

Simulating Spatially Explicit Problems on High Performance Architectures¹

Ewa Deelman²

Information Sciences Institute, University of Southern California, Marina Del Rey, California 90292
E-mail: deelman@isi.edu

and

Boleslaw K. Szymanski

Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180
E-mail: szymansk@cs.rpi.edu

Received July 24, 2000; revised February 1, 2001; accepted March 9, 2001

This paper addresses issues of implementation and performance optimization of simulations designed to model spatially explicit problems with the use of parallel discrete event simulation. A simulation system is presented that uses the optimistic protocol and runs on a distributed memory machine—the IBM SP. The efficiency of parallel discrete event simulations that use the optimistic protocol is strongly dependent on the overhead incurred by rollbacks. This paper introduces a novel approach to rollback processing which limits the number of events rolled back as a result of a straggler or antimessage. The method, called Breadth-First Rollback (BFR), is suitable for spatially explicit problems where the space is discretized and distributed among processes and simulation objects move freely in the space. The BFR uses incremental state saving, allowing the recovery of causal relationships between events during rollback. These relationships are then used to determine which events need to be rolled back. This paper presents an application of BFR to the simulation of Lyme disease. Our results demonstrate an almost linear speedup—a dramatic improvement over the traditional approach to rollback processing. Additionally, BFR is used as a basis of a dynamic load balancing algorithm that migrates load between the simulation processing. Additionally, BFR is used as a basis of a dynamic load balancing algorithm that migrates load between the simulation processes. A brief outline of the algorithm and its potential performance are presented. © 2002 Elsevier Science

¹ Parts of this work were published in the Proceedings of Parallel and Distributed Simulation 1997 and 1998.

² This work was performed while at Rensselaer Polytechnic Institute



1. INTRODUCTION

The research presented in this paper concentrates on the simulation of a class of problems known as spatially explicit problems that have a structure which naturally contains spatial information. Among such problems are battlefield simulations [18, 37], the simulation of personal communications services (PCS) [3, 7, 16, 23], etc. In general, spatially explicit applications consist of a multidimensional space, which is discretized into a multidimensional lattice. Each lattice node can contain information which characterizes a give area. There are also mobile objects present in the space. There are two general classes of events in the system: *local* and *nonlocal*. A local event affects only the state of one lattice node. A nonlocal event, for example, the Move Event, which moves an object from one location to the next, affects at least two nodes of the lattice.

The application that motivated this work is the simulation of Lyme disease. The space, in this case, is two-dimensional, and it is discretized into a two-dimensional lattice. The most important characteristics of the simulation are the mobile objects moving freely in space (mice and deer) and the stationary objects present at the lattice nodes (ticks). The two main groups of events are: (i) local to a node (such as tick bites, death, etc.) and (ii) nonlocal (such as a move from one node to another—*Move event*). Lyme disease is prevalent in the Northeastern United States [4, 24]. People can acquire the disease by coming in contact with an infected tick. Since the ticks are practically immobile, the spread of the disease is driven by their mobile hosts, such as mice and deer. Even though the most commonly known cases of Lyme disease are reported in humans, the main infection cycle consist of ticks and mice (Fig. 1). If an infected tick bites a mouse, the animal becomes infected. The disease can also be transmitted from an infected mouse to an uninfected, feeding tick.

The mice and deer are modeled as individuals, and ticks, because of their sheer number (as many as 1200 larvae/400m² [25]) are treated as “background.” The space is discretized into nodes of size 20 × 20m², which represents the size of the

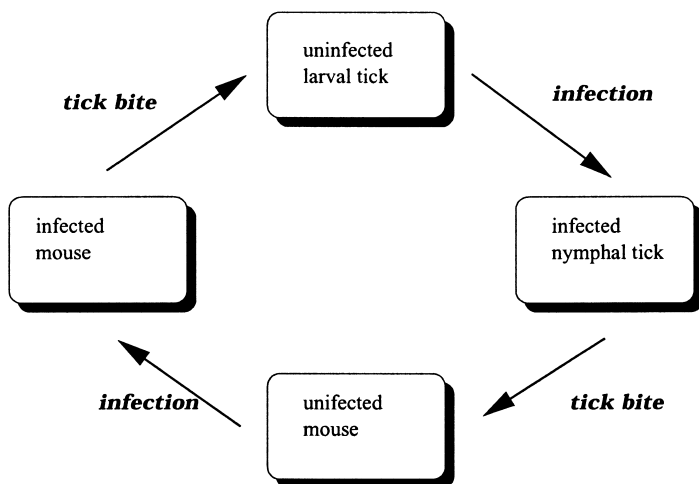


FIG. 1. The cycle of Lyme disease.

home range for a mouse. Each node may contain any number of ticks in various stages of development and various infection status. Mice and deer can move around in space in search of empty nesting sites. The initiation of such a search is described by the *Disperse Event* and the moves by the *Move Event*. Mice and deer can die (*Kill Event*) if they cannot find a nesting site or by other natural causes, such as old age, attacks by predators, and disease. Mice and deer can be bitten by ticks (*Tick Bite*) or have ticks drop off (*Tick Drop*). From the above list of events, only the *Move Event* is nonlocal.

Because of the discrete nature of individual-based models such as the one described above, they lend themselves well to discrete event simulation techniques. Discrete event simulation (DES) is a technique which has been used in modeling communication systems [3, 7, 16, 23], electronic circuits [10], battlefield scenarios [18, 37], etc. In DES the physical system is modeled by a single process consisting of the process state, the clock, and the event queue. Events are kept in a priority queue, where the event with the lower scheduled time has higher priority. The simulation is started by introducing some initial events, which are inserted into the event queue. The simulation progresses as events are first removed from the event queue and then processed. The simulation clock is advanced to the time of the event's occurrence. When an event is processed, the system changes from one state to another.

A population model often envisions a large number of individuals inhabiting a large area. The size of the problem might make sequential simulation too slow. In such cases, an attractive alternative is to use a parallel or a distributed machine. Such a machine possesses the sum of memory and cache on all processors to fulfill the memory requirements, and the combined processing power to speed up the simulation. However, moving to a parallel or distributed platform increases the complexity of the simulation engine. In sequential DES the physical system is represented by a single logical process (LP), whereas parallel discrete event simulation (PDES) [14] brings with it multiple logical processes, each modeling a physical process (PP) in the physical system. The LPs work together to simulate the underlying physical system by sending event messages between each other. The computational challenge is to keep the simulation consistent among the processes to make sure that casually related events are processed in the correct order.

This paper describes how spatially explicit problems can be modeled using PDES and the performance of the simulation optimized by using a novel approach to rollback and dynamic load balancing. The paper is organized as follows. Section 2 introduces PDES concepts and related work in rollback cost reduction. Section 3 describes the traditional approach to simulating spatially explicit problems. This approach will be used as the basis of comparison for our new algorithm, which we called the breadth first rollback (BFR) algorithm and is described in Section 4. This new algorithm is designed to minimize the impact of rollbacks. Incremental state saving is used to expose dependencies between events. Section 4 also explains the challenges of BFR and shows examples of how the algorithm works. As in all parallel computation, the balance of the relative work load of the processors is of importance. Section 5 shows how BFR can be used as the basis for a dynamic load balancing algorithm. Conclusions are presented in Section 6.

2. BACKGROUND AND RELATED WORK

In PDES each LP is composed of state variables, a priority event queue, and a clock, which keeps track of the local virtual time (lvt) of the process. The LPs process events asynchronously according to the timestamps of the events. The clock and the state variables are updated during the processing of an event. LPs can communicate with each other by sending event messages. When an LP receives an event message with a timestamp lower than its lvt, a causality error occurs, because it is possible that the incoming event will change the state of the LP in such a way that the state of the LP at the current lvt will no longer be correct. The following protocols have been developed to deal with causality errors.

The conservative protocol proposed by Chandy and Misra [9] prevents causality errors from occurring by allowing the LPs to process events only when it is absolutely certain that there is no event in the system with a timestamp smaller than the timestamp of the event about to be processed. The conservative approach derives its performance from *lookahead*, the ability to determine when the next event will occur. The lookahead is application dependent and can sometimes be hard to determine or even vary throughout the simulation. The optimistic protocol, on the other hand, allows causality errors to occur and recovers from them. Time Warp (TW), introduced by Jefferson [19], is the best known protocol of the above type. The processing of events is done optimistically in the hopes that no causality errors occur. When an event with a timestamp smaller than the current lvt is received (such a message is known also as a *straggler*), the LP is rolled back to the state just prior to the one before the timestamp of the straggler. The event is then processed and the computation is restarted. If messages were sent out between the logical timestamp of a straggler and the current lvt, corresponding *antimessages* have to be sent to cancel the original messages released. Upon receiving an antimessage, the LP checks if the corresponding event has been processed; if it has, a rollback occurs; otherwise the event is annihilated.

The optimistic approach, however, requires larger amounts of memory than the conservative approach. To support rollback, the state of the LP needs to be saved after each event is processed. In order to reduce the amount of memory used, the state can be saved less frequently [21, 26, 27]. The side effect of smaller checkpoint frequency is the increased cost of performing a rollback. When the state of the LP is large, saving the entire state can be too costly. If additionally the occurrence of an event changes only a small subset of state variables, then the incremental state saving, developed in SPEEDES [35] can be successful. In this state saving method, the part of the state that is affected by an event is saved in the event structure. Upon rollback the affected events are undone and the modified state variables are restored.

To reclaim memory no longer needed, the LPs calculate the *global virtual time* (gvt). The gvt is the minimum virtual (simulation) time of all the LPs and the timestamps of all messages present in the system. By definition, the computation cannot be rolled back beyond the gvt, since there are no events with a timestamp smaller than the gvt in the system. The gvt calculation is an important part of reducing the simulation's memory usage, because any states and messages that

happened before the gvt will never be referred to and can be therefore removed from memory. The process of memory reclaiming is known as *fossil collection*.

Obviously performance is also an issue in TW. A major source of performance degradation is the cost of rollbacks. The time spent on processing events that will later be rolled back is wasted work. So is the time spent restoring the state (especially in incremental state saving). Two most general categories of rollback cost reduction can be classified as follows: protocol modification to improve TW performance and problem partitioning that defines allocation of parts of the physical system to LPs, and the associated issue of mapping of LPs to processors.

2.1. Protocol Innovations

One way to improve performance is to diminish the cost of rollbacks by lessening the impact that a rollback has at one LP on the LPs communication with it.

Lazy cancellation [20] can be used to reduce the number of antimessages sent. When a rollback occurs, the LP does not automatically send antimessages corresponding to the positive messages that have been sent after the time to which the computation is rolled back; rather it recomputes the states from the rollback time. If the processing of events causes the same messages to be sent, then there is no need to cancel them, thus saving the cost of canceling “good” messages. If, however, there are changes, an antimessage has to be sent. The idea is that if during the recalculation phase after rollback, the message would be sent anyway, then it is better not to send an antimessage. This antimessage could cause the receiving LP to roll back unnecessarily. Lazy cancellation involves comparing messages, which is not always easy. Additionally, lazy cancellation can delay the notification of a rollback and thus increase the rollback’s cost.

Lazy reevaluation [14] has been used to determine if a straggler or antimessage had any effect on the state of the simulation. If, after processing the straggler or canceling an event, the state of the simulation remains the same as before, then there is no need to re-execute any events from the time of the rollback to the current time. The problem with this approach is that it is hard to compare the state vectors in order to determine if the state has changed. It is also not applicable to the protocols using incremental state saving.

Both lazy cancellation and lazy reevaluation assume that a rollback at one LP is not likely to affect others. Conversely, the Wolf calls protocol [22] was designed to minimize the impact an LP’s rollback has on others by assuming that a rollback at an LP will most likely trigger rollbacks of other processes. When an LP rolls back, it checks which are the LPs likely to be affected. It then sends a message to these LPs letting them know that a causality error has occurred. The big disadvantage here is that correct computation might be held back unnecessarily.

Limiting the optimism in a simulation (throttling) can also reduce the chance of an expensive rollback. Window-based algorithms [29] set a limit as to how far ahead into the future the LPs can proceed without synchronizing. This prevents LPs from getting far ahead of others and incurring frequent rollbacks. Unfortunately window-based algorithms do not distinguish between useful and erroneous work. The breathing time warp [34] has a dual nature, represented by two phases: time

warp and breathing time buckets. In the first phase, the simulation progresses as in Time Warp. The length of this phase is determined to be the number of events generated in that phase. When the predetermined number of events is reached, the simulation switches to the next phase-breathing time buckets. In this phase events are processed but their messages are not released. The *event horizon* determines the length of the second phase. This time is the timestamp of the earliest new event generated in the current cycle. After both phases are complete, the messages are released and the gvt is calculated.

To improve the performance of throttling algorithms, some researchers [31, 32] have designed hardware to efficiently propagate state information. This hardware allows the simulator to have a view of the global state and allows it to adapt to the progress the LPs are making.

2.2. Problem Decomposition

Another way to improve the simulation's performance is to change the decomposition of the problem. The Local Time Warp (LTW) [28] approach combines two simulation protocols by using the optimistic simulation between LPs belonging to the same cluster and by maintaining a conservative protocol between clusters. LTW minimizes the impact of any rollback to the LPs in a given cluster.

Clustered Time Warp (CTW) [1, 2] takes the opposite view. It uses conservative synchronization within the clusters and an optimistic protocol between them. The reason given for this method is that since LPs in a cluster share the same memory space, their tight synchronization can be performed efficiently. Two algorithms for rollback are presented: clustered and local. In the first case, when a rollback reaches a cluster, all the LPs in that cluster are rolled back. This way the memory usage is good because events that are present in input queues, and that were scheduled after the time of the rollback, can be removed. In the local algorithm, only the affected LPs are rolled back. Restricting the rollback speeds up the computation, but increases the size of memory needed, because entire input queues have to be kept.

The Multi-Cluster Simulator [30], in which digital circuits are modeled, takes a bit of a different look at clustering. First, the cluster is not composed of a set of LPs; rather, it consists of one LP composed of a set of logical gates. These LPs (clusters) are then assigned to a simulation process. This approach is taken because assigning an LP to a single logical gate would result in high scheduling overhead.

3. THE TRADITIONAL APPROACH FOR SPATIALLY EXPLICIT PROBLEMS

3.1. Partitioning

In our simulation, the problem space is discretized into a two-dimensional lattice, where the size of each lattice node is a square whose dimension is application specific. Similar discretization is used, for example, in personal communication services [7], where the two-dimensional space is discretized by representing the network as hexagonal or square cells. In these simulations, each cell is modeled by an LP. In

some simulations, it would be prohibitively expensive to assign one LP to each spatial entity, so then the entities are “clustered” into a single LP, similarly as in circuit simulations. In our work we “cluster” the LPs by using a strip-wise decomposition of the lattice (where the space is partitioned in one dimension) and assigning the resulting partitions (continuous columns of the lattice) to an LP. As can be seen [12], this decomposition lends itself well to dynamic load balancing.

In the case of spatially explicit problems, the issue of partitioning the space between Lps is of importance. Our first implementation partitioned the space into as many Lps as there are processors. To achieve better performance, the space can also be divided into more LPs than there are available processors [13].

3.2. Protocol and Memory Management

The optimistic protocol has been chosen for the heart of the simulation engine. One of the goals of this research is to develop a multipurpose simulation engine for spatially explicit problems. Since application dependent knowledge such as lookahead is not readily available, conservative techniques were not considered. The LPs in the simulation are composed of three modules: The *Event Handler*, the *Message Handler*, and the *Space Manager* (Fig. 2). The Event Handler keeps track of future and processed events as well as the local simulation clock. The Message Handler is responsible for inter-LP communications. The Space Manager is responsible for the spatial data structures and the objects present in that space.

Because the state information is large, we use incremental state saving of information necessary for rollback. When an event is processed, the state information it changes is placed into its local data structure. The event is then placed on a *processed even list*. Events that move an object from one LP to another are also

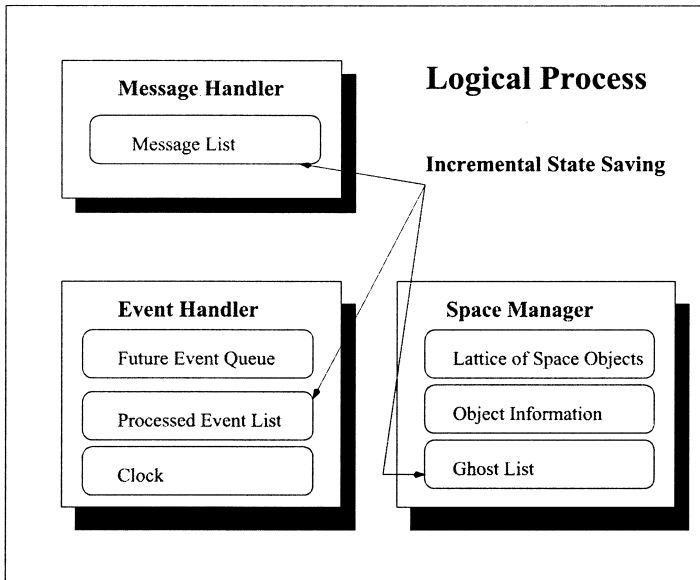


FIG. 2. Structure of the logical process.

placed in a *message list* (only pointers to the events are actually placed on the lists; the resulting duplication is not costly and speeds up sending of antimessages). If an object moves to another LP, the sending LP saves the object and the events that it is sending in a *ghost list* to be able to restore this information upon rollback. The benefit of maintaining a ghost list rather than allowing the Time Warp mechanism to perform the rollback is that a “ghost” maintains all the information about the object that has been sent (such as future messages), so that restoring the state is efficient because it minimizes the amount of information sent between LPs involved in the rollback.

When a rollback occurs, messages on the *message list* are removed and corresponding antimessages are sent out (we use aggressive cancellation). Then the events from the *processed even list* are removed and undone. Undoing an event which involved sending an object to another process entails restoring the objects from the “ghost list” and restoring future events of the object to the event queue. For other events, the parts of the state that have been changed by the events have to be restored. Recently, compiler techniques have been developed that facilitate an automatic saving of data modified by an event [8].

One of the gvt algorithms used in our system is based on the algorithm described in [36]. The gvt is calculated periodically, the interval being determined experimentally. A large interval might not allow memory to be reclaimed often enough and may result in a memory stall. If the interval is too small, the performance of the simulation might suffer. When an LP initiates a gvt calculation or receives a gvt calculation message, it enters a quiet mode. In that state the LP receives antimessages and event messages and processes events. The LP might also roll back, and, as a result, send out antimessages. It does not however, send any positive messages resulting from processing events. The LP puts these messages on the *withheld messages* list. When the gvt calculation is completed, and the new gvt has been broadcast by the processor who initiated the gvt calculation, all LPs release the messages that were withheld. After the gvt calculation, during fossil collection, the

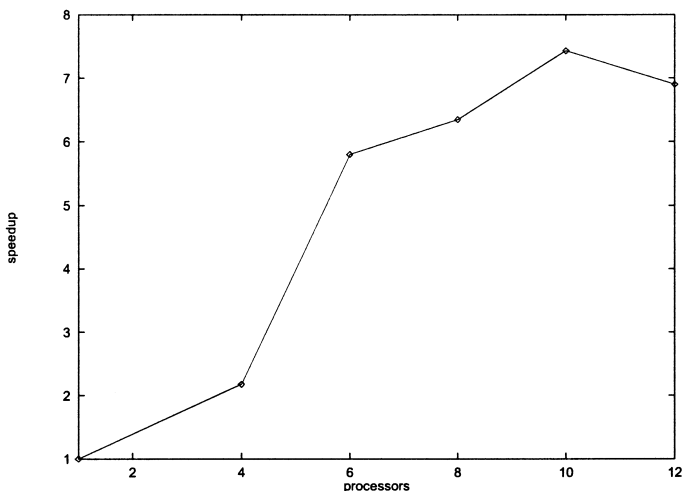


FIG. 3. Speedup for small data set.

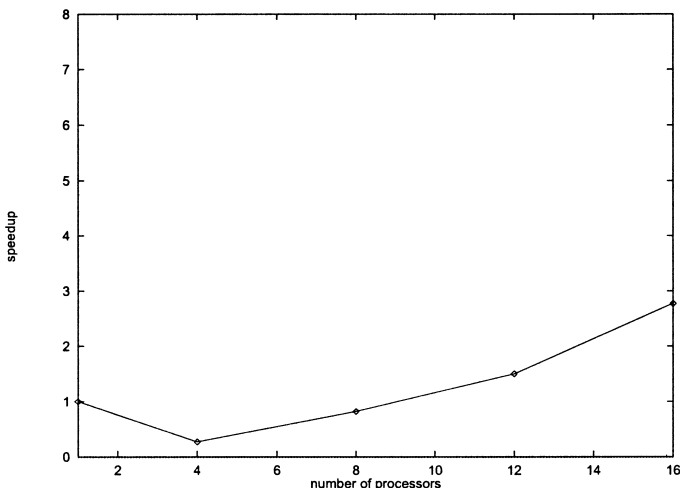


FIG. 4. Speedup for large data set.

obsolete information is removed and discarded from the three lists: the *processed event list*, the *message list*, and the *ghost list*.

The simulation engine was implemented on an IBM SP2, a distributed memory machine. We show results for up to 16 processors. The model was designed in an object oriented fashion and implemented in C++. The communications between processes use the MPI [17] message passing library.

Initial results obtained for a small-size simulation were encouraging (Fig. 3), where a maximum speedup of 7.56 has been achieved with 10 processors. (Because of the small problem size, we ran the simulation for only up to 12 processors.) The best efficiency for that problem size was almost 1 with 6 processors. With more than 10 processors, the speedup decreases as the size of the problem is too small to efficiently utilize increased numbers of processors.

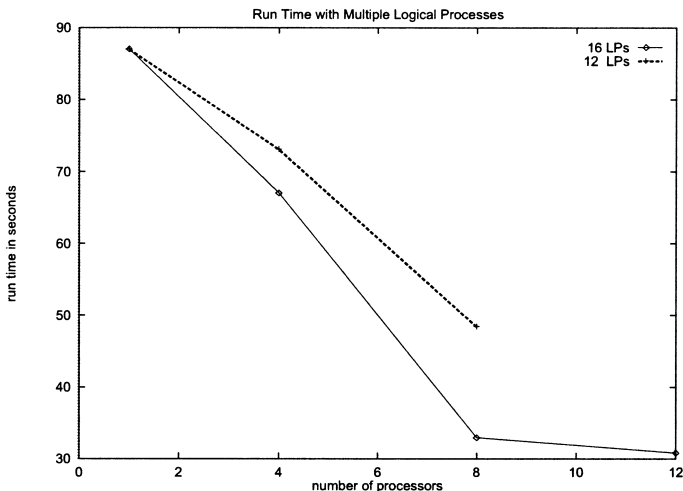


FIG. 5. Running time for large data set and multiple LPs per processor.

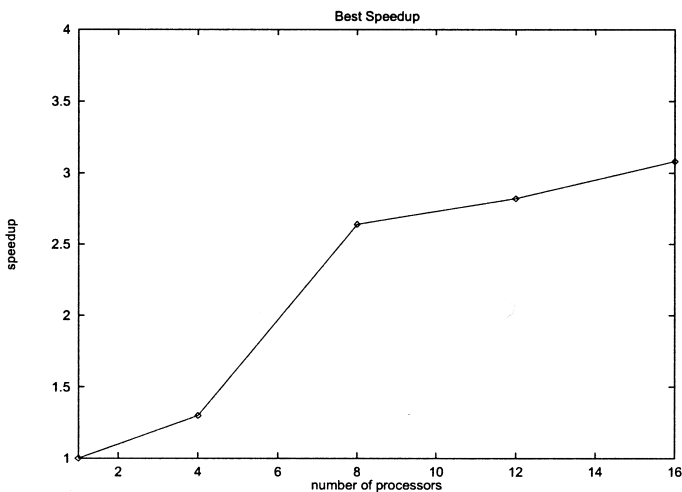


FIG. 6. Speedup with large data sets and 16LPs.

For larger simulations, the speedup was not as impressive as for small problem sizes (Fig. 4). Here, the four processor simulation actually suffers from a considerable slowdown. With 8 processors the performance is no better than sequential simulation. Finally, some speedup is achieved with 12 and 16 processors. The performance degradation is caused by the large space allocation to individual processes resulting from the increased problem size. When a rollback occurs, the entire space allocated to an LP is rolled back. Since we use incremental state saving, we need to “undo” all the events that have been processed in the space allocated to an LP. To minimize the impact of the rollback, we divided the space into more LPs, while keeping the same number of processors. Figure 5 shows the runtime improvement achieved with this approach. The figure shows that using a total of 16 LPs results in better performance than with a total of 12 LPs. When more than 16 total LPs is used, performance is not improved further, because the cost of sending messages and incurring rollbacks is increased.

For the given problem size, the ultimate number of LPs was 16 (Fig. 6), and the best efficiency of 0.33 was achieved with 8 processors. The greatest improvement in performance can be seen for small number of processors, where in all cases speedup is achieved. However, the best speedup is just over 3 with 16 processors.

4. BREADTH FIRST ROLLBACK

Making the simulation as fast as possible requires optimizing two processes: speeding up the forward simulation process and reducing the impact of a rollback on an LP. The forward computation will progress fastest on an LP when the event queue is global to it, so that the choice of the next event is quick. In order to reduce the impact of a rollback, the depth of the rollback must be kept to a minimum. It should not reach into the past further than necessary, and the number of events affected at a given time has to be minimized. For the latter, we can rely on a property of spatially explicit problems: if two events are separated in space sufficiently

far apart that one cannot affect the other (given the current logical virtual time (lvt) of the LP and the time of the rollback), then at most one of these events needs to be rolled back when a causality error occurs.

To realize this concept, we designed a new algorithm, the BFR. In this solution, the nodes of the lattice belonging to an LP (cluster) are allowed to progress independently in simulation time; however, all the nodes in a cluster are under the supervision of one LP. When a rollback occurs in a LP/cluster, only the affected lattice nodes are rolled back, thanks to a breadth-first rollback strategy. This results in an inter-cluster and intra-cluster time warp (TW).

The main innovation in BFR is that all future information is global, and information about the past is distributed among the nodes of the spatial lattice. The future information is centralized to handle the proper and fast scheduling of events, and the past information is distributed to limit the effects of a rollback. One could say that, from the point of view of the future, we treat a partition as a single LP, whereas, from the point of view of the past, we treat the partition as a set of LPs (one LP per lattice node). The performance of the new method results in a close-to-linear speedup.

Local events are easy to roll back. Assume that a local event e at location x and time t triggers an event e' at time t' and location x' (by definition of a local event). If a rollback then occurs which impacts event e , only the state of location x has to be restored to time just prior to time t . While doing that, e' will be automatically “undone.” If, however, the triggering event e is nonlocal and triggers an event e' at location $y \neq x$, then restoring the state of x is not sufficient—it is also necessary to restore the state of y just prior to the occurrence of event e' . Regardless of whether an event is local or nonlocal, the state information can be restored on a node-by-node basis.

To show the impact of a rollback on an LP, consider a straggler or an antimesage arriving at a location (x), marked in the darkest shade in Fig. 7. The rollback will proceed as follows. The events at x will be rolled back to time t_r , the time of the straggler or antimesage. Because incremental state saving is used, events have to be undone in decreasing time order to enable the recovery of state information. Since the rollback involves undoing events that happened at x , each event processed at

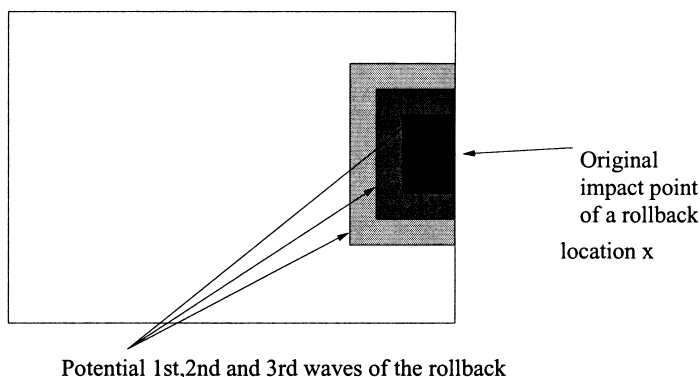


FIG. 7. Waves of rollback.

that node will be examined to determine if that event caused another event to occur at a different location. If an event e at location x is being examined, and it is determined that it triggered an event e' at location $y \neq x$, then location y has to be rolled back to the time prior to the occurrence of e' . Only then is e undone (this breath-first approach gave the name to the new method).

Incremental state saving is an important part of BFR, because it allows us to examine the previously processed events as they are being undone during a rollback. We can then determine whether an event affected events at a neighboring location, and thus exploit the locality present in spatially explicit problems.

Objects can move only from one lattice node to a neighboring one, so that a rollback can spread from one site only to its neighbors. The time of the rollback at the new site must be strictly greater than the one at site x , because we assume a nonzero delay between causally dependent events. The breadth of the rollback is limited by the speed with which simulated objects move around in space.

Figure 7 shows potential waves of rollback, from the initial impact point through three more layers of rollback. As it turns out, the size of the affected area will be usually smaller than the shaded area in Fig. 7, because events at one site will most likely not affect all their neighboring nodes. Obviously, if an event at location x triggered events on a neighboring LP, antimessages have to be sent.

It is interesting to note that each location belonging to a given LP can be at a different logical time. In fact, we do not necessarily process events in a given LP in an increasing-timestamp order. If two events are independent, an event with a higher timestamp can be processed ahead of an event with a lower timestamp. For example, if there are two locations (x and y) at times t and t_1 , respectively, where $t > t_1$ and location x gets rolled back (without affecting location y) to a time $t' < t_1$, then processing of events occurring at location x and times less than t_1 will result in these events being processed after the already processed events at location y which had larger timestamps. A similar out-of-order event processing was mentioned briefly in [33] as CO-OP (conservative-optimistic) processing. The justification is that the requirement of processing events in timestamp order is not necessary for provably correct simulations. It is only required that the events for each simulation object be processed in a correct time order. Due to this type of processing, when we process an event, we have to check the logical time of the node where the event is scheduled. If the logical time is greater than the time of the event, the node has to roll back.

4.1. Challenges of the New Approach

In order to implement BFR, some changes had to be made not only to the simulation code, but also to the model. A major change was made to the Move Event. The question arose: If an object is moving from location (x, y) to location (x_1, y_1) , where should the event be placed as “processed”? If it is placed in location (x, y) , and location (x_1, y_1) is rolled back, there would be no way of finding out that the event affected location (x_1, y_1) . If it is placed at location (x_1, y_1) , and location (x, y) is rolled back, a similar difficulty arises. Placing the Move Event in both processed lists is also not a good solution, because, in one case, the object is moving

out of the location, and, in the other case, it is moving into the location. This dilemma lead us directly to the decision of splitting the event into two: the MoveOut and MoveIn Events. Hence, when an object moves from location (x, y) to location (x_1, y_1) , the MoveOut is placed in the processed event list at (x, y) , the MoveIn at location (x_1, y_1) . The only exception is when location (x_1, y_1) belongs to another LP. In that case, the MoveIn is placed in the *processed event list* at location (x, y) (it will be placed on top of the corresponding MoveOut event), to indicate that a message was sent out.

Although, it might initially seem that the change in the model is artificial, we argue that it is more in the spirit of spatially explicit problems than the original design. From the point of view of the spatial nodes, the move events are really objects moving in and objects moving out, rather than just moves.

Upon rollback, if a MoveIn to another LP is encountered, an antimessage is sent. The result of such a treatment of antimessages, coupled with the breadth-first processing of rollbacks, implies that we are using *lazy cancellation* [20]. An antimessage is sent together with a location (x, y) to which the original message was addressed, to avoid searching the lattice nodes for this information.

Since the MoveIn Event indicates when a message has been sent, no *message list* is necessary. Another affected structure is the *ghost list*. In the original approach, objects and their events were placed on the list in the order that objects left the partition. Now the time order is not preserved; objects are placed on the list in any timestamp order because the nodes of the lattice can be at different times. The nonordered aspect of the *ghost list* poses problems during *fossil collection*. The list cannot merely be truncated to remove obsolete objects. The solution, again, is to distribute that list among the nodes. This is useful for load balancing, as described in Section 5. However, the *ghost list* is relatively small (compared to the *processed event list*), so it is not necessary to distribute the list if no load balancing is performed. It is only necessary to maintain an order in the list based on the virtual time at which the object is removed from the simulation.

Additionally, event triggering information must be preserved. In the original implementation, when an event was created, the id of the event that caused it was saved in one of the tags (the trigger) of the new event. When an event was undone, the dependent future events were removed by their trigger tags from the event queue. In BFR, it is possible that the future event is already processed, and its assigned location has not been rolled back yet. It is prohibitively expensive to traverse the future event list and then each *processed event list* in the neighborhood in search of the events whose triggers match the given event tag. The solution is to create dependency pointers from the trigger event to the newly created event. This way, a dependent event is easily accessed, and the location where it resides can be rolled back. Pointer tracking has been previously implemented for shared memory [14] to decide whether an event should be canceled or not. We also need to know if a dependent event has been processed or not, in order to be able to quickly locate it either in the event queue or in a *processed event list*.

One more change was required for the random number generation. In the original simulation, a single random number stream was used for an LP (during a rollback that stream would be rolled back as well in order to support repeatability). Now,

since the sequence of events executed on a single LP can differ from run to run, the same random number sequence can give two different results! This occurs because even though the same sequence of random number is used, the events are not necessarily processed in timestamp order, so that different results can be obtained. Obviously, nonrepeatability cannot be accepted. The solution that we chose was to distribute the random number sequence among the nodes of the lattice. Initially, a single random number sequence is used to seed the sequences at each node. From there, each node generates a new sequence. As needed, more sophisticated parallel random number generators can be used.

Since time is not uniform across the space, the global virtual time (gvt) calculation cannot be invoked based upon the distance of the local virtual time (lvt) from the previous gvt (as in the original version). Instead, it is invoked after a certain number of messages has been received from other processes since the previous gvt.

4.2. Examples

The following code constitutes the skeleton of the Breath-First Rollback (unoptimized, for clarity).

```
rollback_space (t, x, y) // rollback location x, y to time t {
  For every event E processed at x, y after time t {
    // the events are undone in the order opposite
    // to the one in which they were processed
    // update the local virtual time (lvt) to the time of the event
    if ( E.eventTime < lvt){
      lvt=E.eventTime;
    }
    // make sure to undo the dependent events first
    while there exists an unprocessed dependent event D of E{
      // (E triggered D)
      if (D is at location (x1, y1) != (x, y)){
        if (time of (x1, y1) >= E.eventTime){
          rollback_space(E.eventTime, x1, y1 );
        }
      }
    }
  }
  if (E.event_type == MOVE_IN_EVENT){
    if the new location is outside_bounds of the current LP {
      send out an anti-message for event E
      Obj=object affected by E
      // restore events that were scheduled for Obj when
      // message was sent
      restore_events_from_ghost_list(Obj, t);
    }
  }
}
```

```

undo_event(E);
insert_event(E); // into the event queue
// remove events that E triggered (from the event queue)
remove_scheduled_events(E.trigger);
}
}

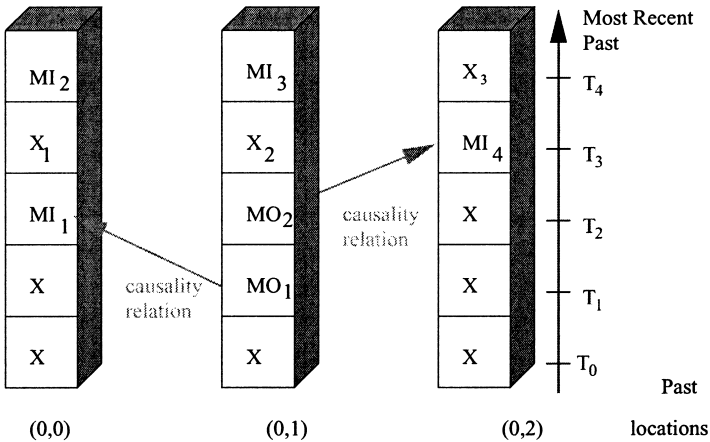
```

To demonstrate the behavior of the BFR algorithm, let's consider the example in Fig. 8. The figure shows processed lists at three different lattice nodes: (0, 0), (0, 1), and (0, 2). The event *MO* is a MoveOut event, *MI* a MoveIn event, and *X* can be any local event.

If we have a rollback for location T_0 at time t_0 , the following will happen: First MI_3 is undone and placed on the event queue. The same is done to X_2 . When MO_2 is being considered, the dependence between it and MI_4 is detected, and a rollback for location (0, 2) and time T_2 is performed. As a result, X_3 is undone and MI_4 is undone. Both are placed on the event queue. Next MO_2 is undone, which causes MI_4 to be removed from the event queue. MO_1 is examined, and (0, 0) is rolled back to time T_1 . MI_2 , X_1 , and MI_1 are undone and placed on the event queue. MO_1 is undone and MI_1 is removed from the event queue.

If the rollback occurred at location (0, 0) for time T_1 , then the three most recent events at location (0, 0) will be undone and placed on the event queue, and no other location will be affected during the rollback. It is possible that the other locations will be affected when the simulation progresses forward. If, for example, an event MO_2 was scheduled for time T_2 on (0, 0) and triggered an event MI_2 on (0, 1) for time T_3 , then location (0, 1) would have to roll back to time T_3 .

Interesting aside: We can have location (x, y) at simulation time 10. The next event in the future list is scheduled for time 20 and location (x_1, y_1) , and processed.



X could be any event
MI- MoveIn event
MO- MoveOut

FIG. 8. View of processed lists at three nodes of the lattice.

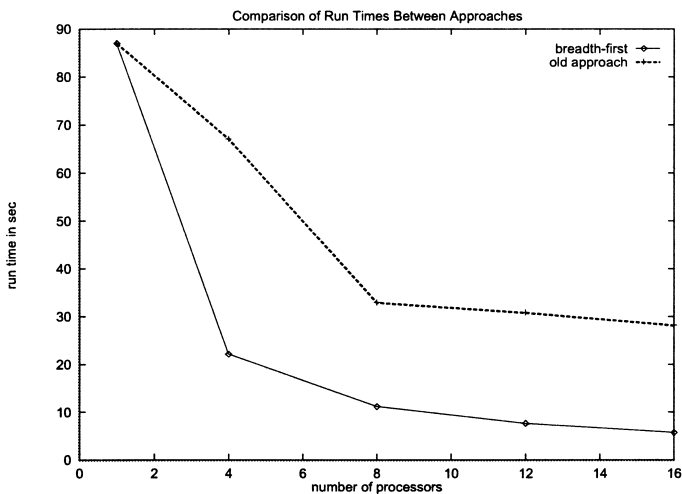


FIG. 9. Comparison of runs with BFR and the traditional approach.

If an event comes in from another process for time 15, we do not necessarily have a rollback. If the event is to occur at location (x, y) , then no rollback will happen. If, however, it is destined for location (x_1, y_1) , localized rollback will occur. As a result, comparing the timestamp of an incoming event to the lvt is not enough to determine if a rollback is necessary.

Figure 9 shows the performance of BFR and illustrates almost linear speedup. The running time of the BFR is considerably less than that of the traditional approach. The performance for 2 processors is the same. However, with 4 processors, BFR is about 3.5 times faster than the traditional approach. The best performance improvement of 4.6 times faster occurs for 16 processors. Looking at the new algorithm, we observe several benefits. The most important benefit is that, when a rollback occurs, we do not need to roll back all the events belonging to a given LP. Only the absolutely necessary events are undone. We also get fewer antimessages being sent as a result of the automatic lazy cancellation. In general, having one LP per processor eliminates on-processor communication delays. There are, of course, some drawbacks to the new method. Fossil collection is much more expensive (because lists are distributed); therefore, it is done only when the gvt has increased by a certain amount from the last fossil collection. It is harder to maintain dependency pointers than triggers, because, when an event is undone, its pointers have to be reset. The pointers have to be maintained when events are created, deleted, and undone, whereas triggers are set only once. There must be code to deal with multiple dependents. There is no aggressive cancellation, but, as can be seen from the results, that does not seem to have an adverse impact on performance.

5. BENEFITS OF BFR FOR LOAD BALANCING

Up to now, we assumed that the load distribution is even. However, if the simulation's load per LP is uneven (for example, when the odd LPs have more load than the even ones), the performance degrades, as shown in Fig. 10, where the

unbalanced computation has a speedup 1.5 smaller than the balanced computation. Luckily, BFR lends itself well to load balancing, since the local (at the node level) history tracking facilitates load balancing.

Since there is one LP per processor, the load migration is based on the migration of space between LPs, unlike in other algorithms designed for PDES where the LPs themselves are migrated between processors (in the multiple LP per processor model [1, 5, 11, 15, 38, 39]).

There are several aspects of dynamic load balancing: defining the metric of LP's load, designing a load balancing algorithm, and moving the load and making sure that the simulation remains true. The algorithm presented here requires the knowledge of loads on all processors (which can be achieved either by load broadcast or centralized load gathering) and assumes nearest-neighbor load migration—this assumption is motivated by the dynamical changes in the load distribution during the simulation run and the desire for spatial continuity within an LP.

To estimate the computational load of an LP future events are counted. The future events will need to be either processed or canceled, and thus provide a good estimate of future computational requirements (although, obviously event cancellation is faster than event processing). To differentiate between events that are scheduled to happen at different times, events are weighted according to the distance in time of the event from the beginning of the simulation. The primary advantage of counting the weights of future events instead of counting processed events, unlike some other algorithms [1, 6, 15], is that this algorithm bases its load estimate on the future of the LP rather than on its past performance. Doing so is valuable when dealing with “hot spots.” Another issue to consider is how to move the load between LPs. To reduce communication overhead, it is necessary to move lattice nodes along with the objects that are present in them and together with the events scheduled for these objects and lattice nodes. As the problem space is strip-divided between LPs (assume division in the horizontal dimension), an overloaded LP can “shed” layers of space in order to balance the load. Thus, the LP can send

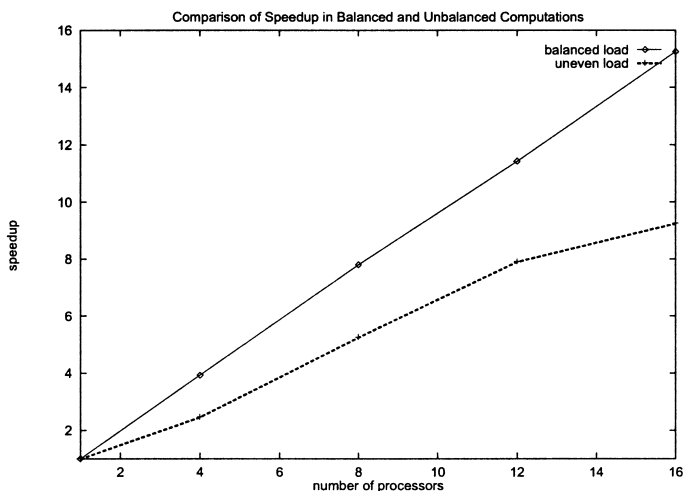


FIG. 10. Speedup for balanced and unbalanced computations.

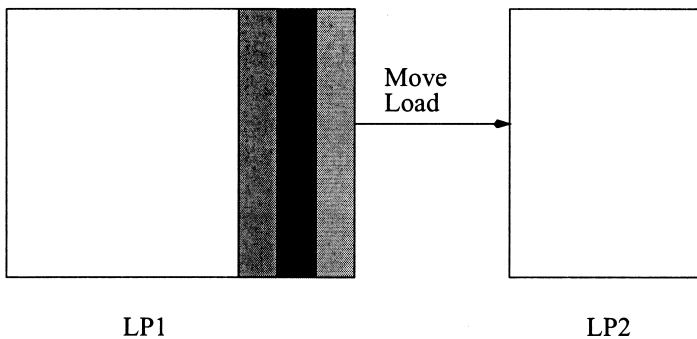


FIG. 11. Moving load between logical processes.

lattice columns to another LP (see Fig. 11) without compromising the shape of the partition. The columns have to be moved in an organized fashion—from the outer edge toward the inside. Moving columns in any other way would result in a fragmented space and would increase the communication overhead.

Each LP keeps track of the events occurring in each column of lattice nodes assigned to it. When an event is created, the cost of the event (weighted by its distance into the future) is added to the load buffer (1D array with the size of the number of lattice columns in an LP) with the index of the column in which the event is scheduled to occur. When the event is processed, sent out, or canceled, the cost of the event is subtracted from the load buffer.

Each LP calculates its own load at the time of the gvt calculation. This information is sent along with the message counts during the gvt calculation. When the gvt is calculated, the LP which initiated and concluded the calculation broadcasts the new gvt along with the loads of all the LPs in the system.

Each LP performs exactly the same algorithm. In order to minimize communications, only one round of communication is used. Each LP is permitted to send its load to at most two other LPs and to receive load from only two other LPs. Nearest-neighbor communications are also enforced, because one would like to keep the space assigned to a given LP contiguous in order to minimize inter-process communications. Since the space is strip-divided, the topology of the LPs is a ring. The algorithm identifies sequences of consecutive processes in the ring that can transfer their load over to underloaded LPs via the end processes. We call such sequences chains. The algorithm finds the dominant (“heaviest”) chains in the ring, since they contain most of the load to be given away. The load balance is performed on that chain. The algorithm is then repeated on the remaining elements of the ring until all the load has been balanced (details of the algorithm can be found in [12]).

Some tolerances must be allowed. That is, if the load of all LPs is within some percentage of the mean, then there is no need to perform the load balancing. The percentage of imbalance is a parameter of the simulation. The tolerance also prevents load “thrashing”—endless shifting of load between two almost evenly loaded neighboring processes.

After each LP calculates how much load it needs to send/receive to/from its neighbors, the columns of lattice node are sent, in addition to all the objects in that

space and the events associated with these objects and space. Since each lattice node contains a list of processed events, these lists are sent, as well. The sending LP will have a new space boundary.

First, columns of space are sent to the neighbor. Next, the objects present in that space need to be sent. It is also possible that an object was in the middle of a move when the load balancing started. In other words, an object was removed from the old location but not yet placed at the new location. In that case, the *MoveIn* event is still in the Future Event Queue. These objects have to be sent, as well. There are also events in the Future Event Queue that are not associated with an object, but rather with a lattice node (placing of ticks at a node, for example). These types of events, which are scheduled to happen in the space being sent, have to be sent, as well.

There might also have been some objects which moved across the new boundary between the receiver and the sender. These objects have to be made into ghosts for the receiving LP. These ghosts, along with the ghosts of the objects killed in the space being sent are sent to the receiver. Finally, the processed events from that space are sent.

Receiving the space is straightforward: the space objects are received first, then the objects, the future events, the ghosts, and finally the processed events. There is some processing on the receiving side due to the restructuring of the space. For example, the processed events have to be placed at the appropriate lattice nodes, the objects have to be placed in the space, and the future events have to be inserted in the Future Event Queue.

Figures 12 and 13 illustrate the results of experiments for two problem sizes: small and large. The small size consists of approximately 10,000 lattice nodes and 2000 objects, whereas the large problem has 40,000 lattice nodes and approximately 4000 objects. In each problem 100 columns of spaces are made “heavy” by having twice as many objects as other columns.

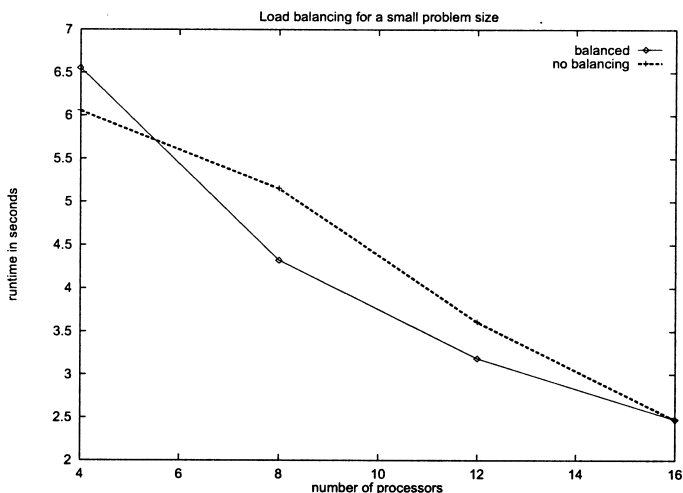


FIG. 12. Load balancing for a small problem size and heavy lattice columns.

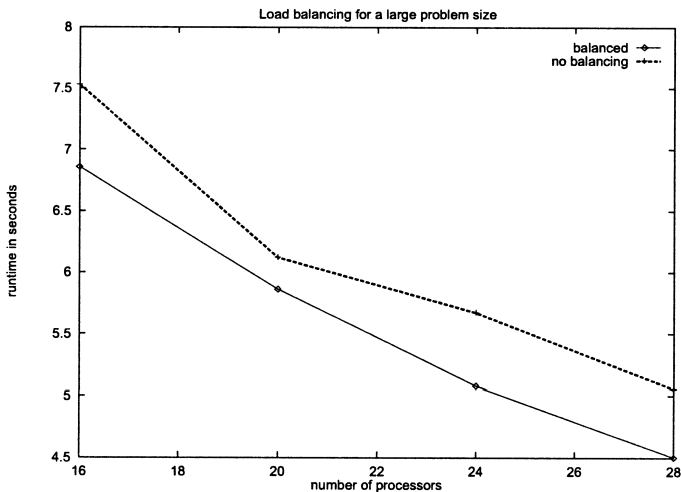


FIG. 13. Load balancing for a large problem size and heavy lattice columns.

Figure 12 shows the results for the small problem size. For that problem size, the results are not impressive. It is interesting to note that although the load balancing does better with 8 and 12 processors, it does not perform well with 4 or 16 processors. For 4 processors, the amount of load that has to be moved might take too long, and for 16 processors, the problem size might be too small to overcome the cost of load calculation. With the large problem size (see Fig. 13), the load balancing algorithm performs well from 16 to 28 processors, with an average improvement of 9% relative to the nonbalanced computation.

6. CONCLUSIONS

We have presented an integrated approach to the simulation of a general class of problems, described as spatially explicit, on distributed memory architectures. We have designed a new simulation protocol and integrated it with a new dynamic load balancing algorithm. We have addressed the issue of efficient simulation by designing a new algorithm for rollback processing and space partitioning. The algorithm is based on the optimistic protocol and relies on the space being partitioned into a multi-dimensional lattice. Rollbacks are minimized by examining the processed event list of each lattice node during rollback, in search of causal dependencies between events which span the lattice nodes. The rollback impacts the minimum number of sites, making the simulation very efficient. As a result, an almost linear speedup is achieved for well balanced computations. In cases where the system performance suffers due to unbalanced load, we use a dynamic load algorithm which we designed to work with BFR. The algorithm migrates load between neighboring LPs while preserving the continuity of the space. Clearly, dynamic load balancing can be useful, but it needs to be applied discriminately. Results indicate that load balancing is most worthwhile when the problem size is large. Also, given the same level of load imbalance, systems with larger numbers of processors appear to perform better, because more of the load can be moved in parallel.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grants BIR-9320264, CCR-9527151, and KDI-9873138.

REFERENCES

1. H. Avril and C. Tropper, The dynamic load balancing of clustered time warp for logic simulations, in "Workshop on Parallel and Distributed Simulation, 1996," pp. 20–27.
2. H. Avril and C. Tropper, Clustered time warp and logic simulation, in "Workshop on Parallel and Distributed Simulation, 1995," pp. 112–119.
3. R. Bagrodia, M. Gerla, L. Kleinrock, J. Short, and T. C. Tsai, A hierarchical simulation environment for mobile wireless networks, in "Winter Simulation Conference, 1994," pp. 1354–1361.
4. A. Barbour and D. Fish, The biological and social phenomenon of Lyme disease, *Science* **260** (1993), 1610–1616.
5. C. Burdoff and J. Marti, Load balancing strategies for time warp on multi-user workstations, *Comput. J.* (1993), 168–176.
6. C. Carothers and R. Fujimoto, Efficient execution of time warp programs on heterogeneous, NOW platforms, *IEEE Trans. Parallel Distrib. Systems* **11** (2000), 299–317.
7. C. D. Carothers, R. M. Fujimoto, and Y. B. Lin, A case study in simulating PCS networks using time warp, in "Workshop on Parallel and Distributed Simulation, 1995," pp. 87–94.
8. C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, Efficient optimistic parallel simulations using reverse computation, in "Workshop on Parallel and Distributed Simulation, 1999," pp. 126–135.
9. K. M. Chandy and J. Misra, Distributed simulation: A case study in design and verification of distributed programs, *IEEE Trans. Software Engng.* **5** (1979), 440–452.
10. Y. Chen, V. Jha, and R. Bagrodia, A multidimensional study on the feasibility of parallel switch-level circuit simulation, in "Workshop on Parallel and Distributed Simulation, 1997," pp. 46–54.
11. M. Choe and C. Tropper, On learning algorithms and balancing loads in time warp, in "Thirteenth Workshop on Parallel and Distributed Simulation PADS '99, 1999," pp. 101–108.
12. E. Deelman and B. K. Szymanski, Dynamic load balancing in parallel discrete event simulation for spatially explicit problems, in "Twelfth Workshop on Parallel and Distributed Simulation PADS '98, 1998," pp. 46–53.
13. E. Deelman and B. K. Szymanski, Simulating lyme disease using parallel discrete event simulation, in "Winter Simulation Conference, 1996," pp. 1191–1198.
14. R. M. Fujimoto, Parallel discrete event simulation, *Comm. ACM* **33** (1990), 31–53.
15. D. W. Glazer and C. Tropper, On process migration and load balancing in time warp, *IEEE Trans. Parallel Distrib. Systems* **4** (1993), 318–327.
16. A. G. Greenberg, B. D. Lubachevsky, D. M. Nicol, and P. E. Wright, Efficient massively parallel simulation of dynamic channel assignment schemes for wireless cellular communications, in "Workshop on Parallel and Distributed Simulation, 1994," pp. 187–194.
17. W. Gropp, E. Lusk, and A. Skjellum, "Using MPI," MIT Press, Cambridge, MA, 1994.
18. J. B. Hiller and T. C. Hartrum, Conservative synchronization in object-oriented parallel battlefield discrete event simulations, in "Workshop on Parallel and Distributed Simulation, 1997," pp. 12–19.
19. D. R. Jefferson, Virtual time, *Trans. Prog. Lang. Syst.* **7** (1985), 404–425.
20. Y. Lin and E. D. Lazowska, A study of time warp rollback mechanisms, *ACM Trans. Model. Comput. Simulations* (1991), pp. 51–72.
21. Y. B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska, Selecting the checkpoint interval in time warp simulation, in "Workshop on Parallel and Distributed Simulation, 1993," pp. 3–10.
22. V. Madiseti, J. Walrand, and D. Messersmith, Wolf: A rollback algorithm for optimistic distributed simulation systems, in "Winter Simulation Conference, December 1988," pp. 296–305.

23. B. A. Malloy and A. T. Montroy, A parallel distributed simulation of a large-scale PCS network: Keeping Secrets, in "Winter Simulation Conference, 1995," pp. 571–578.
24. G. L. Miller, R. B. Craven, R. E. Bailey, and T. F. Tsai, The epidemiology of Lyme disease in the United States 1987–1998, *Laboratory Med.* **21** (1990), 285–289.
25. R. S. Ostfeld, K. R. Hazler, and O. M. Cepeda, Temporal and spatial dynamics of *ixodes scapularis* (Acari: Ixodidae) in a rural landscape, *J. Med. Entomology* **33** (1996), 90–95.
26. F. Quaglia, Event history based sparse state saving in time warp, in "Twelfth Workshop on Parallel and Distributed Simulation PADS '98, 1998," pp. 72–79.
27. F. Quaglia, Combining periodic and probabilistic checkpointing in optimistic simulation, in "Thirteenth Workshop on Parallel and Distributed Simulation PADS '99, 1999," pp. 109–116.
28. H. Rajaei, R. Ayani, and I. Thorelli, The local time warp approach to parallel simulation, in "Workshop on Parallel and Distributed Simulation, 1993," pp. 119–126.
29. P. Reiher, F. Wieland, and D. Jefferson, Limitation of optimism in time warp operating system, in "Winter Simulation Conference, 1989," pp. 765–770.
30. R. Schlagenhaft, M. Ruhwandl, C. Sporrer, and H. Bauer, Dynamic load balancing of a multi-cluster simulation of a network of workstations, in "Workshop on Parallel and Distributed Simulation, 1995," pp. 175–180.
31. S. Srinivasan and P. F. Reynolds, Adaptive algorithms vs. time warp: An analytical comparison, in "Proceedings of the 1995 Winter Simulation Conference, 1995," pp. 666–673.
32. S. Srinivasan and P. F. Reynolds, Npsi adaptive synchronization algorithms for pdes, in "Proceedings of the 1995 Winter Simulation Conference, 1995," pp. 658–665.
33. J. S. Steinman, SPEEDES: A unified approach to parallel simulation, in "Workshop on Parallel and Distributed Simulation, 1992," pp. 75–84.
34. J. S. Steinman, Breathing time warp, in "Workshop on Parallel and Distributed Simulation, 1993," pp. 109–118.
35. J. S. Steinman, Incremental state saving in SPEEDES using C++, in "Winter Simulation Conference, 1993," pp. 687–696.
36. J. S. Steinman, C. A. Lee, L. F. Wilson, and D. M. Nicol, Global virtual time and distributed synchronization, in "Workshop on Parallel and Distributed Simulation, 1995," pp. 139–148.
37. F. Wieland, L. Hawley, A. Feinberg, M. Di Loreto, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson, Distributed combat simulation and time warp: The model and its performance, *Distrib. Simulation* (1989), 14–20.
38. L. F. Wilson and D. M. Nicol, Automated load balancing in SPEEDES, in "Winter Simulation Conference, 1995," pp. 590–596.
39. L. F. Wilson and D. M. Nicol, Experiments in automated load balancing, in "Workshop on Parallel and Distributed Simulation, 1996," pp. 4–11.