Parallel Simulation Today

David Nicol
Dept. of Computer Science
College of William & Mary
Williamsburg, VA 23187-8795

Richard Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta, Ga 30332-0280

Abstract

This paper surveys topics that presently define the state of the art in parallel simulation. Included in the tutorial are discussions on new protocols, mathematical performance analysis, time parallelism, hardware support for parallel simulation, load balancing algorithms, and dynamic memory management for optimistic synchronization.

1 Introduction

Parallel simulation is a highly relevant research area today, given the high computational demands of large discrete-event simulations, and ready availability of high-performance multiprocessors. The number of researchers in the field has increased dramatically in the last few years, from a handful in the early 80's to over a hundred today. The annual Workshop on Parallel and Distributed Simulation has been held six times, drawing over fifty paper submissions in each of the last three years. The annual Winter Simulation has had sessions (and sometimes entire tracks) on parallel simulation throughout this period as well.

The purpose of this paper is to outline the state of the art in six active research areas within parallel simulation to an audience assumed to have already a passing familiarity with the topic. The topics we cover are new protocols, analytic performance analysis, time parallelism, hardware support, load balancing, and dynamic memory management. We conclude the discussion of each topic with ideas for future research.

2 New Protocols

Interest in parallel simulation arose first with the problem of synchronization; it is a problem that has remained the focus of most research in the area. Over the course of 15 years a profusion of new protocols have been proposed; we cannot but touch upon a few of the new ones here. Our intention is to give examples illustrating general trends in protocol research—enhancements to classical Chandy-Misra-Byrant (CMB) style protocols [16, 10], enhancements to Time Warp[44], and new, synchronous protocols.

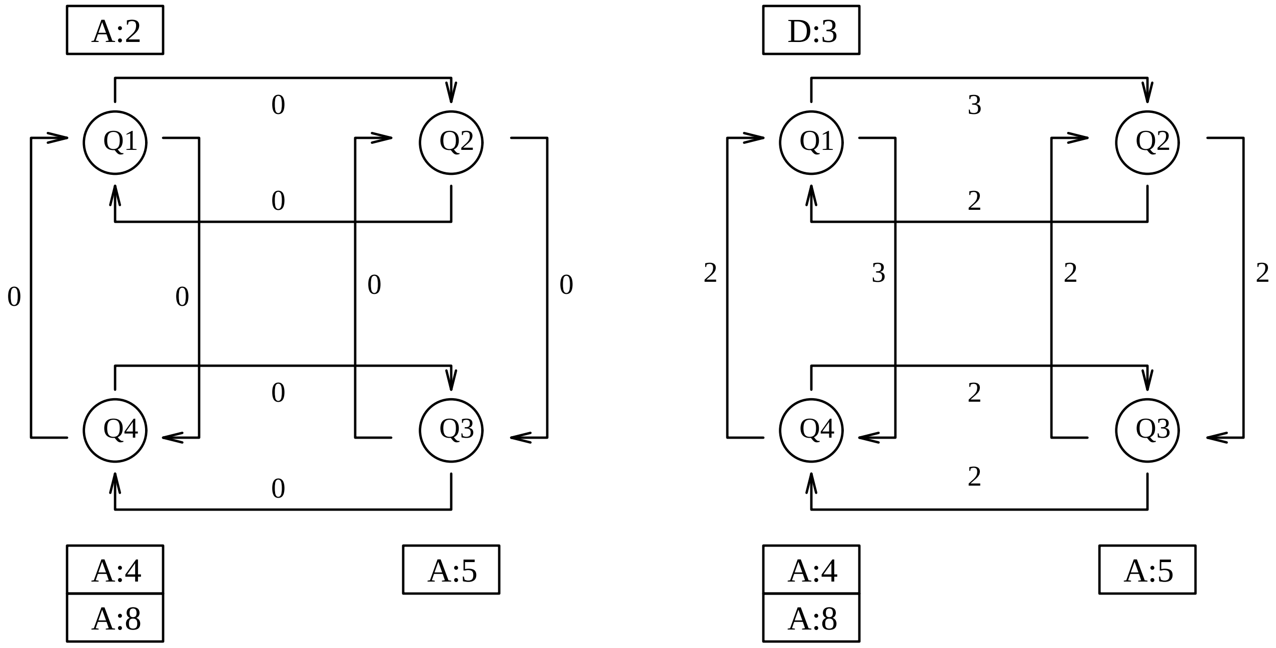
Before discussing the new directions, let us briefly revisit the synchronization problem and the classical approaches to it. Consider the network of four queues illustrated in Figure 1(a). Each queue may route a job to one of two other queues, a communication pattern which forms a simple bidirectional ring. Each queue maintains a list of events; in the figure, for example, A:4 denotes a job arrival event scheduled for time 4. Values on communication arcs (called *link times* hereafter) denote the time-stamp of the last message sent over that arc. Let us suppose that each queue is simulated on its own processor; let us also suppose that the service time of any job is at least 0.1. At the beginning of the simulation, a queue knows its initial job arrival (presumablely placed there as part of initialization), and arc times are initialized to zero. In CMB style protocols, no queue can simulate its first event until it is certain that it will not receive a routed job with a time-stamp less than its first arrival time. Now we have a problem, for the arrival times are all strictly greater then the initial link times. In order to resolve this, every queue reasons "even if I were to receive a job at time 0, that job would require at least 0.1 service time, whence I can promise not to send a job until at least time 0.1"; this reasoning permits the queue to send a *null-message* with time-stamp 0.1 to both queues to which it routes jobs. Since every queue does this, every link time eventually increases to 0.1. Under the CMB rules the queue may receive and process the message associated with the least link time. Eventually a queue receives two null-messages, with the same time-stamp, and these may be processed. As a result, each queue sends two new null messages, now with time-stamp 0.2. This sort of gradual escalating of null-message time-stamps continues until the link times increase to the point of the Q1 arrival at time 2. At this point actual simulation activity begins. Observe that twenty rounds of null-message increments were needed just to reach this point. Suppose the Q1 arrival goes into service, is non-preemptable, and will depart at time 3. Knowing this, Q3 can send null messages with time-stamp 3 ("looking ahead" to the job's completion) to Q2 and Q4, leading to the situation illustrated in Figure 1(b). Continued incremental advances in null-message time-stamps are needed to raise link times to a high enough level so that the Q1 departure at time 3 can be simulated.

The problem with the above scheme is clearly the high volume of null-messages. An optimistic approach such as Time Warp avoids these. In Time Warp, every queue checkpoints its state, then optimistically executes the first event. But this approach has its perils also. The Q2 arrival at time 2 departs at time 3 and may be routed to Q4. Alas, Q4 has likely simulated an arrival at time 4, which must now be undone, along with all messages that may have been sent prior to time 3. It recovers its initial state and simulates the new arrival. Suppose that a single unit of service time is given, and that the job is routed to Q3 at time 4. Since Q3 has already simulated an arrival at time 5, it too must roll back, send anti-messages after messages it erroneously sent, recover its initial state, and simulate the new arrival.

These descriptions are intended to suggest that synchronization protocols typically impose severe overheads. The goal of some current protocol research is to reduce those overheads. Let us now turn to some specific examples.

2.1 Enhancements to CMB algorithms

One of the reasons the CMB example above requires so many null messages is that the null messages carry very little information. If somehow Q1 came to learn that it was essentially waiting for itself and no one else before proceeding, it could clearly execute the arrival at time 2. If it could then learn that no other queue



(a) Initial configuration (b) After first event simulated

Figure 1: Example of 4 queueing nodes, assigned one per processor. Events at initialization time are shown, as are *link times*—the time-stamp on the last message to cross a communication link

.

will send a job prior to time 3, it can then simulate the departure. This observation is explored in [12], the "Carrier Null Message" approach. In standard CMB algorithms null messages propagate through a system—the result of receiving a null message is usually to send a slew of others. In the Carrier Null Message approach one appends a list of visited sites and pending event times to null messages. This information allows a queue to infer when it is free to execute an event, potentially more rapidly than when ordinary null-messages are used. Consider: Q1 initially sends out null messages with time-stamp 0.1, but appends its identity and first event time (Q1,2). One copy of the message is received by Q2, who appends (Q2,∞) and sends it back to not only Q1, but also to Q3. Q3 appends (Q3,5) and sends a copy to Q4, who appends (Q4,4) and sends a copy to Q1. The feedback on both incoming arcs permits Q1 to infer that it may proceed.

Even with carrier null messages, CMB algorithms still generate many null messages. Another optimization, explored in [79], attempts to reduce null message propagation by recognizing when a null message becomes stale. In the earlier example, Q1 sends a stream of null messages to Q2 (and Q4), successive ones increasing in time-stamp by 0.1. Now suppose a null message with time-stamp t arrives from Q1 at Q2's message queue, where it finds an unreceived null message from Q1 at time $s < t$. There is no point in having Q2 process the earlier null message; it may be annihilated. Indeed, *any* message from Q1 that discovers a null message with smaller time-stamp may annihilate the null message.

Still another set of optimizations arise when considering the high cost of message-passing in distributed memory machines. The cost of sending a v byte message is very well modeled as $\alpha + v\beta$, where α is a large fixed startup cost owing (usually) to software overhead, and β is a per byte transfer cost. This provides a strong incentive to pack logical messages together into a single physical message. CMB variations doing this are explored in [87]. A number of issues are examined, including receiver or sender initiated transfer, as well as lazy or eager transmission.

2.2 Enhancements to Time Warp

Another body of work examines optimizations to the basic Time Warp mechanism. The problem addressed by these optimizations is the possibility in Time Warp of a "fast" processor or a set of processors surging far ahead of other processors in simulation time. The danger is greatest when interaction between processors is light and processor loads are uneven. Thrashing may occur, as may cascading rollbacks. For example, some straggler can roll back a fast processor, who has generated a great many messages which are now cancelled. While the slower processes are busy annihilating message/anti-message pairs, some of them rolling back and generating additional anti-messages, the fast processor may surge forward again. While the argument can be made that the fast processor may as well execute optimistically as it has nothing else to do, the countering argument is that there is a non-trivial cost associated with correcting the errors it may make by doing so.

One idea for preventing uncontrolled *chaotic* rollbacks is to cause controlled *preemptive* rollbacks. For example, when one processor needs to rollback it may immediately issue rollback instructions to other processors who will likely have to roll back anyway as a result. One way to view this is as the parallelization of rollbacks that would otherwise occur serially. This idea finds expression in [63]. Another way of implementing this same basic idea is to build periodic—or random—preemptive rollbacks that occur independently of any activity in the simulation model [62]. The idea is to ensure that all processors are more-or-less synchronized in the same region of simulation time, with the hope that rollback cascades are less likely as a result.

A related line of thought is to simply constrain Time Warp's optimism. For example, one may advance simulation time by "windows". Within a window $[t, t+\Delta]$ processors execute standard Time Warp, *except* that no event with a time-stamp greater than or equal to $t+\Delta$ is executed. Once all processors have synchronized at time t (which is itself a non-trivial problem addressed in [69]), a new window $[t+\Delta, t+2\Delta]$ is simulated. This basic proposal is found in [89] and [5]. A similar proposal to extend constrained optimistism to the Bounded-Lag protocol is found in [61].

2.3 Protocols Based on Windows

One emerging theme in protocol research is to study protocols that constrain all concurrent simulation activity to be within some window of global synchronization time. These protocols typically compute, distribute and are controlled by global system information. In this they reflect a philosophical shift away from the roots of parallel simulation in asynchronous distributed system theory.

The algorithms studied in [14, 76, 3, 86, 33] all compute a minimum time defining a time beyond which a processor will not venture until the next window "phase". Typically, this calculation involves lookahead of some kind. For example, in the queueing simulation examined earlier we may take advantage of a non-preemptive queueing discipline, and state-independent service times and routing decisions by *pre-sending* job completions at the point the job enters service, and by *pre-sampling* a job's service time upon recognizing the message reporting its arrival. The algorithm studied in [76] reasons as follows. Since we know all there is to know about the job's departure at the time it enters service, we may as well immediately report the job's arrival at its next queue (this sort of pre-sending is also implicit with Time Warp messages). Using knowledge of the queueing discipline and the assumption that no further jobs will arrive, the queue can at any time compute the time of the next message it will send. That time is necessarily the departure time of the next job to enter service (assuming no further messages arrive). Let us suppose that all processors have simulated up to time t and have synchronized globally. Each processor i is asked to compute the time $\delta_i(t)$ of the next message it will send (in the absence of receiving further messages), and the processors cooperatively compute the minimum $\delta(t) = \min_i\{\delta_i(t)\}$. The window $[t, \delta(t))$ is thus defined, and every processor is now free to simulate all events with time-stamps within this window. Because of the window's construction, and by the practice of pre-sending job departures, we are assured that no message that is sent between processors during this interval has a time-stamp smaller than $\delta(t)$.

Consider how this mechanism would be applied to our earlier example. Initially all processors are synchronized at time 0, as shown in Figure 2(a). Q1 computes $\delta_1(0) = 3$, Q2 computes $\delta_2(0) = \infty$, Q3 computes $\delta_3(0) = 7$, and using a parallel min-reduction they compute $\delta(0) = 3$. Each queue identifies the completion time of the next job to receive service, a calculation made possible by pre-sampling service times (which are illustrated within parenthesis in Figure 2). Only one event occurs in the first window, the arrival at Q1. Upon placing the job in service, Q1 decides that Q4 will next receive the job, and sends a message to Q4 notifying it of the arrival. Q1 also generates a departure event (D) at time 3 and places it in its event list. Figure 2(b) illustrates the resulting situation, just prior to the second window. Note that Q4 pre-sampled the service requirement of its new job to be 1. Now the minimum time of the next message to be sent happens to be the departure time of the new arrival at Q4. Consequently the second window is $[3, 4)$, wherein the departure at Q1 is simulated, the corresponding arrival at Q4 is simulated, and notification of a new arrival at time 4 is given to Q3 (who then pre-samples a service time of 3). The third window is computed to be $[4, 7)$. In this interval Q4 simulates a departure at time 4 and an arrival at time 4, pre-sending notification of that job's departure (at time 7) to Q1, who chooses a service time of 2. Simultaneously, Q3 simulates a job arrival at time 4 (pre-sending its transfer to Q2), and simulates the job arrival at time 5 by marking the job as enqueued (since the server is busy). Upon receiving the arrival at time 7, Q2 pre-samples a service time of 5 units and places the new arrival event in its event list.

The natural question to ask of such algorithms is whether windows tend to admit enough parallel events to be effective. This issue has been addressed for the very algorithm above, as well as for the Bounded Lag algorithm[60]. Both algorithms are *scalable*, which means that their performance characteristics do not degrade as the size of the problem and architecture simultaneously increase. Some insight into this phenomenon is gained if we suppose that a job's service time is always at least $c > 0$. Since the $\delta_i(t)$ value computed by a processor is the completion time of a job that has not yet entered service, one infers that $\delta_i(t) - t \geq c$ for all i, so that the span of simulation time covered by the window is at least c time units wide. The average number of events processed in a window is at least $c\Lambda$, where Λ is the event density (events/unit simulation time) for the entire simulation model. Increasing the problem size increases the event density; the number of events in a window increases proportionally with Λ. Assuming the simulation load is evenly balanced (or that the imbalance does not grow with the number of processors), the number of events a processor executes per window does not decrease if the number of processors and event density simultaneously increase in fixed proportion.

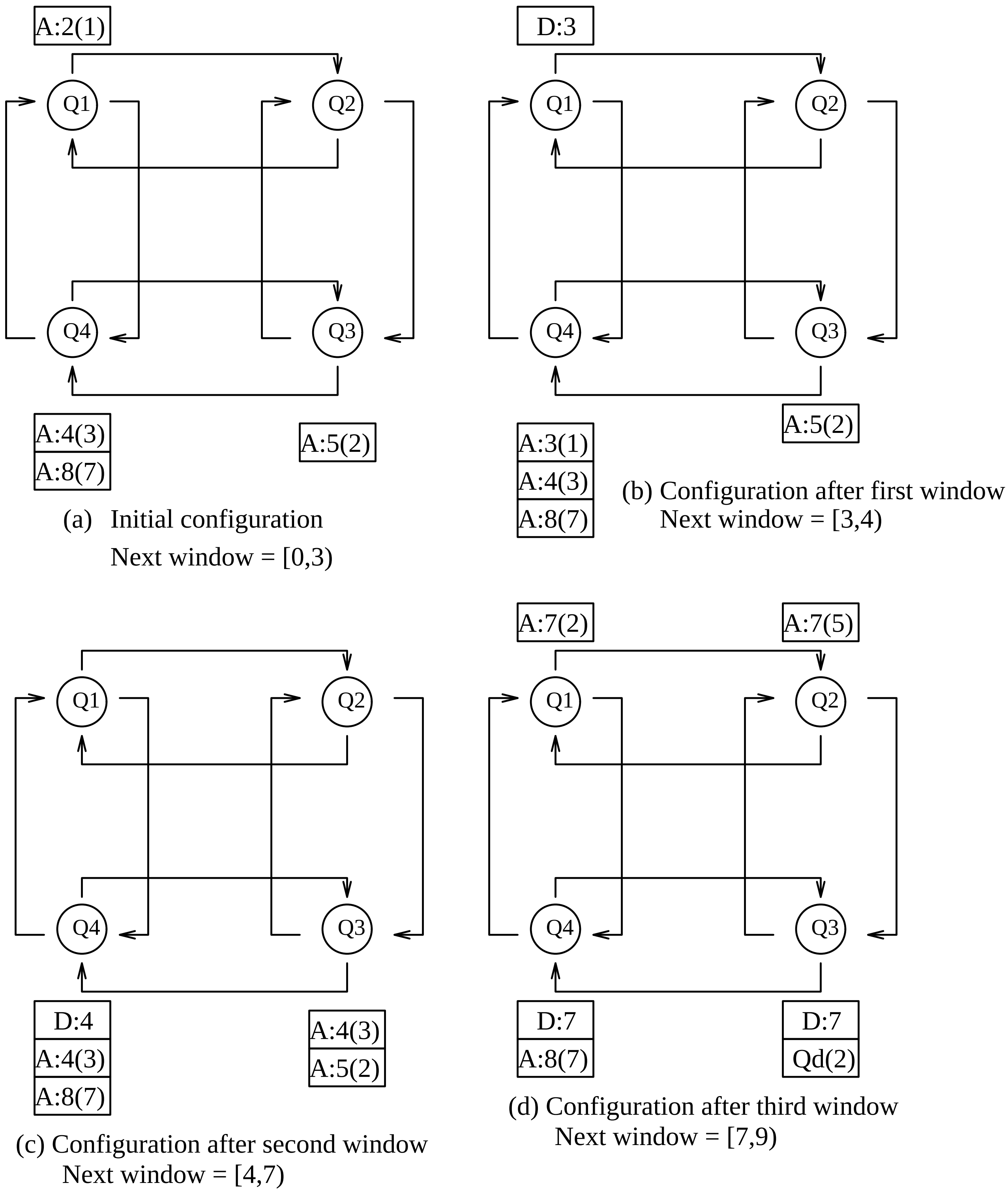


Figure 2: Simulation using conservative windows

Another advantage of window-based protocols is that they are relatively easier to use on SIMD (Single Instruction Multiple Data) architectures. Successful window-based SIMD simulation of a switching network is reported in [8], and of a circuit-switched communication network in [33].

2.4 Application Specific Protocols

It is frequently the case that the importance of an application justifies tailoring a protocol to its special requirements and characteristics. This approach often delivers performance advantages over "general" protocols, which may suffer extra overheads to support circumstances rarely encountered in the application.

One such example is the simulation of digital logic networks. VLSI simulation is notorious for its computational demands, the significance of successful parallelization would be large. Standard CMB and Time Warp approaches have been attempted [85], [64], with only mixed results. Recognizing that feedback loops pose one of the hardest problems for a conservative synchronization algorithm, [20] propose an approach where the network to be simulated is transformed into another (larger) one containing no feedback loops. This algorithm is tested on a latch constructed from two cross-coupled NAND gates.

Another important class of simulation models are continuous time Markov chains (CTMC). A CTMC is a mathematical model that encapsulates the notion of system state, and time duration. Stated simply, a CTMC is always in one of a possibly infinite number of states. Upon entering state $\mathbf{s}$, the CTMC remains in that state for a random period of time (called the *holding time*) which is exponentially distributed, with state-dependent mean $1/\lambda(\mathbf{s})$. At the completion of the holding time the CTMC makes a random transition into another state. The probability distribution of the transition also depends on $\mathbf{s}$. CTMCs are very general constructs, and are often used to model complex computer systems and communication networks.

In a series of papers [42, 71, 72] it is shown that the mathematical structure of CTMC models can be exploited for the purposes of synchronization. Using the notion of *uniformization*, it is possible to simulate a CTMC on a parallel machine in two phases. In the first phase one randomly selects a set of times at which processors will synchronize. That is, for every ordered pair of processors (i, j) we construct a communication schedule of times where i *may* send a message to j. The interarrival times in this schedule are exponentially distributed with a mean $1/\lambda_{\max}$ which is smaller than the mean time of any distribution of times between $i \rightarrow j$ messages. In the second phase, one performs the simulation, selecting a mathematically correct sample path where all actual communication instants are already identified in the pre-computed lists. To ensure that the statistics generated by such a method are correct, whenever a processor reaches one of its pre-selected communication points, it decides randomly whether to send a "real" communication that will affect the recipient processor, or to send a "pseudo" communication to release the other processor from waiting for this message. The probability of sending a real message depends on the state of the simulation at the communication instant. It should be recognized that the ability to pre-select all communication instants removes most of the difficulty of synchronizing a parallel simulation. The only drawbacks to this method are that it is not general, and that it is possible to spend too much time generating and synchronizing upon pseudo communications. The algorithm has been implemented on the Intel Touchstone Delta architecture using up to 256 processors. Speedups in excess of 220 are reported simulating on a moderate sized queueing network where every processor synchronizes with every other processor.

A final illustration of application dependent protocols occurs considering the simulation of Timed Petri

Nets (TPN). The semantics of a TPN simulation do not fit easily into the CMB world-view. As a consequence, extensions to the CMB protocol have been proposed in [48] and [88]. However, it *is* possible to simulation a TPN using a general windowing protocol, as shown in [74].

2.5 Future Directions

Synchronization will always be an interesting area of study. However, the fact remains that a number of different approaches have been shown to work, albeit under varying circumstances and with varying degrees of success. If the practice of parallel simulation is to become widespread, most of the difficult details of synchronization must be embedded within a parallel simulation environment where they remain hidden from the simulation modeler. It seems to us that the critical problems for parallel simulation lie in its automation. The important future work in protocol design lies in developing protocols whose application is automatable to a wide variety of simulation models, and whose overheads are minimal.

3 Analytic Performance Analysis

The last three years has witnessed an explosion of papers on the analytic performance modeling of parallel simulations. A common trait among these are assumptions made for the purposes of mathematical tractability. For example, it is commonly assumed that the time-advance associated with executing an event is an exponential random variable; it is commonly assumed that when sent, a message is routed to some processor selected uniformly at random from among all processors. Markov chains of one kind or another frequently underlie these analyses. Despite obvious limitations, this ground-breaking work in analysis is exciting because it helps to shed new understanding on the potentials—and limits—of parallel simulation. The remainder of the section examines different topical areas of recent analytic work.

3.1 Synchronous vs. Asynchronous

A significant body of work is devoted to comparing different synchronization algorithms. In [24] it is shown that the *average* performance difference between synchronous time-stepping and an optimistic asynchronous algorithm such as Time Warp is no more than a factor of $O(\log P)$, P being the number of processors. The derivation of this result is straightforward. Imagine that each processor executes exactly K "stages" of work, that the execution time associated with a stage is exponentially distributed with common mean μ, and that the computation is finished only after all processors have completed all stages. Given these assumptions, synchronous time-stepping is well modeled by assuming that processors engage in a barrier synchronization after every stage. The average time required for the last processor to finish a stage is no greater than $\mu \log P$, whence the average time to termination is no greater than $K\mu \log P$. If we relax the synchronization requirement—as does Time Warp—then the average time to completion is at least $K\mu$. Consequently, the ratio of synchronous to asynchronous finishing times is no greater than $\log P$. This is actually an extreme case—if the time advance distribution is bounded from above, the performance difference is no more than a factor of 2. For example, suppose the stage processing time is uniformly distributed on $[a, b]$. Then the time required for the synchronous method to finish is no more than Kb, whereas the average time for the asynchronous method is at least $K(a + b)/2$. Their ratio is $2b/(a + b) \leq 2$. While simple, this model serves to show that in a statistical sense, one ought to limit one's expectations of asynchronous vs. synchronous methods.

3.2 Optimality of Optimism

Conditions for the optimality of Time Warp (in the absence of overhead costs) are demonstrated in [51]. At a glance, this result seems intuitive, because Time Warp need never block. However, the analysis is careful to point out that Time Warp need not be optimal if ever a processor executing a piece of work on the critical path in a correct state (which, of course, cannot be known a priori) is rolled back. This causes the effective critical path to lengthen. Again, even though the model is simple and the assumption of zero-cost overhead is unrealistic, some insight is gained into the behavior of the protocols studied.

In a similar vein an interesting asymmetry is demonstrated in [58], with examples showing that Time Warp is capable of arbitrarily better performance than most conservative methods and a proof that the converse is not true. Even though such disparities are rarely observed in practice, the results are interesting in that they highlight precisely how Time Warp can guess right while a conservative method blocks. Likewise, the proof that Time Warp is no worse than conservative methods by a constant factor demonstrates Time Warp's essential resiliency, at least under the model assumptions (constant cost rollbacks, zero cost message passing and state saving).

3.3 Fanout and Time Advance Variability

Two models address themselves to the effects on performance of message-fanout, and (more indirectly) the variability in the probability distribution used to advance simulation time at a processor. A conservative windowing algorithm is compared with Time Warp in [22]. This analysis includes overheads for both methods, and captures the dependence of performance on lookahead. Not surprising, the results of the comparison depend on the magnitudes of the overhead costs. In this model each of P processors is assumed to always be busy. Each event advances the processor's simulation clock by a random amount (different distributions are considered); the time required to process an event is constant. The latter assumption lets us view the system as responding to a global clock, where every "tick" events may be executed. At the end of every event the processor chooses k other processors uniformly at random, and sends them commonly time-stamped messages. The value of this time-stamp depends on the assumed degree of lookahead. Assuming no lookahead, the time-stamp is the time of the simulation clock at the time of transmission. With one "cycle" of lookahead the time-stamp is what the clock value will be after the next event is processed; this essentially models pre-sending completion messages as was seen in §2.3. In the conservative algorithm a processor is not permitted to execute an event until it is certain not to receive a message in its past. No such constraint is placed on Time Warp, but it *is* assumed that a straggler message always causes a rollback. The results show that processor utilization under the conservative method with one-cycle lookahead is proportional to $1/\sqrt{P}$, while the utilization under Time Warp is no greater than $1/k$. Intuitive understanding of the $1/k$ figure is gained by consider the effect achieved when the processor with the least time-stamp (say $t_{\min}$) sends messages to k randomly chosen neighbors. The advance in global virtual time in the next tick is no greater than the difference between $t_{\min}$ and the least time-stamp of the next message sent by any of the k recipients. The distribution of time between $t_{\min}$ and a processor's next message time is the *equilibrium distribution* [83] associated with the time-stamp advancement distribution, which in the case of the exponential is the exponential itself. The *minimum* of k independent exponentials with mean μ is well-known to be exponential with mean μ/k. Consequently, simulation time advances by no more than $(1/k)^{th}$ of a simulation time unit in a tick. A similar analysis gives the upper bound for the conservative method with lookahead. Without lookahead the conservative method achieves a processor utilization of $1/P$—serial processing—which demonstrates its utter reliance on lookahead to achieve good performance.

An interesting point of comparison is developed in [25], where the distributional assumptions concerning simulation time advance and per-event execution time are exactly reversed—an event is assumed to require an exponential processing time (with mean 1), but advances simulation time by a deterministic one unit. One can imagine the behavior of a processor on the simulation time line as taking discrete, single-step jumps forward with exponentially distributed pauses between jumps, and occasional rollbacks to an earlier time-steps. The key idea in this analysis is to consider how long (in processing time) it takes GVT to move forward one step. Consider the instant when the GVT changes. This happens because there was one processor with the least time-stamp n, and it finally moved forward to time $n+1$. In doing so it sent k messages out (with time-stamp $n+1$) which may cause rollback. In any case, we are assured that at the instant the GVT becomes $n+1$, there are at least $k+1$ processors whose clocks have that value (the sender plus k recipients of the message). How long does it take for *all* $k+1$ of these to advance? Since exponentials are memoryless, this is time is the maximum of $k+1$ exponentials, a random variable whose mean is approximately $\log(k+1)$. This means GVT advances at a rate no greater than $1/\log(k+1)$ simulation time units per execution unit. For all but the smallest values of k we have $1/k \ll 1/\log(k+1)$, which shows that the upper bound on Time Warp performance under the new set of assumptions is much greater. Furthermore, the bounds become close to observed simulated rates as k grows.

The only difference between the models in [22] and [25] are distributional, and yet the results are very different. Both analyses look at how GVT advances; the difference in results derive immediately from the stochastic component of GVT advance. In the first model we measure GVT advance per unit execution time as the *minimum* of k exponentials, with the result that on average k execution units are needed to advance GVT by one simulation time unit. In the second model we measure the number of execution time units needed to advance GVT by a single time unit, and find that the *maximum* of k exponentials define this time. The mean minimum of k exponentials is inversely proportional to k, whereas the mean maximum of k exponentials is not proportional to k. It would seem then that the difference between methods can largely be attributed to the different responses of the exponential distribution when taking the minimum as opposed to the maximum of k independent samples. The disparity may just be an artifact of the model assumptions.

3.4 Analysis of Windowing Algorithms

As we have seen already, synchronization algorithms based on windows are becoming increasingly important. One attraction is that they are relatively easier to analyze than are completely asynchronous algorithms, since one's attention need only be focused on one representative window.

The conservative windowing algorithm described in §2.3 is analyzed in [76]. While the details are complex, the general idea is simple. The model assumes that

- event times are distributed as a constant c plus an exponential with rate μ;

- upon completing, an event schedules other events at a random subset of other LPs (the event causation assumptions are very general),
- event-completion messages are pre-sent,
- executing events are not preempted.

The analysis establishes an approximated lower bound on the mean width of the window. Under some additional uniformizing assumptions, this bound is essentially the mean minimum of N random variables $Z_1, \ldots, Z_N$, where each Z_i is the sum of c (possibly zero), plus an Erlang-2 with parameter μ. The mean minimum is proportional to $c + \mu\sqrt{N}$, implying that if the system model simulation activity rate is Λ events per unit simulation time, then at least $\Lambda(c + \mu\sqrt{N})$ events are available to be processed in the window. The paper goes on to show that the overheads involved in computing and communicating the window bound are no greater than those of event-list management, whence the algorithm is asymptotically optimal (assuming the load is balanced).

Another effort analytically examines the cost of widening the conservative window defined above somewhat, thereby finding more events to execute in parallel, but also suffering the risk of being rolled back[21]. Analysis of the extension shows that the window construct prevents rollbacks from cascading very far. Furthermore if state-saving costs are not large, the benefit of extending the window exceeds the costs, and better performance than the conservative window scheme may be achieved.

Essentially the same algorithm is analyzed in [86], but in a very different way. A differential wave equation is constructed expressing the density of events within a window at time t (assuming the window starts at 0). Numerical solution shows excellent agreement both with empirical results, and with the values predicted by the earlier model.

Finally, an analysis of synchronous relaxation is given in [23]. Convergence rate is always (or ought to be) the primary issue with any relaxation algorithm. The key idea behind this analysis is to represent the computation as connected event-lines, where each of N LPs has its own event-line, and logical dependency between LPs is reflected by a "bond" between their time-lines at the appropriate time. The number of iterations required to converge is related to the "height" of the bond graph so constructed, which turns out to have mean proportional to $\log N$.

3.5 Analysis based on Markov chain Models

A number of performance models are based on analysis of a Markov chain one discovers after making sufficiently benign assumptions. Generally it is the analysis of the chain that is difficult, not so much its construction. Let us now describe a few of these models.

First consider a system of two processors. They are loaded so that one advances simulation time at a constant rate A (simulation units per unit execution time), and another at rate B, $A < B$. At the end of every execution tick the "slow" processor sends a time-stamped message to the fast processor with probability p_1; conversely the fast processor sends a message to the slow one with probability p_2. Rollback may occur, and is assumed to require one execution tick. A processor rolling back does not advance simulation time. Now at any time, either the fast processor is ahead in simulation time, or the slow processor is (which may happen immediately following a rollback of the fast processor). The associated discrete-time Markov chain has but two states. Transition probabilities follow immediately from the message probabilities.

A more complex two processor model is analyzed in [26]. Here one assumes that a processor takes only integer-valued time-stamps, and that upon executing an event (assumed to require a deterministic 1 tick) it advances its local clock by a random geometrically distributed amount. With some probability p the processor sends a message to the other, which will roll back if the message time-stamp is less than its own clock. We let $X(t)$ denote the simulation time of one processor at tick t, let $Y(t)$ similarly describe the other processor, and define $D(t) = X(t) - Y(t)$. $D(t)$ is a stationary process, whereas $X(t)$ and $Y(t)$ tend to increase in t. $D(t)$ is a discrete-time Markov chain on the space of all integers, and has a fairly imposing transitional structure since an infinite number of states are reachable from any given state. Solution of the chain's equilibrium probabilities is non-trivial, but can be done exactly.

Markov models of Time Warp on multiple processors have also been developed. The model in [41] assumes

- unlimited memory at each processor;
- message processing is comprised of advancing the simulation clock by an exponential amount, and by sending one message to another processor, chosen uniformly at random;
- the time required to execute an event is exponentially distributed.

The analysis identifies the process holding the least-time uncommitted event as the *GVT-regulator*. Given the time-stamp t of the least-time uncommitted event, we can conceptually identify for any processor the number of events k it has processed with time-stamps greater than t and less than the processor's local clock. This k is the state of the processor. The state changes when either

- The processor is rolled back. This causes the state to decrease.
- The GVT-regulator completes its event. This causes the GVT to advance, which may decrease the processor's state.
- The processor completes an event. This causes the state to increase by one.

A Markov model that accurately describes each and every processor is too large to solve exactly. Consequently, an approximation is made to represent the entire system with one 'representative' processor. This can be defended on the grounds that under the model assumptions each processor's subchain will have the same equilibrium state probabilities. Even so, the transition probabilities can only be approximated, and then only in terms of multiple (≈ 10) model unknowns. Solution requires a fixed-point numerical procedure to solve a set of a dozen or so coupled non-linear equations.

The Time Warp model above was extended in [1] to consider the effects of limited memory in a shared memory system. It is assumed that all memory is allocated from a global buffer, with capacity supporting up to M uncommitted events. The basic assumptions about simulation behavior are the same, except that no processor may execute an additional event if the memory is exhausted. A different Markov chain is analyzed, where the state is the total number of processed but uncommitted messages in the system. The state space is thus finite, as memory is limited. Complex approximations for transition probabilities are developed, and the chain is solved numerically. Performance is measured as the number of messages committed per unit time, a metric from which speedup can be derived.

3.6 Analyses of Time Warp Rollback

The behavior of rollback in Time Warp has fascinated researchers from the very beginning. Some recent analytic work attempts to explain this behavior. Lazy and aggressive cancellation are examined in [55]. Equations for the probability of rollback are derived for some simple queueing networks, as is the probability that a rolled back message is actually correct. This latter probability assesses the utility of lazy cancellation.

A sophisticated model of rollback behavior based in the theory of branching processes is developed in [59]. The model assumes that the effect of processing an event is to generate a random number b of other events. This assumption essentially defines a branching process of event causality. One can view the progress of a simulation in terms of the growth of this tree. Now, if a processor is rolled back to some event e_1, it is necessary (assuming aggressive cancellation) to roll back all events descended from e_1. Another parameter, h, is related to the rate at which information about incorrect events propagates through the system. The analysis identifies a relationship, $b = e^h$ that defines a performance cusp. Rollbacks are rare when $b < e^h$, and recovery is quick. When $b > e^h$ the simulation eventually is swamped with cascading rollbacks. An example of the later phenomenon is illustrated with the simulation of a shuffle-exchange communication network.

3.7 Future Directions

Existing analytic models of parallel simulations range in complexity from being very simple to being very complex. The simple ones have the attraction of clearly exposing some performance feature of interest, and the results obtained using such a model may give some insight into qualitative behavior of parallel simulations with respect to that feature. Complex models may do a better job of predicting behavior, but their results lack an intuitive feeling. In our opinion, open avenues of inquiry include the inter-relationship between synchronization, load balancing, scheduling, and memory management. We believe that the most valuable models will be ones that are sufficiently complex to capture these inter-relationships, yet are sufficiently simple so that the relationships can be explained qualitatively from the results, not just quantitatively from numerical solution.

4 Time Parallelism

The most obvious parallelism in physical systems is due to concurrent activity among spatially separated objects, so-called *space* parallelism. This very parallelism suggests that a parallel approach might be taken. However, there are limitations. For example, if you simulate 100 objects in a domain, then spatial parallelism is likely limited to a factor of 100. After considering synchronization and communication overheads, it may be that the best parallel performance is achieved using only 10 processors.

It has recently been recognized that parallelism can also be found in *time*—when the behavior of a single object at different points in time can be concurrently simulated. Early recognition of this fact is found in [15], where the authors observe that simulations are fixed-point computations, and as such can be executed as asynchronous-update computations. Practical exploitation of time parallelism was first established by work reported in [40], where it was shown how certain queueing systems can be expressed as systems of recurrence relations (in the time domain), which can be solved using standard parallel prefix methods on massively parallel machines. The idea is elegant, and bears further discussion.

4.1 Methods based on Parallel Prefix

Consider a single FCFS G/G/1 queue. There is seemingly little parallelism here; the process appears to be inherently serial. However, supposing that service times and job inter-arrival times are independent of the queue state, there is no reason we cannot pre-sample (in parallel!) a large number of job inter-arrival times $r_1, r_2, \ldots, r_N$ (r_i is the time between the arrival of the $(i-1)^{st}$ and i^{th} jobs), and service times $s_1, s_2, \ldots, s_N$ for the corresponding jobs. Now the basic job of the simulation is to compute, for each job, the amount of time between the job's arrival and its entry into service. Given these *delays*, most statistics of interest can be computed. Denote the delay associated with the i^{th} job as d_i. There is a well-known recurrence relation for d_i:

$$d_i = (d_{i-1} + s_{i-1} - r_i)^+, \qquad \text{for } i = 1, 2, \ldots, N \tag{1}$$

where $(x)^+ = \max\{0, x\}$. It is helpful to view these equations as $d_i = \phi(d_{i-1}, z_i)$, where $z_i = (s_{i-1} - r_i)$ and $\phi(y, x) = (y + x)^+$.

It turns out that one can solve this system of equations using the notion of *parallel prefix*, defined as follows. Given inputs $z_1, \ldots, z_N$ and an associative operator $\circ$, we wish to compute the N partial products $z_1, z_1 \circ z_2, \ldots, z_1 \circ z_2 \cdots z_N$. One can compute all these products in $O(\log N)$ time on a parallel processor with up to N processors; routines for doing so are typically provided in a system library on SIMD machines. The trick to solving equations (1) is to cast them as a matrix recurrence in the semiring where max is the addition operator with identity $-\infty$ and $+$ is the multiplication operator with identity 0. Equation (1) is then expressible as

$$D_i = M_i D_{i-1}$$

where

$$D_i = \begin{bmatrix} d_i \\ 0 \end{bmatrix}, \qquad M_i = \begin{bmatrix} s_{i-1} - r_i & 0 \\ -\infty & 0 \end{bmatrix},$$

and the usual rules of vector and matrix multiplication apply but with scalar addition and multiplication taken to be max and $+$, respectively. Unrolling the recursion we have

$$D_i = M_i M_{i-1} \cdots M_2 D_1.$$

To parallelize, we suppose that the r_i and s_i values are distributed so that processor i holds s_i and r_{i+1}. We may compute the d_i in two steps. In the first, we compute the partial matrix products M'_2, $M'_3 = M_3 M_2$, and so on. As a result, processor i receives M'_i. In the second we compute $D_i = M'_i D_1$ for $i = 2, \ldots, N$, a task made simpler by the fact that D_1 is the zero vector.

This same basic idea can be extended in a number of ways, including networks of feed-forward queues [40], and certain classes of timed Petri nets [4]. The remarkable thing about this approach is that the degree of parallelism we may exploit is limited only by the size of the parallel machine and its memory.

The class of recurrence equations that yield directly to this approach is actually quite constrained. However, even in more general cases there is often some utility in viewing the simulation as the solution of recurrence relations, because one can solve the equations iteratively. The following approach, called "sweeping" in [33] shows how. Consider a communication link that is able to carry K calls simultaneously. If a new call arrives at an instant when the trunk is saturated, the call is lost. Now suppose we presample N call arrival times $a_1 < a_2 < \cdots < a_N$, with N corresponding call durations $s_1, s_2, \ldots, s_N$. For each call i, let $c_i = a_i + s_i$ be the time at which the call completes, *if it is accepted*. The problem is that we don't know whether the call can be accepted without knowing the number of calls being carried at time a_i. Now merge and sort the arrival times and potential departure times into a sequence $e_1 < e_2 < \ldots e_{2N}$. Let f_i denote the number of additional calls that can be carried at the time instant just after event e_i. We may write

$$\begin{aligned} f_0 &= K \\ f_i &= \begin{cases} (f_{i-1} - 1)^+ & \text{if } e_i \text{ is an arrival} \\ (f_{i-1} + 1)^+ & \text{if } e_i \text{ is a departure for an accepted call} \\ (f_{i-1})^+ & \text{otherwise} \end{cases} \quad . \end{aligned}$$

These are tantalizingly close to the equations we solved before; we can express them as $f_i = \phi(f_{i-1}, z_i)$, where (as before) $\phi(x, y) = (x + y)^+$. However, there is a significant difference—at any given departure event e_i, we do not yet know whether the associated call is accepted; we therefore do not know whether $z_i = +1$ or $z_i = 0$ for such an event. However, we can iteratively solve the equations, as follows. Initially classify every call arrival as being unsure. We then iterate, where each iteration uses fast parallel prefix operations whose results classify additional calls as either accepted, or rejected. Iteration continues until every call is classified. We approach the problem by computing lower and upper bounds $\underline{f}_i$ and $\overline{f}_i$ on each f_i. The lower bound is constructed assuming conditions leading to the heaviest load—that every unsure call arrival is accepted and never finishes. Similarly the upper bound is constructed assuming the lightest possible load—that every unsure call is rejected. The resulting equations are

$$\begin{aligned} \underline{f}_0 &= K \\ \underline{f}_i &= \begin{cases} (\underline{f}_{i-1} - 1)^+ & \text{if } e_i \text{ is an accepted or unsure arrival} \\ (\underline{f}_{i-1} + 1)^+ & \text{if } e_i \text{ is a departure for an accepted call} \\ (\underline{f}_{i-1})^+ & \text{otherwise} \end{cases} \end{aligned}$$

and

$$\begin{aligned} \overline{f}_0 &= K \\ \overline{f}_i &= \begin{cases} (\overline{f}_{i-1} - 1)^+ & \text{if } e_i \text{ is an accepted arrival} \\ (\overline{f}_{i-1} + 1)^+ & \text{if } e_i \text{ is a departure for an accepted call} \\ (\overline{f}_{i-1})^+ & \text{otherwise} \end{cases} \quad . \end{aligned}$$

The principle difference between these and the equations for f_i are that we *do* know what argument each e_i offers to $()^+$; call classifications of unsure, accepted, and rejected are known from the previous iteration, and determine these arguments. Given a set of call classifications, the $\underline{f}_i$ and $\overline{f}_i$ values can be computed using parallel prefix. Now, following the solution of $\underline{f}_i$ and $\overline{f}_i$ at a given iteration we attempt to classify additional unsure call arrivals as follows. If e_i is an unsure arrival, and the value $\underline{f}_{i-1}$ satisfies $\underline{f}_{i-1} > 0$, then we may reclassify e_i as accepted. Similarly, if $\overline{f}_i = 0$ we may reclassify e_i as rejected. Such reclassifications improve the state of knowledge about the system; given sufficient iterations every call arrival will be classified.

Eventual convergence is assured, since during any sweep the unsure arrival with least time will always be classified. We note in passing that the general sweep approach applies to a wider range of problems than the single one given here. The motivating problem was a network of similar links, with the additional complications that one attempts to reroute rejected calls through randomly selected alternate routes, and every link reserves some of its capacity for original traffic. Convergence was rapid on a 16K PE SIMD architecture; typically thousands of calls were classified using only a handful of sweep iterations.

4.2 Other Methods

Time parallelism was also noticed in LRU trace-driven cache simulations [43] for MIMD architectures (each memory reference constitutes an event); this observation was extended in [70] for more general replacement policies, and SIMD machines. The latter approach also involves the parallel solution of recurrence equations, but in a less direct fashion than the methods described so far.

A direct approach to time parallelism is to partition the time domain, assigning different processors to different regions of time. A processor p *assumes* some initial state for the system at the beginning point of its interval, say time t, and then simulates its interval. Now the processor whose interval terminates at t may have a different final state at t than the one assumed by p. In this case a *fix-up* operation must be performed. This method will work if the cost of a fix-up is much less than the cost of resimulating the interval. Variations on this idea are found in [2], and [56].

4.3 Future Directions

Time offers another dimension in which we may seek performance gains through parallelism. However, as yet any implementation observed to actually achieve performance gains relies very heavily on the specifics of the problem being simulated. This should not be surprising, given the diversity of ways in which simulation models evolve in simulation time. It seems unlikely then that a general purpose protocol can consistently be effective in exploiting time parallelism. Nevertheless, as seen above, there are some non-trivial examples of important applications that *can* benefit from time parallelism. Future efforts might be directed towards expanding the class of applications where time-parallelism works, in formal characterization of such applications, in generalizing time-parallelism approachs to be less-application specific, and in performance analysis of such approaches.

5 Hardware Support

Hardware support for parallel discrete event simulation has been discussed in the literature for some time. Machines have been developed for simulation of logic circuits (e.g., see [27] for a survey of approaches), however these usually do not allow concurrent execution of events containing different time-stamps. Although these machines do not implement parallel simulation protocols such as those described here, they do demonstrate that there is interest in hardware support in certain portions of the commercial sector.

Hardware support for parallel simulation has been studied largely in three domains:

- new machine organizations for parallel simulation,

- hardware support for state saving in Time Warp, and
- hardware support for dissemination of global information.

The first category involves new machine organizations designed from the start with parallel simulation in mind. The latter two involve "add on" hardware that implements certain time consuming operations used in parallel simulation, and is intended to be attached to an existing parallel or distributed architecture. One advantage of the latter approach is that the hardware designs may more easily "ride the technology wave" as newer, faster, microprocessors and denser memory chips become available.

5.1 Machine Organizations

Machine architectures for parallel simulation have been studied at least 10 years. For example, Georgiadis et al., proposed a multiprocessor implementation for Simula programs in the early 1980's [35]. There, a special purpose parallel simulation engine was envisioned that utilizes a controller processor to manage the execution of the parallel simulator, and determine which processes are available for execution. A program called the *Simula Parallel Process Recognizer* performs a static analysis of the Simula process and builds tables that enumerate possible process interactions, e.g., access to common state variables or invocation of communication primitives ((re)activate, passivate, cancel, wait, or hold). These tables are then used by the runtime mechanism to conservatively synchronize of the execution. The machine architecture itself is a network of processors, with some processors dedicated to performing specific functions, e.g., coordination of process execution. Details of the hardware organization are sketchy, however.

Concepcion describes an for architecture for discrete event simulation called the *hierarchical multibus multiprocessor architecture* (HM^2A) [17]. This architecture is motivated by a methodology that is proposed for constructing hierarchical, modular, simulation models which are then mapped to the multiprocessor. The machine structure is a tree of *clusters* where each cluster includes a collection of "slave" processors (each with local memory) connected by a bus. Each slave contains a connection to the cluster's bus, and a single link to one child cluster at the next level down the tree. A slave at level i in the tree acts as the master for the processors in the level $i+1$ child cluster to which it is attached. The level i processor is referred to as the *coordinator* for the cluster, and is responsible for transmitting messages for data and synchronization to the slave processors of the cluster. Slaves are activated when they receive messages, and are otherwise passive. The cluster at the root of the tree contains a coordinator processor as well as the slaves.

The simulator is specified hierarchically, and is then mapped directly onto the tree structured machine architecture. Coordinators pass work (messages) to their slave processors, then wait until all of the slaves have responded that they are finished. A special bus within each cluster is used to transmit the "done" signals. When the slaves have all indicated that they have completed the task, the coordinator waits for the next task from it's master. In this way, simulation computations propagate up and down the tree, activating simulation models at different levels of hierarchy as needed.

A third machine proposal is the *Virtual Time Machine* [29, 36]. Unlike the above machine organizations, this machine is based on optimistic synchronization. The machine is a shared memory multiprocessor with a special type of memory system called *space-time memory*, and a hardware implemented rollback mechanism.

The most interesting aspect of the machine architecture is its memory system. Consider the following situation: an event at time (say) 100 expects to see the state of the simulator as it existed at time 100, while another event at time 200 expects to see the state as of simulated time 200. If these two events are both examining a common state variable, which "value" should the memory provide, that which existed at time 100 or the one at time 200? Space-time memory addresses this dilemma by providing a two-dimensional addressing structure, in contrast to conventional memory which is viewed as a one-dimensional array of values. Each read and write operation specifies both a conventional word address and a time-stamp address. In the above situation, the event at time 100 specifies that it wishes to see the value that existed at simulated time 100, while the second event specifies a time-stamp address of 200. The memory system manages the multiple versions of the various state variables. A read of a variable at time-stamp T_1 returns the data value written at time T_0 where $T_0 < T_1$ and no writes to that variable occurred with time-stamp between T_0 and T_1. In other words, read operations return the most recently written (in simulated time) data value.

Because events are executed at different time-stamps, synchronization errors might occur. For instance, immediately after an event with time-stamp 100 reads a variable that was written at (say) time-stamp 90, another event at time 99 might modify that same variable. This implies that the read at time 100 received an incorrect value, because the most recent value was that which was written at time 99, not time 90. Errors such as these trigger a Time Warp style rollback mechanism in the Virtual Time Machine. The machine includes hardware that "eaves drops" on the buses leading to space-time memory modules to log memory references, so that it can be determined what needs to be rolled back when an error occurs.

The motivation behind the Virtual Time Machine architecture comes from Time Warp and optimistic synchronization protocols. The machine, however, is envisioned to be a *general purpose* parallel processor based on optimistic synchronization.

5.2 Hardware Support for State Saving in Time Warp

In Time Warp, processes must periodically checkpoint their state in case a rollback later occurs. State saving overheads can incur a significant overhead [28]. One can alleviate this overhead to some extent by reducing the frequency of checkpointing, however, analytic and experimental data suggest that the optimal checkpoint interval may be frequent (e.g., every few events) [80].

Fujimoto, et al. propose a component called the *rollback chip* that provides hardware support for state saving and rollback in Time Warp [31]. This component was the forerunner to the space-time memory system described above. The rollback chip can be viewed as a special memory management unit. A process may issue a "mark" operation to indicate that the state of a data segment must be preserved in case a rollback later occurs. The rollback chip hardware then modifies the addresses of subsequent memory writes to preserve this data. This approach minimizes the amount of copying that is required.

Simulations indicate that state saving overhead can be reduced to only a few percent of the computation. Special caches are used to improve performance. A simplified prototype implementation of the rollback chip has been developed in the commercial sector [11]. Also, the hardware design of the rollback chip has been verified using formal techniques [39].

5.3 Global Synchronization Networks

One of the reasons protocols for parallel simulation are nontrivial is the fact that critical synchronization information is distributed across the multiprocessor system. For instance, in conservative protocols, information indicating which events can be safely processed may be distributed across other processors. Similarly, optimistic protocols require information that is distributed across the system to compute global virtual time.

Reynolds has proposed a hardware mechanism to rapidly collect, operate on, and disseminate synchronization information throughout a parallel simulation system [65, 78]. The hardware is configured as a binary tree, with a processor assigned to each node. To compute GVT for instance, each processor indicates a local minimum among the processes assigned to it, and the tree automatically computes the global minimum in a distributed fashion (each node computes the minimum of its local value and that of its neighbors, and propagates the new minimum up the tree) and distributes the computed value to all processors in the system by broadcasting values down the tree. Simulations indicate that the time required to compute GVT is reduced by orders of magnitude over software based approaches. A prototype system is currently under construction.

5.4 Future Directions

Hardware support is a promising approach because it helps alleviate the overheads associated with parallel simulation protocols, which are sometimes substantial. The key to successfully exploiting special-purpose hardware is to identify those aspects of the parallel simulation mechanism that are bottlenecks which seriously degrade performance. The important future directions of research in hardware therefore are to identify performance limiting factors in new approaches to parallel simulation, and to develop efficient hardware mechanisms to accelerate their performance.

6 Load Balancing

The simulation contains some initial set of logical processes. New processes may be created, or existing processes deleted as the simulation progresses. Ideally, these processes should be distributed across the parallel processor so that (1) all processors remain busy doing useful work all of the time, and (2) interprocessor communication is minimized. The importance of the communication aspect depends on the hardware platform. It is more important in distributed memory and networked workstations where communication is relatively expensive compared to tightly coupled shared memory multiprocessors.

6.1 Classes of Load Management Methods

Static load balancing algorithms distribute a fixed set of processes over the processors in the system. *Dynamic* algorithms allow processes to migrate during the execution of the parallel simulation. Dynamic algorithms are more appropriate if (1) information to achieve proper load balancing is not available until runtime, or (2) the proper distribution of processes to processors changes dynamically throughout the program's execution. A combat model, for example, may exhibit both of these behaviors. It is usually difficult to predict in advance which combat units will interact with which other units, and thereby entail the complex computations and interactions with other simulation processes that accompany the simulation of a battle. Also, these models often move through phases such as (1) advance to engage the enemy, (2) interact with enemy units, and (3) "clean up" after the battle. The computation and communication aspects of the computation are very different in each of these phases, necessitating a different load distribution for each one.

On the other hand, dynamic load management entails certain overheads to collect status information on the system, decide what load balancing actions should be taken, if any, and move computations and data from one processor to another to balance the load. These overheads may negate any positive effects of more evenly distributing the workload or reducing interprocessor communication.

Related to load balancing are load sharing and partitioning schemes. Load sharing refers to the question of selecting the processor to receive a newly created process (as opposed to migrating existing processes). Partitioning refers to subdividing the simulation model to logical processes. If the partitioning of the model to processes is changed during the execution of the simulation, e.g., to achieve a more balanced workload, it is referred to as dynamic repartitioning.

Load balancing has been widely studied for general (i.e., not necessarily simulation) parallel and distributed computation. Many of the techniques that have been proposed, e.g., simulated annealing [46], distributed drafting [68], pressure based load migration [49], among others, can be applied to parallel simulation programs.

6.2 Static Load Balancing Techniques

Early work on static load balancing is found in [77, 75]. The basic idea behind the mapping algorithm is to examine the critical paths through multiple executions of a simulation, and cluster in such a way that the critical paths are left as undisturbed as possible. A dynamic load balancing algorithm is also described that is actually dynamic invocation of the static algorithm, based on a statistical decision process that monitors the simulation's behavior and triggers a remapping when it is probable that the resulting performance gains exceed the total remapping cost. The policy was empirically studied on a parallelized time-stepped combat model [73], where remapping may occur between the advancement and engagement phases of the simulation.

Nandy and Loucks use an iterative, static load balancing algorithm for parallel simulation using the Chandy/Misra/Bryant synchronization protocol (null messages) [67]. The algorithm begins with an initial, random, partitioning of the task graph, and then continually evaluates possible movement of nodes (logical processes) from one partition to another. A gain function is calculated that considers communication costs of the proposed move relative to the existing partitioning in order to estimate the benefit of the move. An additional constraint ensures that equal amounts of computation are assigned to each processor to avoid bottlenecks. As is the case with any static algorithm, this approach assumes much is known about the simulation in terms of computation and communication requirements of logical processes.

Nandy and Loucks use this approach to map simulations of digital logic circuits to a parallel computer. They report performance improvements of up to 25% on eight processors over an algorithm based on selecting random partitions. One would expect larger improvements in performance with more processors because communication overheads then become more significant.

Davoren [19] and Briner [9] also examine static partitioning algorithms for digital logic simulation. Davoren bases his work on the Chandy/Misra/Bryant null message algorithm. He constructs a *locality tree* that is based on the hierarchical design of the circuit through different levels of abstraction (transistors, gates, multiplexers, etc.). This approach of using the design hierarchy to partition the circuit is referred to as *structural partitioning*. The tree is used to approximate communication within the circuit. A divide and conquer approach is then used to map the tree to a grid of processors. The locality tree is divided into subtrees; similarly the processor grid is divided into groups. The subtrees are assigned to the groups, and the process is repeated recursively until individual circuit elements (the leaves of the tree) are mapped to specific processors. The number of gates in each node is used to approximate computation load. Experiments on a transputer network indicate that this approach reduces the amount of interprocessor communication and execution time relative to an alternative mapping scheme whose primary goal is to evenly distributes the workload to processors.

Briner's work is based on Time Warp. He compares a random assignment of logic gates to processors with several different bisection algorithms. The bisection algorithm divides the circuit into two pieces so that communication is minimized between the sub-circuits. The sub-circuits are mapped to different processors. This process is repeated recursively, much like the approach proposed by Davoren. Briner also examined a variation on this approach where communication lines are weighted according to the amount of traffic expected to be sent on the line. Such information can be obtained from a prior simulation of the circuit, or by using knowledge of the probability of changes on signal lines and the logic function performed by the gate.

Briner's experiments on a BBN Butterfly GP-1000 indicate that random partitioning outperforms the bisection algorithms. The modified bisection algorithm only yields a modest improvement over the original algorithm. He reports that hand partitioning based on the hierarchical structure of the computation (such as that proposed by Davoren) yields up to three times better performance compared to the bisection algorithms.

Kravitz and Ackland [47] also examine some simple static partitioning schemes for circuit simulations. Based on empirical studies, they conclude that these approaches yield reasonably good results, and the overhead for dynamic repartitioning does not justify the potential performance gain. Their work is based on time-stepped simulations.

The JPL TWOS (Time Warp Operating System) group performed static load balancing for their Time Warp programs by first collecting a trace of the program's execution. Based on this trace, a task graph showing all dependencies between events is constructed, and a bin packing algorithm used to determine a suitable assignment of processes to processors. The "off-line" nature inherent in this approach led them to develop and rely upon dynamic load management algorithms instead., which are described next.

6.3 Dynamic Load Balancing

As mentioned previously, dynamic load balancing/partitioning attempt to reassign work to processors during the execution of the simulation. Optimistic synchronization mechanisms introduce a new wrinkle to dynamic load balancing: high processor utilization does not necessarily imply good performance because a processor may be busy executing work that is later undone. In this case, it would be beneficial to migrate processes to processors with *high* utilizations because the added load will tend to allocate fewer CPU cycles to the computations that are constantly being rolled back. To address this issue, Reiher and Jefferson propose a new metric called *effective processor utilization* which is defined as the fraction of the time during which a processor is executing computations that are eventually committed [82]. This effectively treats time spent executing wrong computations as idle time. Based on this metric, they propose a strategy that migrates processes from processors with high effective utilization to those with low utilization.

An algorithm that is similar in spirit is proposed in [37]. This algorithm allocates virtual time-slices to processes, based on their observed rate of advancing the local simulation clock. Uniprocessor simulation studies reveal scenarios in which this time-slicing approach achieves significantly better performance than the Reiher and Jefferson algorithm (as much as 33%), and others where the performance of the two methods is comparable.

A second problem in Time Warp is the fact that process migration may be very expensive because processes contain a large amount of history information. Reiher and Jefferson propose splitting a process into *phases* when the process migrates to another processor. Each phase spans a contiguous segment of simulated time that does not overlap with other phases. When migration occurs, the old phase (and its corresponding history information) remain on the original processor, and the new phase begins at the new processor. Rollbacks may span phase boundaries. A phase is logically similar to dynamically creating a new process that is a "clone" of the migrating process in that its state is initialized to the state of process when the migration occurs. Reiher and Jefferson demonstrate that phase splitting and the effective utilization metric are useful to dynamically balance the load in simulations of a communication network, a system of colliding pucks, and a combat models [82].

Goldberg describes an interesting approach to the load distribution problem [38]. If a process becomes a bottleneck, it is *replicated* to form two or more identical copies, each able to execute concurrently with the others. Read requests are sent to one replica, while write requests are sent to all of them. The replicated copies are kept consistent with a Time Warp based synchronization facility. A *Replicated Time Warp* algorithm is defined based on these ideas.

6.4 Future Directions

While load balancing for general parallel processing has been extensively studied, application of these techniques to parallel simulation applications, and their impact on synchronization protocols has not been widely examined. It is not known, for instance, how load distribution and scheduling affect the number of null messages produced by the Chandy/Misra/Bryant algorithm, and only very limited experimental work has studied the impact of load management on rollbacks in Time Warp. With the exception of digital logic network simulations, little work has been completed in evaluating static and dynamic load management approaches in specific application domains. Much work is still required to evaluate precisely when static load balancing mechanisms suffice, and when one should resort to dynamic mechanisms.

Currently, again with the exception of digital logic simulations, partitioning the simulation model to form logical processes is done entirely by the programmer, and is usually governed by the modeler's conceptualization of the system's. This may or may not correspond to the partitioning that maximizes performance. Partitioning the simulation into very fine grained objects is not an appropriate solution because this may lead to inappropriately small computation grains. It is clear that both mapping and load balancing issues require much additional study.

7 Memory Management

While the analyses discussed above are primarily concerned with time performance, a related question is that of memory performance. A growing body of research examines storage utilization of parallel simulations, especially optimistic mechanisms such as Time Warp. In Time Warp, four types of mechanisms have been proposed to limit the amount of memory that is required to perform the simulation:

- fossil collection,
- infrequent state saving,
- rollback-based recovery mechanisms, and
- protocols using limited optimism.

Approaches that limit the amount of optimistic execution in the system implicitly reduce the amount of memory that is required. These were discussed earlier, so we will not dwell on this issue here. The other techniques enumerated above will be described next.

7.1 Fossil Collection and GVT

Optimistic mechanisms maintain information concerning the history of the program's execution in order to enable recover from synchronization errors. In Time Warp, for instance, each process maintains past state vectors in its state queue, processed events in its input queue, and records of previously sent messages (anti-messages) in its output queue. A mechanism called *fossil collection* is provided to reclaim "old" history information that is no longer needed [44]. Fossil collection relies on the computation of a quantity called *global virtual time* (GVT), which will be defined momentarily. Storage used by message buffers and snapshots of process state that are older than (GVT) can be reclaimed and used for other purposes. Even with fossil collection, however, the amount of storage that is required to execute Time Warp programs may be large.

Let us digress for a moment to discuss the computation of GVT. GVT represents a lower bound on the time-stamp of any future rollback. In Time Warp, as originally proposed in [44], rollbacks only arise from receiving positive or negative messages in the past. Further, a process at simulated time T might produce a new (positive) message with time-stamp equal (or only slightly larger than, in systems that do not allow zero time-stamp increments) to T. Therefore, GVT is computed as the minimum among (1) the local clocks (sometimes called local virtual time or LVT) of all processes, and (2) the (receive) time-stamp of all messages in transit, i.e., messages that have been sent but not yet received. As we will see later, certain memory management schemes for Time Warp use a mechanism called message sendback which necessitates a different definition of GVT. If a process has no unprocessed messages in its input queue, then the process's local clock is usually set to infinity. If there are no unprocessed messages or messages in the transit in the entire system, GVT then becomes infinity and the simulation terminates.

In a tightly coupled multiprocessor, computation of GVT is straightforward because one can use a barrier synchronization to "freeze" the computation and obtain a global snapshot of the system, though care must be taken or serious performance degradations may occur, particularly if the system contains a limited amount of memory [1]. However, computation of GVT is more complex in distributed and loosely coupled systems because such snapshots are not so easily obtained. In particular, two problems arise in computing GVT in this context: (1) accounting for transient messages, and (2) race conditions may arise causing an erroneous value of GVT to be computed.

The first problem is usually solved by using message acknowledgments to identify messages in transit. A process must consider the time-stamps of messages that it has sent for which it has not yet received an acknowledgement when it performs its *local* GVT computation. Lin and Lazowska propose a scheme that avoids acknowledgements by having each process communicate with the other processes to which it communicates when it begins a GVT computation in order to identify any transient messages. Details of their algorithm are described in [52].

Race conditions may arise because the individual processors receive the "start GVT computation" signal at different points in time. For example, processor 1 might compute its local minima to be 100. Moments later, a second processor that has not initiated the GVT computation might send the first processor a message with time-stamp 90, receive the acknowledgement, and then advance ahead in simulated time. If the second processor now receives the start GVT signal and computes its local minima, the message it had previously sent (time-stamp 90) is not accounted for in the GVT computation, even though message acknowledgments were used.

The above problem could be solved using a barrier synchronization to ensure that all simulation computations halt before the GVT computation is begun. In fact, Nicol has developed a barrier algorithm for optimistic computations that can effectively serve to compute GVT [69]. The processors agree to synchronize globally at some simulation time t. A processor enters the barrier once it has no events to processor with time-stamps less than t, but rolls back out of the barrier if it later receives a message with time-stamp less than t. The algorithm ensures that a processor not be told it can leave the barrier until all processors have completed all simulation work at times less than or equal to t. Emerging from this barrier, a processor knows the GVT is t; it may perform fossil collection, and proceed optimistically to the next agreed-upon synchronization time. Samadi proposes another approach that tags messages sent after a GVT computation initiates, but has not yet been completed, allowing messages such as that in the preceding example to be accounted for [84].

Other approaches to computing GVT have been proposed. Preiss uses a token passing scheme where the processors making up the simulation are organized in a ring, and continually compute GVT as the token is passed from one processor to the next [81]. This approach has some similarity to ring-based algorithms for detecting deadlock [66]. Bellenot uses a statically defined tree to to initiate, compute, and disseminate GVT values [6]. Reynolds also uses a tree structure to compute GVT in his hardware synchronization network, described earlier [65].

7.2 Incremental and Infrequent State Saving

Nearly all Time Warp based memory management schemes use fossil collection to reclaim state. However, fossil collection is not, by itself, sufficient because the computation may still consume excessive amounts of memory. We will now focus our attention on other mechanisms that are used in conjunction with fossil collection to conserve memory.

When the state vector is large and only a small portion is modified by each event, incremental state saving may be useful. Here, only changes to the state are recorded rather than taking snapshots of the entire state vector, thereby reducing both memory utilization and copying time. A drawback with this approach, however, is that rollbacks become more expensive because the state vector must be reconstructed from the incremental changes. This is problematic because as illustrated in [30], the computation is more prone to unstable execution if rollback costs are high. Nevertheless, Briner uses incremental state saving in an implementation of Time Warp for logic simulations, and reports state saving overheads of only 20% for transistor level simulation, and 60% for gate level simulation [9].

An alternative approach is to save entire state vectors, but reduce the frequency of state saving. To roll back to simulated time T, a process must (1) roll back to the most recent state vector older than T, and (2) recompute forward again to reach simulated time T. Message sending must be "turned off" during the recomputation phase or a domino effect could occur that rolls back the simulation beyond GVT. Like incremental state saving, infrequent state saving also increases the cost of each rollback because on average, the length of each rollback is greater, and the number of events in each recomputation phase is increased. That is perhaps the greatest drawback with this approach.

Although infrequent state saving increases rollback overhead, it also decreases the time required to perform state saving, which can be substantial. This tradeoff suggests that there may be an optimal state saving frequency that balances state saving overhead and recomputation costs. This question has been studied in the context of fault tolerant computation, e.g., see [13, 34]. More recently, Lin and Lazowska considered this tradeoff in the context of Time Warp programs, and show that an error in overestimating the state saving frequency is more costly than an equal magnitude error in underestimating the frequency, i.e., it is better to err on the side of less-frequent-than-optimal state saving in order to maximize performance [53]. In particular, they conclude that m_{opt}, the number of events processed between state-saves, should be set in the range:

$$m_{opt}^{-} < m_{opt} < m_{opt}^{+}$$

where

$$m_{opt}^{-} = \left\lceil \sqrt{\frac{(\alpha-1)\delta}{e}} \right\rceil$$

and

$$m_{opt}^{+} = \left\lceil \sqrt{\frac{(2\alpha+1)\delta}{e}} \right\rceil$$

and α is the number of events executed between rollbacks when state-saving is performed after each event (or equivalently, the number of events executed by the process divided by the number of rollbacks when state-saving is performed after each event), δ is the cost to perform a state-save (i.e., to copy the state vector), and e is the expected execution time of an event.

Preiss, MacIntyre, and Loucks [80] and Bellenot [7] validate Lin's results experimentally. Bellenot also observes that benefits in reducing state saving frequency diminish or become liabilities as the number of processors is increased.

Finally, it might be noted that infrequent state saving economizes on storage for state vectors, but at the expense of storage for event messages. This is because events that are older than GVT, but newer (in simulated time) than the last saved state prior to GVT cannot be fossil collected because they may be needed after a rollback beyond GVT (to reach the last saved state). Storage for these events could be reclaimed if state saving were more frequent. Empirical studies of queueing network simulations indicate, however, that total memory utilization is reduced with infrequent state saving [80].

7.3 Rollback Based Protocols

The strategies discussed thus (fossil collection, incremental/infrequent state saving, limiting optimism) all have the following drawback: if the system does run out of memory, there is no recourse but to terminate the simulation. This is problematic because the "fault" may lie with the Time Warp mechanism itself rather than the application program. It could be that simulation mechanism was too optimistic in executing the program, and as a result, ran out of memory.

Several approaches have been developed to address this concern. The basic idea behind these mechanisms is to roll back overly optimistic computations, and reclaim the memory they use for other purposes. Jefferson first proposed a mechanism called *message sendback* to achieve this effect [44]. In message sendback, the Time Warp executive may return a message to its original sender without ever processing it, and reclaim the memory used by the message. Upon receiving the returned message, the sender will (usually) roll back to the send-time-stamp of the message (i.e., the virtual time of the sender of the message when it was generated), and regenerate it. This rollback causes anti-messages to be sent (assuming aggressive cancellation), and the subsequent annihilations release additional memory resources in the system. Only messages with send-time-stamp greater than GVT can be returned, since otherwise, a rollback beyond GVT might result.

Jefferson's original proposal invokes message sendback when a process receives a message, but finds that there is no memory available to store it [44]. The message with the largest send-time-stamp is returned. Gafni proposes a protocol that utilizes message sendback as well as other mechanisms to reclaim storage used by state vectors and messages stored in the output queue when a process finds that its local memory is exhausted [32].

More recently, Jefferson has proposed an alternative approach called cancelback [45]. While Gafni's algorithm will only discard state in the process that ran out of memory, cancelback allows state in any process to be reclaimed. Messages containing high send-time-stamps are sent back to reclaim storage allocated to messages. This tends to roll back processes that are ahead of others in the simulation.

Message sendback, and therefore cancelback necessitate a new definition of GVT. Messages returned to their sender may initiate rollbacks, so the *send* time-stamps of messages must now be considered in addition to the receive time-stamps. For cancelback, GVT is defined as the minimum among (1) the local clocks of the processes in the simulation, and (2) the send-time-stamp of all messages in transit [50]. The artificial rollback protocol, described below, also uses this definition of GVT.

Another approach, proposed by Lin, is the *artificial rollback* algorithm [50]. When storage is exhausted and fossil collection fails to reclaim additional memory, processes are rolled back to recover memory. The process that is the furthest ahead is rolled back to the time of the second most advanced process. This procedure is repeated until the supply of free memory reaches a certain threshold. We refer to this threshold as the *salvage parameter*. Artificial rollback is semantically similar to cancelback in the sense that cancelback returns messages which cause the sender to roll back, and artificial rollback rolls back the processes directly. The principal advantage of artificial rollback over cancelback is that it is simpler to implement.

Artificial rollback and cancelback have the interesting property that they are able to execute the simulation program using no more memory than that required by the sequential execution that utilizes an event list. Lin refers to protocols such as these that require no more than a constant times the amount of memory required for sequential execution as *storage optimal*. This is an attractive property because it allows the

Time Warp program to execute with whatever memory is available, provided there is enough to execute the sequential version.

One can see that rollback-based policies achieve storage optimality by examining the storage requirements of a sequential simulator. Consider the set of pending events in a sequential simulator at simulated time T. Let us assume that events at time T have not yet been processed. The event list will contain all events that were scheduled prior to simulated time T, but with time-stamp greater than or equal to T. Consider the *parallel* simulator where T is the current value of GVT. The parallel simulator need only remember those events whose send and receive time-stamps "straddle" GVT, and all others, i.e., those with both a send and receive time-stamp greater than GVT, are eligible for deletion. Using this idea, rollback-based memory management schemes can reclaim all memory that would not be needed in the sequential simulator at simulated time T, enabling them to execute using no more memory than the sequential program. The only question that remains is (1) which events among the set that can be deleted should be eliminated and their storage reclaimed, and (2) how much memory should be reclaimed when we run out. As mentioned earlier, conventional wisdom is to reclaim events with high send-time-stamps first (these tend to roll back the processes furthest ahead). The second parameter, the salvage parameter that was defined earlier, is a control for tuning performance.

It is interesting to note that while Time Warp with cancelback or artificial rollback are storage optimal, certain *conservative* simulation protocols are not. Lin et al. [54] and Jefferson [45] show that the Chandy/Misra/Bryant algorithm may require $O(nk)$ space for parallel simulations executing on n processors where the sequential simulation requires only $O(n+k)$ space. Further, Lin and Preiss [57] report the existence of simulations where Chandy/Misra/Bryant have exponential space complexity, and thus utilize much more storage than the sequential simulation. On the other hand, they also indicate that this algorithm may sometimes use *less* storage than that which is required by the sequential simulator. Time Warp with cancelback or artificial rollback always requires at least this much [57].

Of course, a Time Warp program will run very slowly if one only provides the absolute minimum amount of memory. The question of Time Warp performance as the amount of memory is varied has been studied [1]. An analytic model was developed that indicates that for homogeneous workloads, Time Warp requires relatively little memory to achieve good performance, i.e., performance with unlimited memory. In particular, this work indicates that four to five buffers per processor (where a buffer holds a state vector and an event) beyond the amount required for sequential execution achieves performance that is comparable to Time Warp with unlimited memory. This model was validated by experimental measurements of an operational implementation of Time Warp augmented with cancelback.

Further, an experimental study has examined the performance/memory tradeoff using several non-homogeneous workloads, and specifically, workloads designed to have some number of overly optimistic processes that advance, more or less unthrottled, into the simulated future, constrained only the the amount of memory in the system [18]. This provides a clear stress case for any Time Warp system. This study found that Time Warp, augmented with cancelback, can efficiently execute such asymmetric workloads using only a modest amount of memory beyond that required for sequential execution (somewhat more than the symmetric workload case, however), provided the salvage parameter (amount of memory reclaimed when the system runs out) is appropriately set. It was found that setting the salvage parameter too low (e.g., 1 or 2) causes poor performance if the system is memory bound, and setting it too high (the maximal setting will delete everything except that required for sequential execution) also degrades performance because correct computations are unnecessarily rolled back. Between these two extremes, however, performance appears to be relatively insensitive to the salvage parameter setting. Further, it was discovered that an inefficient implementation of the event list (i.e., the input queue) in each process, e.g., a linear list, can have a dramatic, detrimental affect on the performance of the system in limited memory situations.

7.4 Future Directions

Although much has been learned with respect to techniques to control memory utilization in optimistic protocols, important, unanswered, questions remain. Although experimental data provide useful insights as to how controls such as the "salvage" parameter should be set, no mathematical models yet exist to definitively answer this question. Further, although much work has been completed in the context of Time Warp, the performance/memory properties of conservative protocols have not been extensively studied. Mechanisms to ensure storage optimal execution for conservative protocols have not yet been developed.

In Time Warp, fossil collection and GVT computations are used to commit any irrevocable operations, e.g., I/O. Thus far, most of the work in parallel simulation has been focused on simulators that have relatively little I/O. When parallel simulation is used in interactive simulations, rapid commitment of events (and thus GVT computations) becomes critical. The adequacy of parallel simulation techniques, and GVT computation and fossil collection in particular, have not been widely studied in this context.

8 Conclusions

Parallel simulation is a rapidly growing area of research, with significant potential for increasing the size and complexity of models considerable by users to be simulatable in a reasonable amount of time. The field is developing rapidly, growing in many directions. In this paper we give a snapshot of the state of the art, in 1992, of six areas : synchronization protocols, mathematical performance analysis, time parallelism, hardware support, load balancing, and dynamic memory management. For each topic we have identified what we feel are relevant and important directions for future research.

Acknowledgements

The contribution of David Nicol was supported in part by NASA grants NAG-1-1060 and NAG-1-995, NSF grants ASC 8819373 and CCR-9201195. The contribution of Richard Fujimoto was supported in part 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, and NSF grant CCR-8902362.

References

[1] I. F. Akyildiz, L. Chen, S. R. Das, R. M. Fujimoto, and R. Serfozo. Performance analysis of Time Warp with limited memory. *Proceedings of the 1992 ACM SIGMETRICS Conference on Measuring*

and Modeling Computer Systems, 20(1), May 1992.

[2] H. Ammar and S. Deng. Time warp simulation using time scale decomposition. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 11–24. SCS Simulation Series, Jan. 1991.

[3] R. Ayani. A parallel simulation scheme based on the distance between objects. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2):113–118, March 1989.

[4] R. Baccelli and Miguel Canales. Parallel simulation of stochastic petri nets using recurrence equations. In *Proceedings of the 1992 SIGMETRICS Conference*, pages 257–258, Newport, Rhode Island, June 1992.

[5] D. Ball and S. Hoyt. The adaptive Time-Warp concurrency control algorithm. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):174–177, January 1990.

[6] S. Bellenot. Global virtual time algorithms. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):122–127, January 1990.

[7] S. Bellenot. State skipping performance with the time warp operating system. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 53–64. SCS Simulation Series, Jan. 1992.

[8] B. Berkman and R. Ayani. Parallel simulation of multistage interconnection networks on a simd computer. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 133–140. SCS Simulation Series, Jan. 1991.

[9] J. V. Briner. Parallel mixed-level simulation of digital circuits using virtual time. Ph. D. Thesis, Duke University, Durham, N. C., 1990.

[10] R. E. Bryant. Simulation of packet communication architecture computer systems. MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[11] C. A. Buzzell, M. J. Robb, and R. M. Fujimoto. Modular VME rollback hardware for Time Warp. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):153–156, January 1990.

[12] W. Cai and S. J. Turner. An algorithm for distributed discrete-event simulation – the "carrier null message" approach. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):3–8, January 1990.

[13] K. M. Chandy. A survey of analytic models of rollback and recovery strategies. *IEEE Computer*, 8(5):40–47, May 1975.

[14] K. M. Chandy and R. Sherman. The conditional event approach to distributed simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2):93–99, March 1989.

[15] K. M. Chandy and R. Sherman. Space, time, and simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2):53–57, March 1989.

[16] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5):440–452, September 1979.

[17] A. I. Concepcion. A hierarchical computer architecture for distributed simulation. *IEEE Transactions on Computers*, C-38(2):311–319, February 1989.

[18] S. R. Das and R. M. Fujimoto. A performance study of the cancelback protocol for Time Warp. Technical Report GIT-CC-92/50, College of Computing, Georgia Institute of Technology, Atlanta, GA, October 1992.

[19] M. Davoren. A structural mapping for parallel digital logic simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2):179–182, March 1989.

[20] E. DeBenedictis, S. Ghosh, and M.-L. Yu. A novel algorithm for discrete-event simulation. *Computer*, 24(6):21–33, June 1991.

[21] P.M. Dickens. *Analysis of The Aggressive Window Synchronization Protocol*. PhD thesis, University of Virginia, December 1992.

[22] D.M.Nicol. Performance bounds on parallel self-initiating discrete-event simulations. *ACM Trans. on Modeling and Computer Simulation*, 1(1):24–50, January 1991.

[23] S. Eick, A. Greenberg, B. Lubachevsky, and A. Weiss. Synchronous relaxation for parallel simulations with applications to circuit-switched networks. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 151–162. SCS Simulation Series, Jan. 1991.

[24] R. Felderman and L. Kleinrock. An upper bound on the improvement of asynchronous versus sychronous distributed processing. In *Distributed Simulation*, volume 22, pages 131–136. SCS Simulation Series, Jan. 1990.

[25] R. Felderman and L. Kleinrock. Bounds and approximations for self-initiating distributed simulation without lookhead. *ACM Trans. on Modeling and Computer Simulation*, 1(4), October 1991.

[26] R. Felderman and L. Kleinrock. Two processor time warp analysis: Some results on a unifying approach. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 3–10. SCS Simulation Series, Jan. 1991.

[27] M. A. Franklin, D. F. Wann, and K. F. Wong. Parallel machines and algorithms for discrete-event simulation. *Proceedings of the 1984 International Conference on Parallel Processing*, pages 449–458, August 1984.

[28] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.

[29] R. M. Fujimoto. The virtual time machine. *International Symposium on Parallel Algorithms and Architectures*, pages 199–208, June 1989.

[30] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[31] R. M. Fujimoto, J. Tsai, and G. Gopalakrishnan. Design and evaluation of the rollback chip: Special purpose hardware for Time Warp. *IEEE Transactions on Computers*, 41(1):68–82, January 1992.

[32] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3):61–67, July 1988.

[33] B. Gaujal, A. Greenberg, and D. Nicol. A sweep algorithm for massively parallel simulation of circuit-switched networks. Technical Report ICASE Technical Report 92-30, ICASE, July 1992. To appear in *Journal of Parallel and Distributed Computing.*

[34] E. Gelenbe. On the optimum checkpoint interval. *Journal of the ACM*, 26(4):259–270, April 1979.

[35] P. I. Georgiadis, M. P. Papazoglou, and D. G. Maritsas. Towards a parallel simula machine. *Proceedings of the 8th Annual Symposium on Computer Architecture*, 9(3):263–278, May 1981.

[36] K. Ghosh and R. M. Fujimoto. Parallel discrete event simulation using space-time memory. *Proceedings of the 1991 International Conference on Parallel Processing, Vol. 3*, 3:201–208, August 1991.

[37] D.W. Glazer. *Load Balancing Parallel Discrete-Event Simulations.* PhD thesis, McGill University, May 1992.

[38] A. Goldberg. Virtual time synchronization of replicated processes. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 107–116. SCS Simulation Series, Jan. 1992.

[39] G. Gopalakrishnan and R. M. Fujimoto. Design and verification of the rollback chip using HOP: A case study of formal methods applied to hardware design. Technical Report UUCS-91-015, Dept of Computer Science, Univ. of Utah, September 1991.

[40] A.G. Greenberg, B.D. Lubachevsky, and I. Mitrani. Algorithms for unboundedly parallel simulations. *ACM Trans. on Comp. Systems*, 9(3):201–221, 1991.

[41] A. Gupta, I. F. Akyildiz, and R. M. Fujimoto. Performance analysis of Time Warp with multiple homogenous processors. *IEEE Transactions on Software Engineering*, 17(10):1013–1027, October 1991.

[42] P. Heidelberger and D. Nicol. Conservative parallel simuation of continuous time markov chains using uniformization. Technical Report IBM Research Report RC-16780, IBM Research Division, April 1991. To appear in *IEEE Trans. on Parallel and Distributed Systems.*

[43] P. Heidelberger and H. Stone. Parallel trace-driven cache simulation by time partitioning. Technical Report RC 15500, IBM Research, February 1990.

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

[45] D. R. Jefferson. Virtual time II: Storage management in distributed simulation. *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 75–89, August 1990.

[46] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Shevon. Optimization by simulated annealing: An experimental evaluation: Part I, graph partitioning. *Operations Research*, pages 865–892, 1989.

[47] S. A. Kravitz and B. D. Ackland. Static vs. dynamic partitioning of circuits for a MOS timing simulator on a message-based multiprocessor. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3):136–140, July 1988.

[48] D. Kumar and S. Harous. An approach towards distributed simulation of timed petri nets. In *Proceedings of the 1990 Winter Simulation Conference*, pages 428–435, New Orleans, LA., December 1990.

[49] F. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, pages 32–38, January 1987.

[50] Y.-B. Lin. Memory management algorithms for optimistic parallel simulation. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24. SCS Simulation Series, Jan. 1992.

[51] Y.-B. Lin and E. Lazowska. Optimality considerations of "time warp" parallel simulation. In *Distributed Simulation*, volume 22, pages 29–34. SCS Simulation Series, Jan. 1990.

[52] Y-B. Lin and E. D. Lazowska. Determining the global virtual time in a distributed simulation. Technical Report 90-01-02, Dept. of Computer Science, University of Washington, Seattle, Washington, 1989.

[53] Y-B. Lin and E. D. Lazowska. Reducing the state saving overhead for Time Warp parallel simulation. Technical Report 90-02-03, Dept. of Computer Science, University of Washington, Seattle, Washington, February 1990.

[54] Y-B. Lin, E. D. Lazowska, and J-L. Baer. Conservative parallel simulation for systems with no lookahead prediction. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):144–149, January 1990.

[55] Y.-B. Lin and E.D. Lazowska. A study of Time Warp rollback mechanisms. *ACM Trans. on Modeling and Computer Simulation*, 1(1):51–72, January 1991.

[56] Y.-B. Lin and E.D. Lazowska. A time-division algorithm for parallel simualtion. *ACM Trans. on Modeling and Computer Simulation*, 1(1):73–83, January 1991.

[57] Y-B. Lin and B. R. Preiss. Optimal memory management for Time Warp parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):283–307, October 1991.

[58] R. Lipton and D. Mizell. Time Warp vs. Chandy-Misra: A worst-case comparison. In *Distributed Simulation*, volume 22, pages 137–143. SCS Simulation Series, Jan. 1990.

[59] B. Lubachevsky, A. Weiss, and A. Shwartz. An analysis of rollback-based simulation. *ACM Trans. on Modeling and Computer Simulation*, 1(2):154–192, April 1991.

[60] B. D. Lubachevsky. Scalability of the bounded lag distributed discrete event simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2):100–107, March 1989.

[61] B. D. Lubachevsky, A. Shwartz, and A. Weiss. Rollback sometimes works ... if filtered. *1989 Winter Simulation Conference Proceedings*, pages 630–639, December 1989.

[62] V. Madisetti, D. Hardaker, and R. Fujimoto. The mimdix operating system for parallel simulation. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 65–74. SCS Simulation Series, Jan. 1992.

[63] V. Madisetti, J. Walrand, and D. Messerschmitt. Wolf: A rollback algorithm for optimistic distributed simulation systems. *1988 Winter Simulation Conference Proceedings*, pages 296–305, December 1988.

[64] J. Briner, Jr. Fast parallel simulation of digital systems. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 71–77. SCS Simulation Series, Jan. 1991.

[65] P. Reynolds, Jr. An efficient framework for parallel simulations. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 167–174. SCS Simulation Series, Jan. 1991.

[66] J. Misra. Distributed-discrete event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[67] B. Nandy and W. Loucks. An algorithm for partitioning and mapping conservative parallel simulation onto multicomputers. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 139–146. SCS Simulation Series, Jan. 1992.

[68] L. M. Ni, C. W. Zu, and T. B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, October 1985.

[69] D. Nicol. Optimistic barrier synchronization. Technical Report ICASE Report 91-34, ICASE, July 1992.

[70] D. Nicol, A. Greenberg, B. Lubachevsky, and S. Roy. Massively parallel algorithms for trace-driven cache simulation. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 3–11. SCS Simulation Series, Jan. 1992.

[71] D. Nicol and P. Heidelberger. Optimistic parallel simuation of continuous time markov chains using uniformization. Technical Report IBM Research Report RC-17932, IBM Research Division, April 1992. To appear in *Journal of Parallel and Distributed Computing.*

[72] D. Nicol and P. Heidelberger. Parallel simuation of markovian queueing networks using adaptive uniformization. Technical Report IBM Research Report RC-18403, IBM Research Division, April 1992.

[73] D.M. Nicol and P.F Reynolds, Jr. Optimal dynamic remapping of data parallel computations. IEEE Trans. on Computers, 39(2):206–219, February 1990.

[74] D. Nicol and S. Roy. Parallel simulation of timed petri nets. In *Proceedings of the 1991 Winter Simulation Conference*, pages 574–583, Phoenix, Arizona, December 1991.

[75] D.M. Nicol. *The Automated Partitioning of Simulations for Parallel Execution.* PhD thesis, University of Virginia, August 1985.

[76] D.M. Nicol. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, April 1993. To appear. Available as technical report 90-20 from ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

[77] D.M. Nicol and P.F. Reynolds, Jr. A statistical approach to dynamic partitioning. In *Distributed Simulation 85*, volume 15, pages 53–56. SCS Simulation Series, 1985.

[78] C. Pancerella. Improving the efficiency of a framework for parallel simulations. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 22–32. SCS Simulation Series, Jan. 1992.

[79] B. Preiss, W. Loucks, I. MacIntyre, and J. Field. Null message cancellation in conservative distributed simulation. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 33–38. SCS Simulation Series, Jan. 1991.

[80] B. Preiss, I. MacIntyre, and W. Loucks. On the trade-off between time and space in optimistic parallel discrete-event simulation. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 33–42. SCS Simulation Series, Jan. 1992.

[81] B. R. Preiss. The Yaddes distributed discrete event simulation specification language and execution environments. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2):139–144, March 1989.

[82] P. L. Reiher and D. Jefferson. Dynamic load management in the Time Warp Operating System. *Transactions of the Society for Computer Simulation*, 7(2):91–120, June 1990.

[83] H.S. Ross. *Stochastic Processes*. Wiley, New York, 1983.

[84] B. Samadi. Distributed simulation, algorithms and performance analysis. Ph. D. Thesis, University of California, Los Angeles, 1985.

[85] L. Soule and A. Gupta. An evaluation of the Chand-Misra-Byrant algorithm for digital logic simulation. *ACM Trans. on Modeling and Computer Simulation*, 1(4), October 1991.

[86] J. Steinman. Speedes:synchronous parallel environment for emulation and discrete event simulation. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 95–103. SCS Simulation Series, Jan. 1991.

[87] 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):38–43, March 1989.

[88] G. Thomas and J. Zahorjan. Parallel simulation of performance petri nets: Extending the domain of parallel simulation. In *Proceedings of the 1991 Winter Simulation Conference*, pages 564–573, Phoenix, Arizona, December 1991.

[89] S. Turner and M. Xu. Performance evaluation of the bounded time warp algorithm. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 117–128. SCS Simulation Series, Jan. 1992.