THE COST OF TERMINATING PARALLEL DISCRETE-EVENT SIMULATIONS

by

Vasant Sanjeevan

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

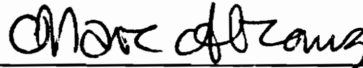
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

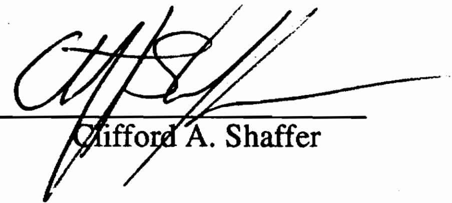
Computer Science

APPROVED:



Marc Abrams, Chairman

Richard E. Nance



Clifford A. Shaffer

June, 1992

Blacksburg, Virginia

c. 2

LD
5655
V855
1992
S322
c. 2

The Cost Of Terminating Parallel Discrete-Event Simulations

by Vasant Sanjeevan

Committee Chairman: Marc Abrams
Computer Science

(ABSTRACT)

Simulation models use many different rules to decide *when* to terminate. Parallel simulations generally use a single, simple rule: each process comprising the simulation terminates after a predefined period of time. A number of parallel simulation protocols have been proposed that enforce constraints on the order in which processes are scheduled in parallel so that the result of a parallel simulation is the same as that of the corresponding sequential simulation. Parallel simulations protocols can be broadly classified into two categories: conservative and optimistic. Conservative protocols can be subclassified into synchronous and asynchronous protocols. In this thesis, our objective is to compare the predicted and measured wall clock running times of parallel simulations for conservative-synchronous and optimistic protocols with and without termination conditions.

We propose eight algorithms for mechanically adding an arbitrary termination condition to a conservative-synchronous non-terminating parallel simulation. Informal arguments about the expected performance of each algorithm are made, and the arguments are confirmed through measurement of the simulation of a torus network with three termination conditions using the conservative-synchronous Bounded Lag protocol on a shared memory multiprocessor. We also propose four algorithms for mechanically adding a termination condition to an optimistic non-terminating parallel simulation. We make informal arguments about the expected performance of these algorithms and report on the actual performance of the simulation of the torus network benchmark with two of these

algorithms and the same three termination conditions using the optimistic Time Warp protocol on a message-passing multiprocessor. In addition to the torus network benchmark for the optimistic protocol, we also report on the performance of a colliding pucks simulation with these two algorithms and three additional termination conditions.

Our study indicates that termination conditions which require exhaustive evaluation introduce substantial running time overhead. We propose and evaluate a scheme to reduce this overhead.

ACKNOWLEDGMENTS

I was extremely privileged to have Dr. Marc Abrams as my thesis advisor. He kept me going at my thesis through some difficult times. He always has a helpful suggestion to offer and is always ready to make time for his students. I always wonder how he is able to pull out a specific publication unerringly from his shelves without hesitation.

Thanks to Dr. Dick Nance and Dr. Cliff Shaffer for reading this thesis and offering their experience and helpful insight.

I could not have pursued graduate study without the moral and financial support of my parents, and many thanks are due to them.

To my friends Vish, Sam, Anil, Mahesh and Arun, thanks for their camaraderie and helpfulness. I will miss those Friday evenings. Thanks to Noopur for lifting my spirits during the final days, and also to the mysterious Rikachan.

Lastly, I would like to thank the personnel of JPL for the use of their resources. I made extensive use of JPL's TWOS to run my simulations.

TABLE OF CONTENTS

Page

Abstract. ii

Acknowledgments. iv

Table of Contents. v

List of Figures. vii

1. Introduction 1

1.1 Important Definitions. 2

1.2 The Parallel Simulation Termination Problem 3

1.3 Parallel Discrete-event simulation. 4

1.3.1 Conservative Protocols. 5

1.3.1.1 Conservative-asynchronous Protocols. 5

1.3.1.2 Conservative-synchronous Protocols. 6

1.3.2 Optimistic Protocols. 9

1.4 General Termination Algorithms Proposed in Literature 10

1.5 Outline of Proposed Work. 16

1.5.1 Conservative-synchronous Protocols. 16

1.5.2 Optimistic Protocols. 17

2. Related Research 18

2.1 Relation to Termination Detection Problem. 18

2.2 Relation to Global State Detection Problem. 19

2.3 Relation to Stability Detection Problem. 19

2.4 Outline of Previous Research. 19

3. Conservative-Synchronous Protocols 22

3.1 Termination Algorithms for Conservative-Synchronous Protocols 22

3.1.1 Algorithm IS23 24

3.1.2 Algorithm IS12 24
3.1.3 Algorithm IP12 25
3.1.4 Algorithm DS23 26
3.1.5 Algorithm DS12 28
3.1.6 Algorithm DS23R. 30
3.1.7 Algorithm DP23 32
3.1.8 Algorithm DP23R. 32
3.2 Conservative-Synchronous Termination Algorithm Performance 32
3.2.1 Benchmarks Used 32
3.2.2 Predicted Performance 35
3.2.3 Measured Performance 37
4. Optimistic Protocols 43
4.1 Termination Algorithms for Optimistic Protocols 43
4.1.1 Algorithm IS 45
4.1.2 Algorithm ES 45
4.1.3 Algorithm IP 46
4.1.4 Algorithm EP 46
4.2 Optimistic Termination Algorithm Performance 46
4.2.1 Benchmarks Used 46
4.2.2 Predicted Performance 47
4.2.3 Measured Performance 50
5. Conclusions 61
References 64
Vita. 66

LIST OF FIGURES

Page

Figure 1: Algorithm For LP i Of A Bounded Lag Simulation 8

Figure 2: Illustration Of Space Time Rectangle For A Simulation With Three Attributes. 11

Figure 3: Example Of A Stable Termination Condition. 14

Figure 4: Example Of A Unstable Termination Condition 14

Figure 5: A 3 x 3 Torus Queuing Network 33

Figure 6: Wall Clock Time required to complete simulation for N^2 threads with no termination condition for the torus network benchmark with the conservative-synchronous protocol. 39

Figure 6a: Efficiency of the torus network simulation without termination condition for the conservative-synchronous protocol for extreme values of N. 39

Figure 7: Wall Clock Time required to complete simulation for N^2 threads with IS23+T0 and DS12+T1 with the conservative-synchronous protocol. 40

Figure 8: Wall Clock Time required to complete simulation for N^2 threads with IS23+T2 and DS12+T3 with the conservative-synchronous protocol. 42

Figure 9: Wall Clock Time required to complete simulation for N^2 LP's with no termination condition for the torus network benchmark with the optimistic protocol. 52

Figure 9a: Efficiency of the torus network simulation for the optimistic protocol for extreme values of N. 52

Figure 10: Wall Clock Time required to complete simulation for N^2 LP's with no termination condition for the colliding pucks benchmark with the optimistic protocol. 53

Figure 10a: Efficiency of the colliding pucks simulation for the optimistic protocol for extreme values of K. 53

Figure 11: Wall Clock Time required to complete simulation for N^2 LP's with IS + T0 with the optimistic protocol. 55

Figure 12: Wall Clock Time required to complete simulation for N^2 LP's

with ES + T1 with the optimistic protocol. 55

Figure 13: Wall Clock Time required to complete simulation for N^2 LP's with FIFO+T1 with the optimistic protocol. 56

Figure 14: Wall Clock Time required to complete simulation for N^2 LP's with IS + T2 with the optimistic protocol. 56

Figure 15: Wall Clock Time required to complete simulation for N^2 LP's with ES + T3 with the optimistic protocol. 58

Figure 16: Wall Clock Time required to complete simulation for K LP's with IS + P0, ES + P1 with the optimistic protocol. 58

Figure 17: Wall Clock Time required to complete simulation for K LP's with ES + P2 with the optimistic protocol. 60

CHAPTER 1: INTRODUCTION

One method of executing discrete-event simulations on parallel and distributed computer architectures is based on space decomposition (Chandy and Sherman 1989): the simulated system is decomposed into a number of subsystems, each of which is modeled by an independently schedulable piece of code called a *thread,* also referred to as a *process* or *task*. Threads execute on one or more physical processors and simulate the subsystems that they model, communicating with each other as they do so. A number of *parallel simulation protocols* (Fujimoto 90) have been proposed that enforce constraints on the order in which threads are scheduled in parallel so that the result of a parallel simulation is the same as that of the corresponding sequential simulation. In general, parallel simulation of a system takes less wall clock time to execute than the corresponding sequential simulation, unless the communication and synchronization overhead between processes outweighs the performance gains of parallel execution.

Most parallel discrete-event simulators can handle only a single, simple termination condition: each thread comprising the simulation terminates when it has simulated up to a predefined *simulation time*, defined in Section 1.1. To apply parallel simulation protocols to any simulation model, algorithms must be developed to use arbitrary termination conditions. The *parallel simulation termination problem* (Abrams and Richardson 1991), defined in Section 1.2, is inherently difficult due to asynchronous computation, and the overhead of evaluating terminating conditions using data distributed among the threads comprising the simulation.

Consider a simulation of two air forces battling one another. The following termination condition illustrates the parallel simulation termination problem: "terminate when one air force has at least twice as many surviving aircraft as the other air force."

This thesis evaluates the performance of two general algorithms to solve the problem

of terminating parallel discrete-event simulations, proposed in (Abrams and Richardson 1991). The algorithms make no assumptions about the simulation protocol or computer architecture used to execute a simulation program; therefore the algorithms apply to any parallel simulation protocol and either synchronous or asynchronous execution on any sequential or parallel architecture. To evaluate the performance of these algorithms, we choose specific parallel simulation protocols and map the algorithms to specific architectures. This thesis proposes mappings of the algorithms to two specific parallel simulation protocols and two architectures, and compares the predicted and measured wall clock running times for both protocols.

1.1 Important Definitions

This section defines terms used in successive sections.

A *discrete-event simulation* computes the time evolution of a set of *attributes*, one of which is *simulation time*. The attributes are assigned initial values prior to the start of the simulation. A discrete-event simulation consists of successive assignments of values to one or more attributes by *events*. An *event* is implemented by a piece of code whose *execution* modifies one or more attributes when it is executed. The new value assigned to a simulation time attribute is always greater than or equal to its previous value. Every event occurs at a single instant of simulation time and is executed *atomically*, that is, the attributes modified by an event appear to change their value simultaneously. The simulation time of an event is the value of the simulation time attribute when event execution completes.

The set of all events is partitioned among a set of *logical processes* (LP's). Each LP is executed by a unique thread. Interaction between LP's is effected through sending and receiving *event messages*. An LP is said to *schedule* an event on a destination LP when it inserts an event in the input queue of the destination LP. An LP is said to *delete* an event from an input queue of an LP when it removes an event from the input queue of the LP.

Each event scheduled by one LP for another is sent in a message with a *timestamp*. The timestamp defines the simulation time at which the destination LP will execute the message. The *local virtual time* (LVT) of an LP is defined as being equal to the smallest timestamp of the set of events that the LP is currently executing or waiting to execute.

In a *parallel discrete-event simulation*, the set of attributes comprising the simulation is partitioned among a number of LP's. The attributes assigned to an LP are termed *local to* that LP.

A *termination condition*, denoted *C*, is a boolean valued function whose domain is a subset of all simulation attributes, including the simulation time attribute. A simulation can terminate at any simulation time *t* such that the values of all attributes required to evaluate *C* are known at time *t* and the evaluation of *C* using these attribute values yields *true*.

Termination conditions are either *stable* or *unstable*. A stable condition remains *true* once it becomes *true* (Chandy and Misra 1988). An example of a stable (respectively, unstable) condition is, "the number of jobs processed by the simulation at simulation time *t* *exceeds (equals)* 100,000."

A simulation *output measure* is the value of a function which is evaluated after the simulation has terminated, using simulation attribute values at the time *t* for which *C* is *true*.

1.2 The Parallel Simulation Termination Problem

We consider the *parallel simulation termination problem* (Abrams and Richardson 1991): given a non-terminating parallel simulation and a termination condition *C*, the problem is to state an algorithm that modifies the non-terminating parallel simulation to:

P1. find a value of simulation time, denoted by *t*, such that function *C* evaluated using simulation attribute values at time *t* has value *true*, and

P2. report the value of each simulation output measure at time *t*.

In this thesis, we assume that there always exists such a time *t*.

This problem formulation is based on the belief that in a commercial parallel simulation system, a user would like to specify a simulation model without worrying about termination (hence the simulation is non-terminating), and then separately specify a variety of termination conditions. This formulation makes it easier to make changes to the simulation and the termination condition independently, which is important because simulation models must withstand modifications occurring through a typical lifetime of many years. The problem formulation permits a termination condition to be automatically superimposed on the non-terminating simulation.

Detecting a time t when C is *true* is difficult for four reasons. First, C is in general a function of simulation attributes that are local to two or more LP's that have some degree of asynchrony. Second, there may be constraints on the times at which C must hold, for example, specifying the "first time" that an event occurs. Third, C may be a complex function, such as a numerical integration, so each evaluation of C may be computationally expensive. Finally, C is generally not stable; therefore at worst C must be reevaluated each time any LP executes an event.

1.3 Parallel Discrete Event Simulation

In a parallel discrete-event simulation, the LP's comprising the simulation execute in parallel, simulating events and interacting with one another at various points in simulation time (Fujimoto 1989). Each event is associated with a simulation time value called a *timestamp*. Each LP has an *input queue* which contains all *pending* events, that is, events that have not yet been executed. The LP removes the event on the input queue with the smallest timestamp and *executes* it. An LP always selects the event with the smallest timestamp, E_{min}, from the event list to execute. The LP cannot choose some other event E_x, because execution of E_x could modify state variables used by E_{min}, and amounts to simulating a system in which the future could affect the past. Errors of this nature are

called *causality* errors.

Parallel simulation protocols can be broadly classified into two categories: *conservative* and *optimistic* (Fujimoto 89). Conservative protocols can be further subclassified into *synchronous* and *asynchronous* protocols. The three protocols are described below.

1.3.1 Conservative Protocols

Conservative protocols strictly avoid the possibility of a causality error ever occurring. They rely on some strategy to determine when it is *safe* to execute an event. More precisely, if E_{min} for an LP has timestamp T_1, and the LP can determine that it is impossible for it to later receive another event with a timestamp smaller than T_1, then the LP can guarantee that causality will be preserved, and therefore it is said to be safe to execute E_{min}. Processes containing no safe events must block, which can limit the number of threads that can be executed in parallel. If a cycle of processes with no safe events arises, each process in that cycle must block, and the simulation deadlocks (Fujimoto 90).

1.3.1.1 Conservative-asynchronous Protocols

Two families of conservative-asynchronous protocols are based on *deadlock avoidance* and on *deadlock detection and recovery*. We describe the deadlock avoidance scheme developed by Chandy and Misra (Chandy and Misra 1979) to illustrate the operation of a conservative-asynchronous protocol. A deadlock detection and recovery scheme is described in (Misra 1986).

In order to determine when it is safe to execute an event, the sequence of timestamps of events sent from one LP to another must be non-decreasing. This guarantees that the timestamp of the last event scheduled for an LP by another LP is a lower bound on the timestamp of subsequent events that might be scheduled for it later by the other LP.

Events arriving on each incoming link are stored in FIFO order, which is also timestamp order because of the above restriction. Each link has a clock associated with it that is equal to the timestamp of the event at the front of that link's queue, if the queue contains a message, or the timestamp of the last received message, if that queue is empty. The LP repeatedly selects the link with the smallest clock and executes any message on that queue's link. If the selected queue is empty, the thread blocks. This protocol guarantees that each thread will only executes events in non-decreasing timestamp order and adheres to the causality constraint.

1.3.1.2 Conservative-synchronous Protocols

Conservative-synchronous protocols (Lubachevsky 1989), (Nicol 1990), (Ayani 1992) execute a loop which determines which events are safe to execute, and then executes them. Barrier synchronizations are used to keep one iteration of the loop from interfering with another. Because barrier synchronizations are necessary, these algorithms are best suited to architectures that permit efficient barrier synchronization, such as shared memory machines.

Each LP calculates a simulation time window to reduce the overhead associated with determining when it is safe to execute an event. The lower edge of the window is defined as the minimum timestamp of any unexecuted event. The upper edge of the window depends on the protocol being used. Only those events whose timestamps lie within the window are eligible for executing.

The advantage of this approach over the conservative-asynchronous approach is that the memory required to store simulation attributes is proportional to and bounded by the size of the time window. Another advantage of conservative-synchronous protocols is that they do not suffer from the blocking problems faced by conservative-asynchronous protocols. Also, on a shared memory architecture, we do not need to lock shared variables

if the window is selected to preclude two LP's from simultaneously accessing any shared variable. A conservative-synchronous protocol could only fail to achieve optimal performance, with all processors being kept busy, if there are insufficient number of events to keep all processors busy (Nicol 1990).

Lubachevsky proposes a conservative-synchronous algorithm called the Bounded Lag simulation protocol, shown in Figure 1 (Lubachevsky 1989). Each LP comprising the simulation asynchronously executes this algorithm in parallel with other LP's and periodically synchronizes with them.

In Figure 1, M denotes the number of LP's. We denote the M LP's comprising a simulation by $LP_1, LP_2, \ldots, LP_M$. The minimum timestamp of all events in the input queue of LP_i is stored in `NEXT[i]`. The estimate of the earliest time when the history at node `i` can be changed by any other node is denoted by `ALPHA[i]`. `ALPHA[i]` is the expected timestamp of the event with the smallest timestamp which can be scheduled for node `i` by any other node. The upper bound on the difference between the simulation times of events being executed by all LP's in the current iteration of the outer loop of the algorithm is called the *bounded lag* of the algorithm and is denoted by B. Stop variable SV is set to *true* when the termination condition becomes *true*. The algorithm assumes in statement S6 that `t` is always assigned a value that is strictly greater than its current value (i.e., execution of each event requires a non-zero amount of simulation time). We assume without loss of generality that each LP has at least two attributes local to it, one required to evaluate C, and the other required to evaluate the output measure function `OMF`. The set of M attributes required to evaluate the termination condition C is represented by array A, and the set of M attributes required to evaluate `OMF` is represented by array O. Variable `E` is a data structure which contains the event last removed from the input queue. Variable `t` contains the timestamp of the event just executed.

Each LP maintains an input queue of events scheduled for it in timestamp order. The

```
constant
    M=number of LP's comprising simulation;
    B=value of the Bounded Lag;
global
    SV:=false:boolean;
    Floor:=0.0, ALPHA[1..M], NEXT[1..M]:real;
    A[1..M]:=initial attribute values to evaluate C:real;
    O[1..M]:=initial attribute values to evaluate OMF:real;
local
    t:real;      /*real variable which contains a timestamp*/
    E:event;     /*data structure which contains event data*/
    i:integer;   /*variable which identifies thread        */

S0: NEXT[i]:= minimum timestamp of all events in LPi's
              input queue at start of simulation;
S1: while (not SV){
S2:    compute ALPHA[i];
S3:    synchronize;      /* barrier 1 */
S4:       while ((NEXT[i] < Floor + B)&&(NEXT[i] < ALPHA[i])){
S5:          E:=next event from input queue
S6:          t:=timestamp of event E;
S7:          execute E;   /* may modify A[i] and O[i] */
S8:          optionally schedule new events for LPi
               or other LP's
S9:          delete the processed events from LPi;
S10:         NEXT[i]:= minimum timestamp of all events in LPi's
                      input queue;
          }
S11:   synchronize;      /* barrier 2 */
S12:   if(i=1) Floor :=min (NEXT[1],NEXT[2],...,NEXT[M]);
S13:   synchronize;      /* barrier 3 */
S14:}
```

Figure 1: Algorithm for LP_i of a Bounded Lag simulation

timestamp of an event defines the simulation time at which the LP will execute the message. The *next* event on an input queue is the event on the queue with the smallest timestamp.

1.3.2 Optimistic Protocols

All events in an optimistic simulation are ordered by *virtual time* (Jefferson 1985). Whenever an LP sends an event message, the timestamp of the event message must equal or exceed the LVT of that LP. Each LP maintains an input queue of all unexecuted events scheduled for it accessable in ascending timestamp order. When an LP finishes executing an event, the operating system scheduler performs a context switch to the LP with the lowest LVT, which then executes the next event in its input queue. In an optimistic protocol, unlike a conservative-synchronous protocol, an LP executes the smallest timestamp event in its input queue regardless of whether it is safe. Whenever an event message arrives with a timestamp smaller than the LVT of the LP, called a *straggler*, the LP detects that it has executed an unsafe event, and must roll back to a time that does not exceed the timestamp of the straggler.

Rollback requires an LP to maintain a history of *states* and associated timestamps as it executes events. The *state* of an LP at a given simulation time consists of all local simulation attribute values at that simulation time instant. As the simulation proceeds, each LP periodically appends a copy of its state along with the current GVT to its state history. When an LP needs to be rolled back to a particular simulation time *t*, the LP retrieves the state with the largest simulation time instant not exceeding *t* from its state history, and assigns to its local simulation attributes the corresponding retrieved values. Also, all event messages which were sent by the LP with a timestamp greater than *t* are canceled by sending *anti-messages* with the same timestamp *t*.

Some actions performed during event execution, such as printing a character on a sheet of paper, cannot be rolled back. Such actions must wait for *commitment*. The

simulation protocol calculates a quantity called *global virtual time* (GVT) to determine when actions can be safely committed. At any point in the simulation, GVT is less than or equal to the minimum of all current LVT's. The protocol calculates GVT periodically. An action *commits* when GVT exceeds the simulation time of that action. GVT is also used to determine when old states can be garbage collected, freeing memory for reuse. An LP with an empty input queue is defined to have an LVT of positive infinity. When GVT reaches positive infinity, the simulation terminates execution.

1.4 General Termination Algorithms Proposed in the Literature

Let W denote a set which contains all simulation times at which a simulation attribute required to evaluate termination condition C changes value. Let n denote the number of simulation attributes required to evaluate C (Note that $n=M$ in Figure 1). Let $A=\{a_0. \ldots .a_{n-1}\}$ denote the set of attributes required to evaluate C. (Set A corresponds to array A in Figure 1). There exists a mapping from each $t \in W$ to an n-tuple containing the values of the n simulation attributes, a_0 to a_{n-1}, required to evaluate C at time t. Set W contains an infinite number of elements, because the simulation is non-terminating. The solution to P1 (find a time t at which $C(t)$ is *true*) proposed in (Abrams and Richardson 1991) is described below.

Solution of Problem P1

The solution to problem P1 views the simulation termination problem as a type of search problem. Consider the space-time rectangle, in which space corresponds to the set of simulation model attributes A and time corresponds to the set of times in W for which some element of set A changes value. This treatment is a modification of (Chandy and Sherman 1989), which uses LP's, and not set A, for the space dimension. Figure 2 illustrates a portion of a space-time rectangle. In Figure 2, attribute a_1 changes value at

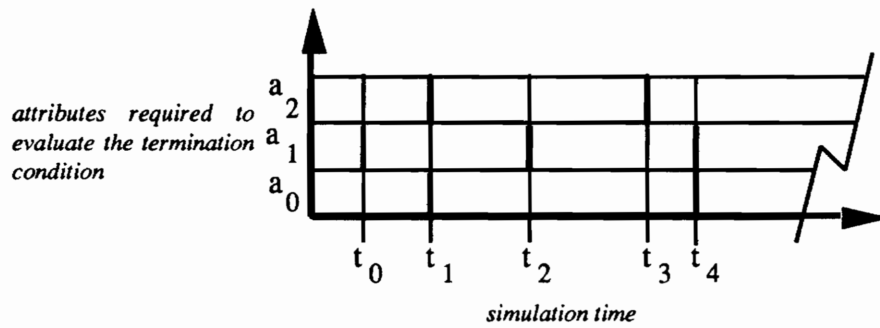


Figure 2: Illustration of space-time rectangle for a simulation with three attributes. The heavy vertical lines denote assignment of new values to attributes

time $t_0 \in W$, both a_0 and a_2 change value at time $t_1 \in W$, and so on. The parallel simulation termination problem is equivalent to searching the space-time rectangle for *any* time $t \in W$ at which condition C, evaluated using the attribute values at time t, yields *true*. It is useful to view P1 as a search problem because our objective is to locate a time t for which $C(t)$ is true by testing a minimum number of times in set W to potentially reduce the wall clock time to run the simulation. We could test each $t \in W$ in ascending order (a linear search), but other orders may potentially reduce the wall clock time to run the simulation.

Two termination algorithms are proposed in (Abrams and Richardson 1991):

> ***Exhaustive Termination Algorithm:*** Evaluate $C(t, A)$ at each simulation time, $t \in W$, in ascending time order until $C(t, A) = true$.
>
> ***Interval Termination Algorithm:*** Choose any subset of W such that the subset contains a time for which $C(t, A) = true$. Evaluate $C(t, A)$ at each simulation time represented by the chosen subset in ascending time order until $C(t, A) = true$.

Detecting a simulation time at which a unstable condition holds requires at worst an exhaustive termination algorithm. This requires the evaluation of C at *every* simulation time $t \in W$, in ascending order, for which an LP changes the value of *any* simulation attribute in the domain of C, until $C(t, A) = true$. The number of evaluations can grow exponentially in the number of attributes in a simulation model. Therefore it could be the case that a terminating parallel simulation requires more wall clock time to execute than a sequential simulation requires.

Interval termination generally requires less wall clock time than exhaustive termination to identify a simulation time at which $C(t, A)$ is *true*. The challenge in using interval termination is choosing a subset of W that contains a time t for which $C(t, A) = true$ before execution of a simulation begins. However such a subset can always be chosen

before execution for a stable condition. Consider Figures 3 and 4, which characterize a stable and a unstable termination condition, respectively. From Figure 3, for a stable condition, interval termination works for *any* subset of W that contains a time t whose value is greater than the first time for which C evaluates to *true* for a stable condition. We can guarantee that such a time t exists in a subset of W if we choose a subset such that for any time $t_1 \in W$, there exists $t_2 \in W$, where $t_2 > t_1$. Examples of subsets of W that work for stable termination conditions are to evaluate the termination condition every tenth time that an attribute changes value, and to evaluate the termination condition every five seconds of wall clock time.

Rules that evaluate the termination condition at fixed intervals have a disadvantage in that the optimal evaluation frequency is generally dependent on the input data to the simulation, and also on constants in the code, if used. Too frequent an interval devotes many processor cycles to evaluating C and can degrade the performance of a parallel simulator. Too infrequent an interval (i.e., once every 10 hours of wall clock time) will allow a simulation to run for a long time after C becomes *true*.

Solution of Problem P2

Three solutions to solve problem P2 (report the value of each output measure) are proposed in (Abrams, Sanjeevan and Richardson 1992). These solutions are independent of the solutions to problem P1. *Dissociative Generation* decouples the simulation from the calculation of output measures. *Retrospective Generation* selects a termination time, denoted T_c, in the past of all LP's, while *Prospective Generation* selects a time T_c in the future of all LP's. All three algorithms require the addition of an operating system thread, referred to as a *measure generator* (MG). Let T_f denote the largest simulation time for which C has been evaluated *false*.

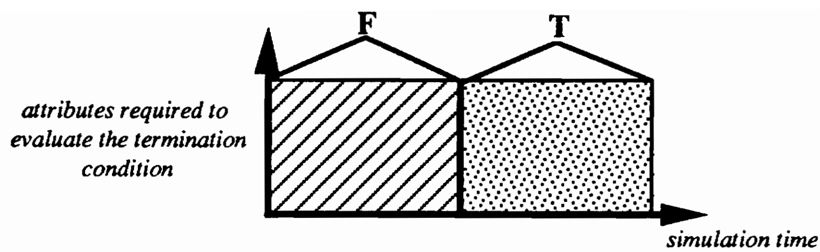


Figure 3: Example of a stable termination condition. "T" and "F" denote regions of space-time in which the condition is *true* and *false*, respectively

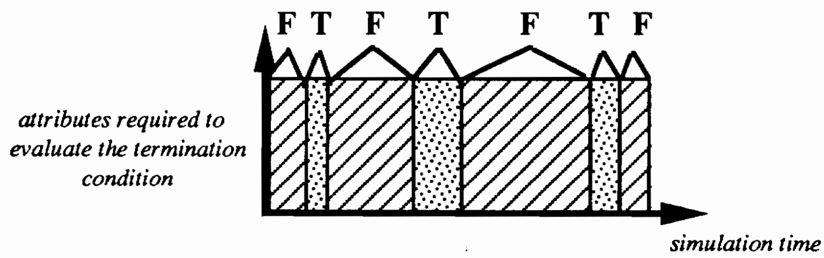


Figure 4: Example of a unstable termination condition

Dissociative Generation (Suitable for protocols without rollback, such as conservative protocols): Each time a new value is assigned to an attribute required for evaluation of any output measure, the LP that owns the attribute makes accessible to MG the simulation time at which the assignment occurs, along with the new value. Whenever T_f changes the termination algorithm communicates this change to MG. When termination is detected, MG evaluates the output measures using attribute values at time T_c.

Retrospective Generation (Suitable for optimistic protocols and any other protocol that can be modified to include rollback, such as conservative-synchronous protocols): When termination time T_c is detected, T_c is distributed to all LP's, and each LP is rolled back to simulation time T_c. Each LP that owns an attribute required to calculate output measures makes the current attribute values, corresponding to time T_c, accessible to MG, and terminates. MG calculates the measure when it has received all required attributes.

Prospective Generation (Suitable for any protocol and all stable conditions, and some unstable conditions): When a simulation time t is found at which C holds, LP's are inhibited from simulating. The maximum of the local clock value of each LP and the send time of each message in transit is calculated, denoted T_H.

Based on the value of t, the Termination Detector selects a time $T_c > T_H$, such that C is guaranteed to be *true* when evaluated at time T_c. The algorithm broadcasts the value T_c to all LP's, and the LP's are permitted to continue simulation. Each LP continues simulation until its local clock equals or first exceeds T_c. Output measures are calculated as described for Retrospective Generation.

It is always possible to select a $T_c > T_H$ at which C holds for stable termination conditions when evaluated at time T_c. By the definition of stability, C is *true* for any T_c that equals or exceeds T_H.

1.5 Outline of this Thesis

This thesis addresses the problem of implementing parts P1 and P2 of the simulation termination problem using a conservative-synchronous protocol on a shared memory architecture and using an optimistic protocol on a message passing architecture. We compare the predicted and measured wall clock running times of both algorithms with and without termination conditions.

Algorithms to terminate optimistic and conservative-asynchronous simulation protocols have been proposed by Abrams and Richardson (Abrams and Richardson 1991, Richardson 1991). They conclude that optimistic protocols have an inherent advantage over conservative-asynchronous protocols for the following reason. Both stable and unstable termination conditions can be implemented in a straightforward manner with a negligible increase in memory requirements with optimistic protocols. However, conservative-synchronous protocols can only handle stable conditions with a negligible increase in memory requirements. A conservative-asynchronous protocol requires an unbounded amount of memory to handle unstable termination conditions. In contrast, conservative-synchronous protocols bound the memory required to handle both stable and unstable termination conditions. This combined with their efficiency and ease of implementation make them attractive for implementing termination conditions. Hence we study conservative-synchronous and optimistic protocols in this thesis.

Chapter 2 presents problems related to the parallel simulation termination problem and solutions proposed to these problems in the literature. It also presents solutions for the parallel simulation termination problem proposed by other researchers.

1.5.1 Conservative-synchronous Protocols: Chapter 3 presents eight termination algorithms for conservative-synchronous protocols and reports on their performance. The most general of these algorithms applies to unstable conditions and increases the storage

required for termination by an amount proportional to the number of attributes used in the simulation model. This implies that storage requirements for a terminating conservative-synchronous protocol can be competitive with storage requirements for the corresponding terminating optimistic protocol. In addition, conservative-synchronous protocols are simpler to implement than optimistic protocols.

We also show that the cost in terms of execution time of adding termination to the Bounded Lag protocol (Lubachevsky 1989) simulating a closed queuing network is constant for stable termination conditions and is proportional to the number of events executed between synchronization barriers for unstable conditions. These conclusions are confirmed through measurements.

1.5.2 Optimistic Protocols: Chapter 4 presents four termination algorithms for optimistic protocols and reports on the performance of two of these algorithms. An inherent advantage of optimistic protocols is that they need not store in memory any attribute values that can be recomputed after termination is detected in order to compute output measures. This reduces the memory required by the simulation protocol at the expense of increased wall clock time to execute the simulation. We show that the cost of adding termination to optimistic protocols is constant or proportional to the size of the simulation for stable termination conditions, and proportional to the simulation size for unstable termination conditions. Another advantage of optimistic protocols is that they can model all kinds of simulations (e.g., conservative protocols cannot model simulations with zero time events).

CHAPTER 2: RELATED RESEARCH

The parallel simulation termination problem is related to two problems that have been studied in the literature: the termination detection problem (Chandy and Misra 1988, Ch 9) and the global state detection problem (Chandy and Lamport 1985), (Li, et. al. 1987), (Spezialetti and Kearns 1986). A brief overview of these two problems and their relation to the parallel simulation termination problem is given in Sections 2.1, 2.2 and 2.3. Section 2.4 presents some approaches taken in the literature to solve the parallel simulation termination problem.

2.1 Relation to Termination Detection Problem

There are many papers that address the classical termination detection problem; Mattern provides an overview of the literature (Mattern 1987). The termination detection problem differs from the parallel simulation termination problem in four ways. First, in the classic problem, "termination detection" means detecting a stable condition, whereas many interesting simulation termination conditions are unstable. Second, in the classic problem, a termination condition is a conjunct of predicates, where each predicate is evaluated by one thread using only variables private to that thread. However, a simulation termination condition generally is an arbitrary function of simulation attributes, each private to one, unique LP. The two examples given in Chapter 1 illustrate this point. Third, the program is idle when the termination condition is *true*, whereas the simulation termination problem presumes that the simulation is non-terminating. Finally, after the termination time t is determined, simulation programs need to compute output measures using simulation attribute values at time t; in general t lies in the past of each LP. No analog to recovering old attribute values to evaluate output measures exists in the classic termination problem.

2.2 Relation to Global State Detection Problem

The global state detection problem requires threads to record their own states and the states of communication channels so that the set of records form a global system state. Global state detection algorithms are employed for termination and deadlock detection. The Time Warp parallel simulation protocol (Jefferson 1985) uses global state detection methods to calculate global virtual time (GVT).

Being able to determine a global state does not help us solve the parallel simulation termination problem. The first problem is to retrieve attribute values from a time in the simulated past in order to calculate output measures. The second problem is to bring the asynchronous threads to a halt once the termination time has been determined. Global states are useful for detecting when termination has occurred, once a local termination condition has been used by each thread to determine when it can stop. The use of a global condition to initiate termination has not been addressed in the literature.

2.3 Relation to Stability Detection Problem

A problem related to termination detection is stability detection (Chandy and Misra 1988, Ch. 11). The stability detection problem is to identify when a stable property holds for an ongoing computation. Similarly, we detect C for a non-terminating simulation. Detecting stable termination conditions is equivalent to the stability detection problem. However, stability detection algorithms work for any algorithm, while we exploit special properties of simulation programs. In addition, we wish to detect unstable conditions.

2.4 Previous Research on the Parallel Simulation Termination Problem

Conservative-synchronous protocols have been proposed by Lubachevsky (Lubachevsky 1989), Nicol (Nicol 1990), Ayani (Ayani 1992), and Sokol (Sokol et. al. 1988). Lubachevsy's Bounded Lag algorithm, shown in Figure 1, is used to implement

our simulation benchmark and termination algorithms, because the pseudo-code for the algorithm is presented in (Lubachevsky 1989) and is relatively easy to implement. Lubachevsky shows that, for *N* LP's executing the Bounded Lag algorithm, at least $O(N)$ events are processed for every iteration of the outer loop of the algorithm. He presents arguments that the performance of the Bounded Lag algorithm is scalable with respect to the problem and machine size, and that the algorithm has good potential for efficient execution.

The main difference between the Bounded Lag algorithm and Nicol's algorithm lies in the computation of the time window within which events can be executed safely during an iteration of the algorithm. If the curve for the running time of a simulation with a termination condition for varying simulation sizes is flat with the Bounded Lag algorithm, we expect the curve for the running times using Nicol's algorithm to be flat as well. If the curve for the Bounded Lag algorithm has a positive slope, we expect the curve for Nicol's algorithm to have a smaller positive slope. Since the underlying mechanism for both algorithms is the same, however, the results of our experiments for the Bounded Lag protocol should be representative of Nicol's algorithm as well.

Most parallel discrete-event simulators can handle only a single, simple termination condition: each thread comprising the simulation terminates when it has simulated up to a predefined simulation time.

Algorithms to add arbitrary termination conditions to optimistic and conservative-asynchronous simulation protocols have been proposed by Abrams and Richardson (Abrams and Richardson 1991, Richardson 1991). These algorithms are described in Chapter 1.

Lin proposes two termination detection algorithms, *TW1* and *TW2*, for optimistic protocols (Lin 1991). The algorithms, however, require changes to the implementation of the simulation protocol itself. Lin gives a detailed description of Exhaustive Termination

for time warp (Lin 1991). His algorithm follows the time warp implementation outlined by Abrams and Richardson (Abrams and Richardson 1991) in that the underlying simulation is augmented by an extra LP to which the underlying LP's send changes in attribute values required to evaluate C. Lin also gives an alternative implementation of Interval Termination to that given by Abrams (Abrams 1992) in which the process initiating GVT computation requests attribute values required to evaluate C.

Lin also derives analytic formulas for estimating detection delay. However, his analysis does not consider the overhead added to the underlying simulation *before* termination is detected.

In contrast to Lin's algorithms, we propose four termination algorithms for optimistic protocols in Chapter 4 which do not require any changes to be made to the simulation protocol, and implement the algorithm entirely in user space.

A theoretical basis for the parallel simulation termination problem and a framework intended to encompass all possible solutions is proposed by Abrams (Abrams 1992). Two general techniques to solve the parallel simulation termination problem are formally defined and developed into a framework that fits all proposed termination algorithms, including exhaustive termination and interval termination. The first technique, *parallel evaluation*, attempts to minimize the time required to detect termination by evaluating the termination condition at different time steps in parallel. The second technique, *reduced evaluation*, attempts to minimize the added time by evaluating the termination condition fewer times, by guessing time steps at which the termination condition is *true*, possibly at the expense of increasing the number of time steps simulated by the underlying simulation.

CHAPTER 3: CONSERVATIVE-SYNCHRONOUS PROTOCOLS

Section 3.1 presents eight termination algorithms for the Bounded Lag protocol. Section 3.2 describes the benchmarks used for evaluating the performance of these algorithms, gives a prediction for the expected performance of these algorithms, and presents the measured performance of the algorithms.

3.1 Termination Algorithms for the Bounded Lag Conservative-Synchronous Protocol

We assume without loss of generality that each LP has exactly two attributes local to it, one required to evaluate C and the other required to compute output measures, as described in Chapter 1. Output measures are computed by function OMF, whose domain consists of the single local attribute of each LP required to compute output measures, as well as the time t at which C evaluates to true. The result computed by OMF is stored in variable U.

We categorize termination algorithms based on three criteria. The first criterion is whether the evaluation of C is done between barriers 1 and 2 or between barriers 2 and 3 of Figure 1. We do not evaluate C between barrier 3 and barrier 1 because this is when its result, stored in variable SV, is read. The second criterion is whether the evaluation of C is done sequentially (by one LP) or in parallel (by more than one LP). The third criterion is whether the evaluation of C is done using an exhaustive or interval termination algorithm. Based on these criteria, we classify our termination algorithms using the following mnemonics:

- D - evaluation of C is done using Exhaustive Termination with Dissociative Generation.
- DR - evaluation of C is done using Exhaustive Termination with Retrospective Generation.
- I - evaluation of C is done using Interval Termination with Prospective Generation

S - evaluation of C is sequential

P - evaluation of C is done in parallel

12- evaluation of C is done between barrier 1 and barrier 2

23- evaluation of C is done between barrier 2 and barrier 3

Not all combinations are interesting. For example, algorithm IP23 is not considered for the following reason: the computation of C needs to be parallelized only if the computation requires more time than event execution between barrier 1 and barrier 2. In the interval termination algorithm, the only work done between barrier 2 and barrier 3 is the evaluation of `Floor` by LP_1. Evaluating C with more than one LP between barrier 2 and barrier 3 will still be a bottleneck for the simulation. Thus, for interval termination, we consider parallelization of the computation of C only between barrier 1 and barrier 2, in parallel with event execution. We do not study the performance of algorithms DP23 and DP23R because the termination conditions we use are not complex enough to justify parallelization of their computation.

We propose eight algorithms to solve problem P1 of finding a termination time t: IS23, IS12, IP12, DS23, DS12, DS23R, DP23, and DP23R. The first three algorithms are instances of the Interval Termination Algorithm (Section 1.4), and the five algorithms which follow are instances of the Exhaustive Termination Algorithm with Dissociative Generation (Section 1.4).

The eight termination algorithms are described below.

3.1.1 Algorithm IS23

```
U:real;
```

Add after S11:

```
if (i=2) SV:= C(t,A);          /*only LP2 evaluates C */
```

Add after S14:

```
if (i=1) U:= OMF(t, O);        /*compute output measures */
```

Evaluating C using algorithm IS23 imposes a negligible overhead for simple global termination conditions because the evaluation is done sequentially by LP_2 between barrier 2 and barrier 3 in parallel with the assignment to `Floor`, which is done by LP_1.

3.1.2 Algorithm IS12

Add the following LP, denoted LP_c, to the simulation which participates in the synchronization barriers of the LP's comprising the simulation:

```
OLD_A[1..M]:=initial attribute values to evaluate C:real;
OLD_O[1..M]:=initial attribute values to evaluate OMF:real;
U, t1:=0.0:real;
while (not SV){
    synchronize;               /* barrier 1 */
    SV:= C(t1,OLD_A);          /* use attributes stored after */
                               /* previous iteration          */
    synchronize;               /* barrier 2 */
    synchronize;               /* barrier 3 */
}
```

Add after S11:

```
t1:=t;
OLD_A[i]:=A[i];
OLD_O[i]:=O[i];
```

Add after S14:

```
if (i=1) U:= OMF(t1,OLD_O); /*compute output measures */
```

Algorithm IS12 uses an additional LP, denoted LP_c, dedicated to evaluating *C*. The LP's comprising the simulation run in parallel with LP_c. Event execution is performed by the LP's between barrier 1 and barrier 2. After each LP passes barrier 2, it copies into memory the attributes from array A and array O into arrays OLD_A and OLD_O, respectively. In the next iteration, the attribute values of the previous iteration, stored in array OLD_A, are used by LP_c to evaluate *C* between barrier 1 and barrier 2 while the other LP's are computing new attribute values for array A and array O through event execution. If *C* evaluates to *true*, LP_c sets the stop variable SV to *true*. LP_1 then redundantly computes `Floor` between barrier 2 and barrier 3, falls out of the main loop with the other LP's, and computes output measures. In comparison with algorithm IS23, algorithm IS12 will perform an extra iteration of the simulation before termination is detected. Algorithm IS12 imposes a memory overhead for storing attributes in OLD_A and OLD_O, as well as the overhead of using an additional LP to evaluate *C*. However, it may be more advantageous to use algorithm IS12 over algorithm IS23 if the evaluation of *C* is a bottleneck for algorithm IS23, because the Bounded Lag protocol takes more time on average to perform event execution between barrier 1 and 2 than it takes to compute the `Floor` between barrier 2 and 3.

3.1.3 Algorithm IP12

Algorithm IP12 differs from algorithm IS12 only in that the evaluation of *C* is itself done in parallel by more than one LP . For example, a combining tree could be used if the operators used in evaluating *C* form an abelian group (Lakshman and Wei 1991); in this case algorithm IS12 requires O(*N*) time, while algorithm IP12 requires O(log *N*) time to

evaluate C. For each iteration of the outer loop of the Bounded Lag algorithm, there are $O(N)/N = O(1)$ events executed by each LP. Parallel computation of C is warranted if the time required to evaluate C is a few times larger than the average time that an LP spends to execute one event. However, the overhead involved in parallelizing the computation of C may negate the benefit of parallel evaluation of C.

3.1.4 Algorithm DS23

This algorithm requires the addition of data structure `Q`, which is shared by all LP's. `Q` is a queue which contains a list of timestamped events which store the attributes required to evaluate C and the attributes required to evaluate OMF, the function which computes simulation output measures. Events are inserted into `Q` by the LP's in timestamp order.

```
QNODE=record{
     i:integer;  /* stores LP's id */
     A:real;     /* stores A[i]    */
     O:real;     /* stores O[i]    */
     t:real;     /* timestamp      */
    }

q:QNODE;
Q:queue of QNODE;
m:integer;
U:real;
```

<u>Add after S7:</u>

```
Lock Q;
q:=create QNODE(i, A[i], O[i], t);
insert q in Q in ascending order of field t;
Unlock Q;
```

Add after S11:

```
if(i=2){
     repeat
        for each node q in Q{
          m:=q.i;
          A[m]:=q.A;
          O[m]:=q.O;
          t:=q.t;
          SV:=C(t,A);
          delete q;
        }
     until (SV=true or Q is empty);
}
```

Add after S14:

```
if (i=1) U:= OMF(t,O)    /* compute output measures */
```

Algorithms IS23, IS12, and IP12 previously discussed cannot detect unstable termination conditions. To do this a simulator must evaluate C after every event execution which changes attributes in array `A`. The implementation of algorithm DS23 maintains a history of simulation attribute values of array `A` and array `O` which changed in value in the last iteration between barrier 1 and barrier 2. Algorithm DS23 does this by inserting events containing attributes in array A and array `O` into `Q` in timestamp order as they are executed. Between barriers 2 and 3, LP_2 evaluates C for each event in `Q` in timestamp order using the stored attribute values in array `A`. The algorithm terminates at the first evaluation of C which yields *true* and output measures are then calculated by evaluating OMF using the stored attribute values of array `O` for that time, which are retrieved from Q. In Section 3.3.2, we show that this algorithm imposes an overhead proportional to the number of events executed during the entire simulation.

3.1.5 Algorithm DS12

```
QNODE=record{
     i:integer;  /* stores LP's id */
     A:real;     /* stores A[i]    */
     O:real;     /* stores O[i]    */
     t:real;     /* timestamp      */
    }

Qodd[M]:queue of QNODE;
Qeven[M]:queue of QNODE
QSORTED:queue of QNODE;
iter_num:=0:integer;
U:real;
q:QNODE;         /* local to each LP */
```

<u>Add after S7:</u>

```
q:=create QNODE(i, A[i], O[i], t);
if (iter_num mod 2 = 0)
   insert q in Qeven[i];
else
   insert q in Qodd[i];
```

<u>Add after S14</u>:

```
if (i=1) U:= OMF(t,O)    /* compute output measures */
```

Add the following LP to the simulation which participates in the synchronization barriers of the LP's comprising the simulation:

```
while (not SV){
   synchronize;               /* barrier 1 */
   if (iter_num mod 2 = 0){
      for i=1 to M do
         for each node q in Qodd[i]{
          insert q in Qsorted in ascending order of q.t;
          delete q from Qodd[i];
         }
   }
   else{
      for i=1 to M do
         for each node q in Qeven[i]{
          insert q in Qsorted in ascending order of q.t;
          delete q from Qeven[i];
         }
   }
  repeat
        for each node q in Qsorted{
         m:=q.i;
         A[m]:=q.A;
         O[m]:=q.O;
         t:=q.t;
         SV:=C(t,A);
         delete q;
        }
   until (SV=true or Q is empty);
   synchronize;               /* barrier 2 */
   iter_num=iter_num+1;
   synchronize;               /* barrier 3 */
}
```

Algorithm DS12 inserts events containing attributes into local queues Qodd or Qeven in timestamp order as they are executed. Attributes are put in queue Qodd for every odd numbered iteration of the outer loop of the algorithm. There is an additional operating system thread, which, on every even numbered iteration, takes all the events from all the Qodd queues, puts them on another queue Qshared in timestamp order, and then evaluates *C*. The algorithm terminates at the first evaluation of *C* which yields *true* and output measures are then calculated by evaluating OMF using the stored attribute values of array O for that time, which are retrieved from Qshared.

3.1.6 Algorithm DS23R

```
QNODE=record{
     i:integer;  /* stores LP's id */
     A:real;     /* stores A[i]    */
     t:real;     /* timestamp      */
    }
q:QNODE;         /* local to each LP*/
Q:queue of QNODE;
m:integer;
OLD_A[1..M], OLD_O[1..M], U:real;
```

Add after S3:

```
OLD_A[i]:=A[i];
OLD_O[i]:=O[i];
```

Add after S7:

```
Lock Q;
q:=create QNODE(i, A[i], t);
insert q in Q in ascending order of field t;
Unlock Q;
```

Add after S11:

```
if(i=2){
    repeat
       for each q in Q{
        m:=q.i;
        t:=q.t;
        A[m]:=q.A;
        SV:=evaluate C(t,A);
        delete q;
       }
    until (SV=true or Q is empty);
}
```

Add after S14:

```
TC:=t;      /*store time t at which termination is detected */
A[i]:=OLD_A[i];          /* rollback to state at barrier 1 */
O[i]:=OLD_O[i];
S1:=while (t < TC);   /*change loop termination condition */
goto S1, and re-execute until time TC;
if(i=1) U:=OMF(TC,O)  /*compute output measures*/
```

In algorithm DS23, all the attributes in array A as well as those in array O are stored in Q. Algorithm DS23R reduces the storage required by Algorithm DS23 by exploiting the fact that the attribute values needed to calculate output measures may in general be different from those required to evaluate *C*. Algorithm DS23R stores only the attributes in array A in Q. The values of attributes in array A and array O are maintained at barrier 1. When *C* becomes true, the time t at which termination was detected is stored in TC, the simulation rolls back to barrier 1 by changing all the attribute values in array A and array O to those stored in arrays OLD_A and OLD_O, and re-executes until time TC. The attributes required to calculate output measures are now available in array O. In comparison to algorithm DS23, algorithm DS23R requires less memory at the expense of having to perform some

extra work after termination is detected.

3.1.7 Algorithm DP23

Algorithm DP23 is a modification of algorithm DS23 in which the evaluation of C for each event occurrence in Q is performed using more than one LP, possibly using a combining tree.

3.1.8 Algorithm DP23R

Algorithm DP23R is a modification of algorithm DS23R in which the evaluation of C for each event occurrence in Q is performed using more than one LP. The roll-back mechanism reduces the memory required to store attributes by not storing attributes required to calculate output measures, as compared to algorithm DP23.

3.2 Conservative-Synchronous Termination Algorithm Performance

3.2.1 Benchmarks Used

The first benchmark we use to measure the performance of our termination algorithms is a simulation of an N X N toroidal network for any non-preemptive service discipline (Figure 3, Lubachevsky 1989). Figure 5 illustrates a 3 x 3 network. A node of the network is a server with an attached input queue of infinite capacity. A server removes an event, also called a job, from its input queue and starts service. The service durations are bounded from below by D seconds, $D > 0$. When the service is complete, the job is either deleted from the system, or dropped in the input queue of one of the four neighbouring servers. Each server of the toroidal network is modeled by an LP. Initially, servers are idle and have an equal number of jobs in their queues.

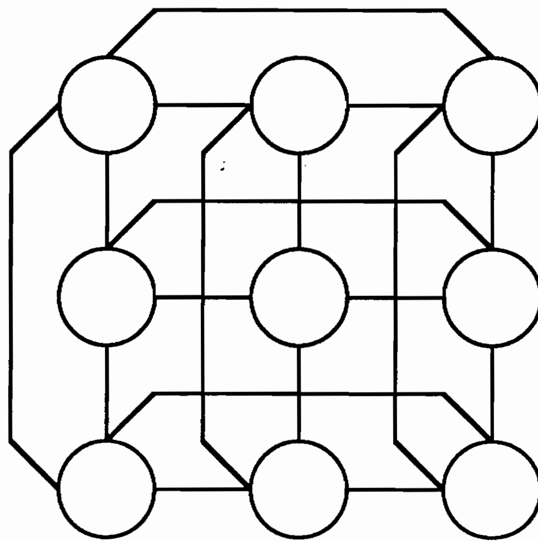


Figure 5: A 3 x 3 torus queuing network. Each circle represents a server of the queuing network. An arc connects two adjacent servers.

Experiment Design: The term *simulation size* is defined as the value of *N* for the torus network benchmark. The number of processors used in a simulation run is denoted by *P*. The initial number of jobs in each server's input queue is chosen to be 200, after verifying by experimentation that this is a large enough number to preclude the possibility of a server running out of jobs to execute in its input queue. We verified this by running the simulation for the extreme values of *N* (*N*=3 and *N*=20) for *P*=1, 2, 4, and 8 for termination algorithm DS23 with termination condition T1, defined below. The number of jobs in the input queue of any server never falls below 100 at any point during the simulation; therefore 200 is not an important constant and any number greater than 100 has no effect on the simulation running times.

We use an exponentially distributed service time with a mean, *M*, of 200 simulation time units and a constant, *D*, of 100 time units added to it. We ran three experiments for the extreme values of *N*, using the combinations of *M, D*=500, 100, *M, D*=500, 250, and *M*, D=500, 500 for *P*=1, 2, 4, and 8. The variance in the simulation running times was less than 1 percent, so we conclude that the simulation running time is not sensitive to the choice of *M* and *D*.

The termination conditions used in the experiments reported on are the following:

T0: Algorithm IS23 & IS12: "stop when the total number of jobs serviced by all LP's *exceeds* 100,000."

T1: Algorithm DS12 and DS23: "stop when the total number of jobs serviced by all LP's *equals* 100,000."

T2: Algorithm IS23 & IS12: "stop when the local virtual time of each LP *exceeds* 100,000 simulation time units."

T3: Algorithm DS12 and DS23: "stop when the local virtual time of any LP *first exceeds* 100,000 simulation time units."

3.2.2 Predicted Performance

Termination conditions are also classified according to their dependence on the simulation size. For example, consider termination condition T0. It would take less simulation time for T0 to become true for a larger network (i.e., larger N) than a smaller one simply because the larger network has a larger number of servers servicing jobs in parallel. On the other hand, termination condition T2 would cause a simulation to stop at a simulation time which is proportional to the simulation size because all LP's would have to execute until simulation time 100,000.

Lubachevsky shows that at least $O(M)$ events are executed for every iteration of the algorithm (Lubachevsky 1989). The average cost of the execution of an event is shown to be $O(1)$. These facts are of interest as the algorithms are potentially scalable with the simulation and machine size. We make use of these facts to analyze the performance of our algorithms.

Algorithm IS23 & IS12 with T0: For this algorithm, each of the $O(M)$ events executed per iteration takes O(1) time to execute. For T0, the total number of events executed up to termination by all LP's is constant. Thus, for fixed P, the total running time of the simulation is O(1) and independent of M. Thus we should have a flat curve for the simulation running time as M is varied.

Algorithm DS23 with T1: For this algorithm, each of the $O(M)$ events executed per iteration takes O(1) time to execute and $O(M)$ time to be inserted onto a shared queue Q, as the length of Q is $O(M)$. For T1, the total number of events executed up to termination by all LP's is constant. Since executing each event takes $O(M)$ time for fixed P, the total running time of the simulation is $O(M)$ and we should get a linear curve for the running time of the simulation as M is varied.

Algorithm DS12 with T1: For this algorithm, each of the O(M) events executed per iteration takes O(1) time to execute. For T1, the total number of events executed up to termination by all LP's is constant. The total running time taken up by event execution is O(1). However, for each iteration of the algorithm, there is an O(M^2) cost to sort the events in timestamp order. The number of sorts performed is inversely proportional to M for fixed P. So the total running time taken for sorting is O(M). Thus, the total running time of the simulation is O(1) + O(M) = O(M) and we should get a linear curve for the running time of the simulation as M is varied.

Algorithm IS23 & IS12 with T2: For this algorithm, each of the O(M) events executed per iteration takes O(1) time to execute. For T2, the total number of events executed up to termination by all LP's is O(M), as all LP's simulate up to the same simulation time. Thus, for fixed P, the total running time of the simulation is O(M) and we should get a linear curve for the simulation running time as M is varied.

Algorithm DS23 with T3: For this algorithm, each of the O(M) events executed per iteration takes O(1) time to execute and O(M) time to be inserted onto a shared queue Q, as the length of Q is O(M). For T3, the total number of events executed up to termination by all LP's is O(M). Since executing each event takes O(M) time for fixed P, the total running time of the simulation is O(M^2) and we should get a quadratic curve for the running time of the simulation as M is varied.

Algorithm DS12 with T3: For this algorithm, each of the O(M) events executed per iteration takes O(1) time to execute. For T3, the total number of events executed upto termination by all LP's is O(M). The total running time taken up by event execution is O(M). However, for each iteration of the algorithm, there is an O(M^2) cost to sort the

events in timestamp order. The number of sorts performed is independent of M for fixed P. So the total running time taken for sorting is $O(M^2)$. Thus, the total running time of the simulation is $O(M) + O(M^2) = O(M^2)$ and we should get a quadratic curve for the running time of the simulation as M is varied.

3.2.3 Measured Performance

We report on simulation experiments using a Sequent S81 shared-memory multiprocessor with 10 80386 processors running Dynix V3.0.18. We use AT&T C++ 1.2.1 under the Presto thread package version 0.4 (Bershad, Lazowska and Levy 1987). We use Presto because it provides lightweight threads and several synchronization primitives for locks. We have the option of preemptive or non-preemptive thread scheduling with a user specified quantum size in the preemptive case. We report results using both these options.

We define the *efficiency* of an algorithm as follows:

efficiency = $T(N, 1) / P * T(N, P)$, where
$T(N, 1)$ - wall clock time to execute simulation for simulation size N and 1 processor.
$T(N, P)$ - wall clock time to execute simlation for simulation size N and P processors.

We use N=3, 4, 6, 8, 10, 16 and 20 for the six combinations IS23+T0, DS23+T1, IS23+T2, DS23+T3, IS12+T0 and IS12+T2. We vary P in powers of two from 1 to 8. Each server is implemented by a thread executing the algorithm with its own copy of all local data. Barrier synchronization is effected using a master-slave mechanism based on examples distributed with Presto 0.4. The value of the bounded lag B is chosen so that each node examines only its nearest neighbors to calculate the earliest time another event could affect its history (i.e., computation of `ALPHA[i]` in S2 of Figure 1). Because we use a shared memory machine, it is easy for LP's to examine values of variables local to other LP's. Variables such as the shared queue `Q` are locked by LP's before being modified.

The data was collected while the simulation processes were the only user processes on the system. The difference in runtimes is less than 1 percent for the extreme values of *N* (*N*=3 and *N*=20), for *P*=1, 2, 4 and 8. The data value used to plot each point on the graphs is the mean of the run times of three experiments. The simulation was run both preemptively (with the quantum size varied between 100 msec and 600 msec) and non-preemptively, but this had no effect on the running times measured, perhaps because no event required more than 100 msecs of wall clock time to execute. For *P*=1, the simulation was run nonpreemptively without any locking of shared data structures such as `Q` because concurrent access is physically impossible with only one processor. The one processor case is, however, not a sequential simulation because there are multiple threads running on the single processor, which imposes an execution time overhead.

The performance of the torus network without a termination algorithm, that is just testing for the `Floor` to exceed 100,000 in Figure 1, yields a running time slightly smaller than the values reported for IS23+T2. The graph for the simulation running time for this case is shown in Figure 6. The efficiency of the torus network simulation without a termination condition is shown in Figure 6a.

Figure 7 shows the running times for IS23+T0 and DS12+T1. We note that, as predicted, the curve for algorithm IS23+T0 is essentially flat, and the curve for algorithm DS12+T1 is linear. There is an appreciable improvement in performance as *P* is increased from 1 to 2, and from 2 to 4. There is not much improvement if we use 8 processors instead of 4. We conjecture that this is due to the low density of events being processed between barrier 1 and barrier 2, compared to the contention introduced by the barrier synchronization mechanism. The graph for the efficiency of both these algorithms is similar to Figure 6a.

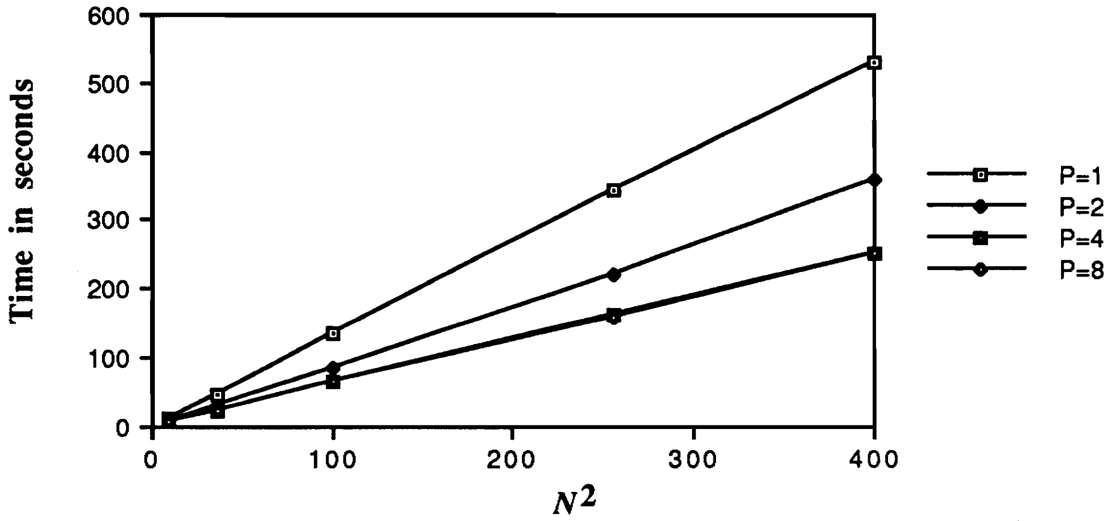


Figure 6: Wall Clock Time required to complete simulation for N^2 LP's for 100,000 simulation time units with no termination condition for the conservative-synchronous protocol with the torus network benchmark.

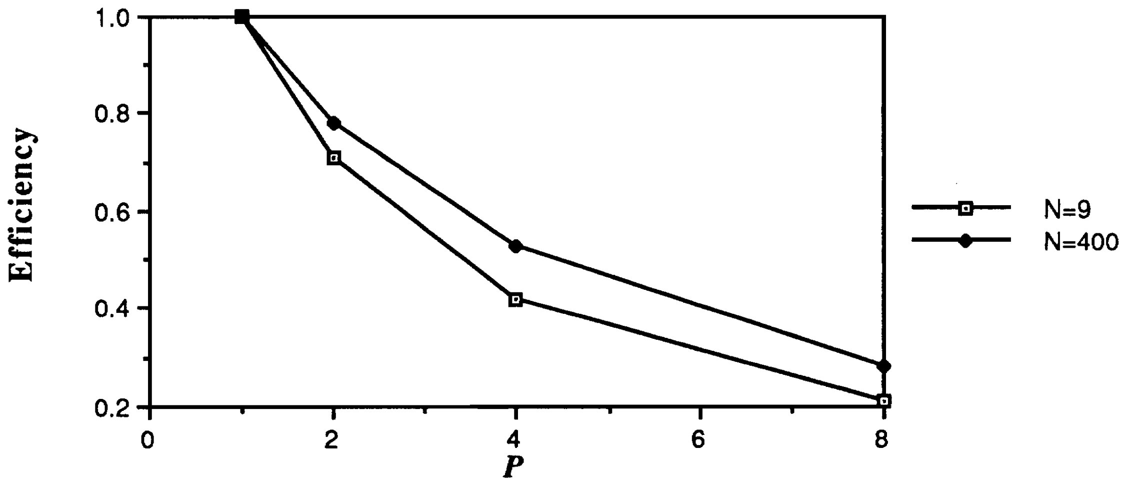


Figure 6a: Efficiency of the torus network simulation for the conservative synchronous-protocol without a termination condition for extreme values of N.

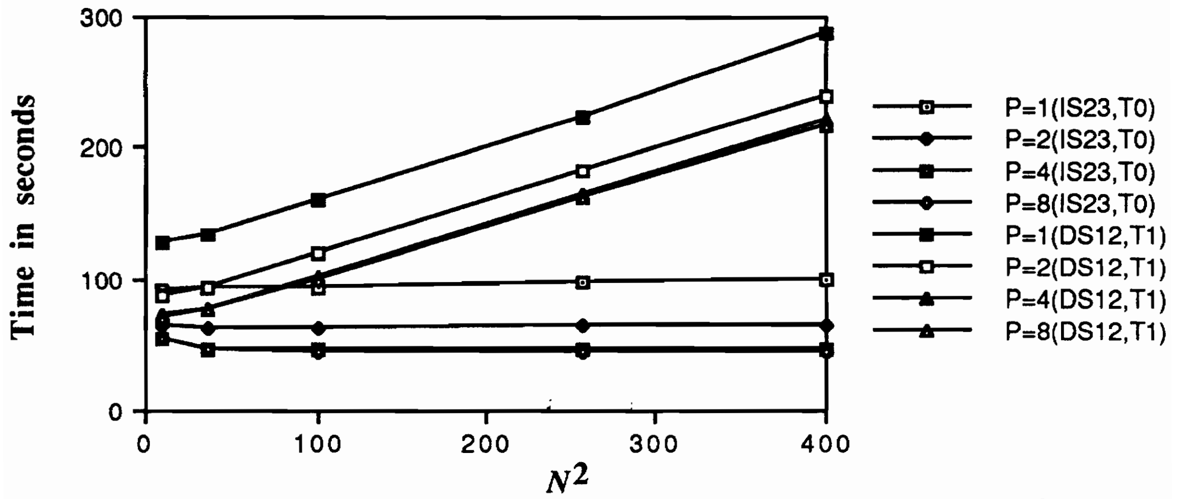


Figure 7: Wall Clock Time required to complete simulation for N^2 threads with IS23+T0 and DS12+T1 for the conservative-synchronous protocol.

Figure 8 shows the running times for IS23 + T2 and DS12 + T3. As predicted, the graph for algorithm IS23+T2 is linear and the graph for algorithm DS12+T3 is quadratic. As in the previous graph, we get performance improvements as we increase *P* from 1 to 2 and from 2 to 4, but not with *P*=8. The graph for the efficiency of both these algorithms is similar to Figure 6a.

The performance of algorithm IS12+T0 is not shown because the curves are essentially the same as that of algorithm IS23+T0. Also, the performance of IS12+T2 is essentially the same as that of IS23+T2. The performance of DS23+T1 and DS23+T2 isslightly worse than that of DS12+T1 and DS12+T3, respectively. We did not study the performance of DS12+T0 because T0 is a stable termination condition and using algorithm DS12 to detect T0 would be inefficient in this case. We note that the results of our experiments are in accordance with the predicted values.

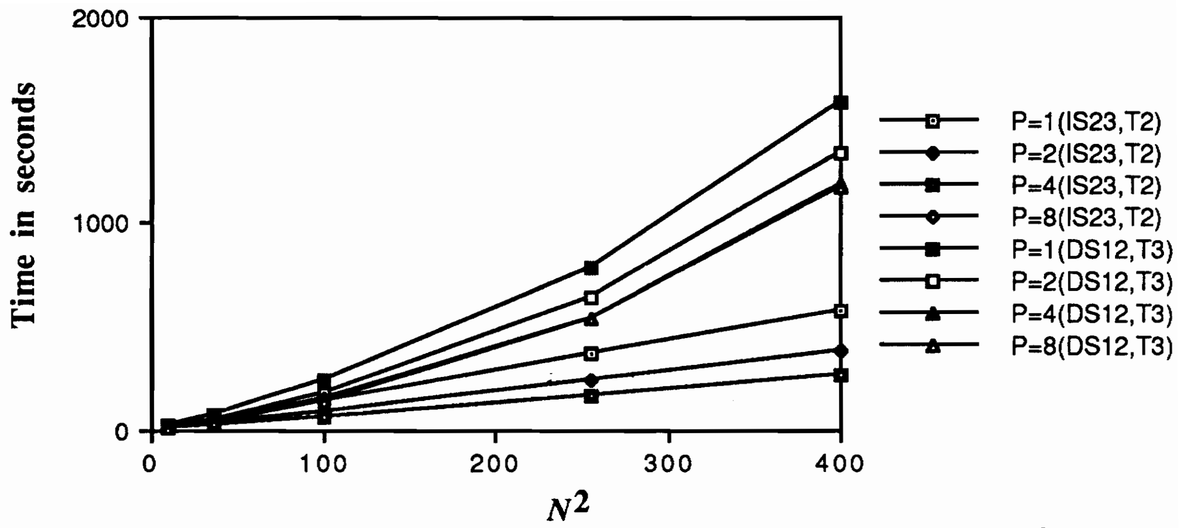


Figure 8: Wall Clock Time required to complete simulation for N^2 threads with IS23+T2 and DS12+T3 for the conservative-synchronous protocol.

CHAPTER 4: OPTIMISTIC PROTOCOLS

4.1 Termination Algorithms for Optimistic Protocols

There are two ways to add a termination algorithm to an optimistic protocol. The algorithm can be incorporated in the protocol itself (as was done in Chapter 3 for the conservative-synchronous protocol), or it can be implemented as a user process and superimposed on the simulation. We take the second route because Time Warp is much more complex than the Bounded Lag algorithm of Figure 1, and we do not have access to documented source code for Time Warp. However, the first case is more efficient, so our results are an upper bound on run times that could be obtained by modifying Time Warp and using the same benchmarks.

Termination detection is done optimistically by a user process in user space. Our algorithms are easier to implement and evaluate than Lin's algorithms because they require no changes to the protocol itself.

We augment the parallel simulation with an LP designated the *termination detector* (TD). The TD runs in parallel with the other LP's comprising the simulation. At the start of the simulation, the TD contains values of all attributes required to evaluate C. When the simulation starts, each LP which contains attributes local to it required in the evaluation of C periodically sends an UPDATE_EVENT message to the TD. The UPDATE_EVENT message contains the local attributes of the sending LP required to evaluate C and has a timestamp equal to the LVT of the LP. The frequency with which UPDATE_EVENT messages are sent to the TD is determined by the simulation size, the nature of the termination condition, and termination algorithm (e.g., Interval vs. Exhaustive from Chapter 1). The TD maintains in its state descriptor a copy of all simulation attributes required to evaluate C. Whenever the TD receives an UPDATE_EVENT message, it updates its copy of all simulation attributes reported by the UPDATE_EVENT message and

then evaluates *C*. The timestamp on UPDATE_EVENT messages provides the time argument to *C*. The TD estimates an attribute value based on the last UPDATE_EVENT message that it received that contained a value for that attribute. If the TD never receives an UPDATE_EVENT message for some attribute, it uses the initial value of that attribute as an estimate of its value. Because the UPDATE_EVENT message could conceivably be rolled back, the evaluation of *C* itself is being done optimistically. If *C* evaluates to *false*, the TD does not perform any action. If, however, *C* evaluates to *true*, the TD will send a TERM_EVENT message to all the LP's and to itself with the timestamp of the UPDATE_EVENT message it just received.

Whenever an LP, including the TD, receives a TERM_EVENT message, it performs the following actions. If the timestamp of the TERM_EVENT message is less than the LVT of the LP, the LP will roll back to a simulation time equal to the timestamp of the TERM_EVENT message, flush the remaining messages in its input queue and advance its LVT to positive infinity. Otherwise, the LP does not need to be rolled back, and it will process the TERM_EVENT message and advance its LVT to positive infinity. The simulation ends when all LP's, including the TD, have no messages left to execute and GVT reaches positive infinity.

It is conceivable that the TD itself might need to be rolled back after it has scheduled TERM_EVENT messages for some or all LP's. This could happen if the TD receives an UPDATE_EVENT message with a timestamp *less* than the timestamp of the message whose execution caused the evaluation of *C* to yield *true*. In this event, out-of-order TERM_EVENT messages sent by the TD cause the receiving LP's to advance their LVT's to positive infinity. This is undone by rolling back all LP's which executed the TERM_EVENT message to a time smaller than the timestamp of the TERM_EVENT and the simulation then proceeds as normal. Consequently, we could never have a simulation terminating as a result of optimistically scheduled TERM_EVENT messages which are

rolled back in the future, because GVT can never reach positive infinity while unexecuted messages exist in the input queue of any LP.

Termination algorithms are categorized by two letter mnemonics, such as ES, using the mnemonics defined in Section 3.1.

4.1.1 Algorithm IS

Algorithm IS implements the Interval Termination Algorithm of Chapter 1 and detects stable termination conditions. It employs a subset of LP's to send an UPDATE_EVENT message to the TD *periodically* with a user specified frequency. Whenever an LP sends an UPDATE_EVENT message, it includes the value of any attribute local to the LP required to evaluate C. The frequency with which UPDATE_EVENT messages are sent depends on the simulation size. Too high a frequency will overwhelm the TD with UPDATE_EVENT messages, and message execution at the TD becomes a bottleneck for the simulation. Too low a frequency is likely to cause the TD to detect termination at a simulation time significantly greater than the first simulation time instant that evaluation of C would have yielded *true*. With this tradeoff in mind, we choose an UPDATE_EVENT message frequency such that the arrival rate of messages at the TD is independent of the simulation size and approximately equal to the arrival rate of messages scheduled for the LP's. This is done by reducing the frequency with which UPDATE_EVENT messages are sent by individual LP's as the simulation size increases and the number of LP's sending UPDATE_EVENT messages increases.

4.1.2 Algorithm ES

Algorithm ES implements the Exhaustive Termination Algorithm of Chapter 1 and detects unstable as well as stable termination conditions. Each time the LVT of an LP advances and any attributes required to evaluate C were modified since the LVT last

advanced, the LP sends an UPDATE_EVENT message to the TD containing the values of all its modified local attributes. This is necessary to be able to detect unstable termination conditions. This algorithm is more likely to create a bottleneck at the TD than algorithm IS due to the large increase in the arrival rate of messages at the TD as the simulation size increases.

4.1.3 Algorithm IP

Algorithm IP is a variant of Algorithm IS in which the only diference is that the evaluation of C is done in parallel.

4.1.4 Algorithm EP

Algorithm EP is a variant of Algorithm ES in which the only diference is that the evaluation of C is done in parallel.

4.2 Optimistic Termination Algorithms Performance

4.2.1 Experiment Design

This section evaluates the performance of algorithms IS and ES with respect to two benchmarks: The first benchmark is the simulation of an N X N torus network described in Chapter 3. The second is a Colliding Pucks simulation (Jefferson and Hontalas, et. al. 1989). Colliding Pucks is a parallel discrete-event simulation of two-dimensional *pucks* moving on a flat surface, colliding with each other and with stationary *cushions* on the border of the simulation space. The number of pucks in the simulation is denoted by K. The flat surface is divided into equal sized *sectors* which form a grid and regulate the movement and interactions of pucks within their boundaries. Sectors are numbered row wise in ascending order starting from zero. Each sector, puck, and cushion comprising the simulation is modeled by an LP. The initial velocities and positions of the pucks are

chosen using a pseudo-random function. The implementation of Colliding Pucks employs simple kinematic and dynamic principles such as conservation of momentum and energy. Since there is total conservation of energy and no friction, the pucks remain in constant motion.

The term *simulation size* is defined as the value of N for the torus network benchmark and as the value of K for the Colliding Pucks benchmark. The number of processors used in a simulation run is denoted by P.

We use seven termination conditions for our experiments. Termination conditions T0, T1, T2 and T3, described in Chapter 3, are used with the torus network benchmark. Termination conditions P0, P1, and P2, described below, are used with the Colliding Pucks benchmark.

P0 (Algorithm IS): "stop when the total number of collisions in all even numbered sectors is *greater than* 500."

P1 (Algorithm ES): "stop when the total number of collisions in all even numbered sectors is *equal to* 500."

P2 (Algorithm ES): "stop when there are *greater than or equal to* 0.75*K pucks in all even numbered sectors."

4.2.2 Predicted Performance

Algorithm IS with T0: Algorithm IS is an interval termination algorithm, and UPDATE_EVENT messages are sent to the TD *periodically*. Each LP sends one UPDATE_EVENT message to the TD each time it executes $N^2/10$ messages. The value $N^2/10$ is selected to ensure that the arrival rate of messages at the TD is independent of N and that termination is detected soon after evaluation of C would first yield *true*.

As N is increased, the simulation time at which termination is detected decreases because the number of servers executing jobs increases. Therefore, we expect a flat curve

for the simulation running time as a function of N for a fixed P. Also, for a fixed N, we expect a decrease in the running time as P is increased.

Algorithm ES with T1: Algorithm ES is exhaustive, and each LP must send one UPDATE_EVENT message to the TD after *every* job it executes so that the TD can detect the *earliest* simulation time at which the 100,000th job in the simulation is executed. Therefore the arrival rate of messages at the TD grows linearly with the number of servers, N^2. This creates a bottleneck at the TD and should increase the wall clock time required for the simulation to terminate as compared to the time taken by an interval termination algorithm. As was the case with termination condition T0, the simulation time at which T1 holds decreases as N increases. Therefore, as N increases, each server sends the TD fewer messages, and this decreases the wall clock time at which the simulation terminates. Of the previous two effects, the tendency of the TD to be a bottleneck is dominant, and we expect a linear curve with a small positive slope for the running time of the simulation as N is increased for a fixed P. As before, for a fixed N, we also expect a reduction in the running time as P is increased.

Algorithm IS with T2: Algorithm IS is an interval termination algorithm, and the simulation time at which T2 holds is independent of N. We only need to have *one* LP send UPDATE_EVENT messages to the TD *every* time its LVT changes as LP's synchronize periodically for every GVT computation. This ensures that the arrival rate of messages at the TD is independent of N and thus message execution at the TD never becomes a bottleneck. It is however conceivable that the single LP we choose to send UPDATE_EVENT messages to the TD may occasionally pick a very large service time from the exponential distribution and schedule an event far in the future. This may degrade performance because the LP could repeatedly execute the event far in the future, send an

UPDATE_EVENT message to the TD, and then get rolled back every time an event with a timestamp in its logical past arrives. As N is increased for a fixed P, we expect the running time to increase linearly and thus get a curve with a small positive slope. Again, for a fixed N, we expect a reduction in the running time as P is increased.

Algorithm ES with T3: Algorithm ES is exhaustive, and again the simulation time at which T3 holds is independent of N. To be able to detect T3, *every* LP sends an UPDATE_EVENT message to the TD *every* time its LVT changes because we need to detect the *first* time that the LVT of *any* LP exceeds 100,000 simulation time units. As was the case in algorithm ES with T1, message execution at the TD is a bottleneck. As N is increased for a fixed P, we expect the running time to increase either linearly with a large slope or quadratically. As before, for a fixed N, we expect a reduction in the running time as P is increased.

Algorithm IS with P0: Algorithm IS is an interval termination algorithm, and the LP's modeling the pucks send UPDATE_EVENT messages to the TD *periodically*. Each LP modeling a puck sends an UPDATE_EVENT message to the TD after every 10 collisions. Since both pucks involved in a collision report a collision to the TD, the TD has to divide its collision count by a factor of 2 to get the correct collision count. As K increases, the simulation time at which termination is detected decreases because the number of collisions per unit simulation time is proportional to K. Therefore, we expect to have a flat curve for the simulation running time as a function of K for a fixed P. Also, for a fixed K, we expect a decrease in the running time as P is increased.

Algorithm ES with P1: Algorithm ES is exhaustive, and the LP's modeling each puck sends an UPDATE_EVENT messages to the TD after *each* collision. As in algorithm IS

with P0, the simulation time at which termination is detected decreases as K increases. However, the arrival rate of messages at the TD might be high enough for it to become a bottleneck. Therefore, we expect to have either a flat curve or a curve with a small positive slope for the simulation running time as a function of K for a fixed P. Again, for a fixed K, we expect a decrease in the running time as P is increased.

Algorithm ES with P2: Algorithm ES is exhaustive, and the LP's modeling even numbered sectors send UPDATE_EVENT messages to the TD *every* time a puck enters or leaves an even numbered sector. The TD accordingly increments or decrements its count of pucks in even numbered sectors. The simulation time at which termination is detected is independent of K. Therefore, we expect to have a curve with a positive slope for the simulation running time as a function of K for a fixed P. For a fixed K, we would expect a decrease in the running time as P is increased. However, the running times are sensitive to the number of pucks in the even numbered sectors, $0.75*K$, used as the termination condition.

4.2.3 Measured Performance

Our simulations were implemented on a BBN Butterfly with 84 processors using JPL's Time Warp Operating System (TWOS) Version 2.7.1 (Jefferson 1987). We use N = 3, 4, 6, 8, 10, 16 and 20 for the four combinations IS+T0, ES+T1, IS+T2 and ES+T3 and K = 2, 4, 8, 16, 32, 64 and 128 for IS+P0, ES+P1 and ES+P2. The values of P used for the parallel simulation is varied from 1 to 64 through powers of 2. We also ran the simulation on JPL's TWSIM sequential simulator.

The simulations were statically load balanced for all runs using the TWOS load balancer. The load balancer calculates the assignment of LP's to physical processors that minimizes overall simulation running time. For the torus network benchmark, load

balancing is done in the following manner for each different value of N. The simulation is first run on a sequential simulator. This generates an output file containing statistics of the relative work done by each LP comprising the simulation. The TWOS load balancer then uses these files to generate load balanced configurations for all values of P. Load balancing for the Colliding Pucks benchmark was done in a similar manner.

TWOS allows a user to specify a simulation time at which all LP's should be terminated. Using a termination time of 100,000 (equivalent to T2), the torus benchmark achieves a 2-3% smaller value for running time than the case of IS+T2, discussed below. The graph of the simulation running times is shown in Figure 9. The graph for the efficiency of the simulation is shown in Figure 9a. The graph of the simulation running times for the colliding pucks simulation with the same termination time is shown in Figure 10. The graph for the efficiency of the colliding pucks simulation is shown in Figure 10a.

Algorithm IS with T0 (Figure 11): The curves have a small positive slope, which we conjecture is due to an increase in message passing overhead as N increases. The parallel simulation running time decreases as P is increased from 1 to 32, but it increases with 64 processors. An LP is in *saturation* when the mean interarrival time of messages to the LP is equal to the mean time required to process each message. We conjecture that with 64 processors, the TD is close to saturation, when it receives messages at a rate which is almost equal to its maximum service rate. The parallel simulation requires more time than the sequential simulation when fewer than 4 processors are used. The running times are, however, sensitive to the rate with which UPDATE_EVENT messages are sent by the LP's, which in this case is $N^2/10$. However, sending UPDATE_EVENT messages with this rate took less wall clock time to terminate the simulation than a rate of $N^2/5$ or $N^2/20$. Sending UPDATE_EVENT messages with a rate of $N^2/5$ took approximately 5 percent more time to run for K=2 and P=1, 2, 4, 8, 16 and 32, and 10 percent more time to run for

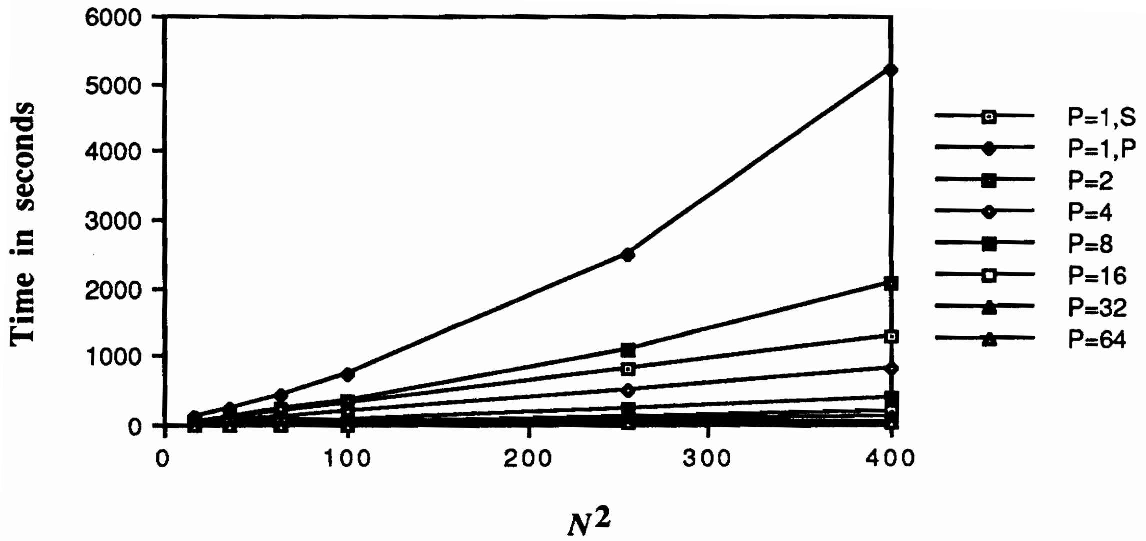


Figure 9: Wall Clock Time required to complete simulation for N^2 LP's for 100,000 simulation time units with no termination condition for the optimistic protocol with the torus network benchmark.

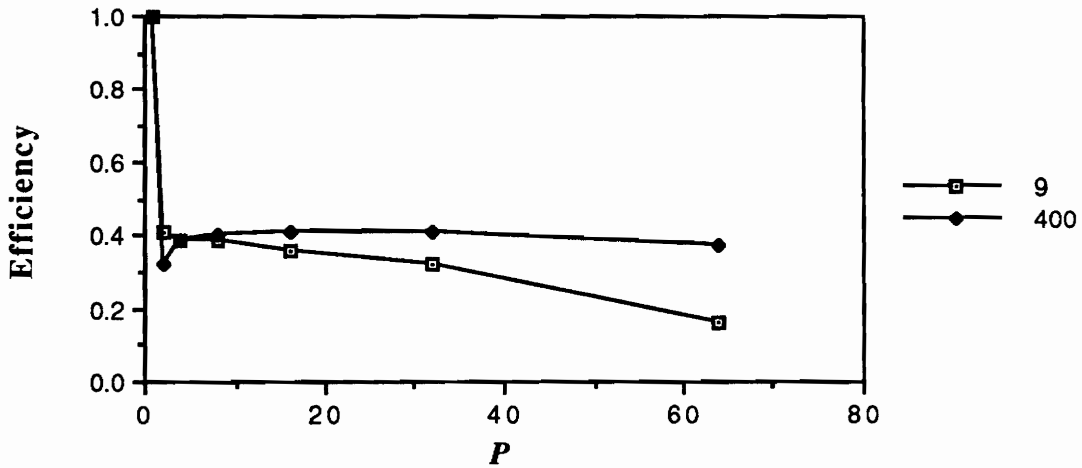


Figure 9a: Efficiency of the torus network simulation for the optimistic protocol for extreme values of N.

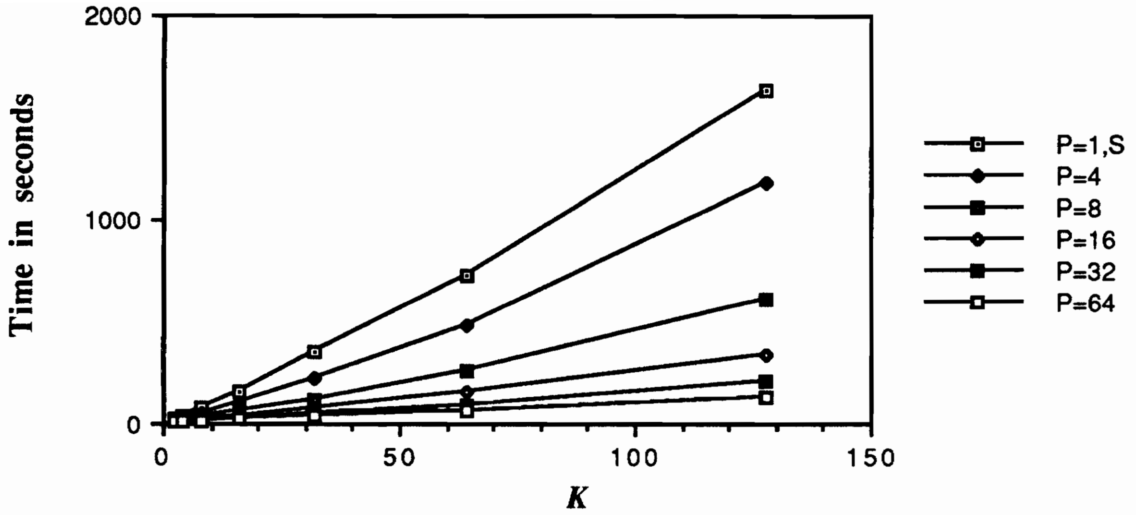


Figure 10: Wall Clock Time required to complete simulation for N^2 LP's for 100,000 simulation time units with no termination condition for the optimistic protocol with the colliding pucks benchmark.

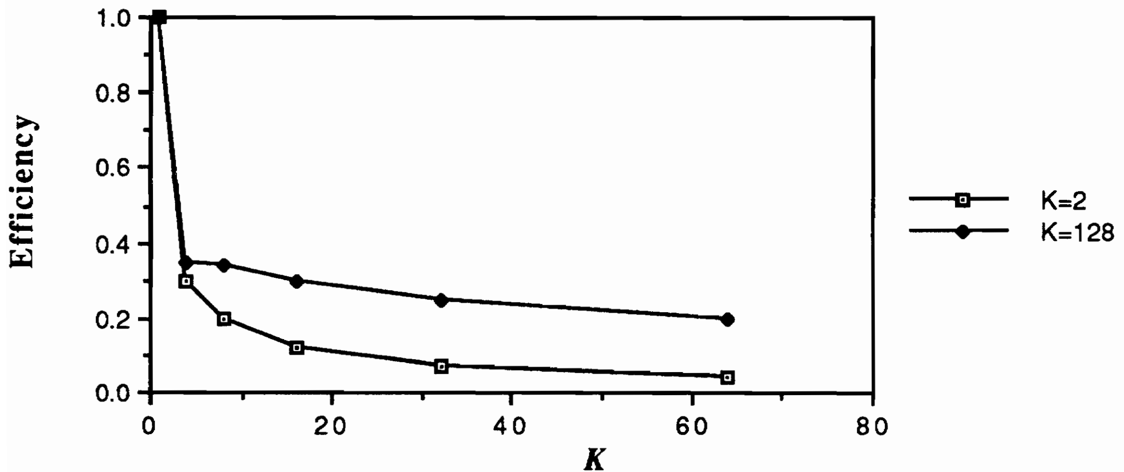


Figure 10a: Efficiency of the colliding pucks simulation for the optimistic protocol for extreme values of K.

K=128 and P=1, 2, 4, 8, 16 and 32. Sending UPDATE_EVENT messages with a rate of $N^2/20$ took approximately 10 percent more time to run for K=2 and P=1, 2, 4, 8, 16 and 32, and 15 percent more time to run for K=128 and P=1, 2, 4, 8, 16 and 32. We compute the efficiency using the formula for efficiency in Section 3.2.3. The graph for the efficiency of this algorithm is similar to the graph in Figure 9a.

Algorithm ES with T1 (Figure 12): The curves have a positive slope, as predicted. The running time decreases as P is increased from 1 to 8, but it becomes disproportionately large and erratic when P exceeds 8. As in the IS+T0 case, we conjecture that the TD is close to saturation at this point. The efficiency of this algorithm becomes very poor as P is increased and is not shown.

Parallel simulation of ES+T1 requires more time than sequential simulation. Therefore, switching from an interval termination algorithm (IS+T0) to an exhaustive termination algorithm (ES+T1) for the torus network benchmark precludes speedup.

To reduce the termination overhead, we reimplemented ES as follows. Each LP is given one FIFO queue for each attribute that is required to evaluate C. The FIFO queues are initially empty. Rather than sending one UPDATE_EVENT message each time an attribute required to evaluate C is assigned a new value, the LP stores the LP's current local clock and the new attribute value into the FIFO queue. When the queue is full, its contents are sent to the TD, and the queue is emptied. The resulting implementation improves the performance of ES+T1 to within a small constant of IS+T1 for the P=8 case using a queue size of 20 (Figure 13). The graph for the efficiency of this algorithm is similar to the graph in Figure 9a.

Algorithm IS with T2 (Figure 14): The curves have a positive slope, as predicted. The running time decreases as P is increased from 1 to 64. The parallel simulation requires

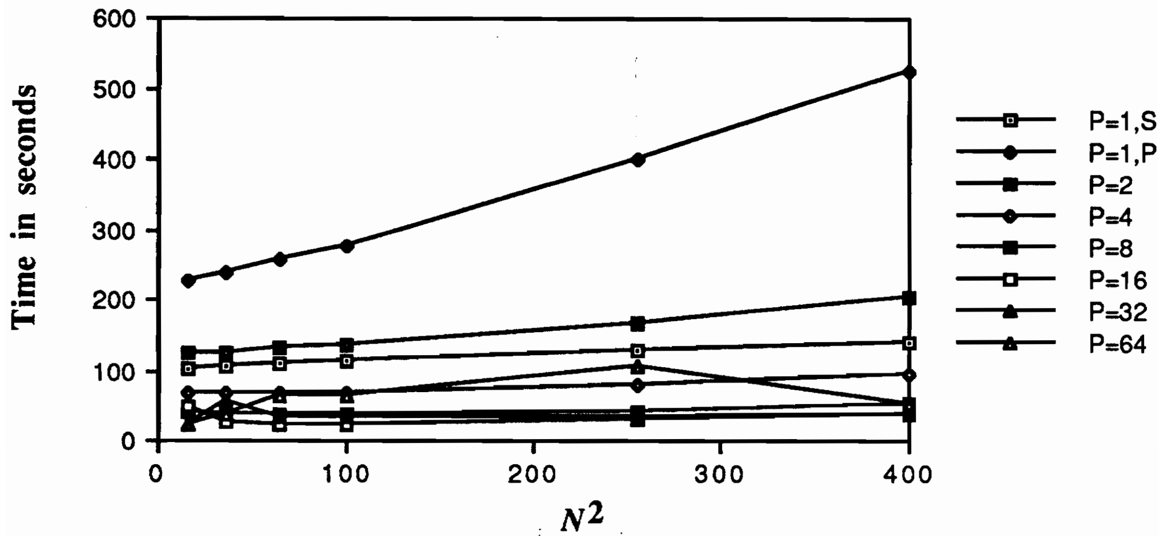


Figure 11: Wall Clock Time required to complete simulation for N^2 LP's with IS + T0 for the optimistic protocol.

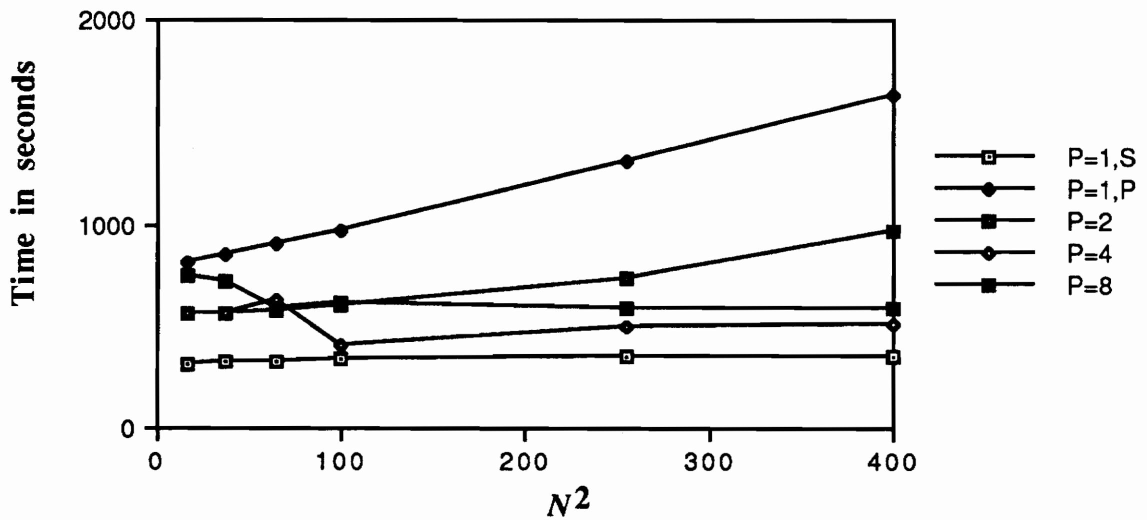


Figure 12: Wall Clock Time required to complete simulation for N^2 LP's with ES + T1 for the optimistic protocol.

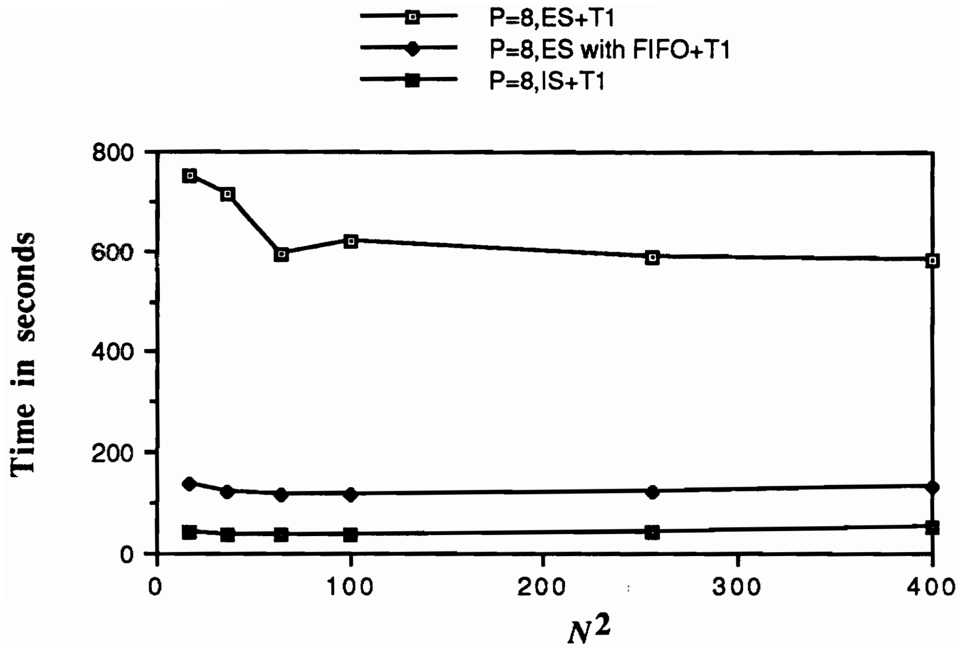


Figure 13: Wall Clock Time required to complete simulation for N^2 LP's with ES + FIFO + T1 and P=8 for the optimistic protocol.

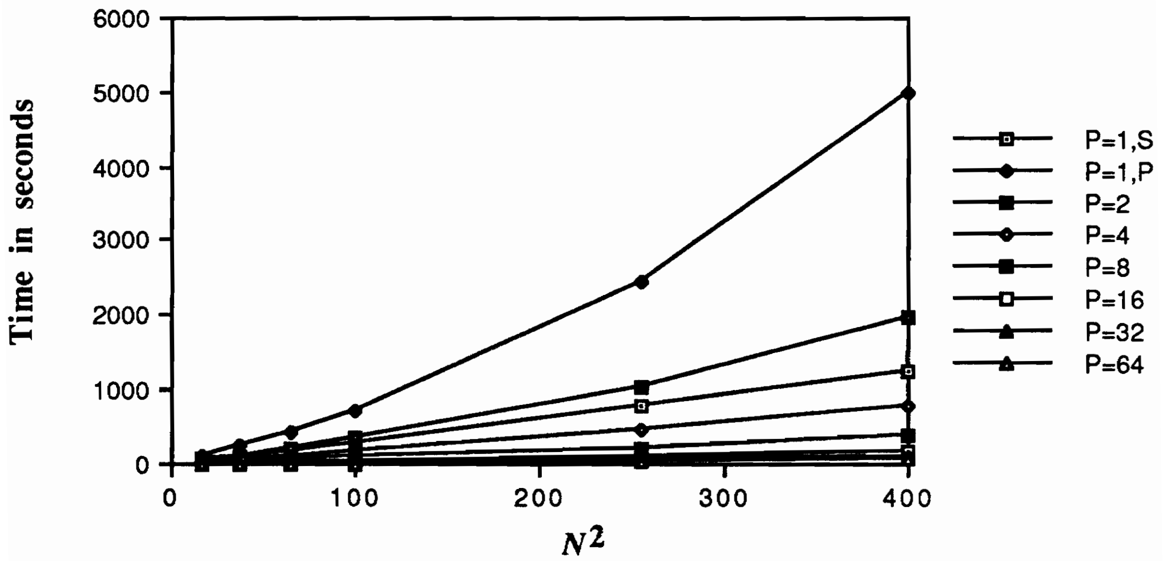


Figure 14: Wall Clock Time required to complete simulation for N^2 LP's with IS + T2 for the optimistic protocol.

more time than the sequential simulation when fewer than 4 processors were used. The graph for the efficiency of this algorithm is similar to the graph in Figure 9a.

Algorithm ES with T3 (Figure 15): The curves have a large slope, as predicted. The running time decreases as P is increased from 1 to 16, but increases with 32 and 64 processors. This could again be due to the TD nearing saturation. Once again the sequential simulation does better than any of the parallel simulations. The efficiency of this algorithm becomes very poor as P is increased and is not shown.

Algorithm IS with P0 (Figure 16): The curves have a hump at $K=16$ but are otherwise essentially flat. The parallel simulation running time decreases as P is increased from 4 to 64. The parallel simulation requires less time than the sequential simulation when more than 4 processors were used. The simulation ran out of memory for $P=1$ and $P=2$. We do not compute the efficiency for this graph as it cannot be characterized by a simple expression. The graph for the efficiency of this algorithm is similar to the graph in Figure 10a.

We also performed experiments with UPDATE_EVENT message sending rates of once every 5 events and once every 20 events for $P=4$, 8, 16, 32 and 64, and $K=2$, 16 and 128. The variance in running times was less than 2 percent and so we conclude that running times are not sensitive to the rate with which UPDATE_EVENT messages are sent.

Algorithm ES with P1 (Figure 16): All the running times for this algorithm are essentially the same as those for algorithm IS+P0. We conjecture that due to the complex nature of this simulation, the overhead due to a high message arrival rate at the TD is negligible compared to the rate with which messages are generated and executed by the

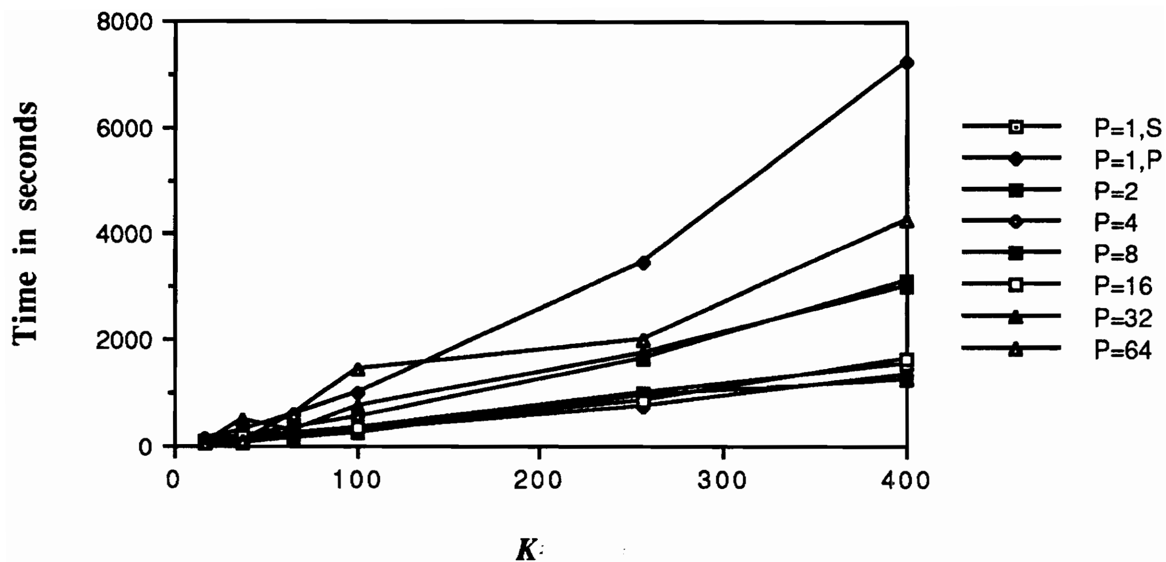


Figure 15: Wall Clock Time required to complete simulation for N^2 LP's with ES + T3 for the optimistic protocol.

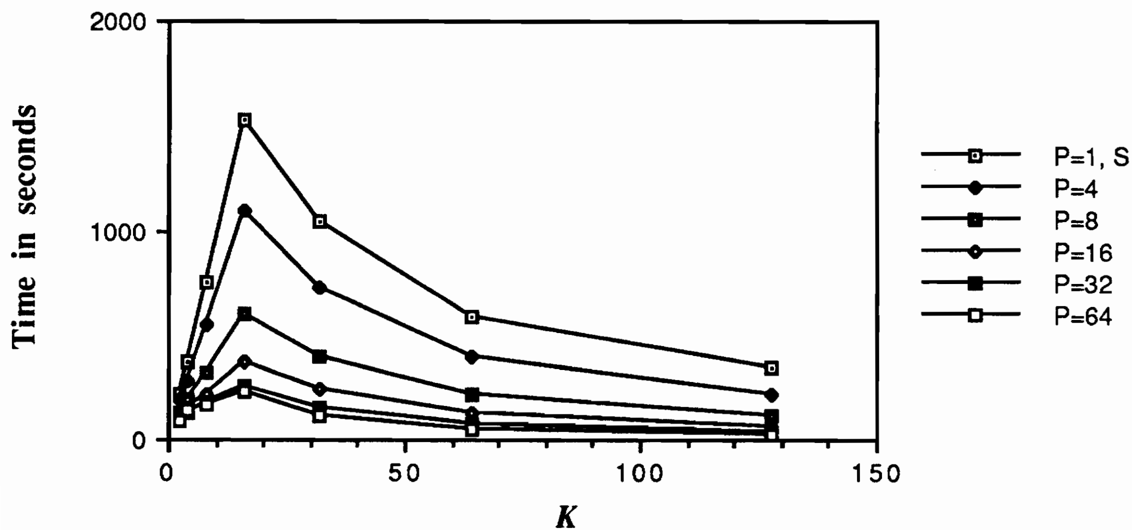


Figure 16: Wall Clock Time required to complete simulation for K LP's with IS + P0, ES + P1 for the optimistic protocol.

simulation itself. Once again, we do not compute the efficiency for this graph as it cannot be characterized by a simple expression. The graph for the efficiency of this algorithm is similar to the graph in Figure 10a.

Algorithm ES with P2 (Figure 17): The curves have a positive slope, as predicted. The parallel simulation running time decreases as P is increased from 4 to 32. The parallel simulation requires less time than the sequential simulation when more than 4 processors were used. The simulation ran out of memory for P=1 and P=2. The graph for the efficiency of this algorithm is similar to the graph in Figure 10a.

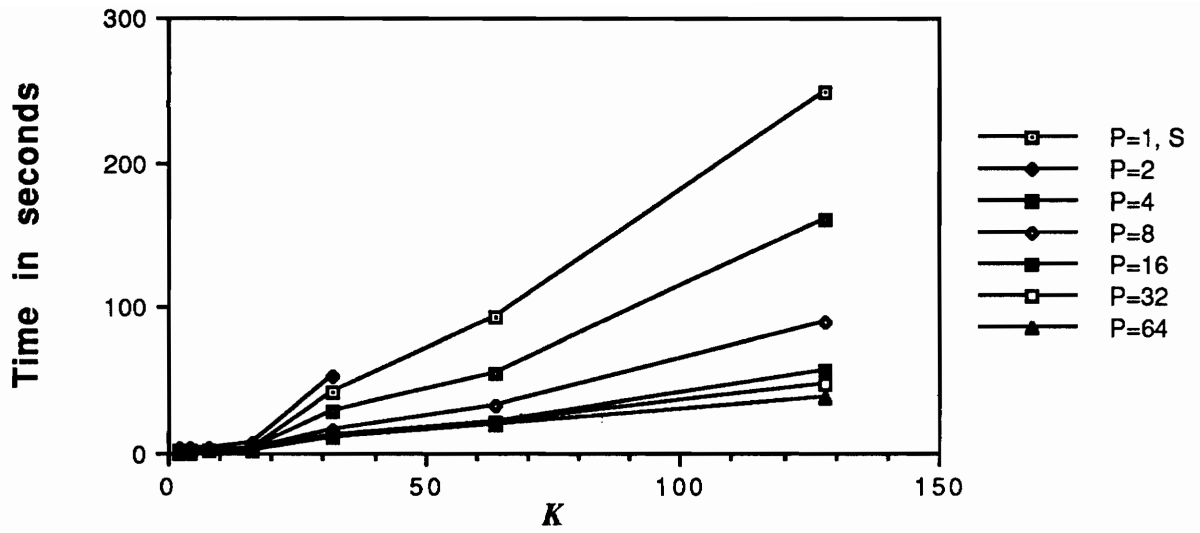


Figure 17: Wall Clock Time required to complete simulation for K LP's with ES + P2 for the optimistic protocol.

CHAPTER 5: CONCLUSIONS

Analysis of the algorithms and the results of our experiments indicate that when the simulation size is sufficiently large and an unstable termination condition that is a function of attributes private to $O(M)$ LP's is used, where M is the number of LP's, then evaluation of the termination condition will dominate the wall clock time required to execute the simulation and reduce or even preclude speedup. At this point finding algorithms to evaluate the termination condition (e.g., using parallel evaluation of C, use of multiple TD's) becomes more important than parallelizing the underlying simulation. For example, a change of a single word in a termination condition (T2 versus T3 for the torus benchmark) can change the time complexity of the simulation from $O(M)$ to $O(M^2)$ (Figure 8). Other unstable condition graphs (Figures 12 and 15) are inconclusive at this point, because the simulation sizes used are insufficient to identify whether the curves are linear or are the initial portions of exponential curves. The storage requirements of the simulations precluded measurement of larger simulation sizes to confirm the graph shapes. In all cases termination increases the simulation running time, and for unstable conditions it does so by a substantial amount.

Output measure generation is a less severe problem. It can be done with zero additional space using rollback, or by restarting the simulation in protocols without rollback, by roughly doubling the real time required for simulation. Alternately, a conservative-synchronous protocol can do output measure generation in additional space proportional to the window size without increasing the real time required for simulation. However, one might limit the window size in a conservative-synchronous protocol, possibly increasing the real time required for simulation, to limit the space required for measure generation.

Conservative-synchronous and optimistic simulations appear to be equally amenable to the addition of termination and output measure generation algorithms. Conservative-synchronous protocols require somewhat more programming effort for unstable conditions, because one must add a rollback mechanism. However rollback is simpler in conservative-synchronous protocols than in optimistic protocols. Conservative-asynchronous protocols require at worst unbounded space, and are recommended only when a stable condition is used.

The termination cost dominates the time required to run a simulation when the wall clock time required for each evaluation of *C* exceeds the wall clock time required to execute an event. Therefore for an unstable condition there should exist a critical value of simulation size, such that when the size is exceeded, the real time required for simulation will dramatically increase. The graphs in this thesis do not exhibit this phenomena, but this is probably due to the fact that the simulation sizes we use are relatively small -- no more than 128 pucks or 400 queuing network servers. In contrast, in a simulation with thousands to millions of LP's, the cost of termination may become a fundamental limitation on speedup.

One issue that still needs to be addressed is to see how our results would change if the IS and ES algorithms were embedded into TWOS itself. How biased are the results obtained by implementing the termination detector as a user process? Algorithm IS + T2 is already implemented in TWOS as a user facility embedded in the simulation system. However it can only detect termination at simulation time instants when GVT is calculated and would detect T2 the first time GVT is calculated after T2 becomes *true*. Other open issues are the assessment of the space overhead required to implement termination

algorithms, and the measurement of the performance of termination algorithms with very large simulations.

REFERENCES

Abrams, M. 1992. *Terminating Parallel Simulations*, Technical Report 92-01, Computer Science Department, Virginia Tech, Blacksburg, Virginia.

Abrams, M. and Richardson, D. S. 1991. "Implementing a Global Termination Condition and Collecting Output Measures in Parallel Simulation," in *Proceedings of the 1991 Workshop on Parallel and Distributed Simulation,* 86-91, Anaheim, CA.

Abrams, M. , Sanjeevan, V. and Richardson, D. S. 1992. "Termination and Output Measure Generation in Optimistic and Conservative-Synchronous Parallel Simulations," Technical Report 92-25, Computer Science Department, Virginia Tech, Blacksburg, Virginia.

Ayani, R. 1992. "Parallel Simulations using Conservative Time Windows," Submitted to Winter Simulation Conference, 1992.

Bershad, B. N., Lazowska, E. D. and Levy, H. M. 1987. "PRESTO: A System for Object-Oriented Parallel Programming," Technical Report 88-01-04, Department of Computer Science, University of Washington, Seattle.

Chandy, K. M. and Misra, J. 1988. *Parallel Program Design: A Foundation,* Addison-Wesley, Reading, Mass.

Chandy, K.M. and Lamport, L. 1985. "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems 3(1)*, 63-75.

Chandy, K. M. and Sherman, R. 1989. "Space-Time and Simulation," In *Proceedings of Distributed Simulation 1989, 53-59*. Tampa, Florida, March 1989.

Fujimoto, R. M. 1989. "Parallel Discrete Event Simulation," In *Communications of the ACM 33*, Oct 1990, 173-179.

Jefferson, D. 1985. "Virtual Time," *ACM Transactions on Programming Languages and Systems* 7: 404-425, July 1985.

Jefferson, D. 1987. "Distributed Simulation and the Time Warp Operating System," In *Proceedings of the 11 th ACM symposium on OS principles*, OS review Vol 21, No. 5, May 1987.

Jefferson, D., Hontalas, P. et. al. 1989. "Performance of the Colliding Pucks simulation on the time warp operating system (Part 1)," In *Proceedings of Distributed Simulation 1989, 3-7*. Tampa, Florida, March 1989.

Lakshman, T.V. and Wei, V.K. 1991. "On Efficiently Computing Functions of Distributed Information," Techinical Memo, Bellcore, Redbank, NJ..

Li, H. F., Radhakrishnan, T. and Venkatesh, K. 1987. "Global State Detection in Non-FIFO Networks," in *The 7th International Conference on Distributed Computing Systems*, 364-370, Berlin, Germany, September 1987.

Lin, Y. B. 1991. On Terminating a Distributed Discrete Event Simulation, Submitted to the Journal of Parallel and Distributed Computing.

Lin, Y. B. and Lazowska, E.D. 1991. "Design Issues for Optimistic Distributed Discrete Event Simulation," Submitted to *IEEE Transactions on Parallel and Distributed Systems,* 1991.

Lubachevsky, B. 1989. "Efficient distributed event-driven simulations of multiple loop networks," *Communications of the ACM 32*, Jan 1989, 111-123.

Mattern, F. 1987. "Algorithms for Distributed Termination Detection," *Distributed Computing* **2**: 161-175.

Misra, J. 1986. "Distributed Discrete-Event Simulation," *ACM Computing Surveys 18*, Vol. 1, 39-65.

Nance, R.E. 1981. "The Time and State Relationships in Simulation Modeling," *Communications of the ACM 24,* Feb 1981, 173-179.

Nicol, D. M. 1990. *The Cost of Conservative Synchronization in Parallel Discrete Event Simulations*, ICASE Report No. 90-20, NASA Langley Research Center, Hampton, Virginia, May 1990.

Richardson, D. S. 1991. *Terminating Parallel Discrete Event Simulations*, M. S. Thesis, Technical Report 91-9, Computer Science Department, Virginia Tech, Blacksburg, Virginia, May 1991.

Sanjeevan, V. and Abrams, M. 1991. "The Cost of Terminating Synchronous Parallel Discrete-Event Simulations," In *Proceedings of the 1991 Winter Simulation Conference,* 642-651, Phoenix, AZ, December 1991.

Sokol, L.M., Briscoe, D. P. and Wieland, A.P. 1988. "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution," In *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3), July 1988, San Diego, CA.

Spezialetti, M. and Kearns, P. 1986. "Termination Detection for Distributed Computation," *Information Processing Letters 18(1)*, 33-36, January 1986.

VITA

Vasant Sanjeevan was born in Calicut, India on December 12, 1966. He did his initial schooling in New Delhi, at the Army Public School. At the age of 12, his family moved to Moscow in the erstwhile U.S.S.R., where he went to school number 591 for three years. There, among other things, he also learnt English in Russian. After returning to India and graduating from high school he joined the Indian Institute of Technology, Delhi, where he pursued a Bachelor's degree in Computer Science and Engineering. Subsequently, he went on a one year traineeship with IBM in Japan, where raw fish and meat became part of his diet. Vasant returned to graduate school in 1989 to pursue a Master's degree in Computer Science at Virginia Tech. After graduating, Vasant will move to Redwood City, California to begin a job with Oracle Corp.