Proc. 11th Workshop on Parallel and Distributed Simulation (PADS97), Lockenhaus, Austria, June 10-13, 1997, '
IEEE Computer Society Press, Los Alamitos, CA, pp. 124-131

Breadth-First Rollback in Spatially Explicit Simulations

Ewa Deelman

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180
deelmane@cs.rpi.edu

Boleslaw K. Szymanski

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180
szymansk@cs.rpi.edu

Abstract

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. 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. Our results demonstrate an almost linear speedup—a dramatic improvement over the traditional approach to rollback processing.

1 Introduction

One of the major challenges of Parallel Discrete Event Simulation (PDES) is to achieve good performance. This goal is difficult to attain, because, by its very nature, discrete event simulation organizes events in a priority queue based on the timestamp of events, and processes them in that order. When porting a simulation to a parallel platform, this priority queue is distributed among logical processes (LPs) that correspond to the physical processes that are being modeled. Because the LPs interact with each other by sending event messages, it is costly to maintain the causality between events. Two basic protocols have been developed to ensure that causality constraints are satisfied [9]: conservative [5] and optimistic. In Time Warp (TW) [11], the best known optimistic protocol, causality errors are allowed to occur, but when such an error is detected, the erroneous computation is rolled back. The research described in this paper utilizes the optimistic protocol and focuses on optimizing rollback processing.

The method of rollback processing we will present is applicable to simulations that consist of a space with objects moving freely in it; the space is discretized into a multi-dimensional lattice and divided among LPs. We make use of incremental state saving techniques [18] to detect dependencies between events. Typical implementations of a rollback in such a setting (used in our previous implementation [7]) is to roll back the entire area assigned to the LP. In this paper, we present a novel approach, termed *Breadth-First Rollback* (BFR), in which the rollback is contained to the area that has been directly affected by the *straggler* (event message with a timestamp smaller than the current simulation time) or *antimessage* (cancellation of an event). We also present the improved simulation speedup and performance resulting from the use of this approach.

The application that motivated this work is a Lyme disease simulation in which the two-dimensional space is discretized into a two-dimensional lattice. The most important characteristics of the simulation are: the mobile objects moving freely in space (mice) 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, mouse deaths, etc.) and (ii) non-local (such as a mouse moving from one node to another—*Move Event*).

The simulation currently runs on an IBM SP2 (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 [10] message passing library.

2 Related Work

There are two inter-related issues that have arisen in optimizing optimistic protocols for PDES. One is the need to reduce the overhead of rollbacks, and the other is to limit the administrative overhead of partitioning a problem into many "small" LPs (as happens, for example, in digital logic simulations). To address

both of these issues, clustering of LPs is often used.

Lazy re-evaluation [9] has been 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, than 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.

The Local Time Warp (LTW) [15] approach combines two simulation protocols by using the optimistic protocol 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 such a choice 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 efficient, 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 [16], 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.

In the case of spatially explicit problems, the issue of partitioning the space between LPs is also of importance. Discretizing the space results in a multi-dimensional lattice for which the following question arises: Should one LP be assigned to each lattice node (which results in high simulation overhead) or should the lattice nodes be "clustered" and the resulting clusters be assigned to LPs? Our first implementation of Lyme disease used the latter approach and assigned spatially close nodes to a single LP, with TW used between the LPs. This was similar to the CTW, except that our implementation did not have multiple LPs within a cluster, to simulate space more efficiently. Unfortunately, this approach did not perform as well as we had hoped, especially when the problem size grew larger, because when a rollback occurred in a cluster, the entire cluster had to roll back.

To improve performance, 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, explained in Section 3. This approach can be classified as 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 facilitate 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 yields a speedup which is close to linear.

3 Breadth-First Rollback Approach

Breadth-First Rollback is designed for spatially explicit, optimistic PDES. The space is discretized and divided among LPs, so each LP is responsible for a set of interconnected lattice nodes. The speed of the simulation is dictated by the efficiency of two steps: the forward event and the rollback processing. The forward computation is facilitated when the event queue is global to the executing LP, so that the choice of the next event is quick. The impact of a rollback is reduced when the depth of the rollback is kept to a minimum: the rollback should not reach further into the past 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 located sufficiently far apart in space, one cannot affect the other (for certain values of the current logical virtual time (*lvt*) of the LP and the time of the rollback), so at most one of these events needs to be rolled back when a causality error occurs.

Events can be classified as local or non-local. A local event affects only the state of one lattice node. A non-local event, for example the Move Event, which moves an object from one location to the next, affects at least two nodes of the lattice. Local events are easy to roll back. Assume that a local event e at location

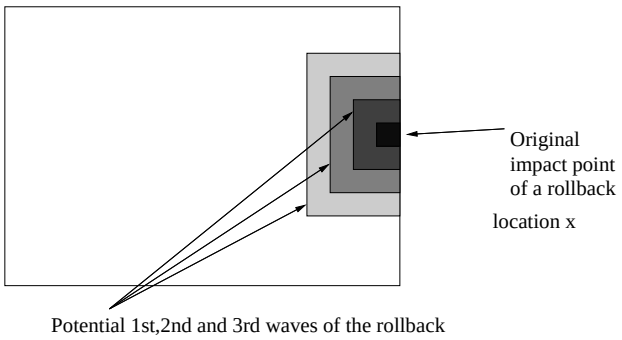

Figure 1: Waves of Rollback.

x and time t triggers an event e_1 at time t_1 and the same 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 the time just prior to time t. While restoring the state, e_1 will be automatically "undone". If, however, the triggering event e is non-local and triggers an event e_1 at location $x_1 \neq x$, then restoring the state of location x is not sufficient—it is also necessary to restore the state of location x_1 just prior to the occurrence of event e_1. Regardless of whether an event is local or non-local, 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 antimessage arriving at a location x, marked in the darkest shade in Figure 1. The rollback will proceed as follows. The events at x will be rolled back to time t_r, the time of the straggler or antimessage. Since incremental state saving is used, events have to be undone in decreasing time order to enable the recovery of state information. The rollback involves undoing events that happened at x. Each event e processed at that node will be examined to determine if e caused another event (let's call it e_1) to occur at a different location $x_1 \neq x$ (non-local event). In such a case, location x_1 has to be rolled back to the time prior to the occurrence of e_1. Only then is e undone (this breath-first wave gave the name to the new approach).

In our simulation, 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 there is a non-zero delay between causally-dependent events. In general, the breadth of the rollback is bounded by the speed with which simulated objects move around in space.

Figure 1 shows potential waves of a rollback, from the initial impact point through three more layers of processing. In practice, the size of the affected area is usually smaller than the shaded area in Figure 1, 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. A similar type of processing was mentioned briefly in [17] 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 (in the forward execution), 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 Comparison With The Traditional Approach

To demonstrate improvements in performance, we present below the model used in our initial simulation, which did not use the BFR method. The space, as previously mentioned, is discretized into a two-dimensional lattice. Similar discretization is used, for example, in personal communication services [4], where the space is discretized by representing the network as hexagonal or square cells. In these simulations, each cell is modeled by an LP. In our research, we have developed a simulation system for spatially explicit problems. The particular application we describe in this paper is the simulation of the spread of the Lyme disease.

In Lyme disease simulation, it would be prohibitively expensive to assign one LP to each lattice node, so we "cluster" lattice nodes into a single LP. Currently, the space is divided strip-wise among the available processors. Of course, other spatial decompositions can be used. To achieve better performance, the space can also be divided into more LPs than there are available processors [8].

The LPs in this simulation are called *Space Managers*, because they are responsible for all the events that happen in a given region of space. If the *Space*

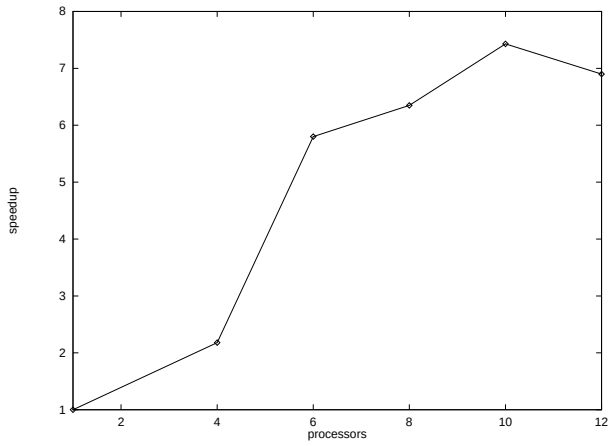

Figure 2: Speedup For Small Data Set (about 2,400 nodes).

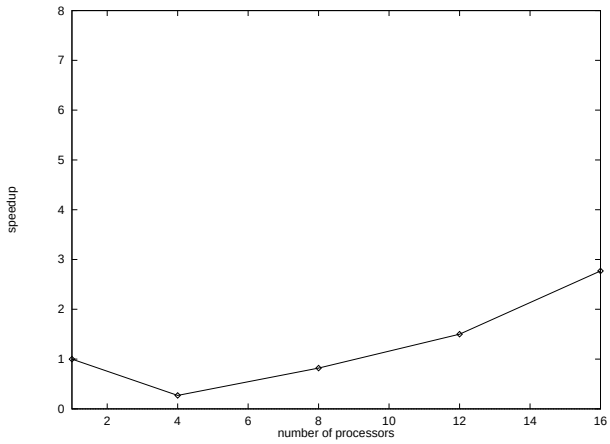

Figure 3: Speedup For Large Data Set (about 32,000 nodes).

Manager determines that an object moves out of local space to another partition, the object and all its future events are sent to the appropriate *Space Manager*. As previously mentioned, the optimistic approach is used to allow concurrent processing of events happening at the same time at different locations.

Because the state information is large, we use incremental state saving of information necessary for rollback. When an event is processed, the state information that it changes is placed into its local data structure. The event is then placed on a *processed event list*. Events that move an object from one LP to another are also 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 corresponding events in a *ghost list* to be able to restore this information upon 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 event 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. During fossil collection, the obsolete information is removed and discarded from the three lists: the *processed event list*, the *message list*, and the *ghost list*.

Initial results obtained for a small-size simulation were encouraging (Figure 2); however, the speedup

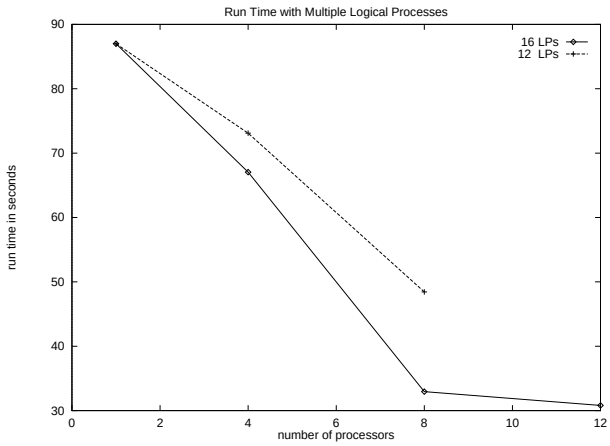

Figure 4: Running Time for Large Data Set and Multiple LPs per Processor.

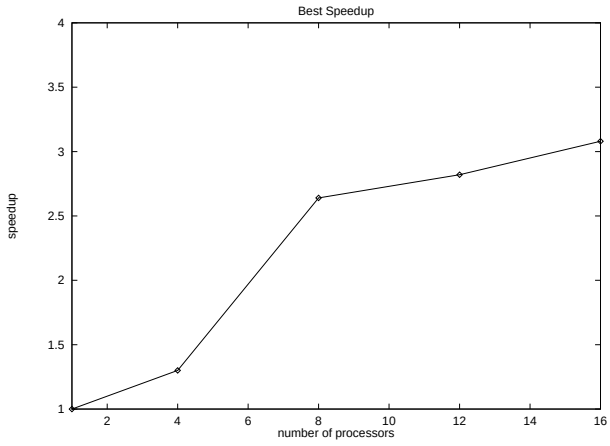

Figure 5: Speedup with Large Data set and 16LPs.

was not impressive for larger simulations (Figure 3). 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. To minimize the impact of the rollback, we divided the space into more LPs, while keeping the same number of processors. Figure 4 shows the runtime improvement achieved with this approach. For the given problem size, the ultimate number of LPs was 16 (Figure 5), and the best efficiency was achieved with 8 processors.

5 Challenges Of The New Approach

In order to implement BFR, some changes had to be made not only to the simulation engine, 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 object 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 motivated us to split the Move 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) and 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.

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, gives us an effect of *lazy cancellation* [12]. 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 non-ordered 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 the final section. However, the *ghost list* is relatively small (compared to the *processed event list*), so it might not be necessary to distribute the list if no load balancing is performed. It is sufficient 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 identity 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 events. This way, a dependent event is easily accessed, and the location where it resides can be rolled back. Pointer tacking has been previously implemented for shared memory [9] to decide whether an event should be canceled or not. In our approach, 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. These numbers are used, for example, in calculating the time of occurrence of new events. Now, since the sequence of events executed on a single LP can differ from run to run, the same random number sequence can yield two different results! Obviously, result repeatability is important, so we chose 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.

6 Examples

To demonstrate the behavior of the BFR algorithm, let's consider the example in Figure 6. The figure shows processed lists at three different lattice nodes: (0,0),(0,1), and (0,2). The event MO is a MoveOut

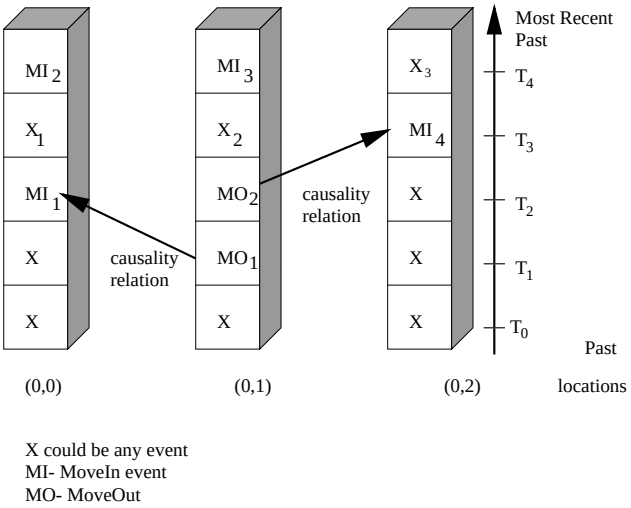

Figure 6: View of Processed Lists at Three Nodes of the Lattice.

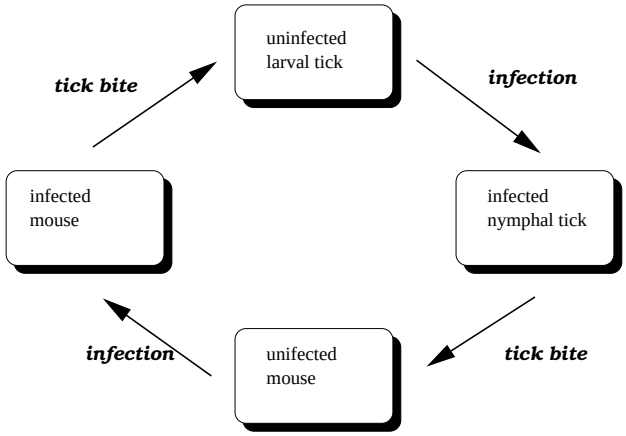

Figure 7: The Cycle of Lyme Disease

event, MI a MoveIn event, and X can be any local event.

If we have a rollback for location (0,1) 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_z was scheduled for time T_2 on (0,0) and triggered an event MI_z 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 t. The next event in the future list is scheduled for time t_1 and location (x_1,y_1), and processed. If an event comes in from another process for time t_2 ($t < t_2 < t_1$), we do not necessarily incur 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 local virtual time is not enough to determine if a rollback is necessary.

7 Application Description

Before we present the results obtained with BFR, it is important to sketch our application—the simulation of Lyme disease. This disease is prevalent in the Northeastern United States [3, 13]. People can acquire the disease by coming in contact with a tick infected with the spirochete, which may transfer into the human's blood, causing an infection. Since the ticks are practically immobile, the spread of the disease is driven by the ticks' mobile hosts, such as mice and deer. Even though the most visible cases of Lyme disease involve humans, the main infection cycle consists of ticks and mice (Fig. 7). 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 seasonal cycle of the disease, and the duration of the simulation, is 180 days, starting in the late spring[6]. This time is the most active for the ticks and mice. Mice, during that time, are looking for nesting sites and may carry ticks a considerable distance [14].

The mice are modeled as individuals, and ticks, because of their sheer number (as many as 1200 larvae/400m^2 [14]) are treated as "background". The space is discretized into nodes of size 20x20m^2, which represent the size of the home range for a mouse. Each node may contain any number of ticks in various stages of development and various infection status. Mice 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 can die (*Kill Event*) if they cannot find a nesting site or by other natural causes, such as old age, attacks by

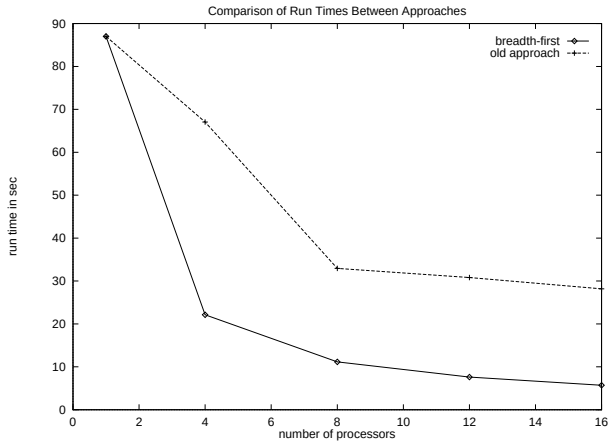

Figure 8: Results: Comparison of Runs With BFR and the Traditional Approach.

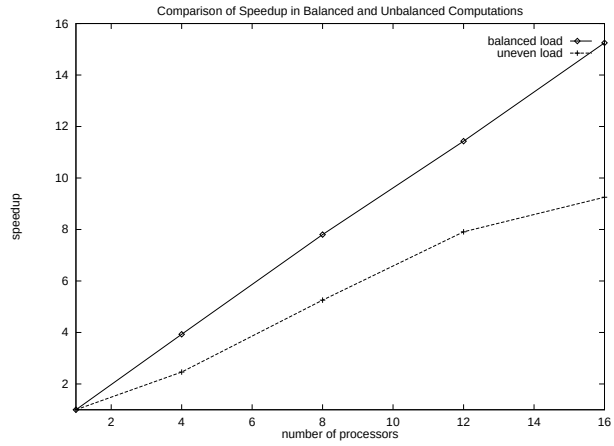

Figure 9: Speedup for Balanced and Unbalanced Computations.

predators, and disease. Mice 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 non-local.

8 Results

Figure 8 shows the performance of BFR and illustrates almost linear speedup. The running time of the BFR is considerably shorter than that of the traditional approach. 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 necessary events are undone. In the traditional approach, the number of events that needed to be rolled back was ultimately proportional to the number of lattice nodes assigned to a given LP. When a rollback occurred, all the events that happened in that space had to be undone. On the other hand, when a rollback occurs in the BFR version, the number of events being affected by a rollback is proportional to the length of the edges of the space that interface with other LPs. In the case of the space divided into strips, the number of events affected by a given rollback is proportional to the length of the two communicating edges. Therefore, when the size of the space assigned to a given LP increases (when the number of LPs for a given problem size decreases), the number of events affected by a rollback in the case of BFR remains roughly constant. In the traditional approach, that number increases proportionally to the increased length of the non-communicating edges. Consequently, we observe that the number of events rolled back using BFR is an order of magnitude smaller than that in the traditional approach.

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 Global Virtual Time 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.

9 Conclusions and Future Work

We have described a new algorithm for rollback processing in spatially explicit problems. 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. Obviously this performance is attainable thanks to a large amount of parallelism existing in the application.

Up to now, we did not address the issue of load distribution. If the simulation's load per LP is uneven (for example, when the odd LPs have more load then

the even ones), the performance degrades, as shown in Figure 9. Another advantage of BFR is that it lends itself well to load balancing, since the local (at the node level) history tracking facilitates load balancing. An overloaded LP can "shed" layers of space in order to balance the load. Nothing special needs to happen on the receiving side. If messages were sent to the space that just arrived, they are simply discarded by the sender of the space and reconstructed from the ghost list by the receiver (we assume that load can only be exchanged between neighboring processes). On the sending side, however, the priority queue has to be filtered in order to extract the future events for the area sent to the new process. In order to decide if there is a need to migrate the load, the event queue can be scanned to determine the event density. Since there is a large number of events in the queue at any given time, this quantity might prove to be a good measure of load. If the density is too high at some process, some of the space can be sent to the neighboring processes.

Acknowledgments

This work was supported by the National Science Foundation under Grants BIR-9320264 and CCR-9527151. The content of this paper does not necessarily reflect the position or policy of the U.S. Government—no official endorsement should be inferred or implied.

References

[1] H. Avril and C. Tropper. The Dynamic Load Balancing of Clustered Time Warp for Logic Simulations. *Workshop on Parallel and Distributed Simulation*, pages 20–27, 1996.

[2] H. Avril and Carl Tropper. Clustered time warp and logic simulation. *Workshop on Parallel and Distributed Simulation*, pages 112–119, 1995.

[3] A. Barbour and D. Fish. The biological and social phenomenon of Lyme disease. *Science*, 260:1610–1616, 1993.

[4] C.D. Carothers, R.M. Fujimoto, and Y.B. Lin. A Case Study in Simulating PCS Networks Using Time Warp. *Workshop on Parallel and Distributed Simulation*, pages 87–94, 1995.

[5] K. M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, 5:440–452, 1979.

[6] Ewa Deelman, Thomas Caraco, and Boleslaw K. Szymanski. Parallel Discrete Event Simulation of Lyme Disease. *Pacific Biocomputing Conference*, pages 191–202, 1996.

[7] Ewa Deelman and Boleslaw K. Szymanski. Continuously Monitored Global Virtual Time in Parallel Discrete Event Simulation. Technical Report 96-18, Department of Computer Science, Rennselaer Polytechnic Institute, 1996.

[8] Ewa Deelman and Boleslaw K. Szymanski. Simulating Lyme Disease Using Parallel Discrete Event Simulation. *Proceedings of the 1996 Winter Simulation Conference*, pages 1191–1198, 1996.

[9] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):31–53, 1990.

[10] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI.* The MIT Press, 1994.

[11] D.R. Jefferson. Virtual Time. *Trans. Prog. Lang. and Syst.*, 7:404–425, 1985.

[12] Y. Lin and E. D. Lazowska. A Study of Time Warp Rollback Mechanisms. *ACM Transactions on Modeling and Computer Simulations*, pages 51–72, 1991.

[13] 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 Medicine*, 21:285–289, 1990.

[14] R.S. Ostfeld, K.R. Hazler, and O.M. Cepeda. Temporal and Spatial Dynamics of *Ixodes scapularis* (Acari: Ixodidae) in a rural landscape. *Journal of Medical Entomology*, 33:90–95, 1996.

[15] H. Rajaei, R. Ayani, and l. Thorelli. The Local Time Warp Approach to Parallel Simulation. *Workshop on Parallel and Distributed Simulation*, pages 119–126, 1993.

[16] R. Schlagenhaft, M. Ruhwandl, C.Sporrer, and H. Bauer. Dynamic Load Balancing of a Multi-Cluster Simulation of a Network of Workstations. *Workshop on Parallel and Distributed Simulation*, pages 175–180, 1995.

[17] J. S. Steinman. SPEEDES: A Unified Approach to Parallel Simulation. *Workshop on Parallel and Distributed Simulation*, pages 75–84, 1992.

[18] J. S. Steinman. Incremental State Saving in SPEEDES using C++. *Winter Simulation Conference*, pages 687–696, 1993.