempt to exploit is essentially the same as the lookahead property used in conservative mechanisms.

The advantage offered by optimistic methods is that unlike conservative approaches that require lookahead to be explicitly programmed into the application, the optimistic approach exploits lookahead in a way that is *transparent* to the application program. For example, suppose the application in the example described above were *not* programmed to exploit lookahead, but instead chose to wait until time $T + L$ to schedule the event. The process would normally accomplish this by sending a message to itself with timestamp $T + L$, and have this "self" event generate the desired message at time "now," i.e., $T + L$. Suppose this self message were executed prematurely. Despite the fact that it is premature, it will generate the correct message because the computation that generates the second message at $T + L$ is (by assumption) invariant to stragglers with timestamps between $T$ and $T + L$. So, if such stragglers do arrive, lazy cancellation will succeed because the self event will recreate the same event message that it created during the premature execution.

The disadvantage of exploiting lookahead in this way (as opposed to specifying it explicitly) is that the overheads are greater. For lazy cancellation, the invariant computation (the self event in the above example) must be reexecuted, and message comparisons are required to determine when the optimization is applicable. Similarly, jump forward requires comparisons of state vectors. Explicitly programming lookahead into the application has its advantages.

The place where using the lazy cancellation approach does pay off is when invariance (i.e., lookahead) cannot be statically guaranteed, but dynamically is usually available. For example, this will be the case in a queueing network where preemption is possible, but seldom occurs because there are few high priority jobs. Here, the application cannot be programmed to exploit lookahead. However, lazy cancellation will still be able to exploit it whenever it is available, i.e., whenever no preemption actually occurs.

## 5.4 Optimistic Time Windows

Time windows, not unlike those proposed for conservative mechanisms, have also been proposed for optimistic protocols [43]. Only events within the time window are eligible for processing. In optimistic methods, the time window is used to prevent incorrect computations from propagating too far ahead into the simulated time future.

The utility of time windows in optimistic mechanisms is also a point of debate. Critics of this method point out that such windows cannot distinguish good computations from bad ones, so they may impede the progress of correct computations. Further, incorrect computations that are far ahead in the future are already discriminated against by Time Warp's scheduling mechanism which gives precedence to activities containing small timestamps. Finally, it is not clear how the size of the window should be determined. Empirical data collected by researchers at JPL suggest that time windows offer only limited advantage [41].

## 5.5 Wolf Calls

Madisetti, Walrand, and Messerschmitt propose a mechanism in WOLF whereby a straggler message causes a process to send special control messages to stop the spread of the erroneous computation [32]. Like the time window scheme, the disadvantage of this approach is that some correct computations may be unnecessarily frozen. Also, the overhead to implement this mechanism becomes excessive in certain applications because many control messages will be required.

### 5.6 Direct Cancellation

Fujimoto proposes a mechanism that uses shared memory to streamline the cancellation of incorrect computations [16]. Whenever an event $E_1$ schedules another event $E_2$, a pointer is left from $E_1$ to $E_2$. This pointer is used if it is later decided that $E_2$ should be cancelled (using either lazy or aggressive cancellation). By contrast, conventional Time Warp systems must search to locate cancelled messages. The advantages of this mechanism are two fold: it reduces the overheads associated with message cancellation, and it speedily tracks down erroneous computations to minimize the damage that is caused. Good performance has been reported on a version of Time Warp that uses direct cancellation [16].

### 5.7 Optimistic Performance

Several successes have been reported in using Time Warp to speed up real world simulation problems. Impressive speedups have been reported by researchers at JPL in simulations of battlefield scenarios [47], communication networks [38], biological systems [14], and simulations of other physical phenomena [25].

Fujimoto has obtained significant speedups on a wide range of queueing networks (as high as 56 using 64 processors), and presents data that indicate that Time Warp far outperforms the deadlock avoidance and deadlock detection and recovery approaches for many networks, even ones that exploit lookahead. The performance differential is particularly dramatic for networks containing preemption [16].

A limited amount of work has been performed in deriving analytic models for Time Warp behavior. Models for the case of two processors have been developed by Lavenberg and Muntz [27] and Mitra and Mitrani [34]. Unfortunately, these models do not generalize to more processors.

Lin and Lazowska derive a condition under which Time Warp will produce optimal performance (i.e., corresponding to the critical path lower bound) [29]. They also identify situations where Time Warp will outperform the Chandy-Misra algorithms. This work assumes that overheads for both Time Warp and the conservative mechanisms are negligible, so the results are largely applicable to the case of large grained events.

### 5.8 Critique of Optimistic Methods

A critical question faced by optimistic systems such as Time Warp is whether the system will exhibit thrashing behavior where most of its time is spent executing incorrect computations and rolling them back. Thus far, the experience of researchers at JPL and Georgia Tech has been that such behavior is seldom encountered in practice, and, when discovered, usually points to a correctable weakness in the implementation rather than any fundamental flaw in the algorithm. Critics argue, however, that no proof yet exists that Time Warp is stable.

An intuitive explanation as to why empirical data suggest stable behavior is that erroneous computations can only be initiated when one processes a correct event prematurely; this premature event, and subsequent erroneous computations, must necessarily be in the simulated time future of the correct, straggler computation. Also, the further the incorrect computation spreads, the further it moves into the simulated time future, lowering its priority for execution since preference is always given to computations containing smaller timestamps. Thus, Time Warp systems tend to automatically slow the propagation of errors, allowing the error detection and correction mechanism to correct the mistake before too much damage has been done. A potentially more dangerous case is when the erroneous computation propagates with smaller timestamp increments than the correct one. It remains to be seen, however, to what extent this behavior can degrade performance, or if such pathological situations arise in practice.

A more serious problem with the Time Warp mechanism is the need to periodically save the state of each logical process. Fujimoto has demonstrated that state saving overhead can seriously degrade performance of many Time Warp programs, even if the state vector is only a few thousand bytes [16]. The state saving problem is further exasperated by applications requiring dynamic memory allocation because one may have to traverse complex data structures to save the process's state. State saving overhead limits the effectiveness of Time Warp to applications where the amount of computation required to process an event can be made significantly larger than the cost of saving a state vector. This may be difficult to achieve for certain applications. A more general solution is to use hardware support [17, 18]. Even ardent supporters of optimism concede that hardware support will probably be required to exploit *fine* grain parallelism.

Optimistic algorithms tend to use much more memory than their conservative counterparts. Though the space-time tradeoffs for optimistic systems are not yet understood, this appears to be an unavoidable aspect of optimism.

Finally, unlike conservative approaches, optimistic systems need to be able to recover from arbitrary errors that can arise because such errors may be erased by a subsequent rollback. Erroneous computations may enter infinite loops, requiring the Time Warp executive to interact with the hardware's interrupt system. In certain languages, pointers may be manipulated in arbitrary ways; Time Warp must be able to trap illegal pointer usages that result in runtime errors, and prevent incorrect computations from overwriting non-state saved areas of memory. Although such problems are, in principal, not insurmountable, they may be difficult to circumvent in certain systems without appropriate hardware support. The alternative taken by most existing Time Warp systems is to leave the onerous task of analyzing incorrect execution sequences to the user.

## 6 CONCLUSIONS

In this paper we have attempted to provide insight into the problem of executing discrete event simulation programs on a parallel computer. We have surveyed existing approaches and analyzed the merits and drawbacks of various techniques. The state of the art in PDES has advanced rapidly in recent years, and much more is now known about the behavior of proposed simulation mechanisms than a few years ago.

Optimistic methods such as Time Warp offer the greatest hope as a "general purpose" simulation mechanism, at least in simulating systems that contain some degree of parallelism. Significant successes have been achieved across a wide range of applications.

Conservative methods offer good potential for certain classes of problems. Significant successes have also been obtained, particularly when application specific knowledge is applied to maximize the efficiency of the simulation mechanism. Conservative methods may find success in packaged simulation systems (e.g., logic simulators) in which the simulation code is optimized for the synchronization algorithm and users only configure the provided simulation modules into specific systems.

An important application area that has not yet been adequately addressed by either optimistic or conservative simulation mechanisms is real time applications. Theories of performance are not sufficiently developed to address this question, though some progress has been made.

## ACKNOWLEDGEMENTS

This work was supported by ONR Contract Number 00014-87-K-0184 and NSF grant DCR-8504826. The author also thanks David Jefferson and members of the Time Warp group at the Jet Propulsion Laboratory for valuable discussions contributing to the development of this paper.

## AUTHOR BIOGRAPHY

RICHARD FUJIMOTO is an associate professor in the School of Computer and Information Science at the Georgia Institute of Technology. He received BS degrees in Computer Science (1977) and Computer Engineering (1978) from the University of Illinois in Urbana-Champaign, and MS (1980) and Ph.D. (1983) degrees from the University of California in Berkeley. Prior to joining Georgia Tech, he has worked for IBM, Hewlett Packard, and the University of Utah. His current research interests are in computer architecture, parallel processing, and simulation. He is a member of ACM, IEEE, and SCS.

Richard M. Fujimoto
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

## References

[1] R. Ayani. A Parallel Simulation Scheme Based on the Distance Between Objects. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.

[2] W. L. Bain and D. S. Scott. An Algorithm for Time Synchronization in Distributed Discrete Event Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3), July 1988.

[3] B. Beckman et al. Distributed Simulation and Time Warp: Part 1: Design of Colliding Pucks. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3):56–60, July 1988.

[4] O. Berry. *Performance Evaluation of the Time Warp Distributed Simulation Mechanism.* Technical Report, University of Southern California, May 1986.

[5] W. Biles et al. Statistical Considerations in Simulation on a Network of Microcomputers. *1985 Winter Simulation Conference Proceedings*, 388–393, December 1985.

[6] R. E. Bryant. Simulation of Packet Communication Architecture Computer Systems. MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[7] A. Chandak and J. C. Browne. Vectorization of Discrete Event Simulation. *Proceedings of the 1983 International Conference on Parallel Processing*, 359–361, August 1983.

[8] K. M. Chandy and R. Sherman. The Conditional Event Approach to Distributed Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.

[9] K. M. Chandy and R. Sherman. Space, Time, and Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.

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

[11] K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison-Wesley, 1988.

[12] J. C. Comfort. The Simulation of a Master-Slave Event Set Processor. *Simulation*, 42(3):117–124, March 1984.

[13] E. W. Dijkstra and C.S. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[14] M. Ebling et al. An Ant Foraging Model Implemented on the Time Warp Operating System. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.

[15] R. M. Fujimoto. *Performance Measurements of Distributed Simulation Strategies.* Technical Report UU-CS-TR-87-026a, Dept. of Computer Science, University of Utah, Salt Lake City, November 1987.

[16] R. M. Fujimoto. Time Warp on a Shared Memory Multiprocessor. *Proceedings of the 1989 International Conference on Parallel Processing*, August 1989.

[17] R. M. Fujimoto. The Virtual Time Machine. *International Symposium on Parallel Algorithms and Architectures*, June 1989.

[18] R. M. Fujimoto, J. Tsai, and G. Gopalakrishnan. Design and Performance of Special Purpose Hardware for Time Warp. *Proceedings of the 15th Annual Symposium on Computer Architecture*, June 1988.

[19] A. Gafni. Rollback Mechanisms for Optimistic Distributed Simulation Systems. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3):61–67, July 1988.

[20] J. B. Gilmer. An Assessment of Time Warp Parallel Discrete Event Simulation Algorithm Performance. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3):45–49, July 1988.

[21] B. Groselj and C. Tropper. A Deadlock Resolution Scheme for Distributed Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.

[22] B. Groselj and C. Tropper. Pseudosimulation: An Algorithm for Distributed Simulation with Limited Memory. *International Journal of Parallel Programming*, October 1987.

[23] B. Groselj and C. Tropper. The Time of Next Event Algorithm. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3), July 1988.

[24] P. Heidelberger. Statistical Analysis of Parallel Simulations. *1986 Winter Simulation Conference Proceedings*, 290–295, December 1986.

[25] P. Hontalas et al. Performance of the Colliding Pucks Simulation on the Time Warp Operating System. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.

[26] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[27] S. Lavenberg and R. Muntz. Performance Analysis of a Rollback Method for Distributed Simulation. In *Performance '83*, Elsevier Science, North Holland, 1983.

[28] Y. Lin and E. Lazowska. *Exploiting Lookahead in Distributed/Parallel Simulation.* Technical Report, Dept. of Computer Science, University of Washington, Seattle, Washington, 1989.

[29] Y. Lin and E. Lazowska. *Optimal Performance of Time Warp Simulation and a Comparison with Chandy Misra Approach.* Technical Report, Dept. of Computer Science, University of Washington, Seattle, Washington, 1989.

[30] B. Lubachevsky. Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks. *Communications of the ACM*, 32(1):111–123, January 1989.

[31] B. Lubachevsky. Scalability of the Bounded Lag Distributed Discrete Event Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.

[32] V. Madisetti, J. Walrand, and D. Messerschmitt. WOLF: A Rollback Algorithm for Optimistic Distributed Simulation Systems. *1988 Winter Simulation Conference Proceedings*, December 1988.

[33] J. Misra. Distributed-Discrete Event Simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[34] D. Mitra and I. Mitrani. Analysis and Optimum Performance of Two Message Passing Parallel Processors Synchronized by Rollback. In *Performance '84*, Elsevier Science, North Holland, 1984.

[35] D. M. Nicol. *The Cost of Conservative Synchronization in Parallel Discrete Event Simulations.* Technical Report, Department of Computer Science, College of William and Mary, June 1989.

[36] D. M. Nicol. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. *Parallel Programming: Experiences with Applications, Languages and Systems*, 23(9):124–137, September 1988. ACM SIGPLAN Notices.

[37] J. K. Peacock, J. W. Wong, and E. G. Manning. Distributed Simulation Using a Network of Processors. *Computer Networks*, 3(1):44–56, February 1979.

[38] M. Presley et al. Benchmarking the Time Warp Operating System with a Computer Network Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.

[39] D. A. Reed, A. D. Malony, and B. D. McCredie. Parallel Discrete Event Simulation Using Shared Memory. *IEEE Transactions on Software Engineering*, 14(4):541–553, April 1988.

[40] P. L. Reiher, R. M. Fujimoto, S. Bellenot, and D. Jefferson. Cancellation Strategies in Optimistic Execution Systems. submitted for publication.

[41] P. L. Reiher, F. Wieland, and D. Jefferson. Limitation of Optimism in the Time Warp Operating System. *1989 Winter Simulation Conference Proceedings*, December 1989.

[42] P. F. Reynolds Jr. A Shared Resource Algorithm for Distributed Simulation. *Proceedings of the 9th Annual Symposium on Computer Architecture*, 10(3):259–266, April 1982.

[43] L. M. Sokol, D. P. Briscoe, and A. P. Wieland. MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3), July 1988.

[44] W. K. Su and C. L. Seitz. Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.

[45] D. B. Wagner and E. D. Lazowska. Parallel Simulation of Queueing Networks: Limitations and Potentials. *Proceedings of 1989 ACM SIGMETRICS and PERFORMANCE '89*, 17(1), May 1989.

[46] D. B. Wagner, E. D. Lazowska, and B. N. Bershad. Techniques for Efficient Shared-Memory Parallel Simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.

[47] F. Wieland et al. Distributed Combat Simulation and Time Warp: The Model and its Performance. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.