Parallel Simulation of Billiard Balls using Shared Variables

by

Peter A. MacKenzie
School of Computer Science, McGill University
Montreal, Canada
June 1996

A Thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

ISBN 0-612-19833-2

ACKNOWLEDGMENTS

I gratefully acknowledge my parents for having made all my acheivements possible.

I would like to thank my thesis supervisor Professor Carl Tropper for his guidance, support, friendly encouragement and help with the manuscript.

I would also like to thank the School of Computer Science at McGill University for providing a productive enviroment for my studies and thesis work.

Finnally, I would like to thank everyone who helped me with my work in one way or another: Ben, Claire, Owain, Brett, Colin, Sean, Alberto, Herver, Azzedine, Luc, Ramesh, Kent, Rafa, Debbie, James, Lynne, Claire and Dave.

ABSTRACT

This thesis presents a conservative algorithm for the parallel simulation of billiard balls. Simulating billiard balls has become an important benchmark for parallel event driven simulation schemes. The approach distinguishes itself in that it makes use of shared variables to enable processors to ascertain the state of the computation at neighboring processors. The table is partitioned into segments which are simulated by different processors. The shared variable corresponds to a region at the boundary between table segments (referred to as the critical region). By making use of shared variables, a significant speed-up over the execution time of a purely conservative approach is obtained.

The algorithm was implemented on a BBN Butterfly, as was a purely conservative algorithm. In the purely conservative algorithm, a processor wishing to process a ball in the critical region waits until the neighbouring processor's simulation time is greater than the time of the event it wishes to process. In our experiments, we examined three population levels of balls- 2400, 4800 and 7200. These populations were chosen to reflect low, medium and high populations of balls. The shared-variable approach resulted in a 30 to 50 percent decrease in execution time with respect to purely conservative approach.

RÉSUMÉ

Cette thèse presente un algorithm conservative pour la simulation parallèle des balles de billiard. La simulation des balles de billiard est devenue un point de référence pour les plans de simulation des événements conduit en parallèle. La façon se distingue parce qu'il utilise les variables partagé pur permettre les processeur à etablissent l'état de la computation aux processeur voisin. La table est cloisonné en segments qui sont simulé par les processeur differents. La variable partagé correspond a une région à la limite entre les segments de table (appliqué comme la région critique). En utilisant les variables partagés, une accélération importante est obtenue par dessus le temp d'execution de la façon strictement conservative.

L'algorithm était executé sur un BBN Butterfly, comme l'algorithm strictement conservative. Dans l'algorithm strictement conservative, un processeur qui veut traiter une balle dans la region critique attend jusqu'a ce que le temp de simulation du processeur voisin est plus que l'événement qu'il veut traiter. En notre expériences, nous avons examiné trois niveaux de population des balles - 2400, 4800, 7200. Cettes populations étaient choisis pour refléter les populations des balles bas, moyens et élevés. La façon de la variable partagé a resulté en une diminution de 30 à 50 pour cent en le temp d'execution de la façon strictement conservative.

CONTENTS

LIST of FIGURES

1 Introduction

Simulation is a useful tool for evaluating many mathematical models of complex real world systems that cannot be evaluated analytically. However, large detailed simulations often require enormous amounts of CPU time. Simulations of large communication networks, gas and oil reservoirs, weather forecasts and military applications, to name a few, can require hundreds of hours of machine time to complete. Executing these simulations proved impractical for even the fastest available sequential machines. This in turn created a need for developing new methods to speed up simulations leading to the development of algorithms for parallel simulations, i.e. partitioning the simulation problem and executing the parts in parallel on multiple processors. Surveys of the area may be found in [Jeff85, Misr86, Gros87, Gross88, Righ89].

1.1 Principles of Parallel Simulation

There are three different types of simulations: continuous, discrete and hybrid. These types of simulations differ in how the state of an object in the system being simulated changes with respect to time. The state in a continuous simulation changes smoothly and continuously as time proceeds, e.g., the flow of liquid through a pipeline and weather modeling. Continuous simulation models often involve difference or differential equations that represent certain aspects of the system. In a discrete simulation, a state changes at a static point in time. Any queueing system such as a job shop would be an example of a discrete simulation. Many simulations are hybrid, i.e. different parts of the same system can be modeled either discretely or continuously. An unloading dock where tankers queue up to unload their oil through a pipeline is an example of a system suited to a hybrid simulation. General discussions of simulations methodology and simulation languages can be found in [Fish78].

A discrete event simulation models a physical system where the state changes at a discrete point in time - these changes in the system's state is referred to as an event. A parallel discrete event simulation must yield the same results as a serial simulation to be considered correct. The ordering of the events with respect to time is of paramount importance. In Figure 1.1, consider two events: E1 at logical process LP1 with timestamp 20 and E2 at LP2 with timestamp 30. Assume that processing E1 causes the creation of event E3 for LP2 with a timestamp less than 30. E3 might affect E2; for instance E3 could modify a state variable used by E2, necessitating sequential execution of all three events.

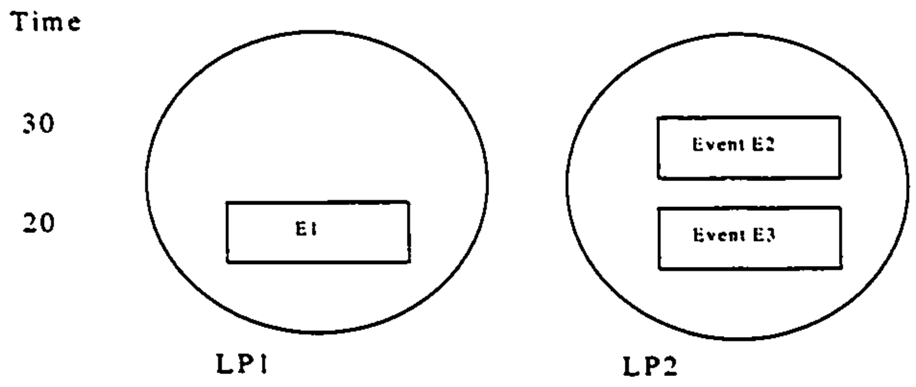

Figure 1.1 Sequence constraint of parallel simulation.

It is the difficulty in overcoming the sequencing constraints that makes parallel discrete event simulation a challenging problem and of interest to researchers in the parallel programming community at large for exploring the synchronization algorithms developed in parallel simulation.

Discrete event simulations can either be synchronous or asynchronous. In a synchronous algorithm, all the objects in the simulation progress through time at the same rate - each object simulates those objects that occur at the same time. In an asynchronous algorithm, different objects can be at different points in (simulated) time. Sowizral and Jefferson [Sowi82] showed that synchronous parallel simulation limits the amount of parallelism that can be exploited and does not produce significant speed ups for many common kinds of systems. We therefore consider only asynchronous discrete event simulation.

In a serial simulation the events are placed in a list and ordered according to event time. The head of the list is the event with the lowest event time. Several data structures have been proposed to represent the event set for discrete event simulation [Comf82] such as heaps and splay trees. A clock is maintained which represents the current simulation time. As the simulation progresses, the head of the list is removed, the simulation clock advances and the event is simulated. The execution of one event may change events later in the list or require new events be added to the list. It would appear that the only way to ensure that events are simulated chronologically, mimicking the real system, is to process them one at a time, resulting in a sequential program. Parallelism exists, however, when two events in the list do not depend on one another - these events can be simulated concurrently. However, if an event is executed earlier in real time and affects state variables used by a subsequent event with an earlier event time, then effectively an event in the future has influenced an event in the past. This is referred to as a causality error. Preventing causality errors in the absence of global time is the fundamental problem of parallel discrete event simulation.

The goal of parallel discrete event simulation [Misr86] is (a) to partition the system being simulated into relatively independent subsystems that communicate with each other in a simple manner (such as passing customers or jobs from one subsystem to another), and (b) to simulate each subsystem on a different processor. This means eliminating the event list and the clock in its traditional form.

In order to guarantee the correctness of the simulation, the following property [Misr86] must hold: the event message (e,t) is generated, received, processed, consumed (i.e. has reached its final destination) or transmitted, respectively, by LPi in the simulation system if and only if the event e was generated, received, processed, consumed or transmitted, respectively, by PPi in the physical system at time t. Details of this proof can be found in [Chand79, Misr86].

A simple parallel simulation can be constructed in which every physical process (PP) in the real system is modeled by a corresponding logical process (LP) [Chan79, Misr86]. For example, a car wash might consist of a pay booth and a number of washers and dryers. One LP could model the pay booth, while other LPs modeled each of the washers and the dryers. Any communication between physical processes is modeled by communication between logical processes within the simulation. There are no other processes that control the synchronization among various processes or any global time in the simulation shared by the processes.

A fundamental concept in parallel simulation is the timestamped message [Chan79, Misr86]. The interactions between physical processes in the real system are modeled by exchanging timestamped event messages between the corresponding logical processes. They also represent sending times of events and the service time added to the received time (when the event is in an output queue). They ensure that events are simulated in proper order [Chan79, Misr86] on each processor.

Instead of a global clock, each LP must maintain its own clock representing the local simulation time (LST). The LST is equal to the time when the last event (e) was processed, i.e. the time in the simulated system which indicates the end of an event that occurred at the corresponding PP. A new LST is determined after each event is processed. The new LST is computed as follows: LST = max(current LST, e.time), where e.time is the time of the last event. The system is simulated for some time period [0,T], where T is referred to as the simulation end-time or termination time. During this time, any LP sends a number of messages to other LPs.

For instance, as shown in Figure 1.2, LPi has two input buffers, one assigned for each link with another processor with events waiting to be processed. LPi selects the message with the smallest time stamp in its input buffers, therefore maintaining the sequencing constraint. Then LPi processes event e1, advances its LST to max(LSTi,e1.time) + Servicetime, where Servicetime represents the time that the physical process PPi would have spent to process the event e1 (e1.time = 5) and generates the new event e4. If Servicetime = 1, then e4.time = 6 and LPi's local simulation time becomes 6 (LSTi = 6).

We now address the fundamental problem of parallel simulation - causality errors. Without a global clock each LP's local time can be different; LPi can be further ahead in simulation time than an LP that sends messages to its input buffer. If for example LPi in Figure 1.2.b received event e2 and processed it before it received event e3 and e3 affected e2, then a causality error will have occurred.

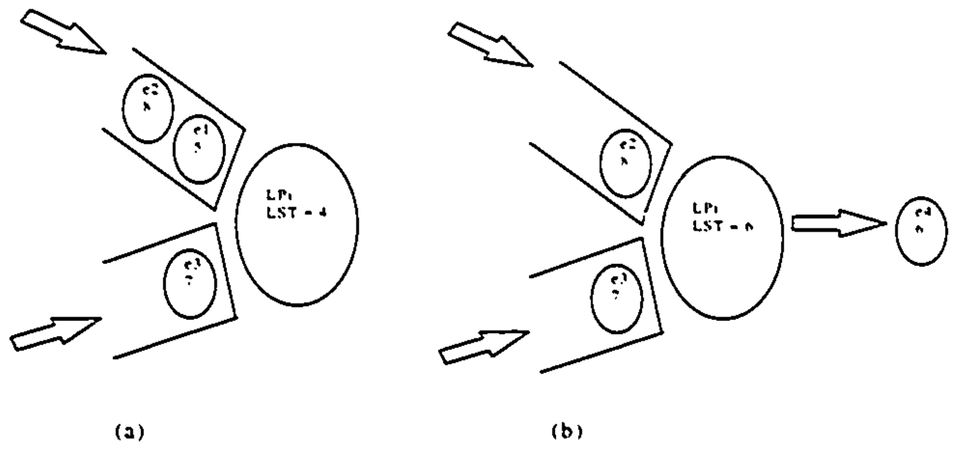

Figure 1.2 Processing at an LP

Techniques for parallel simulation can be classified into two groups, conservative and optimistic algorithms. Good surveys of the parallel simulation literature may be found in [Kaud87, Righ89, Fuji90]. The techniques differ primarily in how they handle causality errors. Conservative algorithms rely on blocking to synchronize processors, while optimistic algorithms rely on detecting synchronization errors at run time and on recovery using a rollback mechanism.

In the conservative approach [Misr86], each process is allowed to proceed if and only if it is certain that it will not receive an earlier event. Consequently, events are always executed in chronological order at any LP. An LP must block if it has an empty event queue and it is not certain if an event earlier than any other event on its other event queues will arrive. Therefore, only unblocked LPs can execute in parallel. In Figure 1.3, LPd must block because one of its input queues is empty. In this snapshot, LPb and LPc are both eligible to execute concurrently. If LPd were to process the message received from LPb with timestamp 30, it would risk receiving an earlier message from LPc with a timestamp of less than 30, causing the events to be processed out of order and risking a causality error.

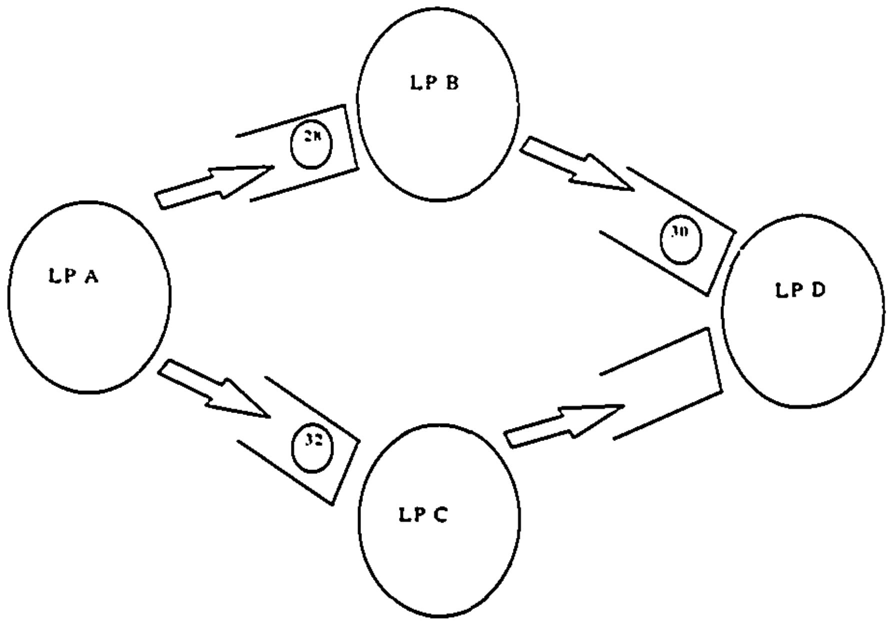

Figure 1.3 A blocked LP

The performance of the parallel simulation can be greatly influenced by the amount of blocking that occurs. If many LPs are forced to block, the simulation can be essentially serialized. If all of the LPs are blocked then the network is deadlocked. Ensuring synchronization and avoiding deadlocks are the central problems in the conservative approach. We survey some methods for conservative simulations in section 1.2 .

In the optimistic approach [Jeff85], a process receives messages from all the communicating LPs and places them in its input queue. Each LP processes events from its input queues until no messages remain or until a message arrives "in the past" (a straggler). A straggler indicates that a message has been processed out of order and the process execution must be interrupted so that a rollback can be performed in order to restore the states of the LPs to before the order was violated. Since events processed in one LP may create new events in other LPs that are sent via messages, those messages sent to other processors must be canceled. The rollback mechanism sends antimessages to those processors. Upon receiving an antimessage, a processor cancels the matching message in its input queue. There are two major versions of Time Warp. The original version is intended for a distributed memory architecture and performs cancellation by antimessages [Jeff82, Jeff85]. Another version is intended for a shared memory architecture and uses direct cancellation without antimessages [Fuji89].

Although Time Warp gains from being deadlock free, it is at the expense of spending a great deal of time rolling back and repeatedly simulating the same events. Another drawback of this approach is that it requires a large amount of memory to save the states of the LPs and an efficient garbage collection algorithm. Most studies of Time Warp have been experimental [Fuji89, Lin89, Made88, Reih90]. Despite these studies, the rollback mechanism is not well understood. A number of memory management schemes have been proposed to reduce the space usage of Time Warp. They range from those that reduce the average amount of memory needed but are unable to recover when it actually runs out of memory, to those that can run the simulation within a large amount of preallocated memory by recovering memory on demand [Das94]. Although there have been studies of the different memory management schemes [Lin92], little has been done in terms of the performance with respect to execution time [Das94].

The algorithm presented in this thesis is conservative in its nature. Given the problem it was designed for, we feel that ensuring synchronization by using shared variables and blocking is preferable to correcting causality errors with an optimistic approach. We therefore concentrate on the conservative approach.

The performance of a parallel simulation executed on a multiprocessor is determined by many factors. One important factor is the partitioning and mapping of processes onto the processors. Often the number of logical processes (LPs), n, is much larger than the number of processors k. In order to construct the simulation, the LPs must be partitioned into k components or clusters of processes and then assigned to the different processors. There has been a great deal of interest in partitioning processes [Bokh81, Bokh87, Ni85, Stan84]. However, the problem of finding an optimal partitioning is found to be NP-hard [Gare79] in all but very restricted cases. Thus research has focused on the development of heuristic algorithms to find suboptimal partitioning solutions. An algorithm should be at least able to find a partition that would decrease the running time of the same simulation with a random partition.

There are many other problems that need to be solved for a parallel simulation to be able to exploit the inherent parallelism of a system fully. The problems include classical parallel processing problems, such as distributed detection and resolution of deadlocks, distributed termination, synchronization, partitioning and mapping processes to processors, flow control, and memory management.

1.2 Previous Work in Conservative Simulation

Major contributions to the work on the conservative approach may be found in Bryant [Brya77], Chandy, Misra and Holmes [Chan79, Chan81, Misr86], Peacock et al. [Peac79],Groselj and Tropper [Gros87, Gros88], Wagner et al. [Wagn89], Bain and Scott [Bain88] and Nicol [Nico88]. A good survey of conservative approaches to parallel simulation can be found in [Fuji90].

Chandy and Misra [Chan79] employ null messages to avoid deadlock and increase the parallelism of distributed simulation. When an LP sends an event message (e,t) to another LP via its output link, it sends a null message with the same timestamp on all its other output links. Since events at each LP are processed in increasing timestamp order, the receiving LP realizes that it can not receive an event on that link earlier than the timestamp of the null message. This allows the receiving LP to simulate al' events with timestamps less than or equal to the minimum timestamp of all the null messages on its input links. However, the arrival of a null message at an LP can cause it to generate another null message. This can result in excessive numbers of null messages being created and sent to all of the LPs. A number of papers have concentrated on optimizing this approach by reducing the number of null messages. For example, in [Su89] the authors send null messages only once the LP has become blocked. With this approach, referred to as eager events, lazy null messages, they were able to report some success in speeding up logic simulation.

Lookahead refers to an LP's ability to predict what will happen in the simulation's future based on the knowledge of the system being simulated and events that have been processed or are waiting to be processed. In particular, it helps the LP determine when it may receive a new event message from a predecessor. Fujimoto [Fuji88] demonstrated that lookahead is critical in determining a conservative algorithms performance. If processes have a good deal of lookahead, conservative algorithms are able to exploit the parallelism of the system.

Some researchers have tried to reduce the number of null messages necessary by using an LP's knowledge of the network. De Vries [DeVr90] proposed considering the overall network of LPs as composed of smaller sub-networks. An LP can then use its knowledge of the local network it belongs to to improve lookahead and reduce the number of null messages required. Cai and Turner [Cai90] use an additional type of null message, carrier null messages. These messages propagate through the system to carry additional information on lookahead and the route taken. The LPs use this dynamic information in order to reduce the number of null messages. Wood and Turner [Wood94] have shown that using the Cai-Turner method will not reduce the number of null messages for certain types of communication graphs, particularly those with nested cycles. They propose an alternative method to the carrier null messages approach in order to extend its applicability to arbitrary graphs.

In [Misr86], Misra suggested using a time out for null message transmission and having null messages annihilate earlier, obsolete, unprocessed null messages on the link. He also describes a class of demand driven techniques in which a process requests time information from the processor on the source side of its input link (the predecessor). In this algorithm, when an LP is blocked or simply idle it can query a predecessor via a request message to obtain a new lower bound on the timestamps of messages which it can consume. No analysis of the performance of their algorithm is given in [Misr86]. If the predecessor receiving the request message is unable to improve upon the senders lower bound, it can in turn send a request message to its predecessors. This approach can lead to deadlock [Cote92, Misr86]. Consequently, a deadlock detection and breaking algorithm must also be employed [Cote92].

The algorithm in [Misr86] assumes that path information is included within the request message. The demand based time synchronization algorithm described by Bain and Scott [Bain88] does not require that path information be propagated in messages between processes.

Instead they use three types of probe messages to detect deadlock and recover, *yes*, *no* and *ryes* (for reflected yes). The *yes* message indicates that the predecessor channel has reached the request time; *no* indicates that it has not and another request might be made; and the *ryes* message indicates the request time has been reached and a cycle was encountered. Once a cycle has been detected in the connection graph, a deadlock in time synchronization can be avoided. No performance analysis is given.

In [Chan81], Chandy and Misra allowed the simulation to run until it deadlocks. When a deadlock was detected they used a deadlock detection mechanism to recover. Deadlock detection mechanisms are described in the literature [Elma86, Chan82, Nata86, Cido87, Bouk93]. Reed, Malone and McCredie performed a distributed simulation of several queueing network models using the Chandy and Misra methods [Chan79, Chan81]. They determined that allowing deadlocks, then detecting them and recovering, is generally superior to methods that prevented deadlocks. They also concluded that Chandy-Misra approaches are not viable for the parallel simulation of queueing networks.

Wagner, Lazowska, and Bershad [Wagn89] used a shared-memory version of the Chandy-Misra algorithm called lazy blocking avoidance. The algorithm waits until a processor is idle before it attempts to provide a new lower bound for blocked LPs. In a shared-memory environment, messages do not need to be sent, events are simply put on and taken off the appropriate LP's queue. They reported speedups that were approximately twice as good as those reported by Reed et al. [Reed88]. However, it should be noted that their algorithms are limited to a shared memory environment.

Lubachevsky [Luba88] proposes an algorithm for distributed discrete event simulation which is based on using a bounded lag restriction. The processing of events concurrently is bounded above by a known finite constant B. An LP is permitted to process an event if its timestamp lies between the interval [μ, μ+B] where μ is the current smallest timestamp in the system and B is the bounded lag parameter. This method is described in greater detail later.

Other windowing methods include those described in [Nico88] and [Ayan92]. Nicol [Nico88] proposed a conservative method similar to [Luba88]. Each LP moves through the simulation advancing its time up to a global ceiling, however, calculating the global ceiling differs from using Lubachevsky's bounded lag. [Ayan92] introduced an approach called Conservative Time Windows (CTW). Again, an LP is permitted to process events within a time window. In contrast to other windowing methods, the size of the window may be different for different LPs. Their experiments indicate that enabling local bounds for the time windows improved the performance of the CTW-algorithm.

Groselj and Tropper [Gros88] clustered several processes into the same processor in order to achieve better efficiency. They proposed an approach referred to as the Time of Next Event algorithm (TNE). Their algorithm is based on the shortest path algorithm [John77, Chand82] and computes the greatest lower bound on all input links for processes assigned to the same processor.

For each empty input buffer, LPj maintains a time value TNEij, which denotes the Time-of-Next-Event. An LP is allowed to execute an event e with timestamp e.time if it can be sure that it will not receive any future events with timestamp less than e.time, i.e., for all j, where the input buffer (i,j) is empty, TNEij $\geq$ e.time, otherwise it must block. Consequently, events are always executed in chronological order and the timestamps of event messages sent by each LP are ordered in non decreasing order. Although it does help unblock blocked processes, it is possible for inter-processor deadlocks to occur. A distributed deadlock breaking algorithm is proposed in [Gros91] to deal with these inter-processor deadlocks.

In [Bouk92] the TNE algorithm was tested on an iPSC/i860 hypercube multiprocessor. Significant performance improvements were realized by using TNE to unblock processes that would otherwise need to block; good speed ups were realized over both a serial simulation and an optimized version of Chandy & Misra null message algorithm. TNE's ability to take advantage of having several LP's assigned to each processor and using the shortest path algorithm to provide lookahead rather than message passing contributed to the good performance. Other factors that play a role in TNE's and any conservative simulation's performance are the lookahead of the service distribution, the number of LPs in the model and the number of processors used [Bouk92].

Parallel simulations are typically aimed at exploiting the parallelism as a result of a spatial decomposition of the model. In [Chan89] the possibility of temporal decomposition is exploited. The problem can be viewed as a space-time graph where the y axis represents the time in the simulation and the x axis is the state of the process being simulated (the space). Each processor maintains the state variables of one or more processes over a period of time. Regions in the space-time graph are influenced by their neighboring regions through event messages (vertical lines) or by sharing common state variables in different periods of time. [Chan89] also describes a relaxation algorithm to overcome the dependency between regions. A process computes the state of its own region using estimates of its neighbours regions (inflows) then informs its neighbours (outflow). If a region correctly estimates its inflows, it will not have to recompute its own region and update its outflows. The space time-graph is correctly filled once there are no new events and no new recomputations.

1.3 Parallel Simulations of Billiard Balls

Many systems can be viewed as collections of objects moving around a defined space and interacting with each other. The objects can range from atoms and molecules to inter-planetary objects acting under the influence of gravity. Molecules move around their environment transitioning between unstable states and stable states as they interact with one another to form new compounds and molecules. The particles which comprise Saturn's rings exemplify the second category of system. Man made systems include cellular communication systems, in which mobile users move in between regions that are allocated different radio bandwidths. Such problems intuitively lend themselves to a parallel simulation algorithm by dividing the space in which the interactions take place and having each processor on a multi-computer simulate those objects within a region.

The difficulty in simulating such a model is that each processor must simulate an object in its own region, although the object's behaviour may be influenced by an object in another processor's neighbouring region. This system can be modelled as the classic network of logical processes communicating via message passing in which the messages contain information pertaining to the behaviour of objects in each processors' region. The two classical approaches to parallel simulation, conservative and optimistic deal with this problem in a manner consistent with their paradigms. The optimistic approach has the processors proceed with simulating the objects in their own region regardless of interaction with neighbours and correct for any errors by rolling back when they occur. The conservative approach blocks and waits until a processor is certain that an object in its own region is not affected by an object in a neighbouring processor at an earlier time.

The algorithm described in this thesis makes use of shared variables in order to allow processors to access the state of its neighbour's objects. The algorithm is conservative in nature in that it does not permit the processing of events which might be out of correct (timestamp) order. The use of shared variables captures more of a simulation's natural parallelism thereby reducing the amount of time a processor would be blocked in a pure conservative approach. If the dynamics of the objects being simulated are understood, it is then possible for the algorithm to make inferences as to whether it can safely proceed, whereas a "typical" conservative approach would have to block.

As a paradigm of the problem we are interested in simulating, we chose a billiard ball simulation. We simulate billiard balls travelling around a table, bouncing off the boundaries and colliding with other balls. The balls are represented in our model by velocity and position at a given time. We simplify the problem by allowing for a frictionless table and ignoring the rotation of the balls.

Both conservative and optimistic methods have been used to simulate billiard balls or similar systems. For example, in [Hont89] the authors used colliding pucks as a benchmark for evaluating the Time Warp algorithm. In [Luba92] an essentially conservative algorithm is used to simulate colliding rigid disks or billiard balls.

The remainder of this thesis is organized as follows. Section 2 contains a summary of previous work on the problem of simulating billiard balls (or rigid disks). In section 3 the parallelism in this problem and our parallel algorithm, making use of shared variables, is described. Section 4 contains the performance results realized from this algorithm. The conclusion follows.

2 Previous Work

In this section we summarize previous work on parallel billiard ball or rigid disk simulations. The first paper [Luba91],however, describes a serial algorithm. It is made use of in both the parallel algorithm described in [Luba92] as well as in the parallel algorithm described in this paper. In addition to describing the algorithm in [Luba92] we discuss the more general approach to distributed simulation in [Luba88]. The last paper [Beck88] describes the parallel simulations of colliding pucks using Time Warp.

In [Luba91] an algorithm for the serial simulation of billiard balls is described. Lubachevsky reports that the serial algorithm running on a VAX-8550 is about as fast as Time Warp [Hont89] on a 32-node hypercube MARK III [Luba92]. For each ball in the simulation the algorithm associates two events with it - an OLD event which represents the last event the ball had and a NEW event which represents the ball's next event or future event. A simulation event can either be a collision with another ball on the table or with one of the table's boundaries. An event consists of three components: time, state and partner. The time component is the time at which the event occurs and therefore the time at which the ball is in the state represented by the state component. The state is a pair of vectors that represent the position of the ball on the table and its current velocity. If the event is a collision with another ball, the partner is represented by a pointer to that other ball. Otherwise it is a collision with a boundary and the partner is NULL.

Processing an event means scheduling the next event for the ball(s) (a collision event will usually involve two balls but on rare occasions it can involve more than two balls; a collision with the boundary will only involve the one ball.) The event just processed becomes the OLD event of the balls involved and the next event scheduled becomes their NEW event. The current simulation time is the earliest NEW event time among the balls. Therefore, each ball's OLD event time is less than the current simulation time and its NEW event time is greater than the simulation time except, of course, for the ball whose event is currently being processed.

The events are kept in a heap data structure with the earliest NEW event on top. The algorithm selects the ball with the lowest NEW event time and determines its new future event. To determine a ball's next event, the algorithm first determines which ball or boundary it will interact with. A function *interaction_time(state1, time1, state2, time2)* returns the time that a ball in *state1* at *time1* will have its next interaction with an object in *state2* at *time2* if no other objects interfere. An object can be another ball or a boundary. If the object represented by state at *time2* is another ball with the same diameter D and if the velocity of both balls remain constant, then *time=interaction_time(state1, time1, state2, time2)* becomes:

$$\text{time} \leftarrow \max(\text{time1}, \text{time2}) + t$$

where,

$$t = \begin{cases} (-b-(b^2 - ac)^{1/2})/a & \text{if } b \le 0 \text{ and } b^2 - ac \ge 0 \\ +\infty & \text{if } b > 0 \text{ or } b^2 - ac < 0 \end{cases}$$

and

$$a = |\text{velocity2} - \text{velocity1}|^2$$

$$b = (\text{position20} - \text{position10}) \cdot (\text{velocity2} - \text{velocity1}),$$

$$c = |\text{position20} - \text{position10}|^2 - D^2,$$

$$\text{position10} = \text{position1} + \text{velocity1}(\max(\text{time1}, \text{time2}) - \text{time1}),$$

$$\text{position20} = \text{position2} + \text{velocity2}(\max(\text{time1}, \text{time2}) - \text{time2}).$$

where u • v denotes the dot product of vectors u and v, and |v| denotes the length of vector v: $|v| = (v \cdot v)^{\frac{1}{2}}$. The algorithm determines the interaction_time for the current ball and each of the N balls and K boundaries. State1 is the OLD event of the current ball (note, the OLD event has just been replaced by its NEW event and the next event is being determined). State2 is the OLD event of the object. State2 corresponds to the OLD event, because until the simulation reaches the time of its future event, it represents the state of that object. If the interaction_time is greater than the object's NEW event, it will not interact with the current ball since the state that the object was in to cause the collision changes before the collision. The ball's next interaction will be with the object with the earliest interaction time.

The algorithm then determines the new state of both balls, if there is a collision between two balls, or the new state of the ball if there is a collision with the boundary. The function *advance(state0, time0, time1)* updates the position of the ball at *state0* and *time0* to the position it will occupy at *time1* if the velocity component of *state0* remains the same during the interval (*time0, time1*). The function *jump(state1,state2)* updates the velocities of each of the objects whose current states are state1 and state2. The result of a collision between balls 1 and 2 with vector velocities v1old and v2old (assuming that energy and momentum are conserved) is computed as follows: the normal components of the initial velocities are switched but the tangential component remains unchanged. If the other object is a boundary then *state2* would be NULL.

If the current ball's next event is a collision with another ball (the object ball), it is possible that the object ball's previous NEW event was a collision with a different, third ball. Since the collision between the current ball and the object ball occurs before the collision with the third ball, the third ball's NEW event needs to be "undone". This requires that the third ball's NEW state be restored to its OLD event state. Figure 2.1 contains the pseudocode for this algorithm.

```
while current_time < END_TIME
        current_time = min of all balls NEW time
                let i be that ball
        [i.OLD] ← [i.NEW]
        P ← minimum time for interaction with any
                other ball. j ← other ball
        Q ← minimum time for interaction with any
                obstacle. k ← is that obstacle
        R ← min{P,Q}
        if R < infinity
                state1 ← advance(state[i.OLD].time[i.OLD].R)
                if Q<P /*then obstacle event*/
                  state[i.NEW] ← jump(state1.k)
                  partner[i.NEW] ← NULL
                else
                  time[j.NEW] ← R
                  state2 ← advance(state[j.OLD].time[j.OLD].R)
                  state[i.NEW].state[j.NEW] ← jump(state1.state2)
                  m ← partner[j.NEW]
                  partner[i.NEW] ← j; partner[j.NEW] ← i
                  if m ≠ NULL and m ≠ i
                        state[m.NEW] ←
                          advance(state[m.OLD].time[m.OLD].time[m.NEW])
                   partner[m.NEW] ← NULL
```
Figure 2.1 Pseudo-code for a serial simulation of billiard balls algorithm

Parallelizing the billiard ball simulation requires that the table be divided up into separate regions which are distributed among the processors and that each processor simulates the balls in its assigned region. Each processor "shares" a border with two or more other processors and has to pass balls back and forth across these borders. This interdependency between the processors, which is inherent in any parallel simulation, requires an algorithm to maintain the causality when the processors' simulation times are different. The following papers deal with this problem.

In [Luba88] Lubachevsky proposes a general algorithm for distributed discrete event simulation which is based on using a bounded lag restriction; the processing of balls concurrently is bounded above by a known finite constant. The algorithm proceeds in iterations. In order to determine the floor for the iteration each processor broadcasts the earliest event in its event pool. The floor is then the earliest event in the simulation; the processors then synchronize. Each processor goes through every event with a time less than $\mu + B$ and determines which events will be enabled for that iteration and marks those that are enabled. An enabled event is one that can be processed in the next iteration. As a result of processing these events, new events may be scheduled or deleted, including events at other processors. Once each processor has completed processing all of its marked events in the current iteration it resynchronizes and determines the floor for a new iteration.

```
while $\mu$ < infinity and not (termination_condition) do
{
  $\mu$ = min t(Mj), $1 \le j \le N$
  broadcast $\mu$ to all N processes
  synchronize

  for all j = 1,...N do
    if $t(Mj) \le \mu + B$
        then detect and mark enabled events in Mj
    synchronize

  for all j = 1,...N do
    if M, contains marked events then
        {
         tj = t(Mj)
         process marked events
         if required, then schedule new events for Ij or other Ik, k does not equal j,
         and/pr delete some events from Ij or other Ik, k does not equal j
         delete the processed events from Ij
        }
  synchronize
}
```

Figure 2.2 Pseudo-code for a parallel simulation of billiard balls algorithm

Determining if an event should be enabled in each iteration involves determining if an earlier event from another processor will affect it. This is determined by using a precomputed quantity, d(k,j), where k and j are processors, which represents the propagation delay. The propagation delay is the amount of time that must past before an event in k can affect an event in j.

The propagation delay is added to the time of the earliest event which can be sent by k: t = *earliest_event_time(k)* + *d(k,j)* if the current event being considered is in j. If event e(j) > t then it cannot be marked as enabled.

The opaque period is the simulation time when one processor is servicing an event and will therefore not be able to create another event on its own processor or another until it is done. If an opaque period exists, then it plays a part in determining if an event can be enabled.

The paper also describes a specialized computer that would perform the operations necessary for this algorithm efficiently. Those operations include synchronization barriers, computing the minimum floor for each iteration, testing if events are enabled and updating events in each processor (i.e. when the processing of an event in one region results in new events being scheduled in another region). Experiments in [Luba89] show that the bounded lag approach to discrete event simulation can realize a speed up of 23.3 over a serial program when simulating a queuing network with 25 processors. A theoretical argument for the simulations' scalability is also given in [Luba89].

The problem with using the bounded lag approach in [Luba88] for the distributed simulation of billiard balls is that there are no propagation delays or opaque periods between adjacent processors that can be precomputed. The time at which a ball may enter a neighbour's region depends on the ball's current position and velocity. This approach would result in very quickly serializing the simulation. The bounded lag method also requires the processors to synchronize in order to maintain the integrity of the simulation. One processor can not be trying to determine which events to enable for a given iteration when another processor is deleting events.

Although the synchronization approach in this algorithm might be appropriate for a SIMD (or a specialized) machine, it would result in costly overhead on a MIMD machine.

[Luba92] details how the bounded lag algorithm is used to simulate the billiard ball problem. Basically, this paper parallelizes the algorithm described in [Luba90] using the bounded lag approach. One result in [Luba92] is that the Time Warp approach to state recovery can be replaced by a simpler method. Consistent with the general bounded lag algorithm, the fundamental observation of this method is that if two disks are separated by a large enough distance, the probability that a causal dependence between their motions will be created in the near future is small. The algorithm rests upon the assumption that there is an upper bound on event propagation speed, the consequence of which is that two balls which are sufficiently far apart can be processed concurrently. The author contends that by using the bounded lag to control the errors that can occur, the simulation does not require the more flexible rollback mechanism employed by Time Warp. Instead, it can use a simple state recovery method.

Within the large sectors controlled by each processor, the space is divided into smaller sectors to reduce the number of balls that need to be searched. Dividing the simulation space into smaller sectors reduces the number of balls that need to be examined in order to determine a ball's next collision, because it is only necessary to examine the balls within the small sector. This, however, is at the expense of having more events since each time a ball moves into another small sector another event must be processed. The optimal trade-off size of a small sector depends on the parameters of the problem to be simulated [Luba92]. [Hont89] also found sectoring to play a factor in performance of rigid disk simulations.

Each processor maintains a list of events. The smallest time in the list of events is referred to as the a_time. The algorithm proceeds in iterations. The floor for each iteration is the min a_time(PEi) for all processors PEi. According to the bounded lag restriction with parameter B, an event e is then processed only if time(e) $\leq$ floor + B, for some value of B.

As mentioned above, each event is associated with a small sector s within which it occurs. An error occurs when a collision is missed or a positive area is occupied by more than one disk at the same simulated time. Events which occur in small sectors close to the borders and which are admissible to process under the above condition may result in new events being scheduled in the large sector which is simulated by the processor or in its neighbouring sector (and processor) or in both sectors. This means that errors can occur if two events in adjacent small sectors on opposite sides of a processor border are processed out of sequence (i.e. with respect to their simulation time). To detect these errors each sector maintains a time t(s) when the last event was processed in the vicinity of s, defined to be the 8 neighbouring small sectors. In Figure 2.3, the vicinity of the small sector at coordinate B2 is B1,C1,C2,C3,B3,A3,A2,A1. t(s) is updated when a new event at time t is processed in sector s. All other sectors in the vicinity of s take the new t(s). The algorithm considers an error to have occurred if *while updating t(s), the new value t is smaller than t(s).*

To reduce the number of errors and to enable their detection, a set of small sectors close to the boundary (called an insulation layer) is defined. These small sectors have more restrictive processing rules. For a disk d, W(d) is the set of all processors adjacent to the location of the event e (in Figure 2.3, W(d) = {P1,P3,P4}). Processing of e is not permitted if for any PE in W(d) a_time(PEi) < time(e). In this case, processing of event e must wait until both the bounded lag restriction time(e) < floor + B is satisfied and for all PE in W(d) a_time(PEi) > time(e).

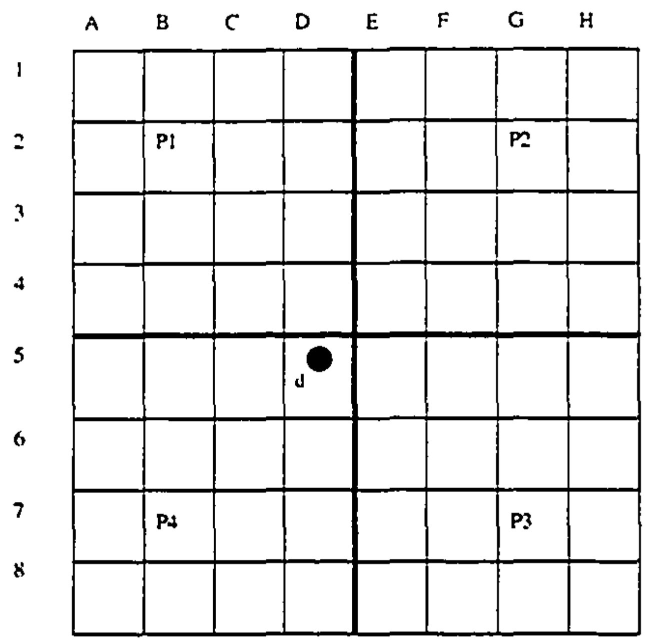

Figure 2.3 Table divided into small sectors

With these processing restrictions, errors will not occur unless disks gain sufficient speed to travel the distance w across the insulation layer in time less than B. Balls outside the insulation layer are scheduled independently of the neighbour's time. So, if a high speed ball was travelling fast enough that it could cross through the insulation layer within the bounded lag B, it could collide with a neighbour's ball. This error will only occur if the ball is travelling at a velocity v where $w/v < B$.

Errors can also occur as a result of a propagation of collisions through a chain of balls that stretch across the insulation layer. A chain of balls occurs when a number of balls close together form a row perpendicular to the boudary between two regions. If the first ball in the row is set in motion, it quickly collides with the next ball in the row which in turn collides with the next. One end of the chain could collide with a ball outside the insulation layer, that was scheduled independently of its neighbours a_time, while at the other end of the chain a ball could collide with a ball from a neighbouring processor. The frequency with which this occurs is small. It grows as the balls are more closely packed together.

By restricting the processing of events with the bounded lag method, error detection has been "filtered", making it possible to have a crude and simple method of state saving and recovery, without the overhead necessary to implement Time Warp's method of rolling back. Just before an iteration begins "nothing is moving", an ideal time to save the state of the simulation. However, saving states at each iteration is inefficient since most of the saved states would be unnecessary as rollbacks "should" occur infrequently. States are saved when the floor exceeds an amount of time equal to pB from the previous saved state. B is the bounded lag and p is an adjustable parameter, $p \geq 1$. When an error is detected the algorithm redoes the entire simulation from its most recent checkpoint.

The parallel program was emulated on a serial machine; the paper does not provide a performance analysis which reports actual execution times on a real machine. The algorithm successfully parallelizes the serial algorithm in [Luba90] without being overburdened by roll backs. In a low density simulation there was approximately one roll back per $5X10^5$ collisions and one per $2X10^6$ collisions for a high density simulation. However, the problem with the execution time of this method could well be the amount of time a processor spends blocked.

In [Beck88] the authors describe how a parallel simulation of colliding pucks was run on the Time Warp Operating System. The problem consists of two types of objects: pucks and sectors. Each object is a process. As in [Luba92] the table is decomposed into sectors. Each sector is an object and is responsible for scheduling collisions within its boundaries. The sectors examine the trajectories of all the pucks within its boundaries to schedule only the earliest predicted collision. The sector reawakens at the time of the collision and causes the pucks involved to change their velocity via messages. The pucks change their velocity and calculate their new trajectory. Pucks send Enter/Depart messages to the sectors as they move from one sector to the other. Because of the dependency between the sectors, errors can occur as balls move from one sector to another; Time Warp is used to correct these errors. This implementation of the colliding pucks simulation was tested in [Hont89]. With 32 processors they reported speed ups of 7.9 over a serial version.

3 Parallelism in a Billiard Ball Simulation

When a billiard ball simulation is divided among multiple processors according to regions, parallelism exists because events in one region can be processed independently of those in other regions. However, as mentioned above, balls close to the boundary between the two regions do not share this luxury. It is as likely that a future event will be a collision with a ball from a neighbouring processor, as it will be a collision with a ball from the same processor. Causality may be violated when two events on opposite sides of a border between two processors are processed out of sequence. Let us define processor Pa's current_time(Pa) as the time of the current event which it is processing. Suppose that processor Pb, which is on the opposite side of the border with Pa, has current_time(Pb) < current_time(Pa), as in Figure 3.1. If we use the serial algorithm described in section 2, the ball currently being processed by Pa is A1 and the time of its NEW event time(A1) is the current_time(Pa). A1's NEW event at time(A1') is to cross the border between Pa and Pb. The ball currently being processed by Pb is B1. At time(B1) it has collided with ball B2 and its new direction is toward the border between Pb and Pa. In fact it will cross the border at time(B1') <

current_time(Pa) and continue on to collide with ball A1 at time(B1") < current_time(Pa). An error will occur if Pa processes A1 and continues to simulate other events.

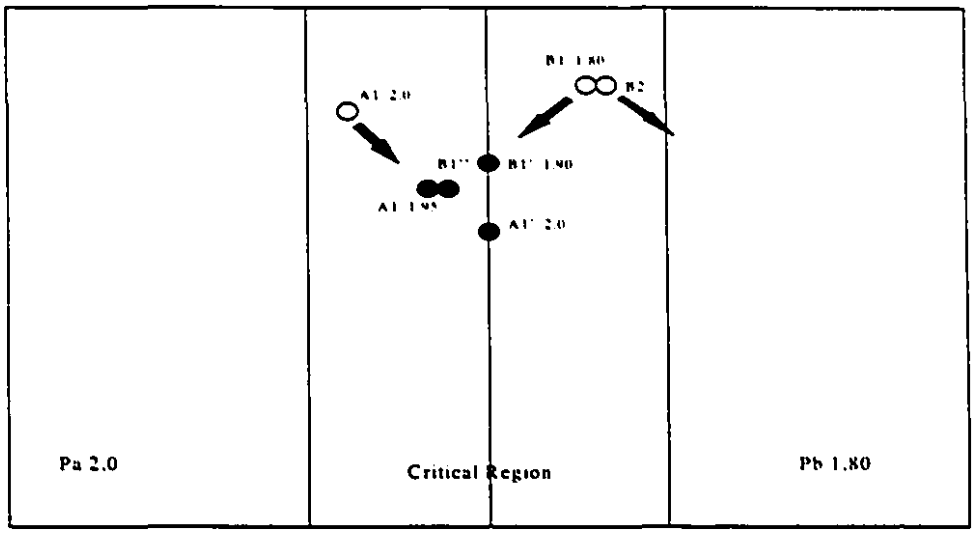

Figure 3.1 Processing error

Balls close to the border can be processed optimistically as in Time Warp [Beck88,Hont89] in which errors are corrected by rolling back. They can also be simulated conservatively so that a processor is not permitted to proceed until it is certain that no ball from a neighbouring processor will cross over the border and cause an error.

[Luba92] is a hybrid approach. A certain amount of freedom is allowed outside of the insulation layer allowing a crude state recovery method to correct any errors. However, inside the insulation layer strict conservatism is enforced.

The shared variable algorithm (described in detail in the subsequent section) shares some features with the algorithm described in [Luba92]. As in [Luba92], it restricts which events may be processed in a manner similar to the bounded lag approach. The simulation does not run in iterations. However, a processor cannot process an event e at time(e) when there is a processor Pi that shares a border with it and has a current_time(Pi) < time(e) - B. B is the distance from the edge of the critical region to the border divided by an estimated maximum velocity. The value of B is adjusted to effect a compromise between giving the processors the freedom to process balls in their region independently of their neighbours, and preventing errors from occurring in the simulation. Between each processor, a portion of the simulation region is designated as a critical region similar to the insulation layer. The critical region is simply a region close to the borders between two processors which has stricter rules for simulating the balls found inside of it. The algorithm treats a ball crossing the boundary between processors and moving into or out of the critical region as an event. The method used to process balls serially within each processor is the algorithm found in [Luba90], described earlier.

It is the way in which balls are simulated inside the critical regions that distinguishes the shared variable approach and which realizes substantial performance improvements. Each processor maintains, along with its event heap, a subset of that processor's events in an ordered list consisting of the events that take place within the critical region. These lists are shared by all neighbouring processors. Processors use these lists to determine when a ball will cross the border and to extrapolate from this information whether they may safely proceed without causing an error. The algorithm avoids the severe blocking restrictions for processing events close to the border which can result in serializing the processors as they wait to schedule these balls. The shared variables method loosens the restrictions on processing balls close to the border and consequently reduces the amount of time processors spend waiting to process balls close to the borders.

3.1 The Algorithm

The algorithm has a different processing regime depending upon whether an event occurs inside or outside of the critical region.

Balls outside the critical region are "protected" by the critical region from missing collisions which could occur with balls in a neighbouring processor's regions. By restricting the processing of balls to when time(e) < current_time(Pneigh) + B , balls outside of the critical region are assured that a ball from another processor cannot cause a violation of causality. For example, assume that as depicted in Figure 3.2, processor Pa is processing ball A1 at time(A1) = 2.0. If the estimated maximum ball velocity is 4.0 units/sec, it would take 1 sec (4.0 units / 4.0 units/sec = 1 sec) to cross the critical region and possibly collide with another ball from the neighbouring region, thereby causing an error. Therefore processor Pa is able to proceed without risk of error if the current_time(Pb) > 1.0 sec (time(A1) - B = 2.0 sec - 1 sec = 1.0 sec) . In the event that balls travel at higher than the estimated maximum speed or there are chains of balls stretching across the critical region, state recovery and resimulation are required [Luba92].

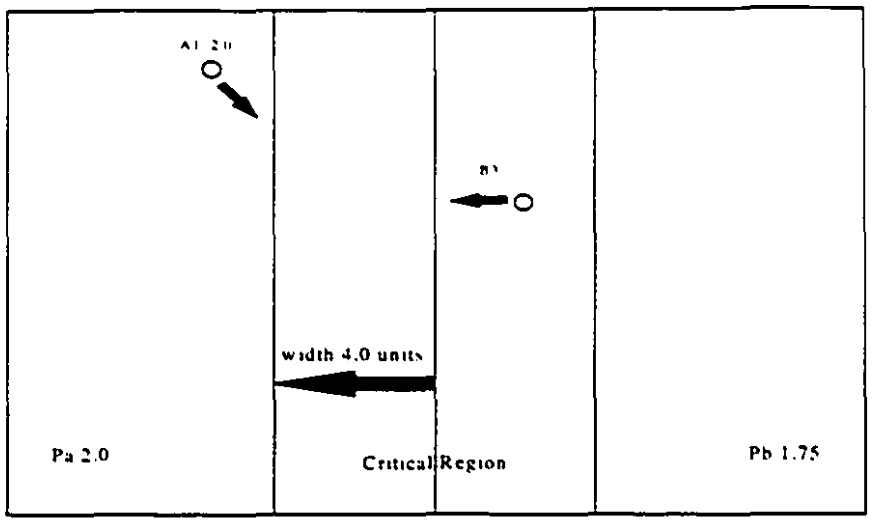

Figure 3.2 The critical region

If the ball Pa is currently attempting to process is in the critical region, then under a conservative algorithm Pa would block until current_time(Pb) > current_time(Pa). The algorithm using shared variables tries to determine if a ball crosses the border before current_time(Pa). If Pa determines that no ball crosses the border before current_time(Pa), then it can avoid useless waiting and can proceed with the simulation. Otherwise, as in the conservative case, it must wait.

In Figure 3.3, processor Pa's current_time(Pa) is 2.0 when it tries to process A1 at time(A1) = 2.0. At the same real time processor Pb's current_time(Pb) is 1.75 when it tries to process a ball outside the critical region. Under "strict conservatism" processor Pa would have to wait since current_time(Pa) > current_time(Pb). Using shared variables, processor Pa can look "inside" of Pb and determine the state of balls near the border. Since the earliest time that a ball could cross the border from Pb is time(B2) > current_time(Pa), Pa can safely proceed.

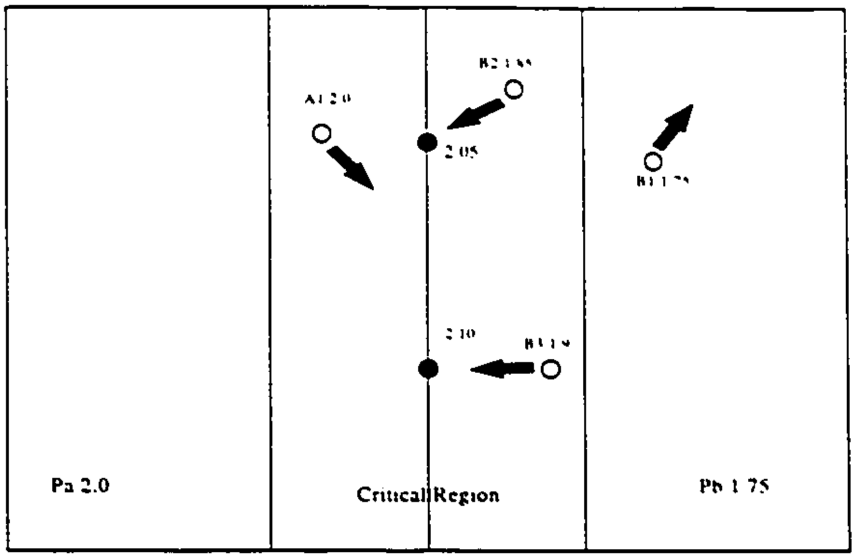

Figure 3.3 Shared variables in critical region

The algorithm first determines if the current ball A1 being processed at time current_time(Pa) is in the critical region. If A1 is not in the critical region, then the processor must wait if current_time(Pa) > current_time[Pi] + B for any processor Pi which shares a border with Pa. As mentioned above, if Pa is greater than B ahead of its neighbour in simulation time it is possible for a ball to cross Pa's critical region and affect its current ball. If current_time(Pa) < current_time[Pi] + B it may proceed with processing its current ball.

If the current ball A1 is inside the critical region, then a ball directly across the border could affect it in time less than B. It is here that a strictly conservative approach might have to block, whereas the shared variable algorithm tries to determine if it may proceed. There are two conditions that must be satisfied in order for the shared variables to be utilized. First, the current_time(Pa) > current_time(Pi) for some processor Pi that is a neighbour of ball A; this indicates that there is a ball in Pi that, depending on its current state, could cross the border and affect A1 before current_time(Pa) and that the algorithm needs to determine if, in fact, it will. If, on the other hand, current_time(Pa) < current_time(Pi), there is no ball in Pi which could cross the border and affect ball A and processor Pa can proceed. The second condition is the same as if the ball is outside the critical region; current_time(Pa) < current_time(Pi) + B. This is necessarily true if current_time(Pa) < current_time(Pi). If current_time(Pa) > current_time(Pi) then the processor needs to know if any of the balls in the critical list of Pi can affect the event being processed for A1. This is not helpful if current_time(Pa) > current_time(Pi) + B since any ball in Pi could affect Pa, hence looking at the balls in the critical region is not enough, and as mentioned above, the processor must wait.

The algorithm also restricts the use of shared variables to times when it is greater than avg_time_between_events ahead of the slower neighbouring processor. This is to ensure that the processor does not uselessly calculate the lower bound when the neighbouring processor may process that event next. If, as in Figure 3.3, the next event which Pb is going to process is the ball B3, then there is little to gain by having processor Pa access a shared variable if the owner will simulate it next anyhow. The payoff for using shared variables occurs when Pb has many other events to simulate outside the critical region earlier than the ball which Pa is waiting on.

To determine that a ball will not cross before time(A1), the algorithm must determine a lower bound on the time when a given ball in the critical region of processor Pi will cross. It does this using the shared variables accessed in the function find_lower_bound described in Figure 3.6. First the algorithm sets the invariant flag_ball_can_affect_me equal to FALSE. This indicates whether or not the algorithm has found a ball in Pi's critical region which could cross the border and affect A1. The algorithm goes through the ordered list of balls in Pi's critical region starting with the ball with the earliest NEW event time, determining their lower_bounds until it comes to a ball with NEW event time(e) > current_time(Pa). Recall that we want to know what happens after its NEW event, because the ball's NEW event represents its future state, the OLD event has already taken place and has been correctly processed. If the ball I1 is in Pi's critical region and its future event time(I1) > current_time(Pa), then it cannot affect A1's current event. However, this is only true if I1's next event remains the same. It is possible that processor Pi will subsequently realize a different NEW event for I1 where time(I1) < current_time(Pa) after Pa has determined that I1 would not affect it. This potential problem is addressed in the following section. If the lower bound for any ball in Pi's critical region is time(e) < current_time(Pa), then the algorithm has found a ball that could affect Pa. Flag_ball_can_affect_me is set to TRUE, and the algorithm exits from the loop since it is no longer concerned with any other balls in the critical region because the processor is already required to wait.

Outside the loop the algorithm checks the value of flag_ball_can_affect_me. If it is FALSE, then the algorithm did not find any ball that would cross the boundary(Pa/Pi) in time(e) < current_time(Pa) and processor Pa can proceed. If the value of flag_ball_can_affect_me is TRUE then the processor must wait. It waits until the earliest NEW event in the critical region cr_time(Pi) > current_time(Pa).

Depending on the position of the ball, it may be necessary for the algorithm to wait for more than one processor. For example, if the ball is in a corner of its region, its future event could be affected by any one of the three regions surrounding it. As in Figure 2.3, the ball d is in the upper right corner of P4. Balls close to the border of P1, P2 or P3 could affect d. Under these conditions the algorithm would have to determine a lower_bound(Pa/Pi) for each of the neighbouring processors i . Figure 3.4 below contains the pseudo-code for the algorithm.

```
where, B and s are constants,
        cr_time is the time of the first event in the critical list
        cr_ball points to the current ball being pseudo simulated
              in the critical list
        Pa is the current processor executing
        Pi is a processor that shares a border with Pa

    if (ball is NOT in critical region)
      for (each processor i which shares a border with Pa)
            if (current_time[Pa] > current_time[Pi] + B)      // if Pa is within B of Pi it can proceed //
              wait() until current_time[Pa] <                 // otherwise it must wait //
                          current_time[Pi] + B
            endif
      endfor
    endif

    else (ball is in cr)
      for (each neighbouring processor i of ball A)
       if (current_time[Pa] > current_time[Pi]) && (current_time[Pa] < current_time[Pi] + B ) // shared
                                                                                  variables can be used //
          if (cr_time[Pi] > current_time[Pi] + avg_time_between_events)
              flag_ball_can_affect_me = FALSE;        // set invariant to FALSE //
              for (cr_ball = first ball in cr_list;   // start at first ball in Pi's critical list //
                cr_ball[NEW] < current_time[Pa];      // loop until find a ball that can't affect Pa //
                cr_ball = next in list)               // look at next ball in list //
                  low_bound = find_lower_bound(cr_ball); // determine lower_bound of when it will cross //
                  if (low_bound < current_time[Pa])   // if low_bound is less than the current_ball time in
                                                        Pa it could be affected by a ball in the cr_list of Pi //
                    flag_ball_can_affect_me = TRUE;   // set invariant to TRUE //
                      break;
                  endif
              endfor
          endif

          if (flag_ball_can_affect_me == TRUE)    // if the invariant is set to TRUE the algorithm has found
              wait until current_time < cr_time[Pi]  a ball that could affect it and must wait //
          endif

     else
       wait() until current_time[Pa] < cr_time[Pi]     // if it is too far ahead that any ball in Pi could affect it
       and current_time[Pa] < current_time[Pi] + B       it must wait //
     endelse

      endfor (each neighbour of ball )
    endelse (in cr)
```

Figure 3.4 Pseudo-code for shared variable algorithm

The position of the NEW event for a ball in the critical region will be within the region simulated by the owning processor. A neighbouring processor can determine how long it will take the ball to reach their shared border by using its NEW event time and NEW event position and velocity. The ball will cross in the time equal to the NEW event time, plus the time to travel the distance to the border. It is not, however, enough to simply use a ball's current velocity and position since a future collision can occur which would speed up its approach to the border. It is necessary to determine the ball's next collision. The find_low_function first determines the balls next event and then uses its new velocity and position to calculate the time required to cross the border between the processors. Figure 3.5 below contains the pseudo-code for the find_lower_bound algorithm.

```
where, cr_ball is the current ball in the critical list

find_low_bound(cr_ball)
  time_added = cr_ball's distance from border / cr_ball->velocity        // calculate time until it reaches the border //
  if (time_added >= 0)                                // if cr_ball is on the border or heading towards it //
     lower_bound_without_collision = cr_ball->time[NEW] + time_added  // lower_bound_without_collision
                                                                      // is the balls current_time + time_added //
     if (time_added != 0)                             // if not on the border have to consider a collision //
        Bj <- ball with minimum interaction_time(cr_ball, each ball in cr_list)  // determine which ball it will collide
                                                                                    with //
        calculate new states after collision between cr_ball and Bj
        calculate new lower_bound for cr_ball with position and velocity after collision
        calculate new lower_bound for Bj with position and velocity after collision
        lower_bound_after_collision equals minimum(lower_bound of Bj, lower_bound of cr_ball)
     endif

     // the lower_bound is the  minimum of the lower_bound_without_collision and lower_bound_after_collision //
     lower_bound = min(lower_bound_without_collision, lower_bound_after_collision)
     return(lower_bound)

  else                               // else it is heading away from the border //
     lower_bound = END_TIME
     return(lower_bound)
  endelse
```

Figure 3.5 Pseudo-code for find_lower_bound

The function first needs to determine the time_added for the ball we are currently looking at in the critical region (referred to as the cr_ball). The variable time_added is the amount of time after the NEW event time until the ball crosses the boundary. If this time is equal to zero then the ball's NEW event is crossing the border. If it is greater than zero then the ball will be moving toward the border after its NEW event. Finally, if it is less than zero it will be moving away from the border. If the ball is moving away from the border the function find_lower_bound returns END_TIME.

The lower bound for the cr_ball (if it does not collide with any other ball) is time_added plus the ball's NEW event time. If the ball is heading toward the border, then the possibility must be considered that a collision could occur which would decrease its arrival time at the border. Balls heading away from the border may have a collision which could result in changing their direction toward the border. Since it would require another ball heading in the opposite direction (i.e. toward the border) the algorithm does not determine the time of a collision until the ball heading toward the border, if one exists, is examined.

The find_lower_bound function treats the cr_ball as though it were the current event on the top of the processor's heap. Since balls are not actually being simulated, they are not taken off the list. If there are balls earlier in the list then the cr_ball, they are assumed to have already been processed - the possibility that the cr_ball's next event is a collision with such a ball has already been considered. Find_lower_bound uses the ball's NEW event to determine when its next collision with a boundary or with another ball in the critical region will take place. To calculate when a ball will collide with a boundary or a another ball the function uses the interaction_time function as does the serial algorithm executed within each processor.

The find_lower_bound function determines the interaction_time between the current cr_ball and every other ball further down in the critical list. Any possible collision with balls earlier in the list would have been considered when the earlier balls' lower_bound was determined. The ball with the earliest interaction_time is assumed to be the next ball to collide with the current cr_ball. For example, the function interaction_time would be passed the current cr_ball's NEW event time and state, as though it were the event being processed within its own region. Time2 and state2 would be the time and state of the OLD events of all the other balls further down in the (ordered) critical list. The cr_ball's next event is assumed to be a collision with the ball with the lowest interaction time. The state of the balls after the collision is determined using the advance and jump functions, and is then used to determine a new lower_bound.

The lower_bound_after_collision is the earliest time either one of the balls involved will cross the border with their new states after the collision. The find_lower_bound function returns the earliest of the low_bound_without_collision and low_bound_after_collision. As is discussed below it is possible that the collision in the critical region may not occur if an earlier collision with another ball is found. It is therefor necessary to return the earliest of the low_bounds in case another collision is found and one of the balls involved in the original collision is able to reach the border quicker than had the collision occurred. This is analogous to the owning processor realizing that a ball will collide with another ball that is already scheduled to collide with a third ball, so the third ball's state is restored to its state before the collision.

There are two apparent problems with this algorithm. First, as mentioned above, the NEW event for the current cr_ball may not be the correct event. The NEW event may not be correct if later, in real time, the owning processor realizes that the cr_ball will collide with another ball earlier then the NEW event time. This could cause an error if a neighbouring processor did not "see" the other collision and determined that the cr_ball would not affect it based on the previous NEW event. For the NEW event to change, the cr_ball would have to collide with a ball from either outside the critical region or from within the critical region. If it were with a ball from within the critical region, the processor calling the find_lower_bound function will have seen this collision and determined if the balls involved would affect it. Collisions with balls further down in the critical list than the cr_ball will have been examined when determining the cr_ball's next event. Collisions with balls earlier in the critical list then the cr_ball will have been examined when those earlier balls where the cr_ball. A collision between the cr_ball and a ball from outside the critical region however, will not yet have been encountered. Recall that we are not so concerned with the exact time of the cr_ball's next event but rather with whether it will interfere with the current event of the processor ahead in simulation time. Just as in the case of processing balls normally outside the critical region, this would require at least an extra B units of time for the ball outside the region to collide with the ball inside the critical region which could then move across the border. Since our processor is already required to be within B time units of its neighbours this would not cause an error. This addresses, as previously mentioned, why it is sufficient to calculate the lower_bound of only those balls that have NEW event time(e) < current_time(Pwaiting). A ball with a NEW event time(e) > current_time(Pwaiting) will not affect Pwaiting since time(e) happens after Pwaiting's current event. Only if the balls NEW event time changes from time(e) to time(e'), where time(e') < current_time(Pwaiting), could it potentially affect the current ball being processed by Pwaiting. If the collision is with a ball from outside the critical region then it would take too long for it to cross the border.

The second problem is that we are only simulating the first layer of events. We do not consider the case in which a ball collides with a ball and then collides with another ball which speeds its way to the border. Such considerations would necessitate a method of state recovery to correct any errors which result from not taking into account these secondary collisions. In our experiments we limited the occurrence of these errors by using an appropriately large critical region and limiting the density of the balls in the simulation.

Future projects include trying to realize further performance improvements by storing and sharing balls' NEW events that were calculated using the find_lower_bound function. If one processors has fewer balls to simulate in its region for a given time period, it will frequently be further ahead in simulation time than its neighbours. This will cause it to use the shared variables to infer if its neighbours' balls will affect it. Once the neighbouring processor catches up in simulation time it could use the NEW events calculated by the find_lower_bound function instead of recalculating them itself. The owning processor would have to be able to determine if anything had occurred in the critical region since the neighbour processor had made the calculations that could cause them to be incorrect.

Another project is to implement the algorithm on a distributed memory machine. Shared variables could be implemented using distributed memory [Mehl93].

4 Experiments

The experiments were run on an BBN butterfly. The butterfly is a shared memory multiprocessor in which each processor accesses shared memory using a shared bus. Each node of the butterfly is a 68200 Motorola processor. The simulation was programmed in C using the MACH1000 operating system, a UNIX-like operating system with an expanded library for synchronous functions, including semaphores and shared memory. Each processor runs at 16 MHz.

The simulations were done with a 200X200 square unit table and with 1 unit diameter balls. As we will shall see, the most important factor in affecting the speed of the simulation is the density of the balls on the table. The experiments were run with between 1200 and 7200 balls on the table. Their initial position and velocities vectors were randomly generated using the C library rand() function. The balls' positions were allowed to occupy any position on the board provided that a ball was not touching or overlapping another ball. The x and y coordinates of the velocity vectors were between the range of -4 and 4 units per second. The number of processors used were 4, 9, 16 and 25.

We compare our algorithm to one which does not use shared variables. When such a processor needs to process a ball in the critical region it must wait until the neighbouring processor's simulation time is greater than the time of the event it wishes to process. There is no information available for the processor to use in order to determine if it may proceed. The algorithm is identical to the one with shared variables in every other respect.

A larger population of balls clearly results in more events occurring during the same period of time in the simulation because there are more balls to be simulated, therefore they collide with one another more often. When balls are closer together they are more likely to collide with a nearby ball as they move about the table then they are to travel long distances between collisions. When the balls travel long distances they are processed less often, enabling the processors to do other work. If the density becomes high, the balls begin to move small distances, collide with a nearby ball, change direction and continue doing this until they come to a stage in which they are effectively "vibrating". In [Luba92], balls expand. As this happens, they become more closely packed and eventually start vibrating. This situation lends itself well to parallel simulation since the dependency between the processors is based on balls moving across boundaries, something which will not occur frequently when the balls are quasi-stationary. Furthermore, with very high density simulations the processors will advance at approximately the same rate because events will always be available to simulate at each processor.

We conducted experiments between a low population simulation of 2400 balls to a high population of 7200 balls. The more densely populated the critical region becomes, the more likely a ball is to be involved in more than one collision on its way toward a border between two processors. As mentioned before, since the algorithm bases its lower bound only upon the next event , it is possible to miss a second collision that may cause an error as an unexpected ball crosses the border. It is also only under these conditions that the possibility of a "high surge in event propagation speed" really exists. As in [Luba92] a crude state recovery mechanism would be necessary to correct these errors. This remains a project for future work. In our experiments a population of 7200 balls prove to be the limit where a state recovery mechanism would not be necessary to correct these errors. With populations greater than 7200 balls they will begin to approach a vibrating state and the ease with which vibrating balls can in theory be simulated in parallel should more than make up for lost time spent in state recovery [Luba92].

As can be seen in the graphs in Figures 4.1, 4.2 and 4.3, significant speed ups relative to a simulation without shared variables occur for each of the populations. When we go from a low density population of 2400 to higher density population of 4800, the time saved with the shared variables increases by a modest amount. Without shared variables, as the density increases there are more balls in the critical region to simulate, forcing a processor to move in lock step with its neighbours. However, with shared variables there is more opportunity to use this shared data to recognize opportunities in which it is possible to avoid blocking. Table 4.1, 4.2 and 4.3 shows that there were more occasions with 4800 and 7200 balls than with 2400 balls when the shared variable algorithm could determine that it could safely proceed.

The percentage decrease in execution time gained through the use of shared variables begins to level off at a population of 7200 balls, although there are still impressive reductions in execution time. The shared variable method's execution time is between 34% less with 9 processors and 49% less with 25 processors. These improvements are comparable to those obtained with 4800 balls, which are comparatively 36% and 53% . With more balls in the critical regions there is a greater dependency between the processors as the balls are bounced back and forth across the boundary. The processors are forced to simulate in lock step as they share these balls - no method to reduce the amount of time spent waiting can completely overcome this. As can be seen from Table 4.4 the number of occasions when the algorithm with shared variables is able to avoid waiting with 7200 balls is less than number of occasions with 4800 balls.

As the number of processors increases, the improvement in execution times increases for the shared variable method over the non-shared variable method. With more processors "dividing" the table space, the percentage of space near borders between two processors increases, causing dependencies between the processors.

Fundamentally, the purpose of using shared variables is to reduce the amount of waiting time. As can be seen from Tables 4.1, 4.2 and 4.3, the amount of time a processor wastes waiting for other processors is greatly reduced using shared variables. Figures 4.1, 4.2 and 4.3 show this improvement in execution time. With 2400 balls and 9 processors the time of the shared variable approach is 33% less than that of the algorithm without shared variables. Those improvements rise to 41% with 16 processors and 48% with 25 processors. Similar results occur with the other simulation sizes. With 4800 balls and 9 processors there is a 36% reduction in the execution time, with 16 processors there is a 46% reduction and with 25 processors there is a 53% reduction. Note that with a population size of 2400 balls the increase in execution time from 16 processors to 25 processors results from the number of balls per processor being too small - the simulation is overwhelmed by the overhead of passing messages.

An important influence on the algorithms' efficiency is the size of the critical region. The larger the size of the critical region, the larger is the view a processor has of its neighbour's state. Furthermore, with a larger critical region the likelihood of errors occ·urring [Luba92] decreases. The cost of an increased size is having more balls to maintain in the shared variables which neighbours must examine in order to determine a lower bound. In our experiments, we attempted to choose the size of the critical region with these two factors in mind.

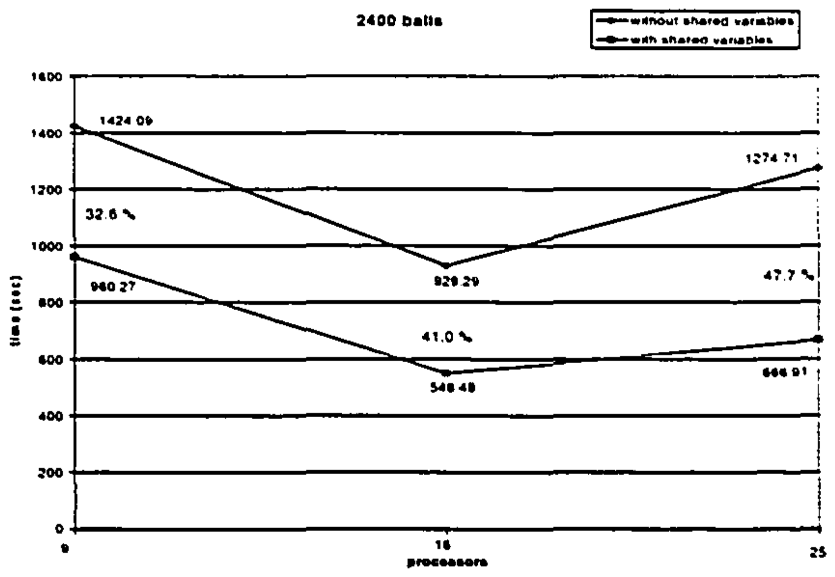

Figure 4.1 Shared variables vs. without shared variables - 2400 ball population

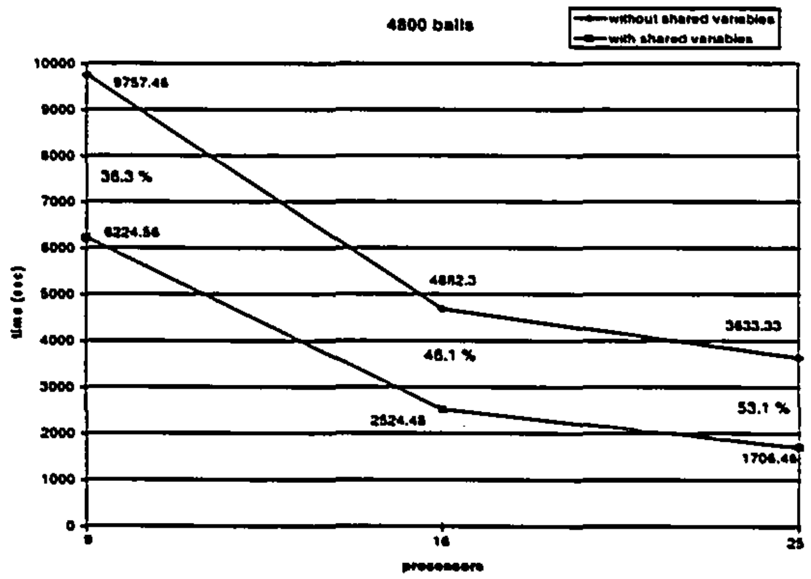

Figure 4.2 Shared variables vs. without shared variables - 4800 ball population

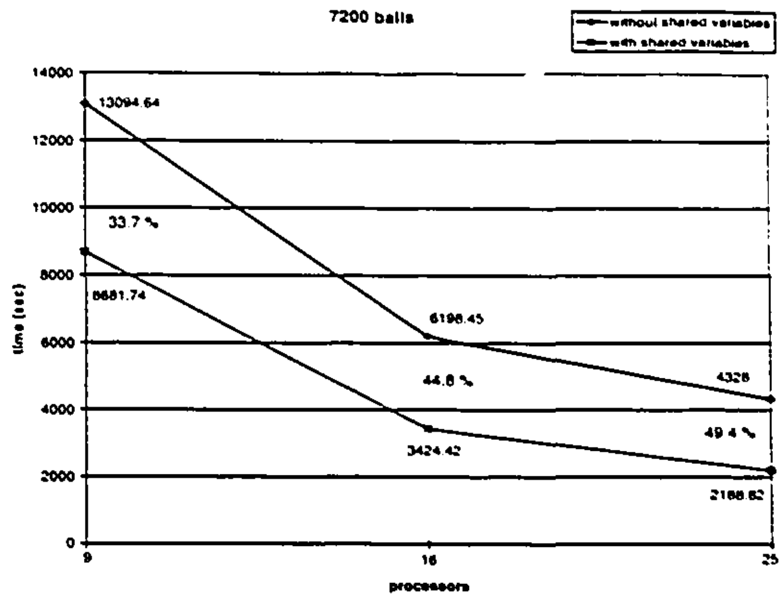

Figure 4.3 Shared variables vs. without shared variables - 7200 ball population

processors	with shared variables	without shared variables
9	1341	6376.2
16	683.26	3379.3
24	383.08	2258.8

Table 4.1 Average total amount of time (sec) processors spent waiting with a population size of 7200 balls

processors	with shared variables	without shared variables
9	2593	4940.3
16	604.04	2824.3
24	357.08	1847.1

Table 4.2 Average total amount of time (sec) processors spent waiting with a population size of 4800 balls

processors	with shared variables	without shared variables
9	232	752.68
16	102.74	425.02
24	83.82	342.69

Table 4.3 Average total amount of time (sec) processors spent waiting with a population size of 2400 balls

processors	2400 balls	4800 balls	7200 balls
9	53.11	131	104.4
16	49.75	149.5	137.31
24	50.24	152	126.52

Table 4.4 Average number of occasions when able to avoid blocking

5 Conclusion

We have presented in this paper, an algorithm for the parallel simulation of billiard balls which makes use of shared variables. The shared variables correspond to a region at the boundary of the (table) segments which are simulated by each processor. As neighbouring processors have access to the contents of these variables, the amount of blocking that is necessary in a conservative simulation is greatly reduced. A project for future work is to implement algorithms to read the shared variables on a distributed memory machine.

Experiments were performed on the BBN Butterfly, in which a purely conservative version of this simulation was compared to one which makes use of our algorithm. Three population levels- 2400, 4800 and 7200 balls were used. In this approach, when a processor wants to process a ball in the critical region, it blocks until the neighbouring processor's simulation time exceeds the time of the event it wishes to process. The shared variable approach resulted in a 30 to 50 percent decrease in the execution time compared to this approach.

Under conditions of extremely high population, it is possible that a simple checkpointing scheme would be necessary such as the one suggested in [Luba92]. This is a project for future investigation.

Bibliography

[Bain88] Bain, W. L. and Scott, D. S., "An Algorithm for Time Syncronization in Distributed Discrete Event Simulation", in Proc. SCS Multiconference on Distributed Simultation, 1998, pp30-33.

[Beck88] Beckman, Brian, et al., "Distributed Simulation and Time Warp Part 1: Design of Colliding Pucks", Proceedings of the SCS Multiconference on Distributed Simulation, Vol. 19, no. 3, (July).

[Bokh81] Bokhari, S. "On the Mapping Problem", IEEE Trans. on Comp., Vol. C-30, (3), March 81, pp 207 - 214.

[Bokh87] Bokhari, S. "Assignment Problems in Parllel and Distributed Computing", Kluwer Academic Publishers, Boston, 1987.

[Bouk92] Boukerche, A., and Tropper, C., "Parallel Simlation on the HypercubeMultiprocessor", Journal of Distributed Computing, 1992, pp 1 - 17.

[Bouk93] Boukerche, A., and Tropper, C., "Parallel Simulation of Communicating Finite State Machines", in Proc. of the 1993 SCS on Parallel and Distributed Simulation, vol. 23, no. 1, 1993, pp. 143-150.

[Brya] Bryant, R. E., "Simulation of Packet Communication Architecture Computer Systems", MIT/LCS/TR-188, Massachusetts Institute of Technology, Cambridge, Mass., Nov. 1977.

[Chan79] Chandy, K.M, and Misra J., "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs", IEEE Trans. on Software Engineering, SE-5, Sept. 1979, pp 440 - 452.

[Chan81] Chandy, K.M. and Misra J., "Asynchronous Distributed Simulation via Sequence of Parallel Computations", CACM, Vol. 24, No.4, April 1981, pp 198-206.

[Chan82] Chandy, K.M. and Misra J., "Distributed Computation on Graphs: Shortest Path Algorithms", Common ACM 25(11), 1982, pp 833-837.

[Chan89] Chandy, K. M., and R. Sherman, "Space-Time and Simulation", in Proceedings of the 1989 SCS Multiconference on Distributed Simulation, 1989, 53-57.

[Cido87] Cidon, I., Jaffe, J. M., Sidi, M., "Local Distributed Deadlock Detection by Cycle Detection and Clustering", IEEE Trans. Software Eng. SE-13, 1987, pp 3-14.

[Comf82] Comfort, J. C., "The Design of a Multi-microprocessor Based Simulation Computer-I", Proceedings of the Fifteenth Annual Simulation Symposium, March 1982, pp 45-53.

[Cote92] Cote, C. and Tropper C., "On Distributed and Pseudosimulation", 1992 Workshop on Parallel and Distributed Simulation, SCS, Vol. 24, no. 3, Jan 1992, pp. 97-106.

[Das94] Das, S., R., and Fujimoto, R., M., "An Adaptive Memory Management Protocol for Time Warp Parallel Simulation.", in Proc. 1994 ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems, pp. 201-210, May 1994.

[Elma86] Elmagarmid, A. K., Datta, A. K. and Liu, M. T., "Deadlock Detection Algorithms in Distributed Database Systems", Proc. IEEE Int. Conf. on Data Eng., Los Angeles, California, Feb., 1986.

[Fish78] Fishman, G., S., "Principles of Discrete Event Simulation", Wiley, New York, 1978.

[Fuji88] Fujimoto, R. M., "Performance Measurements of Distributed Simulations Strategies", In Proc. 1988, SCS Multiconference on Distributed Simulation, Vol. 19, No. 3, Feb. 1988, 14-20.

[Fuji89] Fujimoto, R. M., "Time Warp on a Shared Memory Multi-Processor", in Proceedings of the 1989 International Conference on Parallel Processing, Aug. 1989, 242-249.

[Fuji90] Fujimoto, R. M., "Parallel Discrete Event Simulation", in Proc. of the Winter Simul. Conf., 1989, 19-28.

[Garc79] Garey, M. R., and Johnson, D. S., "Computers and Intractability: A Guide to the Theory of NP-completeness", W.H. Freeman and Company, New York, 1979.

[Gros87] Groselj, B., and Tropper, C., "Pseudosimulation: An Algorithm for Distributed Simulation with Limited Memory", International Journal of Parallel Programming, Vol. 15, No. 5, Oct. 1987, pp 413 - 456.

[Gross88] Groselj, B., and Tropper, C., "The Time-of-Next-Event Algorithm", in Proceedings of the 1988 Distributed Simulation Conference, SCS Simulation Series, Vol. 19, No. 3, Feb. 1988, San Diego, CAL, pp 25-29.

[Gros91] Groselh, B., and Tropper, C., "The Distributed Simulation of Clustered Processes", Distributed Computing, vol. 4 pp. 111-121, 1991.

[Hont89] Hontalas, P., et al. "Performance of the Colliding Pucks Simulation on the Time Warp Operating Systems", Distributed Simulation, SCS Simulation Series, vol 21, no. 2, pp 3-9.

[Jeff82] Jefferson, D. R., and H. Sowizral, H., "Fast Concurrent Simulation using the Time Warp Mechanism, Part I: Local Control", Computer Science Department, U.S.C., TR-83-204, Los Angeles, CAL 90089-0782.

[Jeff85] Jefferson, D. R., "Virtual Time", ACM Trans. Prog. Lang. Syst. 77, (3), July 1985, 405-425.

[John77] Johnson, D. B., "Efficient Algorithms for Shortest Paths in Sparse Networks", J ACM, Vol. 24, 1977, pp 1-13.

[Kaud87] Kaudel, F. J., "A Literature Survey on Distributed Discrete Event Simulation", Simuletter, Publication of SIGSIM, ACM Press, Vol. 18, June 1987, 11-21.

[Lin89] Lin, Y.-B., and Lazowska, E. D. "A study of Time Warp Rollback Mechanisms", TR 89-09-07, Department of Computer Science and Engineering, University of Washington, 1989.

[Lin92] Y. Lin, "Memory Management Algorithms for Optimistic Parallel Simulation", Information Sciences, vol. 77, pp. 119 - 140, 1992.

[Luba88] Lubachevsky, B., "Bounded Lag Distributed Discrete Event Simulation", Distributed Simulation, SCS Simulation Series, 1988, pp 183-191.

[Luba91] Lubachevsky, B., "How to Simulate Billiards and Similar Systems" Journal of Computational Physics 94, 1991, pp 255-283.

[Luba92] Lubachevsky, B., "Simulating Colliding Rigid Disks in Parallel Using Bounded Lag Without Time Warp", Distributed Simulation, SCS Simulation Series, vol 22, no. 1, pp 194-202.

[Made88] Madisetti, J., Walrand J., and Messerschmitt D., "WOLF: A Rollback Algorithm for Optmistic Distributed Simulations", 1988 Winter Simulation Conference Proceedings, Dec. 1988.

[Mehl93] Mehl, H. and Hammes, S., "Shared Variables in Distributed Simulation", Proc 7th Workshop on Parallel and Distributed Simulation (PADS93), 1993, IEEE Computer Society Press, vol.23, no. 1, pp 68-76.

[Misr86] Misra, J., Chandy, K. M., "Distributed Discrete-event Simulation", ACM Computing Surveys, 18(1), March 1986, 39-65.

[Nata86] Natarajan, N., "A Distributed Scheme for Detecting Communication Deadlocks", IEEE Trans. Soft. Eng., Vol. SE-12, No. 4, pp. 531-537, April 1986.

[Nico88] Nicol, D. M., "Parallel Discrete-Event Simulation of FCFS Stochastic Queuing Networks", in Proc. of the ACM SIGPLAN Symposium on Parallel Programming, Enviroments, Applications, and Languages, Yale University, July 88.

[Ni85] Ni, L. et. al., "Distributed Drafting Algorithm for Load Balancing", IEEE Trans. Sof. Eng., Vol. 11(10), 1986.

[Peac79] Peacock, J. K., Wong, J. W., and Manning, E. G., "Distributed Simulation Using a Network of Processors", Computer Networks 3, (1), Feb. 1979, 44-56.

[Reed88] Reed, D. A. and Maloney, A. "Parallel Discrete Event Simulation: The Chandy-Misra Approach", in Proc. 1988 SCS Multiconference on Distributed Simulation, 1988, 8-13.

[Reih90] Reiher, P., Fujimoto, R., Bellenot, S., and Jefferson, D., "Cancellation Strategies in Optimistic Execution Systems", Proceedings 1990 SCS Multiconference on Distributed Simulation, 1990.

[Righ89] Righter, R., Walrand J. C., "Distributed Simulation of Discrete Event Systems", in Proceedings of IEEE, Vol. 77, No. 1, Jan. 1989, 99-113.

[Stan84] Stankovic, J. and Sidhy, I., "An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups", Proc. 4th Int. Conf. Distributed Comput. Systems, 1984.

[Su89] Su, W. K., and Seitz, C.L., "Variants of the Chandy-Misra-Bryant Distributed Simulation Algorithm", in Proceedings of the 1989 SCS Multiconference on Distributed Simulation, Vol. 21, No. 2, March, 1989, 38-43.

[Wagn89] Wagner, D. B., Lazowska, E. D. and Bersahd, B., "Techniques for Efficient Shared Memory Parallel Simulation", in Proc. 1989 SCS Multiconference on Distributed Simulation, 1989, 29-37.