PERFORMANCE OPTIMIZATION OF PARALLEL DISCRETE EVENT SIMULATION OF SPATIALLY EXPLICIT PROBLEMS

By

Ewa Deelman

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

Dr. Boleslaw K. Szymanski, Thesis Adviser

Dr. Thomas Caraco, Member

Dr. Franklin Luk, Member

Dr. David Musser, Member

Dr. Charles Stewart, Member

Rensselaer Polytechnic Institute
Troy, New York

November 1997
(For Graduation December 1997)

PERFORMANCE OPTIMIZATION OF PARALLEL DISCRETE EVENT SIMULATION OF SPATIALLY EXPLICIT PROBLEMS

By

Ewa Deelman

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Rensselaer Polytechnic Institute
Troy, New York

November 1997
(For Graduation December 1997)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

This thesis has benefited greatly from the wisdom and guidance of my adviser, Prof. Boleslaw Szymanski. For his support, his understanding, and his assistance, I am deeply grateful.

Prof. Tom Caraco of the Department of Biology at The University at Albany has contributed his expertise and helped ensure that the simulations which motivated this work reflect biological reality. It has been a pleasure to collaborate with Tom.

I also wish to thank the other members of my committee, Prof. Franklin Luk, Prof. Dave Musser, and Prof. Charles Stewart, for their comments, suggestions, and last, but certainly not least, their time.

During my graduate career, I've been privileged to have a remarkable group of friends, who have shared with me their insight, their good humor, occasionally their apartments, and frequently their Happy Hours. I could not have conducted this work, let alone finished it, without the help of Andrew, Bill, Bowden, Chak, Charles, Daveed, Jim, Louis, Mary Anne, Mohan, Nathan, Peter, Teek, Wes, and many others.

Finally, I thank my family, my mother Teresa, Adam, my father Wojciech, and Alicja for the support and patience which enabled me to undertake and complete my studies. At each step, they have been by my side, encouraging me.

Most of all I thank my wonderful and supportive husband Peter, who not only encouraged me and helped me along the way, but also showed by his example how to pursue and achieve one's dreams. This work would never have been written without his love and caring. Therefore, I would like to dedicate my thesis to Peter Wessel Deelman.

ABSTRACT

The research presented in this thesis describes the simulation of spatially explicit problems using Parallel Discrete Event Simulation (PDES) with an optimistic protocol. Spatially explicit problems are characterized by continuous space in which objects reside and move freely. The space is heterogeneous and the interactions between objects are frequent and localized. The space is discretized and partitioned among processes, each simulating an area of space. From the point of view of the parallel simulation processes, communications are frequent and take place between nearest-neighbors. These problem characteristics have an impact on three areas of Parallel Discrete Event Simulation: the Global Virtual Time (GVT) calculation overhead, the performance cost incurred by rollbacks, and load balance. Improvements are proposed for each of these performance issues.

The bottleneck in PDES is memory. Since the simulation uses large amounts of memory, the GVT is used to synchronize processes and discard obsolete system information. In this work a new algorithm, the **Continuously Monitored Global Virtual Time** (CMGVT) is presented. Unlike other GVT algorithms, the CMGVT allows processes to calculate the GVT based on the local information constantly available to each process. System information, such as the Local Virtual Time of each process and information about messages in transit, is appended to simulation messages. Three variants of the CMGVT algorithm with progressively increasing overhead are described and their performance is compared.

To address the issue of rollback overhead, a novel approach to rollback processing which limits the number of events rolled back as a result of a straggler or antimessage is described. 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.

A dynamic load balancing algorithm based on the BFR processing method is described. Load predictions are determined based on the future events that are scheduled for a given Logical Process (LP). The information about the load of the processes is gathered and distributed during the Global Virtual Time calculation. Each LP calculates the new load distribution of the system. The load is balanced by moving spatial data between neighboring LPs in one round of communications. In the class of problems investigated in this thesis, the LPs can be described as being elements of a ring from the point of view of communication. Under this assumption, an algorithm that balances the load in a ring and achieves the smallest possible maximum after-balance load is presented.

The spread of *Lyme disease* in nature is a spatially explicit problem of current relevance, involving a number of interacting individuals constrained by differing spatial/territorial and temporal scales. Experimental data show the performance of the CMGVT, BFR and the load balancing algorithms when applied to the simulation of Lyme disease.

CHAPTER 1
INTRODUCTION

This research focuses on performance issues in optimistic Parallel Discrete Event Simulation (PDES) for spatially explicit problems. PDES is becoming an important simulation technique whose main advantage is that it allows one to intuitively model systems that have a discrete temporal nature. The greatest challenge in PDES is parallelization, since most Discrete Event Simulation systems are sequential in nature. The driving structure of the simulation is a priority-based event queue ordered by the time of the occurrence of the events. The simulation progresses as events are removed from the queue and processed.

The question might be posed: Can an inherently sequential computation be successfully parallelized? Yes—parallelism can often be extracted from the system, and speedup can be achieved. Speedup is not the only reason why one would want to parallelize applications. Parallel computers also increase the amount memory and cache available to the simulation. The memory of a single processor is usually not adequate to obtain significant results.

In sequential Discrete Event Simulation (DES) [1, 2], the physical system is modeled by a single Logical Process (LP) consisting of the process state, the clock, and the event queue. When the simulation is brought to a parallel or distributed architecture, the physical system is decomposed into several LPs, each with its own state, clock, and event queue. The challenge in managing multiple LPs is to preserve causality between events. The two main approaches developed to solve this challenge are based on conservative and optimistic protocols [3, 4]. The former prevents causality errors by limiting each LP's progress in time, so, although causality is guaranteed [5], tight synchronization between LPs is introduced. In Time Warp (TW) [6], the best known optimistic protocol, causality errors are allowed to occur; however, recovery requires rolling back the computation to the time just prior to such an error. Once the rollback is completed, the computation is restarted.

The simulation system described here is based on the Time Warp protocol.

This thesis addresses three research areas in Time Warp-based simulation: **effective synchronization** between Logical Processes, **minimizing the cost of rollbacks**, and **load balancing**.

In order to support rollback, it is necessary to save the constantly changing state information, as well as the incoming and outgoing messages. Therefore, the major drawback of the optimistic protocol is the amount of memory it requires. One way to reclaim memory is to determine which data residing in memory are no longer needed. This may be done by computing the Global Virtual Time (GVT)—the minimum Local Virtual Time (LVT) of all the LPs and of the timestamps of all the messages in transit. By definition, there are no events or messages in the system with a timestamp lower than the GVT, so all information (states and messages saved), that refers to times lower than the GVT can be removed from the system. A new GVT algorithm, which continuously computes a GVT estimate, is presented here. This algorithm, called the Continuously Monitored Global Virtual Time (CMGVT), keeps track of all messages in transit by appending information about them to event messages as well as to antimessages (messages used to cancel erroneous event messages). The algorithm distinguishes itself from other GVT algorithms in that it does not require special synchronization rounds in order to calculate the GVT. The CMGVT is computed on each process based on the information constantly available to it. Three versions of the algorithm have been developed. The difference between the versions is the amount of information that is sent between the LPs.

Partitioning a problem into Logical Processes is very important in PDES. It is advantageous to have few Logical Processes, because it facilitates the scheduling of the LPs and reduces context switching. However, when the size of the state of LPs is large, the rollbacks can be costly. When a rollback occurs at a given LP, the state of an entire LP has to be restored to the state just prior to the occurrence of a causality error. Alternatively, if incremental state saving is used, only the state variables affected by an event are saved. When a rollback occurs, the events that have been processed erroneously have to be undone, thus restoring the state variables. When an LP is large, many unnecessary events are rolled back. The new Breadth-First Rollback (BFR) presented in this thesis is designed specifically for spatially explicit problems, where the space is discretized into a lattice. BFR takes advantage of the scheduling benefits of a small number of LPs by partitioning the lattice into as many clusters as there are available processors and assigning an LP to each cluster. To reduce the impact that a rollback has on the cluster, BFR allows each node of the lattice to be rolled back individually. BFR has a dual view of an LP (lattice node cluster)—from the point of view of the future, the BFR views the cluster as one LP, and, from the point of past, each lattice node is viewed as an LP.

The final contribution to Time Warp-based PDES is the design of a load balancing algorithm. The general approach to load balancing in such simulations has been to migrate Logical Processes between processors. Here, because the system is designed exclusively for spatially explicit problems, load balancing can be performed by migrating space, along with the objects present in that space. The goal of the algorithm is to distribute the work evenly among the LPs, assuming that there is a one-to-one mapping between the LPs and the processors. The load migration makes use of BFR. Since the events at the lattice can be rolled back on a lattice node-by-node basis, the nodes can be easily migrated from one LP and incorporated into the next.

Although it is often attractive to choose a well understood problem as the basis for study of new algorithms, it is nevertheless more exciting to be able to reach for a "real world" application in the hopes of addressing not only pure computer science issues, but also practical issues of interest to other scientific fields. The following criteria were used in deciding on an appropriate application: the problem had to provide sufficient parallelism to make it interesting from the point of view of parallelization, as well as to be sufficiently dynamic to warrant the application and the investigation of various aspects of PDES, such as load balancing techniques. The simulation of the spread Lyme disease in nature incorporated both issues.

One might ask: Why would one want to use PDES to model Lyme disease? PDES allows the simulation of the problem in an individual-based manner. This technique allows one to reason about what kind of events can happen to an individual, unlike traditional ecological modeling, where differential equations, which describe system behavior, are simulated. In the differential equation approach, the more complicated the system becomes, the more complex the equations become, sometimes making them too hard to understand or to analyze. Individual-based simulation is much more intuitive and therefore easier to design and understand.

The goal of the research presented here is to design a PDES framework for simulating spatially explicit problems such as the simulation of the spread of Lyme disease in nature and to investigate and design algorithms that optimize the performance of such simulations.

Chapter 2 gives a general introduction to Parallel Discrete Event Simulation and its applications. Chapter 3 describes the simulation system in detail. Chapter 4 gives background in memory management algorithms developed for optimistic PDES. It describes various algorithms designed to reduce the amount of memory used by a Logical Process as well as some created to reclaim memory already used. Chapter 5 explores Global Virtual Time algorithms and describes the Continuously Monitored Global Virtual algorithm in its three variants. The performance of the CMGVT is empirically compared to that of a well known GVT algorithm. Chapter 6 describes the general class of spatially explicit problems and the Lyme disease application in some detail. Results of simulation runs are presented. Chapters 7 and 8 describe the Breadth First Rollback algorithm and its use in load balancing. The performance of the algorithms versus that of the traditional approach is demonstrated. Finally, Chapter 9 summarizes the contributions made in the field of optimistic Parallel Discrete Event Simulation and sketches possible future research directions.

CHAPTER 2
PARALLEL DISCRETE EVENT SIMULATION

2.1 Discrete Event Simulation

Many systems have a discrete nature. Such systems can contain instantaneous events such as flipping a light switch or being bitten by an insect. When such events are part of the modeled system, it is natural to use discrete time modeling techniques. Looking at the light switch case, the system can be in one of two states: one indicating that the light is on and the other that the light is off. There are two events, turning the light on and turning the light off. Each occurrence of the event may cause a state change. The time of the termination of the event is considered as the event's time of occurrence.

The general components of a discrete event simulation system are simulation objects, events, and a simulation engine, consisting of state variables, an event queue, and a clock. Events are kept in a priority queue, where the event with the lowest scheduled time has the highest priority. The simulation is started by introducing some initial events, which are inserted into the event queue. The simulation progresses as events are removed from the event queue and 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. For example, if the state of the light is *on*, and an event "turn light off" occurs, the state of the light will change to *off*. Processing of an event can also trigger new events. When an event is created, the time of the occurrence can be predicted based on a probability distribution for the type of the event. The newly created event is then placed in the priority queue based on the time of its occurrence.

2.2 Parallelization

The general concept of Parallel Discrete Event Simulation (PDES) is similar to the sequential version. The physical system being modeled is decomposed into several Logical Processes (LPs), each of which has its own clock, event queue, and state

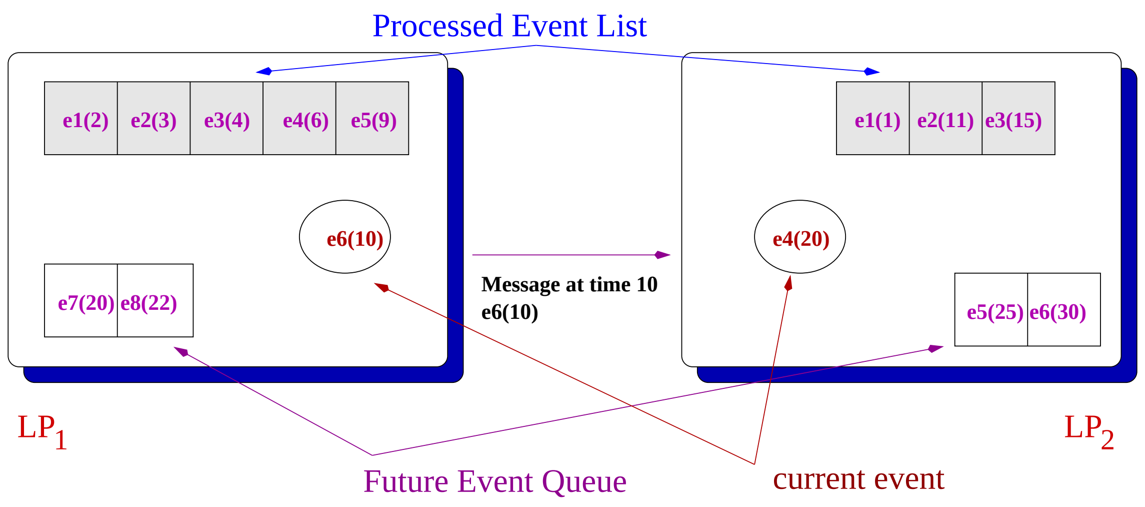


Figure 2.1: Causality Error in a Two Logical Process System.

variables. LPs communicate with each other via messages. During parallelization the problem of correctness arises. Each LP in the simulation must process events in a non-decreasing timestamp order, to preserve the causality relationship between events. Causality errors can arise because each LP advances at its own pace. Some LPs may be early in the simulation, some further. When an LP which progresses fast through the simulation time (local virtual time) and receives an event message from a "slower" LP, that message will be in its simulated past, indicating a possible causality error. Figure 2.1 depicts a causality error between two LPs. LP_1 has processed events $e_1 \ldots e_5$ and progressed to local virtual time 10. LP_1 then sends an event message (e_6 with timestamp 10) to LP_2. LP_2 processed its own events $e_1 \ldots e_3$ (now processing e_4) and is at local virtual time 20 at the time it receives the message e_6. Since the message is timestamped with time 10, a causality error occurs. The event contained in the message had to be processed before event e_2 at LP_2.

The two main approaches developed to solve the causality challenge are based on the conservative and optimistic protocols [3]. The former prevents causality errors by limiting each LP's progress in time, so although causality is guaranteed [5]. As a result, tight synchronization between LPs is introduced. The optimistic approach allows causality errors to occur; however, recovery requires rolling back the computation to the time just prior to such error [6]. Once the rollback is completed, the computation may be safely restarted.

2.3 Conservative Approach

The general idea behind the conservative approach [5, 3] is that LPs do not process any new events until there is a certainty that no event with a lower timestamp can arrive. The messages sent between LPs are assumed to be sent in chronological order. The receiving LP has preallocated buffers for each LP that can be sending it a message, and it reads off messages from each buffer in a FIFO manner. In order for the LP to process an event from the event queue, it has to check all incoming buffers. If all buffers contain messages with timestamps larger than the timestamp of the event in the event queue, then it is *safe* to process that event. The major problem is that two or more LPs can deadlock waiting for the others to send messages with timestamps sufficiently advanced. Figure 2.2 shows a deadlock situation. Here none of the LPs has any messages waiting to process. Each LP has a non-empty event queue, but each is unable to process an event. LP_1 cannot process event at time 12, because there might be a message coming from LP_3 at time t ($7 < t < 12$). LP_2 cannot process the event scheduled for time 17, because there might be an event coming from LP_1 at time t ($3 < t < 17$). Similarly LP_3 also cannot progress. Obviously, if one has a global snapshot of the system, LP_1 has the event with the smallest timestamp.

The problem of deadlock can be approached in two ways. One is to try to avoid deadlock, and the other is to recognize the deadlock and to break it. Avoiding deadlock can be done by sending null messages [5]. The null messages are not simulation events, they are used only for synchronization purposes. They contain the smallest possible timestamp of an event an LP can send to another. Of course one can immediately see that processes might be flooded with null messages. Deadlock can also be overcome by detecting it and breaking it.

2.3.1 Lookahead

Conservative simulations derive their good performance from *lookahead* [3], the ability to predict when an event will occur. Consider the example of a tan-

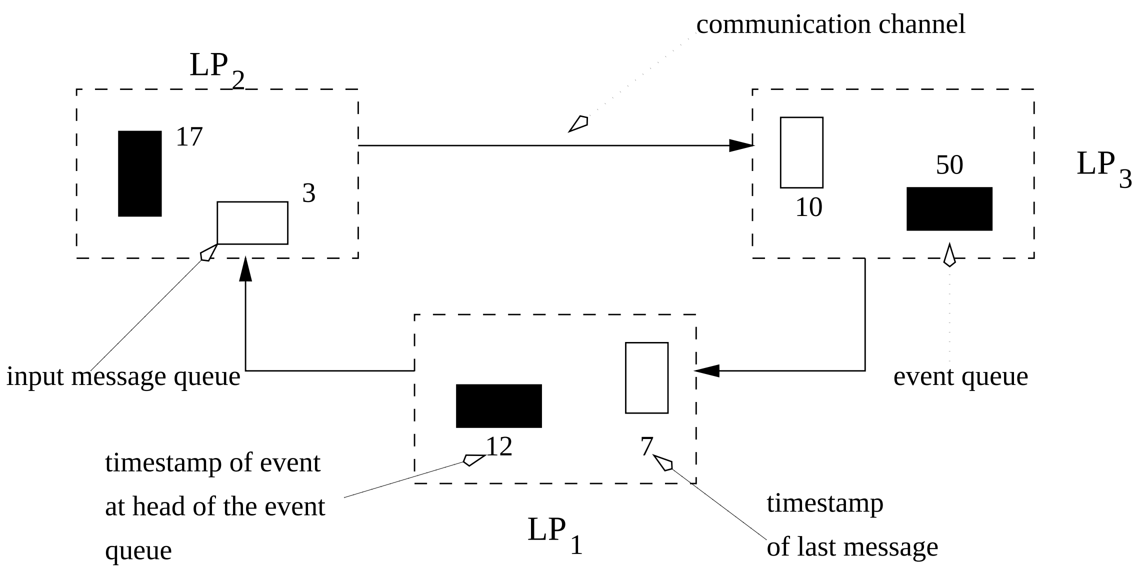

communication channel
LP2
17
3
50
10
LP3
input message queue
event queue
12
7
timestamp of event at head of the event queue
LP1
timestamp of last message

Figure 2.2: Deadlock Situation With 3 LPs Involved.

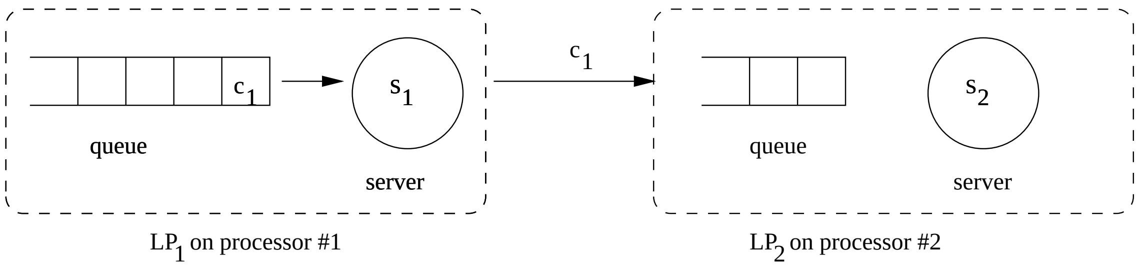


Figure 2.3: Two-server Tandem Queue.

dem queuing network (fig.2.3) with two servers s_1 and s_2 and two processors. LP_1 simulates the behavior of s_1 and LP_2 simulates the behavior of s_2. When a customer c_1 arrives at s_1 at time t_1, and departs by time t_2, LP_1 send a message to LP_2 containing the event of arrival of c_1 at time t_2. The event is then processed by LP_2. In such system, LP_1 can presample the service times for the customers, and when a customer arrives, it can immediately send an arrival message to LP_2. This way LP_2 can find out about the arrival before the customer is serviced at LP_1. This would be particularly useful if LP_2 could receive customers from other servers, where LP_2 would have to decide which event to process next. The major problem with lookahead is that it is very application specific. In ecological simulations, the time between events is very small, giving a small lookahead, and resulting in poor performance. Therefore the conservative approach was not chosen.

2.4 Optimistic Approach

Parallelism can be also achieved by using optimistic methods where causality errors rather than being avoided, are accepted and resolved using rollback. Each LP processes events according to their timestamp, and when an event e_s with a timestamp (t_s) smaller than the local clock arrives (that event is also known as a *straggler*), the computation is rolled back to time t_s. The computation is then restarted, and the event e_s is processed. The best known optimistic protocol is Time Warp [6]. The name Time Warp was used by Jefferson because the clocks of the LPs need not agree, and the LPs can go both forward and backward in simulation time.

There are several aspects to the rollback. To roll back the computation, one has to restore the system state to the state which was saved with a greatest simulation time smaller than the timestamp of the *straggler*. Also, the queue of incoming messages has to be restored. To support rollback, all the states of the computation leading up to the causality error, as well as all the messages that have been received have to be saved. Finally, the messages that have been sent since the time of the *straggler* have to be canceled, since they might be invalidated through the restarted computation. The messages canceling the initial (positive) messages are known as *antimessages*. When an LP receives an *antimessage*, it rolls back the computation to the time before the corresponding positive message was processed. The positive message is then canceled. If an *antimessage* arrives before the positive message has been processed, the positive message is annihilated. There is no possibility of the *domino effect* because the computation can roll back only to the point in time of the process which sent the initial *straggler* (the message that caused this particular rollback to occur).

2.5 PDES Applications

2.5.1 Research Applications

The most widely studied and implemented applications are queuing networks [7]. They are well suited to PDES because the time of processing an event is small compared to the overhead of synchronization and state saving. Therefore it is easy to analyze new GVT and state saving algorithms and to see where their performance degrades and maybe how to improve it. There are also other synthetic spatial applications, such as *Colliding Pucks* [8, 9], which have been used to develop simulation protocols. In Colliding Pucks, pucks move on a flat surface. The pucks can collide with each other and with obstacles. The surface is divided into sectors. Each sector is responsible for the movement of pucks in that sector. The *Sharks World* problem is another widely investigated system[10, 11]. Here sharks and fish swim in a toroidal world. When a shark gets close to a fish it eats it. Sharks can also die when they are unable to move for some period of time. Both *Sharks World* and *Colliding Pucks* display very dynamic behavior and spatial aspects, which have been used to analyze and stress test PDES protocols.

2.5.2 Spatially Explicit Applications

There are of course some more practical applications. PDES have been used to perform combat simulations [12, 13]. Two opposing armies are introduced into the field. One army is usually the aggressor and seeks out the opponent in order to attack it; the other army is usually the defender. Their tactics differ in that the attacker seeks out a weakness in the opponent, whereas the defender spreads out its troops evenly. When the armies come close to each other, they fight, and casualties are computed. “Armies” of ants have also been simulated [14]. Sources of food were distributed in an area. The ants followed the scent of food to its source. When they found the food, they ate it. New food could be added to continue the process. The simulation of *Personal Communications Services* (PCS) is of current interest [15, 16, 17, 18, 19]. PCS provide communication capabilities to cellular phone users. A geographical area is divided into small service areas, each of which has a number of radio ports, each with a certain number of channels. When a mobile phone customer, talking on the phone, moves from one service area to another, the call has to be transferred to a new channel in the new service area (a *handoff* occurs). The goal of the simulation is to establish how many cells and channels should be placed in a new area to be able guarantee good cellular phone service. A similar application is that of the National Airspace System [20]. The system simulates the behavior of the air traffic control system. The space is divided into three-dimensional sectors, and *handoffs* occur from one sector to another. Major airports are themselves sectors. Studying the average delay experienced by aircrafts is the main goal of this simulation.

2.5.3 Mapping

One issue not addressed thus far is how are the LPs created? What makes up an LP? How does one determine on which processor should a given LP reside? These are definitely questions that one has to answer when faced with designing a model for an application. In queuing systems, part of the answer is rather obvious: assign an LP to each server. As to how one should distribute LPs among processors, it is not so clear, so random solutions are often used. Each processor will receive an even number of randomly selected LPs. When simulating digital circuits, one also often assigns an LP to each gate. This is appropriate when the number of gates is small, but when the number increases into 100,000 and more, the idea of one LP per gate is less feasible. In this case it might be profitable to cluster gates together to form groups of gates, each of which is represented by a single LP [21]. Strongly connected components are used to determine such clusters.

A different decomposition and mapping problem appears in spatially explicit problems such as *Colliding Pucks*, *Shark World*, *Personal Communication Services*, or combat simulations. Here it is not appropriate to have an LP associated with each object such as a puck, and let pucks decide when they collide with each other, because of the complexity involved in each puck trying to figure out if it will collide with another puck (each puck would have to send its position and speed to all other pucks, making the communications very expensive). On the other hand, if an LP is assigned to the space, it becomes a sequential simulation. The solution is to subdivide the space. In *Colliding Pucks* the space is divided into sectors. At the beginning of the simulation each sector is responsible for the pucks and obstacles present in it. During the course of the simulation, the obstacles stay within their sectors, but the pucks move. A single puck, which does not encounter any obstacles or other pucks can move several sectors at a time. This poses a problem—should the pucks be transferred from sector to sector so that each sector is responsible only for the objects in its sector (this involves data movement); or should the messages go back to the object's home sector (this involves heavy communication). In *Colliding Pucks* load management heuristics move objects closer to the sectors with which they communicate with the most.

There are also applications where there are several decompositions, such as in the National Airspace System. The system simulates the airplane traffic in the United States. One decomposition is to divide the geographical area into cells. Each cell is then simulated by an LP. The airplanes are the objects moving around in space. Another decomposition would be to view each airport is an LP and objects such as airplanes are moved among them. Mapping of LPs onto processors is not clear due, for example, to different airports with varying traffic. Another issue making the mapping problem harder, besides different traffic load at the airports, are the different time zones. At the beginning of the day the airports in the eastern United States are very active, whereas the West Coast hubs are idle. Through the day, the air traffic load shifts towards the West. Also bad weather conditions can unexpectedly unbalance the computation, creating "hot spots" of heavy activity.

Finally, ecological simulations can be classified as spatially explicit problems [22]. Here, the domain is a geographical area. In that space objects are present (for example animals). The space is discretized into a lattice (here two-dimensional). Each node of the lattice can contain local information such as the type and amount of food or water available, temperature and humidity conditions. The objects move freely in that space, moving within a node and between them. A specific instance of the ecological simulation is the simulation of the spread of Lyme disease in the environment [23]. This application has motivated this work and is described in detail in chapter 6.

CHAPTER 3
COMPUTATIONAL MODEL

The simulation system is designed in an object oriented way using C++. The target architecture is the IBM SP2, an MIMD, distributed memory machine. The hardware configuration used in this work has 32 processors. The processes in that system communicate by sending and receiving messages. The communications between processes use the MPI [24] message passing library.

3.1 Problem Partitioning

Without *a priori* information about the configuration of objects and their events, an even distribution is assumed. Thus, it is desired that the space being simulated be divided equally among the Logical Processes. The objects in the space are assigned to the LP to which the space belongs.

First, the space is discretized into a lattice (in this work, two-dimensional space is simulated). Then the problem of partitioning the spatial lattice has to be considered. One approach is to model each lattice node by a single Logical Process. In that case, the "forward" processing overhead is high due to scheduling and context-switching costs incurred when the process with the event with the next smallest time is scheduled. However, there are also benefits of a one-to-one lattice node to LP mapping. For example, the cost of state saving is low, because only the state of one lattice node has to be saved at one time. Moreover rollback is also simple, because the state of only one lattice node has to be restored. A second approach, which minimizes the overhead in the forward execution, is to cluster lattice nodes. Each cluster is then mapped onto an LP. However, state saving is more complex in this case, because the state of the LP is composed of the states of several lattice nodes. Also, rollbacks are more costly, because, when a rollback occurs, the state that needs to be restored is large. Since, ultimately, the size of the simulations could be in the millions of lattice nodes, and assigning an LP to each node and managing this number of LPs would be very difficult and costly, the decision was made that

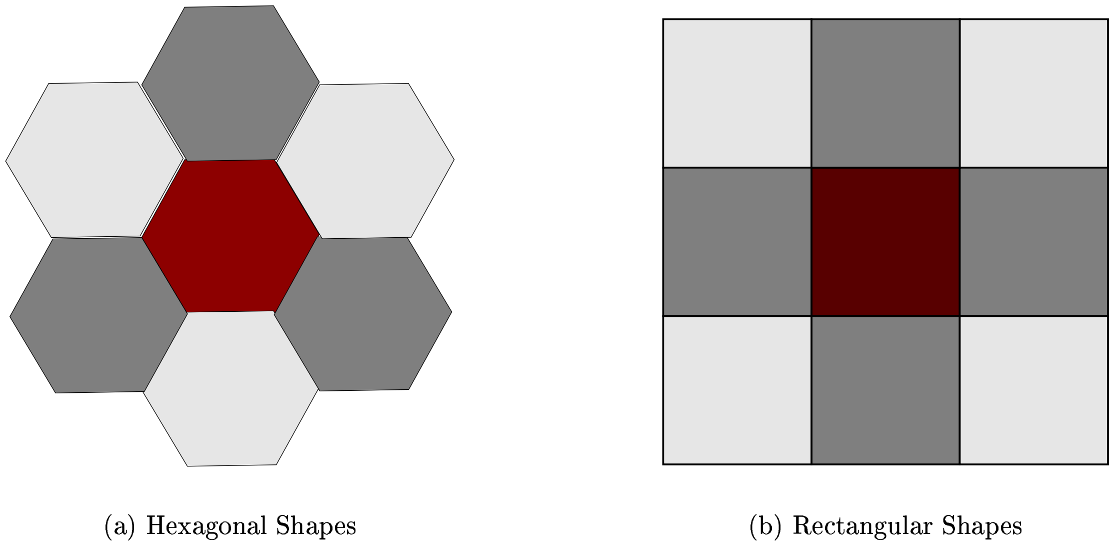
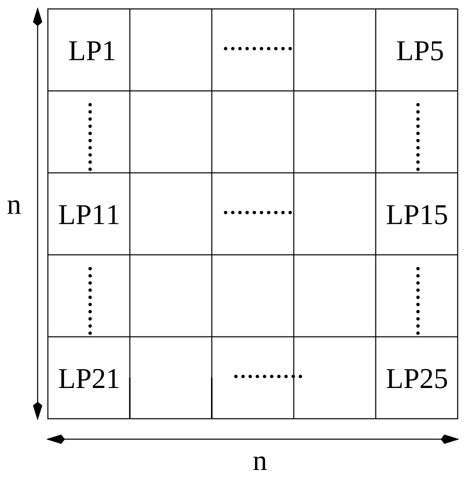

(a) Hexagonal Shapes (b) Rectangular Shapes

Figure 3.1: Common Lattice Node Shapes.

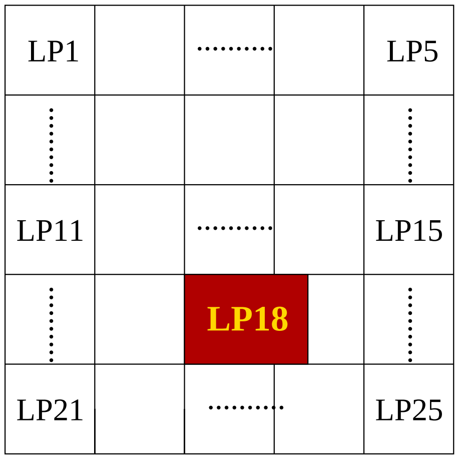
the lattice nodes would be clustered, with each cluster assigned to an LP. Initially, the space is divided into as many clusters as there are available processors.

Another issue to consider is how to shape the lattice nodes and the lattice clusters. A possible lattice node shape is that of a hexagon as shown in Figure 3.1(a). This geometry has been used, for example, in Personal Communication Services simulations [17]. Hexagonal shapes are attractive, because each element has only six neighbors (potential communicating processes). The problem is that, when clustered, the edges of the partition become ragged, and determining if a given area belongs to a given LP is complex. Another possibility for the lattice node shape is that of a rectangle, or, more specifically, a square as shown in Figure 3.1(b). In this case, the number of communicating processes grows to 8; however, the edges are more regular, and, when clustered, determining if a given lattice node is within a given LP is easy. In this work, the nodes have a square shape.

The issue of the shape of the cluster itself remains. If the lattice is divided in both dimensions into rectangles—Figure 3.2(a)—the size of the communicating edges is $4n/p$, where p is the number of processors and assuming the space is of the size n^2. This partitioning, however, does not lend itself well to dynamic load balancing of space assigned to an LP, because the movement of space boundaries will



(a) Rectangular Partitioning



(b) Rectangular Partitioning with Load Balancing

Figure 3.2: Common Partitioning Shapes.

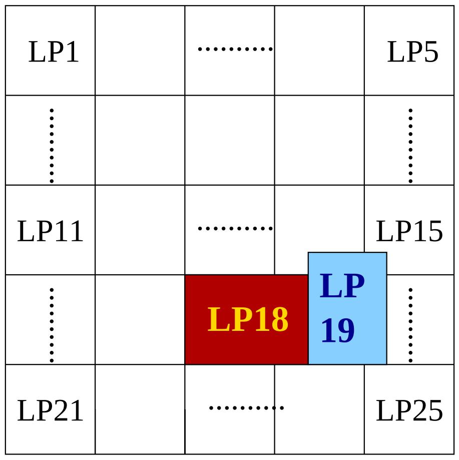


Figure 3.3: Irregular Partitions Shapes as the Result of Load Balancing.

cause irregular shapes—Figure 3.2(b). If the load balancing is performed by resizing the space for which a given LP is responsible, the following situation can occur: In Figure 3.2(b), LP_{18} acquired additional space as a result of the load migration. Therefore, if LP_{19} (see Figure 3.3) needs to resize, it can only enlarge either to the right or to the left to maintain the partitioning shapes. If LP_{19} wants to grow in the up or down direction, the partitioning becomes irregular (LP_{14} and LP_{15} in Figure 3.3).

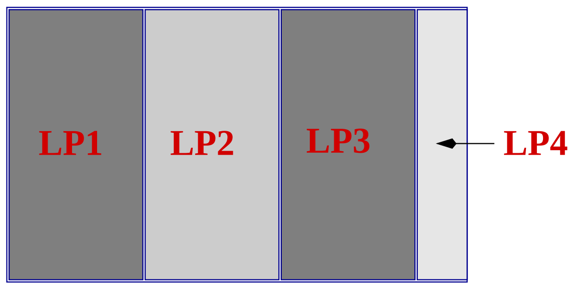


Figure 3.4: Strip Partitioning.

Because load balancing needs to be a part of the simulation system, a strip partitioning was chosen. The space is divided into strips in the dominant direction (Figure 3.4). Initially, the number of strips is equal to the number of available processors. During load balancing the strips can be easily resized. Another advantage of this partitioning is that each LP needs to communicate with only two others. The drawback is that the communication boundaries are larger ($2n$) than in the case of rectangular partitioning.

3.2 Data Structures

There are three major groups of objects in the simulation: spatial, mobile, and temporal (events). Each of these objects has the base class from which new classes can be derived. The *Space object* class contains basic information about a lattice node, such as its size and location, and the mobile objects present. A new class can be derived from the space object and contain additional information about that space, for example, the amount of food or water available. The base *Mobile object* class contains object location, age, id, and object type. For Lyme disease simulations, an animal object can be derived from the base class. This object will contain additional variables such as infection status, ticks present, dispersal status, and dispersal direction. The base *Event object* class is composed of:

- the event id
- the scheduled event time—the time at which the event was scheduled to happen
- the event time—the time of the occurrence of the event

- the trigger—the id of the event that caused it to be created
- the pointer to the object that the event will affect
- the location where the event is to occur
- the event status: processed or not

The important virtual functions associated with the *Event object* are *process* and *undo*. The *Move Event* and local events such as the *Kill Event* are derived from the base event class. Each of these events has its own process and undo functions appropriate for that specific class. If an event causes another event to happen, the triggering event will either keep the id of the dependent event or maintain a pointer to it. It is also assumed that the dependent event has to be scheduled for a time strictly greater than that of the triggering event.

3.3 State Saving

There are two possible choices for state saving: full or incremental [25, 26, 27]. In full state saving, the state of the entire LP is saved after each event or after some number of events (more details are included in Chapter 4). The state can also be saved incrementally [28, 29]. In the latter case, when an event is processed, the system saves the state variables this event modifies in its internal data structures. Upon rollback, the events are undone and the modified variables are restored. The advantage of incremental state saving is that it requires substantially less memory than full state saving. The disadvantage is that restoring the state on rollback (by undoing events) can be expensive.

In spatially explicit simulations, there are two general classes of events: local and non-local (specifically the *Move Event*). A local event can affect the state of the object at a given lattice node and the state of that node. The *Move Event* affects the state of the object and the state of two lattice nodes—the node from which the object is moving and the node to which the object is moving. Since the space assigned to an LP is large (at least thousands of nodes), the state of an LP is large, and a single event does not change much of that state. In the simulation system

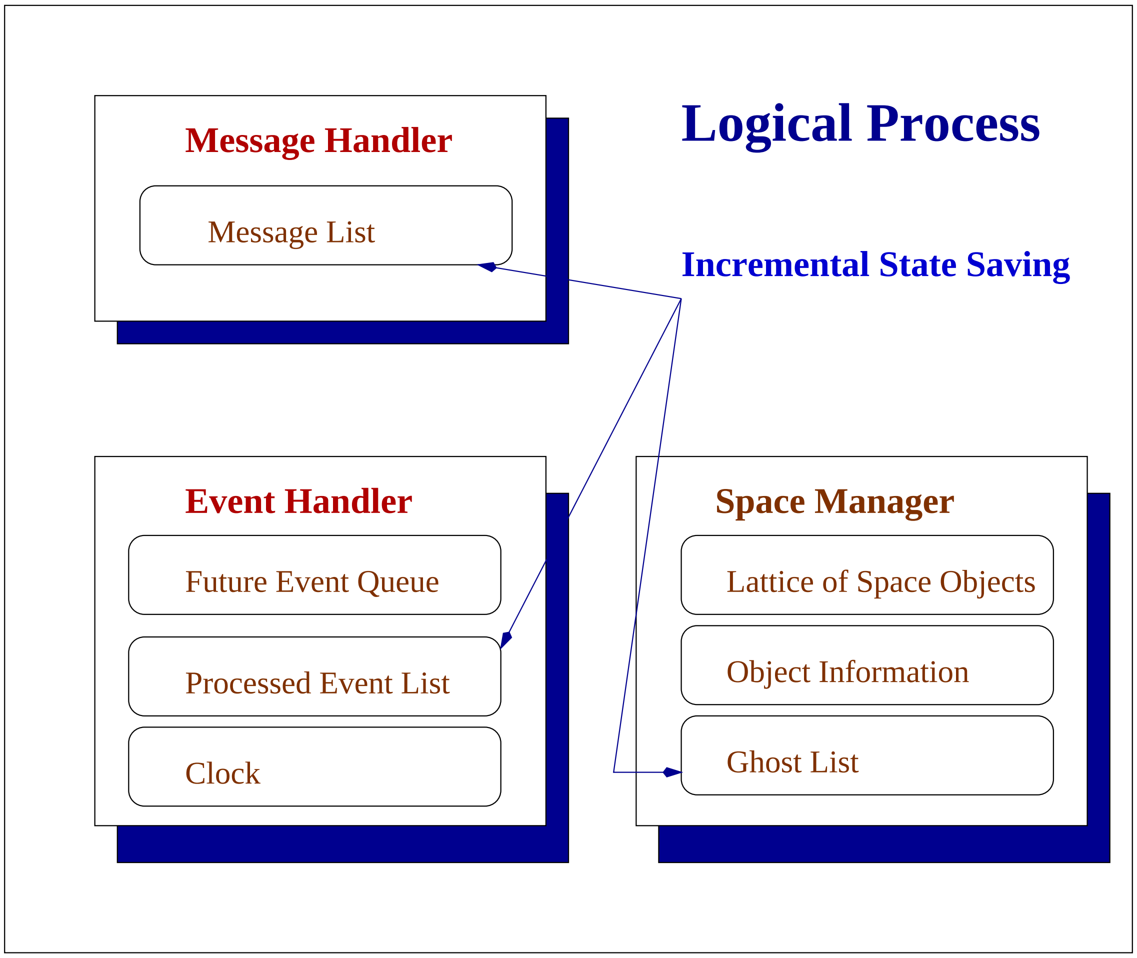


Figure 3.5: Structure of the Logical Process.

described here incremental state saving is used, because each event changes only a small part of the large state.

When a process runs out of memory, it first tries to start a new GVT calculation. If the GVT does not reclaim enough memory, then the LP rolls back to the GVT.

3.4 The Structure of a Logical Process

Each Logical Process is composed of three modules: the *Event Handler*, the *Message Handler* and the *Space Manager* (Figure 3.5).

The Event Handler contains the Future Event Queue, the *Processed Event List*, and the Clock, which keeps track of the Local Virtual Time. The Event Handler is responsible for the processing of all events. In the forward execution, it dequeues an event from the Future Event Queue and calls the appropriate process function for that event. When the processing is complete, the event is placed in the *Processed Event List*. The saving of modified state variables is made in the process function. Finally, the simulation clock is advanced to the time of the occurrence of the event.

Event Handler in Forward Execution

- `Remove event from the Future Event Queue`
- `EventPtr->process()`
- `Put the event on the Processed Event List`
- `Advance Clock`

Upon rollback, the reverse actions are performed. The event is removed from the *Processed Event List*, and the appropriate undo function is called. If the event was not canceled by an antimessage, it is placed on the Future Event Queue. The events that were triggered by the event being undone have to be canceled (removed from the Future Event Queue), because they are no longer valid. Finally, the simulation clock has to be set back.

Event Handler upon Rollback

```
• Remove event from the Processed Event List
• EventPtr->undo()
• Put the event on the Future Event Queue (if appropriate
• Delete dependent events from the Future Event Queue
• Set back Clock
```

The *Space Manager* is responsible for the movement of objects in space. When a *Move Event* occurs, the Space Manager removes the object from its current lattice node. If the object is moving to a lattice node within the same Logical Process, the object is simply placed at the new lattice node. At that time other information might be provided by the Space Manager. It can indicate whether the space is occupied or how much food is available. If the move is not local, the object and all its future events have to be sent to the new LP. The Space Manager collects the future events for the object from the Future Event Queue and passes them along with the object to the *Message Handler*. The Message Handler is in charge of sending and receiving messages between LPs. The Space Manager also places the objects and future events in the *Ghost List* in case the message has to be canceled.

Space Manager: Move Event Forward Execution

```
• Remove Object from current location
• if the Move is local
    – put the object at new location
    – return any additional information
• else
    – send object and its future events to new LP
    – put object and events on the Ghost List
```

When a Move Event is rolled back, the Space Manager steps in. If the object is present in the space assigned to the Local Process, the object is removed from the lattice node. If the object does not belong to the current LP, then the object was sent out to another process. The object can therefore be found on the Ghost List. The Space Manager removes the object from the list and restores the future events to the Future Event Queue. Finally, the object is returned to its previous location.

Space Manager: Move Event upon Rollback

```
• If the object is local
    – remove from current location
• else
    – move object from the Ghost List
    – restore events from the Ghost List
• Put object in previous location
```

When the *Message Handler* sends an event message to an LP, it places that message on the *Message List*. When a rollback occurs, the Message Handler removes messages from the Message List and sends corresponding antimessages to the appropriate LP. Then the Event Handler undoes the necessary messages.

This LP decomposition is unique to the work presented in this thesis. Usually, an LP is treated as a single entity, which has state variables, a priority-based event queue and a clock. Here, the LP is composed of an *Event Handler*, *Message Handler* and a *Space Manager*. This decomposition allows for the better understanding and modeling of the spatial aspects of the problem.

An object oriented language was chosen because it fits well with the logical decomposition of the model. It also facilitates the development a general purpose simulation engine. Because of the modularity of the program it is easy to incorporate new algorithms into the existing program. For the same reason the software is easy to understand and maintain.

3.5 Global Virtual Time Calculation

The Global Virtual Time calculation is a very important part of the simulation system, because it allows the recovery of memory that is no longer needed. Informally, the GVT is defined as the earliest event timestamp of an unprocessed event in the system, if the processes are synchronized. The GVT algorithm discussed here is based on the algorithm used in SPEEDES [30]. The general idea of this algorithm is to flush out all the messages that are present in the system. Besides the calculation phase itself, the LP has to decide when to start the calculation. The calculation can be scheduled to occur after a certain number of events have been processed, or after a given amount of simulated time has elapsed, or after some wall clock time has passed. Each LP can initiate the computation by sending a GVT message to its neighbor. Since, in this system, the topology of the LPs is essentially a bidirectional ring, the message is sent in an arbitrary direction. However, once the direction is established, the GVT messages have to follow that direction. The GVT message contains the number of messages and antimessages sent and received by the LP and its current Local Virtual Time (LVT). When an LP (other than the initiating LP) receives a GVT message, it adds its own send/receive counts to the incoming information, appends its own LVT, and sends the information to its neighbor. When an LP enters the GVT calculation phase, it stops sending event messages. Antimessages are still sent in order to clear out any impending rollbacks. If the initiating LP receives the GVT message, it checks whether the sum of messages sent is equal to the sum of messages received. If it is, then there are no more messages in the system, and the GVT is taken to be the minimum of the LVTs in the system. If the sums do not match, a new round of the GVT calculation is started. When the GVT is obtained, it is broadcast to all LPs in the system. When an LP receives the GVT information, *fossil collection* is initiated. During this phase, each LP traverses the lists that hold the past information (the Processed Event List, the Message List, and the Ghost List), and truncates them.

CHAPTER 4
MEMORY MANAGEMENT IN OPTIMISTIC PARALLEL DISCRETE EVENT SIMULATION

Time Warp is plagued by three problems: memory, memory, and memory. During the simulation, the Logical Process allocates dynamic data structured to save the state of the system, the incoming messages, and the outgoing messages. The LP consumes memory, increasingly so as the simulation progresses. It is possible the LP runs out of memory before reaching the end of the simulation. In shared memory systems, a single LP, running ahead of others can consume the entire memory available, thus preventing itself and the other LPs from continuing. In distributed memory systems, if a one-to-one LP-to-processor mapping is used, an LP can run out of memory, but that will not affect the progress of other LPs in the system. If a many-to-one LP-to-processor mapping is used (see section 6.5.1), the situation on a given processor is similar to the case for shared memory systems. In order to deal with memory problems, several approaches have been devised. They can be divided into two main groups: memory saving and memory reclaiming techniques.

4.1 Memory Reclaiming

An obvious way to regain memory is based on the Global Virtual Time (GVT) calculation. The GVT is the minimum virtual time of all the LPs and of the timestamps of all messages that have been sent but not yet received, and therefore rollbacks cannot occur to a time before the GVT. This property implies that all events that happened before the GVT have been committed, so any information that has been saved that refers to events before the GVT can be removed from the system, thus freeing memory (the process of memory reclaiming is known as *fossil collection*). Obviously, the GVT calculation and the subsequent fossil collection are applicable to both distributed and shared memory systems.

4.1.1 Message Sendback

When the GVT calculation fails to reclaim memory, which can easily happen when one LP falls far behind the others, the other LPs keep processing and saving states, and might eventually use all the available memory. Several solutions have been proposed [6]. One of them is to send back the messages that have been received (*message sendback*—a shared memory method) [31]. The sending back of the messages might not only cause the original sender to roll back, but it will also slow down an LP that might be sending too many messages. Obviously, one cannot send back messages whose timestamps are earlier than the GVT. When an LP that has run out of memory sends back a message, this message might roll back the original sender to the time of the send. If the original sender sent the message at time 20, progressed to time 100, and then received back its original message, it would have to roll back to time 20. It would thus free the memory it had used between times 20 and 100, hopefully allowing the LP that ran out of memory to continue. If an LP uses up all the available memory, and runs ahead, thus sending lots of messages, this method will slow it down by forcing it into rollback.

4.1.2 Artificial Rollback

One can also artificially roll back [32] (also for shared memory), which means to undo some computation—preferably enough to let the GVT advance and reclaim even more memory. The rollback allows reclaiming some memory, because the states and the output messages that were saved after the time to which the LP is rolling back can be discarded. The LP that is the farthest in virtual time is rolled back. How far to roll back is usually considered to be application-dependent, but some researchers propose to roll back to the time of the second farthest LP. For example, if there are LPs with virtual times $10, 20, 30$, and 100, then the LP with time 100 will roll back to virtual time 30. This process continues until enough memory is reclaimed. The outcome is that the fastest LP—the one that "ran away"—is slowed down, and memory is reclaimed. The GVT might advance, because, while the processes farthest ahead are rolling back, the processes that are behind have time to catch up. Even if the GVT does not advance immediately, the results can be beneficial, since memory is being reclaimed.

4.1.3 "All-In-One" Method

One can also use a mixture of the techniques described above. The idea proposed by Gafni [32, 33] picks one of the LP's resources (an input message, an output [output in the sense of output to another LP—not output device] message, or a state) and annihilates it. If an input message is being sent back, it might cause a local rollback, and maybe the rollback of the receiving process. This is analogous to *message sendback*. If an output message is picked, an *antimessage* is sent to cancel the positive message previously sent. The sending of an *antimessage* enables a local rollback. The sender has to roll back to the time before the event that caused the original message to be sent. This will free memory (this method is applicable to distributed memory systems). The *antimessage* can have either of two effects on the receiving LP. It can cause it to roll back if the original message has already been processed (thus freeing more memory). If the message is still waiting in the input queue, it is annihilated, and no further memory is reclaimed. If a state is to be canceled, the process rolls back to the time when that state was computed. This might also involve the rollback of other LPs, since, during the rollback, the LP will most likely send out *antimessages* corresponding to the original messages sent.

4.1.4 Probabilistic Synchronization

The MIMDIX system [34] is an operating system based on the optimistic protocol that provides system calls necessary for developing distributed simulations. Besides the usual system calls, it utilizes a *probabilistic synchronization* mechanism: at probabilistically computed intervals, the processes synchronize. When a process starts the synchronization phase (in this case it broadcasts to other processes), it sends its local virtual time to all other processes. When a process receives a synchronization message, it discards all positive and negative messages whose timestamp is larger than that of the sender. This allows easy reclaiming of memory from processes that are too far ahead. Probabilistic synchronization is useful both in shared and distributed memory systems.

4.1.5 Pruneback

Another way to deal with a *memory stall* (running out of memory) is to *prune back* some of the states [33]. The characteristic of this approach is that it leaves the process at its current *local virtual time*, whereas the techniques mentioned above do not. The choice of the states to delete is not obvious, but the general guidelines are that one should not remove the current state or the state at which the GVT was calculated. One of the benefits of this method is that it does not involve rollback. Memory is reclaimed by throwing away some of the states previously saved. Preiss *et al.* claim that the choice of which states to prune is implementation-dependent, but some of their empirical studies show that one out of every four states of an LP can be pruned. The disadvantage of this method, when compared to *artificial rollback* is that here one needs to prune more states to reclaim the same amount of memory. This is because artificial rollback, also deletes input and output messages. Preiss also notes that pruneback seeks to manage memory in an optimistic manner since it hopes that the states being pruned back will never be needed. However, if a rollback occurs to time t_i, and the corresponding state s_i has been pruned, the simulation has to roll back to s_{i-1} (if it exists), and restart from there. The computation from state s_{i-1} to s_i is known as *coasting forward*, since no messages are sent out (all messages previously sent between states s_{i-1} and s_i are correct).

4.2 Memory Reduction

A good place to save on memory is in the reduction of the state information saving. The question becomes: is there really a need to save the state of the system after each event? An argument in favor of doing so is that rolling back the system, to the time before an antimessage or a *straggler*, can be done quickly. In Figure 4.1, it can be seen that if an antimessage arrives for the event e_5, the system state can be restored to s_4 and the computation restarted.

4.2.1 Sparse Checkpointing

If the goal is to save some memory, it is possible, for example, to save only every fifth state (s_0,s_5,s_{10}). This cuts the memory requirements by a factor of five,

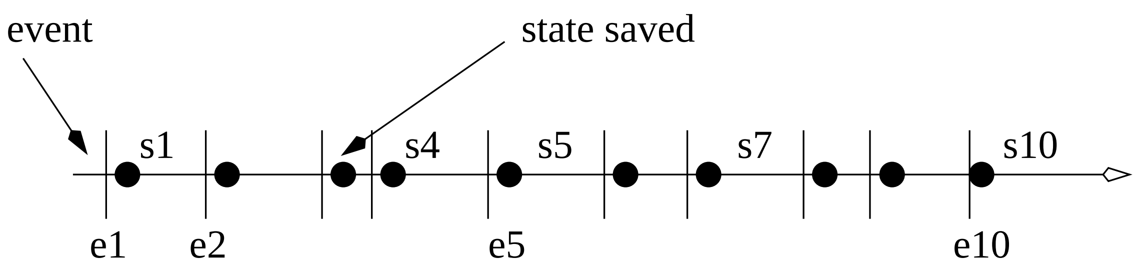


Figure 4.1: State Saving After Each Event.

but, of course, at a price. Now if an antimessage for the event e_5 is received, the state of the system has to be restored to s_0 and the simulation has to *coast forward* to the time before the event e_5. While *coasting forward*, computation is redone, but no messages are sent, since these have been previously sent out and are still correct. Only the state is modified. This method [35] causes more computation to be redone, and therefore might take more time than the saving of every state. The problem is a classic time/space tradeoff and research has been done on selecting the appropriate state-saving interval [36]. It has been shown that under certain circumstances it is preferable to reduce the amount of state saving even at the cost of incurring more rollback overhead. This method could be called a "preventive pruneback," because it anticipates that the LP might run out of memory, and does not save all the states of the computation. Pruneback, on the other hand, deletes states only when a memory problem occurs.

4.2.2 Incremental State Saving

A different approach is used in SPEEDES [25], in which a method called *incremental state saving* is introduced. Full states are not saved during the computation. Only the portions of the state that have changed due to the occurrence of the event are saved. When an event wants to change the state of a variable in the simulation, it does so by exchanging this variable with a similar variable which is stored in the data structure of the event. When a rollback occurs, the state of the variable that was changed by the event is restored. For a banking simulation, the state of the system would be the balance of every account carried by the bank. Even if some accounts are inactive, their balance would be saved every time an event occurs (or in the case of a different checkpointing interval, according to a corresponding schedule) if full state saving is used. That can prove to consume lots of memory, and is in this case wasteful. Incremental state saving, on the other hand, stores only new information. Instead of whole states, it only saves the events (or parts of them) in a *rollback queue*. When a rollback occurs, events are removed from that queue and undone. Incremental state saving is obviously very beneficial when changes in the system state are small. The disadvantage is that one cannot immediately roll back to the desired state.

4.3 Over-optimistic?

Interestingly, it is not always good to let the LPs process as fast as they can. As previously mentioned, run-away LPs can cause memory problems by consuming all the available memory. Another problem is that such “fast” LPs get hit with lots of rollbacks. If they are at virtual time 1,000 and the other LPs are at around 100, the “slow” LPs will most likely send messages to the “fast” LP and will roll it back to 100, undoing all the computation between time 100 and 1,000. Also, the “fast” LP will probably have sent lots of messages, which it will have to follow with *antimessages*, and the “slow” LPs will have to do a lot of bookkeeping.

4.3.1 Time Bucket Synchronization

Several methods have been developed that are geared towards reducing the amount of optimism (number of messages an LP processes into the future). The idea behind them is to keep the computation “near” the GVT. These methods include *Time Bucket Synchronization* [37], where LPs can only process events within a small interval time T past the GVT before synchronizing.

4.3.2 Window-Based Protocols

In some systems two time windows are used [38]. Dickens *et al.*, who already had a conservative protocol called YAWNS, tried to improve the system’s performance. Optimism was added to the system. The system originally had only a conservative time window, which started at the GVT and represented the interval during which the events could be processed “safely” without the threat of a *straggler*. To that window another, this time optimistic, window was added. The optimistic window extended from the conservative window into the future. At the end of the optimistic time window, processes had to synchronize. The new window added the penalty of incurring rollbacks due to *stragglers*; therefore, a rollback mechanism had to be added. The size of the optimistic window was determined to be application-dependent, set by the user. Empirical results have shown that the optimistic extension to the system performs better than its conservative counterpart as the number of LPs grows. However, no analysis had been done to show how this approach compares to the completely asynchronous one. It was pointed out [3] that windowing mechanisms might not only reduce the amount of incorrect computation but might actually impede the progress of correct computation.

CHAPTER 5
CONTINUOUSLY MONITORED GLOBAL VIRTUAL TIME

5.1 Global Virtual Time Algorithms

Since optimistic PDES save a considerable amount of data to support rollback, the Global Virtual Time algorithm is used to determine which information in the system is obsolete and can be discarded. The GVT is the minimum local virtual time of all the LPs and the timestamps of all messages sent but not yet received. There are two main strategies for finding that minimum. One is to halt the simulation, synchronize the process, and take the minimum. The second is define a consistent cut through the local virtual time of the processes and take the minimum of the messages sent but not yet received and of the local virtual time of the LPs (see sections 5.1.2). By definition, the computation cannot be rolled back beyond the GVT, because no events with a timestamp smaller than the GVT can be created or left unprocessed. Below, several approaches to the GVT calculation are described.

5.1.1 Tracking Messages in Transit

The major difficulty in the GVT calculation involves accounting for messages in transit. Even though all LPs might have an LVT $> t_x$, it is possible that a message with a timestamp $t_m < t_x$ has been sent but not yet received. Upon receipt of the message, the receiving LP will have to roll back to the last state saved just prior to t_m. To keep track of messages in transit, some approaches involve acknowledging every message received while keeping track of the messages that were sent but not acknowledged [39]. Each process keeps a list of messages it has sent. Upon receipt of a message, the receiving LP sends an acknowledgment to the original sender. When the sender receives the acknowledgment, it removes the corresponding message from its unacknowledged list. The GVT is calculated by gathering the local virtual times and the unacknowledged message list and taking the minimum of all LVTs and the timestamps of all unacknowledged messages.

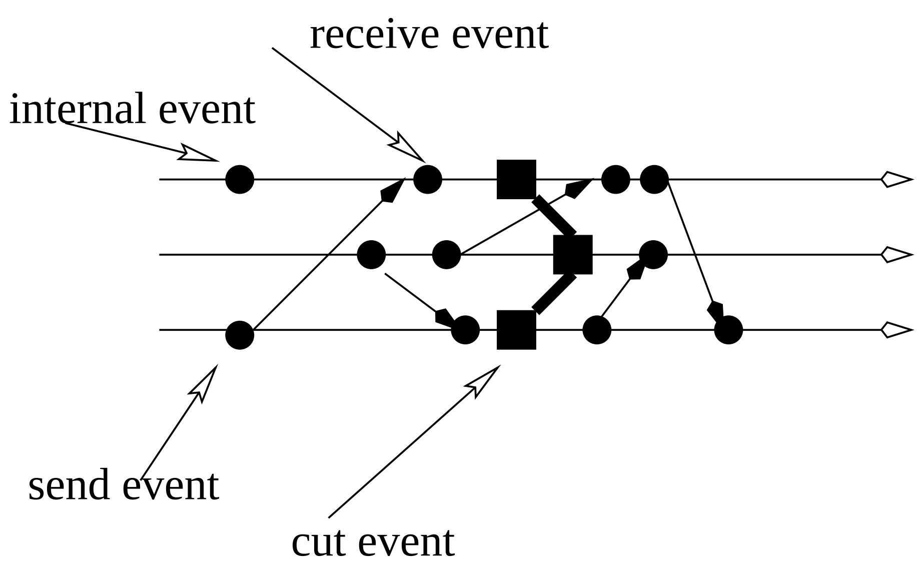


Figure 5.1: Time Diagram With Cut Events.

Another approach used in GVT calculations is to keep lists of messages sent and received [40]. These lists can be globally gathered and their difference determined. Then the minimum timestamp of the messages in transit and minimum of all local virtual times can be computed. Unfortunately, the lists and the messages carrying them can get very long.

If messages carry sequence numbers, they can be acknowledged by sending the highest consecutive message number received [41]. For example, assume LP_1 sends messages numbered from 1 to 50 to LP_2. Assume also that, when LP_2 receives a control message that signals the start of the GVT calculation, it already received messages numbered from 1 to 45. In such a case, LP_2 needs only to send an acknowledgment to message 45 informing LP_1 that messages 46 to 50 are still in transit. Based on this information, LP_1 can compute the minimum local virtual time as the minimum timestamp of unacknowledged messages.

5.1.2 Cut Based Algorithm

There are also ways of calculating the GVT by generating cuts through the time diagrams [41]. Figure 5.1 shows such a cut. Each horizontal line represents CPU time on each processor. The cut is generated by inserting *cut events* into the LP's input queue. The *cut events* are only control events—they are used to signal

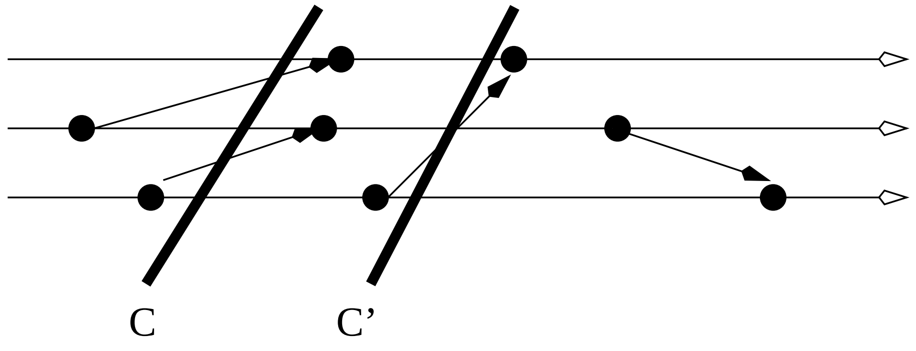


Figure 5.2: Time Diagram With 2 Cuts.

the LP that a global snapshot is being taken. Mattern [41] creates two cuts close together in the time diagram (C and C') (Figure.5.2). At time C',the minimum of all timestamps sent after C gives the lower bound on the timestamps in transit at C'. Initially every process is white. A process is colored red to the right of the cut C (by the cut event). Every process counts the number of white messages it sends and receives (the messages are colored with the sender’s color). After cut C (when the processors are red), every process remembers the minimum timestamp of all red messages it sends. At C' the number of white messages sent and received is gathered, and the minima of red messages are also collected. If all the white messages that have been sent have been received, the minimum time of red messages is equal or less than the GVT. If not all white messages have been received, another cut has to be found.

5.1.3 Barrier Synchronization

To aid with the GVT calculation, an optimistic barrier was presented by Nicol [42]. The barrier permits optimistic entry, but does not allow processes to go past it optimistically. This means that a process can enter the barrier based on its currently processed event having its simulation time equal to or larger than the time of the barrier. Such a process can then be rolled back by a straggler or an antimessage. This process is not allowed to go through the barrier unless it has sent and received all the messages up to the time of the barrier. The algorithm relies on a tree structure to synchronize processes. First, the processes synchronize pairwise, then the *leader* of the pair synchronizes with the *leader* of another pair and so on until the root of the tree is reached and all processes have been synchronized. Message counts are kept in order to see if there are any messages in transit. If at any point a process fails to synchronize, for example if the process did not receive all the messages that were sent to it, the LP leaves the barrier to receive the outstanding messages. When the process's counterpart does not synchronize, a time out is enforced. The process can leave the barrier, in which case the process can reenter the barrier at the level it has left it.

A centralized message tracking algorithm was proposed by Bauer [43]. In this algorithm, the processes send information messages to a central process. The messages contain information about what messages were sent and received via communication channels that are predetermined and set at the beginning of the simulation. The central process combines the available information and redistributes its knowledge back to the processes. This approach, however, suffers from a communication bottleneck, since one process will be flooded with incoming messages.

5.1.4 Computation During GVT Calculation

In the SPEEDES system, the GVT is calculated, but the simulation is not halted [30]. When this calculation is initiated, the processes enter a *risk free* mode, in which, although they continue to process local events, they do not send any positive messages; however, antimessages are sent in order to minimize impending rollbacks. During the GVT calculation phase a rollback may happen on any process, and it may cause other processes to roll back. Possibly, a process which has sent its LVT and unacknowledged message information, receives an antimessage. In this case, the process will rollback and send out antimessages. However, these rollbacks would occur anyway, because the antimessages have already been sent or will be sent after the GVT calculation has been completed. On the other hand, in this phase, the LPs will not produce new events for the other processes. We have implemented this algorithm in our system and used it as a basis of comparison with our CMGVT.

5.1.5 Target Virtual Time

Although all the algorithms discussed so far focused on calculating the GVT, a different point of view was presented in [44], where a *TVT* (Target Virtual Time) was proposed. The idea is to determine when a process has crossed that time. In the algorithm, the initiating process sends a *Target Event* to every other process in the system. The scheduled time of the event is the TVT. This event is treated as any other event would be, so that it can be enqueued into the input queue or it can cause a rollback. When the Target Event is processed, the LP sends a *report* to the initiating process. In that report the LP includes the number of messages sent and received between the previous GVT and the TVT, and the number of rollbacks incurred. The initiator collects all the reports, and if the sum of all messages sent equals the sum of all messages received, the TVT becomes the new GVT.

5.2 Continuously Monitored Global Virtual Time

The *Continuously Monitored Global Virtual Time* (CMGVT) algorithm [45] is designed to monitor the progress of the simulation by using locally available information as well as by keeping track of the message traffic. All messages contain a serial number. Messages are assumed to arrive in the order that they were sent. For example, if a process A sends two messages (1 and then 2) to process B, then process B will first receive message 1, then message 2. This assumption holds true for most message-passing environments. No acknowledgment messages are sent out. The general idea behind the algorithm is to propagate through the system the information about the LVT of the processes and about all the messages being sent and received. This is achieved by making an LP append to the event messages and antimessages its knowledge about the LVT of the LPs in the system, the number of messages that were sent by all of them, and the messages in transit. Indirect knowledge—the knowledge the sender has about the knowledge of the neighboring LPs about the system is also included. Direct and indirect knowledge is used to infer which messages are still in the system. Once a process receives a message from another process, it knows at least as much about the system as the sender does.

The idea of piggy-backing system information has been used before in distributed systems. The *vector clock* [46, 47], which consists of a vector with a size equal to the number of processes, describes the logical progress of each process in the system. This clock can be used, for example, for causal ordering of messages in a distributed environment. A *matrix clock* [48, 49], which is represented by a $p \times p$ matrix, where p is the number of processes, describes the knowledge that a particular process has about the knowledge all the processes in the system have about each other. The matrix clock is mostly used to discard obsolete system information[50].

These clocks are inadequate for optimistic PDES because they rely on the assumption that logical clocks can only move forward. In optimistic PDES, the LVT can also move backward, due to rollback. There is, however, one measure of the simulation that is monotonically increasing: the number of messages being sent by each process. The CMGVT's "logical clock" keeps track of the knowledge of the number of the messages sent in the system, and, in order to maintain the knowledge about the LVT of all processes as well as the knowledge of the messages in transit, an additional data structure, described below, is supported.

The CMGVT uses two basic structures, which are maintained locally by each process: the *Message Matrix* and the *Table of Forcing Vectors.*

5.2.1 Main Data Structures

The **Message Matrix** (MM): an entry (j, k) in MM_i, belonging to LP_i, represents the knowledge that LP_i has about the knowledge LP_j has about the total number of messages that LP_k has sent. Not all the entries in the MM are present. Only the knowledge each LP has about its logical neighbors is needed (LP's with which the process is communicating). Only one row of the matrix contains all the entries (row i in MM_i). This row describes the knowledge LP_i has about the entire system. Thus, the size of the matrix is $p + (p-1) \times K$, where p is the number of processes and K is the number of logical connections each process has to others (usually $K \ll p$, and often $K = O(1)$). For simplicity of explanation, the algorithm for the case in which $K = p$ is described; i.e., it is assumed below that all processes are connected with one another.

$$MM_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 5 & 2 & 1 \\ 0 & 3 & 2 & 0 \\ 0 & 2 & 1 & 1 \end{bmatrix} \tag{5.1}$$

The sample MM matrix (Eqn. 5.1) is located on LP_1 (by convention processes are counted starting from 0) and is denoted by MM_1. The system has only four LPs. The entry (1,1) with the value 5, denoted by $MM_1(1,1)$, indicates that LP_1 knows that LP_1, (in this case itself) has sent out five messages. LP_1 also knows that LP_0 has sent no messages $(MM_1(1,0))$, LP_2 two messages $(MM_1(1,2))$, and LP_3 one message $(MM_1(1,3))$.

The MM also contains the knowledge of the owner process (here LP_1) about the knowledge that other processes had when they last communicated with LP_1. Row 0 describes the knowledge of LP_0 about processes LP_0 to LP_3. Clearly, here LP_1 does not have any information about LP_0's knowledge of the system. However, MM_i provides information about the knowledge of processes LP_2 and LP_3. For example, LP_1 knows that LP_2 knew of three messages sent by $LP_1(MM_1(2,1))$, two messages $(MM_1(2,2))$ sent by itself, and none sent by LP_0 $(MM_1(2,0))$ or LP_3 $(MM_1(2,3))$. The Message Matrix does not provide any information about which messages have been accounted for. This information is maintained in a *Table of Forcing Vectors* (TFV).

A *forcing* $(F(t,n,c))$ represents the basic information about a message, and is composed of three data: the virtual time t at which the event in the message is scheduled to happen, the identity n of a process sending the message, and the current outgoing message count c on that process. This tuple was named a forcing, because, if the message represented by the forcing is received by and LP with an lvt higher than the virtual time in the forcing, than that LP will be *forced* to roll back. Each process has a table (the TFV) indexed by the LP id. The entries in this table contain the current known logical virtual time of each LP and the Forcing Vectors for each process. The forcings form the components of the vectors. If a forcing is placed at entry i in the table, it means the message represented by the forcing was sent to LP_i. It is possible that an LP receives information about an incoming message before it receives the message itself.

$$TFV_1 = \begin{bmatrix} 0 & [F(10,1,4), F(10,2,1)] \\ 17 & \\ 19 & [F(7,1,2)] \\ 26 & [F(9,1,3)] \end{bmatrix} \tag{5.2}$$

A simple TFV is shown in Eqn. 5.2. This TFV belongs to LP_1. Forcing $F(10,1,4)$ represents a message sent by LP_1 to LP_0; the message has the serial number 4 and was sent at virtual time 10. The second entry in this row, $F(10,2,1)$ indicates that LP_1 also knows that LP_2 has sent its first message to LP_0 at time 10. These messages are unacknowledged, because LP_1 has no information about LP_0 (see Eqn. 5.1). It can be seen that LP_1 sent its second message to LP_2 at time 7 and the third at time 9 to LP_3. TFV_1 also shows that LP_1 thinks that LP_2's LVT is 19 and that LP_3's is 26. Unless LP_2 rolls back before receiving the message from LP_1, it will have to roll back to the state prior to time 7.

Theoretically, the forcing vectors can grow without limit, but they can be easily bounded to a finite size at the cost of increased communication. In the system described here, if the forcing vector gets too long for a certain LP, a query message is sent to it. Upon receipt of this message, the queried process sends an answer message containing up-to-date local information.

At the beginning of the simulation the MM has all entries equal to zero, the LVT for every entry in the table is set to the simulation start time, and no forcings are present. When a process sends or receives an event message or antimessage, it updates its MM and TFV.

5.2.2 Update on Send $LP_i \rightarrow LP_j$

```
1. MM_i(i,i)++

2. add F($t_s$,i,MM_i(i,i)) to TFV_i[j]

3. TFV_i[i].lvt = $t_s$
```

Every time processes communicate, the sender sends along its Message Matrix and its Table of Forcing Vectors. Since the major cost of sending a message is the initiation of the send, adding extra information to the message does not considerably increase the cost of sending the message. The table and the matrix are updated just before the send operation is performed. When LP_i sends a message to LP_j with timestamp t_s, $\mathrm{MM}_i(i,i)$ is incremented by one. A new forcing is also created—$\mathrm{F}(t_s, i, c)$, where c is the current outgoing message count ($c = \mathrm{MM}_i(i,i)$). This vector is added to the entry j in the TFV_i. The current LVT of LP_i is inserted into the TFV_i at entry i. The MM_i and the TFV_i are appended to the message being sent from LP_i to LP_j.

5.2.3 Update on Receive $LP_j \rightarrow LP_i$

```
Updating the Forcing Vector:

                For every entry $k$ in the $TFV_i$
1. for every Forcing $F(t,n,c)$ in Forcing Vector ($TFV_i[k]$)
   if ($MM_j(k,n) \geq c$)
    remove $F$ from $TFV_i[k]$
   (message acknowledged, info given by sender)

2. for every new Forcing $F(t,n,c)$ in Forcing Vector ($TFV_j[k]$)
   if ($MM_i(k,n) < c$)
    add $F$ to $TFV_i[k]$
   (message not acknowledged, info provided by receiver)
```

Upon receipt of a message, the local Message Matrix and the Table of Forcing Vectors have to be updated based upon the new information received. The TFV is updated first. Updating the TFV involves checking the forcings in each vector against the Message Matrix. The local forcings are checked against the incoming Message Matrix, and the incoming vector's forcings are checked against the local Message Matrix. Of course, the incoming forcing $F(t, j, c)$ at entry i, where c is the current message number, is automatically acknowledged since it refers to the current message. The unacknowledged forcings are entered into the local Table of Forcing Vectors at the appropriate entries, and the acknowledged forcings are discarded. The results remain in the local data structures.

Consider the TFV at process i with a non-zero vector length at entry $k \neq i$ and $k \neq j$, where LP_j is the sender. Let this entry contain a forcing $F(t_x, i, c_x)$. This means that a message with timestamp t_x and serial number c_x was sent by LP_i to LP_k. Since the forcing is still present in TFV_i, LP_i does not know if LP_k knows about that message. The question is: Does LP_j have any information indicating that LP_k knows about the message (i.e., whether LP_k received the message)? To answer this question, the incoming Message Matrix (MM_j) is analyzed. Assume that $MM_j(k, i) = c_y$, so LP_k knows of at least c_y messages that LP_i has sent. If $c_y \geq c_x$, it implies that LP_k knows of the message described by the forcing $F(t_x, i, c_x)$. This forcing can be discarded since it has been acknowledged (indirectly) by LP_k. If, however, $c_y < c_x$, it means that, to LP_j's knowledge, LP_k did not receive the message. The forcing then has to remain in the table.

The LVTs present in the TFV are updated based upon which process has the most up-to-date information. This is determined based upon the number of messages of which each process is aware. The more messages a process knows about, the more recent its information is. The LVT for entry k is taken from the process which knows about the most messages sent by the process LP_k.

Updating the *Local Virtual Times:*

1. $TFV_i[i].lvt$ `is unchanged since the receiver knows its LVT best.`
2. $TFV_i[j].lvt = TFV_j[j].lvt$, `update the sender's LVT based on the sender's information.`
3. `for all other entries in the` TFV_i

```
(if sender knows more about $k$ than the receiver)
if ($MM_j[j,k] > MM_i[i,k]$)
 $TFV_i[k].lvt = TFV_j[k].lvt$
```

Updating the Message Matrix:

1. `Update what` i `knows about` j`:` $MM_i(i,j) = MM_j(j,j)$
2. `Update what` i `knows about others based on what` j `knows about others:` $\forall_{0 \leq k < p \wedge k \neq i}$

   $MM_i(i,k) = max(MM_i(i,k), MM_j(j,k))$
3. `Update the knowledge that` i `has about the knowledge of others:`

   $\forall_{0 \leq k,l < p, \wedge k \neq i}$

   $MM_i(k,l) = max(MM_i(k,l), MM_j(k,l))$

Next, the local Message Matrix has to be updated. First, LP_i's knowledge about LP_j has to be updated (step 1 above). Then, LP_i's knowledge about other LPs is compared to LP_j's information about the others (step 2). The rest of the entries are also updated based upon the most current knowledge (step 3). If LP_j knows more about LP_k's knowledge than LP_i knows about it, then $MM_j(k,x) > MM_i(k,x)$. Hence, $MM_i(k,x)$ is updated to the most recent value ($MM_j(k,x)$). Steps 1-3 are separated purely for clarity of description. In the implementation all three steps can be collapsed into one.

Most of the work is done in maintaining the MM and TFV data structures. Thus the GVT calculation is very simple: an LP just takes the minimum of all the LVTs and the minimum of all the forcings in the TFV. Obviously, the GVTs calculated by each of the LPs in the system need not be the same, since their information about the system will most likely be different. Also, it is up to each LP to decide when it wants to perform the GVT calculation. It can do so periodically, or only when on the verge of running out of memory.

5.3 Proof of Correctness

To prove that the CMGVT is a valid estimate of the GVT, assume that each process maintains two clocks:

1. a logical clock, c, counting the number of sent and received messages. c is updated each time a new message is sent or received (this is a slight change from the previous section which just simplifies the description of the proof, but is otherwise irrelevant for the implementation).

2. a simulation clock, t_s, which shows the simulated time of the event that was most recently processed. t_s is updated after processing each event or after sending or receiving a message.

For convenience, we assume that if the message is sent or received, then both c and t_s are changed at the same instant at which the event, brought in or sent out by the message, has been processed. With this assumption, it is clear that c is monotonically increasing and for each value of c there is a sequence of simulation time values $t_1(c), \ldots, t_k(c)$ that are assumed during the simulation (as a result of processing local events) while the process holds its logical clock at c. Because causal ordering is preserved during local event processing, then $i < j$ implies that $t_i(c) \leq t_j(c)$. Let $t(c)$ denote the smallest value in this sequence. Then $t(c) = t_1(c)$.

Let vector $C = [c_0, \ldots, c_{p-1}]$ represent a state of the simulation. For each process i there is a set of messages $F_i(C)$ that were sent by state C but not received at this state yet (more formally, if a message $m =< j, c'_j, i, t_s >$ from process j to process i is sent with logical clock c'_j and received at logical clock c'_i then $m \in F_i(C)$

if and only if $c'_j \leq c_j$ and $c'_i > c_i$). Let $f_i(C)$ be the minimum simulation time carried by messages in F_i. The simulation state C corresponds to many different vectors of simulation time on participating processes. Let $mGVT(C)$ denote the minimum GVT for all the vectors that can arise in state C. From monotonicity of the sequence $t_1(c), \ldots, t_k(c)$ it follows that $mGVT(C)$ is the GVT for the state of the computation defined by the simulation time vector $t(C)$.

Below, we prove that $mGVT(C) = \min_{0 \leq i \leq p-1}(t(c_i), f_i(C)) \geq LVT(C, i)$, where $LVT(C, i)$ is an estimate of the GVT produced by the described algorithm on process i. We will proceed by induction over the vector C.

1. For $C = \mathbf{0} = [0, \ldots, 0]$ we have $LVT(\mathbf{0}, i) = 0 = \min_{0 \leq i \leq p-1}(t(c_i), f_i(C)) = mGVT(\mathbf{0})$.

2. Assume that the relation holds for state $C = [c_0, \ldots, c_{p-1}]$ and that the simulation progresses to a new state $C' = [c'_0, \ldots, c'_{p-1}]$ in which process i, $0 \leq i \leq p-1$ changes its logical clock, so for all $j \neq i$ $c'_j = c_j$ and $c'_i = c_i + 1$. By definition of $\min_{0 \leq i \leq p-1}(t(c'_i), f_i(C'))$ there is a process or an undelivered message with such a time, so $mGVT(C') \leq \min_{0 \leq i \leq p-1}(t(c'_i), f_i(C'))$. To prove equality we just need to show that $\min_{0 \leq i \leq p-1}(t(c'_i), f_i(C')) \geq \min_{0 \leq i \leq p-1}(t(c_i), f_i(C))$. Two cases need to be considered:

   Process i sent the message. Let the message sent out be $m =< i, j, c_i, t_s >$. Clearly $F_j(C') = F_j(C) \cup m$ and for all other processes (including i) their F sets were not changed. Similarly, for all processes $l \neq i$ $t(c'_l) = t(c_l)$. Because $t(c_i + 1) = t_s \geq t(c_i)$ and $f_j(C') \geq t_s \geq t(c_i)$, then

   $$\min_{0 \leq l \neq i \leq p-1}(t(c'_l), f_l(C')) \geq \min_{0 \leq l \neq i \leq p-1}(t(c_l), f_l(C))$$

   Process i received the message. Let the received message be $m =< j, i, c''_j, t_s >$, where $c''_j \leq c_j$. It is clear that $m \in F_i(C)$ and $m \notin F_i(C')$ and no other set F_l was changed. Also, i is the only process on which $t(c'_l)$ changed. However, $t(c'_i) \geq \min(t(c_i), f_i(C))$. Let $mGVT_i(C) = \min_{0 \leq l \neq i \leq p-1}(t(c_l), f_l(C))$. Clearly,

$mGVT_i(C) = mGVT_i(C')$. Thus,

$$\min_{0 \le l \ne i \le p-1} (t(c'_l), f_l(C')) = \min(t(c_i{+}1), f_i(C'), mGVT_i(C)) \ge qmGVT(C).$$

From inspection of the algorithm it is clear that for all $j \ne i$, $LVT(C', j) = LVT(C, j)$. Since $mGVT(C') \ge mGVT(C)$, as shown above, we need to consider only the relation of $LVT(C', i)$ to $mGVT(C')$. If $LVT(C', i) = LVT(C, i)$ then the inductive step is proven; otherwise, assume that

$$LVT(C', i) > mGVT(C'). \tag{5.3}$$

Let $k_l(c_i)$ be the logical clock of process l known to process i at state c_i and, similarly, $m_l(c_i)$ be the minimum simulation time of the forcings imposed by process l and known to process i by time c_i. From assumption (5.3), for each $l \ne i$ $t(k_l(c'_i)) = t(k_l(c_i)) > mGVT(C')$, and $m_l(c_i) > mGVT(C')$. By definition of $mGVT(C')$, the message not received at state C' that carries the smallest simulation time stamp tm_s must have $tm_s \le mGVT(C')$, and it must be sent by process i. Indeed, assuming otherwise implies that all messages sent or received by the processes at logical times $c \in [1, c_l]$ carry a simulation time stamp $> mGVT(C')$, contradicting the definition of $mGVT$[1]. By inspecting the algorithm, it is clear[2] that such a message must be present in the forcings of process i so $m_i(c_i) \le mGVT(C')$, contradicting assumption (5.3).

5.4 Example

To demonstrate the algorithm in action, consider the following scenario. In each of the following tables, the first column represents the simulation step on each LP, the second represents the LVT, and the last represents an activity (e for event processing, $x \rightarrow y$ for a message being sent from x to y, and $x \leftarrow y$ for x receiving a message from y). The timestamp of each event sent in the message is equal to the

[1]For messages sent at $c \in [1, k_l(c_i)]$, this follows from forcings known to i, and, for messages sent at $c[k_l(c_i)+1, c_l]$, this conclusion follows from $t(k_l(c_i)) \ge LVT(C')$.

[2]The only messages that are removed from the forcing set of process i are those that are known to be delivered (see Section 5.2.3).

local LVT.

LP_0

step	LVT	action
0	0	e
1	2	0→2
2	3	0→1
3	4	e
4	9	0←2

LP_1

step	LVT	action
0	0	e
1	1	e
2	3	1→0
3	5	1←0
4	6	1←2

LP_2

step	LVT	action
0	0	e
1	10	2→0
2	15	2→1
3	20	2←0
4	2	e

Let's look at what is happening on LP_1:

At step 0 no messages have been sent and the local virtual times are initialized to 0 (Eqn. 5.4).

$$MM_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, TFV_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.4}$$

At step 2, after sending a message to LP_0, the "1" in the matrix (Eqn. 5.5) shows one message sent by LP_1 and the *forcing* $F(3,1,1)$ at entry 0 represents the message sent by LP_1 to LP_0 at virtual time 3 (and the fact that it is LP_1's first message sent).

$$MM_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, TFV_1 = \begin{bmatrix} 0 & F(3,1,1) \\ 3 & \\ 0 & \end{bmatrix} \tag{5.5}$$

At step 3, LP_1 receives a message from LP_0. That message contains MM_0 and TFV_0 (Eqn. 5.7). They correspond to the MM_0 and TFV_0 at step 2 on LP_0. Entry (0,0) in MM_0 shows that LP_0 has sent two messages. Since these messages are not acknowledged, they are represented by the two forcings in TFV_0. The local forcings (in TFV_1, Eqn. 5.6) are compared to the incoming knowledge (MM_0, Eqn.5.7). $F(3,1,1)$ is checked against entry $MM_0(0,1) = 0$. Since it indicates that LP_0 does not know of the first message sent by LP_1, the forcing remains in TFV_1. Now the forcings in TFV_0 are compared to the knowledge that LP_1 has about the system (MM_1). $F(3,0,2)$ is automatically removed, since it represents the current message. $F(2,0,1)$ is placed in TFV_1 because entry $MM_1(2,0) = 0$ shows that LP_2 knows of no messages sent by 0. The LVTs are also updated based on the most recent information. Since the message came from LP_0, $TFV_1[0].lvt$ is set to $TFV_0[0].lvt = 3$. The coefficients in MM_1 are just the maximum of the corresponding coefficients of MM_0 and MM_1. $newMM_1$ shows that LP_1 knows that LP_0 sent out two messages. This gives LP_1 the updated TFV ($newTFV_1$) and updated MM ($newMM_1$) (Eqn. 5.8).

$$MM_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} TFV_1 = \begin{bmatrix} 0 & F(3,1,1) \\ 5 & \\ 0 & \end{bmatrix} \tag{5.6}$$

$$MM_0 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} TFV_0 = \begin{bmatrix} 3 & \\ 0 & F(3,0,2) \\ 0 & F(2,0,1) \end{bmatrix} \tag{5.7}$$

$$newMM_1 = \begin{bmatrix} 2 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$newTFV_1 = \begin{bmatrix} 3 & F(3,1,1) \\ 5 & \\ 0 & F(2,0,1) \end{bmatrix} \tag{5.8}$$

At step 4, after receiving a message from LP_2, MM_1 is as in Eqn. 5.8, and TFV_1, TFV_2 and MM_2 are shown in Eqn. 5.9. Updates of MM_1 and TFV_1 are performed. $F(3,1,1)$ is checked against $MM_2(0,1) = 0$ and $F(2,0,1)$ against $MM_2(2,0) = 0$. Both forcings stay since neither LP_0 nor LP_2 know of these incoming messages. The incoming forcings are compared against the local MM. Comparing $F(10,2,1)$ against $MM_1(0,2)$ leaves it unacknowledged. $F(15,2,2)$ is the current message and therefore is automatically acknowledged. The LVT in the $TFV_1[2].lvt$ is updated to the most recent information of 15. MM_1 contains the maximum of coefficients of MM_1 and MM_2. The updated structures are shown in Eqn. 5.10.

$$MM_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 2 \end{bmatrix} \quad TFV_1 = \begin{bmatrix} 3 & F(3,1,1) \\ 6 & \\ 0 & F(2,0,1) \end{bmatrix}$$

$$TFV_2 = \begin{bmatrix} 0 & F(10,2,1) \\ 0 & F(15,2,2) \\ 15 & \end{bmatrix} \tag{5.9}$$

$$newMM_1 = \begin{bmatrix} 2 & 0 & 0 \\ 2 & 1 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$

$$newTFV_1 = \begin{bmatrix} 3 & F(3,1,1) & F(10,2,1) \\ 6 & & \\ 15 & F(2,0,1) & \end{bmatrix} \tag{5.10}$$

Eqn. 5.10 shows that the LVTs provided by the LPs are not sufficient to calculate the GVT—the forcings need to be taken into consideration. In the described scenario, the GVT is 2, because LP_2 has a forcing for that time.

5.5 Space Complexity

Obviously, only forcings for communicating processes are ever created. If there is a forcing $F(t, x, c)$ at entry y in the TFV_i, it can be removed only when LP_x knows how many messages LP_y has sent $(MM_i(x, y))$. This implies that only entries indicating communicating processes are needed in the MM, so there must be K entries in each row, where K is the connectivity of a process. If, for example, LP_1 communicates with LP_4, the entries $MM(1, 4)$ and $MM(4, 1)$ are needed. When processes do not communicate, the corresponding entries in the matrix are 0. Additionally, for each LP, the entry $MM(x, x)$ is needed to maintain the number of messages sent by that process which allows the owner of MM to update what it knows about the sender. Additionally the full row i in matrix MM_i is needed to enable the owner of MM to update the LVT information. In summary, the Message Matrix is of size $K \times (p-1) + p$, where K is the connectivity of an LP. In the case of two-dimensional spatial problems, K is often less or equal to eight.

The size of the TFV is $(m+1) \times p$, where m is the maximum number of forcings in a vector and is a parameter of the simulation. Thus the size of the additional overhead introduced by the CMGVT is $(K + m + 2) \times p - K = O(mp)$.

5.6 Three Levels of Knowledge

To fully explore the effect of the amount of knowledge sent between the LPs on the quality of the GVT estimate, three versions of the CMGVT algorithm were investigated [51]. The algorithm described above contains the **TRANSITIVE** knowledge an LP has about the system. The knowledge of the full system is sent between processes. This allows non-communicating processes to find out about each other through an intermediary process. In the second version, **INDIRECT**, only the knowledge that each LP has about its direct neighbors is gathered and communicated. Each LP appends the entire MM as above and the LVTs of the other LPs are also included, but only the forcings for its neighbors are added (in effect, these are the forcings representing the outstanding messages the LP sent to its neighbors). This implies that non-communicating processes do not know directly about their respective outstanding messages. The third version, **DIRECT**, contains the least amount of information. Only the LPs knowledge about itself is included. Here the forcings are maintained locally by the LP, but are not sent. Only the minimum time of the forcings is sent along with the LVT information. The MM is now only a vector (row i on LP_i), representing the number of messages the sender is aware of. The performance of these three versions is presented.

5.7 Performance of the Three Versions

First, the runtime differences of the algorithm's three versions are presented. Intuitively, the TRANSITIVE version, which contains most information and message processing, is expected to take longer to run than the INDIRECT and DIRECT versions. This prediction was tested with three different message loads (small, medium and large). The medium message load is 1.3 times greater than the small load, and the large load is 2.3 times greater than the small message load. In all three cases (Figures 5.3, 5.4, 5.5) the DIRECT approach performed best, followed by the INDIRECT and TRANSITIVE versions. When less than 12 processors are used, the algorithms performed similarly.

In order to analyze how well the algorithms were able to estimate the GVT, the closeness of the GVT estimate to the LVT was analyzed. This difference is significant, because it is directly proportional to the amount of memory needed by an LP. When the difference is large, significant amounts of memory are necessary to hold state information. The following curves represent the average difference between the LVT and the GVT of the system LPs. Again three different message loads were considered. From Figure 5.6, it can seen that when the LPs communicate infrequently, the TRANSITIVE version performs best. This is because the larger amount of information contained in the message allows the GVT to be estimated more accurately. When the message load increases (Figures 5.7 and 5.8), the differences become smaller.

It is interesting to note that Figures 5.7 and 5.8 indicate that the INDIRECT and DIRECT versions are not easy to compare. Sometimes the former is better, other times the latter. Such varied performance is caused by the finite size of the forcing vectors. In the DIRECT method, an LP that keeps sending messages without receiving acknowledgments adds forcings to its local data structure. As a result, the forcing vector grows too long and the LP queries the recipients of the messages directly. Consequently, such an LP will receive the most up-to-date information from the recipients and will have a good GVT estimate. When the flow of messages increases, the need for update diminishes and therefore the performance is more consistent.

When choosing a version of the CMGVT algorithm, one has to take into account memory requirements. When there is enough memory available, the DIRECT approach is the fastest. However, if memory is a constraint, the TRANSITIVE version will give better results. Another important factor in the good performance of the DIRECT method is the fact that the interactions in the simulated system are local with nearest-neighbor communications. If the communications were farther reaching, the inference factor of the TRANSITIVE method might play a bigger role in giving a good GVT estimate.

5.8 Comparison of CMGVT and SPEEDES

The performance of the DIRECT CMGVT algorithm is also presented in comparison to the algorithm used in SPEEDES described in sections 3.5 and 5.1.4. In

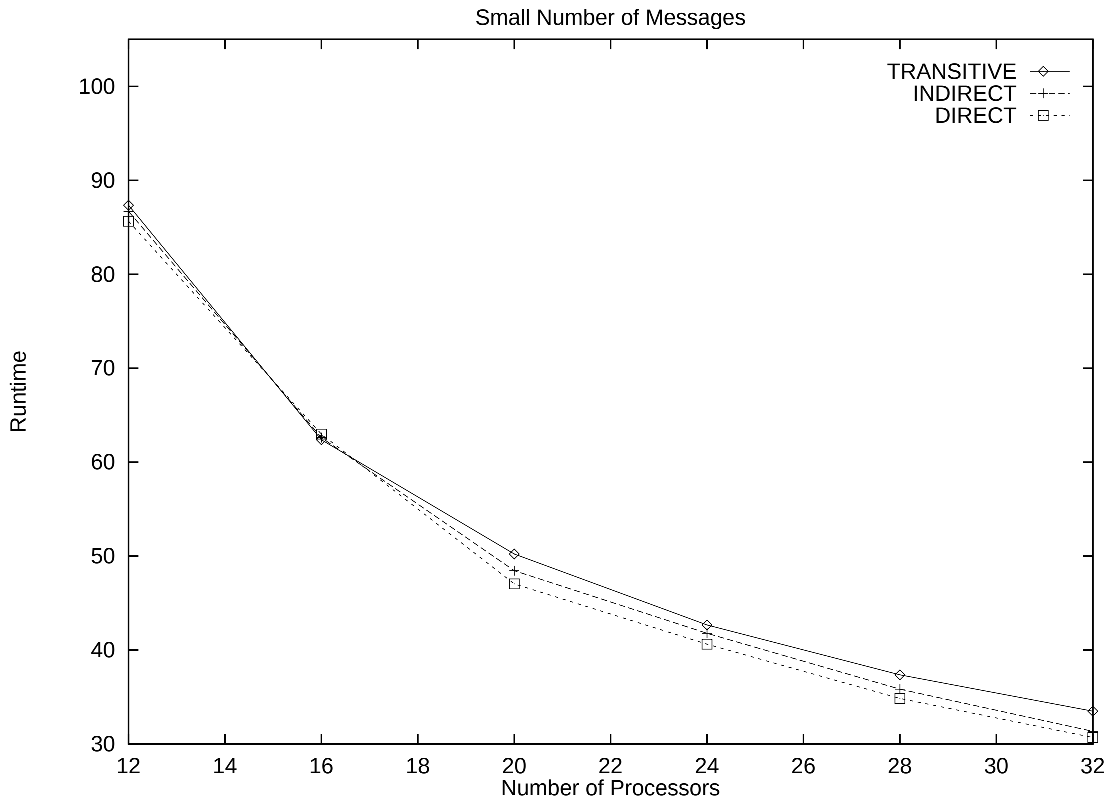


Figure 5.3: Runtime with a Small Number of Messages.

SPEEDES, the GVT is calculated by flushing the messages out of the system and then exchanging LVT information among the processes. The CMGVT does not use any synchronization rounds; it does not need to interrupt the flow of the simulation. Instead it relies on the information locally available. In the results presented here, the computational and communication load is distributed equally among processes. The one-to-one process to processor mapping is used. Performance of the algorithms is presented with a medium message load.

Figure 5.9 shows the runtime for both algorithms with a relatively small problem size (48,000 node lattice and 12,000 mice). The performance of the two methods is similar; however, there are slight differences. The speedup curves (Figure 5.10) show that SPEEDES performs slightly better for a small data set. When the number of processors is increased beyond 20, the performance of both algorithms degrades because the communication to computation ratio increases (the size of space assigned to each process decreases).

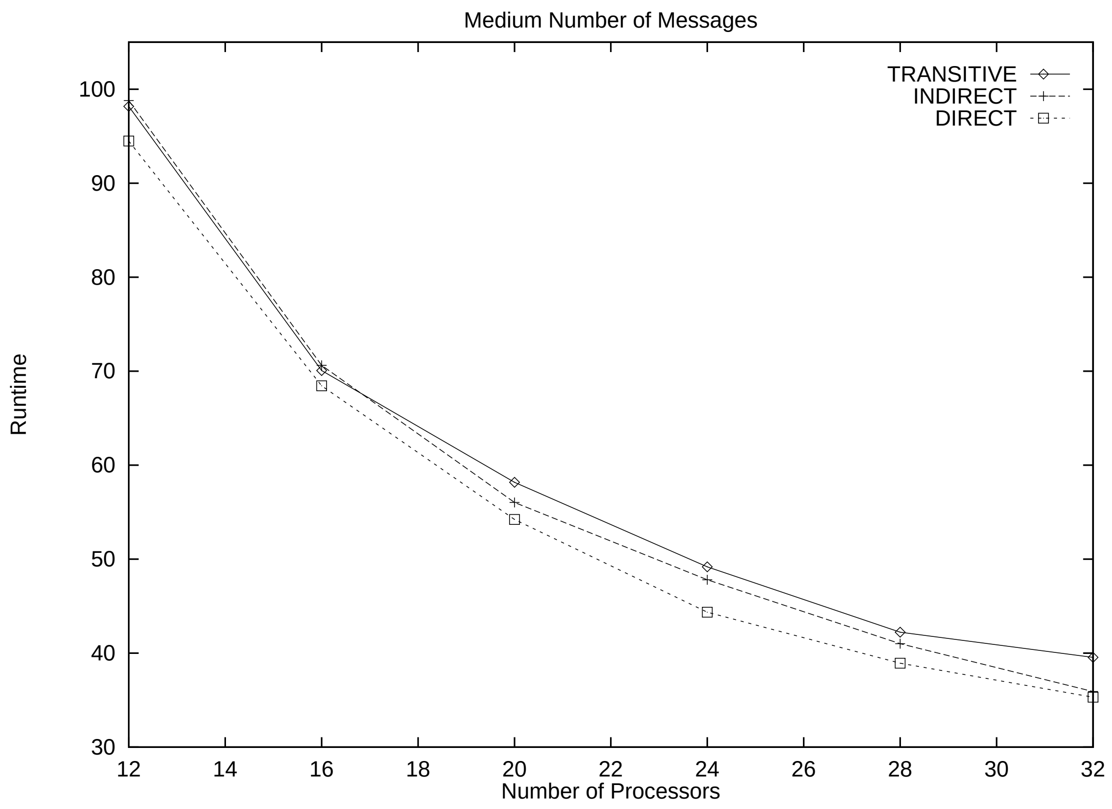


Figure 5.4: Runtime with a Medium Number of Messages.

In order to be able to demonstrate the performance of the algorithms with a higher number of processors, the problem size has to be increased [52]. Figure 5.11 shows the runtime for SPEEDES and the CMGVT with about 200,000 nodes and almost 50,000 mice. The graph shows that the SPEEDES algorithm performs better than the CMGVT up to 20 processors. However when the number of processors increases to 24, 28 and 32, the runtime of the CMGVT continues to decrease, whereas the SPEEDES runtime starts to level off. This shows that the increased message size due to the additional Message Matrix and the Table of Forcing Vectors has a smaller detrimental effect on performance than the increased synchronization time imposed by the larger number of processors.

It is important to mention the role of the number of messages sent in the system. The above results were obtained for a system in which processes exchange messages with each of their neighbors continuously throughout the simulation. This is typical of the spatially explicit problems that are being studied here. However,

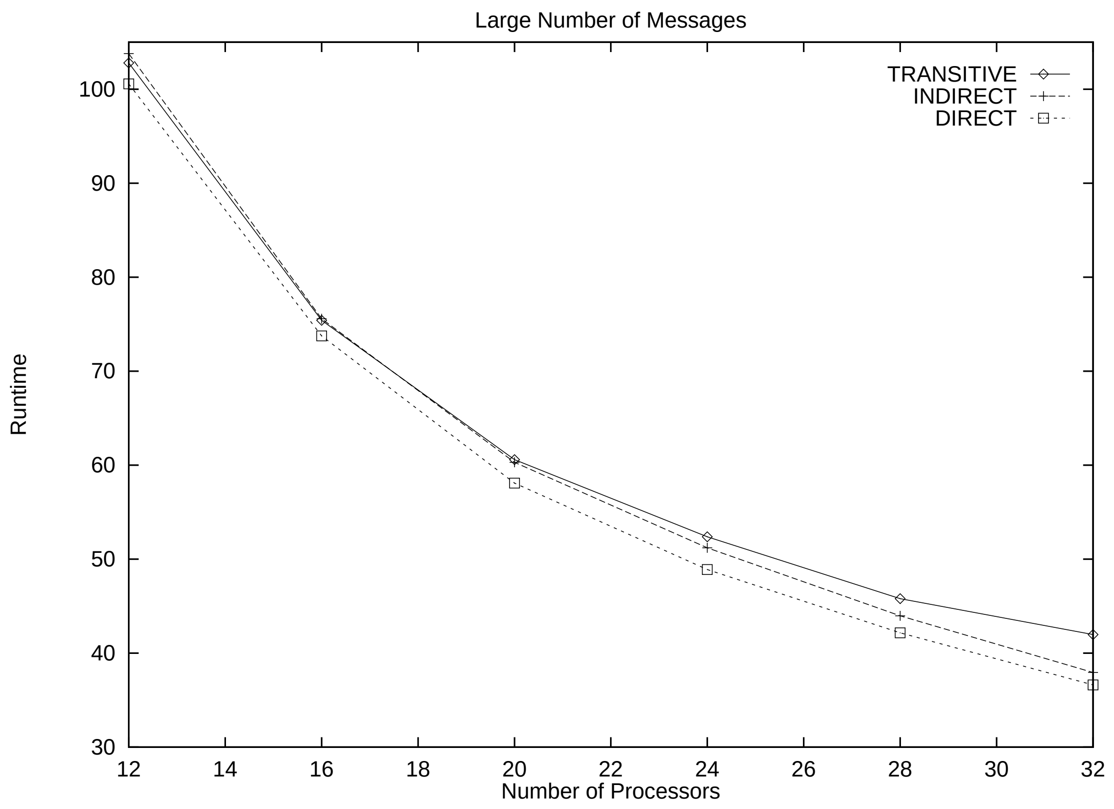


Figure 5.5: Runtime with a Large Number of Messages.

it is possible to design a system in which processes do not communicate frequently. In such systems, the SPEEDES algorithm will probably perform better. Since few messages are being sent, they can be flushed out quickly. In the case of the CMGVT, due to the lack of incoming information, the processes have to query each other in order to be able to update the local data structures. The querying process might therefore slow down the progress of the simulation.

In conclusion, the SPEEDES GVT algorithm is suitable for systems where the processes communicate infrequently. When the processes are communicating with each other, the CMGVT is an appropriate algorithm to use, especially when more than 20 processors are used.

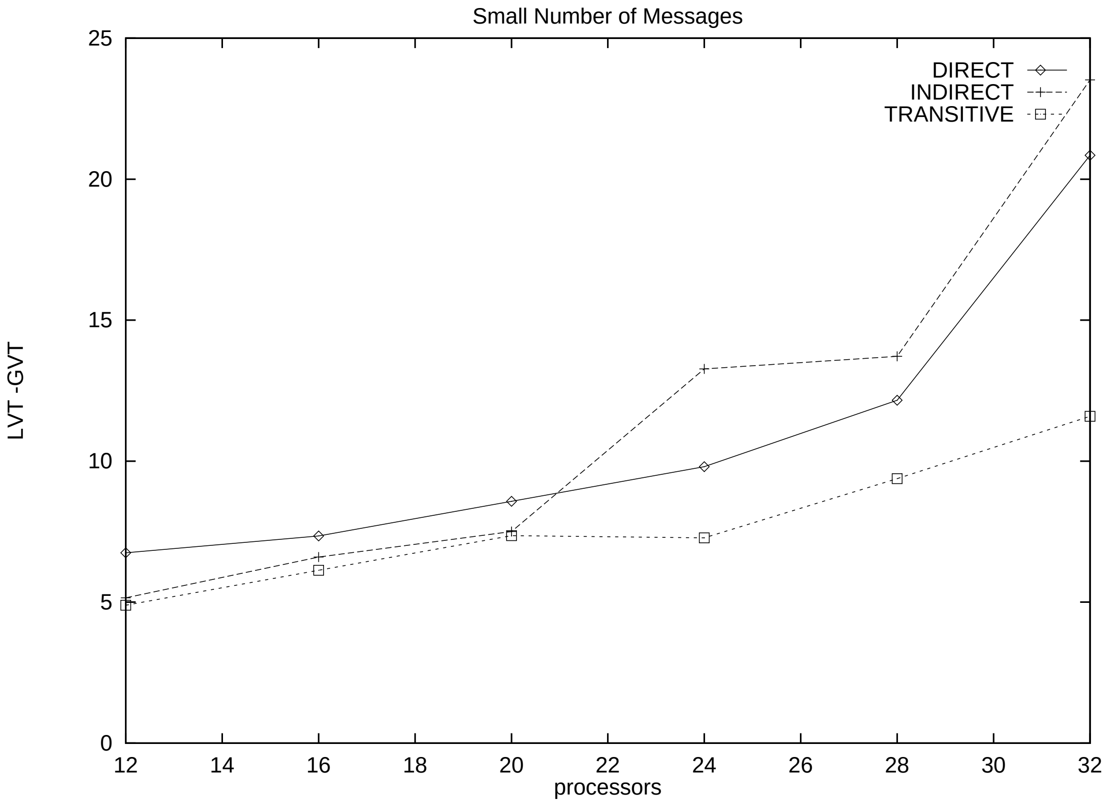


Figure 5.6: Small Number of Messages.

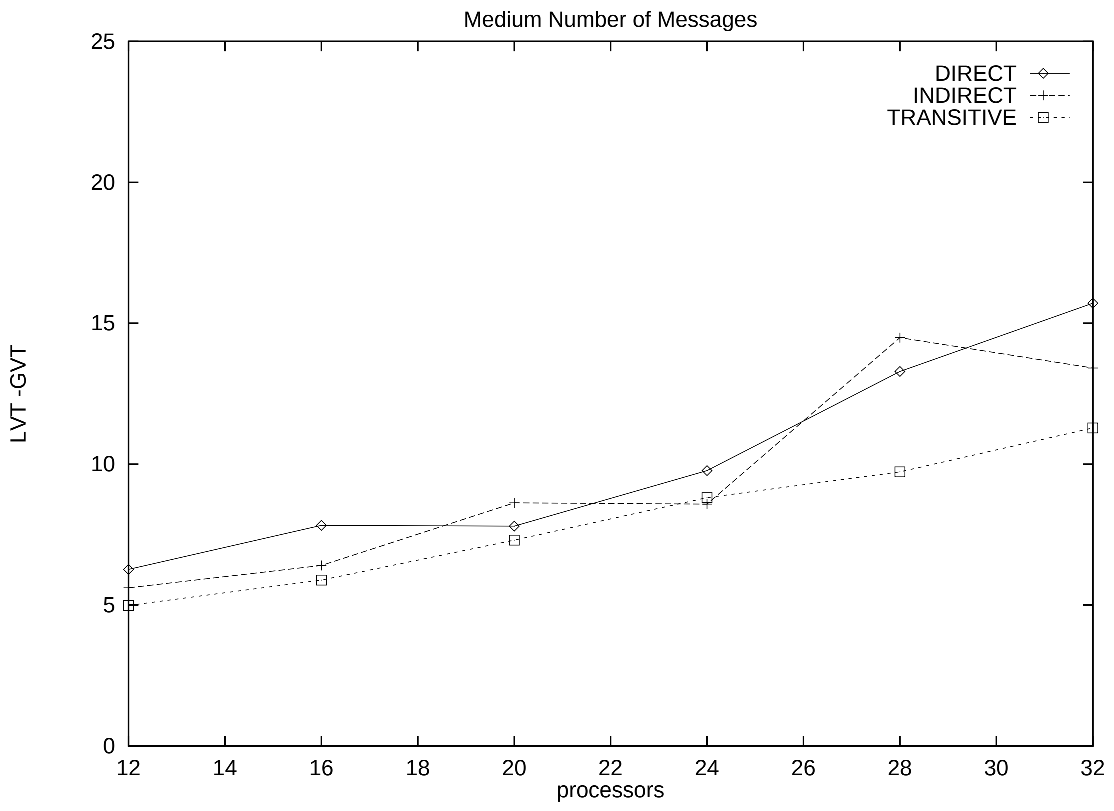


Figure 5.7: Medium Number of Messages.

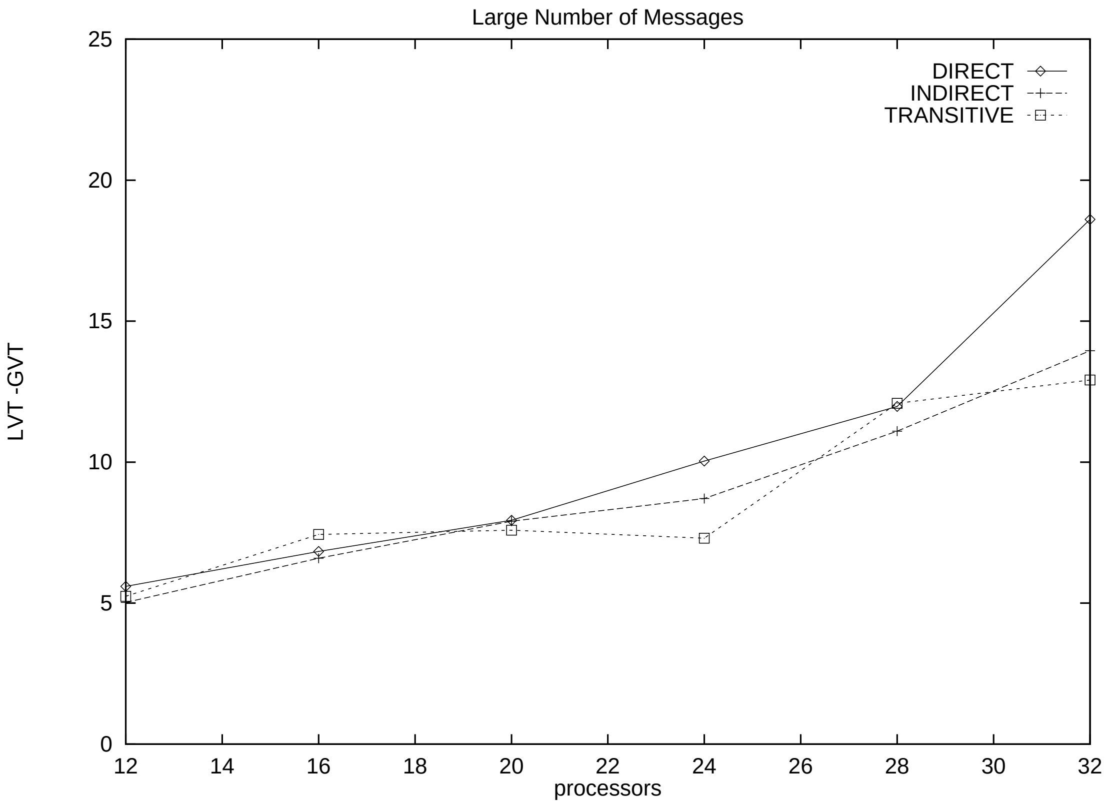


Figure 5.8: Large Number of Messages.

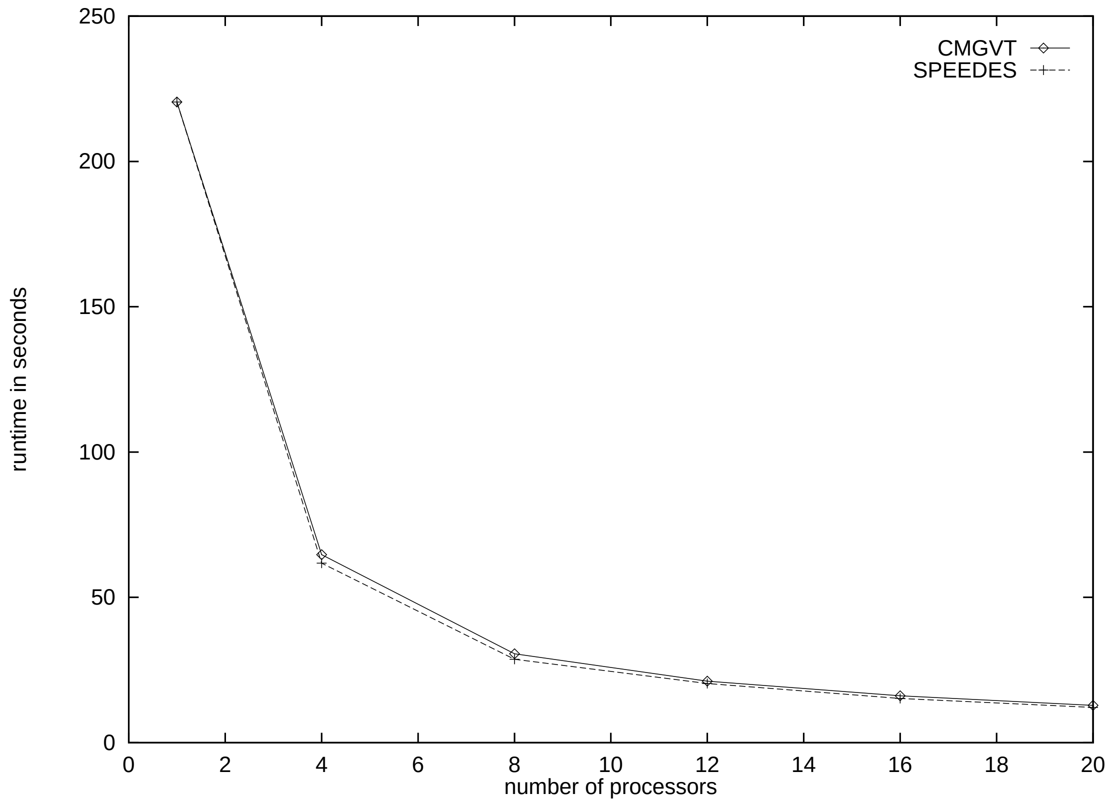


Figure 5.9: Runtime for a Small Data Set.

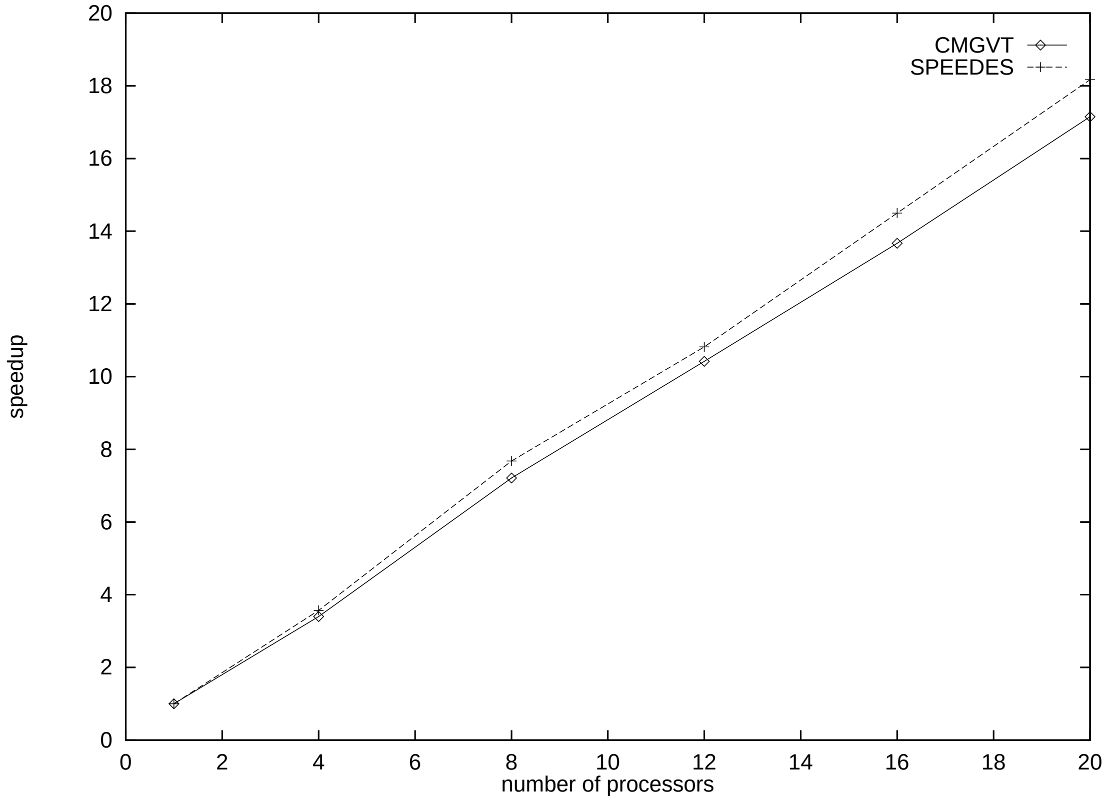


Figure 5.10: Speedup for a Small Data Set.

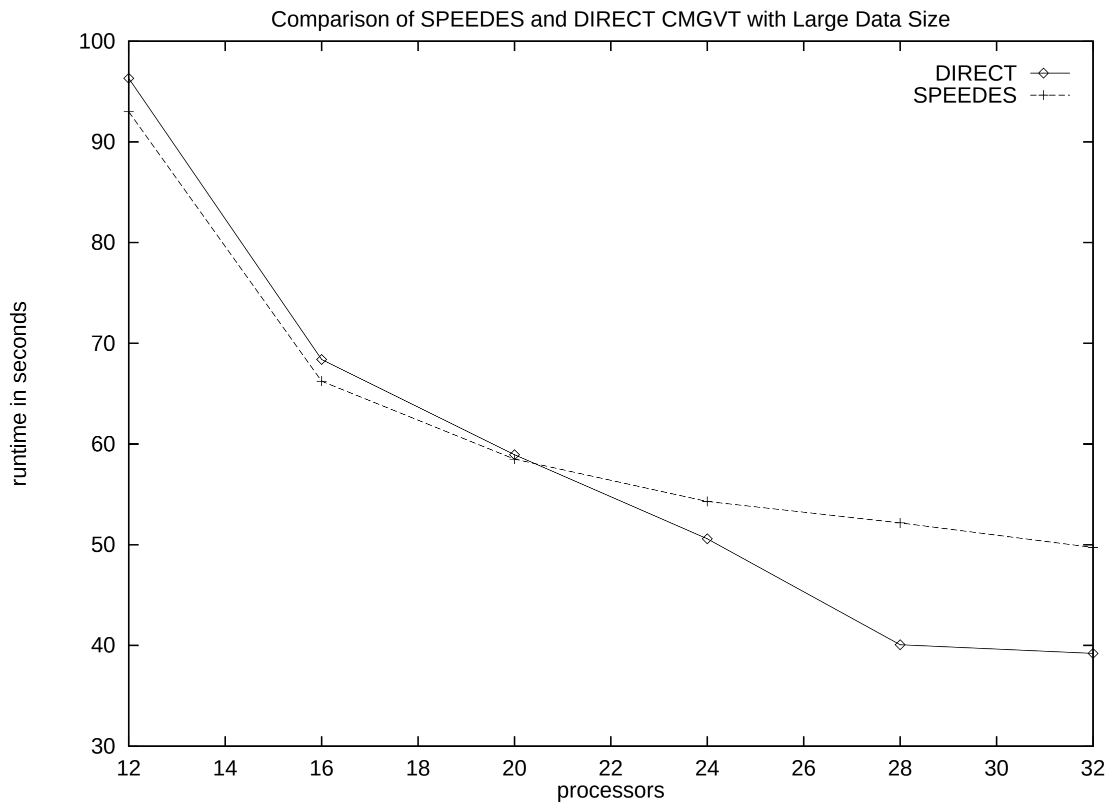


Figure 5.11: Runtime for a Large Data Set.

CHAPTER 6
SPATIALLY EXPLICIT PARALLEL DISCRETE EVENT SIMULATION AND ITS APPLICATIONS

6.1 Lyme Disease

Humans become susceptible to Lyme disease, by acquiring a pathogen, the spirochete *Borrelia burgdorferi*, that normally infects small mammals and an insect vector [53, 54, 55, 56]. The blood-feeding vector is the deer tick *Ixodes scapularis*. Immature ticks (larvae and nymphs) usually feed on the white-footed mouse *Peromyscus leucopus*. However, immature ticks also may bite a variety of mammals and birds. Humans bitten by an infectious nymph may subsequently develop Lyme disease [57]. Adult ticks are less generalized; they feed on white-tailed deer *Odocoileus virginianus* [58, 55].

The Lyme disease phenomenon is driven, at its ecological basis, by the cycle of infection passing from tick to mouse to the next generation of ticks (see Figure 6.1). Larval *I. scapularis* hatch during summer. The larvae that obtain a blood meal from a mouse then overwinter as inactive nymphs. The following spring these nymphs quest for a second blood meal. Those nymphs that are successful in attacking a mouse advance to the adult stage. The adults soon attack deer, where they feed again and also mate. Gravid females drop off the deer that they have parasitized and they lay their eggs, completing the two-year life cycle [59, 56]. The "inverted" seasonal abundance of the immature-tick stages maintains the spirochete. Infected nymphs transmit the pathogen to susceptible mice in the spring. When summer arrives, newly hatched larvae feed on the same mice, acquire the spirochete, and so complete the cycle of infection.

The Lyme disease epidemic is ordinarily depicted at the regional geographic (i.e., spatial) scale and among-year temporal scale [57]. The local spatial scale implies an area occupied by a single deme of the white-footed mouse. In our model, individual mice shift their home range within the local area [61]. As they disperse, mice can experience heterogeneity in numbers of infectious ticks. Furthermore,

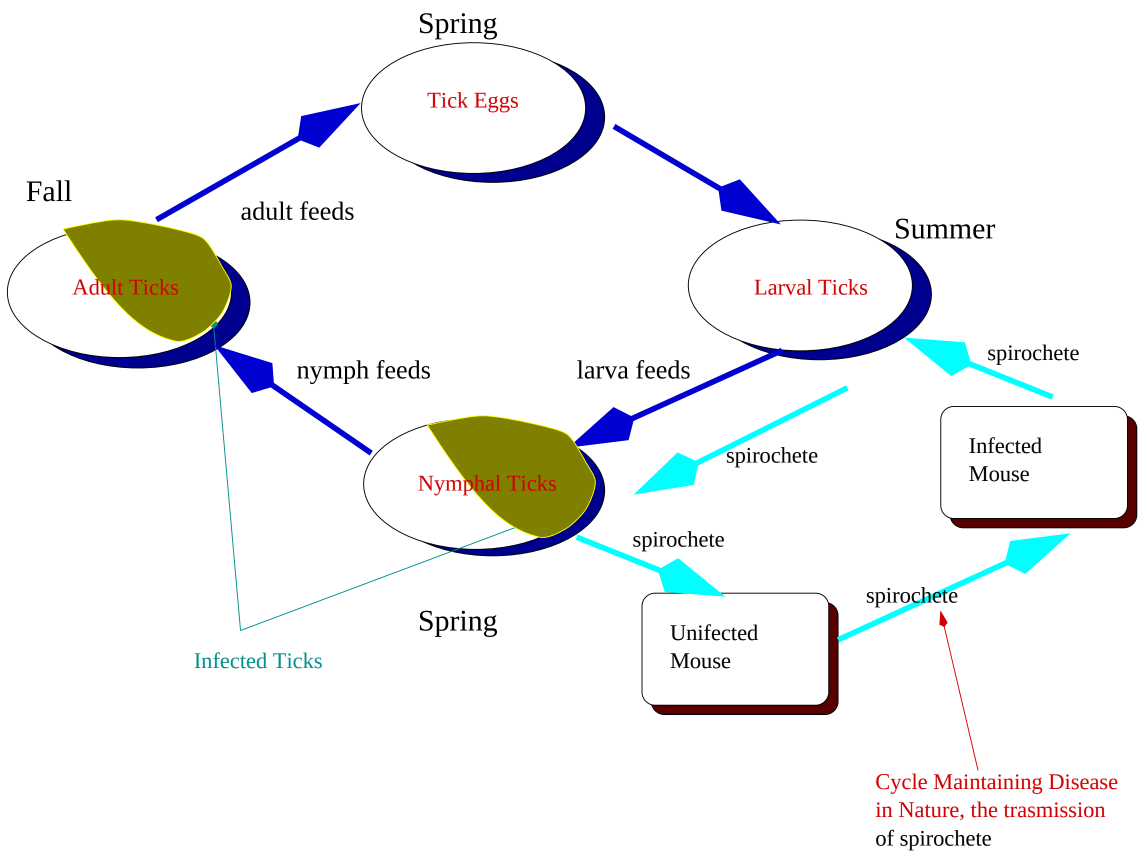


Figure 6.1: Lyme Disease.
[60]

dispersing mice may carry attached ticks between sites within the local area. Either the mice or the ticks parasitizing the mice may carry the pathogen. The simulations extend over the part of the year during which the cycle of infection occurs, the 180 days elapsing between the appearance of questing nymphs and the completion of feeding by the next generation's larvae (Figure 6.2).

This computational model is termed "individual-based" [62], because the mouse population is an ensemble of different individuals. Each individual is tracked according to its:

i. age

ii. spatial location on a rectangular lattice,

iii. load of parasitic ticks, and

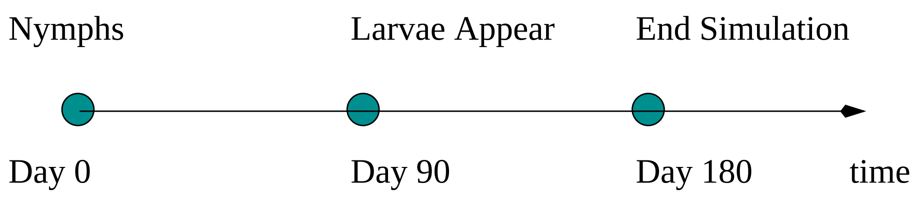


Figure 6.2: Time Line of the Simulation.

iv. infection status (susceptible or infected).

At each lattice node the number susceptible and infected ticks in each of the three developmental stages (larval, nymphal, adult) is counted. The individual-based model treats population descriptions, such as frequency of infection among mice and the spread of infected ticks across the lattice, as consequences of events occurring at the level of individual mice [63]. A single simulation includes mortality and dispersal, but not reproduction. However, the output of one simulation can be mapped through natality functions to produce the initial condition for the next simulation.

6.2 Individual-Based Modeling

Ecological modeling often involves simulating differential equations which describe various species' rates of population growth. Increasing system complexity, however, requires more complex equations making the resulting model hard to understand or analyze. Individual-based modeling may be an attractive alternative. One can establish relatively simple rules for events that affect individual organisms. Simulation of the individual-based model allows different individuals to experience different events; system behavior summarizes the ensemble of individual demographic histories.

Differential equations represent "homogeneous" behavior of a mass of individuals and require a significant number of individuals at each grid point; therefore, each grid point represents a relatively large space which is treated homogeneously. Since individual-based models proceed from simpler assumptions and avoid the demographic averaging of differential equations, they offer important advantages for theoretical ecology.

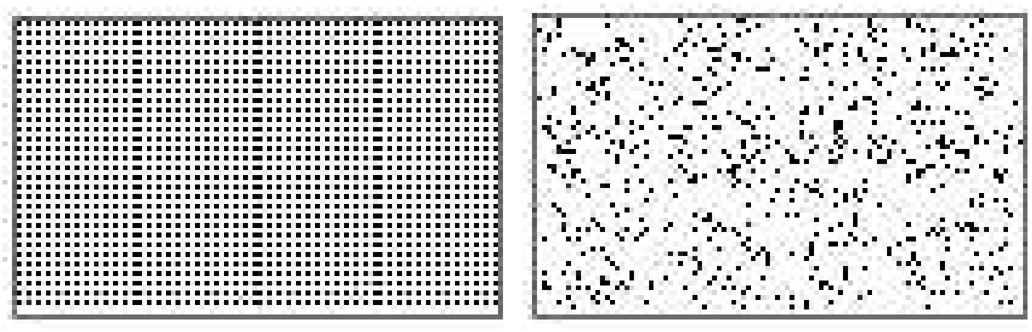
Individual-based modeling allows one to reason about what kind of events can happen to an individual. For example, one can think of an animal searching its home range for a nesting site. One can imagine that the mouse will look randomly in its own home range, and that, if it finds a suitable location, it will stay there. The analysis can also be more complex. A single, infected mouse or a group of mice can be tracked. Individuals with different genetic characteristics can be treated differently—some may be susceptible to diseases to which the general population is not susceptible.

The space where mice and ticks reside has a toroidal shape (wrapped in both directions). This eliminates edge effects. The mice are treated as individuals. Ticks, because of their density (as many as 1200 larvae/400m^2) [64], cannot be treated as individuals. This means that every node of the spatial lattice will contain a "*Tick Blob*", which consists of several categories of ticks:

- questing larvae
- questing nymphs
- non-questing nymphs (nymphs that have transformed from larvae and are in an inactive state)
- adult ticks (adults that have transformed from nymphs and are in an inactive state)

Each tick category is subdivided according to the presence/absence of the spirochete infection. No transovarial infection is assumed for ticks. Larvae are born without the infection. They acquire the spirochete when feeding as larvae, which later transform into non-questing nymphs. Non-questing during our 180 day simulation, they will become questing the year after. The spirochete can also be spread from an infected mouse to a feeding nymph (which, in turn, transforms into an adult tick).

The mice can disperse in the environment. They can be bitten by ticks, infected with the spirochete, or infect the feeding ticks with the parasite. The mice can also die, either as a result of failing to find an open site (lattice node) during dispersal, or due to some other causes.



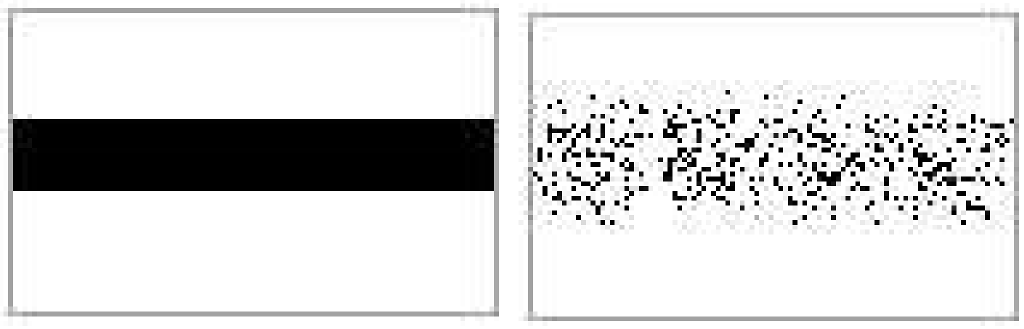
(a) Day 1 (b) Day 180

Figure 6.3: Even Distribution.

6.3 Initial Simulation

Mouse movements are modeled as follows: the mouse enters the dispersal state drawn according to a geometric distribution function with expectation $1/\theta_d$ days. If the mouse is to disperse, the animal randomly selects one of eight directions for dispersal. It moves in that direction with a waiting time derived from the exponential distribution with expectation $1/\theta_m$. If the first site the animal encounters is open (no other mouse present), the mouse stops and survives dispersal with certainty. If the first site is occupied by another mouse, the number of steps (n_s) the mouse has taken since dispersal is updated. The mouse then dies with probability $(n_s/r)^2$, where r is the maximum number of steps a mouse can take before dying. If it survives, it attempts to disperse to the next site in the same direction. After r unsuccessful steps ($n_s = r$) the mouse dies with certainty.

Figure (6.3) shows mice movements in an environment where the mice are evenly distributed, with free space available between the nodes occupied by the mice. There are more mice at the boundaries of processors to show where the strip-wise space decomposition occurs (the darker vertical lines in part a of the figure). The simulation starts with 1,560 mice and ends after 180 days with 52.4% of mice dead due to natural causes, and 0.32% dead due to lack of space. Figure (6.4) shows the mice movements in an environment where each of the nodes in a "populated region" is occupied. On the first day there are 1,500 mice in the simulation, and by the last day 22.2% of the mice die due of lack of space, and another 41.7% die by



(a) Day 1 (b) Day 180

Figure 6.4: Band Distribution.

other causes.

6.3.1 Mice Simulation Events

Many events can affect a mouse. There are several object functions that create events, which are in turn inserted into the future event queue. Figure 6.5 shows a diagram of possible events associated with a mouse.

The following object functions are related solely to the mouse:

- *Disperse* creates a new *Disperse Event* and enqueues it. The event time is derived from an exponential distribution with a mean of d days (in these simulations $d = 20$ is used).
- *Start_moving* puts the mouse in a dispersal state (sets dispersal flag), resets the direction of move, and calls *move*.
- *Move* picks a new direction (if it is not defined yet), determines when to move (dictated by an exponential distribution), creates a corresponding *Move Event*, and enqueues it.
- *Next_move* "flips a coin". If the number is within the death probability, the function makes a new *Kill Event* (instantaneous) and enqueues it; if, on the other hand, the mouse survives, *move* is called.

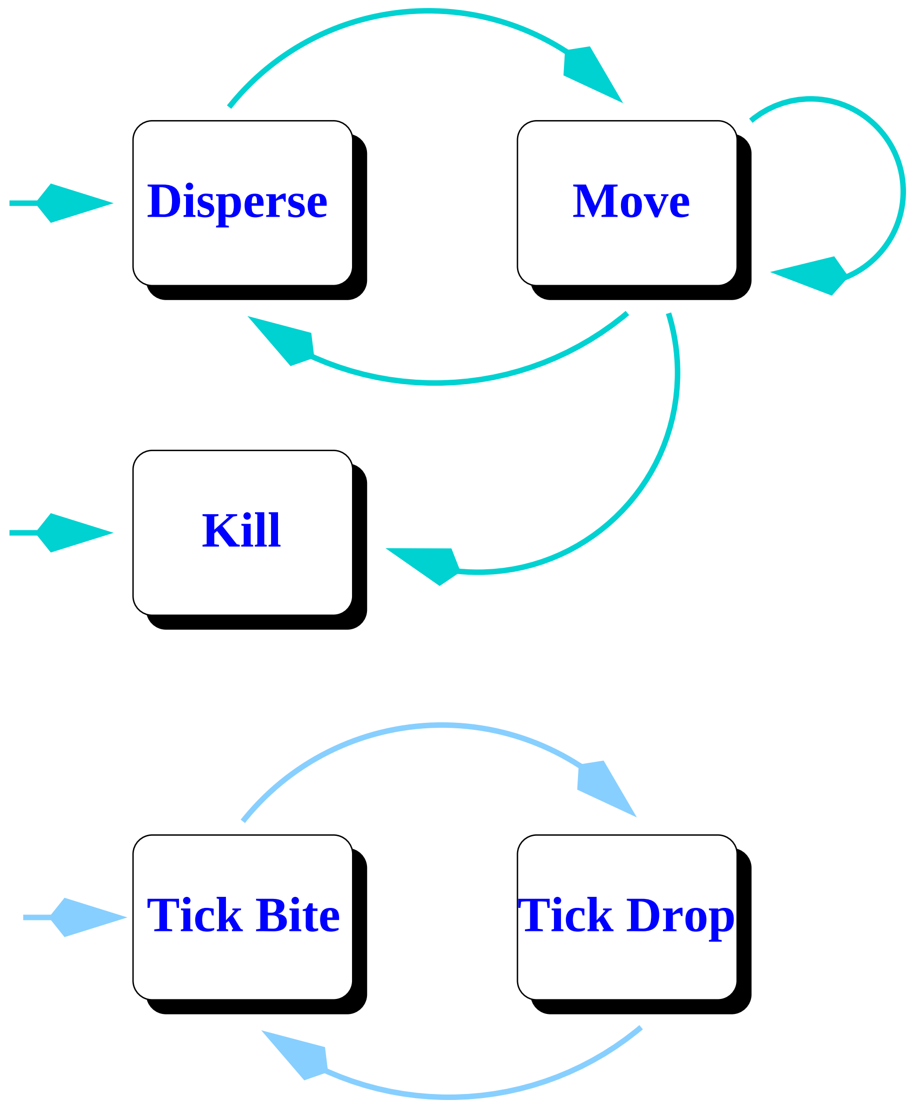


Figure 6.5: Possible Events for a Mouse.

Events are dequeued from the event queue and processed according to the following rules defined for each event type:

- *Move Event*: the Space Manager removes the object from the location where the object resides. If the object is going out of bounds an event message is sent to the process to which the object is moving. If the object is not going out of bounds, the object is placed at the new location. If that location is already occupied, the number of steps that that object has taken is increased and *next_move* is called. If the site is empty, the number of steps that the object has taken is reset to zero, the object's dispersal status is changed to non-dispersing, and a new disperse event is created.

- *Disperse Event*: function *start_moving* is called.
- *Kill Event*: the space where the object is located is found, and the object is removed from that location. Now, all other events that have been scheduled in the future for that object are impossible and are removed from the event list and put along with the object on the *ghost list*. When an event is processed, it is inserted into the *processed event* list.

"Undoing" events is a crucial part of the rollback mechanism. Undoing an event restores the state of the system to the one just before the time that the event was processed. Below are the functions used to undo the effects of events occurring in the system:

- undo *Move Event*: The current location of the object is found. If that object is in the space assigned to the LP, the object is removed from that location. If the LP does not contain that object, it means that the object was sent out to another LP. Therefore the object and its future events can be found on the ghost list, and can be restored from it (see section 3.4). In either case the object is then placed at its previous location. The number of dispersal steps is decremented.
- undo *Disperse Event*: The dispersal status is changed to non-dispersing and the direction of dispersal is reset.
- undo *Kill Event*: The object is removed from the *ghost list* and is put back in the location where it was killed. All the events that were put along with the object into the *ghost list* are restored by being reinserted into the *future event queue*.

6.3.2 Ticks

Although ticks are not modeled as individuals, their densities are updated at the time mouse events occur. Not all individual tick bites are counted, since studies show that there can be as many as five individual larval bites per day. Accordingly, multiple bites are combined into one: ten larval bites or five nymphal bites at one time. At the beginning of the simulation only nymphs that have over-wintered are present. They are then questing nymphs. At about the 90^{th} day eggs hatch, larval ticks enter the simulation, and the *Tick Blobs* at each spatial node are updated. When a mouse is bitten by a *Tick Blob*, the number of ticks at the lattice node where the mouse is located is decreased by the number that bit the mouse. When the *Tick Blob* drops off the animal, tick densities at the lattice node are increased. It is also assumed that when the mouse dies, the *Tick Blob* (if any) present on the mouse dies. A mouse can be bitten by a *Tick Blob* as long as there are enough questing ticks on the node of the lattice. This assumption implies that there is a threshold for both larval and nymphal ticks beyond which no new bites are noticed. There are two types of bites: a nymphal bite and a larval bite. They are treated independently, because they are temporally independent for most of the simulation. The events involving ticks are:

- *Tick Bite*: For larvae/nymphs: update the tick densities in the *Tick Blob* according to the mortality function which has an exponential distribution with rates as high as 95% for larvae, and 90% for nymphs. If the number of larvae/nymphs falls below a certain threshold, exit, otherwise, schedule a new *Tick Drop* event with an exponential waiting time and remove the larvae/nymphs from the *Tick Blob* at the node. For the bite a new *Tick Blob* consisting only of larvae/nymphs is created. The ratio of infected/uninfected larvae/nymphs in the new blob reflects the ratio present in the *Tick Blob* on the lattice node. If the mouse is not infected and there are infected ticks in the new *Tick Blob* the mouse becomes infected. If the mouse is infected, it infects the uninfected ticks in the *Tick Blob*.

- *Tick Drop*: update the tick densities in the *Tick Blob* according to the mortality function. Add the ticks present in the *Tick Blob* on the mouse to the densities present in the node. The addition is complex, because the larvae present on the mouse become non-questing nymphs, and nymphs become non-questing adults.

Undoing the events entails restoring the *Tick blob* at the node and restoring the infection status of the mouse. It can be seen that the tick densities at lattice nodes will be updated only when a mouse is present in the area, but ticks die even in an area where no mice are present. The solution used is to update the tick densities at empty (mice-free) locations during the *fossil collection* stage (right after the new GVT calculation).

6.4 Ecological Results

The goal of this simulation is to show that the spread of Lyme disease is directly related to the dispersal of mice. The simulation was run on an IBM SP2, a distributed memory MIMD machine. The following graphical results are shown for four processors. Each of the four processors had about 400 mice and a 100x60 lattice. Each lattice node is assumed to be 400m^2. Initially all mice are uninfected by the spirochete. Figure 6.3 and Figure 6.4 show the distributions of mice used in the simulations. Figures 6.6-6.9 represent the presence of the disease in a given location; if there is at least one infected tick in the *Tick Blob* at a given location, the figure will show a data point. The infected ticks can be either questing ticks or ticks that have had their blood meal. Mice are not explicitly depicted in the figures. The figures depict simulations at different points in time.

Figure 6.6(a) shows the initial configuration of infected nymphs. The uninfected nymphs are placed similarly and the larvae will be added to the simulation on the 90th day in the same pattern. Notice that the larvae will not be infected, since no transovarial transmission of disease is assumed. The mice are distributed evenly as in Figure 6.3. Figure 6.6(b) shows that the disease is dying out due to the nymphal mortality. It is sustained in places where the nymphs have fed on mice and dropped off as adult ticks. Figure 6.6(c) shows the presence of the disease at the end of the 180th day. It shows that initially uninfected larvae fed on infected mice, received the pathogen, and dropped off as infected non-questing nymphs.

A question is posed: Is the spread of the disease correlated with the mouse dispersal rate? To answer that the dispersal rate was increased by a factor of four. The results are depicted in Figure 6.7. The simulation shows that the disease spreads faster among both the questing nymphs (Figure 6.7(a)) and the questing larvae (Figure 6.7(b)). In comparing the final configurations (Figure 6.6(c) and Figure 6.7(c)), it can be noticed that the density of the disease-carrying ticks is much higher when mice are dispersing faster. This is because the faster the mice disperse, the more area they can cover.

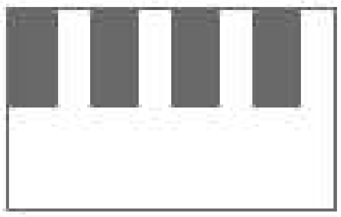
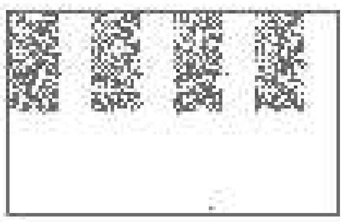
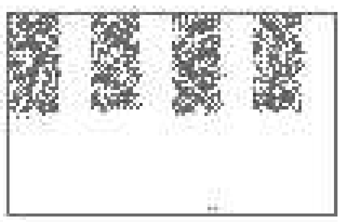
An interesting issue is whether and how the distribution of mice affects the spread of the disease. To answer that, the mice were distributed band-wise as depicted in Figure 6.4, and the initial configuration of nymphal and larval ticks remains the same (Figure 6.8(a)). The disease dies off in the area where no mice are present (Figure 6.8(b)). This is due to the fact, that the questing nymphs fail to find a blood meal and therefore die. The final configuration (Figure 6.8(c)) shows that the spread of the disease is increased by the presence of non-infected larvae.

The next set of figures shows the same configuration for ticks and mice, but the mice are dispersing four times faster. The disease spreads farther and the spatial density is higher for faster dispersing mice. The results correspond the current understanding of the spread of the disease. In Figure 6.10 a graph of the spread of the disease through the ticks is presented; the decomposition is band-wise and mice disperse fast. The first two bars represent the total number of questing nymphs and the number of infected questing nymphs at the beginning of the simulation. The next two bars depict these nymphs after they took a blood meal and molted into non-questing adults. The fifth and sixth bars represent the larvae on day 90, and the last two bars represent these larvae after they have transformed into non-questing nymphs. The infection ratio might seem high, but the parameters that were chosen for the model are the most favorable for the spread of the disease.

6.5 Performance

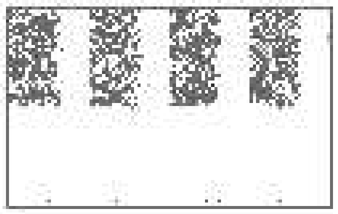
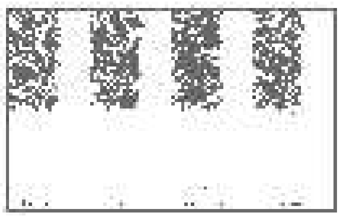
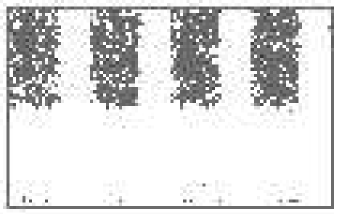
Good speedup was achieved for small data sets: 2,400 lattice nodes with 800 mice initially [65]. The results are shown in Figure 6.11. The speedup grows with the number of LPs for up to 10 processors. With 12 processors the communication overhead becomes large, decreasing the overall performance.

When the lattice size is increased to 32,000 nodes and 8,000 mice, with the

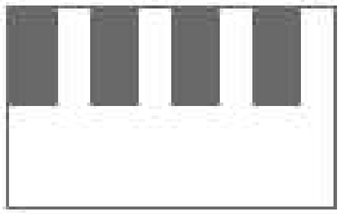
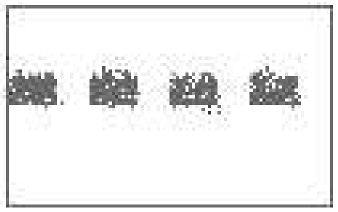
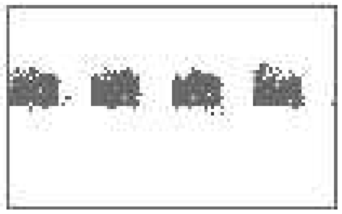
(a) Day 1 (b) Before Larvae (c) Day 180

Figure 6.6: Distribution of Infected Ticks. The Mice are Distributed Evenly and Disperse Slowly.

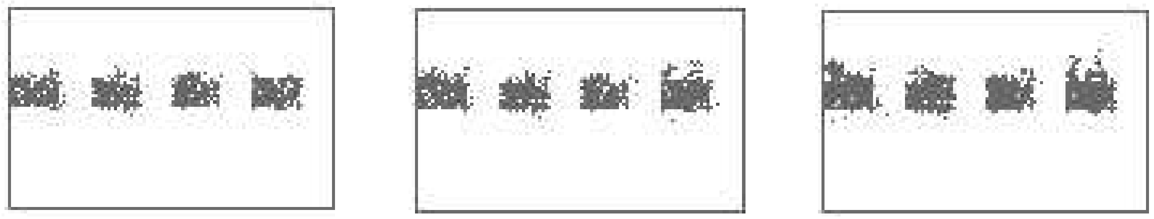
(a) Before Larvae (b) After Larvae (c) Day 180

Figure 6.7: Distribution of Infected Ticks. The Mice are Distributed Evenly and Disperse Fast.

(a) Day 1 (b) Before Larvae (c) Day 180

Figure 6.8: Distribution of Infected Ticks. The Mice are Distributed Band-Wise and Disperse Slowly.



(a) Before Larvae (b) After Larvae (c) Day 180

Figure 6.9: Distribution of Infected Ticks. The Mice are Distributed Band-Wise and Disperse Fast.

same distribution of mice and ticks, the speedups are less impressive (Figure 6.12). This is caused by rollbacks whose cost is proportional to the size of the lattice for which each LP is responsible. With 4 processors, the lattice size per LP is large—8,000 nodes. When a rollback occurs, all the events that happened in the affected time in all of the 8,000 nodes have to be rolled back.

Several methods of reducing the possibility and cost of rollbacks were investigated.

6.5.1 Multiple Logical Processes per Processor

First, the number of strips into which the lattice is divided is increased, thus decreasing the area assigned to each LP. The processes are mapped by the job scheduler of the IBM SP2; therefore, a process is not aware if processes with which it exchanges data are run on the same processor as it is running. Hence, it always calls interprocess communication for such exchanges, slowing down the execution. The results of dividing the problem into as many as 20 LPs on up to 16 processors are presented in Table 6.1. The best times for 4 processors are achieved when each processor has 5 LPs. Still, there is no speedup (the sequential time is 227 sec.). The speedup with 8 processors and 16 LPs is very small, around 1.6. The speedup with 12 processors is best when 20 LPs per processor are used, and is equal to 2.2. With 16 processors the speedup improves slightly to 2.8.

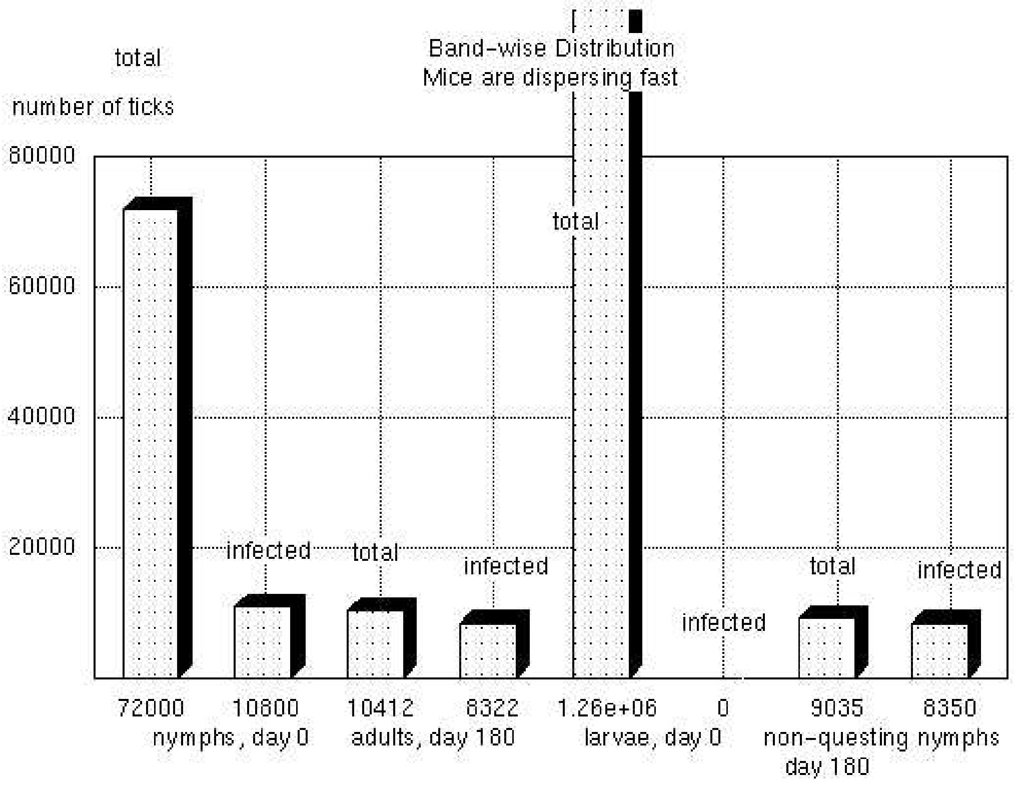


Figure 6.10: Spread of Disease in Various Tick Types.

Table 6.1: Runtime in Seconds for Multiple *LP*s per Processor

	Number of *LP*s			
Processors	8	12	16	20
4	502.8	510.07	289.96	226.85
8	275.76	280.4	139.7	249.29
12	-	149.42	116	103.7
16	-	-	82	98.16

6.5.2 Curbing Optimism

Another way to decrease the impact of rollbacks is to curb the optimism by allowing each LP to process events only within a limited time into the future. This type of execution throttling was also used by Das [66], where the LPs are allowed to simulate only a certain amount of time past the GVT. In this research an LP allowed to advance only by 20 or 30 days ahead of the average LVT of others (this average can be calculated during the GVT calculation). The results are shown in

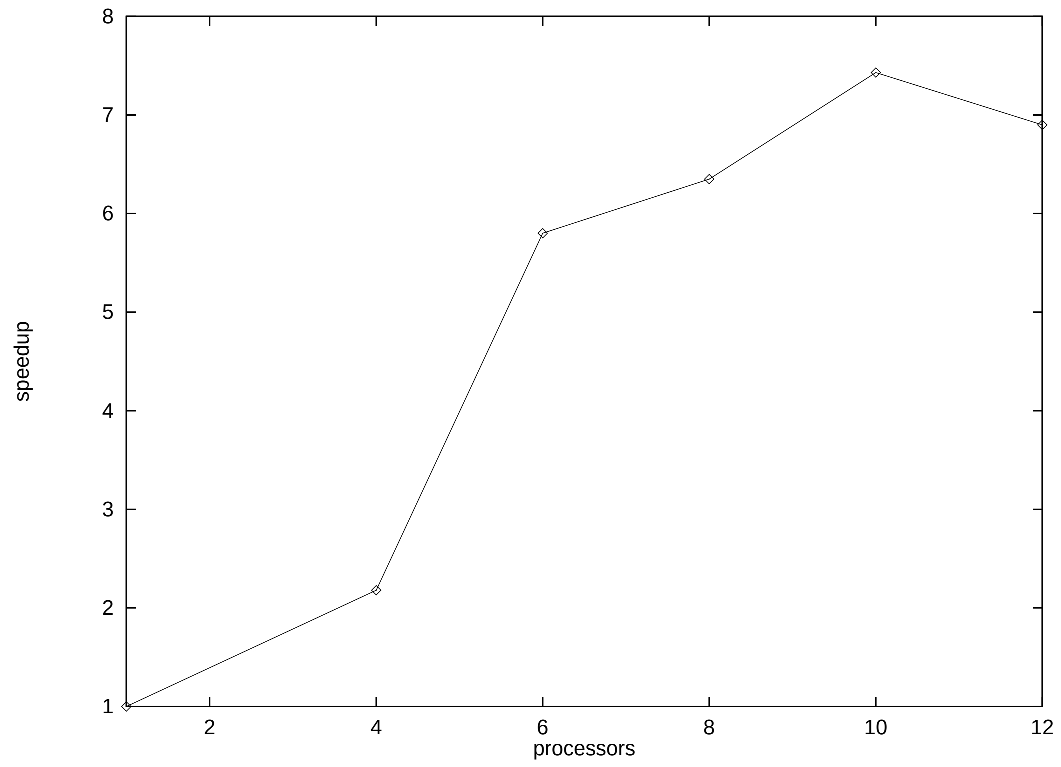


Figure 6.11: Speedup for a Small Data Set.

Table 6.2. The performance of this method was better than in the previous case only for 16 processors, with the best speedup of 3.7 for the 30-day time cap.

Table 6.2: Curbing the Optimism (time in sec.)

Processors	20 Days	30 Days
4	769.27	936
8	177.46	224
12	122	128.48
16	> 200	61.79

The combination of both the use of multiple LPs per processor and curbing the optimism to processing only up to 30 days ahead of the average LVT was investigated. The results are shown in Table 6.3. The combined method resulted in an overall improvement over each of the component methods with the most significant improvement for 12 processors.

The number of rollbacks by using multiple LPs and curbing optimism was

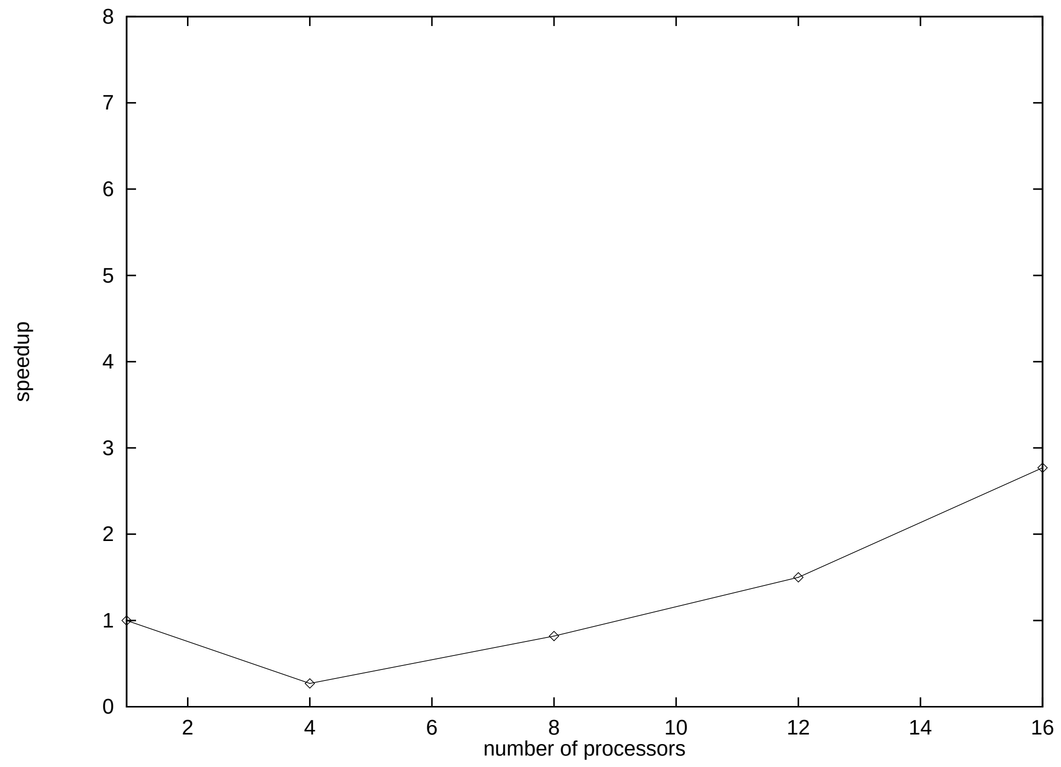


Figure 6.12: Speedup for a Large Data Set.

Table 6.3: Multiple LPs and Curbed Optimism (time in sec.)

	Number of LPs			
Processors	8	12	16	20
4	492.98	379.37	359.6	242.55
8	224.0	215.23	118.17	188.7
12	-	128.48	88.81	100.27
16	-	-	61.79	74.49

reduced. The dramatic results of a typical run on 8 processors are shown in Figure 6.13. With the increase of the number of LPs, the average number of rollbacks per LP decreases significantly. However, the number of rollbacks is not the only measure of performance. For example, the average number of rollbacks for 20 LPs is smaller than for 16 LPs , but the runtime is higher. This is because an increase in the number of LPs per processor intensifies the contention for the CPU, which slows the entire simulation. The average number of rollbacks decreased as the optimism of the simulation was curbed. For 8 processors, 8 LPs, and an optimism cap of 30

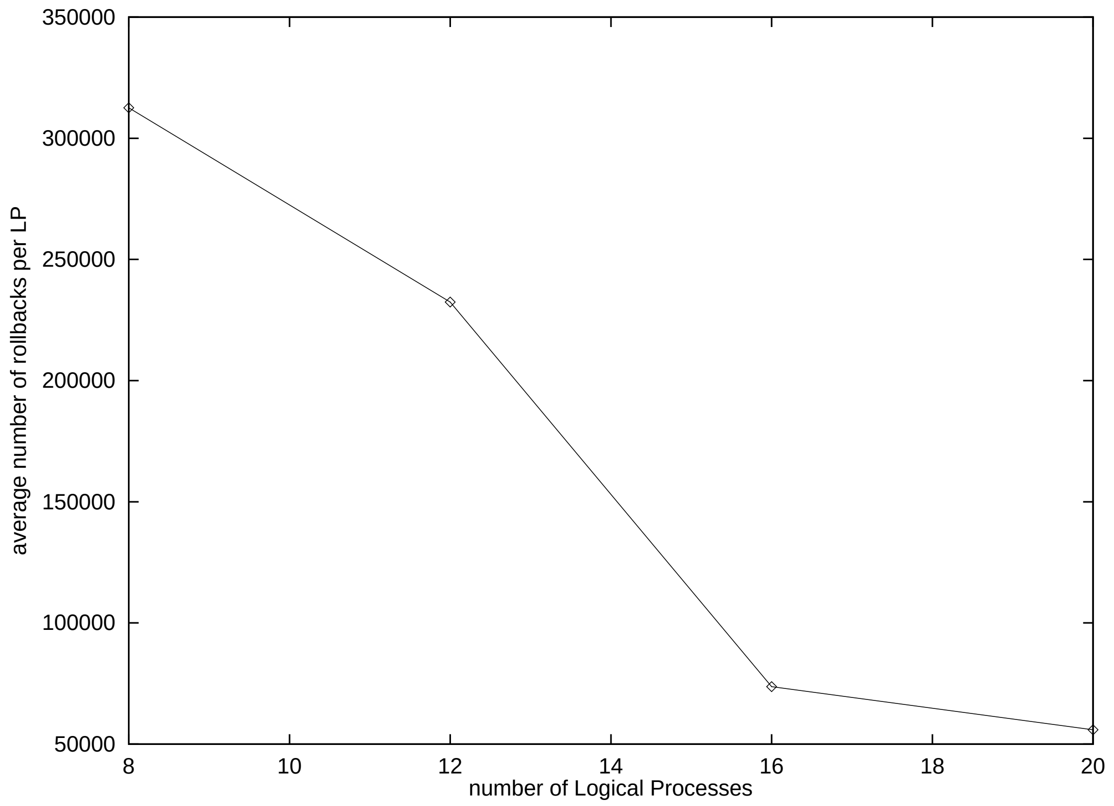


Figure 6.13: Average Number of Rollbacks for 8 Processors.

days, there is an average 136,355 rollbacks per LP; for a 20 day cap, the average was 92,770.

Still, the performance results were not as good as one would like. The cost of rollbacks is still too high. This led to the design of the new algorithm, the Breadth-First Rollback algorithm which minimizes the impact of rollbacks on a Logical Process (see Chapter 7).

CHAPTER 7
BREADTH-FIRST ROLLBACK

The method of rollback processing presented here is applicable to spatially explicit simulations. Incremental state saving techniques [25] are used to detect dependencies between events. Typical implementations of a rollback in such a setting (used in our previous implementation [45]) is to roll back the entire area assigned to the LP. In *Breadth-First Rollback* (BFR) [67], the novel approach, the rollback is contained to the area that has been directly affected by the *straggler* or *antimessage*.

7.1 Problem Partitioning

Two inter-related issues 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 performed, for example, in digital logic simulations). To address both of these issues, clustering of LPs is often used.

Lazy re-evaluation [3] 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, 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.

The Local Time Warp (LTW) [68] 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) [69, 70] 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. As a result, 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 [71], in which digital circuits are modeled, takes 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? The original 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 one would hope, 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 an LP/cluster, only the affected lattice nodes are rolled back, thanks to a breadth-first rollback strategy, explained in Section 7.2. 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 to an LP, 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, a partition is treated as a single LP, whereas, from the point of view of the past, the partition is viewed as a set of LPs (one LP per lattice node). The performance of the new method yields a speedup which is close to linear.

7.2 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 computation 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, one 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 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

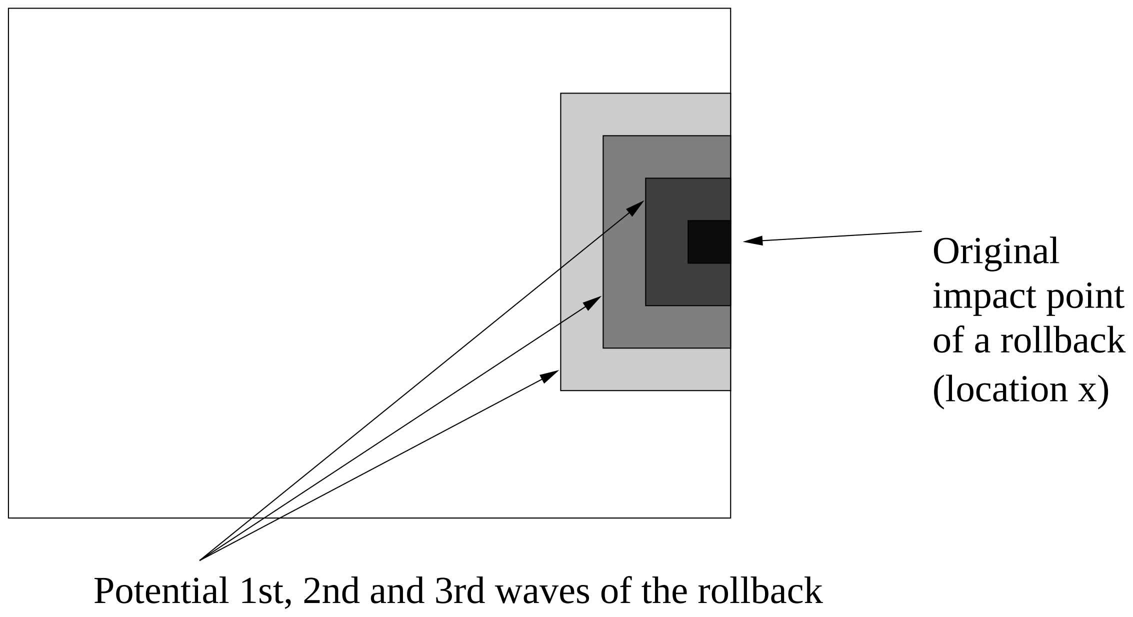

Original impact point of a rollback (location x)
Potential 1st, 2nd and 3rd waves of the rollback

Figure 7.1: Waves of Rollback.

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 7.1. 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 (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 the Lyme disease 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 7.1 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 7.1, because events at one site will most likely not affect all their neighboring nodes. Obviously, if an event at location x created messages for 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, there is no need to 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 [26] 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 an event is processed (in the forward execution), the logical time of the node where the event is scheduled has to be checked. If the logical time is greater than the time of the event, the node has to roll back.

7.3 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. The past information, which includes the processed event list, is distributed among the lattice nodes. Therefore, a change needed to be 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. When the move is non-local, (location (x_1, y_1) belongs to another LP), the MoveIn is placed in the *processed event list* at location (x, y). The MoveIn event is then sent to the appropriate process. When this event is received and processed, it will be placed at location (x_1, y_1).

Upon rollback, if a MoveOut 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* [72]. 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 MoveOut 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 chapter 8. 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 [3] 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.

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

7.4 Examples

The following code constitutes the skeleton of the Breadth-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 and including time t {
  // the events are undone in the order opposite
  // to the one in which they were processed
  // update the local virtual time (lvt)
  if ( E.eventTime < lvt ) {
     lvt = E.eventTime;
  }
  // make sure to undo the dependent events first
  while there exists an unprocessed dependent event  {
  // (E triggered D)
  if (D is at location (x1,y1) != (x,y) ) {
    if ( time of  (x1,y1) >= D.eventTime )  {
       rollback_space( D.eventTime, x1, y1  );
    }
  }
 }
 if (E.event_type == MOVE_OUT_EVENT)  {
   if the new location is outside_bounds   {
     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 event queue)
 remove_scheduled_events(E.id);
 }
}
```

To demonstrate the behavior of the BFR algorithm, let's consider the example in Figure 7.2. 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 (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

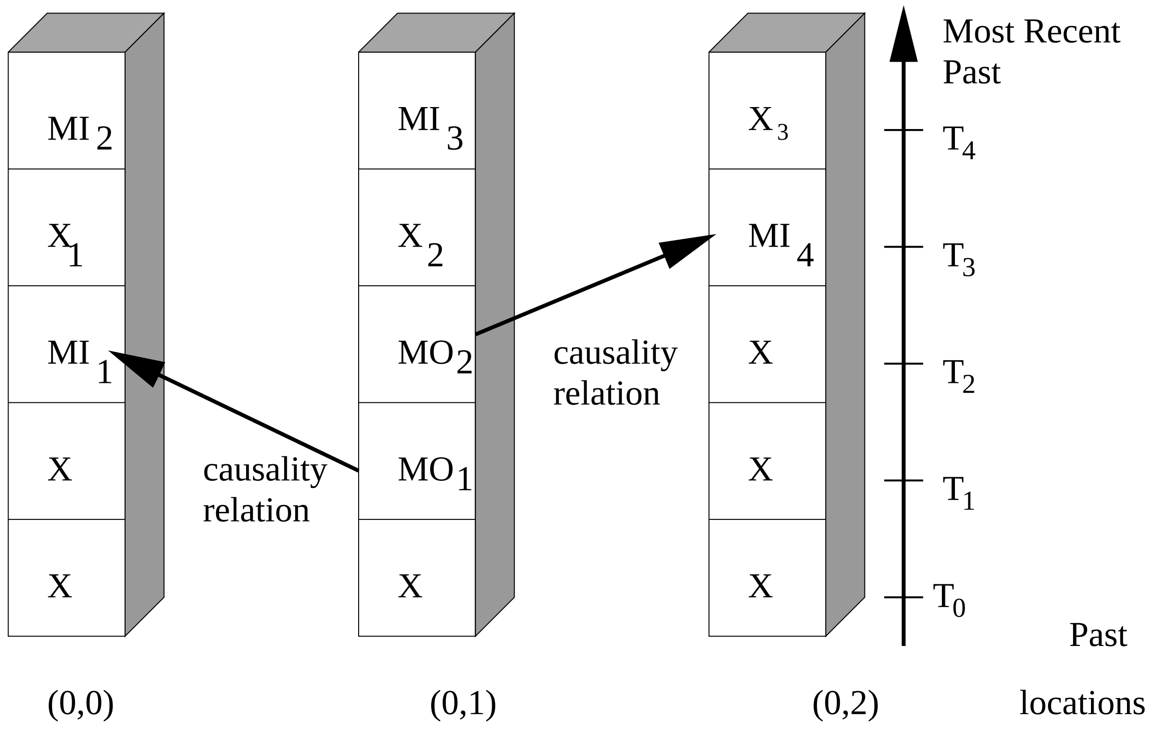


Figure 7.2: View of Processed Lists at Three Nodes of the Lattice.

rollback for location (0,2) and time T_3 is performed. This rollback needs to roll back all events up-to and including time T_3, which would result in a rollback of location (0,2) to time T_2. 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 occurs at location (0,0) for time T_1, then the three most recent events at location (0,0) would be undone and placed on the event queue, and no other location would be affected during the rollback. It is possible that the other locations would 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

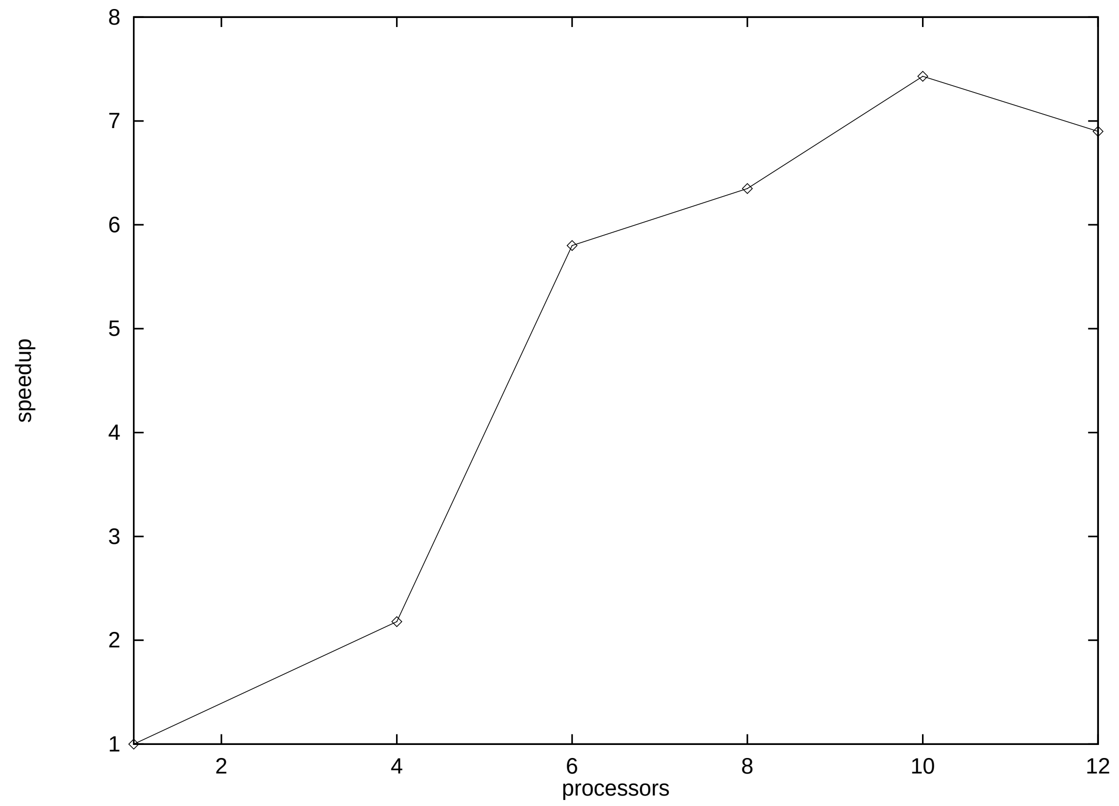


Figure 7.3: Speedup For Small Data Set (about 2,400 nodes).

(0,1) for time T_3, then location (0,1) would have to roll back to time T_3.

Interesting aside: suppose location (x,y) is at simulation time t, and the next event, is scheduled for time t_1 and location (x_1,y_1) is processed. If an event comes in from another process for time t_2 ($t < t_2 < t_1$), there does not necessarily need to be 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.

7.5 The Performance of Breadth-First Rollback

7.5.1 Comparison With The Traditional Approach

To demonstrate improvements in performance, the results are compared to those of the model of the initial simulation, which did not use the BFR method (see chapter 6).

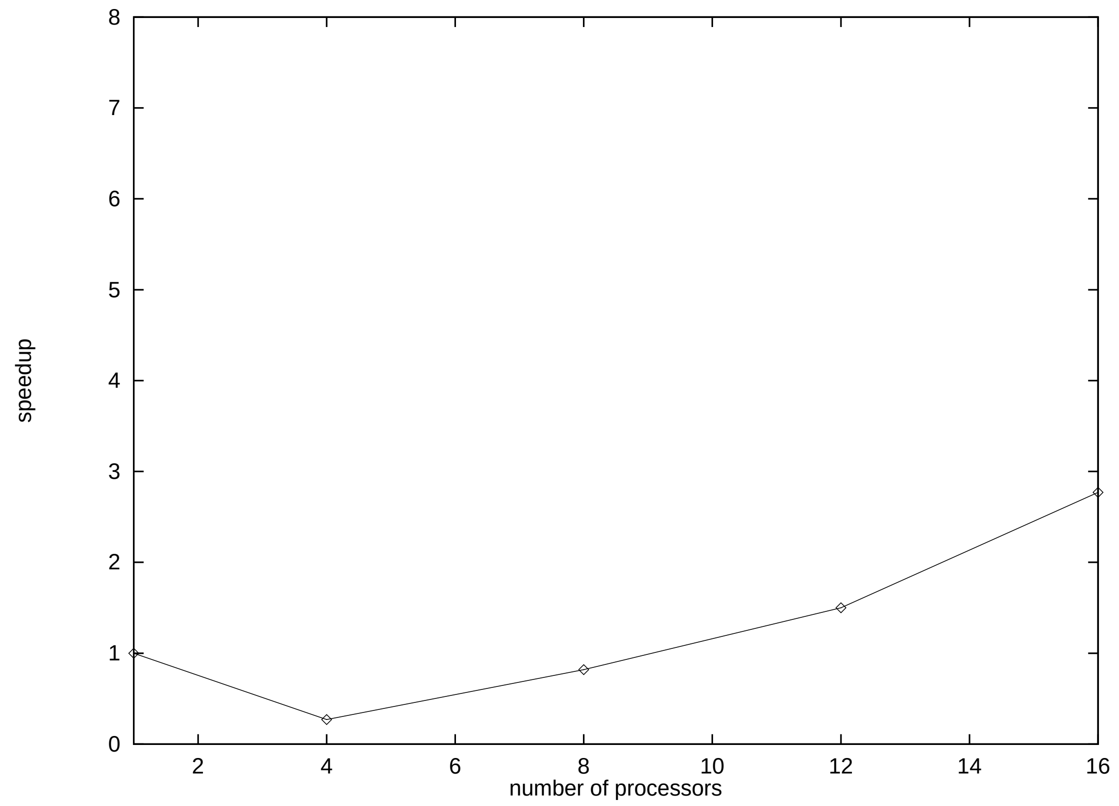


Figure 7.4: Speedup For Large Data Set (about 32,000 nodes).

Initial results obtained for a small-size simulation using the traditional approach were encouraging (Figure 7.3); however, the speedup was not impressive for larger simulations (Figure 7.4). 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, the space was divided into more LPs, while keeping the same number of processors. Figure 7.5 shows the runtime improvement achieved with this approach. For the given problem size, the ultimate number of LPs was 16 (Figure 7.6), and the best efficiency was achieved with 8 processors.

Figure 7.7 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 one is that, when a rollback occurs, not all the events belonging to a given LP need to be rolled back. Only the necessary events are undone. In the

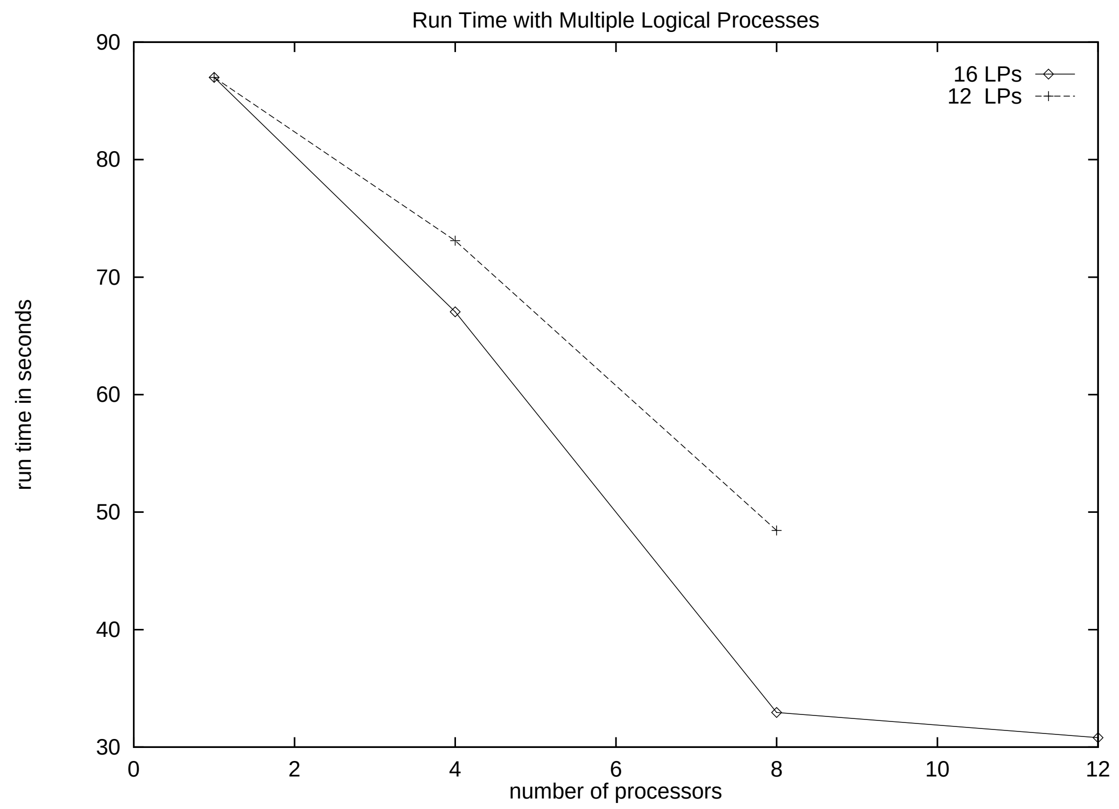


Figure 7.5: Running Time for Large Data Set and Multiple LPs per Processor.

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.

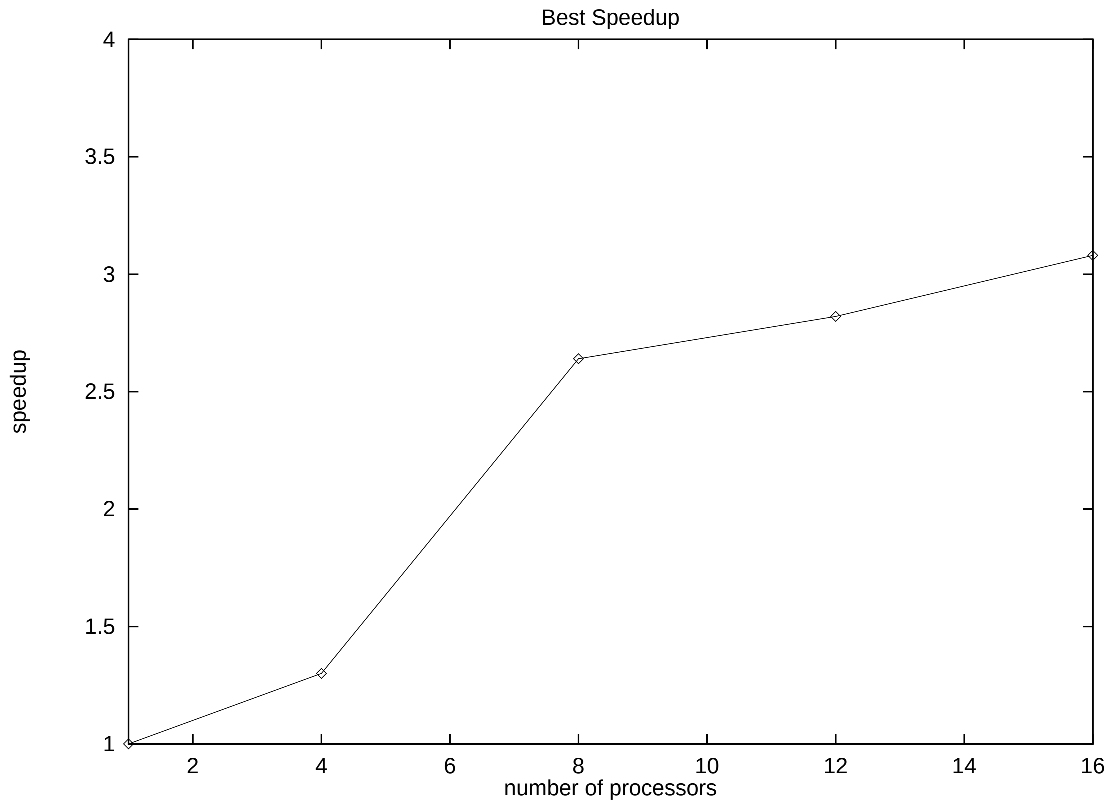


Figure 7.6: Speedup with Large Data set and 16LPs.

There are also fewer antimessages being sent as a result of the hybrid lazy cancellation. In general, having one LP per processor eliminates on-processor communication delays, and context switches are minimized. 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 also be code to deal with multiple dependents. There is no aggressive cancellation, but, as can be seen from the results, that does not appear to have an adverse impact on performance.

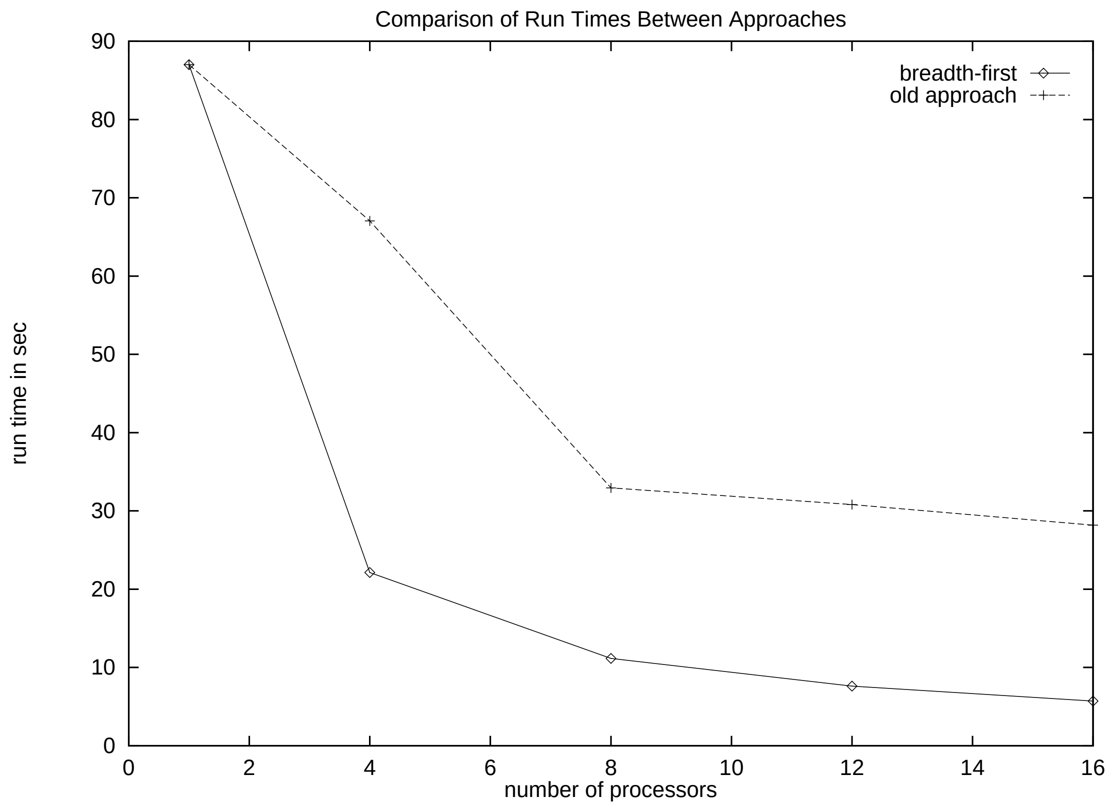


Figure 7.7: Results: Comparison of Runs With BFR and the Traditional Approach.

CHAPTER 8
DYNAMIC LOAD BALANCING

8.1 Impact of Work Load Balance on Performance

Up to now, the empirical results presented in this thesis have been obtained with the load evenly distributed among processes. However, in most applications, such an assumption is frequently unrealistic. For example, in ecological simulations, objects occasionally concentrate into a small areas, which are referred to as “hot spots” of activity. If the simulation’s load per Logical Process is uneven, its performance suffers. Figure 8.1 illustrates this degradation for a case in which odd LPs have one and a half times more load than even ones.

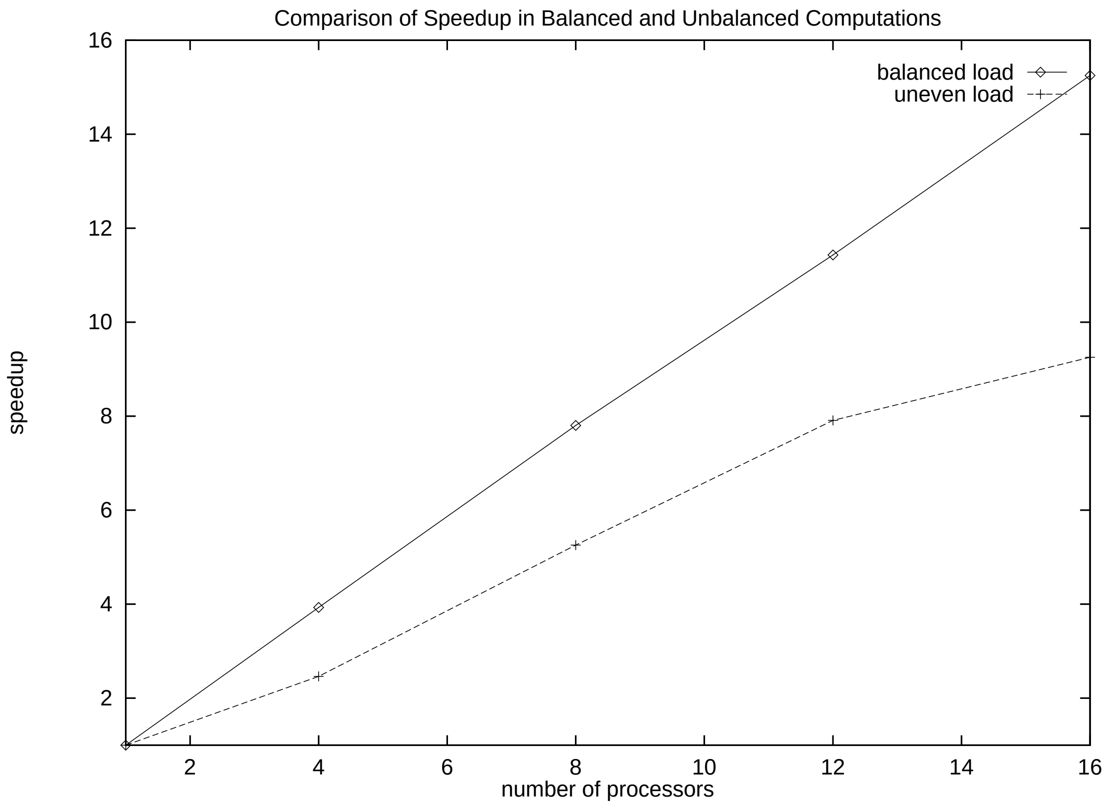


Figure 8.1: Speedup for Balanced and Unbalanced Computations.

8.2 Load Balancing for General Problems

Load balancing is often used in parallel computation to improve program performance. When the load is not evenly distributed, processes proceeding quickly through the computation might need to wait for the slower processes, thus slowing down overall system performance. If there is *a priori* knowledge about the distribution of data in the program, static load balancing can be applied to the processes in the system prior to the actual computation. The new load distribution can even be computed on a sequential machine before the start of the parallel execution. Frequently, however, *a priori* knowledge is not available, or the problem is dynamic. That is, even if the load is initially distributed evenly, it may become imbalanced during the computation. In such situations, dynamic load balancing is often advantageous or even necessary. Dynamic load balancing monitors or calculates the load of the system during the computation and shifts the load between processors while the program is running. This method of load balancing also involves making decisions about how frequently to balance the load or to monitor the balance.

Dynamic load balancing algorithms can be divided into nearest-neighbor techniques, in which load is moved between neighboring processors, and global techniques, in which load can be moved from one processor to any other processor in the system [73]. Nearest-neighbor algorithms are, by nature, iterative; thus, a few iterations of the algorithm may be required to load-balance the entire system. Iterative techniques can use the *diffusion*, *dimension exchange*, or *gradient* models to decide how the load should be moved. In the diffusion model [74], each processor "diffuses" some of its load to its neighbors, while simultaneously requesting some load from other neighbors. The dimension exchange method [75] is similar to the diffusion method, but the processor balances its load with that of its neighbors one by one dimension in multidimensional grids or hypercubes. The gradient method restricts the movement of the load in the direction of the "heaviest" processor [76].

In addition to deciding which processes should participate in the load balancing, it is also necessary to decide how the load will be measured, which, in turn, dictates what load should be moved. In models where there is one process per processor, the amount of data belonging to a process is the indicator of a processor's load. The amount of data, its distribution, and any of its characteristics that might influence inter-processor communications, are taken into consideration when deciding how much and which data to move between processors. In models where there is a many-to-one process-to-processor mapping, it is natural to migrate processes between processors in order to achieve an even load distribution in the system. The challenge of this approach is to estimate the "weight" of a given process. Also, inter-process communications have to be examined in order to keep the communication costs low. For example, if two processes are communicating frequently with one another, it is often advantageous to keep them on the same processor.

8.3 Load Balancing in Parallel Discrete Event Simulation

Generally, CPU utilization is a good measure of how much load a given processor has; however, in PDES, this measure is meaningless. If, for simplicity, there is one LP per processor, it is possible that that processor has a high CPU utilization but that it is continuously processing forward ahead of the others and then rolling back, thus performing no useful work. A metric often used in PDES is the number of committed events (events with timestamps smaller than the GVT). Much work in load balancing for PDES has been done using the many-to-one process-to-processor mapping; therefore, process migration algorithms have been the most frequently researched. Both static and dynamic algorithms have been designed.

In static techniques the simulation is run once. Data about the number of events that each LP has processed, as well as the number of messages each has sent, are gathered. This data is used in three different load balancing algorithms. The first algorithm (LBalloc1) places the LPs in an ordered list according to the processing time [77]. This list is then traversed, and the heaviest, unplaced LP is assigned to the processor with the smallest cumulative load. The second algorithm places the LPs in a chain which is then partitioned into as many subchains as there are processors [78]. The goal of the linearization is to keep heavily communicating processes next to each other in the chain. The last algorithm improves on the previous method by assuring that there will not be either two heavily loaded or two lightly loaded LPs next to each other in a chain [77]. This algorithm reduces the possibility of many heavy processes being placed on the same processor.

Another load measure used in dynamic load balancing is the simulation advance rate [79], defined as the rate at which a process advances its LVT as a function of the amount of CPU time it is given. The goal is to make the LPs progress evenly in the simulation time. The slower LPs have proportionally more CPU time dynamically assigned to them then the faster LPs. Process migration has been proposed for cases in which the load imbalance is too great and cannot be accommodated by CPU time slice adjustment. This method is applicable only to distributed memory systems, where the many-to-one LP-to-processor mapping is used. The load balancing is implemented by background processes running on each processor. This approach, however, might be too system-intrusive and might be difficult to implement outside of a simulated parallel system [79].

Finally, since LPs are often clustered on processors (see section 6.5.1), dynamic techniques have been designed specifically for such systems [69]. In such cases entire clusters of LPs are moved between processors. The load of a cluster can be defined as the number of events that have been processed by all the LPs in a given cluster since the last load balancing calculation. In addition to event messages, stragglers and rolled back events are counted. The load balancing algorithm matches up the heaviest and lightest processors, and attempts to migrate the clusters so that the load of the processors is approximately equal. This process is done iteratively until the system is balanced within a predefined tolerance. This algorithm also takes into account communication costs. When a cluster is targeted from the heavily loaded processor, the cluster which would generate the least communication overhead is picked for migration.

8.4 Dynamic Load Balancing in Spatially Explicit Problems

In the research presented in this thesis, a different approach to dynamic load balancing is used. Since the best performance results were obtained using the Breadth-First Rollback algorithm with a one-to-one LP-to-processor mapping, the load balancing algorithm is designed to be used with that model. Therefore, no process migration is performed. Instead, data belonging to the LPs is migrated between processes. Since data are being moved rather then processes, the load metrics used in process-migration-oriented load balancing cannot be expected to apply here. In this thesis, a load balancing algorithm is presented that requires 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).

It is important to decide how to calculate and move the load. There are several metrics that can be used to determine the load of a given LP. One possibility is to count the number of objects in the space assigned to a given LP. The advantage of this method is that this count can be easily obtained. A possible disadvantage is that this metric may not give an accurate representation of the load. If, for example, there are many objects in a given space, but there are no events scheduled for these objects, then the LP will not have much work. It is also possible that an LP could contain few but "very active" objects, and it will be busy. A better approach would be to count the number of events that are scheduled for a given LP. Here, also, some considerations have to be taken into account. It is possible that an LP (say LP_1) has few events scheduled for the immediate future but many events scheduled for the far future, and another LP (call it LP_2) has many events to process soon. Even if the event count is equal for the two LPs, the workload is not equal, because LP_1 will process events far into the future and incur rollbacks, thus performing useless work. Meanwhile, LP_2 will be simulating immediate events. It would be more profitable to distribute the events in such a way that both LPs will work on the immediate future.

In the research presented here, events are counted in order to determine the computational load of an LP. 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, as in previously described load balancing algorithms, 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

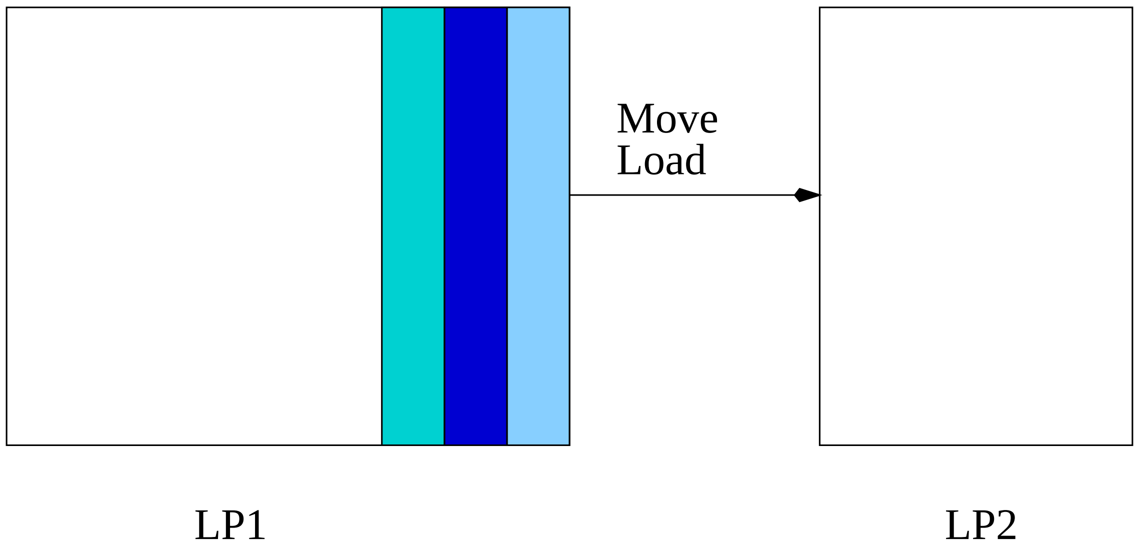


Figure 8.2: Moving Load between Logical Processes.

dealing with “hot spots.” In the previous approaches, the LP would have to be in the midst of the “hot spot” computation before the additional load would be discovered. However, since new events are constantly created, the future events represent only an estimate of the future load of an LP. The algorithm cannot, for example, judge if a given “hot spot” will be persistent or not. It is possible that the algorithm will balance the load so that the load of a “hot spot” is evenly distributed, only to discover that that concentration of events has dissipated, and a new rebalance is called for.

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.

An advantage of BFR is that it lends itself well to load balancing the space along with the objects and events, since the local (at the node level) history tracking facilitates load movement. An overloaded LP can “shed” layers of space in order to balance the load. For the remainder of this chapter, it is assumed that the spatial lattice of the problem is strip-divided in the horizontal dimension. Thus, the LP can send lattice columns to another LP (see Figure 8.2) without compromising the shape of the partition. The columns have to be moved in an organized fashion—from the

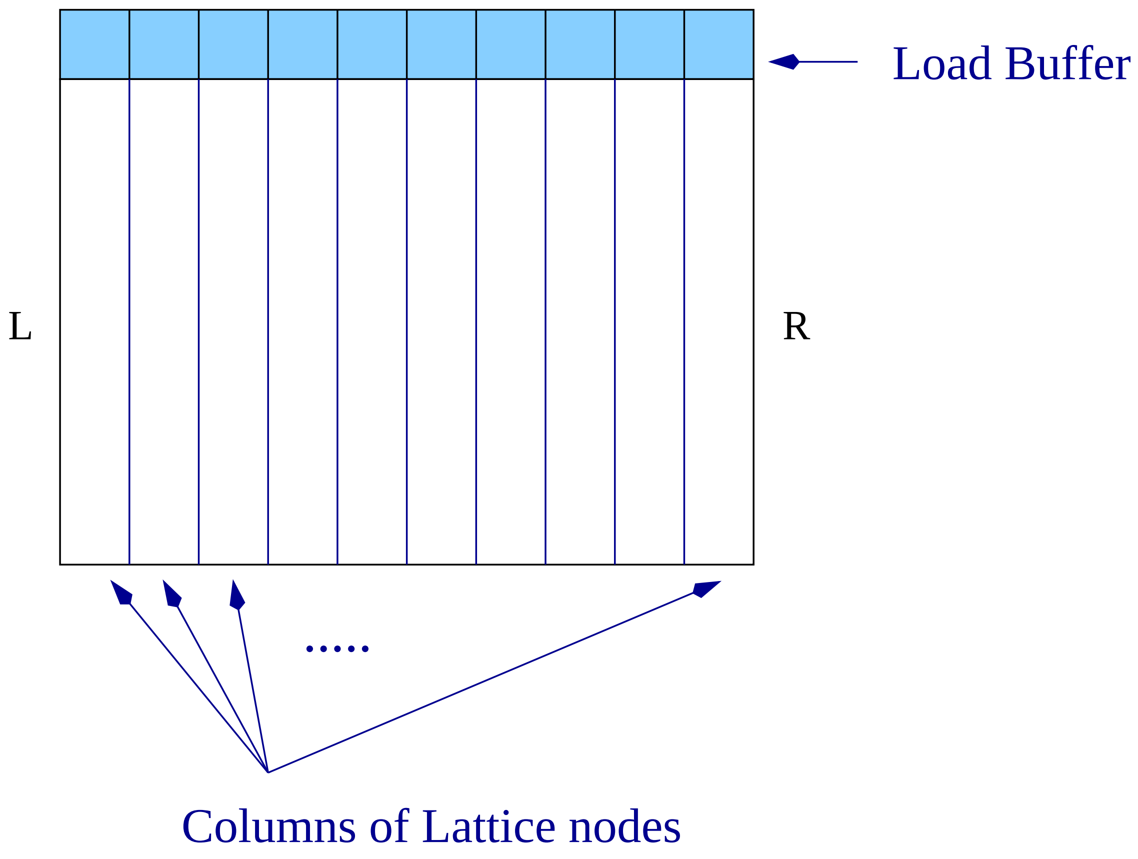


Figure 8.3: Counting the Computational Load of a Logical Processes.

outer edge toward the inside. Moving columns in any other way would result in a fragmented space and would increase the communication overhead. Although the communication cost is not explicitly included in the load calculation, it is implicit in the type of load migration (space, objects, and events versus object-only migration) and in the type of space allowed to be moved (only contiguous space layers). The load balancing algorithm is integrated with the centralized GVT algorithm used in SPEEDES (see section 3.5). Hence, this implementation gathers the load information together with the Local Virtual Time from all the LPs and then distributes this information back to LPs.

8.5 Load Calculation Phase

Each LP keeps track of the events occurring in each column of lattice nodes assigned to it (Figure 8.3). When an event is created, the cost of the event (defined below) is added to the load buffer 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. The load at each LP$_i$ is calculated as follows:

$$load_i = \sum_{L \leq j \leq R} E_j,$$

where, j is the column number, and L, and R are the space boundaries. $E_j = \sum_{0 \leq k < n(j)} 2^{(-time(e(k)))}$, where $n(j)$ is the number of events in column j, and $time(e)$ is the time of the occurrence of the event e.

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.

8.6 Calculation of the load distribution

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 finds the dominant ("heaviest") consecutive processes 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. The following paragraphs describe the details of the algorithm.

Consider a ring of n processes: $p_0, p_1, \ldots, p_{n-1}$. Each process p_i in the ring has the pre-balance load $a_i \geq 0$ and post-balance load b_i. One-step, nearest-neighbor communications is used for load balancing during which each process sends data to at most two neighbors and then receives data from at most two neighbors. Let x_i denote data exchanged between process p_i and process $p_{i \oplus 1}$ with the convention that $x_i > 0$ means that process p_i sends data and $x_i < 0$ signifies that process p_i receives data[3]. Accordingly, $-a_{i\oplus 1} \le x_i \le a_i$ and $b_i = a_i - x_i + x_{i\ominus 1}$.

Let $\overline{a} = \sum_{i=0}^{n-1} a_i/n = \sum_{i=0}^{n-1} b_i/n$ be the average load of the processes in the ring. The quality of the load balance can be measured by the load of the most loaded processes whose progress in the simulation will be slowest, so they will force under-loaded processes to roll back, resulting in wasted CPU time. Hence, the dynamic load balancing can be formulated as the following min-max problem:

Min-Max Problem: Given a ring of n processes with each process p_i processing load $a_i \ge 0$, for each pair of processes $p_i, p_{i\oplus 1}$ find a load transfer x_i that minimizes:

$$\max_{i=1}^{n-1}(a_i - x_i + x_{i\ominus 1}), \text{ under the constraints: } (\forall 0 \le i < n : -a_{i\oplus 1} \le x_i \le a_i).$$

This problem can be solved by the $O(n^2)$ algorithm presented below [80].

Let b_{opt} denote the optimum value of the maximum, post-balance load for the ring. That is $b_{opt} = \max_{i=0}^{n-1} b_i$, for optimal post-balance loads b_i. A *chain* $c_{k,l}$ in the ring is a sequence of processes $p_k, p_{k\oplus 1}, \ldots, p_{k\oplus(l-1)}$ starting at process p_k and of length l. A chain $c_{k,l}$ is a *subchain* of a chain $c_{K,L}$ iff for some i, $k = K \oplus i$, $0 \le i < L$, and $l \le L - i$.

Definition: The load $s_{k,l}$ of a chain $c_{k,l}$, where $2 < l \le n$ is defined as $s_{k,l} = \sum_{i=k\oplus 1}^{k\oplus(l-2)} a_i/l$.

Let $b_{max} = \max_{0\le k<n, 2<l\le n} s_{k,l}$. A chain $c_{k,l}$ is called maximal iff $s_{k,l} = b_{max}$ and *dominant* if it is the longest maximal chain.

Note that the load that can be transferred out of a chain, denoted here by $o_{k,l}$, is located at its end-processes (p_k and $p_{k\oplus(l-1)}$), so $o_{k,l} \le a_k + a_{k\oplus l\ominus 1}$. Hence, it is clear that $b_{opt} \ge b_{max}$, because for the dominant chain $c_{k,l}$, $b_{opt} \ge \sum_{j=l}^{k\oplus l\ominus 1} b_j/l = \sum_{j=k}^{k\oplus l\ominus 1} a_j/l - o_{k,l} \ge \sum_{j=k\oplus 1}^{k\oplus l\ominus 2} a_j/l = b_{max}$.

Theorem: $b_{opt} = \max(\overline{a}, b_{max})$.

Below, the theorem is proved constructively by providing an algorithm that defines valid load transfers to achieve the maximum post-balance load of $\max(\overline{a}, b_{max})$.

[3]In the following, operations $\oplus$ and $\ominus$ denote integer addition and subtraction in modulo n arithmetic.

First, it should be noticed that there are at most n^2 distinct subchains within a ring of size n, so by simple enumeration b_{max} and the dominant chain of the ring are found in $O(n^2)$ steps.

```
/* avg denoted the average load */
0. avg=a[0]/n; bmax=a[0]/3; ldominant=3; kdominant=n-1;
/* find maximal chains of length 3 */
1. for k=1 to n-1
     avg=avg+a[k]/n
     if (a[k]/3>bmax)
        bmax=a[k]/3
        kdominant=k-1
/* now go through longer chains */
2. load=a[n-1]
   for l=2 to n-2
       load+=a[l-2]
   /* try all possible positions */
   for k=0 to n-1
       load+=(a[(k+l-1)mod n]-a[(k-1)mod n])
       if (load>=bmax*(l+2)
          bmax=load/(l+2)
          ldominant=l+2
          kdominant=(k-1) mod n
/* end of algorithm */
```

By simply renumbering the nodes if necessary, it can be assumed without loss of generality that the dominant chain is $c_{1,ldominant}$, with length *ldominant*. If there is imbalance, then the computation of load transfers necessary to improve load balance depends on relation of the average load $\overline{a}$ and the load of the dominant chain.

Case 1: $b_{max} \geq \overline{a}$.

The load transfers are defined as follows:

1. For processes in the dominant chain the following loads are calculated:

$$x_0 = -a_1;\; x_{ldom} = a_{ldom};\; x_i = a_i - b_{max} + x_{i-1} \text{ for } i = 1, \ldots, ldom - 1.$$

2. The setting of the post-balance load to b_{max} extends iteratively beyond the right end of the chain:

$$x_i = \max(a_i + x_{i-1} - b_{max}, -a_{i\oplus 1}) \text{ while } b_i = b_{max}$$

   and an index of the last defined load transfer is denoted as re, so by definition $b_{re\oplus 1} = 0$.

3. If $re < n-1$, then another extension is defined iteratively through the chain's left end as:

$$x_i = \min(b_{max} - a_{i\oplus 1} + x_{i\oplus 1}, a_i) \text{ while } b_i = b_{max} \text{ for } i = n-1, n-2, \ldots .$$

   Similarly as in the previous case, an index of the last defined load transfer is denoted as le.

   If $le > re + 2$ then the algorithm is applied recursively to the (newly created) ring $p_{re+1}, \ldots, p_{le}$ in which the pre-balance loads on processes $re + 1$ and le is changed to 0. Otherwise $x_{re} = 0$.

4. To avoid unnecessarily underloaded processes, a simple adjusting step for processes outside the dominant chains can be performed. Each process sending out the load may check if the post-load at the destination will become bigger then its own and then change its transfer to have the post-loads equal. This step does not change the theoretical optimality of the algorithm but may slightly improve its performance and is inexpensive to execute. Note that this adjusting step preserves the correctness of the solution.

It follows directly from the transfer definition that for $0 < i < n$, $b_i \leq b_{max}$ and $b_0 = a_1 < b_{max}$, because chain $c_{0,ldom+1}$ is not maximal. Hence, the correctness of the above algorithm is established by the following lemma.

Lemma 1: The load transfers defined in the above algorithm satisfy all conditions of the min-max problem.

Proof: It has to be shown that $-a_{i+1} \leq x_i \leq a_i$.

First, let's consider the processes in the chain and process 0 (i.e., $0 \leq i \leq ldom$). From the iterative definition of x_i in the algorithm it follows that $x_i = \sum_{j=2}^{i} a_j - i * b_{max}$. The inequality $-a_{i+1} \leq x_i$ holds for $i = 0$ and $i = ldom$ by definition. Assume that this inequality holds for $i-1$ and consider transfer x_i for $0 < i < ldom - 2$. From the load in chain $c_{i+1,ldom-i}$ we have $b_{max} \geq (ldom * b_{max} - \sum_{j=2}^{i+1} a_j)/(ldom - i)$, so $\sum_{j=2}^{i+1} a_j \geq i * b_{max}$ and $\sum_{j=2}^{i} a_j - i * b_{max} \geq -a_{i+1}$ which proves that $x_i \geq -a_{i+1}$. We also have $x_{ldom-1} = ldom * b_{max} - (ldom-1) * b_{max} = b_{max} > -a_{ldom}$ as well as $x_{ldom-2} = 2 * b_{max} - a_{ldom-1} \geq -a_{ldom-1}$ which completes proof of this property.

Since $x_0 = -a_1$, we have $x_1 = -b_{max} < 0 \leq a_1$; also $x_{ldom} \leq a_{ldom}$. Assume that this inequality is true for all integers less than i, where $2 < i < ldom$ and consider a subchain $c_{1,i}$. From the load in chain $c_{1,i}$ we have $\sum_{j=2}^{i-1} a_j = i * s_{1,i} \leq i * b_{max}$. Hence, $x_i = \sum_{j=2}^{i} a_j - i * b_{max} = a_i + \sum_{j=2}^{i-1} a_j - i * b_{max} \leq a_i$. This completes the proof that the transfers inside the dominant chain are correct.

Now, consider a transfer x_i, where $ldom < i \leq re$. By definition, $x_i \geq -a_{i \oplus 1}$. It needs to be shown that $x_i \leq a_i$ for $ldom < i \leq re$ which, from the definition of transfers for this case, is equivalent to proving that $x_{i-1} \leq b_{max}$. To this end consider chain $c_{1,i}$. Since it cannot be maximal, we have $\sum_{j=2}^{i-1} a_j = i * s_{1,i} < i * b_{max}$. However, $x_{i-1} \leq \sum_{j=2}^{i-1} a_j - (i-1) * b_{max} < b_{max}$, proving this property for this case.

If necessary, the same argument applies to $le \leq i < n$ and the transfers for $re < i < le$ are correct by recursive application of the algorithm. □

Case 2: $\overline{a} > b_{max}$.

In this case, the transfers are defined as:

$$x_0 = \min_{i=1}^{n-1}(i\overline{a} - \sum_{j=0}^{i-1} a_i); \;\; x_i = a_i + x_{i-1} - \overline{a} \text{ for } i = 1, 2, \ldots n-1.$$

It is clear that for all $0 < i < n:\ b_i = \overline{a}$, therefore the correctness of the above algorithm is established by the following lemma.

Lemma 2: The load transfers defined in the above equations satisfy all conditions of the min-max problem and the post-balance load at process 0 is $\overline{a}$.

Proof: From the iterative definition of transfers, we have $x_i = \sum_{j=1}^{i} a_j + x_0 - i * \overline{a}$, so $b_0 = a_0 - x_0 + x_{n-1} = -x_0 + \sum_{j=0}^{n-1} a_j + x_0 - (n-1) * \overline{a} = \overline{a}$.

By definition, $x_0 \leq a_0$ and $x_0 \leq i*\overline{a} - \sum_{j=1}^{i-1} a_j$, so also $x_i = \sum_{j=1}^{i} a_j + x_0 - i*\overline{a} \leq a_i$. It needs to be shown that $x_i \geq -a_{i\oplus 1}$ for all $0 \leq i < n$, or, equivalently, that $x_0 \geq -a_{i\oplus 1} + i * \overline{a} - \sum_{k=1}^{i} a_k$. Assume that for some $0 \leq i < n$ this inequality does not hold and because x_0 is defined as $\min_{j=0}^{n-1}(a_j + j * \overline{a} - \sum_{k=1}^{j} a_k)$, then for some $0 \leq j < n$ we also have $x_0 = a_j + j * \overline{a} - \sum_{k=1}^{j} a_k$. As a result, it is assumed that for some i, j the following inequality holds:

$$a_j + j * \overline{a} - \sum_{k=1}^{j} a_k < -a_{i\oplus 1} + i * \overline{a} - \sum_{k=1}^{i} a_k. \tag{8.1}$$

If $i \leq j \leq i\oplus 2$ then inequality (8.1 simplifies to $a_j + a_{i\oplus 1} + (j-i)*\overline{a} < \sum_{k=i\oplus 1}^{j} a_k$ contradicting the assumption that $a_i \geq 0$ for all $0 \leq i < n$ and its corollary that $\overline{a} \geq 0$.

If $j > i + 2$, then consider chain $c_{i+1,j-i}$. Inequality (8.1) simplifies in this case to $(j-i)\overline{a}* < \sum_{k=i+2}^{j-1} a_k$, contradicting the assumption that $\overline{a} > b_{max}$.

Finally, if $i > j$, then inequality (8.1) becomes $\sum_{k=j}^{i\oplus 1} < (i-j) * \overline{a}$, again contradicting the assumption that $\overline{a} > b_{max} \geq s_{i,n+j-i}(n+j-i)$. □

Since this algorithm is performed by all the LPs, each will know how the load is being distributed. A structure indicating load movement between neighbors is maintained. Since there is only one round of load migration, a few iterations may be required to distribute the load of a "hot spot". On the other hand, that spot might dissipate by the next load calculation due to the dynamic nature of the simulation.

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.

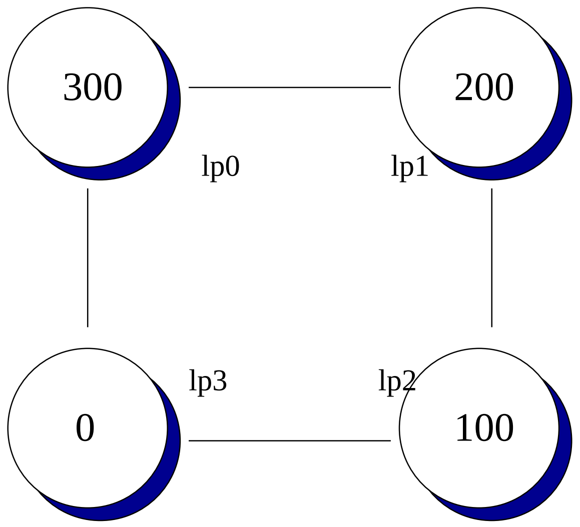


Figure 8.4: Original Load Graph. Example 1.

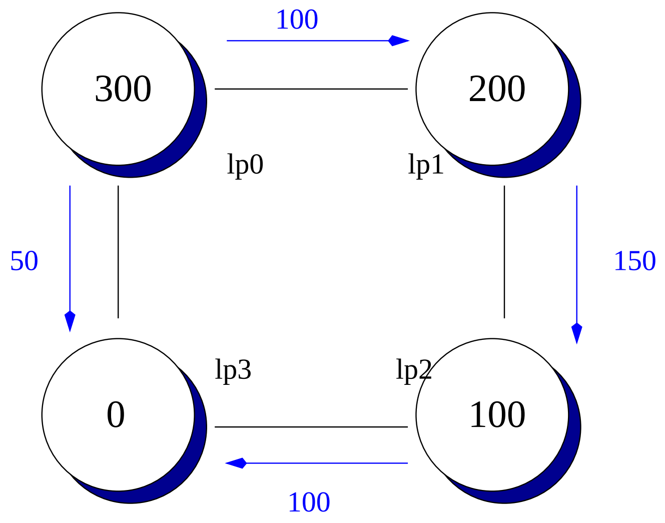


Figure 8.5: Load Balancing for Example 1.

8.6.1 Examples of Load Balancing

It is important to note that, although the algorithm described above calculates the necessary load movements, the actual sending and receiving of the load is not done until later, after all the calculations have been completed. Consider the load distribution as depicted in Figure 8.4. The system contains 4 LPs and the figure shows the load at each LP. The ring represents the communication connections between the LPs. During the first phase of the load calculation the loads of the LPs are gathered. The total load ($load_{system}$) is determined to be 600, making the average

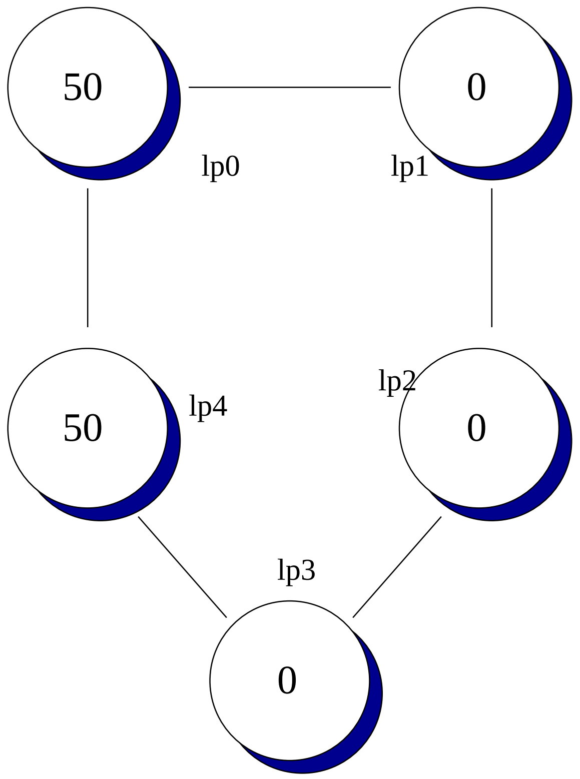


Figure 8.6: Original Load Graph. Example 2.

load ($\overline{load}$) 150. Now b_{max} is calculated. The dominant chain is $0-300-200-100$, giving $b_{max} = (300+200)/4 = 125$. The length of that chain is $l_{dominant} = 4$ and it starts at $k_{dominant} = 3$. Here, the dominant chain encompasses the entire ring. Now the chain is renumbered (to $c_1, 4$) to be able to calculate the loads, with index 1 now indicating the old p_3 (giving a chain starting with index 0, $100-0-300-200$. Since $bmax < \overline{load}$, the second case applies. The following flows are calculated:$x_0 = 100$, $x_1 = a_1+x_0-avg = 0+100-150 = -50$, $x_2 = a_2+x_1-avg = 300-50-150 = 100$, and $x_3 = a_3+x_2-avg = 200+100-150 = 150$. The flows are depicted in Figure 8.5. After load migration, each LP will have the same amount of load (150).

Sometimes, it is impossible to load balance a system fully in one round of communications. Figure 8.6 shows just such a system $(50-0-0-0-50)$, for which $\overline{load} = 100/5 = 20$. Here, the dominant chain starts at index $k_{dominantant} = 3$ and has a length of $ldominant = 4$; b_{max} is 25. The chain is renumbered, giving the chain $0-50-50-0$, starting with index 1. Since $b_{max} > \overline{load}$, the first case applies. The following flows are calculated: $x_0 = -a_1 = 0$, $x_4 = a_4 = 0$,

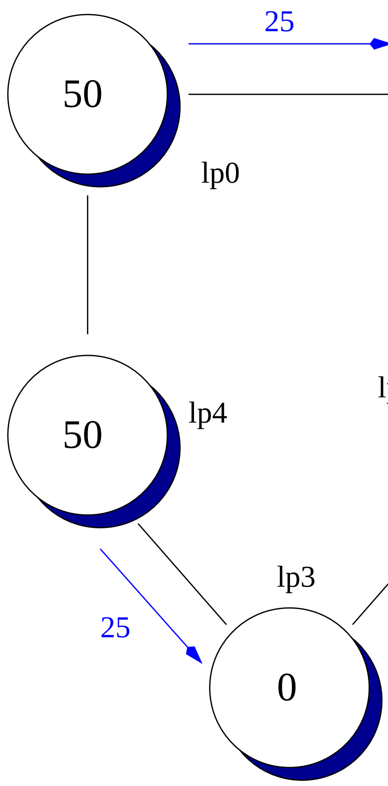


Figure 8.7: Load Balancing for Example 2.

$x_1 = a_1 - b_{max} + x_0 = 0 - 25 + 0 = -25$, $x_2 = a_2 - b_{max} + x_1 = 50 - 25 - 25 = 0$, and $x_3 = a_3 - b_{max} + x_2 = 50 - 25 + 0 = 25$. The flows are shown in Figure 8.7. The best load balance that can be achieved in this ring is $25 - 25 - 0 - 25 - 25$.

Load–Column Mapping So far, the load movement was calculated by the LPs. Now that each LP knows how much load it has to give away, it needs to calculate which columns of space it will actually send. It scans the load buffer of event costs. It should err on the side of giving too little rather than too much. An error exists with the projection of a one-dimensional structure (the load buffer) onto a scalar—load that needs to be moved. As a result, the movement of columns will generally not give an exact load balance.

8.7 Load Migration

Now the load migration has to be performed. Since a process cannot send and receive load simultaneously, a schedule has to be chosen. For example, LPs with

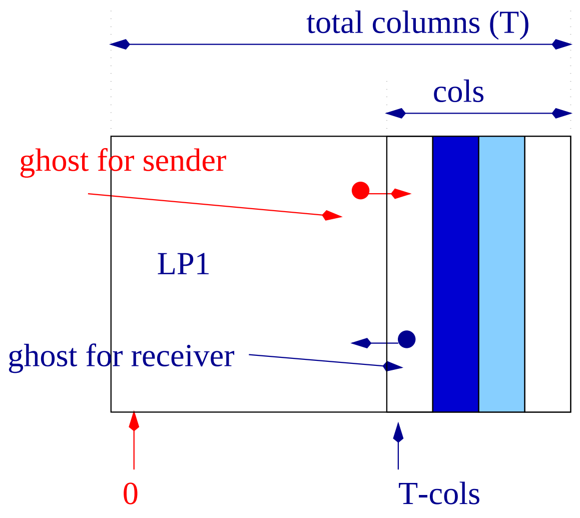


Figure 8.8: Moving the Load to the Right.

odd ids start sending the load to their neighbor, and LPs with even ids start by receiving load from their neighbor. Then the roles are reversed.

When sending load, columns of lattice nodes will be sent, in addition to all the objects in that space and the events associated with these objects. 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. Let's assume that LP_1 is sending columns of space to the right, as shown in Figure 8.8. On the right, the new boundary for LP_1 is $T - cols - 1$, where T is the initial, total number of columns, and $cols$ is the number of columns being sent.

It is possible that there were events already processed that involved the move of an object from column $T - cols - 1$ to column $T - cols$. In that case, a new ghost has to be created and placed on the ghost list, so that when the $MoveOut$ event is undone (which will now be a move out of the local space), the object and its future events can be restored. When a ghost is created, the appropriate processed event lists and the Future Event Queue have to be scanned to retrieve future events for the object.

When the local ghost list is updated, 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 (from $T - cols$ to $T - cols - 1$). These objects have to made into ghosts for the receiving LP. These ghosts, along with the ghosts of the objects killed in the space being sent—columns:[T-cols,T)—are sent to the receiver. Finally, the processed events from that space are sent.

Load $LP_i \rightarrow LP_j$

- build ghost for sender (LP_i)
- send space objects
- send objects
- send future events for space objects
- build ghost for receiver (LP_j)
- send ghosts
- send processed events

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.

Receive Load $LP_i \leftarrow LP_j$

- receive space objects
- receive objects
- receive future events for space objects
- receive ghosts
- receive processed events

It is also possible to avoid sending "past" information associated with the space, such as processed event lists or ghosts, by rolling back the space being sent to the GVT and then sending that space.

8.8 Load Balancing Results

Several load balancing experiments were conducted. For the first two experiments, two adjacent processes were made "heavy" by assigning twice as many objects to the space belonging to these processes as were assigned to the remaining processes. Figure 8.9 shows the performance of the load balanced computation as compared to the unbalanced computation. A relatively small problem size of approximately 10,000 lattice nodes and 2,000 objects were used, and a tolerance of 30% was applied. For this configuration, the results are inconclusive. Apparently, the benefits of good load distribution are negated by the cost of load balancing. The problem was then increased to 40,000 lattice nodes and approximately 4,000 objects. The results are shown in Figure 8.10 for a series of tolerances ranging from 10 to 40 percent. For all the levels of tolerance used for this particular problem, the algorithm performs well. Also, for 20 or more processors, the load balanced algorithm outperforms the unbalanced version. It should be pointed out that the load imbalance decreases as the number of processors increases, because the size of the space assigned to each LP decreases (with only two LPs heavily loaded). So, in fact, the load is highly imbalanced in the 16 processor case, and the cost of the load movement overpowers the benefits of an even load distribution.

Figures 8.11 and 8.12 illustrate the results of experiments for which the load was kept constant by making 100 (25% of columns for small problem size and 12.5 % for large problem size) of the middle columns of spaces "heavy"—containing twice as many objects as other columns. Figure 8.11 shows the results for the small problem size. Again, 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 Figure 8.12), the load balancing algorithm performs well from 16 to 28 processors.

In conclusion, 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.

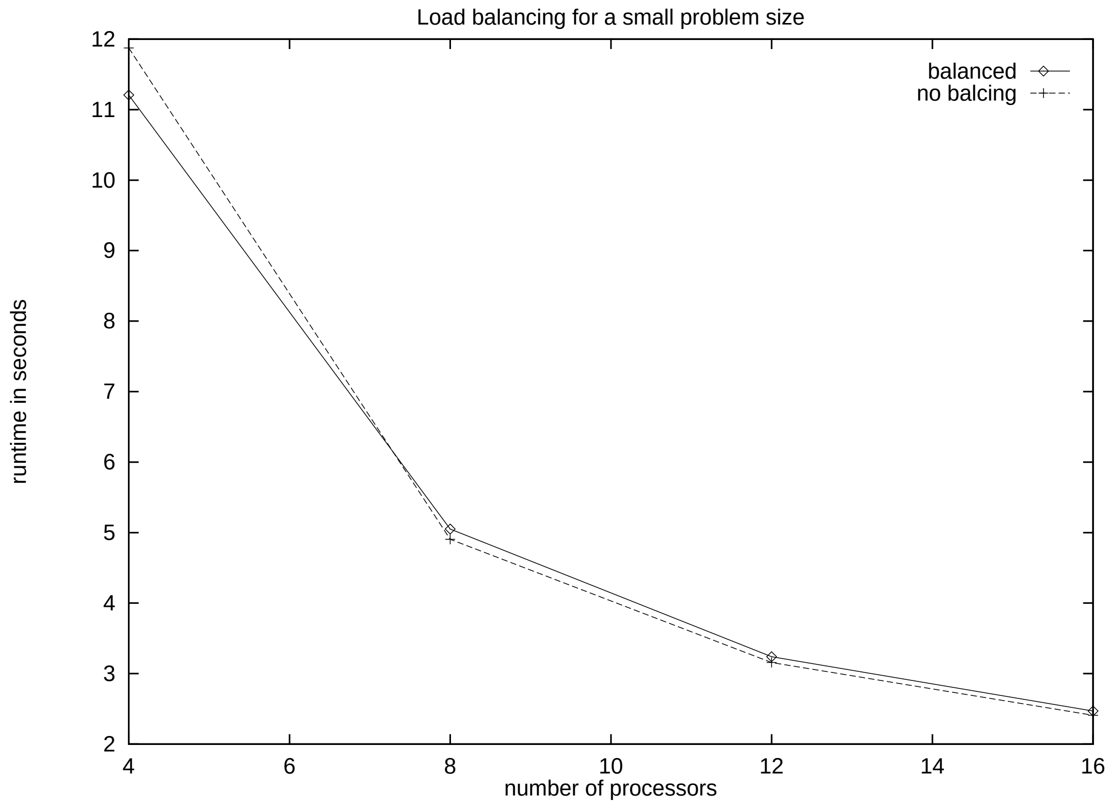


Figure 8.9: Load Balancing For a Small Problem Size and Two Heavy Processes.

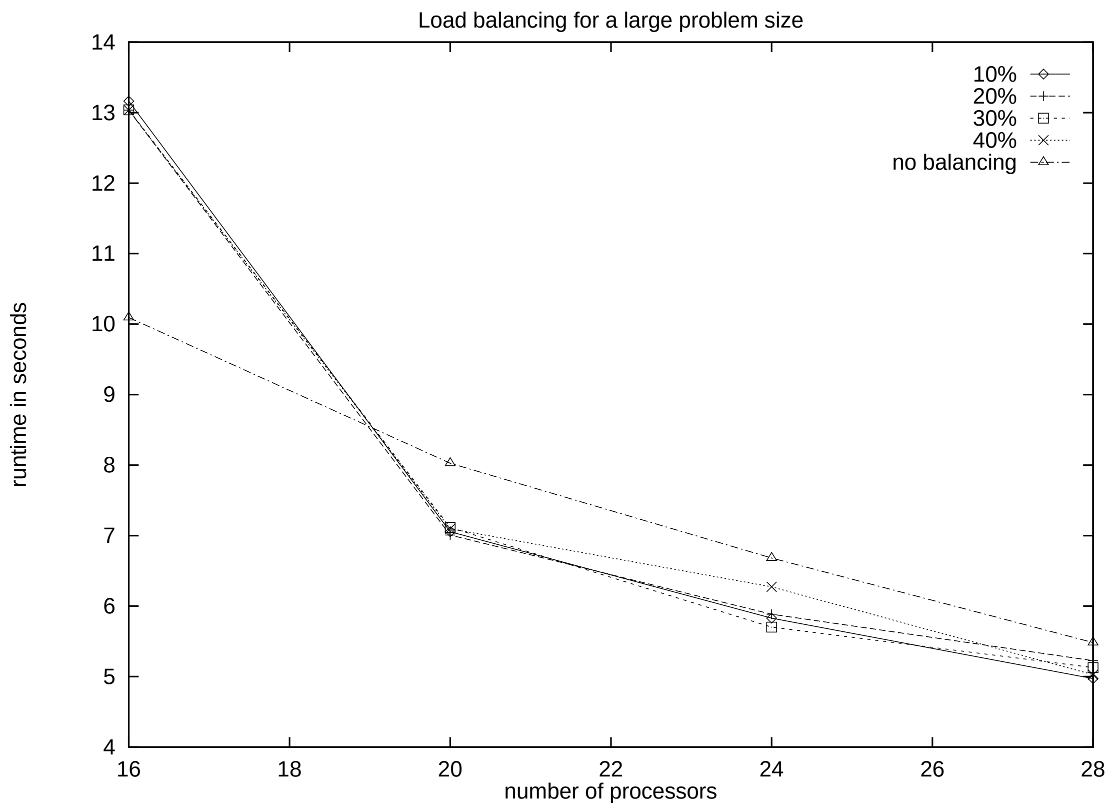


Figure 8.10: Load Balancing For a Large Problem Size and Two Heavy Processes.

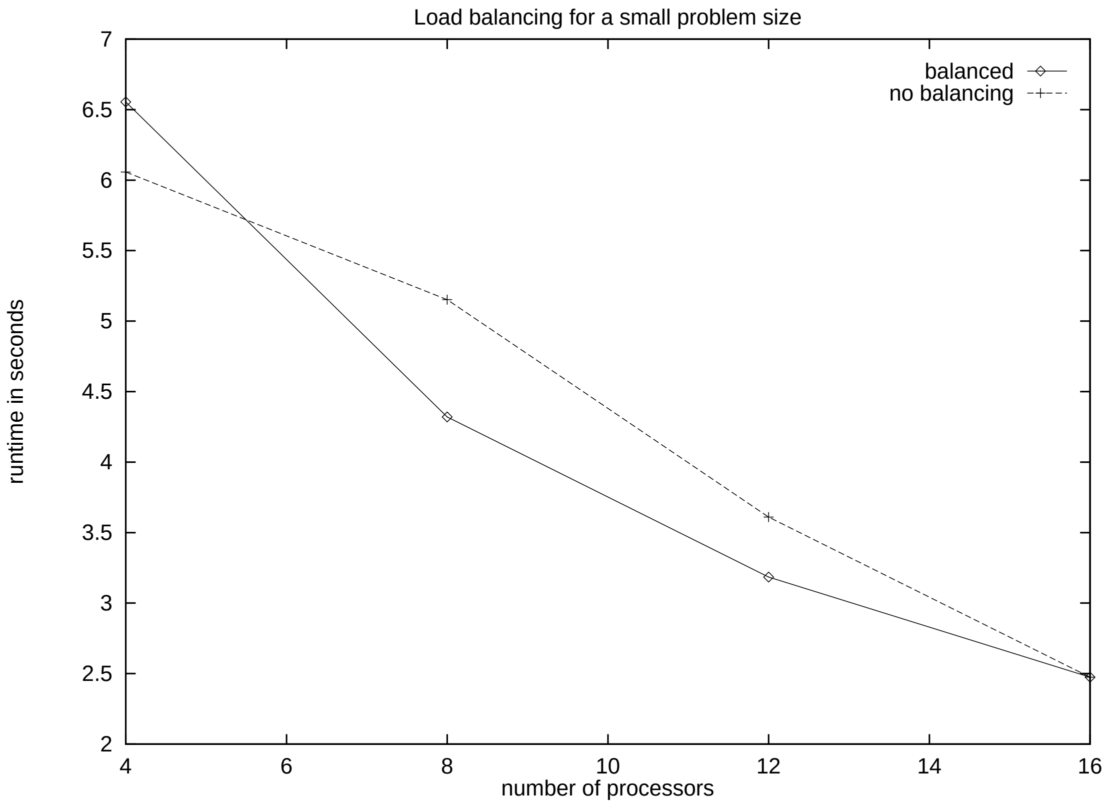


Figure 8.11: Load Balancing For a Small Problem Size and Heavy Lattice Columns.

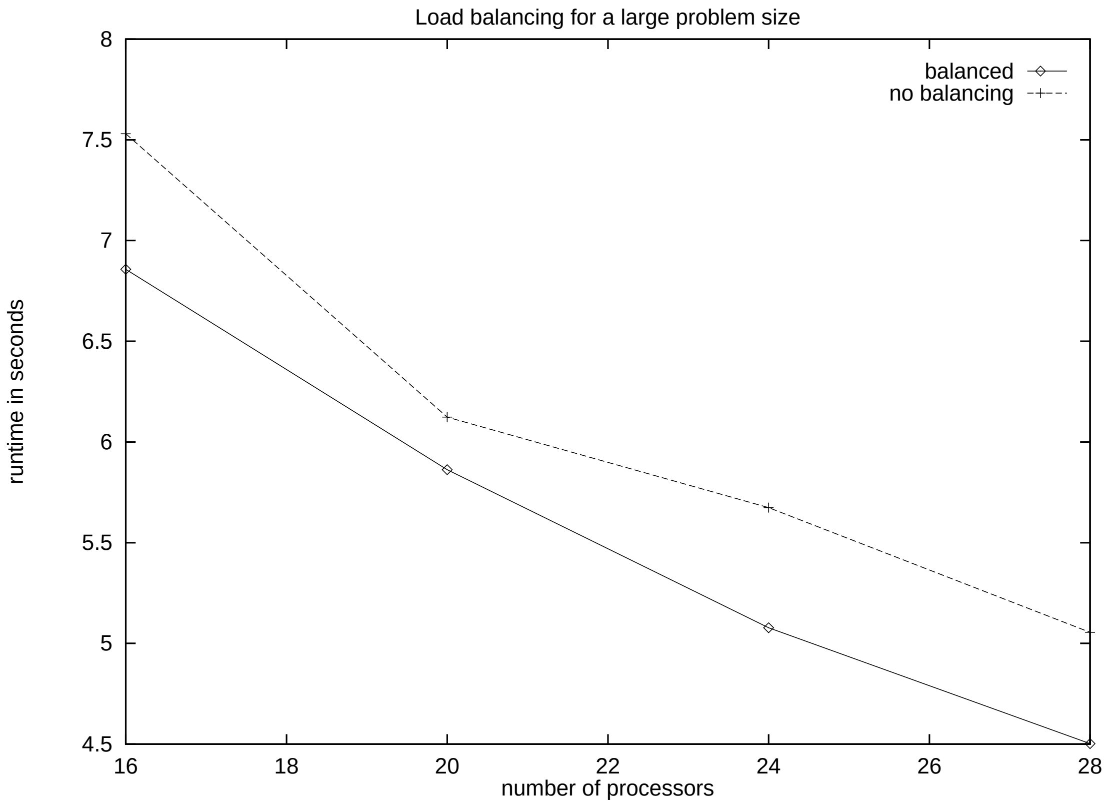


Figure 8.12: Load Balancing For a Large Problem Size and Heavy Lattice Columns.

CHAPTER 9
CONCLUSIONS AND FUTURE WORK

9.1 Computer Science Contributions

Important simulation domains such as battlefield, air-traffic-control, or ecological simulations can be categorized as spatially explicit problems. In such simulations, it is vital to capture the spatial complexity in order to be able to simulate the problem correctly and efficiently. Parallel Discrete Event Simulation lends itself well to this class of problems, because PDES enables the simulation engine to adapt to the heterogeneous aspect of spatially explicit simulations. It is important to be able to design simulation algorithms that are designed to optimize the performance of this general class of problems.

The work presented in this thesis focused on spatially explicit problems which are characterized by

- a continuous space in which objects reside and move
- a non-homogeneous environment which leads to the development of "hot spots" of activity
- local interactions between objects, which leads to nearest-neighbor communications between processes
- a small delay between events

Based on these characteristics of spatially explicit problems, the research presented here described and implemented algorithms that improve the performance of simulations for this class of problems. The performance benefits were obtained by developing three new algorithms for optimistic Parallel Discrete Event Simulation. A new Global Virtual Time calculation algorithm has been created—the Continuously Monitored Global Virtual Time (CMGVT) algorithm. The CMGVT algorithm is designed to make use of the messages being sent between Logical Processes. It appends to the messages Local Virtual Time information along with essential information about the outstanding messages in the system. This algorithm performs well when compared to a well-known GVT algorithm. Three versions of the algorithm were described and shown to perform well under various circumstances. Some variants were better suited for systems where memory is an issue, and some gave better performance when memory is not restricted.

A new algorithm, Breadth-First Rollback, was designed to minimize the impact of rollbacks. The simulated space is discretized and divided into as many areas as there are processors. Each area is simulated by an LP, each with a dual nature. From the point of view of the future, the LP is viewed as a single entity, but from the point of view of the past each LP is a cluster of LPs, each simulating a different unit of space. In this algorithm, if a rollback affects a given spatial area, the events that occurred in that area are examined, and their impact on the neighboring locations is analyzed. If there are no dependencies between locations, only the events in the initial space are rolled back. BFR reduces the number of events that are rolled back in a given LP, resulting in a significant performance improvement and a close-to-linear speedup.

A dynamic load balancing algorithm was designed for spatially explicit problems. The load is balanced between the LPs by sending the space, the objects, and the future events of the objects between LPs. The load balancing algorithm uses information about the future events scheduled for the LPs to predict the load of the LPs. Events closer to the beginning of the simulation are given more weight than the events farther into the future. A centralized algorithm is integrated into the GVT calculation phase and is used to determine the load of the system and the necessary load movements. The load itself is migrated between neighboring processes.

A simulation system has been designed to support spatially-explicit, ecological simulations. In particular, simulating the spread of Lyme disease in nature has been implemented. The goal of this system is to provide an understanding of the mechanisms that drive the disease at the level of its basic components—mice, deer, and ticks.

9.2 Open Problems

Although the Continuously Monitored Global Virtual Time algorithm is used for a class of problems where interactions are localized, it might be interesting to investigate how the three variants of the algorithm behave for long-range interactions. It might be the case that inferences made by the TRANSITIVE version of the algorithm give a better GVT estimate than the DIRECT method, where only knowledge about neighboring processes is exchanged.

The Breadth-First Rollback algorithm proved to be successful in the simulation of spatially explicit problems. It is believed that this method might achieve good performance in other problem domains, such as the simulation of digital circuits [81]. When an LP simulates a sub-circuit composed of several logical gates, it might not be necessary to roll back the entire LP—only the affected gates need to be rolled back. From the point of view of the future, one LP would simulate the entire sub-circuit, but from the point of view of the past, each gate would be an LP. The dependencies in this case are directed from the output of one gate to another.

The load balancing algorithm was designed in a centralized way. An attractive area to explore would be to integrate load balancing with the asynchronous CMGVT calculation. Load information can be added to the GVT information in the messages, and neighboring LPs could exchange load between one another if an imbalance is detected. The advantage of the decentralized approach would be improved scalability and minimized synchronization between processes.

It would be interesting to design and integrate into the simulation engine an interactive module which would allow the user to interact with the simulation. Allowing the user to view the simulation as it evolves, to be able to add or to remove events, or to zoom in on a given area of space, might help the user to better understand the issues involved in a given application. A related idea is to incorporate “rewinding”. It might be illuminating for a user to stop the simulation at a given point in simulated time and “rewind” a bit to see how a given result came about. The rewind capability can be easily supported by the rollback mechanism, which is already an integral part of the system.

LITERATURE CITED

[1] C.G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis.* 1993.

[2] U. W. Pooch and J.A. Wall. *Discrete Event Simulation: A practical Approach.* CRC Press, 1993.

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

[4] R. M. Fujimoto. Parallel and Distributed Simulation. *Winter Simulation Conference*, pages 118–125, 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] D.R. Jefferson. Virtual Time. *Trans. Prog. Lang. and Syst.*, 7:404–425, 1985.

[7] R.M. Fujimoto. Parallel and Distributed Discrete Event Simulation Algorithms and Applications. *Winter Simulation Conference*, pages 106–114, 1993.

[8] B. Beckman, M. DiLoreto, K. Sturdevant, P. Hontalas, L. Van Warren, L. Blume, D. Jefferson, and S. Bellenot. Distributed simulation and Time Warp Part 1: Design of *Colliding Pucks*. *Distributed Simulation*, pages 56–60, 1988.

[9] P. Hontalas, B. Beckman, M. DiLoreto, L. Blume, P. Reiher, K. Sturdevant, L. Van Warren, J. Wedel, F. Wieland, and D. Jefferson. Performance of the Colliding Pucks simulation on the time warp operating systems. *Distributed Simulation*, pages 3–7, 1989.

[10] D.M. Nicol and Scott E. Riffe. A "conservative" Approach to Parallelizing the Sharks World Simulation . *ICASE Report No. 90-67*, 1990.

[11] R.V. Hanxleden and L.R. Scott. Load Balancing on Message Passing Architectures. *Journal of Parallel and Distributed Computing*, pages 312–323, 1991.

[12] 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. *Distributed Simulation*, pages 14–20, 1989.

[13] J. B. Hiller and T. C. Hartrum. Conservative Synchronization in Object-Oriented Parallel Battlefield Discrete Event Simulations. *Workshop on Parallel and Distributed Simulation*, pages 12–19, 1997.

[14] M. Ebling, M. Di Loreto, M. Presley, F. Wieland, and D. Jefferson. An ant foraging model implemented on the time warp operating system. *Distributed Simulation*, pages 21–26, 1989.

[15] Y.B. Lin and P.A. Fishwick. Asynchronous Parallel Discrete Event Simulation.

[16] 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.

[17] 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. *Workshop on Parallel and Distributed Simulation*, pages 187–194, 1994.

[18] B. A. Malloy and A. T. Montroy. A Parallel Distributed Simulation of a Large-Scale PCS Network: Keeping Secrets. *Winter Simulation Conference*, pages 571–578, 1995.

[19] R. Bagrodia, M. Gerla, L. Kleinrock, J. Short, and T.C. Tsai. A Hierarchical Simulation Environement for Mobile Wireless Networks. *Winter Simulation Conference*, pages 1354–1361, 1994.

[20] F. Wieland, E. Blair, and T. Zukas. Parallel Discrete-Event Simulation (PDES): A Case Study in Design, Development, and Performance Using SPEEDES. *Workshop on Parallel and Distributed Simulation*, pages 103–110, 1995.

[21] R. Schlagenhaft, M. Ruhwandl, C. Sporrer, and H. Bauer. Dynamic load balancing of a multi-cluster simulator on a network of workstations. *Workshop on Parallel and Distributed Simulation*, pages 175–180, 1995.

[22] K. Glass, M. Livingston, and J. Conery. Distributed Simulation of Spatially Explicit Ecological Models. *Workshop on Parallel and Distributed Simulation*, pages 60–63, 1997.

[23] E. Deelman, T. Caraco, and B. K. Szymanski. Parallel Discrete Event Simulation of Lyme Disease. *Pacific Biocomputing Conference*, pages 191–202, 1996.

[24] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI.* The MIT Press, 1994.

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

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

[27] J. S. Steinman. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. *Workshop on Parallel and Distributed Simulation*, pages 95–103, 1991.

[28] R. Ronngren, J. Montagnat M. Liljenstam, and R. Ayani. Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation. *Workshop on Parallel and Distributed Simulation*, pages 70–77, 1996.

[29] Automatic Incremental State Saving. D. West and K. Panesar. *Workshop on Parallel and Distributed Simulation*, pages 78–85, 1996.

[30] J. S. Steinman, C. A. Lee, L. F. Wilson, and D. M. Nicol. Global Virtual Time and Distributed Synchronization. *Workshop on Parallel and Distributed Simulation*, pages 139–148, 1995.

[31] Y.B. Lin. Memory Management Algorithms for Optimistic Parallel Simulation. *Workshop on Parallel and Distributed Simulation*, pages 43–52, 1992.

[32] S.R. Das and R.M. Fujimoto. A Performance Study of the CancelBack Protocol for Time Warp. *Workshop on Parallel and Distributed Simulation*, pages 135–142, 1993.

[33] B.R. Preiss and W.M. Loucks. Memory Managment Techniques for Time Warp on a Distributed Memory Machine. *Workshop on Parallel and Distributed Simulation*, pages 30–39, 1995.

[34] V. K. Madisetti and D. A. Hardaker. The MIMDIX Environment for Parallel Simulation. *Journal of Parallel and Distributed Computing*, 18:473–483, 1993.

[35] S. Bellenot. State Skipping Performace with the Time Warp Operating System. *Workshop on Parallel and Distributed Simulation*, pages 53–61, 1992.

[36] Y.B. Lin, B.R. Preiss, W.M. Loucks, and E.D. Lazowska. Selecting the Checkpoint Interval in Time Warp Simulation. *Workshop on Parallel and Distributed Simulation*, pages 3–10, 1993.

[37] J. S. Steinman. Breathing Time Warp. *Workshop on Parallel and Distributed Simulation*, pages 109–118, 1993.

[38] P.M. Dickens, D.M. Nicol, P.F. Reynolds, and J.M Duva. Analysis of Optimistic Window-based Synchronization. *ICASE Report No. 94-27*, 1994.

[39] G. Tel. *Topics in distributed algorithms.* Cambridge University Press, 1991.

[40] T. Lai and T. Yang. On distributed snapshots. *Inform. Process. Lett.*, 25:153–158, 1987.

[41] F. Mattern. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing*, 18:423–434, 1993.

[42] D.M. Nicol. Global Synchronization for Optimistic Parallel Discrete Event Simulation. *Workshop on Parallel and Distributed Simulation*, pages 27–34, 1993.

[43] H. Bauer and C. Sporrer. Distributed Logic Simulation and an Approach to Asynchronous GVT-Calculation. *Workshop on Parallel and Distributed Simulation*, pages 205–208, 1992.

[44] A.I. Tomlinson and V.K. Garg. An Algorithm for Minimally Latent Global Virtual Time. *Workshop on Parallel and Distributed Simulation*, pages 35–42, 1993.

[45] 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.

[46] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, 1991.

[47] L. Lamport. Time, Clocks, and the Ordering of Events in Distributed Systems. *Communications of the ACM*, pages 558–565, July 1978.

[48] M. Raynal and M. Singhal. Logical Time: Capturing Causality in Distributed Systems. *IEEE Computer*, pages 49–56, February 1996.

[49] M. Raynal. About Logical Clocks and Distributed Systems. *Operating System Review*, 26:49–55, January 1992.

[50] S. K. Sarin and N. A. Lynch. Discarding Obsolete Information in Replicated Database System. *IEEE Transactions on Software Engineering*, pages 39–47, January 1987.

[51] Ewa Deelman and Boleslaw Szymanski. System Knowledge Acquisition in Parallel Discrete Event Simulation. *Proceedings of the 1997 IEEE International Conference on Systems Man and Cybernetics*, 1997.

[52] Ewa Deelman and Boleslaw Szymanski. Continuously Monitored Global Virtual Time Event Simulation. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1–10, 1997.

[53] A. Spielman, M.L. Wilson, J.F. Levine, and J. Piesman. Ecology of Ixodes dammini-borne human babesiosis and Lyme disease. *Annual Rev. Entomology*, 30:439–460, 1985.

[54] R.S. Lane, J. Piesman, and W. Burgdorfer. Lyme borreliosis: relation of its causative agent to its vectors and hosts in North America and Europe. *Annual Review of Entomology*, 36:587–609, 1991.

[55] S. Sandberg, T.E. Awerbuch, and A. Spielman. A comprehensive multiple matrix model representing the life cycle of the tick that transmits the agent of Lyme disease. *J. Theor. Biol.*, 157:203–220, 1992.

[56] H.S. Ginsberg. *Ecology and Environmental Management of Lyme Disease*, chapter Geographical spread of Ixodes dammini and Borrelia burgdorferi. Rutgers Univ. Press, New Brunswick, 1993.

[57] D.J. White, H.G. Chang, J.L. Benach, E.M. Bosler, S.C. Meldrum, R.G. Means, J.G. Debbie, G.S. Birkhead, and D.L. Morse. The geographic spread and temporal increase of the Lyme disease epidemic. *Journal of the American Medical Association*, 266:1230–1236, 1991.

[58] M.L. Wilson, A.M. Ducey, T.S. Litwin, T.A. Gavin, and A. Spielman. Microgeographic distribution of immature Ixodes dammini ticks correlated with that of deer. *Medical and Veterinary Entomology*, 4:151–159, 1990.

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

[60] F.S. Kantor. Disarming Lyme Disease. *Scientific American*, pages 34–39, September 1994.

[61] J.O. Wolff, K.I. Lundy, and R. Baccus. Dispersal, inbreeding avoidance and reproductive success in white-footed mice. *Animal Behavior*, 36:456–465, 1988.

[62] O.P. Judson. The rise of the individual-based model in ecology. *Trends in Ecology and Evolution*, 9:9–14, 1994.

[63] E. McCauley, W.G. Wilson, and A.M. deRoos. Dynamics of age-structured and spatially structured predator-prey interactions: individual-based models and population-level formulations. *American Naturalist*, 142:412–442, 1993.

[64] 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.

[65] E. Deelman and B. K. Szymanski. Simulating Lyme Disease Using Parallel Discrete Event Simulation. *Winter Simulation Conference*, pages 1191–1198, 1996.

[66] S. R. Das. Estimating the Cost of Throttled Execution in Time Warp. *Workshop on Parallel and Distributed Simulation*, pages 186–189, 1996.

[67] E. Deelman and B. K. Szymanski. Breadth-First Rollback in Spatially Explicit Simulations. *Workshop on Parallel and Distributed Simulation*, pages 124–131, 1997.

[68] 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.

[69] 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.

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

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

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

[73] C. Xu and F. C. M. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, 1997.

[74] G. Cybenko. Load Balancing for Distributed Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[75] S. Ranka, Y. Won, and S. Sahni. Programming a Hypercube Multicomputer. *IEEE Software*, 5:69–77, 1988.

[76] F. C. H. Lin and R. M. Keller. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering*, 13:32–38, 1987.

[77] L. F. Wilson and D. M. Nicol. Experiments in Automated Load Balancing. *Workshop on Parallel and Distributed Simulation*, pages 4–11, 1996.

[78] L. F. Wilson and D. M. Nicol. Automated Load Balancing in SPEEDES. *Winter Simulation Conference*, pages 590–596, 1995.

[79] D. W. Glazer and C. Tropper. On Process Migration and Load Balancing in Time Warp. *Workshop on Parallel and Distributed Simulation*, pages 318–327, 1993.

[80] B. K. Szymanski. private communication.

[81] R. Bagrodia. private communication.