PERFORMANCE OF TIME WARP UNDER SYNTHETIC WORKLOADS

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

Abstract

The parallel hold (*PHOLD*) model is proposed for performing performance analyses of *parallel* discrete event simulation programs. The PHOLD model is based on the hold model that is widely used to evaluate the performance of sequential event list algorithms.

Using the PHOLD model, an empirical performance evaluation of the Time Warp mechanism was performed. Performance measurements of Time Warp executing on a shared-memory multiprocessor were completed using both saturated (more parallelism than processors) and unsaturated (less parallelism than processors) workloads. Workload parameters that were tested included computation grain size and variance, spatial locality, temporal locality, and lookahead.

Based on these experiments, it appears that Time Warp performance is very robust across a wide range workloads. Speedups as high as 32 using 64 processors were obtained under fairly adverse conditions (a fully connected network containing no lookahead and minimum timestamp increment of zero). Workloads containing more favorable parameters obtained speedups as high as 54 on 64 processors.

1 Introduction

The heavy computational demands of large discrete event simulation programs, coupled with the increasing availability of parallel computers, has heightened interest in parallel discrete event simulation (PDES). Specifically, this paper is concerned with simulations of *asynchronous* systems where the synchronous, lock step mode of operation degenerates to largely sequential execution because too few simulator events occur at a single point in simulated time. The exploitation of parallelism in these applications has been elusive because the global notion of time does not easily map to a distributed computer; sophisticated clock synchronization algorithms are required to ensure that cause-and-effect relationships are faithfully reproduced by the simulator.

Jefferson's Time Warp mechanism based on the Virtual Time paradigm offers great potential as a "general purpose" PDES strategy [4]. Recently, numerous successes have been reported using Time Warp to speed up simulations, e.g., see [2, 7]. These results are encouraging, but provide only limited insight into the behavior of Time Warp as a function of characteristics of the simulation application.

The work described here characterizes the performance of Time Warp for various *synthetic* workloads covering a wide range of application specific parameters. By *synthetic* we mean the workloads do not attempt to simulate any real-world system, but rather, attempt to emulate essential characteristics of a wide range of "typical" simulations that arise in practice. This allows one to assess performance based on certain characteristics of the application without becoming entangled in the details and peculiarities of a specific application. The workloads are designed to be relatively simple in order to facilitate understanding the behavior of the simulation algorithm. Here, we use synthetic workloads to test the *robustness* of Time Warp performance as various workload parameters are changed.

2 The Parallel Hold Workload Model

Because parallel simulation algorithms perform the same logical function as the event list in a sequential simulator (ensuring proper sequencing of events), it is natural to examine the workload models that are used there. In sequential simulation, the *hold model* has become the most widely used approach for evaluating the performance of event list (priority queue) implementations (e.g., see [5]). In this model, the event list consists of a fixed number of unprocessed events, each containing a timestamp to indicate its priority. The iterative step used by the model is to first dequeue the smallest timestamped event, and then enqueue a single new event using a timestamp equal to that of the removed event plus some increment selected from a probabilistic distribution. The parameters of this model are the size of the event list, the distribution of the timestamp increment function, the initial distribution of the timestamps of events in the queue, and the initial shape of the data structure holding the events.

Though adequate for evaluating *sequential* simulations, the hold model does not consider certain characteristics that greatly affect the performance of *parallel* simulation programs. In particular, the model says nothing concerning the spatial characteristics of the application,

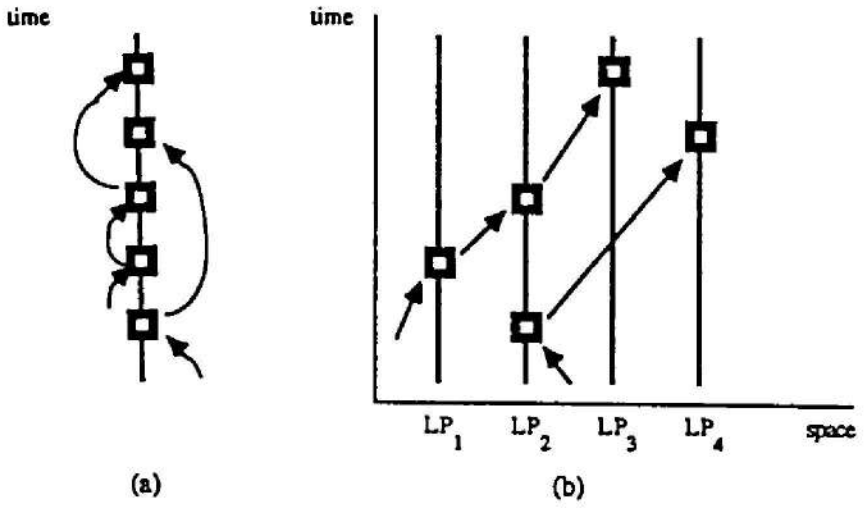


Figure 1: Space-time diagram. (a) one dimensional. (b) two dimensional.

which greatly impacts the amount of parallelism that is available. Therefore, the hold model must be extended to accommodate analyses of PDES algorithms.

We propose the *parallel hold model* (PHOLD) for this purpose. We previously reported use of an earlier version of this workload model to evaluate the performance of conservative simulation algorithms based on deadlock avoidance and deadlock detection and recovery [1].

The sections that follow first define the workload model with the aid of space-time diagrams. A specific instance of the model is developed for evaluating the Time Warp mechanism. Finally, results of the performance study for Time Warp are presented.

2.1 Space-Time Diagrams

We use space-time diagrams (time refers to simulated time) to illustrate the behavior of the simulation. From this perspective, the sequential hold model is *one dimensional*, with simulated time as the dimension. The diagram in figure 1a depicts the operation of this model. Each arc represents the scheduling of an event. The time coordinate of the head and tail denote, respectively, the simulated time at which the event is to occur and the simulated time at which it was scheduled. We refer to a sequence of arcs $E_1, E_2, \ldots$ where processing event E_i causes E_{i+1} to be scheduled, as a *thread*. The head of E_i will always coincide with the tail of E_{i+1} in the space-time diagram. When the sequential hold model is used, the diagram in figure 1a can be viewed as M threads mapped onto the linear time scale, where M is the number of elements in the event list. In actual simulations, one event can schedule any number of other events; this leads to trees rather than the linear threads described above.

The PHOLD model extends the space-time graph to higher dimensions. A two dimensional model is depicted in figure 1b. The spatial axis is discrete, and represents the state of the entire simulation. Most existing PDES algorithms are based on process oriented paradigms, so we will assume each spatial coordinate represents a distinct logical process. In particular, let us assume that there are N logical processes $LP_0 \ldots LP_{N-1}$ and process LP_i is mapped to spatial coordinate i.

Like the hold model, PHOLD assumes there is a fixed number of threads, with the exact number specified as a parameter. An arc extending from space-time coordinate (s, T_s) to (r, T_r) denotes the fact that an event at LP_s and simulated time T_s sent an event message to LP_r with timestamp T_r, where $T_r \geq T_s$.[1] In Time Warp, T_s and T_r denote the send and receive time of the message, respectively.

The benchmarks used to evaluate Time Warp are based on a minor extension of the two dimensional model: *two* dimensions are used to represent the spatial characteristics of the simulation rather than one. Each logical processes occupies a unique, integral Cartesian coordinate position in the two-dimensional plane. The point (S_x, S_y, T) de-

[1] For some simulation mechanisms, it may be necessary to use a strict inequality, i.e., $T_r > T_s$.

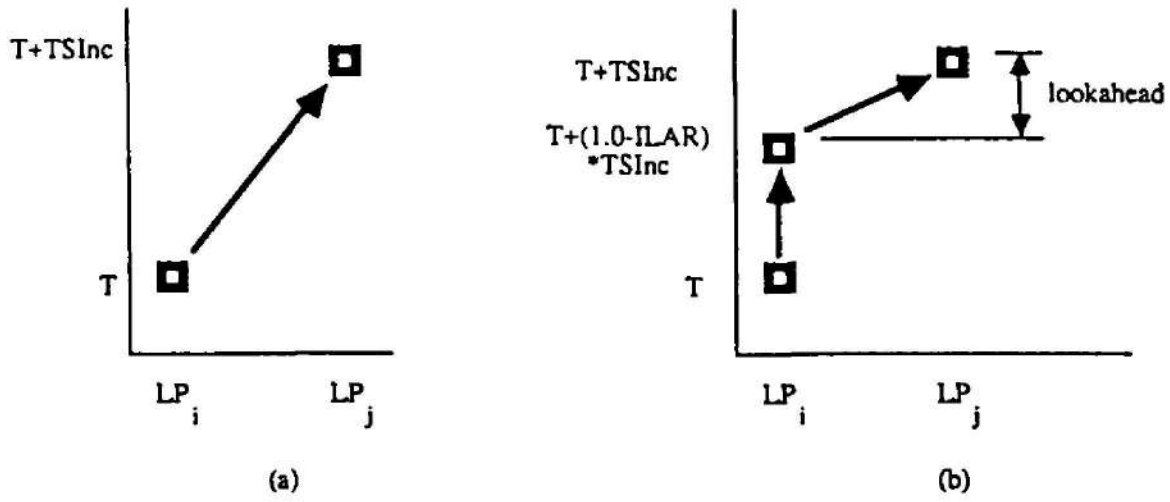


Figure 2: Lookahead in the PHOLD model. (a) an event when lookahead is not considered. (b) a model that considers lookahead.

notes the state of the logical process at Cartesian coordinate (S_x, S_y) at simulated time T. The *movement function* parameter of the model governs the incremental movement of an event in space, just as the timestamp increment function governs the movement in simulated time. The extension of space to two dimensions better reflects simulations of many real world systems, e.g., networks, and interacting physical entities moving across a plane, than the one dimensional models.

2.2 Model Parameters

The PHOLD workload model contains the following parameters:

1. *number of logical processes:* this provides an upper bound on the amount of parallelism that is available.

2. *message population:* this positive integer indicates the number of causality threads migrating through the space-time diagram. It is identical to the "event list size" parameter of the hold model.

3. *timestamp increment function:* this function is identical to that used in the sequential hold model. Given an event E_1 at point (S_x, S_y, T) in the space time diagram, and the state of the corresponding logical process, it returns the (receive) timestamp of the new event scheduled by E_1.

4. *movement function:* this function describes the spatial movement of threads in the space-time diagram. Given an event E_1 at point (S_x, S_y, T) in the diagram, and the state of the corresponding logical process, it returns the spatial coordinate of the event scheduled by E_1.

5. *computation grain:* this function defines the amount of computation required to process a single event, excluding the time required to schedule the next event. In general, the granularity depends on the event and process state.

6. *initial configuration:* this function defines the location (in space and time) of each initial event before the simulation begins.

It is assumed that all user computation performed by the workload model is the result of processing an event; no computations may be "spontaneously" created. One other important aspect of the workload model is that we do not explicitly model event cancellations by the application program. We model simulators programmed to allow the *possibility* of cancellation, though not the cancellation itself, through a *lookahead* parameter. This aspect of the workload model will be described momentarily. Lookahead is incorporated into the model through the timestamp increment and movement functions.

2.3 Lookahead

It is widely recognized that the degree to which processes can "look ahead" into the simulated time future can have a dramatic impact on the performance of PDES algorithms [1]. If a process at simulated time T can predict with complete certainty all events it will generate up to simulated time $T + L$, we say the process has lookahead ability L.

We characterize the lookahead of the process by a parameter called the *inverse lookahead ratio* (ILAR), where $0 \leq ILAR \leq 1$. This quantity is essentially the reciprocal of the lookahead ratio parameter defined in [1]. ILAR is defined on an event-by-event basis as the lookahead divided by the timestamp increment assigned to the scheduled event; it is assumed that the lookahead is no greater than the timestamp increment (having more lookahead provides no benefit in the model). In order to send an event message with timestamp increment $TSInc$, the simulated time of the process sending the message must have advanced in simulated time up to within $ILAR * TSInc$ (the process's lookahead) of the timestamp of the message.

Distribution	Expression	Grain μsec	Rate 1 ev/sec	Rate 2 ev/sec
Deterministic	1.0	190	1827	1857
Biased	0.9 + 0.2 rand	258	1362	1630
Uniform	rand	250	1271	1599
Exponential	-ln(rand)	339	1137	1405
Bimodal	0.95238 rand + if rand $\leq$ 0.1 then 9.5238 else 0	260	1303	1605

Table 1: Timestamp Increment Distributions. rand returns a random value uniformly distributed between 0 and 1.

Lookahead is illustrated graphically in figure 2. Figure 2a shows a single event in a space-time diagram without any consideration to lookahead. Figure 2b reflects the operation of the simulator in processing this event when lookahead is considered. The logical process first schedules an event to itself with timestamp increment $(1.0 - ILAR)TSInc$ because at that point, it can "see" sufficiently far into the future to schedule the subsequent event. The thread is continued by a second event with timestamp increment $ILAR*TSInc$ (the lookahead of the process). Assume the three event times in figure 2b are T, $T + (1.0 - ILAR)TSInc$, and $T + TSInc$. The self event is essential because events containing a timestamp in the interval $[T, (1.0 - ILAR)TSInc]$ could affect the event scheduled to occur at $T + TSInc$. Here, to simply the model, we assume this "effect" *excludes* the possibility of changing the timestamp on the event at $T + TSInc$.

The extreme points of the inverse lookahead ratio (0.0 and 1.0) are noteworthy because they represent cases that frequently arise in practice. For example, an ILAR of 1.0 (extremely good lookahead) arises in a queueing network simulation using a first-come-first-serve service discipline. There, the arrival event of a job at the next server in a tandem queue can be scheduled as soon as the arrival event at the current server is processed because the computation of the departure time is invariant to the arrival of subsequent jobs. Conversely, an ILAR of 0.0 corresponds to the case when one first schedules a departure event to one's self before scheduling the arrival, a common way of programming a queueing network simulator. Simulators modeling preemption also behave in a similar fashion, although in that case, the preempting event changes the timestamp of the event it preempts, a behavior that is disallowed in our simplified workload model.

In our study of the Time Warp mechanism, we assume the amount of the timestamp increment ($TSInc$ above) is selected from one of the following distributions (see table 1): deterministic, biased, exponential, uniform, or bimodal. The exponential and uniform distributions use a minimum value of 0.0. These distributions are commonly used in performance evaluations of sequential event list implementations of simulation programs and represent distributions often observed in practice [5].

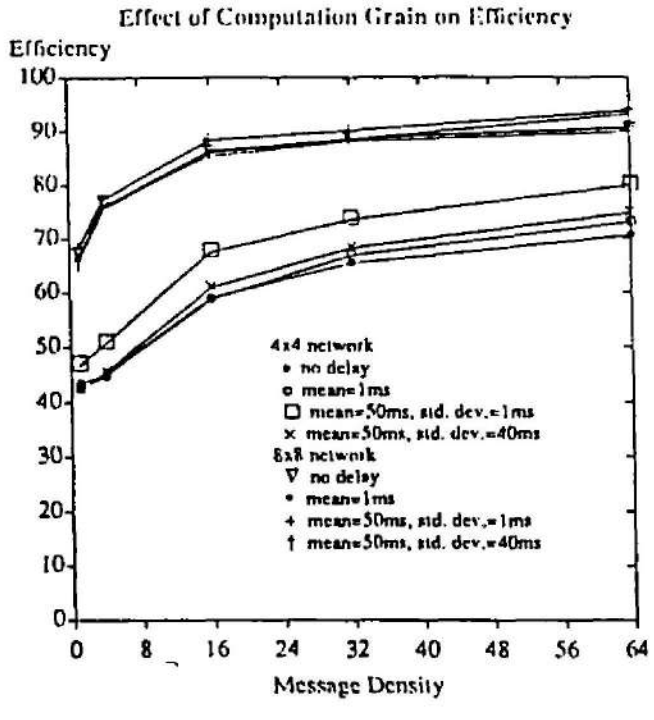
We also assume that initially, events are uniformly distributed among the logical processes. The initial timestamp distribution is selected according to the timestamp increment function.

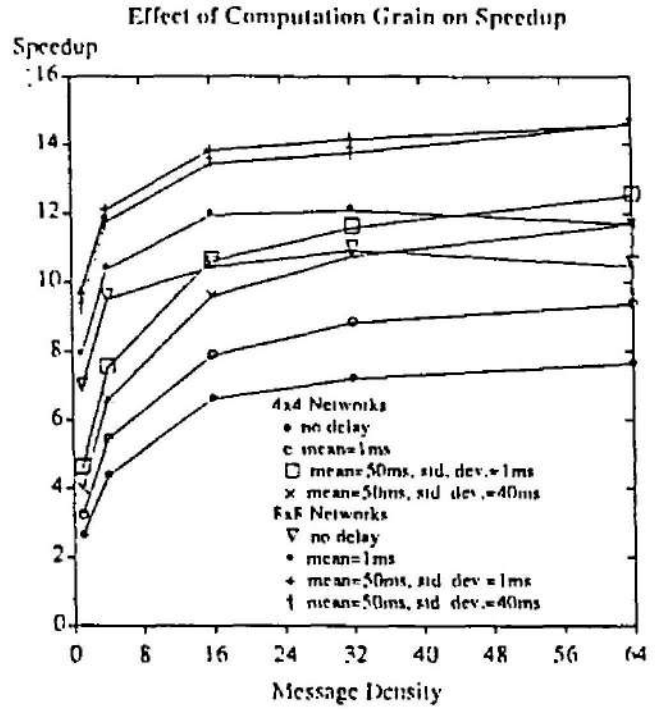
2.4 Spatial Locality

We define the movement function to allow spatial locality to be easily parameterized. Each process may only send messages to a fixed set of other processes called its *neighborhood*. We assume that the neighborhood for process LP_i consists of those processes that are closest to LP_i in the two dimensional coordinate plane. The size of the neighborhood controls the degree to which processes exhibit spatial locality.

The second aspect of the movement function is the selection of a specific process within the neighborhood. In our initial experiments,

(a)





(b)

Figure 3: Effect of varying the size and distribution of the event computation. (a) efficiency (b) speedup.

each process in the neighborhood is equally likely to be selected. Experiments using other selection functions are planned.

3 The Parallel Simulation Testbed

An 18 processor BBN Butterfly GP1000 multiprocessor was used in most of the experiments described here. Each node contains a 16 MHz MC68020 microprocessor, MC68881 floating point coprocessor, MC68851 paged MMU, 4 MBytes of main memory, and an AMD 2901 microcoded engine for interfacing the processor node to the switch. The interconnection switch is configured as an Omega network. Instructions that access non-local memory require approximately five times more time than those accessing local memory.

The GP1000 uses a version of the Mach operating system. Processors used for each experiment were dedicated to the application to avoid interference from other users. Care was taken to ensure that the amount of memory used by each processor was much less than the 4 MBytes available on each node in order to minimize perturbations due to the paging mechanism. Also, pages were pre-written during the initialization phase of the simulation to further reduce interference from the memory system. To evaluate the affect of operating system overhead, selected simulations were also performed on Chrysallis, a primitive operating system for the Butterfly that has minimal interference with the application program. Speedup figures under Chrysallis were typically 10% higher than those obtained under Mach, so we consider the performance data reported here to be conservative.

The logical processes mapped to a single processor were executed within a single Mach process to reduce the cost of context switches. After initialization is completed, all synchronization is performed using primitive atomic operations (e.g., atomic-add) that are provided by the Butterfly hardware.

The Time Warp system uses direct cancellation, an optimization using shared memory to streamline the cancellation of incorrect computations [2]. All measurements reported here use aggressive cancellation.

Processes are mapped to processors by partitioning the two dimensional spatial plane into a grid, and mapping the processes in each grid sector to a separate processor. This minimizes interprocessor communications when the application has good spatial locality. No dynamic load balancing or process migration is performed.

4 Performance Measurements

Performance measurements were made using the PHOLD model described above. The two principal metrics used are speedup relative to a sequential simulator and efficiency. Efficiency is defined as the number of events that were eventually committed (i.e., neither rolled back nor cancelled) divided by the total number of events executed. Except in the cases where the computation grain was artificially increased by inserting delays, each data point represents the execution of over a million committed events.

A set of "default" parameters were defined for initial benchmarks. The default settings assume the process topology is a toroid, and each neighbor is equally likely to be selected. The exponential timestamp increment is used, and good lookahead assumed. No delay is inserted to artificially increase the computation grain. Initial experiments varied a single parameter from the default setting, and later experiments varied many parameters in order to construct some very challenging benchmarks. Unless stated otherwise, all experiments use 16 processors on the GP1000.

4.1 The Sequential Simulator

The sequential simulator was developed by modifying the parallel simulator; all code related to parallel execution (e.g., locks for synchronization) was removed. Identical application code is used by both the parallel and sequential simulator.

The priority queue used by the sequential simulator is implemented using a splay tree [6]. Empirical evidence suggests that splay trees are among the fastest methods for implementing an event list [5]. The splay tree code used by Jones in his experiments was translated to C, and adapted for use in the testbed simulator. It uses the top-down version of the splay algorithm, and is optimized to efficiently execute event-list operations.

Table 1 indicates the amount of execution time required to process an event in the workload model for each of the timestamp increment functions. This number indicates the size of the computation grain in the "no delay" case. Also indicated are measurements of the number of events processed per second (the *event rate*) for event lists of size 16 and 4096. The event rate achieved by the *parallel* simulator is this figure multiplied by the speedup.[2] In addition to the event execution and splay tree operations, the sequential simulator requires approximately 250 microseconds overhead for each event to perform other miscellaneous functions, e.g., allocating storage from the free list, loop overhead, message copying (one per scheduled event), statistics collection, etc.

4.2 Varying the CPU Grain and Distribution

Our first set of experiments were designed to evaluate the effect of the computation grain size and distribution on performance. The computation time to process each event is selected from a random variable, and a busy wait loop was inserted to increase the total computation time of the event (excluding sending the event) up to this value. If the selected value is smaller than that required to otherwise process the event (as described above), no delay is inserted.

Four cases were examined: no delay inserted (i.e., the granularity corresponds to the figures reported in table 1), a computation grain selected from an exponentially distributed random variable with a mean of 1 millisecond, and grains selected from normally distributed random variables with mean of 50 milliseconds, and standard deviation of either 1 or 40 milliseconds.

The efficiency of the Time Warp mechanism for these computation grains is shown in figure 3a as the message density (message population divided by the number of processes) is varied. Experiments using 16 and 64 processes, i.e., 1 and 4 processes per processor, respectively, were performed. Increasing the computation grain from a few hundred microseconds to (on average) 50 milliseconds improves efficiency in the one process per processor case by as much as 10%, but less than 5% in the 4 process per processor case. The latter case achieves much higher efficiency figures because there is much more parallelism, so processors are more likely to be working on correct events.

In Time Warp, errors tend to propagate more rapidly relative to the speed of the correction (anti-messages) if the grain of computation is small. This accounts for the improvement in the efficiency of the algorithm with larger grains. However, the effect of computation grain is modest, indicating that the efficiency of the mechanism is largely unaffected by computation grain. The amount of parallelism is a much more important factor.

[2] Except when the event grain is artificially increased, or if ILAR is not 1.0. In the latter case, "self" events containing even smaller grains are also executed.

Number of Processes	ILAR=1.0		ILAR=0.0	
	dens=1	dens=16	dens=1	dens=16
16	5.6	11.3	4.0	5.9
64	19.1	42.7	12.9	19.6
256	71.4	160.0	46.2	72.0

Table 2: Average parallelism for fully connected network.

Speedup figures are shown in figure 3b. It can be seen that the computation grain has a much more significant effect on speedup than efficiency. This is because the larger computation grain tends to mask the differences in overhead between the sequential simulator and Time Warp. This behavior exists for *any* parallel computation, and is not unexpected. It is worth pointing out, however, that Time Warp achieves respectable speedups even with small grains of computation, and high variance in the grain size.

4.3 Average Parallelism

Here, *average parallelism* is defined as the total amount of time required to process events (excluding all simulator overheads, e.g., for synchronization) divided by the length of the critical path through the simulation. This indicates the amount of speedup that a hypothetical machine containing an infinite number of processors and *zero synchronization overhead* could obtain. The average parallelism figure reported here should be regarded as an upper bound on the speedup that can be achieved.[3]

A simulation tool developed at the Jet Propulsion Laboratory was used to determine the critical path length [3]. A version of the PHOLD workload using a fully connected network and fixed execution time per event was implemented. Average parallelism measurements of these workloads were then performed. Average parallelism for 16, 64, and 256 process simulations with message densities of 1 and 16 are shown in table 2. Data for ILAR values of 1.0 and 0.0 are shown.

Consider the data points in figure 3 that yielded the poorest performance, i.e., the simulations containing one process per processor, and a message density of 1. Here, speedups ranged from 2.6 when no delay is added, to 4.6 when event times are 50 milliseconds. The average parallelism is only 5.6, so under the circumstances, Time Warp is actually performing reasonably well. We make special mention of this simulation because it contains substantially *less* parallelism than the number of processors. This "unsaturated" case is considered particularly stressful for Time Warp because it provides incorrect computations ample opportunity to spread very rapidly.

4.4 Spatial Locality

The next set of experiments were designed to test the degree to which spatial locality affects Time Warp performance. For these experiments, the size of the neighborhood of each logical process was varied. Experiments ranging from the nearest neighbor (toroid) connection up to a fully connected network were performed.

Figure 4a shows the effect of varying spatial locality on efficiency. The horizontal axis indicates the size of the neighborhood divided by the number of logical processes in the network. The end points of each curve denote the toroid connection (the left end point) and a fully connected network (right end point). Processes do not send messages to themselves, accounting for the fact that the right end point does not lie at locality 1.0. The curves correspond to benchmarks using 1, 4, and 16 processes on each of the 16 processors. As can be seen, the efficiency of the Time Warp mechanism is virtually unaffected by changes in spatial locality. As in the previous set of experiments, efficiency is determined by the amount of parallelism in the application, which is largely controlled by the number of logical processes and the message density. The relative orderings of the curves in figure 4a are consistent with this observation.

Speedup curves are shown in figure 4b. Somewhat higher speedups are obtained when locality is high. This is because more messages are sent to processes on the same processor as the sender if locality is high, reducing the overhead for message communications. This effect is not observed in the 16 process simulation because each processor contains only one process, so all messages require non-local communications, regardless of the size of the neighborhood.

4.5 Varying the Timestamp Increment Function

The next round of experiments examined the effect of the timestamp increment function on performance. The timestamp increment affects *temporal* locality. Jefferson has hypothesized that good temporal locality enhances the performance of the Time Warp mechanism in a manner similar to the way locality improves virtual memory systems [4].

Figure 5a shows Time Warp efficiency as a function of message density for the 1 process and 4 processes per processor cases as the timestamp increment function is varied. In both cases, the uniform,

[3] Actually, one might be able to achieve somewhat higher speedups, depending on particulars of the implementation of the event list and the parallel simulator. Here, we consider this possibility to be remote because a reasonably efficient sequential simulator is used to derive speedup figures.

Distribution	Bias	16 processes		64 processes	
		den=1	den=16	den=1	den=16
Deterministic	1.0	8.3	15.0	30.4	57.0
Biased	0.97	8.2	14.7	30.3	57.3
Uniform	0.66	6.6	12.2	23.2	46.6
Exponential	0.50	5.6	11.3	19.1	42.7
Bimodal	0.13	4.5	12.0	15.5	45.7

Table 3: Average parallelism as timestamp distribution changes (fully connected network).

exponential, and bimodal distributions yield significantly lower efficiencies than the deterministic and biased distributions. The variation in efficiency as the timestamp increment function changes is often as much as 30% in the one process per processor case, but typically from 5 to 10% in the four processes per processor case. These results are consistent with Jefferson's assertion that good temporal locality improves performance (the deterministic and biased distributions exhibit very good temporal locality). We present an alternative explanation below.

The higher efficiency figures for the biased and deterministic distributions is a consequence of the fact that these distributions lead to greater parallelism in the PHOLD model. In the *sequential* hold model, the biased and deterministic distributions cause new events to tend to have timestamps larger than the other events already residing in the queue, so new events are usually inserted near the end of the event list, assuming ties are resolved in FIFO order. The behavior of the priority queue resembles that of a FIFO queue.

In the PHOLD model, this FIFO like behavior is preserved in the queue of each logical process. If event E_a on processor PE_a schedules a new event E_x on processor PE_b, and E_x contains a timestamp larger than any other event on PE_b, then E_a may be able to execute concurrently with these other events. On the other hand, if E_x contained a timestamp *smaller* than the events on PE_b, as would be the case if the queue were being used in a LIFO-like fashion, E_a would have to be processed *before* these events, precluding the possibility of parallel execution. Thus, "FIFO-like" queue behavior leads to better parallelism, accounting for the better performance of the biased and deterministic distributions.

This aspect of queue behavior is quantitatively described as *bias*, with a bias of 1.0 indicating purely FIFO behavior, and 0.0 indicating purely LIFO behavior [5]. The biases of these timestamp distributions are listed in table 3. The relative orderings of the efficiency curves are consistent with the biases of these distributions. The corresponding speedup curves for these experiments are shown in figure 5b.

Table 3 also shows the average parallelism of simulations using different timestamp increment functions (as before, a fully connected network is assumed). It is seen that the average parallelism generally increases with the bias of the distribution, as expected.

4.6 Effect of Lookahead

The next set of experiments were designed to evaluate the effect of lookahead on performance. The preceding experiments assumed an ILAR value of 1.0 (good lookahead), so each process could immediately generate a new event as soon as the "causing" event were received. With smaller ILAR values, each process must first send a message to itself (and receive it) before generating the event, as was described earlier.

Efficiency curves for the 1, 4, and 16 processes per processor cases at message densities of 1 and 16 are shown in figure 6a. The corresponding speedup curves are shown in figure 6b. As can be seen, lookahead has a modest impact on performance except for the case when the message density is high, and there is zero lookahead (ILAR equal to 0.0), where a significant degradation occurs.

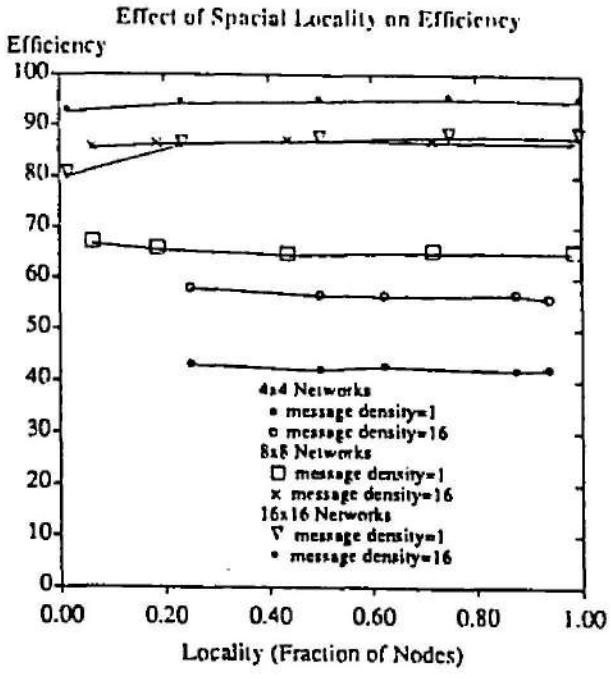
In the zero lookahead case, the process computes a timestamp increment $TSInc$, and then sends a message to itself using that value. The send and receive timestamps on the "self" message are T and $T + TSInc$, respectively, where T is the time of the original event. When the self message is received, a new message with timestamp increment of zero is created, and sent to another process.

If a straggler message later arrives in the simulated time interval $[T, T + TSInc]$, it will roll back the "self" message and all of the subsequent computations that were spawned, assuming aggressive cancellation is used. Such a straggler is more likely to occur if the message density is high, so, as seen in figure 2, the degradation is much more severe at high densities.

Average parallelism measurements quantitatively characterize this behavior (see table 2). The poor lookahead case with message density of 1 contains only about two-thirds as much parallelism as the case where lookahead is good, and only *half* as much (or less) when the density is 16. Therefore, one would expect a certain amount of degradation with any parallel simulation mechanism, particularly in the high message density case. The observed speedup figures are not too surprising when considering the amount of parallelism that is available, and the use of only 16 processors and small grained events.

The Time Warp degradation is more severe when there are fewer processes per processor. This is also not surprising when one considers the amount of parallelism that is available. The one process per processor case already contains insufficient parallelism to keep all of

(a)



(b)

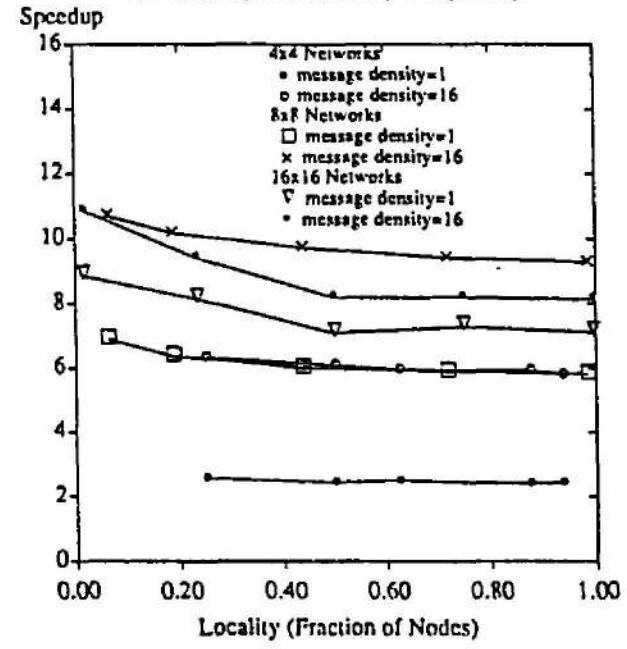
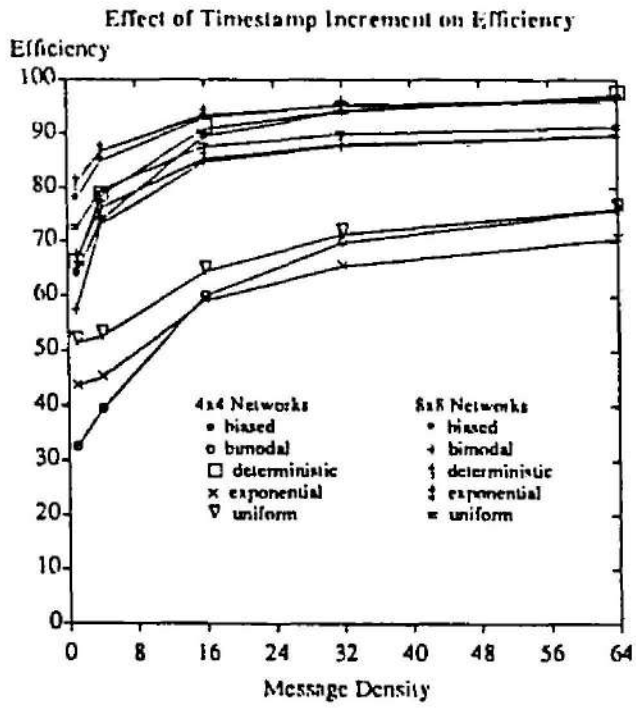


Figure 4: Effect of varying spatial locality. The left end point of each curve represents a toroidal interconnect, and the right end point a fully connected network. (a) efficiency (b) speedup.

(a)



(b)

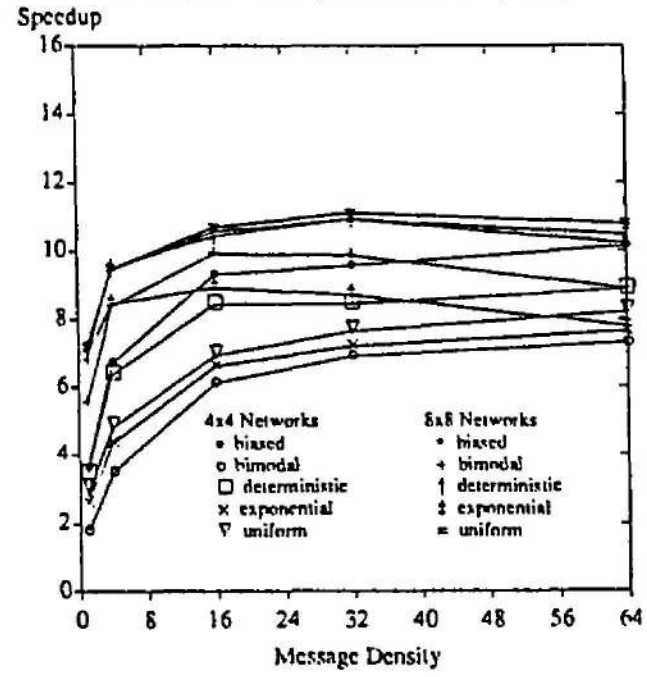


Figure 5: Effect of varying the timestamp increment function. (a) efficiency (b) speedup.

processors busy, even if lookahead is good. Therefore, loosing half of this limited parallelism translates almost directly into loosing half of the available speedup. On the other hand, configurations containing a larger number of processes contain some "excess" parallelism that is not being exploited, due to the limited number of processors. Thus, it is not surprising that they encounter a less severe performance degradation.

4.7 Mixed Parameter Settings

The experiments that have been performed so far were constructed by modifying a single control variable while other parameters remained at "default" values. We relax this constraint in our final set of experiments, and vary many parameters to examine the additive effect of combining many unfavorable characteristics into a single workload. Specifically, we consider the case of a fully connected network with no lookahead (i.e., ILAR is 0). An exponential timestamp distribution is used (poor temporal locality), and *no* delay is inserted to artificially inflate the computation grain. Because this workload combines (1) a topology with high connectivity, (2) minimum timestamp increment of zero, (3) no lookahead, and (4) modest computation grain, it represents a particularly challenging test case for *conservative* simulation mechanisms.

These experiments were performed on a Butterfly I multiprocessor housed at the University of Maryland in order to gain access to a larger number of nodes. This machine is an earlier version of the Butterfly that is based on the 68000 microprocessor (rather than the 68020), and contains no floating point coprocessor. The latter fact tends to increase the process granularity because floating point operations must be performed in software.

The efficiency and speedup curves for 64 and 256 process simulations are shown in figures 7a and 7b respectively, as the number of processors is varied from 8 to 64. Experiments were performed using message densities of 1 and 16. The 64 process simulation yields maximum speedups of 10.4 (message density of 1) and 12.5 (density of 16) at 64 processors (one process per processor). As seen in table 2, these two simulations contain average parallelisms of 12.9 and 19.6, respectively, accounting for the modest observed speedup. For the 256 process simulations, speedups of 25.4 (density 1) and 31.8 (density 16) were obtained, in spite of the adverse conditions under which the simulation operates.

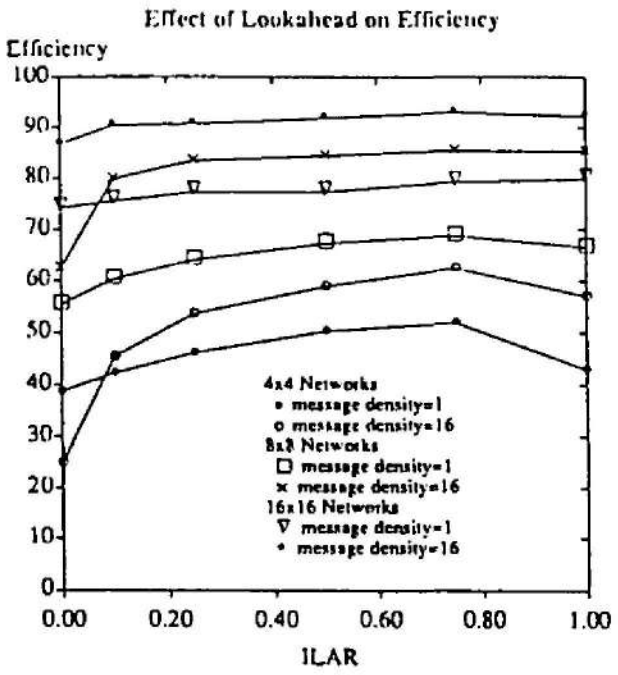
For comparison, we also performed experiments with good lookahead (ILAR of 1.0) and good spatial locality (toroidal interconnect). Other parameters remained at their previous, unfavorable values. Results of these experiments are also shown in figures 7a and b. This modification (particularly the addition of good lookahead) significantly improves performance. Speedups as high as 54.3 (using 64 processors) were obtained.

5 Conclusions

The two principal contributions of this paper are to extend the hold model to allow it to be used for evaluating parallel simulation algorithms, and the use of this model to empirically evaluate the performance of the Time Warp mechanism. The principal extension to the hold model was the addition of consideration for *spatial* aspects of the workload.

Exercising the Time Warp mechanism on benchmarks constructed from the PHOLD model indicate that Time Warp performance is very robust across many different workloads. The *efficiency* of the mechanism is relatively insensitive to the size of the CPU grain and its variance, as well as the degree of spatial locality. Absolute performance (as measured in speedup) does show some variance to these parameters, due to multiprocessor overhead and additional interprocessor communications respectively. Such factors affect all parallel programs, so they are not unexpected.

(a)



(b)

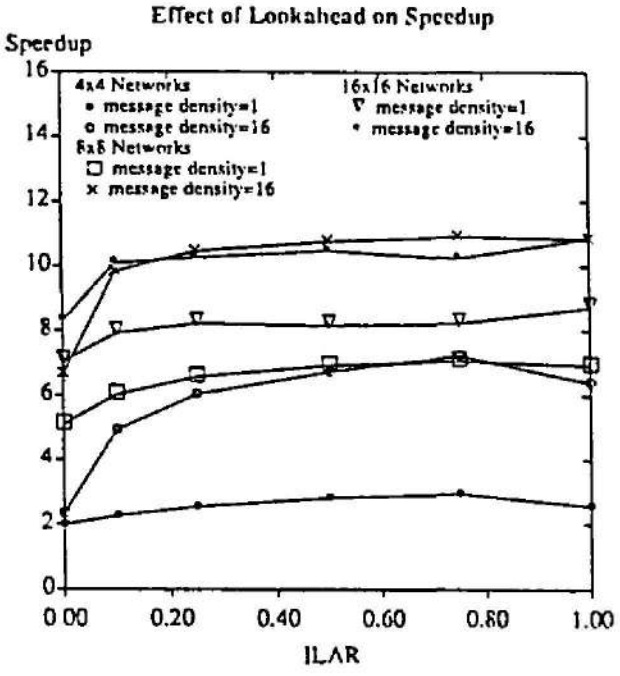
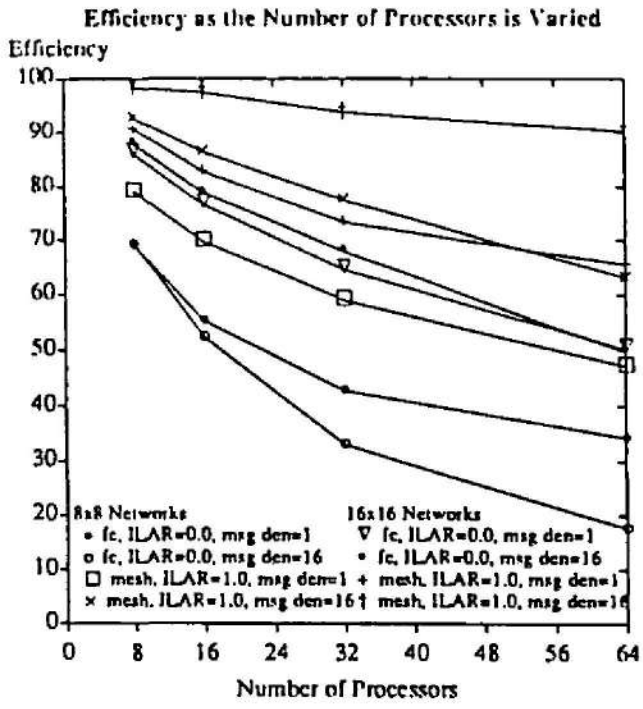


Figure 6: Effect of varying lookahead. (a) efficiency (b) speedup.

(a)



(b)

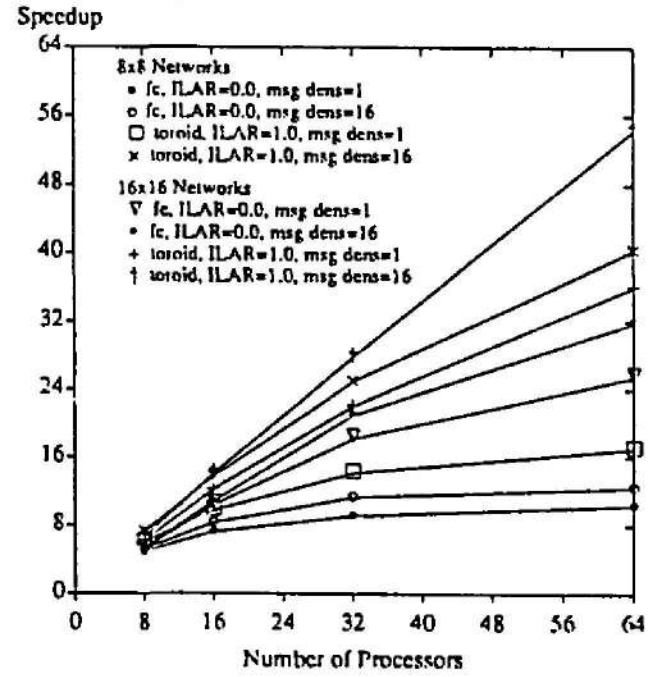


Figure 7: Effect of combining several unfavorable characteristics into a single workload. (a) efficiency (b) speedup.

The timestamp increment function was noted to have a significant impact on performance. This is attributed to differences in temporal locality which affect the amount of parallelism that is available in the simulation. Similarly, lookahead can have a significant impact on performance, particularly when there is none. This is again attributed to a reduced level of parallelism if lookahead is poor. Finally, we constructed a workload by combining the parameter settings for which Time Warp had the most trouble. Under such adverse conditions as a fully connected network with no lookahead, Time Warp was able to extract much of the available parallelism, even when there was much less parallelism than the number of processors, and produce a significant speedup.

Overall, our experiments to date have indicated that Time Warp performance is largely determined by the amount of parallelism in the application, and of secondary importance, the computation grain and locality of communications. These properties apply to *all* parallel program; therefore our results are positive in the sense that they indicate that synchronization overhead (excluding state saving; see [2]) does not overly burden the computation. It is in this sense that we report that Time Warp performance appears to be relatively *robust*.

Ideally, one would like to have a general purpose simulation vehicle that is adept at extracting whatever parallelism is available in the application program, and produce a corresponding speedup subject only to the amount of computing resources that are employed to solve the problem. Proponents of optimism have argued that among existing approaches, Time Warp offers the best potential for providing such a vehicle. Our experiments to date using synthetic workloads and queueing network simulations tend to support this claim. Additional work using other synthetic and actual workloads is planned to further evaluate this claim.

Acknowledgements

This work was supported in part by NSF grant number CCR-8902362 and a fellowship from the Jet Propulsion Laboratory. The splay tree code that was used for these experiments was originally developed by Douglas W. Jones (with assistance from Srinivas R. Sataluri), and translated to C by David Brower. JPL provided tools for measuring average parallelism.

References

[1] R. M. Fujimoto. Performance Measurements of Distributed Simulation Strategies. Technical Report UU-CS-TR-87-026a, Dept. of Computer Science, University of Utah, Salt Lake City, November 1987 (to appear in Transactions of the SCS).

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

[3] P. Hontalas, D. Jefferson, and M. Presley. Time Warp Operating System Version 2.0 User's Manual. Technical Report JPL D-6493, Jet Propulsion Laboratory, June 1989.

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

[5] D. W. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

[6] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[7] F. Wieland, L. Hawley, A. Feinberg, M. DiLorento, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson. Distributed Combat Simulation and Time Warp: The Model and its Performance. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2), March 1989.