Inventor: Jeffery S. Steinman
Contractor: Jet Propulsion Laboratory

JPL Case No. 18414
NASA Case No. NPO-18414-1-CU
Date: January 17, 1992

SYNCHRONOUS PARALLEL SYSTEM FOR EMULATION AND DISCRETE EVENT SIMULATION

AWARDS ABSTRACT

A synchronous parallel system for emulation and discrete event simulation having parallel nodes responds to received messages at each node by generating event objects having individual time stamps, stores only the changes to the state variables of the simulation object attributable to the event object and produces corresponding messages. The system refrains from transmitting the messages and changing the state variables while it determines whether the changes are superseded, and then stores the unchanged state variables in the event object for later restoral to the simulation object if called for. This determination preferably includes sensing the time stamp of each new event object and determining which the new event object has the earliest time stamp as the local event horizon, determining the earliest local event horizon of the nodes as the global event horizon, and ignoring events whose time stamps are less than the global event horizon. Host processing between the system and external terminals enables such a terminal to query, monitor, commmand or participate with a simulation object during the simulation process.

(NASA-Case-NPO-18414-1-CU) SYNCHRONOUS PARALLEL SYSTEM FOR EMULATION AND DISCRETE EVENT SIMULATION Patent Application (NASA) CSCL 09B

N92-24045

Unclas

NASA CASE NO. NPO-18414-1-CU

PRINT FIG. 1

NOTICE

Serial Number: 07/880,211

Filed Date: January 21, 1992

NRO-JPL

MAY 13 1992

Serial No. 07/880,211
Filing Date 1-21-92
Contract No. NAS7-918
Contractor Caltech/JPL
PATENT APPLICATION
Pasadena CA 91109
Date (City) (State) (Zip)

JPL Case No. 18414
NASA Case No. NPO-18414-1-CU
Attorney Docket No. JPL/001-92

SYNCHRONOUS PARALLEL SYSTEM FOR EMULATION AND DISCRETE EVENT SIMULATION

ORIGIN OF INVENTION

The invention described herein was made in the performance of work under a NASA contract, and is subject to the provisions of Public Law 96-517 (35 USC 202) in which the contractor has elected not to retain title.

TECHNICAL FIELD

The invention relates to discrete event simulation of objects using a plurality of synchronous parallel computers in communication with each other so that the objects being simulated may interact.

BACKGROUND ART

Discrete event simulation of objects on a single digital processor is not very difficult. In the standard approach, all events associated with a simulated object are tagged with a time index, inserted in an event queue, and maintained in increasing time order by the event queue as events in the simulation are scheduled at discrete points in time. Simulation proceeds in the computer by processing the event from the queue having the lowest time index. The resulting simulation of events in sequence is thus defined by the time indices.

Processing an event can affect the state variables of an object and can schedule new events to occur in the future for one or more simulated objects. This interaction of cause and effect requires that new events generated be tagged with time indices greater than or equal to the current simulation time index. The generated new events are simply inserted into the event queue in their proper time index sequence.

Discrete event simulation on parallel processors is necessarily very different from the single processor approach described above. (See D.A. Reed, "Applications: Distributed Simulation," Multicomputer Networks: Message-Based Parallel Processing, The MIT Press, Cambridge Massachusetts, pp. 239-267, 1987.) While it is clear that real world objects may interact concurrently in time, it is not always obvious how to rigorously simulate them on parallel processors. The event queue approach presents the problem of having each processor of the parallel array continually determine whether it should process the next event in its queue, or wait because a new event with an earlier time index is arriving from another processor. Moreover, the simulation program would have to be optimistic that events tagged for simulation at a later time index would not be dependent upon the results of other events triggered by events simulated conservatively up to the time of the next event in the queue.

Various techniques have been proposed to solve this problem, each with its respective strengths and weaknesses. This background discussion will cover only the parallel simulation techniques that are relevant to the understanding of the present invention.

The simplest time driven approach to parallel simulation makes use of the **causality** principle as illustrated in J.S. Steinman, "Multi-Node Test Bed: A Distributed Emulation of Space Communications for the Strategic Defense System," Proceedings of the Twenty-First Annual Pittsburgh Conference on Modeling and Simulation, Pittsburgh, 1990. The causality principle allows for events scheduled between time **0** and time **T** to be processed conservatively in parallel up to the event horizon at time **T**.

The event horizon for a cycle is defined to be the point in time where an event to be processed has a later time index than the earliest new event generated in the current cycle. Simulation errors can occur **if** events are processed optimistically beyond the event horizon. For this scheme, known as the **time-bucket** approach, the minimum time delay **T** between an event and any of its generated events must be known in order to predict the event horizon. Parallel processing can then take place in cycles of duration **T**. As long as the minimum time interval between events and the events that they generate is known, the simulation can proceed in time cycles of duration **T**.

This time-bucket approach has the important property of requiring very little overhead for synchronization. For example, each processor in the Hypercube array of processors need only synchronize with all of the other processors at the end of every cycle, after which all processors increment their simulation time in unison by the amount **T** and proceed to simulate other scheduled events.

Despite the low synchronization overhead of the time-bucket approach, there are some major drawbacks to that approach. The cycle duration T must be large enough so that each processor is able 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. Another important problem is the balancing of the work load. Because of the synchronous nature of the time-bucket approach, when one processor has more work to do than other processors in a cycle, the simulation will be inefficient. Because of these drawbacks, a more flexible approach is needed.

Optimistic discrete event simulation approaches must allow for event simulation to occur in error, but when one does occur, a **roll-back** algorithm is needed to undo the erroneously simulated event. Various optimistic approaches have been proposed (L. Sokol, D. Briscoe and A. Wieland, "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution," Proceedings of the SCS Distributed Simulation Conference, Vol. 19, No. 3, pp. 34-42, 1988; K. Chandy and R. Sherman "Space Time and Simulation," Proceedings of the SCS Distributed Simulation Conference, Vol. 21, No. 2, pp. 53-57, 1989.) By far the most popular optimistic approach is the **time-warp** operating system (D. Jefferson, "Virtual Time," ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, pp. 404-425, 1985) in which simulation errors are handled by the generation of **antimessages** which cause the simulation to roll back to a time before the simulation error occurred.

Because some events can generate future events, and they in turn can generate other future events, cascading of the error may occur which complicates the roll-back algorithm. Messages and state variables must be saved for each processed event in order to be able to implement a rollback algorithm if it becomes necessary.

Traditional time-warp implementations have required a large amount of memory overhead. That memory overhead could be better used for the simulation data. While it is true that as long as the roll-back overhead is small compared to the average amount of time it takes to process an event, the time-warp approach will have high performance, but larger data processing units typically execute programs faster, thereby increasing the occurrences of time warp. In that case, the memory overhead of time warp could reduce the overall simulation performance to an unacceptable level.

STATEMENT OF THE INVENTION

A new method has been developed for synchronous parallel environment for emulation and discrete event simulation. Central to the new method is a technique called **breathing time buckets (BTB)** which uses some of the conservative techniques found in the prior-art time-bucket synchronization, along with some of the optimistic techniques of the prior-art time-warp approach.

An event is created by an input message generated internally by the same processor or externally by another processor. A system for routing messages from each processor to designated processors, including itself (hereinafter referred to as a "multirouter" directs the message to the processor that is intended to process the event. The events are defined through various virtual functions by the user during initialization. It is through these virtual functions that events are processed. Note that multiple messages for an object with the same time index will generate multiple events for thate object, not a single event for multiple messages. The events are thus initialized by data contained within the messages. After initialization the messages are discarded, and each event is attached to its own simulation object.

A processor **optimistically** performs its calculations for the event and generates messages to schedule future events to be generated in the same processor or any other processor, but the generated messages are not immediately released. Changes required in the variables of the object affected by the event are calculated and stored. Immediately afterwards the changes calculated are exchanged for the values of the affected variables of the object. If for any reason the variables should not yet have been affected, such as because an event processed by another object generates a message for the affected object in its past, the event being generated must be rolled back. That is accomplished in the **BTB** algorithm by exchanging back the computed changes for the old values of the affected variables and canceling any messages generated but not yet released. In that manner, the shortcomings of the prior-art time-bucket technique are overcome in most situations by permitting events to be optimistically processed, and if it results that a message should not have been processed, the processed event is rolled back and any messages generated in the processing of the event are discarded.

External interactions are made possible by using a host program connected to the parallel computers that services communications between external user modules and the parallel computers. A useful interactive capability is the ability for a user to query or monitor the state of simulation objects while the simulation is in progress. For this purpose, the simulation system of parallel computers constitutes a large data base of objects that can be accessed from a user module. Further useful interactive capabilities are to issue commands from the outside world (which schedules events within the parallel simulation), and to synchronize external modules dynamically.

The novel features that are considered characteristic of this invention are set forth with particularity in the appended claims. The invention will best be understood from the following description when read in connection with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram illustrating the object-based architecuture at a single node of the invention.

FIG. 2 is a timing diagram illustrating three successive cycles of operation of the invention and the event horizons thereof.

FIG. 3 is a block diagram illustrating the architecture of the main program of the invention.

FIG. 4 is a block diagram illustrating the operation of the invention using a time warp protocol.

FIG. 5 is a block diagram illustrating the operation of the embodiment corresponding to Fig. 4 whenever an antimessage is transmitted.

Fig. 6 is a timing diagram illustrating the operation of a preferred embodiment of the invention.

Fig. 7 is a timing diagram illustrating one method of operating the embodiment corresponding to Fig. 6.

Fig. 8 is a timing diagram illustrating the preferred method of operating the embodiment corresponding to Fig. 6.

Fig. 9 illustrates how processed events are globally sorted in accordance with the invention.

Fig. 10 is a graph illustrating one aspect of the performance of the invention.

Fig. 11 is a timing diagram illustrating two regimes for responding to an earlier viewed event of the simulation.

Fig. 12 is a block diagram illustrating the host interactive architecture of the invention.

DETAILED DESCRIPTION OF THE INVENTION

The object-based architecture of the simulation process of the invention carried out at each node is illustrated for a single simulation object in Fig. 1. Discrete event simulation of objects begins with some basic steps for a single processor, such as a processor at a node of a Hypercube. First an event object is initiated by an input message **10** for a simulation object received via a multirouter **11** from the same processor or another processor. Time tagged messages received are queued in an event library **12**. Multiple messages for a simulation object with the same time index will generate multiple event objects for the simulation object.

All event objects are user-defined as to their inherent capabilities from a base-class of generic simulation objects, where the term "objects" refers to object oriented programming techniques used to simulate physical objects assigned to processors for simulation of events, such as missiles, airplanes, tanks, etc., for simulation of war games, for example.

Event objects **14** are initialized by data contained within the messages received. After an event object is initialized, the message for it is discarded. Each event object is then attached to its own simulation object by a pointer to the simulation object **15**.

Processing an event object in a processor is done in multiple steps that are written by the user into the simulation program. In the first step, an event object optimistically performs its calculations and generates messages **13** to schedule future events. However, the event object of the input message **10** is not immediately executed, i.e., the state of the simulation object, is not changed, and the messages for future event objects are not immediately released. Instead, the state changes and the generated messages are stored in the event object **14**. Only the changes of the simulation object state variables are stored within the event object **14**.

In the second step, the state variable changes that were computed in the first step are exchanged with the simulation object **15** so that the event object then has the old state values and the simulation object has the new values. For example, the state variables may consist of 1000 bytes. If the event requires only four bytes to be changed, only those four bytes are saved and exchanged. If rollback is later required, another exchange restores the previous state of the simulation object.

This feature, referred to as "delta exchange," reduces memory used in optimistic simulations at the expense of having to supply the exchange code in the simulation. Performing a delta exchange involves negligible time, so that rollback is carried out efficiently when needed without the need of special-purpose hardware.

The simulation program may include as part of delta exchange, the step of each time writing out to files these deltas. The simulation may then be rewound if rollback is necessary through several pairs of steps resulting in a reverse delta exchange for several events in sequence 16, thus restoring the changes in reverse order from the files.

A delta exchange completes the first phase of carrying out an event, but as just noted, although the state of the simulation object is changed in the first phase, it can be rolled back. In the second phase, further processing is carried out, such as cleaning up memory, or sending messages **13** out to this and/or other processors and to graphics for record or display. This phase is carried out only after the event object is known to be valid so that there is no possibility of a rollback being required. Consequently, it is usually performed much later in time than the two steps in the first phase, but always without changing the state variables of the simulation object.

SPEEDES Internal Structure:

While other multiple-synchronization systems (or test beds) have been developed, 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).

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 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.

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).

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 one 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.

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. Further, because the Delta Exchange mechanism normally has low overhead, special-purpose hardware to support rollback efficiently may not be necessary.

The Delta Exchange mechanism has the added benefit of permitting fast rewind capabilities. Much like 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.

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 out 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.

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 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 continually maintained. The primary list is sorted, while 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. How the invention processes event objects in successive cycles defined by an event horizon is illustrated in Fig. 2, which is discussed in detail below with reference to the description of the Breathing Time Buckets simulation protocol. Basically, in Fig. 2 events 20 generated during one cycle of the simulation become pending events 22 during the next cycle. Each cycle only processes those pending events 22a which do not occur beyond the event horizon 24 of that cycle. Those pending events 22b which occur beyond the event horizon are not processed during the current cycle.

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, if enough events are collected in the secondary queue on the average for each cycle.

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 crates 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 illustrated in Fig. 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. Many events are typically processed in this step. When it is determined that enough events have been processed and that it is time to synchronize, the global simulation time (for example, 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.

Message Sending

SPEEDES uses both synchronous and asynchronous message sending approaches. Time Warp uses the asynchronous style, while the other algorithms synchronously send their messages.

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.

SPEEDES SIMULATION PROTOCOLS

As illustrated in Fig. 3, the SPEEDES main program interfaces through a generic event queue with any one of several different protocols, including the well-known protocols of time warp event queue, time bucket event queue and sequential event queue. This section briefly discusses the well-known parallel simeulation protocols supported by SPEEDES, while 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.

Sequential Simulation

When SPEEDES runs on one node, the sequential event queue object is automatically created. All the overhead for message sending and rollback is removed. The user still generates messages for his 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).

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 is known, the simulation can proceed in time cycles of duration T. This approach is called Time Bucket Synchronization. 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 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 the value zero. Thus, simulations that can run under time Bucket synchronization are a subset of all parallel discrete event simulations.

Time Warp

The Time Warp algorithm has been heavily discussed in the literature. 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 immediately handed to the TWOSMESS server object supported by SPEEDES. This object assigns a unique 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.

Referring now to Fig. 4, 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 automatically constructed from the message and is handed to the Time Warp event queue object. 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 Delta Exchange method for all the events processed by that object in reverse order and generating antimessages. Aggressive cancellation is used.

Referring now to Fig. 5, 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.

Periodically (typically every 3 seconds of wall-clock time), the Global Virtual Time (GVT) 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, as their antimessages are no longer needed.

BREATHING TIME BUCKETS

The original SPEEDES algorithm (Breathing Time Buckets) is a new protocol or windowing parallel simulation strategy with some unique properties. Instead of exploiting lookahead on the message receiver's end or using preknown or calculable delays, it uses optimistic processing with local rollback. However, unlike other optimistic windowing approaches, it never requires antimessages. Local rollback is not a unique concept either. However, the Breathing Time Buckets algorithm allows full connectivity between the simulation objects (often called logical processes).

Fundamental Concepts

The essential synchronization concept for Breathing Time Buckets is the causality principle. Like 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.

Referring now to Fig. 6, 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, as illustrated in Fig. 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 erroneously generated. 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.

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 event up to their local event horizon and have stopped at a synchronization point.

The next step is for node to synchronously communicate its value 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 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 which were 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 mlog(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 the 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 stimulated object. Thus, rollback overhead should remain small. Antimessages are never needed because bad messages (which would turn into bad events) are never released.

Asynchronous Broadcasts

If the Breathing Time Buckets algorithm 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.

A simple mechanism to solve this problem incorporates an asynchronous broadcast mechanism that tells all the nodes when a local event horizon is crossed, and is illustrated in Fig. 8. When one node crosses its local boundary, it broadcasts this simulation time to all the other nodes. when a node receives one of theses broadcast messages, it may determine that it has gone beyond the point of the other node's boundary; thus, it should stop processing. ON the other hand, 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.

Non-Blocking 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 evenly balance the work load on each node so 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 does not hurt to continue processing events beyond the horizon. It might pay to be optimistic and hope that he 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 force each node to send a special message to a central manager when it thinks that 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 for 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, 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, an interrupt is simultaneously fired on each node and a flag is set telling us that all nodes have crossed the event horizon.

While 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.

Local Rollback

One further improvement can be made to the Breathing Time Buckets algorithm. Events that are 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 queueing network. Supporting this capability involves more overhead, but it may be essential for a large class of simulation applications.

INTERACTIVE SPEEDES

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

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.). Humans are then allowed to interact with the simulation based on information that was previously released.

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, then 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, further steps are required to merge them into a single globally sorted list. The steps to do this on a parallel computer are illustrated in Fig. 9 and are as follows:

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

$$T\ merge = (m/N)\ \log_2 N$$

It would appear that parallel simulations require an additional amount of work to send globally sorted event information out 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 globally processed in 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 them 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:

$$T\ sort = (m/N)\ [\log_2(m) - \log_2 N]$$

When combining Tmerge and Tsort, the superlinear speedup is exactly cancelled. 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.

Simulation Tie Advancement Rate (STAR) Control

If humans are allowed to interact with a simulation while it is in progress, then 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 tougher 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, as illustrated in the graph of Fig. 10. While 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 ten, using a different approach, then slowing it down to real time would be easier than when using algorithm with a STAR equal to two. 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 the 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 in time 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 event sin 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 happen. 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, then there should be no pauses to throttle the simulation. If the simulation operates in cycles, then 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).

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 message 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 COMMAND section) with the parallel simulation based on the output that has been received (see the QUERY and MONITOR sections). 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, as illustrated in Fig. 11. This is one of the major drawbacks of conservative algorithms.

External Modules

Referring now to Fig. 12, interactive SPEEDES accommodates external interactions by using a host program 30 to service communications between the central parallel simulation 32 and the outside world. The host program allows external modules 34 to establish connections to the central parallel simulation using, for example, UNIX Berkeley Sockets.

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

QUERY

A very useful capability interactive SPEEDES supports is the ability to QUERY the stat of simulation object while the simulation is in progress. The simulation can be viewed as a large database of object 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.

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.

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.

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. The 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 on purpose or accidentally), this blocking mechanism is automatically removed.

While the invention has been described in detail with specific reference to preferred embodiments thereof, it is understood that variations and modifications thereof may be made without departing from the true spirit and scope of the invention.

SYNCHRONOUS PARALLEL SYSTEM FOR EMULATION
AND DISCRETE EVENT SIMULATION

ABSTRACT OF THE INVENTION

A synchronous parallel system for emulation and discrete event simulation having parallel nodes responds to received messages at each node by generating event objects having individual time stamps, stores only the changes to the state variables of the simulation object attributable to the event object and produces corresponding messages. The system refrains from transmitting the messages and changing the state variables while it determines whether the changes are superseded, and then stores the unchanged state variables in the event object for later restoral to the simulation object if called for. This determination preferably includes sensing the time stamp of each new event object and determining which the new event object has the earliest time stamp as the local event horizon, determining the earliest local event horizon of the nodes as the global event horizon, and ignoring events whose time stamps are less than the global event horizon. Host processing between the system and external terminals enables such a terminal to query, monitor, commmand or participate with a simulation object during the simulation process.

APPENDIX A

The following appendix is the listing of the C-language computer code used to implement the invention using the breathing time buckets protocol.

```
// speedes_evtq.H header file

#ifndef speedes_evtq_object
#define speedes_evtq_object

#include "evtq.H"
#include "cycle.H"
```

```
                    speedes_evtq object
```

```
class C_SPEEDES_EVTQ : public C_EVTQ {

  private:

  protected:

   C_CYCLE *cycle;         // cycle synchronizing object
   C_QUEUE *qext;          // queue of externally generated events

  public:

   C_SPEEDES_EVTQ();
    virtual void temp_process();
    virtual void perm_process();
    virtual void find_gvt();

};


#endif
```

```
// speedes_evtq.C method file

#include <stdio.h>
#include "Cros.H"
#include "speedes_evtq.H"

#include "defunc.H"

#define OPTIMIZE
#define INFINITY 1.0e20
```

```
          C_SPEEDES_EVTQ : construct an event queue object
```

```
C_SPEEDES_EVTQ::C_SPEEDES_EVTQ() {

  printf("SPEEDES_EVTQ created\n");

  cycle = new C_CYCLE();
  qext = new C_QUEUE();

}
```

```
               temp_process - process events phase 1
```

```
void C_SPEEDES_EVTQ::temp_process() {
  int i;
  int nleft;
  double tblock;
  double tmin;
  double tearly;
  double tevent;
  C_EVENT *event;
  C_EVENT *ext_event;

  cycle->start();     // this starts the cycle management

  n_temp = 0;
  localt = tend;
  tearly = gvt + 0.99*minstep;          // all events with times less than tearly can be
processed conservatively

//...... if there is a minimum time step, use that information to run faster

  event = (C_EVENT *)top;
  for (i=0; i<n_items; i++) {
    tevent = event->get_time_tag();
    if (tevent >= tearly) break;

    if (!event->get_processed()) {
      event->temp_process();
      event->exchange();
      event->calculate_tmin();
      event->set_processed();
    }

//...... update tmin

    tmin = event->get_tmin();
    if (tmin < localt) localt = tmin;

    n_temp++;
    event = (C_EVENT *)event->get_link();

  }

//...... now do the rest of the events optimistically

  if (!cycle->check(localt, tevent)) {

    nleft = n_items - n_temp;

    for (i=0; i<nleft; i++) {

      if (!event->get_processed()) {

//...... cycle->check() manages asynchronous broadcasts and non-blocking syncs.

        if (cycle->check(localt, event->get_time_tag())) break;

        event->temp_process();
        event->exchange();
        event->calculate_tmin();
        event->set_processed();

      }

      tmin = event->get_tmin();
      if (tmin < localt) localt = tmin;

//...... handle externally generated events

      ext_event = external_event(tearly);
      if (ext_event != NULL) {
        qext->push_bot(ext_event);
        if (ext_event->get_time_tag() < localt) localt = ext_event->get_time_tag();
        if (cycle->check(localt, event->get_time_tag())) break;
      }

      n_temp++;
      event = (C_EVENT *)event->get_link();

    }

  }

//...... check if blocking messages are expected to arrive

  if (objects->get_blocking()) {
    objects->update_block();
    tblock = objects->get_tblock();
    cycle->min_tmin(tblock);
  }

  cycle->stop();                // stop the current cycle

}
```

find_gvt - find the minimum global time for safe processing

```
void C_SPEEDES_EVTQ::find_gvt() {

  gvt = cycle->get_nextgvt();          // the event horizon is globally determined

}
```

perm_process - process events phase 2

```
void C_SPEEDES_EVTQ::perm_process() {
  int size;
  int i,len;
  char *message;
  C_EVENT *event;
  C_HOLDER *holder;
  C_HEADER *header;

//...... loop over all of the events that can be processed

  n_perm = 0;
  n_roll = 0;

  multirouter->reset();

  while (n_items) {

//...... pull out processed events, processes them further (garbage collection) and
collect their generated messages

    event = (C_EVENT *)pop_top();
    if (gvt < event->get_time_tag()) {
      push_top(event);
     break;
    }

    event->perm_process();
    event->sendmess(q1nd);              // this only collects messages in the multirout
object

    qproc->push_bot(event);
    if (event->get_time_tag() < tend) n_perm++;

  }

//...... send and receive messages synchronously (crystal router on hypercubes)

  multirouter->multirout();

//...... delete the already processed event using free lists

  while (qproc->length()) {
    event = (C_EVENT *)qproc->pop_top();
    evtype->delete_event(event);
  }

//...... get the incoming messages and turn them into future events

  while((message = multirouter->getmess(size)) != NULL) {
   header = (C_HEADER *)message;
   if (header->ext) {
     host_user->send_message((C_EM_HEADER *)message);        //      external
message
   }else{
     event = (C_EVENT *)evtype->message_event(message);      //      internal
message turns into an event
     qproc->push_bot(event);
   }
  }

//...... get messages that were generated locally on my node and turn them into
events

  len = q1nd->length();
  for (i=0; i<len; i++) {
   holder = (C_HOLDER *)q1nd->pop_top();
   message = holder->get_buff();
   header = (C_HEADER *)message;
   if (header->ext) {
     host_user->send_message((C_EM_HEADER *)message);
     evtype->delete_message(holder);
   }else{
     event = (C_EVENT *)evtype->message_event(message);
     qproc->push_bot(event);
     evtype->delete_message(holder);
   }
  }

//...... sort all of the new events

  qproc->concat(qext);
  qproc->sort();

//...... attach these new events to their appropriate objects

  len = qproc->length();
  event = (C_EVENT *)qproc->ptop();
```

```
 for (i=0; i<len; i++) {
   event->attach_object();
   n_roll += event->roll_back();        // rollback objects with events in their past
   event = (C_EVENT *)event->get_link();
 }

//...... merge these new events into the event queue

 merge(qproc);

}
```

NASA Case No. NPO 18414-1-CU

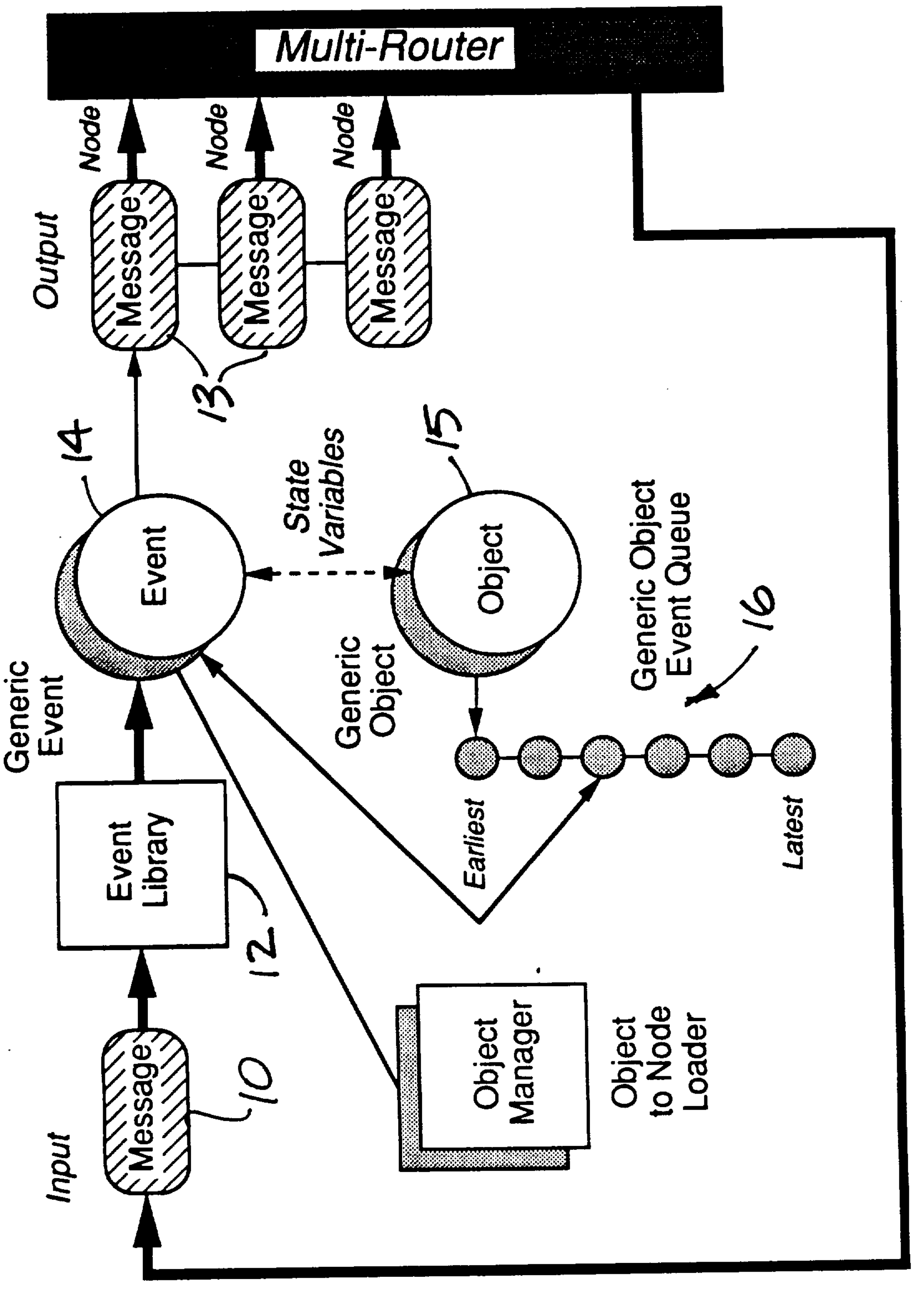


FIG. 1

NASA Case No. NPO 18414-1-CU

of

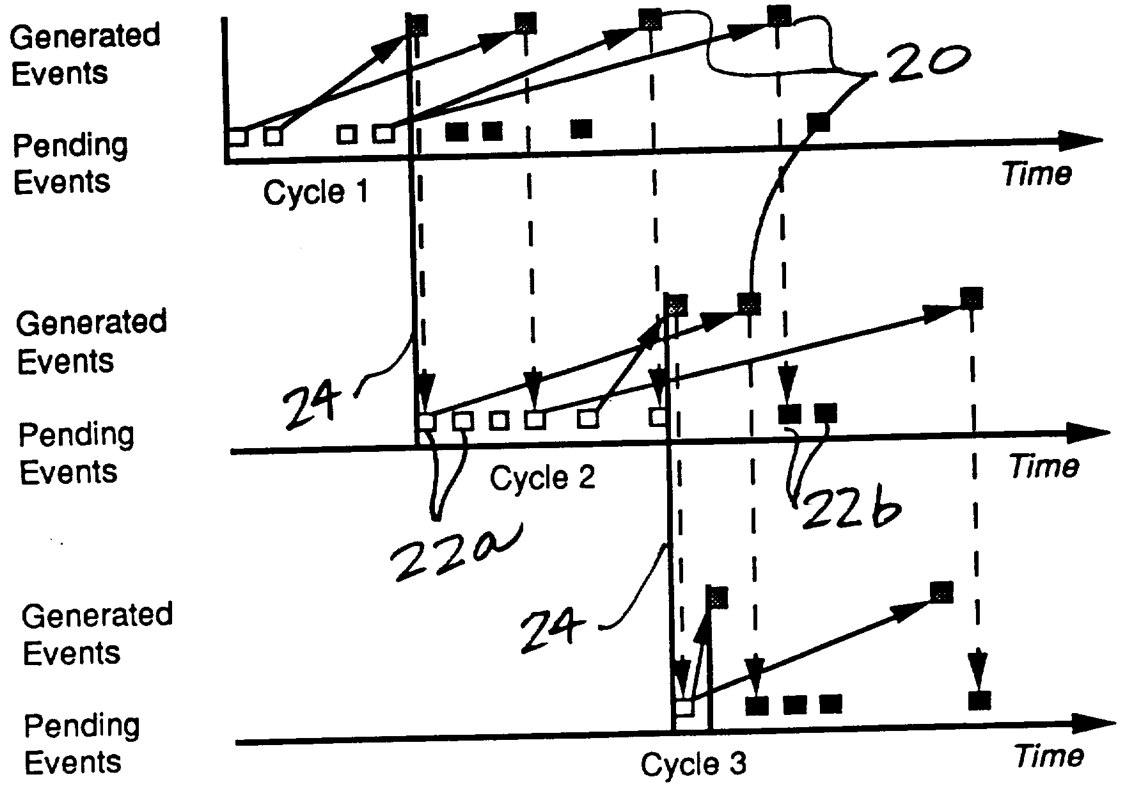


FIG. 2

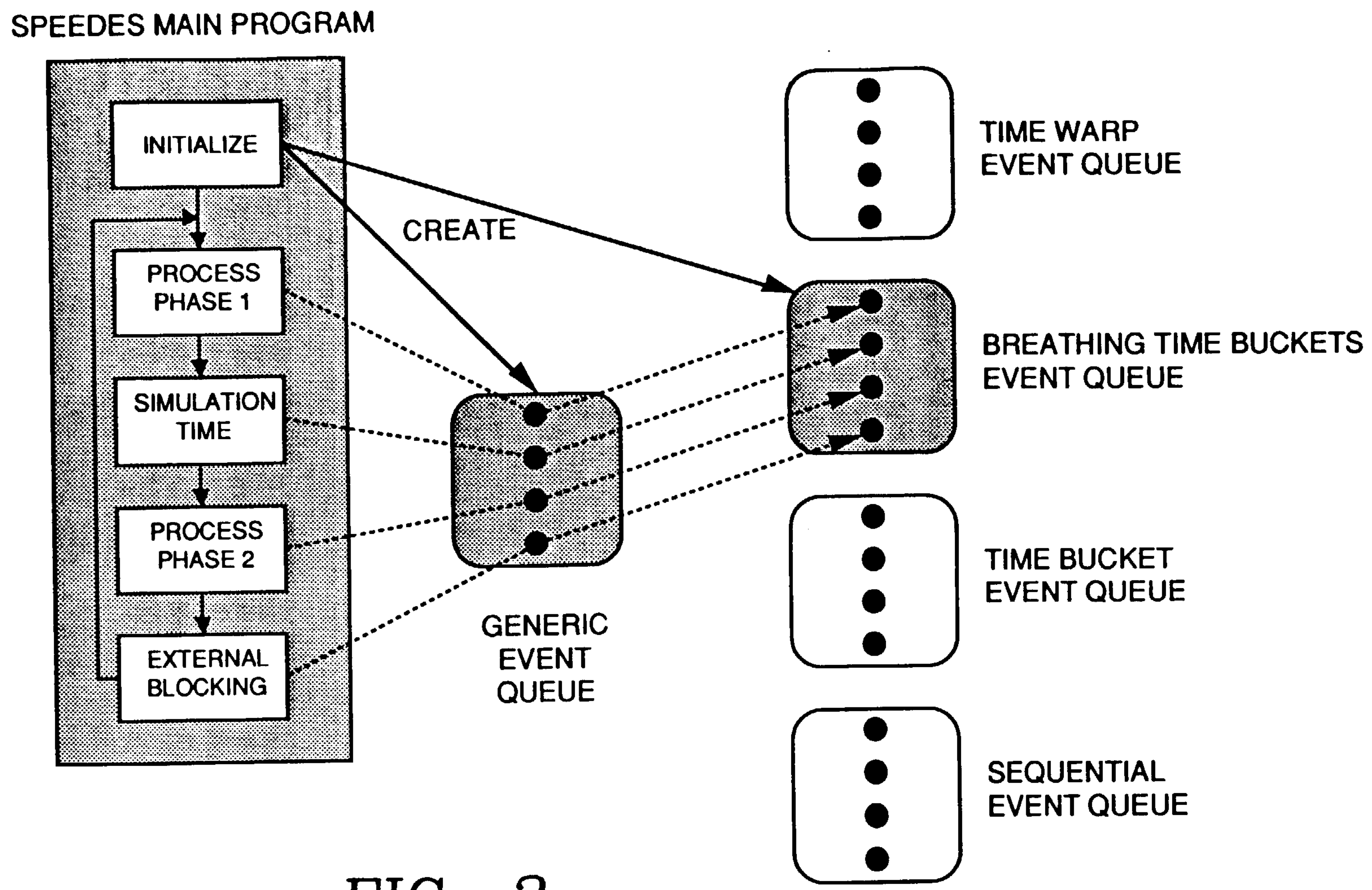

SPEEDES MAIN PROGRAM
INITIALIZE
PROCESS PHASE 1
SIMULATION TIME
PROCESS PHASE 2
EXTERNAL BLOCKING
CREATE
GENERIC EVENT QUEUE
TIME WARP EVENT QUEUE
BREATHING TIME BUCKETS EVENT QUEUE
TIME BUCKET EVENT QUEUE
SEQUENTIAL EVENT QUEUE

FIG. 3

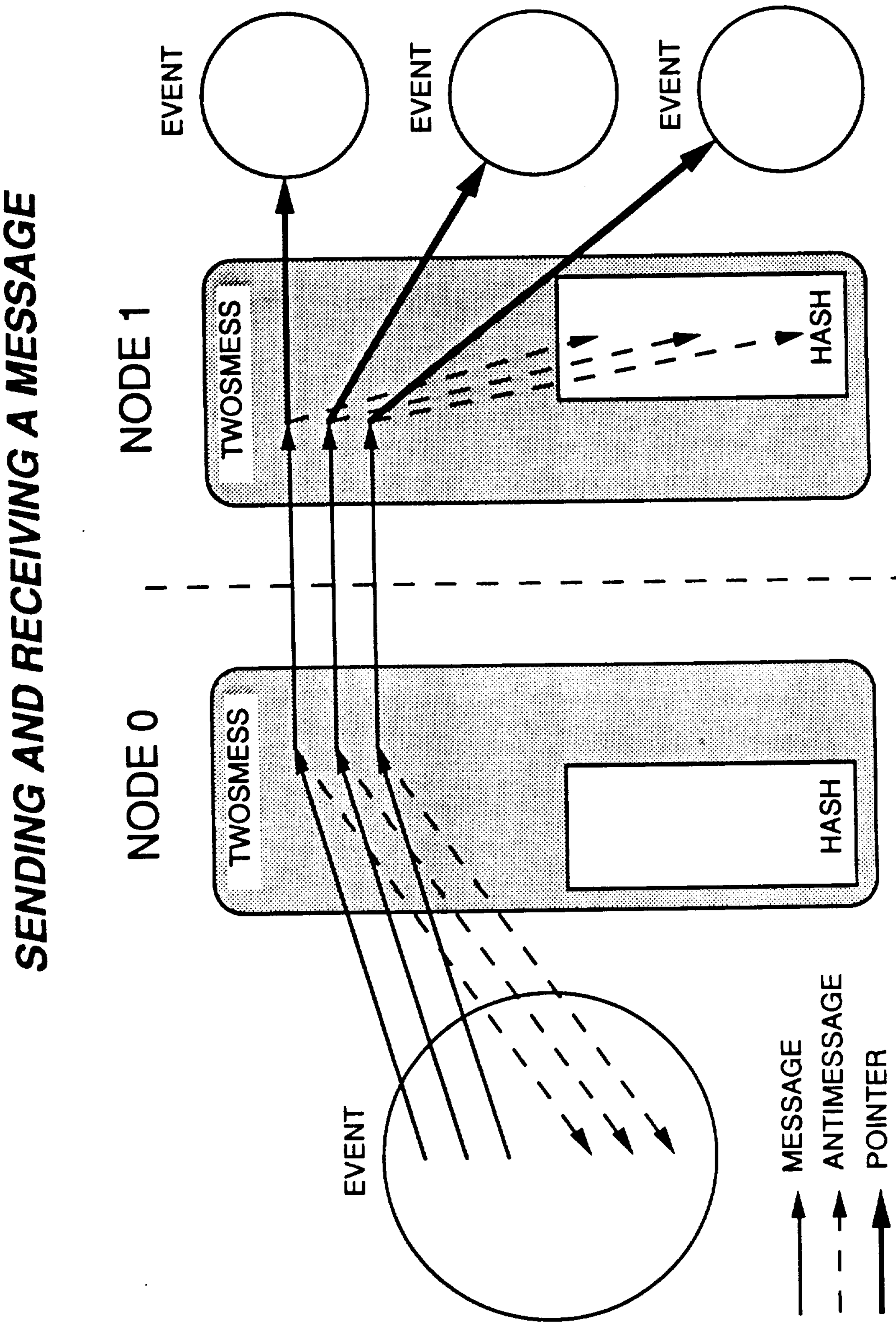

SENDING AND RECEIVING A MESSAGE
NODE 0
NODE 1
TWOSMESS
HASH
TWOSMESS
HASH
EVENT
EVENT
EVENT
EVENT
MESSAGE
ANTIMESSAGE
POINTER

FIG. 4

NASA Case No. NPO 18414-1-CU

SENDING AND RECEIVING AN ANTIMESSAGE

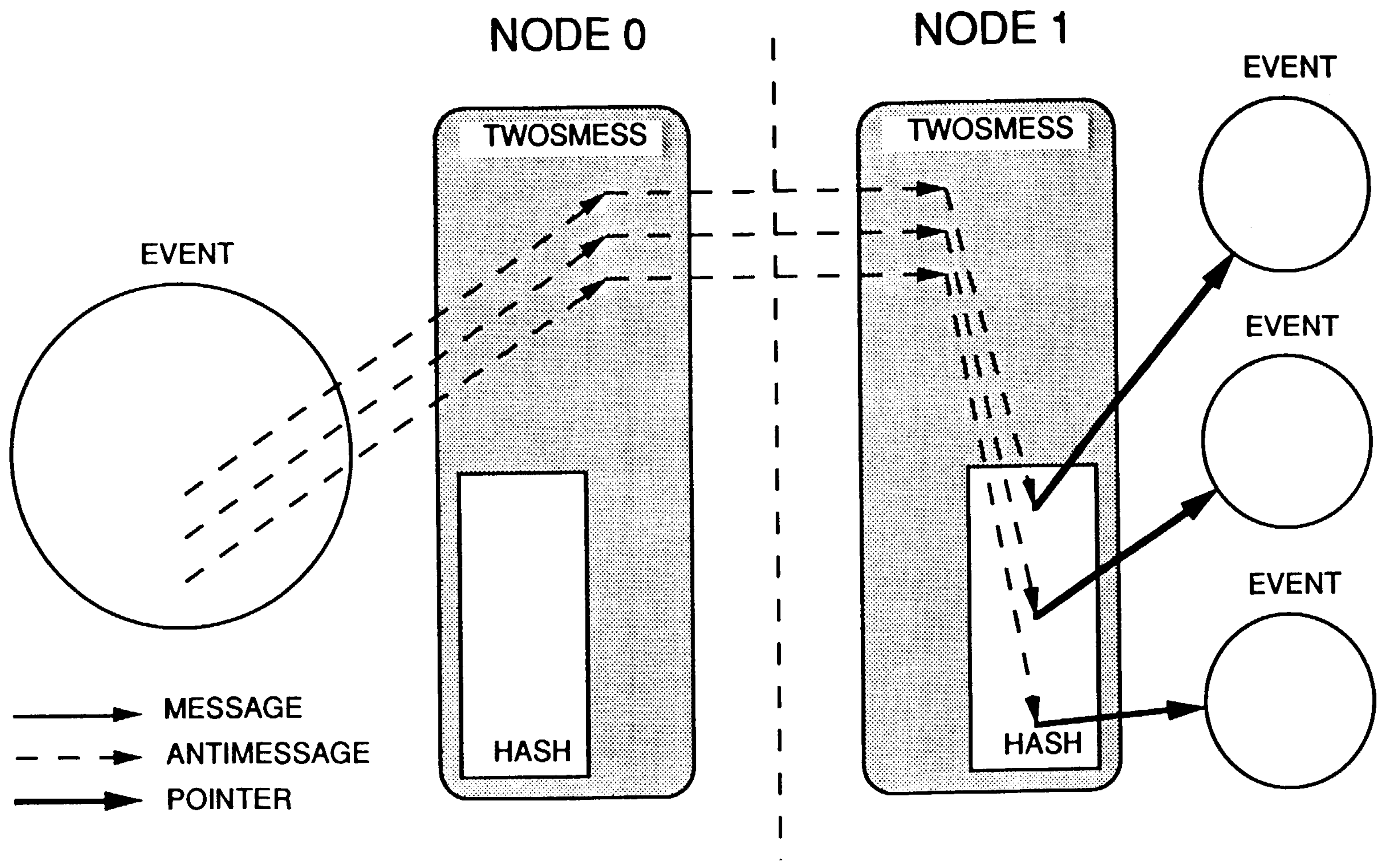


FIG. 5

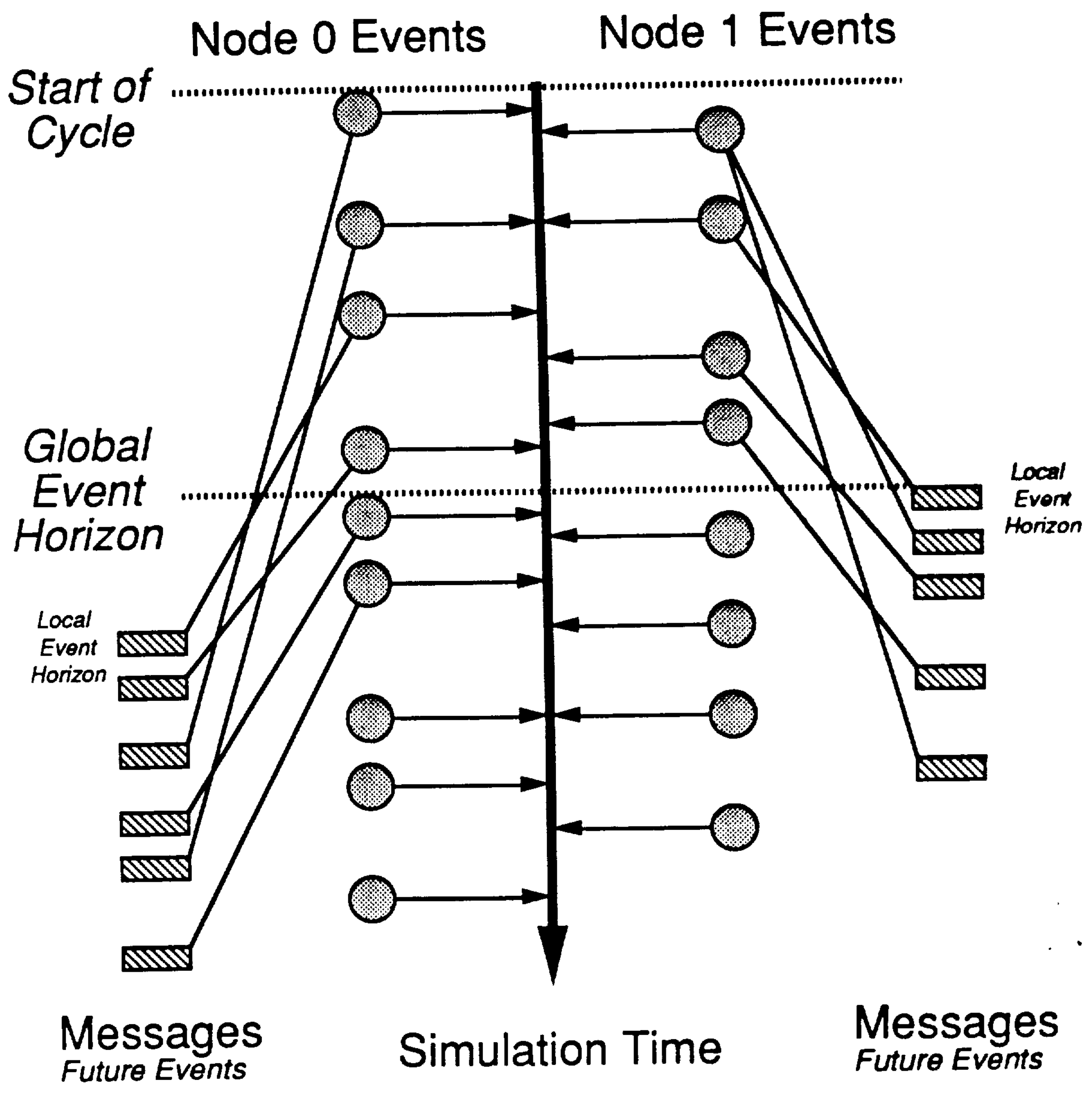
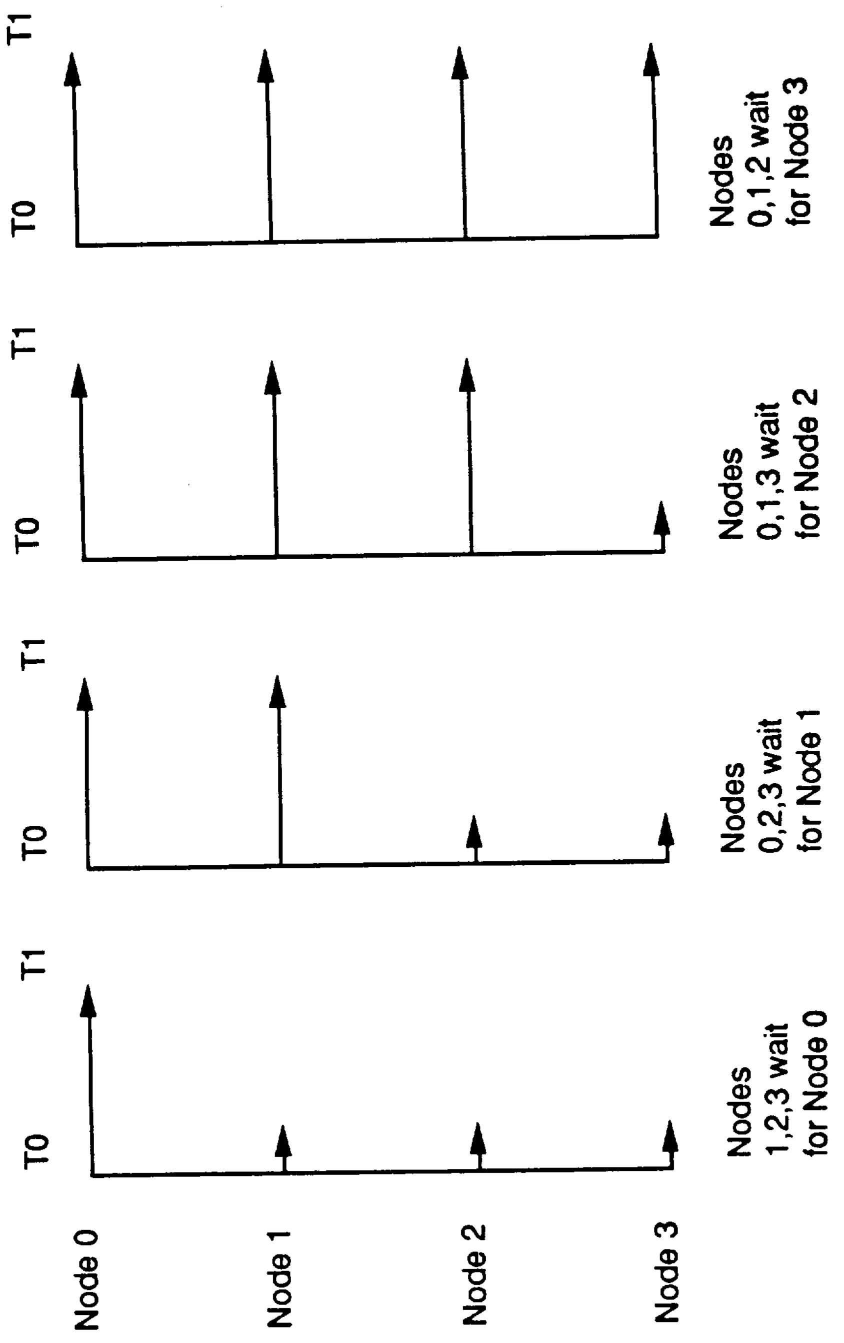

Node 0 Events
Node 1 Events
Start of Cycle
Global Event Horizon
Local Event Horizon
Local Event Horizon
Messages
Future Events
Simulation Time
Messages
Future Events

FIG. 6

NASA Case No. NPO 18414-1-CU

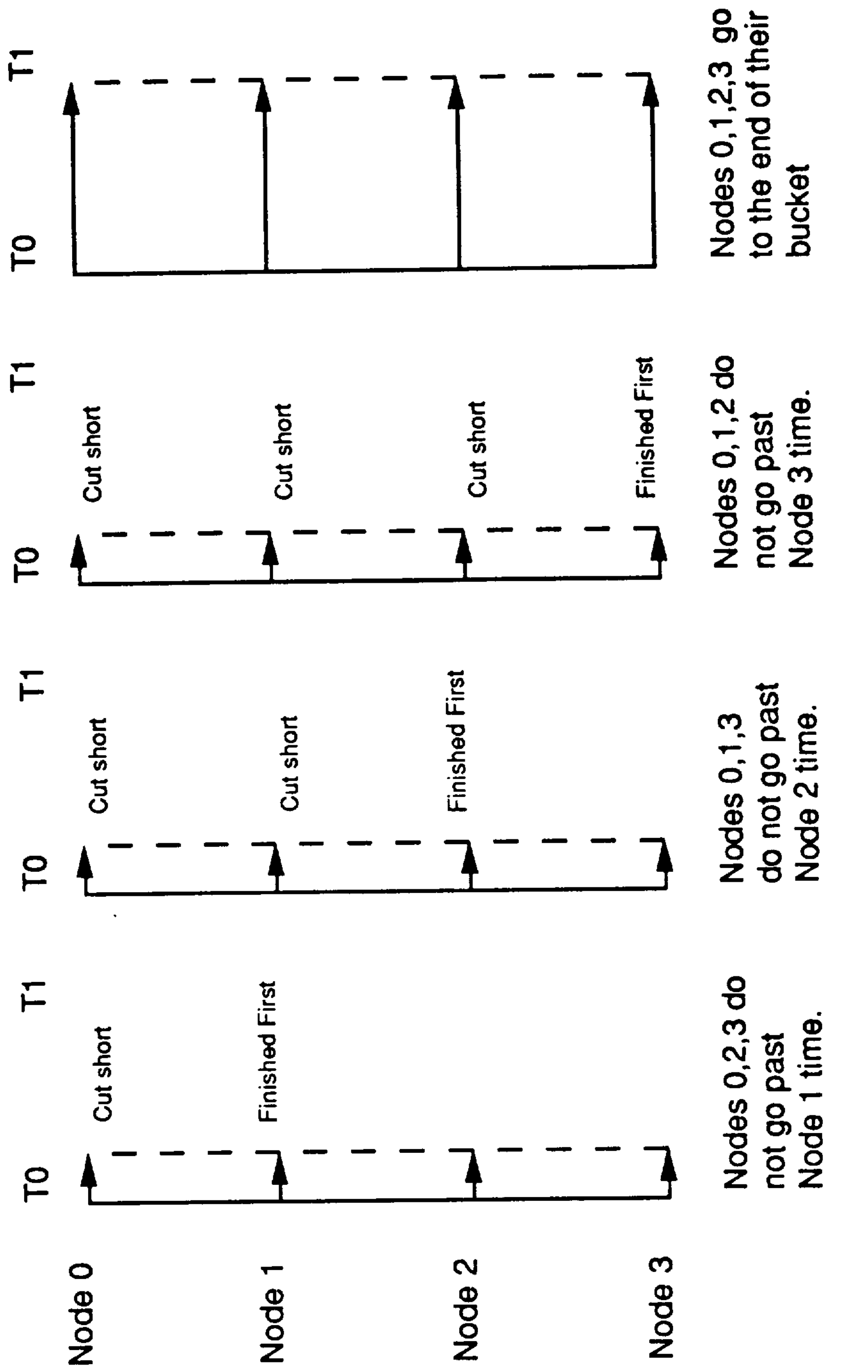


Node 0
Node 1
Node 2
Node 3
T0
T1
Nodes 1,2,3 wait for Node 0
Nodes 0,2,3 wait for Node 1
Nodes 0,1,3 wait for Node 2
Nodes 0,1,2 wait for Node 3

FIG. 7

NASA Case No. NPO 18414-1-CU

of



Node 0
Node 1
Node 2
Node 3
T0
T1
Cut short
Finished First
Nodes 0,2,3 do not go past Node 1 time.
T0
T1
Cut short
Cut short
Finished First
Nodes 0,1,3 do not go past Node 2 time.
T0
T1
Cut short
Cut short
Cut short
Finished First
Nodes 0,1,2 do not go past Node 3 time.
T0
T1
Nodes 0,1,2,3 go to the end of their bucket

FIG. 8

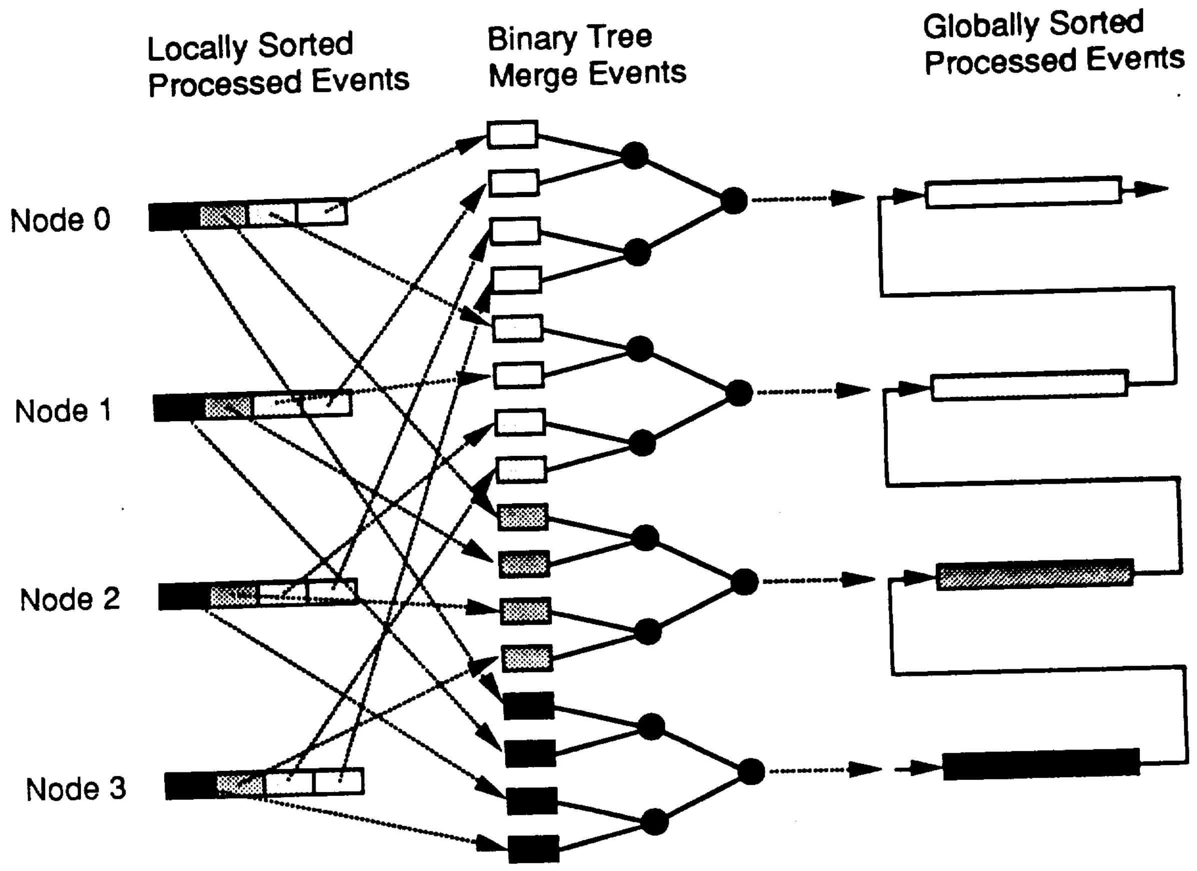

Locally Sorted Processed Events
Binary Tree Merge Events
Globally Sorted Processed Events
Node 0
Node 1
Node 2
Node 3

FIG. 9

NASA Case No. NPO 18414-1-CU

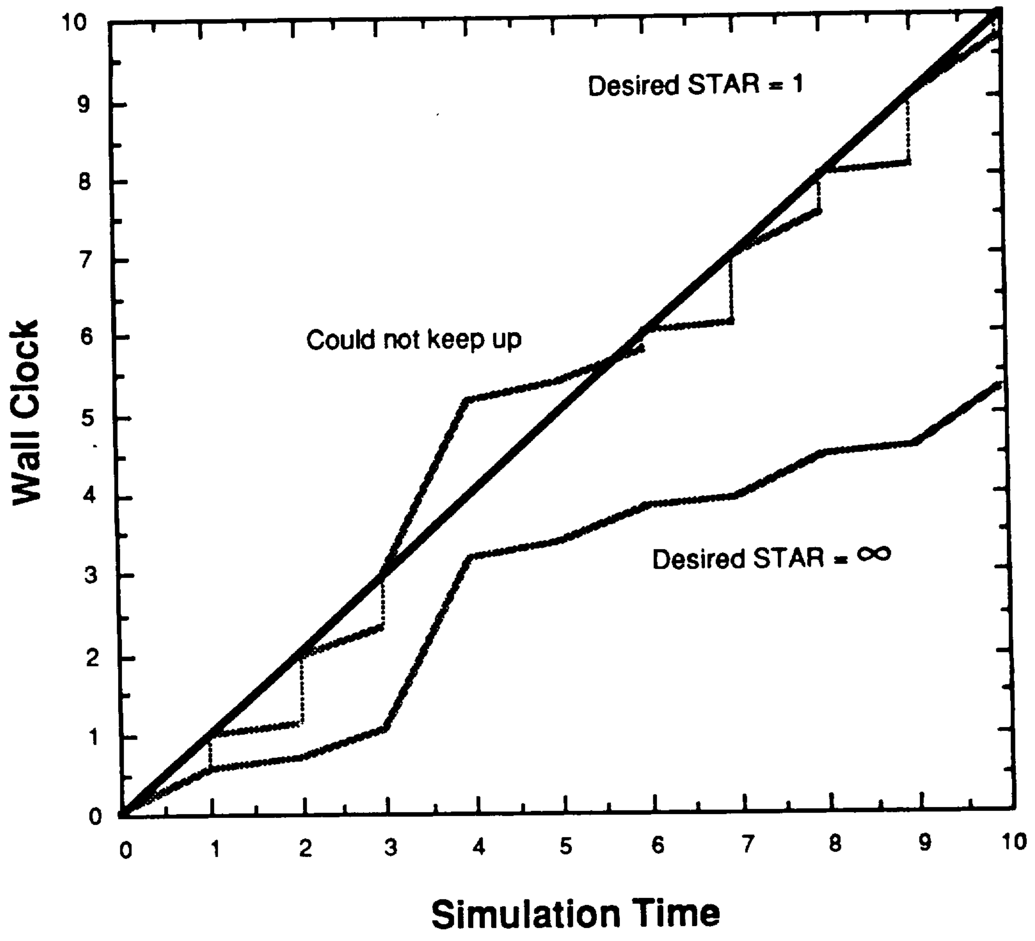


FIG. 10

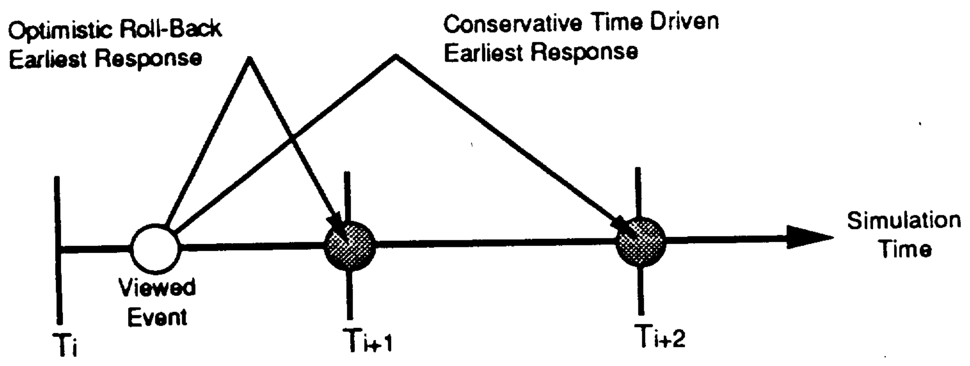

Optimistic Roll-Back Earliest Response
Conservative Time Driven Earliest Response
Simulation Time
Viewed Event
Ti
Ti+1
Ti+2

FIG. 11

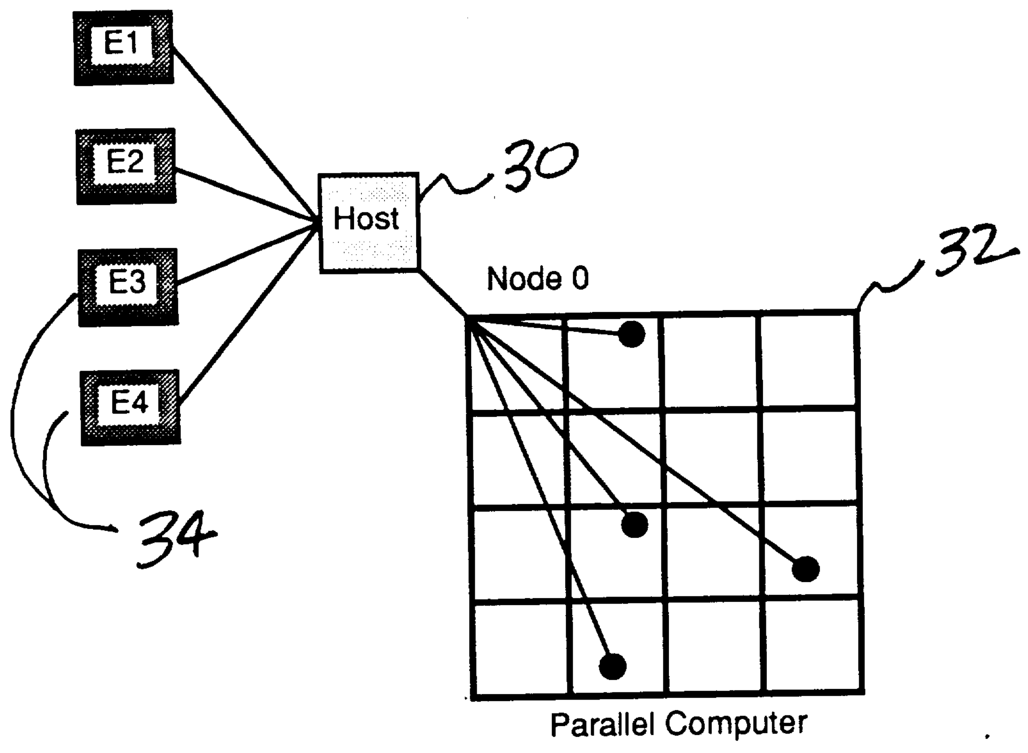

E1
E2
E3
E4
Host
30
Node 0
32
34
Parallel Computer

FIG. 12