International Journal in Computer Simulation 2, 251–286 (1992)

SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation

JEFF S. STEINMAN
Jet Propulsion Laboratory
California Institute of Technology

Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) is a unified parallel simulation environment. It supports multiple-synchronization protocols without requiring users to recompile their code. When a SPEEDES simulation runs on one node, all the extra parallel overhead is removed automatically at run time. When the same executable runs in parallel, the user preselects the synchronization algorithm from a list of options. SPEEDES currently runs on UNIX networks and on the California Institute of Technology/Jet Propulsion Laboratory Mark III Hypercube. SPEEDES also supports interactive simulations.

Featured in the SPEEDES environment is a new parallel synchronization approach called *Breathing Time Buckets*. This algorithm uses some of the conservative techniques found in Time Bucket synchronization, along with the optimism that characterizes the Time Warp approach. A mathematical model derived from first principles predicts the performance of Breathing Time Buckets.

Along with the Breathing Time Buckets algorithm, this paper discusses the rules for processing events in SPEEDES, describes the implementation of various other synchronization protocols supported by SPEEDES, describes some new ones for the future, discusses interactive simulations, and then gives some performance results.

SPEEDES, parallel discrete-event simulation, interactive simulation, Delta Exchange, Breathing Time Buckets, Time Warp, CO-OP, Proximity Detection event horizon

1 INTRODUCTION

Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) began in May 1990 as a C++ simulation environment that featured a new algorithm (*Breathing Time Buckets*) for synchronizing parallel discrete-event simulations. As the environment matured, it became desirable to make direct

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the U.S. Air Force Electronics-Systems Division, Hanscom Air Force Base, MD, through an agreement with the National Aeronautics and Space Administration.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the U.S. government or the Jet Propulsion Laboratory, California Institute of Technology.

Correspondence and requests for reprints should be sent to Jeff S. Steinman, Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Mail Stop 138-310, Pasadena, CA 91109.

■ Manuscript received January 5, 1991; Manuscript revised December 3, 1991.

comparisons with other synchronization strategies. It was decided that the easiest way to make those comparisons would be to support the other algorithms within the SPEEDES environment. This would also allow more accurate and unbiased comparisons, since the different synchronization techniques would use the exact same application code.

Building this capability was not too difficult, since most of the utilities required by the various other algorithms were already supported in an object-oriented manner within SPEEDES. For example, it took only three weeks to include Time Warp (using aggressive cancellation) as one of the algorithms. By January 1991, the SPEEDES environment could support multiple-synchronization algorithms and interactive simulations.

2 SPEEDES INTERNAL STRUCTURE

Although other multiple-synchronization systems (or test beds) have been developed [1], one reason for the success of SPEEDES is its unique object-oriented design. To begin this discussion, we first break event processing into some very basic steps (see Figure 1).

2.1 Creating an Event

An event is created by a message. Note that multiple messages for an object with the same time stamp will generate multiple events, not a single event with multiple

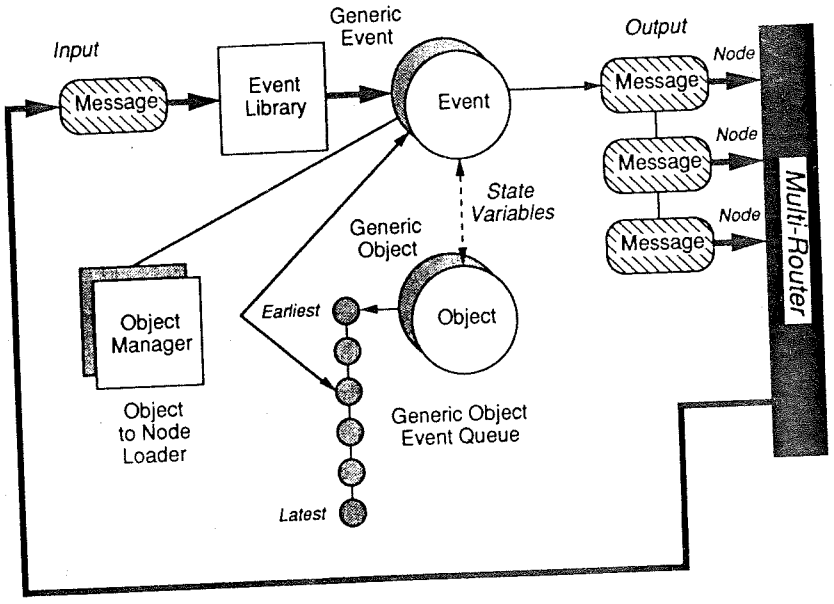


Figure 1. SPEEDES event.

messages. Events are separate objects in C++ and should not be confused with simulation objects. User-defined events inherit capabilities from a base-class generic event object, which defines various virtual functions. It is through these virtual functions that events are processed.

An important optimization is in the use of free lists for memory management. SPEEDES manages old messages and events in a free list and reuses them whenever possible. This speeds up memory management and avoids the memory fragmentation problem.

2.2 Initializing an Event

Events are initialized by data contained within the message through a user-supplied virtual initialization function. After the event is initialized, the message is discarded into a free list. Each event is then attached to its own simulation object (i.e., the event object receives a pointer back to the simulation object).

2.3 Processing an Event: Phase 1

Processing an event is done in multiple steps that are all supported with C++ virtual functions written by the user. In the first step, an event optimistically performs its calculations and generates messages to schedule future events. However, *the simulation object's state must not change*. In addition, messages that would generate future events are not immediately released.

The event object itself stores changes to the simulation object's state and the generated messages. Only variables affected by the event are stored within the event object. Thus, if a simulation object contains 50,000 bytes and an event requires changing one of those bytes, only that 1 byte is stored within the event. There is no need to save copies of all 50,000 bytes of the object in case of rollback.

2.4 Delta Exchange

In the second step, the values computed in Phase 1 are exchanged with the simulation object. This exchange is performed immediately after the first step. After an exchange, the event has the old state values and the simulation object has the new values. Two successive exchanges (in the case of rollback) then restore the simulation object's state.

When an event is rolled back, there are two possibilities concerning messages that were generated by the Phase 1 processing. One is that the messages have already been released. In this case, antimessages must be sent to cancel those erroneous messages. The other is that the messages have not been released yet. In this case, the messages are simply discarded.

The Delta Exchange mechanism greatly reduces memory consumption in optimistic simulations. However, it has the drawback of forcing the user to supply

the exchange code. Errors could creep into the simulation if care is not taken in this step.

Performing the Delta Exchange method normally involves a negligible amount of time. Thus, sequential simulations are still efficient even when this extra step is performed. Furthermore, because the Delta Exchange mechanism normally has low overhead, special-purpose hardware to support rollback efficiently may not be necessary [2].

The Delta Exchange mechanism has the added benefit of permitting fast rewind capabilities. Similar to an efficient text editor that saves only the keystrokes (i.e., changes to the text file), the Delta Exchange mechanism saves the changes to the simulation objects. These changes (stored in events) can be written out to files. The simulation can be rewound by restoring the changes in reverse order. This is like hitting the *undo* button in a text editor. The rewind capability can be used for restarting the simulation after crashes, check-point restarting, *what if* analysis, or playback.

2.5 Processing an Event: Phase 2

In the third step, further processing is done for an event. This usually involves cleaning up memory or sending external messages to graphics. This step is performed only after the event is known to be valid, in other words, when there is no possibility for the event to be rolled back. This step is usually performed much later in time than the previous two steps. The simulation programmer should not assume that the simulation object contains valid state information when processing in Phase 2. The processing done in this step must not change the state variables of its simulation object.

2.6 Managing the Event List

One of the most time-consuming tasks in supporting discrete-event simulations can be managing the event list. Managing a sorted list of future events can cripple the performance of a low-granularity simulation. In parallel discrete-event simulations, such management often leads to superlinear speedup. SPEEDES makes use of a new technique for handling the event list.

The basic idea of this new technique is that two lists are maintained continually. The primary list is sorted, whereas the secondary list is unsorted. As new events are scheduled, they are put into the secondary list. The earliest event scheduled to occur in the secondary list is preserved. When the time to process this event comes, the secondary list is sorted and then merged into the primary list. The time stamp of this critical event is sometimes called the *event horizon* (see the discussion in Section 4).

This simple approach for managing the event list is faster than single-event insertions into linked lists. It can also outperform some of the more complicated data structures such as splay trees and priority heaps [3, 4], if enough events collected in the secondary queue on the average for each cycle (see Figure 2). The

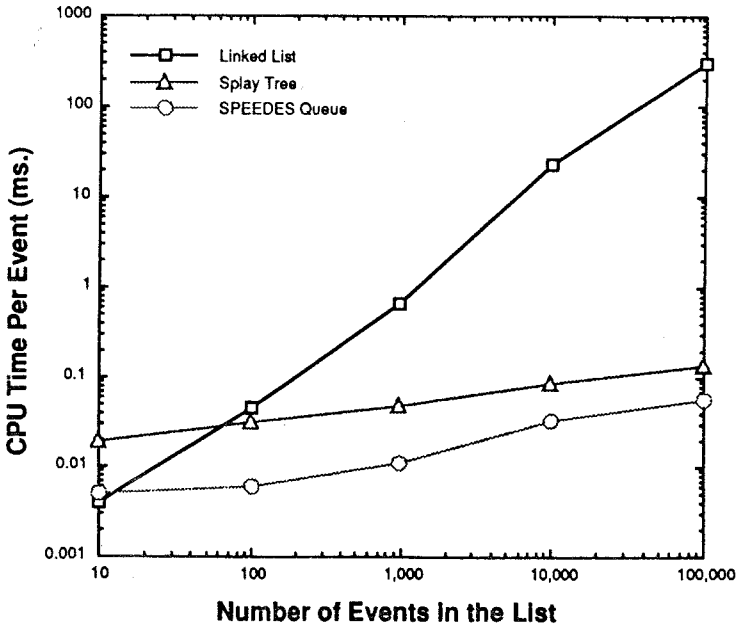


Figure 2. Timings for three event-list data structures. These measurements were made on a Silicon Graphics IRIS4D. The hold model was used with each event generating a single new event pushed into the future by the beta density distribution $\beta(t) = Ct^{n1}(1 - t)^{n2}$. For this measurement, $n1 = n2 = 20$. This beta density distribution looks like a bell-shaped curve bounded by 0 and 1, with a mean of 0.5 and a sigma of 0.0762.

average number of events collected per cycle can be calculated analytically. This is described later in the paper.

2.7 Event Queue Objects and Multiple Protocols

In a SPEEDES simulation, the user does not supply the main program. The main program is provided by SPEEDES, which, during initialization, reads in a standard file to configure the simulation. The user can select the synchronization protocol by modifying this file.

SPEEDES supports multiple-synchronization protocols by creating an appropriate event queue object. Each protocol has its own specific event queue C++ object, which is created during initialization. Each event queue object is then responsible for performing its specific synchronization algorithm for the simulation. Event queue objects must follow the rules for event processing (Phase 1, Delta Exchange, Phase 2).

In the creation of C++ objects that make use of inheritance, the lower base-class objects are constructed before the higher ones. Thus, when the main program creates one of the event queues, the generic base-class event queue object is constructed first. The constructor of this base-class automatically calls the user code

that creates all the simulation objects and initializes them with their starting events. This is how the user plugs his code into the SPEEDES environment.

After initialization, the main program in SPEEDES loops until the simulation is done. During each loop, four virtual functions (see Figure 3) are called for the event queue object:

1. Process Phase 1
2. Simulation Time
3. Process Phase 2
4. External Blocking

Phase 1 and Delta Exchange event processing is performed for events during the event queue Process Phase 1 method. The global simulation time [e.g., global virtual time (GVT) in Time Warp] is then determined in the Simulation Time method. Cleanup, synchronous message sending, and further event processing are done in the Process Phase 2 method. If the simulation expects the outside world to send a message that must arrive before the simulation can continue, blocking is done in the External Blocking method.

2.8 Message Sending

SPEEDES uses both synchronous and asynchronous message-sending approaches. Time Warp uses the asynchronous style, whereas the other algorithms synchro-

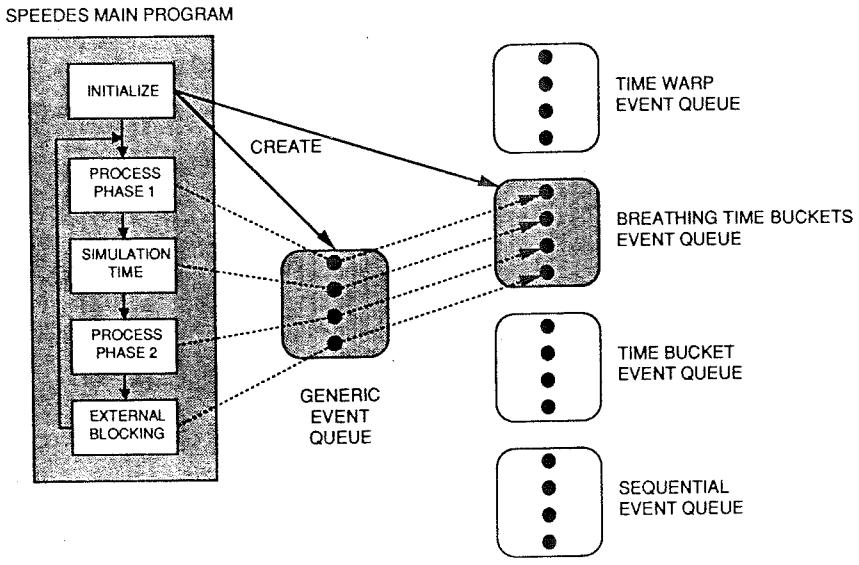


Figure 3. SPEEDES internal processing structure. The simulation objects are created and initialized during the construction of the base-class event queue object. The synchronization strategy is accomplished through the creation of the appropriate event queue object (which, in this example, is the Breathing Time Bucket event queue). Virtual functions allow the main program to call generic methods that are supported by each of the specific event queue objects.

nously send their messages. For example, the Crystal Router [5] is used when running on hypercubes. Synchronous message sending using the Crystal Router has been observed to be about five times faster than its asynchronous counterpart, because collisions are avoided and buffers are preallocated.

There are two extremes for event processing and message sending. In one extreme, events take very little CPU time to be processed; message sending is the bottleneck. Here, synchronous message sending wins because it is faster. In the other extreme, events take a very long time to be processed; event processing is the bottleneck. In this case, message-sending delays do not affect the simulation's performance and it does not matter whether synchronous or asynchronous approaches are used. However, somewhere between these two extremes is a boundary where one approach may be better than the other.

3 SPEEDES SIMULATION PROTOCOLS

This section discusses briefly the well-known parallel simulation protocols supported by SPEEDES, whereas the next section explains the new parallel simulation approach, *Breathing Time Buckets*, in more detail. Following the discussion of Breathing Time Buckets, we describe some new protocols that look promising for efficient parallel simulation.

3.1 Sequential Simulation

When SPEEDES runs on one node, the sequential event queue object is created automatically. All the overhead for message sending and rollback is removed. The user still generates messages for his or her events, but they are not queued up for transmission. Instead, they are turned into events directly. The Delta Exchange mechanism is also used. The combined overhead for message generation and Delta Exchange has been observed to be less than 1% for low-granularity events (i.e., events in which the system overhead dominates).

3.2 Time Bucket Synchronization

One of the simplest approaches to parallel simulation makes use of the *causality principle*. As long as a minimum time interval, T, between events and the events that they can generate are known, the simulation can proceed in time cycles of duration T. This approach is called *Time Bucket Synchronization* [6]. It has the important property of requiring very little overhead for synchronization. Each node must synchronize with all the other nodes at the end of every cycle, after which all nodes increment their simulation time in unison by the amount T.

Despite the low synchronization overhead, the Time Bucket approach has some drawbacks. The cycle duration T must be large enough for each node to process enough events to make parallel simulation efficient. However, the cycle duration

T must also be small enough to support the required simulation fidelity. Load balancing over the small time interval T can also be a problem.

In most discrete-event simulations, the time step T is unknown or, even worse, has a value of zero. Thus, simulations that can run under Time Bucket synchronization are a subset of all parallel discrete-event simulations.

3.3 Time Warp

The Time Warp algorithm has been heavily discussed in the literature [7–9]. SPEEDES offers a unique set of data structures for managing the event processing in its version of Time Warp.

When an event is processed, it may generate messages. These messages are handed immediately to the TWOSMESS server object supported by SPEEDES. This object assigns a unique identification (ID) to the outgoing messages and stores the corresponding antimessages back in the event. Note that antimessages are not complete copies of the original message, but are very short messages used for bookkeeping. All of this is done transparently for the user.

When a message arrives at its destination, an antimessage is created and stored in the TWOSMESS hash table. The hash table uses the unique message ID generated by the sender. An event is constructed automatically from the message and is handed to the Time Warp event queue object (see Figure 4). This event is put in the secondary queue if its time stamp is in the future of the current simulation. Otherwise, the simulation rolls back.

Rollback restores the state of the simulation object, which means calling the

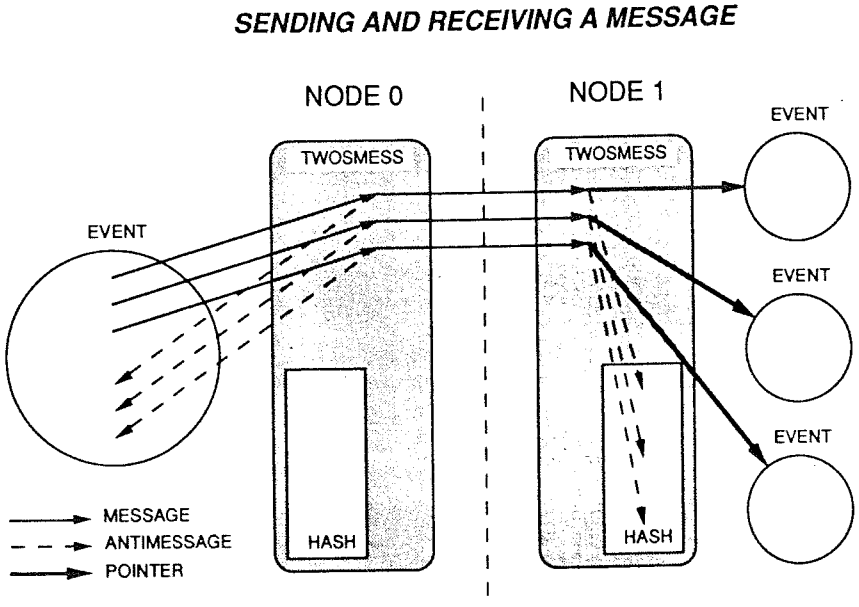


Figure 4. Message sending and receiving in Time Warp.

Delta Exchange method for all the events processed by that object in reverse order and generating antimessages. Aggressive cancellation is used.

Antimessages are stored in the events and are simply handed to the TWOSMESS object. When these antimessages arrive at their destinations, the hash table already contains pointers to the events that they created. Those events are then rolled back (if already processed) and marked as *not valid* (see Figure 5).

Periodically (typically every 3 s of wall-clock time), the global virtual time is updated. The GVT represents the time stamp of the earliest event unprocessed in the simulation. One problem in determining the GVT is in knowing whether messages are still floating about in the system. This problem is solved by having each node keep track of how many messages it has sent and received. Fast synchronous communications are used to determine when the total number of messages sent equals the total number of messages received. When this condition is true, no more messages are in the system and the GVT can be determined.

After the GVT is known, cleanup is performed. The memory for all processed events with time stamps less than or equal to the GVT is handed back to the SPEEDES memory management system (free lists). The hash tables are also cleaned up, because their antimessages are no longer needed.

4 BREATHING TIME BUCKETS

The original SPEEDES algorithm (*Breathing Time Buckets*) is a new windowing parallel simulation strategy with some unique properties. Instead of exploiting

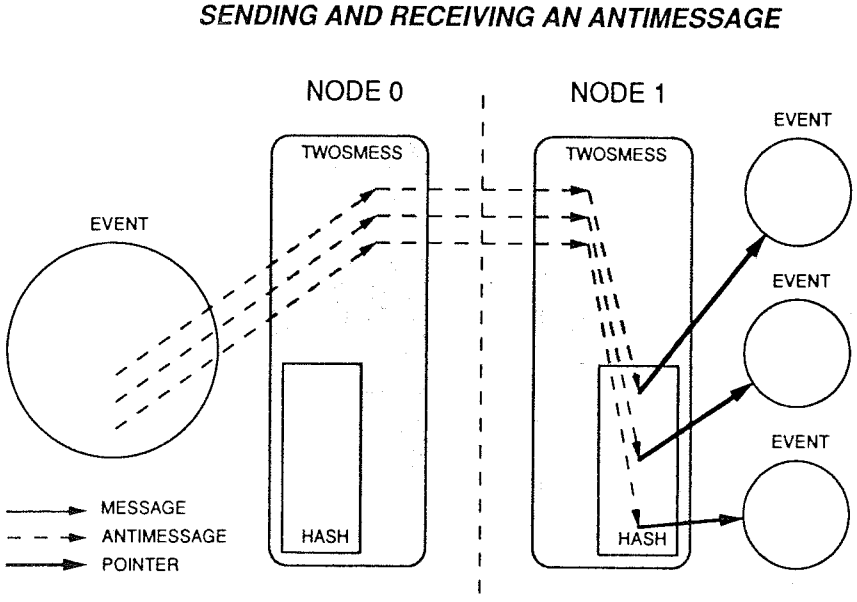


Figure 5. Antimessage sending and receiving in Time Warp.

lookahead on the message receiver's end [10, 11] or using preknown or calculable delays [12], it uses optimistic processing with local rollback. However, unlike other optimistic windowing approaches [13], it never requires antimessages. Local rollback is not a unique concept either [14]. However, the Breathing Time Buckets algorithm allows full connectivity between the simulation objects (often called *logical processes*).

4.1 Fundamental Concepts

The essential synchronization concept for Breathing Time Buckets is the causality principle. Similar to the Time Bucket approach, the Breathing Time Buckets approach processes events in time cycles. However, these time cycles do not use a constant time interval T. They adapt to the optimal width, which is determined by the *event horizon*. Thus, in each cycle, the maximum number of causally independent events (ignoring locality) is processed. This means that no limiting assumptions are made that restrict the simulation as there are in the Time Bucket approach. Deadlock can never occur, since at least one event is always processed in a cycle.

The event horizon is defined as the time stamp of the earliest new event generated in the current cycle (much like the event list management previously described). Processing events beyond this boundary may cause time accidents. Thus, events processed beyond the event horizon may have to be rolled back. The *local event horizon* for a node is defined as the time stamp of the earliest new event generated by an event on that node. The global (or true) event horizon is the minimum of all local event horizons (see Figure 6). The event horizon then defines the next time step T.

To determine the global event horizon, optimistic event processing is used. However, messages are released only after the true event horizon is determined, so antimessages are never required. Rollback simply involves restoring the object's state and discarding messages generated erroneously. Thus, the Breathing Time Buckets algorithm eliminates all the potential instabilities due to excessive rollback that are sometimes observed in Time Warp. This will be demonstrated later in this paper.

4.2 Determining the Event Horizon

Determining the event horizon on a single processor is not very difficult. It is much more challenging to find in parallel. For now, assume that each node is allowed to process its events until its local event horizon is crossed. At this point, all nodes have processed events up to their local event horizon and have stopped at a synchronization point.

The next step is for each node to communicate its value synchronously for the local event horizon. The minimum of all these is defined to be the global event horizon. In other words, the earliest time stamp of a message waiting to be released

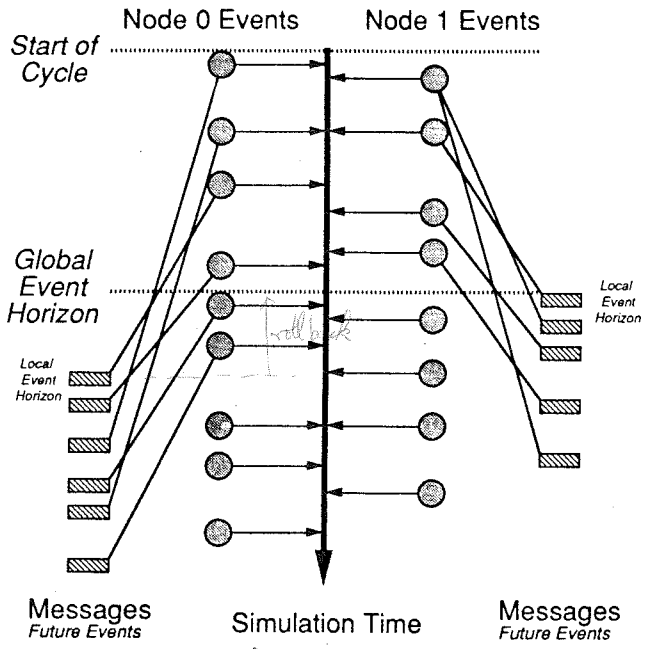


Figure 6. Distributed event queues. Each node processes events in two stages: The first stage allows events to change an object's state variables and allows messages corresponding to future events to be generated. If the time stamp of an event is greater than the earliest time stamp of a locally generated message, a local causality violation may occur and processing is stopped at the local event horizon. A global minimum time value of all generated messages is determined by the global simulation time (GST) via a synchronous communication to determine the true event horizon, and the second stage of event processing begins. The second stage of processing sends all messages for events with time stamps less than or equal to the GST to their proper destinations. Rollback of processed events beyond the GST boundary may take place if necessary.

is identified. The global event horizon is then used to define the global simulation time (GST) of the system.

After the GST is defined, all events with time stamps less than or equal to this time are made permanent. This means that messages generated by events that had time stamps less than or equal to the GST are routed through the hardware communication channels to the appropriate node containing the destination object. When messages arrive at their destination nodes, they are fed into the *event library*, which converts messages into events.

These new events are not immediately inserted into the event queue. Rather, they are collected in a temporary queue as described previously. When all the new events are finally created, the temporary queue is sorted, using a merge sort algorithm that has $m \log(m)$ as a worst-case sort time (for m events). After the temporary queue of new events is sorted, it is merged back into the local event queue.

There is an obvious problem with what has been described so far. Some of the nodes may have processed events that went beyond the GST (i.e., the true event

horizon). An event, which is attached to a locally simulated object, must be rolled back if any of the newly generated events affect that same object in its past. Rollback involves discarding the messages generated by the event (which have not yet been released because the time stamp of the event is greater than the GST) and exchanging state variables back with the simulated object. Thus, rollback overhead should remain small. Antimessages are never needed because bad messages (which would turn into bad events) are never released.

4.3 Asynchronous Broadcasts

If the Breathing Time Buckets algorithmn ended here, it would have a limited number of applications. Pathological situations could arise if the algorithm was not modified. For example, Figure 7 shows how an unbalanced work load could affect performance. The problem with Breathing Time Buckets as presented so far is that all nodes wait for the slowest node to finish. A modification to the basic algorithm is needed to circumvent this problem.

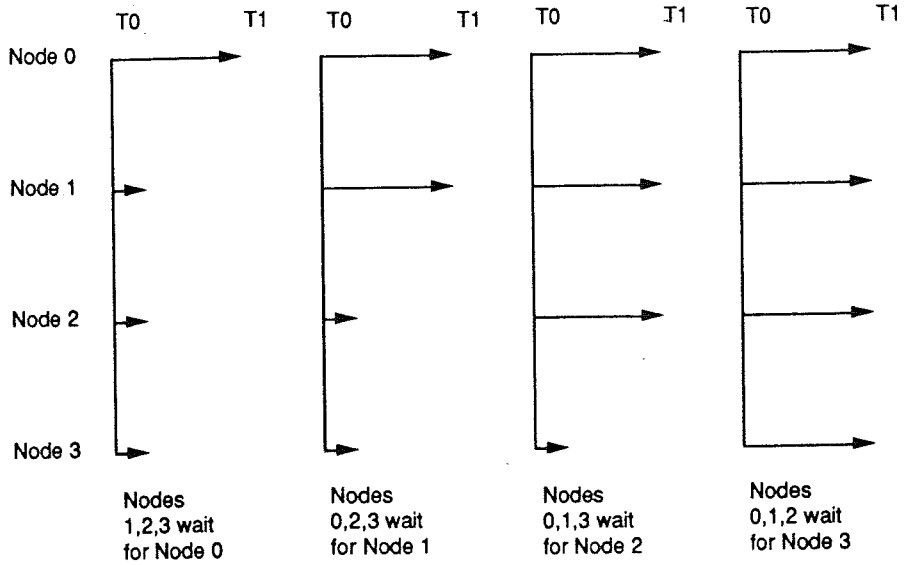


Figure 7. The basic mechanism breaks down for this special case. Assume that there are four nodes processing events starting at GST = T_0. Also assume that the CPU time required to process a cycle is proportional to the cycle interval. If each node locally processes its events until its event horizon is crossed, serialization of the simulation can occur. Imagine that, in the first cycle, node 0 locally processes events all the way to T_1. The other nodes finish their cycles early (assume they stay at T_0) and wait for node 0 to finish. In the next cycle, node 1 was able to make it all the way to T_1. Node 0 already calculated its events in the previous cycle (unless rollback was required). Nodes 2 and 3 still finish early and wait for node 1 to complete. This continues until finally node 3 is able to process events to T_1. The total time required for this series of cycles is four times the time for one node to process events from T_0 to T_1. Thus, no speedup is achieved.

A simple mechanism to solve this problem incorporates an asynchronous broadcast mechanism that tells all the nodes when a local event horizon is crossed (see Figure 8). When one node crosses its local boundary, it broadcasts this simulation time to all the other nodes. When a node receives one of these broadcast messages, it may determine that it has gone beyond the point of the other node's boundary; thus, it should stop processing. Conversely, the node may not have reached that time yet, so processing should continue. It is very likely that the first node to cross its local event horizon (in wall-clock time) has a greater value for this boundary than another node. If this happens, a second node will broadcast its time as well. Multiple broadcasts may occur within each cycle.

It is important to get a proper view of the broadcast mechanism. Runaway nodes that process beyond the true event horizon while the rest of the nodes are waiting can ruin the performance of the Breathing Time Buckets algorithm unless something is done. The proper view of the broadcast mechanism is that it aids in speeding up the processing by stopping runaway nodes. The asynchronous broadcasts are in no way required by Breathing Time Buckets to rigorously synchronize event processing. The broadcasts function in the background and only aid in enhancing performance.

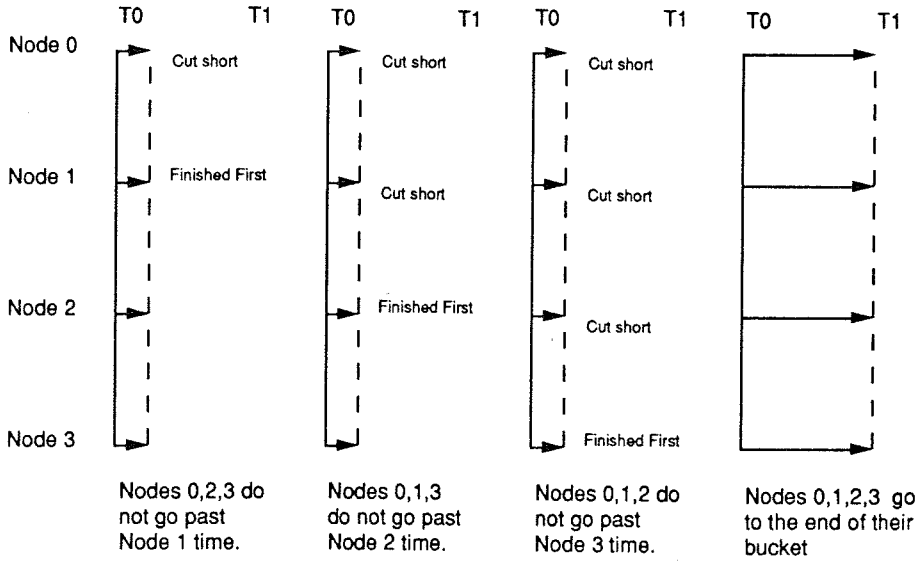


Figure 8. The asynchronous broadcast mechanism stops runaway nodes. The problem encountered in Figure 7 is solved when nodes that finish early asynchronously broadcast their event horizon to all other nodes. The total amount of CPU time spent in the first three cycles is very small compared with that in the last cycle, where all four nodes are allowed to process events in parallel up to time T_1. Thus, overall parallel speedup is achieved. Note that even if the first three cycles are inefficient (and possibly serialize), if a sufficient amount of work is done in the last cycle, then overall speedup may still approach the theoretical upper bound (i.e., the number of nodes).

4.4 Nonblocking Sync

With the asynchronous broadcast mechanism designed to stop runaway nodes, the Breathing Time Buckets algorithm becomes a viable solution to support general-purpose discrete-event simulations. However, there still is room for improvement. It is wasteful for nodes that have crossed their local event horizon to sit idle waiting for other nodes to complete their processing. Note that this problem always arises in the world of synchronous parallel computing. It is important to balance the work load evenly on each node so that the time spent waiting for the slowest node to finish its job is minimized.

The Breathing Time Buckets algorithm, as described so far, suffers from this same "waiting" problem. An observant simulation expert might ask, "Why do you insist on stopping just because the event horizon has been crossed?" In fact, there really is no reason to stop processing events until *all the* nodes have crossed the horizon! Erroneously processed events can always be rolled back without much overhead (because no communications are involved). Therefore, it is not detrimental to continue processing events beyond the horizon. It might be beneficial to be optimistic and hope that the processed events with time stamps greater than the event horizon do not have to be rolled back. The trick then is to efficiently find out when all the nodes have finished.

One way to support this needed mechanism would be for each node to send a special message to a central manager when it thinks it has crossed the event horizon. When the central manager receives this message from all nodes, it broadcasts a message back to the nodes saying that it is time to stop processing events for this cycle. This approach is used when running Breathing Time Buckets on a network of Sun workstations over Ethernet. This mechanism has the good characteristic of being portable. However, it is not scalable to large machines.

Other ways to solve this problem exist, using scalable asynchronous control messages, shared memory, or reduction networks [15], but a better solution would be to use a global hardware line. The idea here is that when each node crosses the event horizon, it sends a signal on a hardware global line. When all the nodes have done this, and interrupt is fired simultaneously on each node and a flag is set telling us that all nodes have crossed the event horizon. This solution has been tested on the California Institute of Technology/Jet Propulsion Laboratory (Caltech/JPL) Mark III Hypercube and works extremely well.

Although the Breathing Time Buckets algorithm does not require global hardware lines for synchronization, making use of the global line has been observed to enhance the performance by as much as 15% over the asynchronous control message approach.

4.5 Local Rollback

One further improvement can be made to the Breathing Time Buckets algorithm. Events generated locally (i.e., messages that do not leave the node) do not have to participate in the event horizon calculation. Rather, they can be inserted into

the event list and possibly be processed within the same cycle. This capability is very important for simulations in which events schedule future events for the same object. A good example of this would be a preemptive priority queuing network. Supporting this capability involves more overhead, but it may be essential for a large class of simulation applications.

4.6 Analytic Model for the Event Horizon

When designing software, programmers frequently find it difficult to predict performance. In particular, it is usually very difficult to predict the performance of parallel discrete-event simulations. Time Warp is a good example of this since it is difficult to predict all sequences of events that might lead to rollback [16]. Because performance is so difficult to predict, most applications tend to shy away from the more aggressive simulation strategies. Ultraconservative algorithms (which may be less efficient, but are more understandable), such as time-driven approaches, tend to dominate real-world applications. In this section, formulas will be derived that can be used to predict the performance of the Breathing Time Buckets algorithm.

For this analysis, assume that each event generates a single new event as it is processed. The time tag of the generated event is described by a random density function $f(t)$. Using this model, the total number of events, n, in the simulation is constant. Our goal is to determine the average number of events, m, processed in a cycle. It should be noted that while others have performed similar calculations [11], we derive our results for general systems using equilibrium conditions and differential equations.

After determining m for a stochastic simulation, we should note that if m is much larger than the number of nodes, N, the Breathing Time Buckets algorithm should be able to achieve high parallel performance. Some definitions must be made at this point:

n = total number of events at start of cycle
m = average number of events processed in a cycle
t = simulation time. $t = 0$ is the start of cycle
$f(t)$ = random distribution for event generation
$F(t)$ = cumulative probability distribution of $f(t)$
$G(t)$ = $1 - F(t)$
$P(t)$ = probability of next cycle boundary exceeding t
$\rho(t)$ = density of pending events. $\rho_0 = \rho(0)$
$\delta t(t)$ = average time interval between events, $= 1/\rho(t)$

We start by assuming that the total number of events n and the random distribution function $f(t)$ are known. Furthermore, we assume that the first scheduled event occurs at time $t = 0$. Our first step is to obtain an expression for the equilibrium event density $\rho(t)$ for the simulation. Figure 9 shows conceptually how, under equilibrium conditions, the shape of the event density $\rho(t)$ remains unchanged as events are processed.

Assume that the first event is processed at time 0. The second event is scheduled

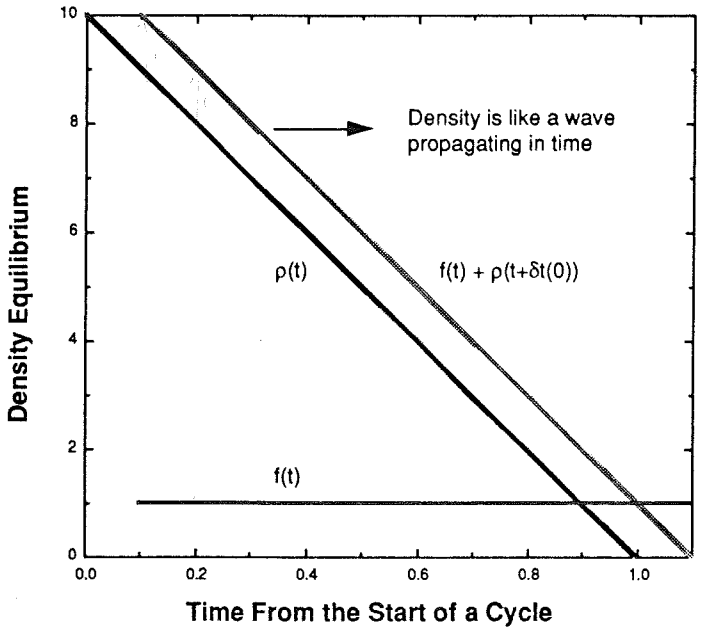


Figure 9. Equilibrium for the density of events. Assume that $f(t) = 1$ for $0 < t < 1$ and that $\rho_0 = 10$. After the first event is processed, the next event occurs at time $\delta t = 1/\rho_0$. A new event is also generated with a time stamp distributed according to $f(t)$. The shape of the new density function is the old density function shifted by δt added to $f(t)$. In equilibrium, the shape of the density function is unchanged. Thus, one can view the density function as a wave propagating in time.

to occur at time $\delta t(0)$. Adding the new event (generated by the first event) back into the event density distribution should restore the density function back to its original shape [except shifted in time by $\delta t(0)$]. In other words, the shape of the event density distribution relative to the starting time of the earliest event in the simulation should be time-invariant. The event density can then be viewed as a wave propagating in time. This leads to the following equilibrium equation:

$$\rho(t) = \rho(t + \delta t(0)) + f(t),$$

where $\delta t(0) = 1/\rho_0$.

Using $\rho(t + \delta t(0)) \approx \rho(t) + \delta t(0)\rho'(t) = \rho(t) + \rho'(t)/\rho_0$, a simple differential equation can be constructed:

$$0 = \rho'(t)/\rho_0 + f(t).$$

The desired solution to this equation is $\rho(t) = \rho_0 G(t)$. Note that the total number of events can be determined by

$$n = \int_0^\infty \rho(t)dt.$$

This gives a way to solve for ρ_0 if n is known:

$$\rho_0 = n \Big/ \int_0^\infty G(t)dt.$$

Now that the event density has been found, the next step is to determine the probability $P(t)$ that an event with time stamp t is processed in the current cycle. We start by considering the first event. It is always safe to process the first event. Thus, $P(t_0) = 1$. The next event occurs at time $t_1 = t_0 + \delta t(t_0)$. The probability of this event being in the current cycle is just the probability that the first event scheduled its new event with a time tag greater than t_1. Since $F(t)$ is the probability of the event being generated with a time tag less than or equal to t (relative to the generating events time tag), $G(t) = 1 - F(t)$ is the probability that the event has a time tag greater than t. This means that $P(t_1) = G(t_1 - t_0)$. In general, an event at time t will be in the current cycle if all the previous events in the cycle generate their events with time tags greater than or equal to t. This probability can be constructed in a general way.

If we define $t_{n+1} = t_n + \delta t(t_n)$ as the time tag of the $(n + 1)$th event from the start of the cycle (note that the zeroth event is the first event of the cycle), then the probability of this event being in this cycle is the product of all the probabilities of the previous events generating their events with time tags greater than t_{n+1}. This is described mathematically as

$$P(t_{n+1}) = G(t_{n+1} - t_0)G(t_{n+1} - t_1)G(t_{n+1} + t_2)\cdots G(t_{n+1} - t_n).$$

A very good approximation to this expression is simply to relate the probability of the next event being in the current cycle to the previous event. The error introduced by making this approximation is at most second order in $F(t)$. Because $F(t)$ must be small for the important terms ($0 << P(t) < 1$), this error is negligible. This is expressed as the following general equation:

$$P(t + \delta t)) = P(t)G(t + \delta t(t)).$$

Figure 10 shows how this can be visualized by successively processing events and using the function $G(t + \delta t(t))$ to modify $P(t)$.

Using the density calculated earlier, it is possible to process one event at a time, moving $\delta t(t)$ steps of size $1/\rho(t)$. At each step, an additional factor of $G(t + \delta t(t))$ is multiplied into the probability function. This may be expanded into the following differential equation:

$$P(t) + P'(t)/\rho(t) = P(t)G(t) + P(t)G'(t)/\rho(t).$$

Now, using the fact that $G(t) = 1 - F(t)$ and that $G'(t) = -f(t)$, this equation may be simplified to the form

$$P'(t) = -P(t)[\rho(t)F(t) + f(t)].$$

The solution to this equation is then

$$P(t) = \exp\left(-F(t) - \int_0^t \rho(\tau)F(\tau)d\tau\right).$$

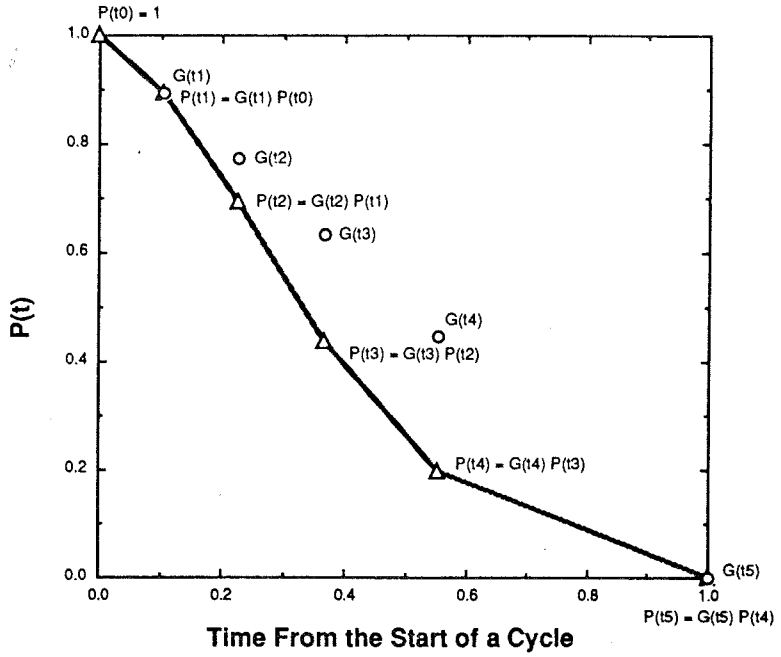


Figure 10. Probability that an event occurring at time t will be included in the current cycle. Assume that $f(t) = 1$ for $0 < t < 1$ and that $\rho_0 = 10$ ($n = 5$). The probability of each successive event being processed in the current cycle is equal to the probability of the previous event being processed in the current cycle times $G(t)$. The circle data points represent the factor of G in the product that determines $P(t)$.

Finally, the average number of events processed in a cycle can be integrated using the density function $\rho(t) = \rho_0 G(t)$:

$$m = \int_0^\infty \rho(t)P(t)dt.$$

One very important class of event generation distributions should now be noted. If there is a minimum time delay T between an event and its generated event (i.e., a simulation that might lend itself to the standard Time Bucket approach), then $P(t)$ and $G(t)$ are equal to 1 for $0 < t < T$. This is a very useful piece of information because it allows us to make a conservative estimate quickly of the number of events m processed in a cycle:

$$m > \rho_0 T.$$

Another very important class of simulations uses exponential event generation distributions $f(t) = e^{-t}$. For this distribution, m can be calculated analytically. Without going through the steps, we get

$$m = \sqrt{2\pi n}e^{1/2n}[\text{erf}(\sqrt{n} + 1/\sqrt{n}) - \text{erf}(1/\sqrt{n})],$$

where $\text{erf}(x)$ is the area under the standard normal curve (mean = 0, standard

deviation = 1) from 0 to x. For large n(i.e., $n > 10$), this expression simply reduces to

$$m = \sqrt{\pi(n + 1)/2} - 1 - (1/2n).$$

In practice, it is usually easier to compute m numerically. Figure 11 shows the average number of events m in each cycle for different values of n using the flat distribution $f(t) = 1$ for $0 < t < 1$. The average number of events m processed in a cycle was measured directly for the case where $n = 10{,}000$. The measured value for m was 176.5 ± 0.3, while the theoretical prediction was 175.9 (in excellent agreement). Note that even with the possibility of zero time delay between an event and its generated event, the Breathing Time Buckets algorithm still processes a large number of events in each cycle when the total number of events n is large.

5 FUTURE SPEEDES PROTOCOLS

SPEEDES is not limited to the protocols described earlier. Other protocols could easily be supported within SPEEDES [7, 9]. Some of these are described below.

5.1 Time-Driven Approach

One very interesting simulation application is neural networks. Events in a neural network simulation tend to use very little CPU time. Furthermore, neural net-

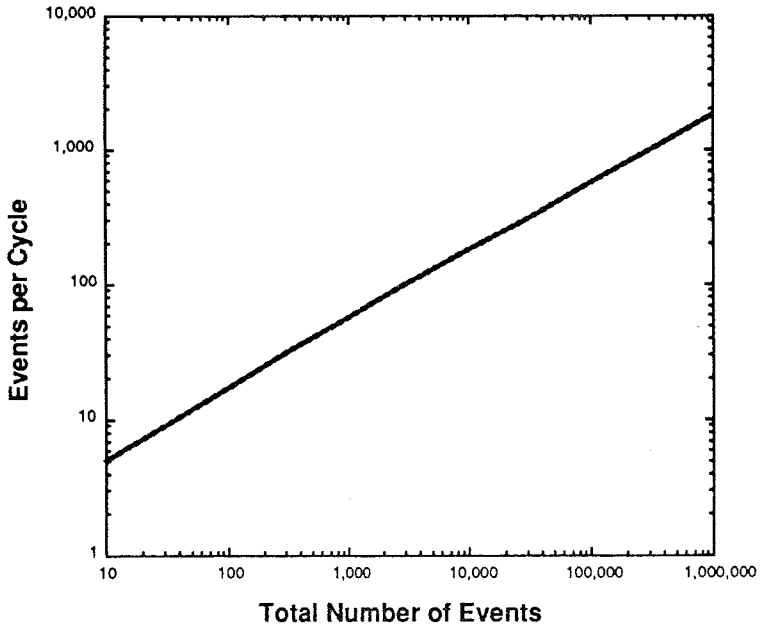


Figure 11. The average number of events processed in a Breathing Time Buckets cycle using $f(t) = 1$ for $0 < t < 1$. Notice that as n gets larger, Breathing Time Buckets can become more efficient because there are more events in a cycle.

works tend to have huge numbers of neurons with high connectivity. A change in the state of a neuron may cause large numbers of messages to be sent.

Neural network simulations may not perform very well under Time Warp because of the low-granularity events and the large fan-out of messages. Breathing Time Buckets should perform better than Time Warp because neural network simulations tend to be large and because this algorithm does not suffer from the rollback of the fan-out type of messages. However, a simple time-driven mechanism should be faster than either of these approaches.

Neural network simulations are usually written with events occurring at integer times. Thus, time is not a continuum, but is, rather, discrete. Furthermore, events are typically scheduled one or two units of time into the future, not thousands of time units into the future. This has major implications for event list management: Sorting is not needed. Rather, a mechanism that clumps events with the same time into individual lists should be used. A small number of lists managed in a circular array would support such a simulation.

A discrete-time event management system should be easy to implement in the SPEEDES framework. It would be interesting to then compare the performance of the various event synchronization algorithms with a neural network simulation.

5.2 CO-OP (Conservative-Optimistic) Approach

Parallel simulations normally schedule their events by strict time order within their nodes [17]. In other words, the event with the earliest time stamp on a node is always processed next. This is not required for provably correct simulations. The only requirement is that the events for each object should be processed in their correct time order. This is where the CO-OP algorithm can improve the performance of a parallel simulation.

It is possible to imagine a situation in which the object with the earliest event to be processed on a given node knows that it should wait for another message to arrive before it can correctly process its event. For example, the object may be waiting for a reply from a query. Another object on the same node may have its first event occurring at a later simulation time, but knows its event is valid.

If we could process events using their knowledge of probability of correctness (see Figure 12), we could eliminate many rollbacks. It may even be possible to prove that this approach is optimal in an analytic sense by correctly balancing conservative and optimistic processing.

5.3 Local CO-OP Approach

One very good example of how an application of the CO-OP system could be used is in simulations of networks with sparse connectivity. This information (sometimes called *locality*) can be used to help synchronize the simulation. Computer hardware simulations are good examples of this type of network. Conservative synchronization methods have been used extensively for these types of simulations, with mixed results [18, 19]. Here we present an alternative optimistic approach.

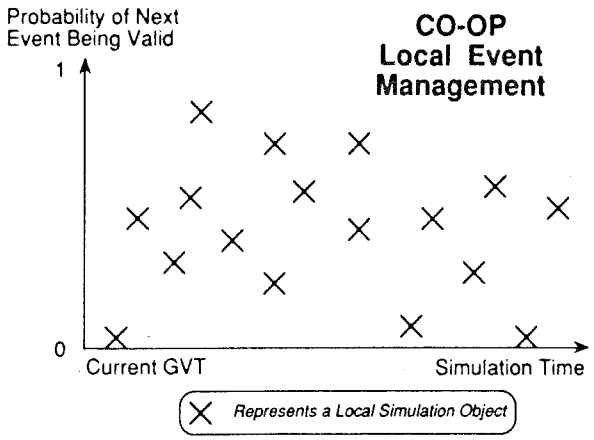


Figure 12. CO-OP event handling. All the objects residing on a given node are shown here. The simulation time represents the time stamp of the next event for that object. Each of these objects also estimates its probability of being correct. Note that the earliest event on the node does not have to have the highest probability of being correct. The CO-OP approach processes events in terms of their probability of being correct, not by strict time order.

Each object in such a simulation has a set of input message queues. As long as messages are waiting to be processed in each of an object's input queues, it is safe to process the earliest message. When any of the input queues are empty, it may or may not be correct to process the earliest message from the other queues because an earlier message might arrive in the empty one.

We could define a scheduling variable T_{sort} for each object; this variable does not necessarily reflect the simulation time of the object. Objects will then remain sorted according to T_{sort}. As events are processed and input messages arrive for an object, the object's value for T_{sort} will change.

Objects with an empty message queue could set T_{sort} to be the time stamp of the earliest message waiting to be processed. Objects with full message queues could set their T_{sort} value to be the previous GVT, so their next event would be processed before other objects with empty message queues (even if their event would have occurred later in simulation time than unprocessed events for other objects). Furthermore, objects with empty message queues are naturally prioritized based on their relative time from the GVT (the standard way of scheduling events in Time Warp). In addition, the classical techniques for exploiting lookahead [10, 19] can be included in this approach.

6 INTERACTIVE SPEEDES

This section discusses the difficulties of supporting interactive simulations. We then describe how SPEEDES solves these problems.

6.1 Simulation Output

In an interactive parallel simulation involving humans, information pertaining to events that have been processed is released to the outside world. Humans can view these data in various forms (graphics, printouts, etc.) and are then allowed to interact with the simulation based on information that was released previously.

When a simulation runs on a single computer, using a sorted event queue, events are processed in their correct time order. If the results of processed events were released to the outside world, they would naturally be viewed in their correct time order. This is not true for parallel simulations.

In parallel simulations that operate in cycles, each node has its own local event queue. Assume that m events are processed globally for a particular cycle and that there are N nodes. Then each node has m/N locally processed events (assuming perfect balance). While these processed events are maintained in their proper time order locally, additional steps are required to merge them into a single globally sorted list. The steps to do this on a parallel computer are given in Figure 13.

The time cycle boundaries t_i and $t_i + 1$ are known. Assume a flat distribution for the time stamps of the processed events. Each node breaks up its processed event queue into N sublists, each of length m/N^2. Every sublist passes to a different node k, where $k = 0, 1, 2, \ldots, N - 1$. The lower time boundary of each sublist residing on node k is $t_i + k(t_i + 1 - t_i)/N$. All events in each of the sublists on node 0 have time stamps less than those on node 1, etc. At this point, each node performs a local merge sort of its N sorted sublists using a binary search tree.

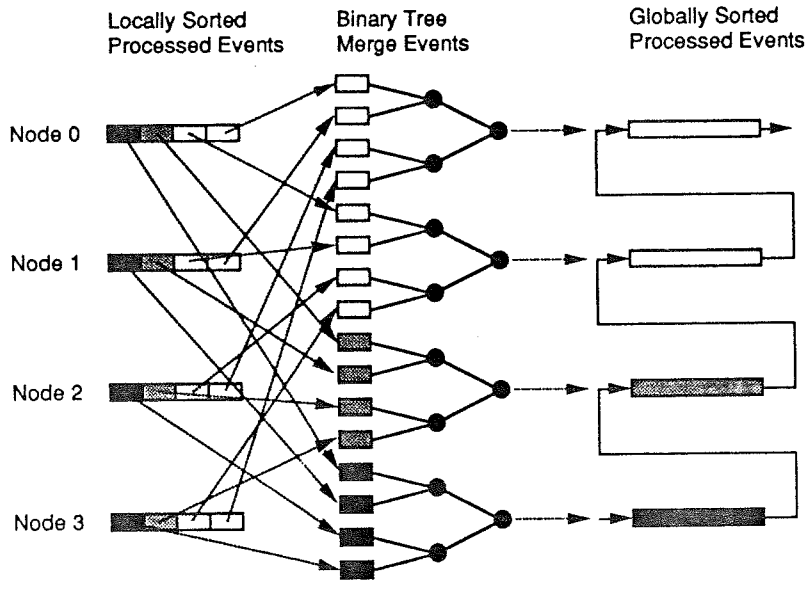


Figure 13. Producing a globally sorted list of processed events.

Merging the N sublists on each node takes $(m/N)\ \log_2 N$ steps. Thus, the time for merging these lists can be written as

$$T_{\text{merge}} = (m/N)\ \log_2 N.$$

It would appear that parallel simulations require an additional amount of work to send globally sorted event information to the external world. However, there is more to consider.

Imagine a simulation in which each event generates a single new event. If m events are processed globally in a particular cycle, then each node will receive, on the average (assuming perfect balance), m/N new events. Thus, m/N new events must be inserted back into each local event queue. This can be accomplished by first sorting the m/N events and then merging them back into the local event queue.

Sorting m events for a simulation running on one node takes $m\ \log_2(m)$ steps. If perfect speedup is attained, one might naively expect it to take $[m\ \log_2(m)]/N$ steps for N nodes. However, each node's performing the task of sorting m/N events only takes $(m/N)\ \log_2\ (n/N)$ steps. There is an apparent superlinear speedup in maintaining the event queue! The amount of time it takes to sort m events on N nodes is better than a factor of N compared with the time on one node. The time for maintaining the event queue can also be written as

$$T_{\text{sort}} = (m/N)[\log_2(m) - \log_2 N].$$

When combining T_{merge} and T_{sort}, the superlinear speedup is exactly canceled. There is no contradiction to the theoretical upper bound for parallel speedup. The best way to understand the apparent superlinear speedup (which is always present in parallel simulations that use local event queues) is to realize that information is lost if the processed events are not regathered into a single globally sorted list for the purpose of output.

6.2 Simulation Time Advancement Rate Control

If humans are allowed to interact with a simulation in progress, it is important for the simulation to advance smoothly in time. In other words, the Simulation Time Advancement Rate (STAR) should be as close to a constant as possible, and equal to one if real-time interaction is desired. Interactive parallel simulations must be able to control the advancement of simulation time with respect to the wall clock.

One important principle in controlling the STAR is that it can always be slowed down; it is always more difficult to speed it up. For example, if a simulation can run two times faster than real time (from start to finish), then pauses can always be added to the simulation to slow it down to real time if desired (see Figure 14, p. 274). Although the average STAR may run two times faster than real time, the instantaneous STAR at any given time can vary. At times, the instantaneous STAR may be slower than real time. Three important points must be made.

First, the parallel simulation algorithm should run as fast as possible. For example, if the same simulation could run with a STAR equal to 10, using a different approach, then slowing it down to real time would be easier than when using an

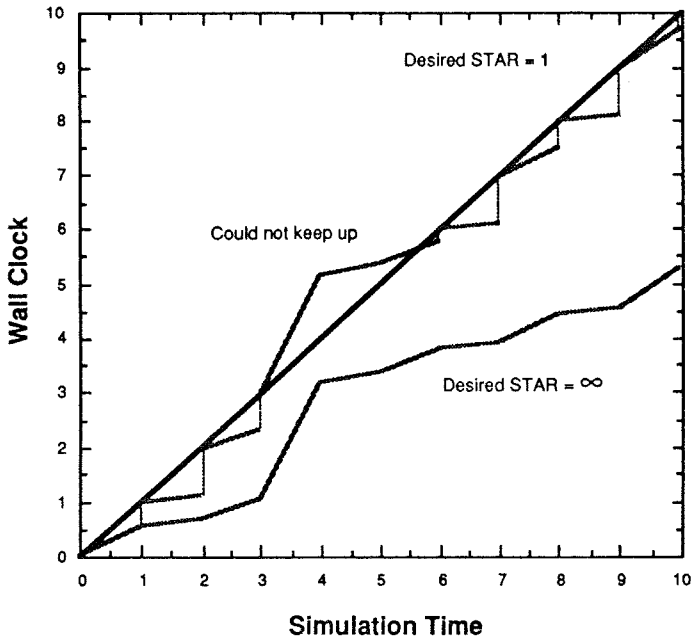


Figure 14. Controlling the STAR by pausing the simulation at the end of each cycle to wait for the wall clock to catch up.

algorithm with a STAR equal to 2. The first and most important goal for any interactive parallel simulation approach should be to run as fast as possible.

Second, a mechanism to smooth the STAR is needed. If the simulation is allowed to progress significantly into the future, the results of the simulation can be buffered. The results can then be released to the external world smoothly in time (i.e., throttled by the wall clock). However, when the outside world interacts with a simulation operating in this manner, rollback may be required to bring the simulation back to the time that was perceived by the user. Rollback due to external interactions requires saving the state of all simulated objects at least as far back as when the interaction occurred. If the simulation is allowed to progress too far into the future, an enormous amount of memory will be required for rollback state saving.

Another option for smoothing the STAR is to process events in large cycles and then, as a rule, not allow external interactions to occur until the next cycle. If the cycles are large enough, then the STAR will be smoothed. The cycles must be throttled by the wall clock to maintain the desired STAR. However, large cycles may force an undesirable time granularity into the interactive simulation, and the user may not be able to interact as tightly with the simulation as desired. Furthermore, the information for each processed event coming from the simulation should also be throttled by the wall clock to avoid a *choppy*-looking simulation.

Third, regardless of whether or not the simulation keeps up with the desired STAR, rigor should always be maintained. Simulation errors (or time accidents) resulting from an attempt to control the STAR should never be allowed to occur.

Setting the desired STAR to infinity should have the same meaning as letting the simulation run as fast as possible.

If the simulation cannot keep pace with the desired STAR, there should be no pauses to throttle the simulation. If the simulation operates in cycles, it could possibly catch up in the next cycle (and should be allowed to). A resolution for the desired STAR should be specified to determine acceptable performance (in other words, how far the simulation can lag behind the desired STAR and still be within specs).

6.3 Human Interactions

In the past, it has been very difficult to support interactive parallel discrete-event simulations. Consider, as an example, the Time Warp algorithm as implemented in SPEEDES. In Time Warp, each node keeps track of its own simulation time. Because of the optimistic event processing, there is no certainty of correctness beyond the GVT. Therefore, Time Warp can release to the outside world only those messages that have time stamps less than or equal to the GVT. Note that we assume that the outside world (e.g., graphics, humans, and external programs) cannot be rolled back.

If only viewing the results of a simulation were desired, there would be no problem. Output from the simulation could be buffered and released only at GVT update boundaries. However, when the outside world tries to interact with the simulation, the situation becomes more difficult.

Humans like to interact (see the Section 6.4.1) with the parallel simulation based on the output that has been received (see Sections 6.4.2 and 6.4.3). The earliest time the user can interact with the simulation is at the GVT. Otherwise, the law governing external rollbacks would be violated. The goal for interactive parallel simulations is to allow the human to interact as tightly with the simulation as possible.

In the SPEEDES implementation of Time Warp, an unexpected external message received from the outside world can cause an object to roll back to the GVT. This allows the tightest interactions. Because conservative algorithms (such as Time Bucket synchronization) do not support rollback, they do not permit the same tight interactive capabilities (see Figure 15, p. 276). This is one of the major drawbacks of conservative algorithms.

6.4 External Modules

Interactive SPEEDES accommodates external interactions by using a host program to service communications between the central parallel simulation and the outside world (see Figure 16, p. 276). The host program allows external modules to establish connections to the central parallel simulation using UNIX Berkeley Sockets.

One important characteristic of the SPEEDES approach is that external modules (i.e., external computer programs that would like to be a part of the simulation) are not required to participate in any of the high-speed synchronization protocols.

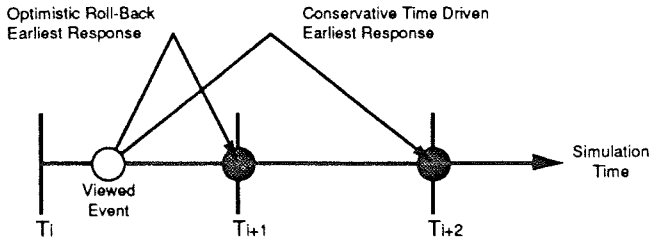


Figure 15. Earliest user interaction based on a viewed event. T_i, T_{i+1}, and T_{i+2} represent time cycle boundaries for the conservative Time-Driven approach and, similarly, GVT updates for optimistic approaches such as Time Warp. In both cases, the user views an event in the first cycle. Meanwhile, the second cycle is being processed. Because conservative time-driven simulations do not support rollback, the earliest human interaction can occur at the start of the third cycle. Optimistic approaches that support rollback can handle the interaction at the start of the second cycle. Optimistic approaches are capable of handling tighter interactions than conservative approaches, at the expense of rollback.

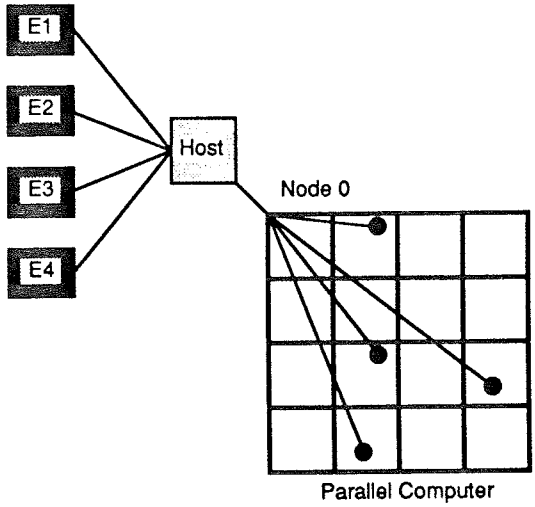


Figure 16. Assumed topology for interactive SPEEDES. Fine-grain objects are simulated on the parallel computer. External models (users) can communicate with the simulation objects by routing messages through the host.

Instead, a hybrid approach is used. This is extremely important for interactive simulations over networks that have high latencies. The high-speed central simulation runs on the parallel computer and provides control mechanisms to the outside world.

External modules view the parallel simulation much as a central controller views it. The external modules are still event-driven, but they must not communicate too often with the central simulation. Otherwise, the simulation will be bogged down by the large communication latencies.

Interactive SPEEDES does not make any assumptions concerning the number

of external modules or human users participating in the simulation. In fact, the number can change during the course of a simulation. The connection procedure simply involves establishing a communication socket to the host.

All the synchronization algorithms within the SPEEDES environment currently support the interactive capabilities described subsequently.

6.4.1 Query

A very useful capability interactive SPEEDES supports is the ability to query the state of a simulation object while the simulation is in progress. The simulation can be viewed as a large database of objects that change in time. The QUERY function allows an external user to probe into the objects of the simulation to determine how they are performing.

6.4.2 Monitor

The MONITOR capability allows the state of a particular simulated object to be monitored as its events occur. The effect of every event for that object can be sent back to the external monitoring module. This can be extremely useful as an analysis tool for studying the behavior of various components within the parallel simulation.

6.4.3 Command

The COMMAND function supported by interactive SPEEDES allows a user to send a command (or generate an event) to a simulation object. This permits users to change the simulation while it is in progress. Commands should work in conjunction with the QUERY and MONITOR functions so the user can change the simulation based on what is perceived.

6.4.4 External Module

The last interactive function SPEEDES supports is the control of an external module from within the parallel simulation. It is assumed that external modules are remote objects that tend to have long opaque periods between communications. They are controlled by an object simulated on the parallel computer. The external module attaches itself to a simulation object and then is controlled by that object.

External modules do not participate in the high-speed synchronization algorithms supported internally within SPEEDES. Rather, they are given input messages with a start time, an end time, and their data to process. When the external module has completed processing its data, a *done* message is sent back to the controlling simulation object. This causes another message to be sent back to the external module, and processing continues.

If the done message has not arrived before the appropriate simulation time, the parallel simulation (which is running faster than the external module) waits. If the done message arrives early, the external module (which is running faster than

the parallel simulation) will have to wait for the simulation to catch up before it receives its next message. When an external module disconnects from the simulation (whether purposely or accidentally), this blocking mechanism is removed automatically.

7 CALTECH/JPL MARK III HYPERCUBE

The Caltech/JPL Mark III 64-Node Hypercube supported the SPEEDES operating system for the simulation studies described below. Each node of a Mark III hypercube has a 68020 microprocessor and contains 4 Mbytes of local memory available to the application. The Centaur operating system (a mixture of synchronous and asynchronous communication utilities) and the useful synchronous global line were also used to support SPEEDES. Table 1 shows the relevant Mark III communication delays.

8 PERFORMANCE RESULTS

Because of the multiple-synchronization algorithms supported within SPEEDES, the SPEEDES environment can also serve as a test bed for parallel simulation studies. To date, only two simulations have been written in the SPEEDES environment. However, they show some very interesting trends. In each of the simulations, all the algorithms got the same exact answer, which validates the correctness of their implementations.

8.1 Queuing Network

A simulation of a fully interconnected FIFO queuing network was used as the original benchmark for the Breathing Time Buckets algorithm. In this simulation, messages were simply relayed randomly from one server to another (with the appropriate queuing delays included). A random flat distribution was used for the message service time. In addition, a minimum service time, T, was introduced in the queuing model to allow the Time Bucket synchronization method to participate in the comparisons. Spin loops that waste CPU time were supported as a run-time option so event processing granularities and work-load imbalances could also be studied. Communication propagation delays that might exist in real-world queuing networks were also specified at run time.

The main focus of the queuing network study was to verify the validity of the Breathing Time Buckets algorithm. Most of the analysis that follows was performed with Breathing Time Buckets. Table 2 lists the relevant parameters for five network configurations that were studied. Note that the number of events in the system at any time is a constant because each event generates another single event when it is serviced by the queue. The total number of events processed by the simulation is accumulated in the last column of Table 2.

Table 1. 64-Node Mark III Communications

SPEEDES Usage	Mark III Utility	Overhead
Non Blocking Sync.	*nb_sync()*	0.01 ms
Asynchronous Broadcast	*send_msg_w()*	1 ms
Update GST	*combine()*	1 ms
Route messages	*multirout()*	100,000/sec.

Table 2. Network Configurations

Network Name	Num. Queues	Num. Events	Service Time	Comm. Time	Spin Loops	Total Events
Spin Loop	8192	8192	0	0.2 - 1	50 ms	30,416
Min. Comm.	8192	8192	0	0.2 - 1	0	353,290
8192 Queues	8192	8192	0 - 1	0 - 1	0	256,349
1024 Queues	1024	8192	0 - 1	0 - 1	0	382,742
64 Queues	64	8192	0 - 1	0 - 1	0	411,884

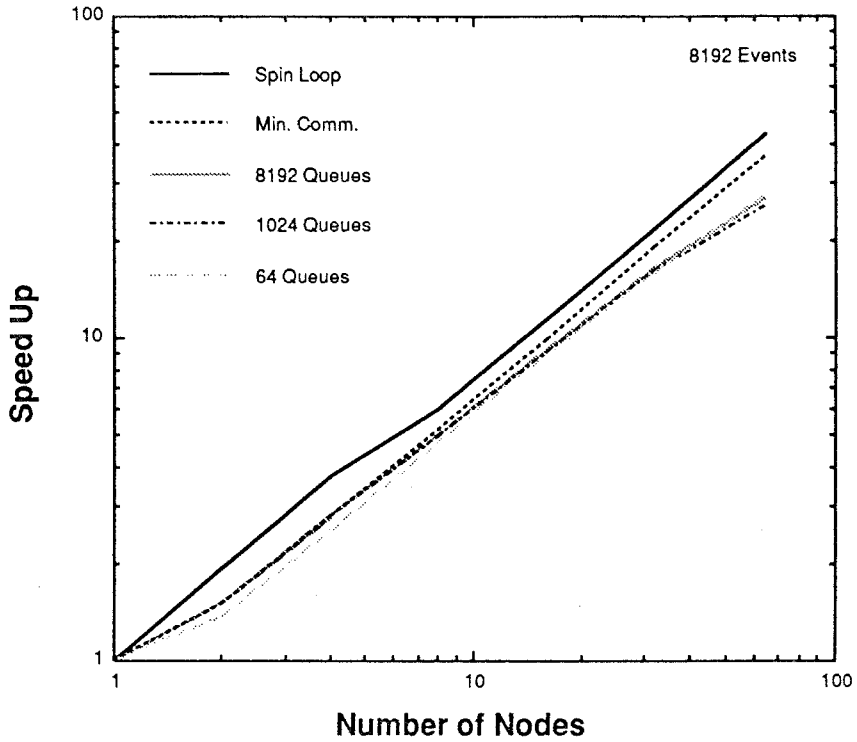


Figure 17. Speedup curves. Table 2 describes the configuration of the networks for each of these plots. Speedup is measured relative to the efficient single-node version supported within SPEEDES.

The *Spin Loop* network had the best overall performance because the extra parallel processing overhead became negligible compared with the event processing time. The *Min. Comm.* network also had excellent performance because many events could be processed in each Breathing Time Bucket cycle. However, as

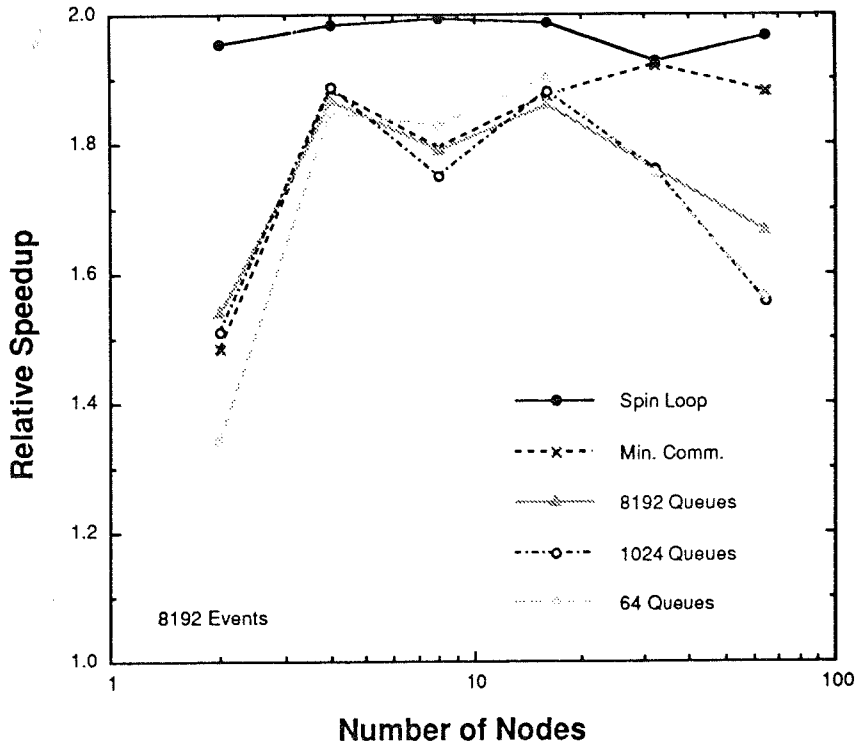


Figure 18. Relative speedup measured successively at each step as the number of nodes is doubled. The first point of each curve (which is the CPU time for one node divided by the CPU time for two nodes) demonstrates the extra parallel overhead required by Breathing Time Buckets. The last point of each curve shows how much speedup was gained when going from 32 to 64 nodes.

expected, there was a large drop in efficiency when going from one node to two nodes because of the parallel processing overhead. The last three networks (*8,192, 1,024,* and *64 Queues*) describe different levels of network saturation (the total number of events in the network is a constant 8192). That is, the networks have 1:8:128 events on the average for each queue in these respective simulations. The Breathing Time Buckets algorithm was expected to have better performance for networks with high saturation because events tend to wait a long time in their queues before being serviced. However, results showed that this was not necessarily the case: One server in the network tended to race ahead of the others. This caused the simulation of the network to behave very much like the unsaturated case. It was observed that during transitions (i.e., when the ID of the fastest server in the network changed), a larger number of events were serviced in parallel with tremendous efficiency.

Figure 17 shows the speedup performance obtained for the five network configurations. The number of events remained a constant 8,192 for each case. The speedup performance was measured relative to an efficient single-node version supported within SPEEDES (with rollback and communication overhead re-

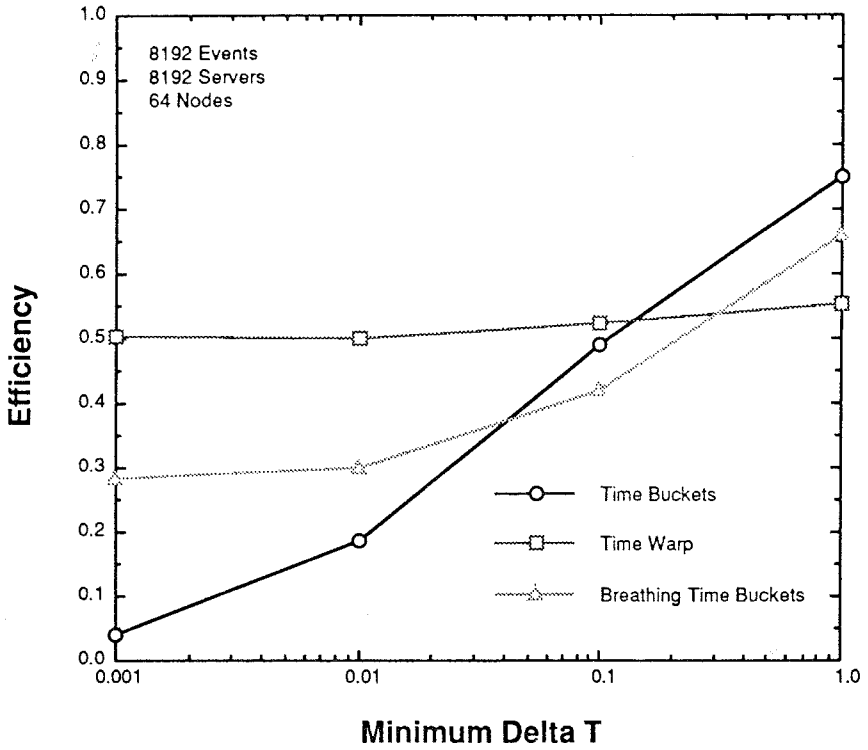


Figure 19. A small, fully interconnected queuing network on 64 nodes. The efficiency is plotted as a function of the minimum service time, *T*.

moved). It is interesting to notice how Breathing Time Buckets behaved when going from one node to two nodes. Although some speedup was achieved, the overhead for parallel processing became evident.

Figure 18 depicts the relative speedup each time the number of nodes is doubled for the five networks described earlier. There is a noticeable loss of efficiency when going from one node to two nodes. However, even when going from 32 to 64 nodes, large relative speedup factors (better than 1.6) are still observed. This is a good indication that performance saturation (typically noticed in parallel simulations) has not been reached. Although the curves are moving downward, Breathing Time Buckets might be expected to continue demonstrating speedup even for 128 nodes or more. This is because the number of events in a cycle for these networks is typically a few hundred. Larger queuing networks have been simulated with even better parallel performance. However, the test simulation must be small enough to fit in the memory of one node for accurate speedup measurements.

To test the system's limits, a network of 262,144 queues with 262,144 initial events (the *8,192 Queues* simulation scaled by a factor of 32) was simulated on both 32 and 64 nodes. Processing 24,000,000 events on 32 nodes took 4,716 s, but took only 2,447 s on 64 nodes. This relative speedup factor of 1.93 is far from the parallel saturation plateau. Furthermore, a network with 500,000 queues and 500,000

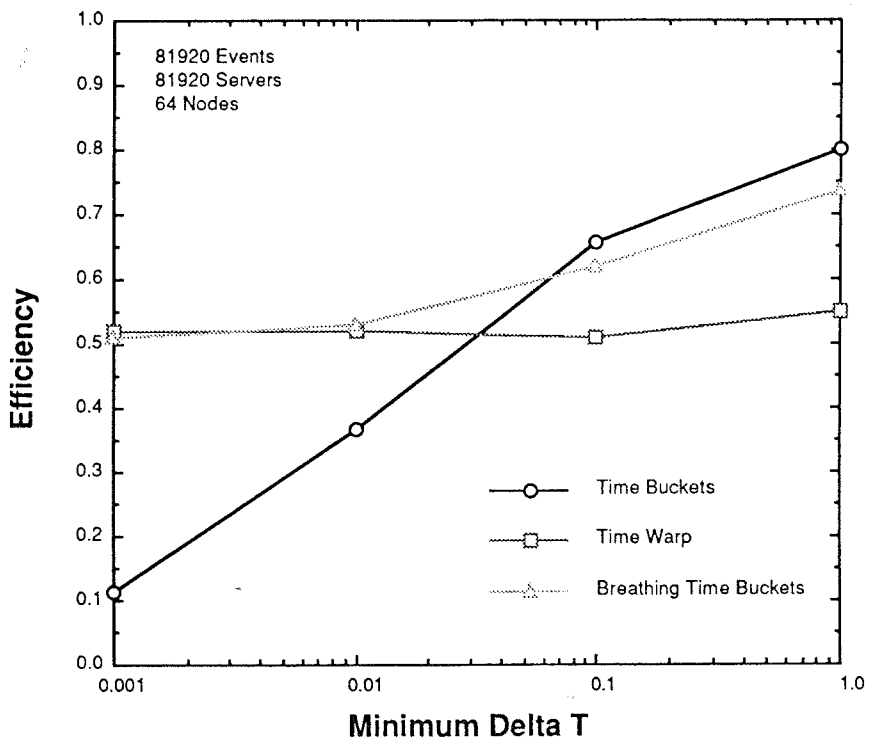


Figure 20. A large, fully interconnected queuing network on 64 nodes. The efficiency is plotted as a function of the minimum service time, *T*.

initial events was simulated on 64 nodes. A total of 84,000,000 events were processed in 10,000 s. A speedup factor in excess of 32 was estimated by comparing the average event processing time with the single-node event processing time measured by the *8,192 Queues* simulation (they should be similar). These large simulations confirm the intuition that Breathing Time Buckets becomes more efficient as the problem gets larger.

The Breathing Time Buckets algorithm was also compared with Time Buckets and Time Warp on 64 nodes. Figure 19 shows a small network of 8,192 servers and 8,192 events. As the minimum service time was increased, Time Buckets and Breathing Time Buckets improved. Time Warp performed quite well and was not affected by changes in the minimum service time. Time Warp also exhibited very few rollbacks.

Figure 20 shows the queuing network's performance when the network was made 10 times larger. Now there were 81,920 servers and events. The Breathing Time Buckets algorithm was more efficient now that it had more events to process in each cycle. It performed as well as or better than Time Warp.

It is easy to see how the Breathing Time Buckets algorithm converges with the Time Bucket algorithm when the minimum service time is increased. The Breathing Time Buckets algorithm always processes at least as many events in a time cycle as the Time Bucket approach. When the Time Bucket synchronization is efficient,

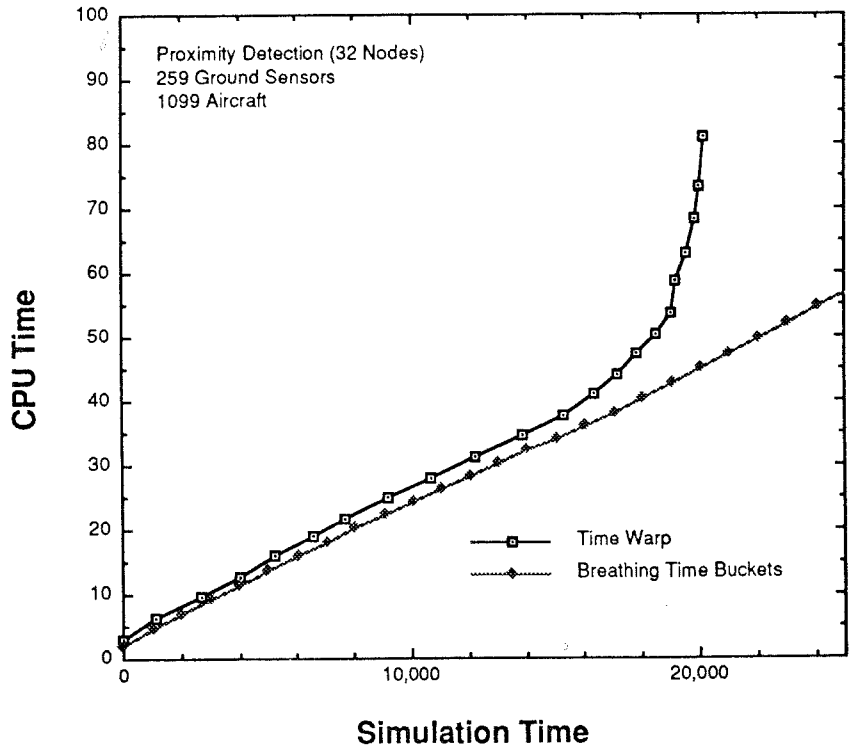


Figure 21. Proximity detection on 32 nodes. Time Warp begins to exhibit a rollback explosion at about 18,000 s into the simulation. This is evident by the simulation time stagnating while the CPU time increases.

Breathing Time Buckets will also be efficient. However, the converse may not be true, as is shown in Figure 20 for small T.

8.2 Proximity Detection

The second simulation implemented in the SPEEDES environment involves a fascinating solution to a very tricky problem for parallel simulations, namely, Proximity Detection. Very often (especially in military simulations), objects need to know which objects are within their sensing range. In other words, "Who can see whom?"

A new solution to this problem, the *Distribution List Approach*, has been developed [20]. A report on this technique is still forthcoming, but the preliminary results are quite startling.

The Proximity Detection simulation involves large numbers of aircraft randomly flying all over the globe and detected by large numbers of radar sensors. Figures 21 and 22 show a comparison between the Time Warp and Breathing Time Buckets approaches. In this simulation, the radar sensors were performing their proximity detection, but were not permitted to scan. This made the simulation's

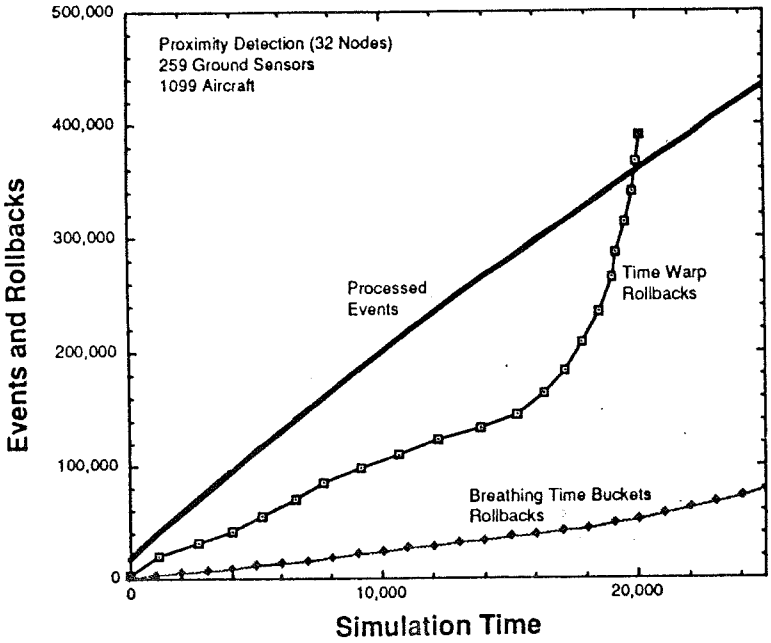


Figure 22. Proximity detection on 32 nodes. Time Warp begins to exhibit a rollback explosion at about 18,000 s into the simulation. This is evident by the increased number of rollbacks. Note that the Breathing Time Buckets algorithm does not suffer from this problem because antimessages are not required.

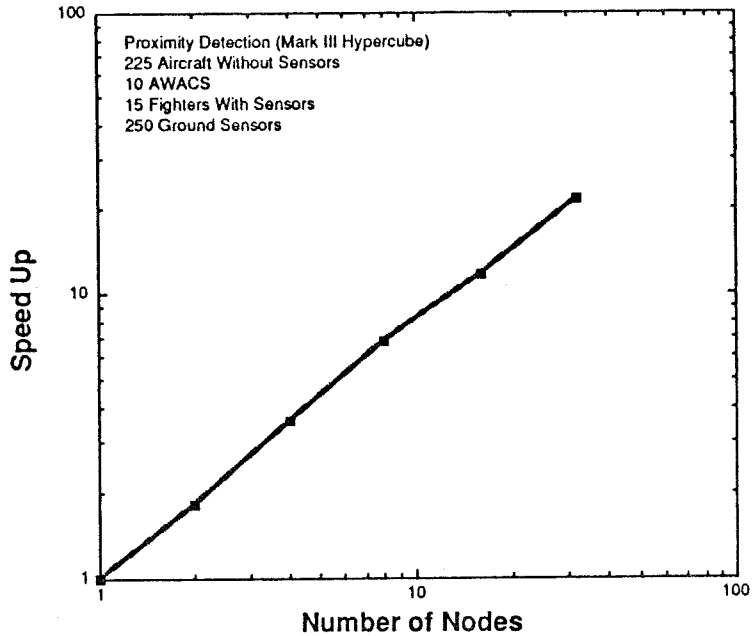


Figure 23. Proximity detection speedup as a function of the number of nodes using Time Warp.

granularity very low. We saw a rare but startling effect at about 18,000 s into the simulation: Time Warp began a rollback explosion. Because the Breathing Time Buckets approach does not require antimessages, there is no such effect.

Figure 23 shows another run of the Proximity Detection simulation using the Time Warp algorithm (Time Warp here was faster than Breathing Time Buckets by a small margin). In this simulation, the radar sensors were permitted to scan. This increased the simulation's granularity. Excellent speedup (in the low 20s on 32 nodes) was achieved, and there was no sign of speedup saturation or rollback explosion.

9 SUMMARY

This paper has demonstrated how an object-oriented approach for building parallel discrete-event simulation synchronization mechanisms can support multiple protocols without requiring the user to recompile code. SPEEDES offers efficient sequential simulation, Time Bucket Synchronization, Breathing Time Buckets (a new algorithm), Time Warp, and interactive hybrid protocols. Other synchronization strategies, such as the Time-Driven and CO-OP approaches, can easily be supported and look promising.

To date, two simulations have been written in the SPEEDES environment (Queuing Network and Proximity Detection). Results show that there is no "one best synchronization mechanism." All synchronization mechanisms have scenarios in which they perform best. Furthermore, performance is hardware-dependent. SPEEDES reduces the risk of forcing the simulation user to commit to the wrong approach. It offers options.

REFERENCES

[1] P. Reynolds, "A Spectrum of Options for Parallel Simulation," *Proc. of 1988 Winter Simulation Conf.* 1988, pp. 325–332.

[2] R. Fujimoto, J. Tsai, and G. Gopalakrishnan, "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp," Technical Report UUCS-88-011, Department of Computer Science, University of Utah, Salt Lake City, 1988.

[3] D. Jones, "An Empirical Comparison of Priority-Queue and Event-Set Implementations," *Communications of the ACM*, Vol. 29, No. 4, pp. 300–311, 1986.

[4] D. Sleator, and R. Tarjan, "Self-Adjusting Binary Search Trees," *Journal of the ACM*, Vol. 32, No. 3, pp. 652–686, 1985.

[5] G. Fox, *Solving Problems on Concurrent Processors*, Vol. 1, Englewood Cliffs, NJ: Prentice-Hall, 1988.

[6] J. Steinman, "Multi-Node Test Bed: A Distributed Emulation of Space Communications for the Strategic Defense System," *Proc. of Twenty-First Annual Pittsburgh Conference on Modeling and Simulation*, Vol. 21, Part 3, May 1990, pp. 1111–1115.

[7] R. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, Vol. 33, No. 10, pp. 30–53, 1990.

[8] D. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404–425, 1985.

[9] F. Kaudel, "A Literature Survey on Distributed Discrete Event Simulation," *Simuletter*, Vol. 18, No. 2, pp. 11–21, 1987.

[10] Y. Lin, and E. Lazowska, "Exploiting Lookahead in Parallel Simulation," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 4, pp. 457–469, 1990.

[11] D. Nicol, "Performance Bounds on Parallel Self-Initiating Discrete Event Simulations," Technical NASA Contractor Report 182010, ICASE Report No. 90-21, 1990.

[12] B. Lubachevsky, "Bounded Lag Distributed Discrete Event Simulation," *Multiconference on Distributed Simulation*, February 1988, pp. 183–191.

[13] L. Sokol, B. Stucky, and V. Hwang, "MTW: A Control Mechanism for Parallel Discrete Simulation," *International Conference on Parallel Processing*, Vol. 3, August 1989, pp. 250–254.

[14] D. Dickens, and P. Reynolds, "SRADS with Local Rollback," *Proc. of SCS Multiconference on Distributed Simulation*, Vol. 22, No. 1, January 1990, pp. 161–164.

[15] P. Reynolds, "An Efficient Framework for Parallel Simulations," *Proc. of SCS Multiconference on Advances in Parallel and Distributed Simulation*, Vol. 23, No. 1, 1991, pp. 167–174.

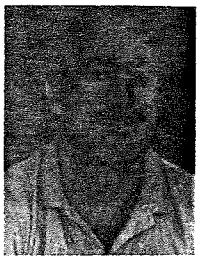
[16] R. Felderman, and L. Kleinrock, "Two Processor Time Warp Analysis: Some Results on a Unifying Approach," *Proc. of SCS Multiconference on Advances in Parallel and Distributed Simulation*, Vol. 23, No. 1, 1991, pp. 3–10.

[17] Y. Lin, and E. Lazowska, "Processor Scheduling for Time Warp Parallel Simulation," *Proc. of SCS Multiconference on Advances in Parallel and Distributed Simulation*, Vol. 23, No. 1, 1991, pp. 11–14.

[18] K. Chandy, and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 5, pp. 440–452, 1979.

[19] R. Fujimoto, "Lookahead in Parallel Discrete Event Simulation," *International Conference on Parallel Processing*, Vol. 3, 1988, pp. 34–41.

[20] F. Wieland, F. Sanders, and J. Steinman, personal communication, 1991.



Jeff Steinman received his B.S. degrees in computer science and mathematical physics from California State University Northridge in 1980. He then began working at Hughes Aircraft Company in the Radar Systems Group for four years while studying physics at UCLA. In 1988, he received his Ph.D. in experimental particle physics from UCLA, where he measured the quark content of high energy virtual photons generated at the Stanford Linear Accelerator Center. Since then, he has been working as a member of the technical staff for the Hypercube Project at the Jet Propulsion Laboratory building simulations for strategic missile and air defense. Two outcomes of his work have been the SPEEDES operating system and the Breathing Time Buckets algorithm for