DISTRIBUTED SIMULATION, ALGORITHMS AND PERFORMANCE ANALYSIS

Behrokh Samadi

1984
CSD-850006

UNIVERSITY OF CALIFORNIA

Los Angeles

Distributed Simulation, Algorithms and Performance Analysis

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Computer Science

by

Behrokh Samadi

1985

The dissertation of Behrokh Samadi is approved.

[illegible]

D.R. Jefferson

[illegible]

T.M. Liggett

[illegible]

R.T. Nelson

[illegible]

R.R. Muntz, Committee Co-Chair

[illegible]

D.S. Parker, Committee Co-Chair

University of California, Los Angeles

1985

To my father

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I wish to extend my appreciations to my advisors, Professors Stott Parker and Richard Muntz for their guidance, encouragement and patience.

My thanks to the other committee members, Professors D. Jefferson, T. Liggett, and R. Nelson for their review of this dissertation. Special thanks to Professor Liggett who spent his invaluable time working with me on some parts of this dissertation. I would also like to thank Professor L. Kleinrock who served as the committee member at an earlier time.

My appreciation to Dr. Henry Sowizral for his guidance, advice and enthusiastic support during the course of this work.

Special thanks to Dr. Stephen Lavenberg for his support in initiating this research and the materials of Chapter 4.

I also like to thank Professor Eli Gafni for his useful comments and advice on this work.

This research was partially supported by the research project "Research in Distributed Processing", Contract No. N00014-C-79-0866, under the direction of Professor A. Avizienis and supervision of Professor Stott Parker. Additional support was provided by the Rand Corporation under the System Development Foundation grant #58 with the supervision of Dr. Henry Sowizral. I would like to thank both organizations for their financial support.

VITA

March 24, 1952 ----	Born, Tehran, Iran
June 1974 -----------	B.Sc. in Mathematics, University College London, England
June 1976 ------------	M.S. in Computer Science, Stanford University
1976-1978-------------	Computer Science Instructor, Tehran University of Technology
1978-1979-------------	Programmer/Analyst, Telecommunications Company of Iran
1980-1983 ------------	Research Assistant, UCLA
1983-Present --------	Information Sciences Consultant, Rand Corp

ABSTRACT OF DISSERTATION

Distributed Simulation,

Algorithms and Performance Analysis

by

Behrokh Samadi

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1985

Professor D.S. Parker, Co-Chair

Professor R.R. Muntz, Co-Chair

Simulation is one example of an application that shows great potential benefits from distributed processing. The conventional approach to simulation, that of sequentially processing the events, does not exploit the natural parallelism existing in some simulation models. This is particularly true in large models, where submodels often interact weakly and can be simulated in parallel. The decreasing cost of multiprocessor systems also suggests that a distributed approach to simulation can be workable. Moreover, such an approach can be very attractive, since time and memory limitations, often major constraints with simulation programs, may be alleviated by distributing the load among several processors.

Distributed simulation requires a set of processors that can communicate by sending messages along the links of a communication network or via a shared memory. The processors each simulate a submodel of the overall model and interact when necessary. Submodel interactions produce the interprocessor communication in the simulator.

Two methods for distributed simulation are studied in this thesis. Both methods are applicable to discrete time simulation models and are fully distributed in the sense that they require no central control. In one method, each processor can simulate independently as long as it is certain that no events will arrive that belongs to the past of the simulation process. In the second method, processors are not concerned about future arriving events. They simulate independently and roll back if an event arrives that belongs to the past.

The thesis consists of two parts. The first presents some centralized and distributed algorithms for efficient utilization of the second method. The issue of load balancing is also discussed in this part and some heuristic algorithms are presented.

The second part of the work consists of mathematical modeling and analysis of models of both methods. The analysis gives some insight into the effects of different system parameters on the performance. The performance of each method is compared with the other and also with single processor simulation. The mathematical models are then confirmed and complemented with the simulation results. Finally, results of the implementation of the second method are presented.

CHAPTER 1

Introduction

System simulation is a technique of solving problems by observing the changes of a dynamic model of a system over time [GOR 69]. A *model* is defined to be the body of information about the system, gathered for such a study. The types of changes within the system identify different types of systems. If changes occur smoothly and continuously over time, the system is called a *continuous system*. However, if the changes occur at discrete points in time, the system is a *discrete system*. It is often not possible to refer to a system as totally continuous or discrete. Rather, the approach to modeling a system and the attributes of interest classifies the system and its model as being continuous or discrete.

The mathematical model of a continuous system typically consists of differential equations. An example is the continuous model of an inventory control system. The rates at which orders are placed or met are used to construct differential equations that describe attributes of interest such as the inventory level.

Throughout this work, only the problem of simulation of discrete systems is studied. Therefore, the term system and simulation model will implicitly refer to discrete systems and models. In a discrete model, changes, or *events*, occur at discrete points in time. An example of such a model, is a bank teller. The points of arrival and departure of customers are some events of interest that occur at discrete points in time.

In the conventional approach to discrete system simulation, the basic structure is a queue of simulation events or an *event queue*. An event denotes a change in the state of the system to be simulated. The time at which the change occurs is the *simulation time* or the *time of occurrence* of the event. Events are sorted in increasing order of simulation time in the event queue. In the *event based* simulation, the processor simulating the system, also called the *simulator*, removes the event with the lowest simulation time and advances its *simulation clock* to that of the event. The elapsed computation time is called the *real time* of the processor. Simulation of one event, may result in generation of one or more events in the future of the simulation process. The generated events are placed in the proper position in the event queue.

The next event to be simulated may be one of the latest generated events. So, a processor cannot remove more than one event simultaneously, if strict sequencing of events is essential. However, if the system to be simulated consists of concurrent processes, strict sequencing of events is not necessary. It is only essential that events within each process be simulated in sequence.

In addition, at the points of interaction of two processes, the events need to be ordered.

An example of such a loosely coupled system is a distributed file management system. As long as transactions on different sites are independent of each other, the order of simulation of the events should only be observed for events occurring within each site. Consider four transactions a,a' on site A and b,b' on site B. a is independent of b and b'. However, assume that a' will result in generation of another transaction a'' for site B. An order in which events may occur in the simulated system is abb'a'a''. As a result of loose coupling, simulation of the events corresponding to the transactions do not have to follow the above order exactly. In fact, simulation in any of the following orders is computationally equivalent to the above serial execution [BEG 80].

abb'a'a'' , aa'bb'a'' , aba'b'a'' , bb'aa'a'' , baa'b'a''

For such a loosely coupled system, a distributed simulation method can be effective.

The distributed simulation techniques studied here require a set of processors that can communicate by sending messages along the links of a communication network or via a shared memory. The simulation model is divided into submodels that can operate concurrently. The processors each simulate a submodel and interact when necessary. In the above example, each of the sites is a reasonable choice for a submodel to be simulated by a single processor. Submodel interactions produce the interprocessor communication in the simulator. Since the event queue is not centralized in such an environment, a major task of a distributed simulation algorithm is to synchronize event processing at the points of interaction.

Maintaining the correct simulation time sequence is termed ***synchronization***. A number of different methods for distributed simulation have been suggested in the past. Distributed discrete event simulation methods vary in the degree of looseness of the synchronization and the approach to control of the system. Throughout this work, two particular methods for distributed simulation are studied. Both methods impose a distributed control over the operations and no shared variables are used.

In one of these methods, a processor will not simulate unless it is certain that all the future arriving events (in real time) belong to the future of the simulation. In the other method, the processors are not concerned about future external events. If any such event arrives that has happened in the past of the simulation process, the processor adjusts its operation by rolling back to a consistent state in the past. Some differences are obvious. One method may unnecessarily keep a processor waiting. Consequently deadlock can occur. The other, introduces the extra overhead due to the rollback operation.

In this thesis, the emphasis is on the second method described above. Algorithms are suggested that will result in efficient implementation of that method. A study of the performance of both methods is done by mathematical modeling and analysis. In the analysis, the performance of the two methods are compared. In addition, each distributed simulation method is compared with single processor simulation. The analytic results are confirmed and complemented by simulation. Finally, results of an actual implementation of the second method are presented.

1.1. The Environment

In this section, we discuss the simulation model and the simulation process in single and multi-processor environments and establish the requirements and assumptions for the material in the following chapters.

The simulation models considered are assumed to be discrete event models. The simulation process in the single processor case is performed with the conventional event based simulation approach. The processor maintains an event queue which contains the non-simulated events in their simulation time order. As mentioned, the simulation time of the event is the time of occurrence of that event in the system to be simulated. The *next event* to be simulated is the event that has the lowest simulation time. The processor removes the next event from its event queue, updates its simulation clock to the simulation time of the event and simulates that event. As a result of simulation of an event one or more events may be generated that are placed in the proper position in the event queue. At any point in time, there may be multiple events that have the lowest simulation time. The real time order of simulation of such events or the order in which the next event is being selected, depends on the model.

To simulate a similar model with multiple processors, the model needs to be decomposed into ***submodels*** or ***objects*** (both terms will be used). The submodels correspond to subsystems in the system to be simulated that can operate concurrently. One or more submodels are assigned to each processor. Similar to the conventional approach, each processor maintains its own event queue and simulation clock. Two submodels need to interact when one (S_1) generates an event that is to occur in the other, (S_2). To pass the generated event to the relevant submodel, the processor simulating S_1 sends a message carrying the event to the processor simulating S_2. Such a message will be called an ***event message*** and the simulation time of the event it is carrying will be referred to as the simulation time of the message. The message carries other information such as the identities of the source and the destination processors. Different distributed simulation methods may require other information to be included in the message. This will be described when the methods are being studied.

A processor that receives an event message, can in general be in any state in its simulation process. The distributed simulation method needs to make sure that events that arrive from outside the submodel, ***inter-submodel*** events, are simulated in the proper order with respect to the events that are generated internally, ***intra-submodel*** events.

In our study of distributed simulation methods, we make the following assumptions about the simulation model.

M1. The model is decomposable into submodels, such that the operations of the submodels and their interactions can be clearly defined. The decomposition process is not discussed in this work.

M2. Each submodel is assigned to one and only one processor.

M3. Events within each submodel must be simulated in increasing order of simulation time. However, independent events occurring at different submodels do not require such strict ordering.

M4. The order of simulation of events with the same simulation time is immaterial (This can be weakened to restrict only events within the same submodel).

M5. The simulation model is ***non-terminating*** in the sense that the simulation process does not terminate as a result of absence of simulation events..

M6. The simulation model is ***sound***. That is, both (i) the simulation clock of the processor in single-processor simulation is a non-decreasing function of real time and (ii) given a non-terminating model, the simulation clock increases in finite time.

M7. Given the seed of the random number generator, the simulation model is ***deterministic***. i.e. the current state of the model and the current input from the event queue can uniquely determine the next state.

M8. Every event that is generated as a result of simulation of another event has a simulation time strictly greater than the simulation time of the event that caused its generation. In other words, the simulation time of generation of an event is strictly less than the simulation time of occurrence of that event.

M9. Only a bounded number of simulation events can be generated or simulated in finite time.

M10. Every event requires finite and non-zero processing time. Note that the finiteness follows from M6.

Considering these assumptions, the following shows the steps taken by every processor in simulating the submodel and interacting with other processors. Note that in what follows, events are assumed to be in the proper order. The synchronization problem will be discussed later.

```
while not done do
begin
    remove the next event from the event queue;
    increase the simulation clock of the processor to the simulation time of
    the event;
    if the event belongs to another submodel then
    /* event is generated by this submodel but is to occur at a different one
    */
        send the event to the relevant processor;
    else
    begin
        simulate the event;
        put any events generated on the event queue;
    end
end
```

According to the above, a processor sends an inter-submodel event only after its own simulation clock reaches the simulation time of the event. In other words, the event was generated at an earlier time, but it is transmitted only when the sending processor simulates up to the point of occurrence of the event.

An alternative approach is to send the inter-submodel event at the time of its generation. This means that the processor that generates an inter-submodel event allows the event to affect the receiving submodel, possibly before it reaches that time itself. The performance tradeoffs depend on the model being simulated. If generated inter-submodel events cannot be canceled, the latter approach may be advantageous. However, if event cancellation occurs frequently, the former approach is a better choice. The term *future scheduling* refers to the second approach in transmission of inter-submodel events. To be consistent with the analytic models of Chapters 4 and 5, the former approach is assumed throughout this work. It should be mentioned that the materials of Chapters 2 and 3 are independent of this assumption. However, the implementation results of Chapter 6 are based on a model that utilizes the second approach.

In the model we adopt for the operation of the processors, a processor and the inter-submodel event it is transmitting have the same simulation time at the time of transmission. Note that this does not exclude the simulation of models that have a non-zero simulation time delay in inter-submodel interactions. The delay can be modeled and simulated by either the sending or the receiving processors.

In addition to the above assumptions (M1-M10) on the simulation model, we need to make the following general assumptions regarding the processing environment. These are assumed to apply throughout this work, unless otherwise mentioned.

G1. Processors are working correctly with no errors, infinite loops, unnecessary idle periods and breakdowns.

G2. Sufficient storage is available to store messages and maintain all data structures. i.e. no blocking can occur as a result of unavailable storage.

G3. The communication medium is reliable. Messages are delivered to their destination in finite time without any errors.

G4. The communication medium does not necessarily preserve the FCFS ordering of the messages. i.e. messages transmitted between the same source-destination pair may not be received in the order they have been sent.

Note that the assumptions M1-M10 and G1-G4 are not too restrictive. In fact, except for M3,M4 and M8, these assumptions are satisfied by many simulation models. The more restrictive assumptions, M3 and M4, are required to ensure concurrent operations in the model. As will be discussed in chapter 2, assumption M8 removes the possibility of infinite loops in the process of simulation.

1.2. Previous Work

A number of different methods for distributed simulation have been suggested in the past. Methods vary with the nature of the simulation model, continuous or discrete, and with their approach to synchronization and control. Detailed description of some of these methods appear in [PWM 79] and [PEA 80]. In this section, two methods for distributed event-oriented simulation are presented.

The processors simulating each submodel and the links along which they transmit event messages, form a directed graph which we call the *communication graph* of the model. In this work, if such a graph cannot be identified, a completely connected graph is assumed. Considering this graph, the processors have two distinct ways in which to assure proper sequencing of events while guaranteeing loose synchronization. In one of these methods, each processor can simulate independently as long as it is certain that no external events will arrive that will be in its simulation past. In the other method, the processors do not require this knowledge. Each processor simulates independently until it receives an event in its simulation past. When such an event arrives, the processor rolls back to a point prior to the simulation time of the incoming event.

1.2.1. Distributed Method Suggested by Bryant, Peacock et. al. and Chandy et. al.

The first approach, that of strictly requiring the proper sequencing of events on each link, has been dealt with in great detail in several papers [BRY 77,79], [CHM 78,79,81], [PWM 79] and [PEA 80]. Their approaches are basically similar. The basic algorithm, the so called "Link Time" algorithm, by Peacock et al., [PEA 80], is as follows.

A time stamp is assigned to each link of the communication graph. Its value, is the simulation time of the earliest non-simulated event message transmitted on that link. Strict sequencing of events on each link is observed. Therefore, a processor can safely simulate up to the time given by the smallest time stamp on its incoming link. Processors remove the event with the minimum time stamp, simulate to that point and then repeat the cycle by removing the next event. Clearly, the minimum value can be determined only when all the links contain at least one non-simulated event. A problem arises when a link is empty; in this case the processor has to wait for the arrival of an event message on the empty link. Waiting can create deadlock if processors form a directed or even undirected cycle in the communication graph. Figure 1-1 shows two cases of deadlock. Labels on the links are the time stamps and those with a "—" sign, specify an empty link.

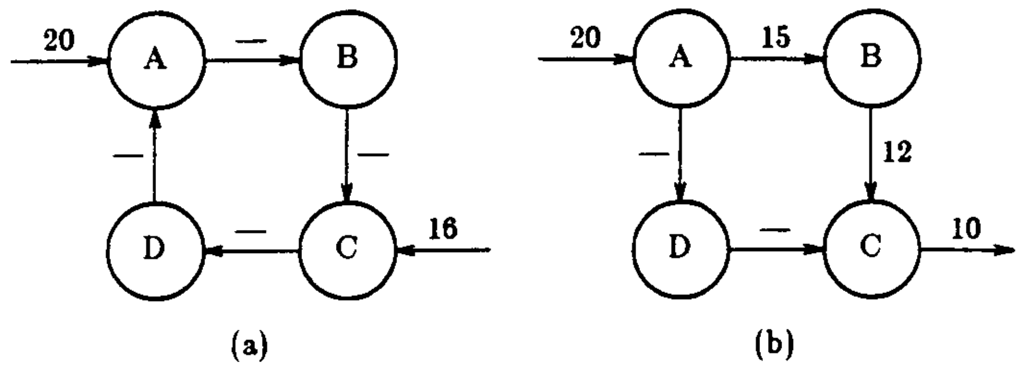

Figure 1-1. Examples of Deadlock

In Figure 1-1 (a), all the processors are blocked. A and C cannot advance although each has an event on one of its incoming links. This type of deadlock remains indefinitely if no deadlock detection and removal scheme exists. In part (b), the blocked processor C cannot advance because A is not generating event messages for D frequently enough. With unlimited buffer storage, C can advance as soon as A generates an event for D and D does so for C. Deadlock can also occur if there is limited buffer space on the links.

To avoid deadlock, the notion of a *null message* has been introduced. A null message is a simulation message that has no content except for a time stamp. Once a processor transmits an event message with time stamp *s* on one of its outgoing links, it will also transmit a null message with the same time stamp on all of its other outgoing links. This way, a processor informs all its neighbors that it is not sending any event messages prior to s. Therefore, the neighbors can use this value in their computation of the next safe simulation period.

The null message can prevent occurrence of deadlock in the case (b) of Figure 1-1. A sends a null message with time 15 to D and D will in turn, send one with the same time stamp to C. C can update the link time of (D,C) to 15 and then safely simulate to 12. However, the deadlock of case (a) is not resolved with this method.

The deadlock shown in (a) can be resolved if the system to be simulated has the property that an event with simulation time s, can only generate events that occur at a time $\geq s+\epsilon$, where ϵ is a fixed non-zero value. In such a case, the null message with this lower bound ϵ can prevent the deadlock problem. With such an assumption, given A's simulation clock is 15, A will send a null message with time stamp $15+\epsilon$ to B. B is certain that it will not be generating any events for any time less than ϵ away, retransmits the null message with time stamp $15+2\epsilon$ to C. C will either simulate to time 16 if $16<15+2\epsilon$, or it will increment the time stamp of the null message and send it to D again. The null message will go through the loop, until its time stamp exceeds 20 at A or 16 at C. In fact, it is enough that only one of the processors in the cycle have the mentioned non-zero property in the simulation clock advance.

The method, although simple and workable, can be very inefficient if the time increments are small. Note that when $\epsilon=1$ and A's and C's ingoing links have time 1000, over 900 messages need to be sent until the normal simulation can resume. Bryant [BRY 77], [BRY 79], further improves this method by adding a more global information gathering algorithm to the system. In this extension, processors are grouped into equivalence classes. Each class is a strongly connected component of the communication graph. Within each class, the local clock values (similar to the minimum link time values), are broadcast along a tree spanning all the processors of the class. Using this information, the processors in each class can find the minimum link time of all the links incoming to the whole class and then compute a safe simulation period. Larger time increments can be obtained this way.

As an extension to the Link Time Algorithm, Chandy and Misra [CHM 79] have assumed the existence of a prediction facility for the simulation model. Each processor can correctly predict certain steps of the simulation process in the future. For each submodel, there is a prediction function L, which at any time t, can correctly predict the behavior of the model until a time $s>t$. This is shown by $L(t)=s$. The prediction function uses the current state and the history of the system to make a prediction. It is assumed that the predictions improve as real time increases. i.e. if $L(t)=s$, then $L(t+\Delta t)\geq s$.

Chandy and Misra [CHA 81], also consider the case of finite buffer storage. Deadlock becomes even more probable if processors are blocked as a result of unavailability of buffer storage. To avoid this type of deadlock, they suggest a deadlock detection and recovery scheme. In this approach, the processors simulate until deadlock. Deadlock is detected by a dedicated process that runs concurrently with the simulation and checks for the occurrence of deadlocks. Recovery is done distributively by finding the processor(s) that can proceed and therefore break the deadlock. The approach is similar to Bryant's global information gathering algorithm. The performance of the method proposed by Chandy et. al. has been measured by simulation and the results appear in [SEE 79]. In this work, the simulation models are networks of queues, with and without feedback. The results show the dependency of the performance measure on the number of branches connecting the queues and the performance degradation with feedback which is to some extent due to the number of null messages transmitted.

1.2.2. The Rollback or Time Warp Method

The Rollback or Time Warp method is a different approach to distributed simulation, suggested by Jefferson and Sowizral [JES 82]. This method gives more autonomy to the processors performing the simulation. The processors simulate independently and synchronize their operations at the points of interactions.

The method requires that the simulation model be represented by objects that interact with each other by sending messages. Messages consist of event messages carrying simulation events or control messages that regulate the distributed operations. Event messages carry two time stamps that are the simulation times of generation and the occurrence of the event.

A group of simulation objects is assigned to a processor. Every processor has a *scheduling queue* that gives the information about which object to simulate next. Throughout this work, the scheduling order is considered to be the conventional event scheduling. i.e. in non-decreasing order of simulation times. However, because of the existing concurrency, other scheduling policies may improve performance.

An object has an *input queue* and an *output queue*. These queues contain all the simulation messages received or sent by the object respectively. The input queue is in fact the event queue of the object. The processor's event queue consists of its objects input queues. In addition; the processor maintains a *state queue* that consists of the state of the objects at regular points in time *(checkpoints)*. For this purpose, the processor needs to save the state of all its assigned objects regularly.

The *simulation time of an object*, called the *Local Virtual Time* (LVT) by Jefferson and Sowizral [JES 82], is the simulation time of that object's last simulated event. Similarly, the *simulation time of a processor*, or *Processor's*

Virtual Time (PVT) is the simulation time of its last simulated event. Note that this definition applies to the case where the scheduling is in increasing order of simulation times.

Processors simulate events in the proper simulation time order. Generated events are sent to the processor on which the receiving object resides. When transmitting messages (simulation or control messages), all objects whether they reside on the processor or not, are treated similarly. In [JES 82], the event messages are sent at the simulation time of generation of the event. (Note that this is different from our model of processor operation discussed in section 1.1. This difference is made to simplify the presentation here and will have an affect only on the performance of the system, if it has an affect at all.) If the simulation time of the event, s_e, is greater than the simulation time of the receiving object, s_o, the event is simply queued in the input queue of the receiving object and the simulation continues. However, if $s_e < s_o$, i.e. if the event has occurred in the simulation past of the object, then the receiving processor's simulation is interrupted. The receiving processor has to roll the object back to a consistent state in the past. The rollback operation involves the following steps :

(i) **Restoration**

The processor must restore the state of the object to a checkpoint s_c, immediately prior to the simulation time of the received event, s_e.

(ii) **Cancellation**

Once an object rolls back, it needs to undo everything it has done during $[s_e, s_o]$. This includes cancellation of all event messages that have been generated by the object during the simulation of this interval. These messages will be referred to as *obsolete* event messages. For the cancellation step, Jefferson and Sowizral [JES 82], suggest that for each obsolete event message an *anti-message* be transmitted by the rolled back object. An anti-message contains the same text, time stamp and address as the original message. It is also treated very similarly to the original message. The rule is that when an event message and its anti-message meet inside the same input or output queue, they are both removed. Hence, if the original or its anti-message has not yet been simulated, the two will meet in the output queue of the sending object or the input queue of the receiving object. This will immediately result in removal of both. If either of the two messages are simulated, the receipt of the other will cause a rollback and a subsequent removal of both from the input queue.

In one approach to cancellation, called *aggressive cancellation*, the anti-messages are transmitted immediately after the object's state is restored to the time of the checkpoint prior to the simulation time of the event. Jefferson [JEF 84] suggested a variation to the aggressive approach in which an obsolete event message is canceled only after the processor reaches the simulation time at which a previous event message had been sent and the message is proven to be obsolete *(lazy cancellation).* This approach is superior to aggressive cancellation if events simulated are relatively independent and cancellation of one does not affect many others. However, in the cases where the inter-submodel events are dependent, the lazy cancellation can degrade the performance by delaying the cancellation of obsolete event messages.

The above methods for cancellation does not require that the communication media have the FCFS property. If messages are delivered in that order, a different approach may be taken that will not use as many messages to cancel the obsolete messages. A rolled back object need only send one message per destination that has received obsolete messages. The canceling message carries the simulation time to which the object has rolled back. The object receiving a cancellation message can take the appropriate actions once it finds the point in simulation time to which the other object has rolled back. A minor modification makes it possible to use the second approach to cancellation without requiring the FCFS property.

(iii) **Simulate forward**

The final step of the rollback operation is to resimulate the behavior of the object up to a time equal to the simulation time of the event that caused rollback. If the states of the objects are saved after simulation of every event, the checkpointing is *perfect* and the "simulate forward" step is not needed. Otherwise, this step is required to reproduce the past states of the object that have not been saved within the interval (s_c,s_e). With the deterministic property of the simulation model (assumption M7, section 1.1), the states that were reached during the previous simulation of this simulation time interval, will be exactly reproduced in the "simulate forward" phase. Consequently, events that have been generated during (s_c,s_e) need not be canceled in the cancellation phase. Hence, during the "simulate forward" period, the processor does not transmit any generated events.

Performing the above three steps, the rollback operation is completed for that object. The simulation can be resumed from the simulation time s_e. Clearly it is possible that the simulate forward step can be interrupted by receipt of an event that causes another rollback.

An object that is rolled back may in turn roll back one or more objects. This is possible if one or more obsolete events have already been simulated. If not, only their removal from any queues are required. Therefore, the occurrence of a single rollback can propagate among the processors and cause many others to rollback, possibly more than once. Figure 1-2 shows this effect.

The vertical lines in Figure 1-2 are the points at which the state of the objects are saved (checkpoints). The arrows show the transmission of the event e_i from one object to the other. The simulation time of the events are in the order of the indices of the arrows. Note that the vertical arrows in this

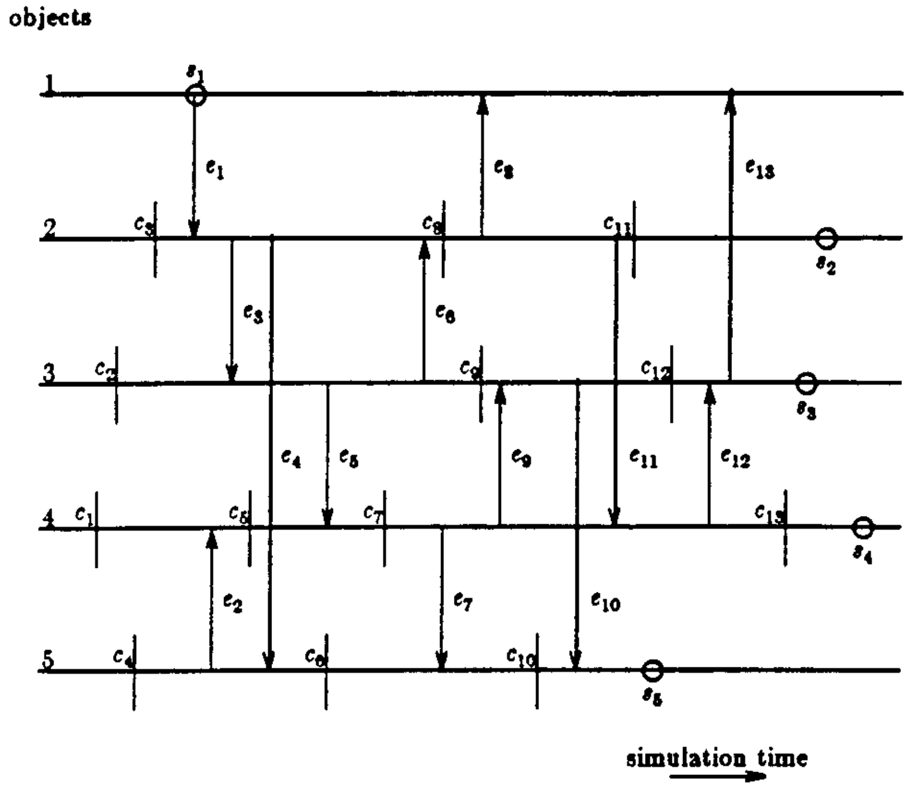

Figure 1-2. Multiple Rollback Effect

Table 1-1. An Undesirable Rollback Order.

object	receives cancellation message	rolls back to	sends cancellation messages for
2	e_1	c_3	e_3, e_4, e_8, e_{11}
1	e_8	—	—
3	e_3	c_2	e_5, e_6, e_{10}, e_{13}
4	e_{11}	c_7	e_7, e_9
1	e_{13}	—	—
5	e_{10}	c_{10}	
4	e_5	c_5	
5	e_7	c_6	
5	e_4	c_4	e_2
4	e_2	c_1	

Figure specify the distributed simulation model we assumed in Section 1.1. i.e., an inter-submodel event is transmitted when the object that generated the event reaches the simulation time of the event.

In Figure 1-2, object 2 has to rollback after it receives an event with simulation time s_1 from object 1. The circles show the simulation time of each object at the time that the rollback occurs. Depending on when the cancellation messages are received by each object, affected objects are rolled back in different orders in real time. An undesirable order for the example of Figure 1-2 is given in Table 1-1.

With respect to the Time Warp simulation, we make the following assumptions that apply throughout this thesis.

T1. To cancel obsolete messages, the aggressive cancellation approach is used.

T2. The rollback operation never fails and requires finite processing time.

Considering the two methods described, the potential advantages of the Time Warp method are:

(i) The processors simulate continuously instead of waiting, although sometimes the work they do is not useful. Clearly this can be an advantage if the processors are dedicated to simulation.

(ii) There is no overhead for detection and removal of deadlock.

The potential disadvantages are :

(i) More storage is required to store the history that may be needed in case of rollback.

(ii) Some work may become obsolete as a result of simulating ahead of incoming events.

(iii) Extra processing time is required for the rollback operations. This includes checkpointing, restoring the state and simulating forward.

1.3. Scope of the Thesis

This work is divided into two parts, algorithms and performance analysis. The first part, Chapters 2 and 3, include extensions to the Time Warp method.

Chapter 2 will begin with a discussion of conditions under which simulations with the Rollback method will progress in real time. Next, two algorithms are presented that deal with some aspects of this method. The first distributed algorithm finds a lower bound on the effective simulation time of the system. The second algorithm deals with the multi-rollback effect and suggests an approach to find the optimal rollback order to minimize rollback overhead. This way each processor will have to roll back at most once if it is affected by another processor's rollback. The latter algorithm requires that some global information be obtainable by a single processor.

In Chapter 3 the problem of assignment of objects to processors, or the so-called "load balancing" problem, is discussed. For a majority of distributed systems, the following issues are of major concern.

(i) Deadlock. The system has to be deadlock free, or an efficient deadlock detection and removal scheme needs to exist. The Time Warp method is inherently deadlock free so this issue is immaterial.

(ii) Division of task into subtasks. We must divide a simulation model into submodels that can efficiently be run in a distributed environment. This problem is not discussed in this work. It is assumed that the model is divided effectively into submodels and that enough submodels exist that will not overload the system.

(iii) Task allocation. Finally, an efficient policy is required that will assign the subtasks to the processing elements such that throughput is maximized.

It is the last issue that is discussed in Chapter 3. In some cases, the best object assignment policy is a dynamic one that responds responds quickly and efficiently to the changes in the system. However, in this work only the static load balancing problem is discussed. The static strategy can be used as an initial policy and then complemented by a dynamic one. In fact, because of the nature of simulation, it is often possible to obtain a large amount of information about the behavior of the system ahead of time, so static load balancing is often reasonable. This information can be used to suggest an efficient object-processor configuration.

In part two of this work, the performance aspects of the two distributed simulation methods are studied. Chapter 4 includes a model of Time Warp that exploits weak interactions between submodels. The model is a fairly simple one and the mathematical analysis is limited in that only the two processor case is considered. However, the results do provide some insight into the effects of system parameters on performance. The mathematical analysis of the two processor model is complemented with a K-processor simulation.

Chapter 5 includes a model of the Link Time algorithm with the extension of a prediction function. The statistical assumptions are similar to those of Chapter 4. Similar to the previous model, the mathematical analysis is limited to the two processor case. Simulation of the same model extends the results to a K-processor system for comparison. The performance measure in Chapters 4 and 5 is defined to be the simulation time advance per unit real time in a K-processor environment over the same parameter in a single-processor case. This performance measure enables us to both compare the models of the two distributed simulation methods and compare distributed and centralized simulations.

Chapter 6 contains the implementation results of the Time Warp method. This approach for distributed simulation has been implemented at the Rand Corporation. The system performing the simulation consists of five Xerox Dolphin machines that run a version of Interlisp. The Dolphin processors are dedicated to the simulation and a Vax-750 on the same ethernet gathers statistics about the simulation. The application or the simulation model is the simulation of a version of Game of Life [GAR 70].

Finally, in Chapter 7 a summary and conclusion of the work is presented and future research topics in this area are suggested.

CHAPTER 2

The Time Warp Method and Related Algorithms

In this chapter, some aspects of Time Warp simulation are studied and two algorithms are presented for efficient implementation of this simulation method.

First, we discuss the progress of simulation time with Time Warp method. Recall the soundness property of the model defined under M6, Section 1.1. That is, the simulation clock of the processor in a single processor simulation is a non-decreasing function of real time and increases in finite time given that the model is non-terminating. In a distributed simulation with no central control, there is no concept of a central simulation clock that can show the progress of simulation time. Therefore, we need to have a global variable that can show the progress. This variable can truly represent the simulation time advance if at any point in the execution no events with smaller simulation time are generated. Such a variable will be called the *effective simulation time* of the system. To further define effective simulation time, we need to introduce some other terms.

An event message that is transmitted by a processor is called an *in-transit* event message until it is placed within the input queue of the receiving object.

An object's ***simulation progress time*** or *spt*, is defined to be the simulation time of its earliest event that can possibly cause generation and/or transmission of other events. Specifically,

spt = simulation time of the object when the object is being simulated.

= simulation time of the earliest event on the object's input queue when the object is not the current object that is being simulated.

= simulation time of the event that caused rollback when the object is in rollback operation.

Given the above, we define the term ***minimum simulation time*** or according to [JES 82], ***the global virtual time***, *GVT* at time *t*, as

GVT(t) = min (min *spt* of objects at *t*, min simulation time of in-transit event messages at *t)*

It will be shown later that the *GVT(t)* is the effective simulation time of the system at time *t*.

2.1. Simulation Time Progress

The rollback operation and its possible propagation among other processors raises a question regarding the simulation time progress of the system. The question is whether there is a situation in which processors never perform any useful work. It is shown below that the assumptions listed in Chapter 1, Section 1.1 are sufficient to prove that the system is guaranteed to simulate ahead and perform useful work that will not be undone. Under these conditions, the effective simulation time of the system, is a non-decreasing function of real time and increases in finite time with non-terminating models. To show this, we need to prove the following.

Lemma 1

The *GVT* of Time Warp simulation is a non-decreasing function of real time, i.e., for every point in real time t and $T \geq 0$, $GVT(t+T) \geq GVT(t)$.

Proof

This result is proved by contradiction. Assume that there is a t such that for some $T>0$, $GVT(t+T) < GVT(t)$.

This implies that at real time $t+T$, there is either an object with *spt* $=$ X or an in-transit event message with simulation time X such that $X < GVT(t)$. Since at real time t such an object or event did not exist, an event must have been transmitted after t that is in-transit at $t+T$ or caused an object to roll back. But this is also not possible. Every object at real time t has either (i) simulated up to the simulation time *GVT(t)* correctly (i.e. will not roll back to a simulation time less than *GVT(t)*), or (ii) it will receive an event that causes it to roll back to that time.

Considering case (i), the object will not generate events with simulation time smaller than *GVT(t)* after t according to the soundness of the simulation model (assumption M6). In case (ii), even when the checkpointing is not perfect, the object will simulate to the time of the event that caused rollback in finite time (G1-G3, T3). Furthermore, it will not transmit any events that are generated during the "simulate forward" period (follows from M7 and the Time Warp definition). Once it simulates up to the time *GVT(t)*, all subsequent generated events have a simulation time $\geq GVT(t)$.

Hence, after time t, no events can be generated that have a simulation time less than *GVT(t)* and so the statement of the lemma is correct. \

According to the above, all objects at time t have either correctly simulated up to (not including) the simulation time *GVT(t)* or they can do so in finite time. Hence, the value of *GVT* at any time can be considered as the effective simulation time of the system.

We showed that the effective simulation time cannot decrease. In what follows, it will be seen that it increases in finite time.

Theorem 1

In Time Warp simulation, for every real point in real time t, there is a $T>0$ such that $GVT(t+T)>GVT(t)$.

Proof

Assume that the statement of theorem is not correct, i.e., there is a point in real time t such that for all $T>0$, $GVT(t+T)\leq GVT(t)$. But, according to the above lemma we have, for every real time t and $T>0, GVT(t+T)\geq GVT(t)$.

From the last two statements it follows that there is a point in real time t such that for all $T>0$, $GVT(t+T)=GVT(t)$.

This implies that the system remains in effective simulation time $GVT(t)$ indefinitely. According to assumptions G1-G3 and T2 (sec. 1.1), such an state cannot be the result of failure of operations or infinite loops outside the simulation process. Considering M8, events with simulation times $GVT(t)$ or less must have been generated at simulation time less than $GVT(t)$. Now let,

$$t_0 = \min(\tau : GVT(\tau)=GVT(t))$$

With the definition of GVT, all events with a simulation time less than $GVT(t)$ must have been generated at real times less than t_0. Now, the number of events with simulation time less than $GVT(t)$ is bounded (M9). According to M10, all such events must eventually be processed in finite time and we know that any events they generate must either (i) have a time stamp greater than $GVT(t)$ or (ii) they have been generated before t_0. Hence, the system cannot remain in effective simulation time $GVT(t)$ indefinitely. So the statement of the theorem is correct. \

Lemma 1 and theorem 1 show that with Time Warp simulation the simulation time progresses. However, we did not discuss the rate at which the simulation time advances compared to single processor simulation. Clearly, this depends on many issues. Some of these issues are studied in chapters 3,4 and 6.

2.2. Minimum Simulation Time (GVT) Algorithm

GVT has several important uses. First and most important, it helps resolve an issue of major concern in Time Warp simulation: memory usage. The amount of storage that maintains state and input/output queues grows as the simulation progresses. In a large or complex simulation, these queues can become enormous. However, when the simulation has advanced far enough, some earlier entries of these queues will typically not be used. It is desirable to reduce storage usage by removing the unnecessary information from the queues periodically.

As was mentioned in the previous section, at any real time t, the earliest time to which an object can roll back is bounded below by the immediate checkpoint prior to the *GVT* at t. Recall that *GVT* at any time is the minimum of the simulation progress times of all objects and in transit messages at that time. Therefore, all messages with a simulation time prior to a checkpoint before *GVT* may be discarded. The Time Warp storage problem may be mitigated by repeatedly finding *GVT* and removing unnecessary information.

In addition to its garbage collection aspect, the *GVT* can be useful in other areas of Time Warp simulation. For example :

(1) Since the effective simulation time at any time t is the *GVT* at that time, this value can be used to determine when a process has terminated.

(2) The *GVT* value can be used by some objects to control the flow of messages. Objects may inspect this value, and, if their current simulation time is too far ahead of it, may temporarily suspend operation.

(3) The speed of simulation and the relative speed of the processors can be estimated with the *GVT*. These ratios can be used to balance the load dynamically between processors by transferring objects from slower processors to faster processors.

We now discuss the *GVT* algorithm. Note that by simply letting all the processors broadcast their simulation times, it is not possible to get a value for *GVT*. The problem is caused by the existence of messages still in transit. A processor that broadcasts its simulation time may later receive an event message that causes it to rollback and therefore invalidate its reported value.

The goal is to have an algorithm that distributively finds *GVT* and also does not interrupt the simulation process except for a short period of time. The algorithm must run concurrently with, but not alter, the simulation process. Note that the exact value of *GVT* may not be obtainable if the simulation is in progress while the algorithm is running.

2.2.1. A GVT Algorithm

The algorithm presented here finds a lower bound on *GVT* and needs the following preparation by the system:

(1) Each processor has a *mode* of operation that is either normal or find.

(2) Receipt of event messages must be acknowledged by the receiving processor. Each processor maintains a list of all its unacknowledged messages which will be called the ***unacknowledged message list.*** Note that this acknowledgement is different from the acknowledgements that may exist in the communication protocol. We require that the acknowledgement be done at the simulation level. Once an event message is received by a processor and put in the input queue of the receiving object, an acknowledgement is sent to the sending processor. Each acknowledgement message carries the mode of the acknowledging processor.

(3) One of the processors is assigned to initiate the algorithm. The process of selecting this processor is not discussed here in detail. One way is to assign one of the processors permanently to this job at the beginning of each simulation run. An alternative is to let the processors select the initiator by a process such as voting. The important point is that for each run of the algorithm one and only one processor be the initiator.

(4) The initiating processor knows the total number of processors currently executing and can communicate with all other processors by sending and receiving messages.

The initiating processor starts the *GVT* computation by executing the following routine:

```
Find_GVT
begin
    newgvt = $\infty$ ;
    proc_reported = 0 ;
    broadcast message START ;
end Find_GVT ;
```

Upon receipt of a START message, a receiving processor executes:

```
Process_Start
begin
    mode = find;
    min1 = minimum simulation time of unacknowledged messages;
    min2 = minimum simulation time of acknowledged messages carrying
        mode= find since the end of the previous GVT computation
        (min2 = 0 if no previous GVT computation was made);
    min3 = the processor's simulation time;
    local_min  = min (min1,min2,min3) ;
    send a message REPORT(local_min) to the initiating processor;
end Process_Start ;
```

In what follows, the variable local_min defined in the above routine will be referred to as the *local minimum*. Each time the initiating processor receives a REPORT (reported_value) message, it executes:

```
Process_Report (reported_value)
begin
    newgvt = min (newgvt, reported_value);
    proc_reported = proc_reported +1;
    if (proc_reported == NUMBEROFPROCESSORS) then
        broadcast message NEWGVT (newgvt) ;
end Process_Report ;
```

Upon receipt of the message NEWGVT, each processor executes :

```
Process_NewGVT(newgvt)
begin
    mode = normal;
    update the value for GVT;
    remove all the information belonging to a time prior the immediate
            checkpoint before newgvt;
end Process_NewGVT;
```

2.2.2. Algorithm Correctness

In this section, we give a proof that the algorithm produces a correct lower bound on *GVT*. Let us begin first with an informal discussion of its operation.

In this algorithm every processor is responsible for considering the simulation times of its assigned objects as well as the simulation times of all the event messages that have been sent by this processor but have not been accounted for in the *GVT* computation by the receiving processor. At real time t, the set of in-transit messages transmitted by a processor is a subset of its unacknowledged messages. The minimum simulation time of the latter is therefore a lower bound on the minimum simulation time of the former.

In Process_Start, the value of the second variable (min2) is included in the computation of the local minimum to avoid problems that can arise in situations similar to the one shown in Figure 2-1.

The vertical lines in Figure 2-1 indicate the times at which processors receive the START message and respond with the local minima. Note that processor 2 has received an acknowledgement for the message with time s_1. However, the acknowledgement has been sent after processor 1 has reported its local minimum at which time it had not received that message. If processor 2 looks only at its unacknowledged messages, it will not consider the time s_1. If that message causes a rollback at processor 2, then the resulting *GVT* can be incorrect.

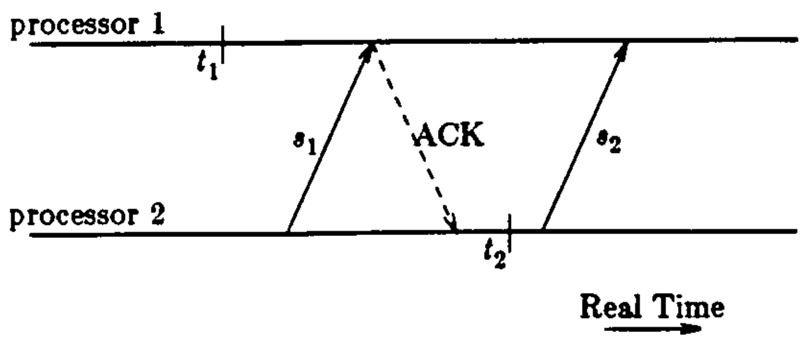

Figure 2-1. Example of an Incorrect Result

The problem occurs because all the processors do not report their local minimum at the same time and the processor sending the acknowledgement for message with time s_1, has done so after it has computed its local minimum. Although the message has been received, its simulation time has not been considered by the receiving processor in its computation of local minimum. To avoid the problem, a processor needs to know whether the acknowledgement for a message has been sent before or after the computation of the local minimum. If the acknowledgement has been sent after the computation then the processor sending the event message is still responsible for its time stamp.

An acknowledgement that carries a mode normal informs the processor receiving the acknowledgement that its time stamp will be considered in the *GVT* computation by the processor that sent the acknowledgement. However, an acknowledgement carrying mode find requires that the processor that sent the message include the time of the message in its calculation of the local minimum.

To prove the correctness of the algorithm, we prove the following.

Theorem 2

Let t_1 and t_K, be the points in real time at which the first and last processors receive the START message. The result of the algorithm, GVT^*, is a lower bound on $GVT(t_K)$.

Proof

Recall that

$$spt_i(t) = \textit{simulation progress time of processor } i \textit{ at } t$$

and

$$est_i(t) = \textit{simulation time of the earliest transmitted event from processor } i\textit{, in transit at time } t$$

Then

$$GVT(t) = \min_i(min(spt_i(t), est_i(t)))$$

Also, let $\{t_i,\ i=1,..,K \text{ and } t_i \leq t_{i+1}\}$ be the points in real time at which the processors receive the START message. Now suppose that the theorem is not correct and $GVT^* > GVT(t_K)$. Let $X = GVT(t_K)$. We first prove the following lemma.

Lemma 2

If $X<GVT^{*}$, then at least one object i rolls back to a simulation time $\leq X$ after its reporting time, t_i.

Proof of the lemma

According to the definition of GVT, if X is the value of $GVT(t_K)$ then at time t_K either (i) there is an object i that has a simulation time equal to X or (ii) there is an in-transit event message with simulation time X.

If (i) is true then an object $i(i\neq K)$ must have rolled back to a simulation time $\leq X$ after its reporting time t_i. Therefore, if (i) is true then we are done. Now if we assume (i) is false, then (ii) must be true. The object receiving the in-transit message has a simulation time $>X$ at t_K. If the simulation time of the object is equal to X, then similar to the case (i), we can show that the object must have rolled back at least once after it sent its report. If the simulation time of the object is greater than X, then the object rolls back once it receives the in-transit message with simulation time X. In any case, (i) or (ii), at least one object rolls back to a simulation time $\leq X$ after it sends its report and the statement of the lemma is correct. \

Back to the theorem

Now, consider one such object i_1, that rolls back to a simulation time $\leq X$ after its reporting time. The rollback has been caused by receipt of an event from another object i_2. Similar to i_1, object i_2 must have rolled back to a simulation time $\leq X$ after its reporting time. Otherwise, it would have accounted for the time X in its report. Following the above argument, we have objects $i_1, i_2, \ldots, i_n$ such that each object i_j rolled back to a simulation time $\leq X$ after its reporting time and then sent an event with a simulation time $\leq X$ to i_{j-1} $(1 < j \leq n)$. Also, each such event that caused rollback was transmitted during the time interval (t_1, t_K).

Consider the last object i_n in this list. Note that a "last" object does exist, since only bounded number of events can be transmitted during the finite time interval (t_1, t_K). The event that caused this object to rollback has been sent by an object i_{n+1} and it does not have the same properties as those mentioned above. Then, i_{n+1} sent the event either (i) before its reporting time or (ii) after the reporting time but i_{n+1} did not rollback to a time $\leq X$ after its report and before it sent the event.

In the case (i), i_{n+1} does not receive an acknowledgement for that event from i_1 or it receives one with the "find" mode. In the case (ii), the reported local minimum of i_{n+1} is less than the time of the event it sends to i_1. In either case, i_{n+1} must have accounted for the simulation time of that event in its report. Since simulation time of the event was $\leq X$, the GVT algorithm could not have produced a value $> X$. Hence, the statement of the theorem is correct and the algorithm returns a value GVT' such that $GVT' \leq GVT(t_K)$.

We showed that the algorithm computes a correct lower bound on *GVT* once the initiator starts the computation. We need to discuss how different runs of the algorithm affect each other.

The *GVT* computation process consists of a ***broadcast*** by the initiating processor (START message), a response as an ***echo*** by all processors (REPORT) and a further broadcast by the initiator (NEWGVT), to give the final result. A processor's operation as far as the *GVT* computation is concerned, consists of cycles of pairs of (normal,find) mode. This is shown in Figure 2-2. In this Figure, processor 1 is the initiating processor. Arrows with labels b and e correspond to broadcast and echo messages, respectively.

Note that the second broadcast does not take place until all processors have responded to the first with their local_min. Then it is not possible for two processors to be in two normal modes, belonging to two consecutive runs of the algorithm. However, it is possible that a START message belonging to the next run reaches a processor, before it receives the NEWGVT message of the previous run. The problem can be alleviated by a second echo to acknowledge receipt of the NEWGVT message. An alternative is to make sure that a processor remembers not to change its mode to normal in Process_NewGVT unless the total number of START and NEWGVT messages it has received are even.

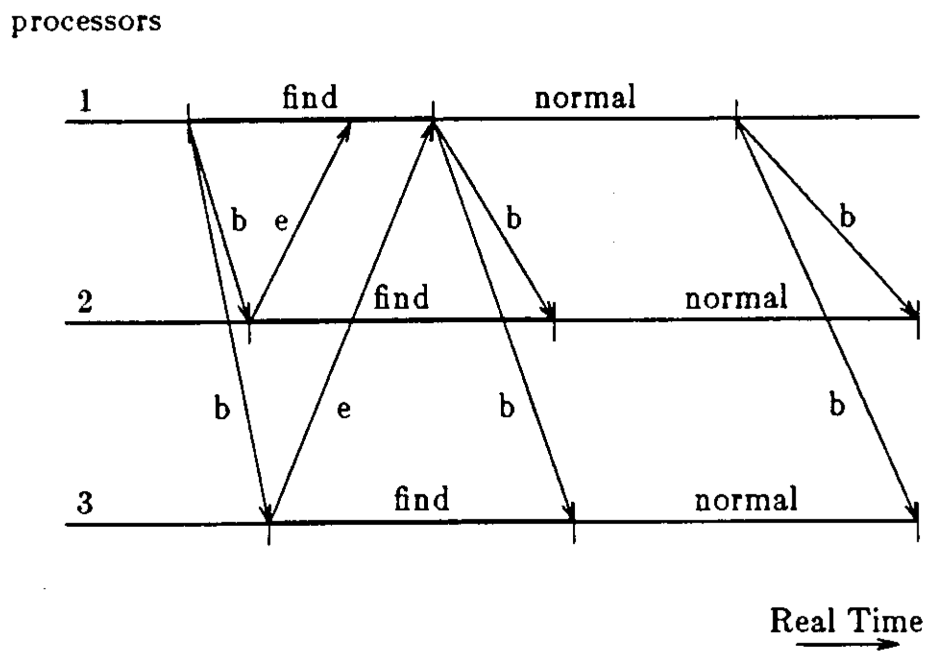

Figure 2-2. Cycles of Normal and Find Modes.

2.2.3. More about the GVT Algorithm

The frequency of initiating a *GVT* computation depends on the application. A processor may ask the initiator to start whenever it needs more storage. It may also depend on the simulation rate of any of the processors.

The problem of finding the *GVT*, is similar to some existing problems in distributed systems. Issues such as determining a consistent global state of the system or termination of a distributed process are similar to the *GVT* problem. These issues have been studied in the past [DIS 80], [FRA 80], [MIC 82] and [CML 83]. However, often these solutions require that the communication media preserve the FCFS property, i.e., messages transmitted from the same source to the same destination are received in the same order they have been sent.

One of the complications of distributed decision making processes is the existence of in-transit messages. A valid local view of a process can be invalidated by the arrival of a (previously in-transit) message, a message that has not been accounted for by any of the sending or receiving processors. With the FCFS property, the sending processor can delegate the responsibility for existing in-transit messages to the receiving processor. This can be accomplished by transmitting a control message to the receiving processor. The receipt of the control message assures the receiving processor that no in-transit messages exist on that link that belong to a time prior to the sending time of the control message. The receiving processor can then make decisions, knowing that any other message after that time will be accounted for by the sending processor and will not invalidate the result.

Without the FCFS property the receipt of messages needs to be confirmed by acknowledgements. A link can be assumed to be clear of any in-transit messages if the sending processor has received all the acknowledgements. Waiting for acknowledgements before making a decision can be unnecessarily time consuming. The algorithm presented above reports a local view of the problem without exchanging any information with the processor's neighbors. The response is immediate and is based on knowledge that has been gathered ahead of time.

In addition, no particular broadcast mechanism is assumed here. A broadcast mechanism such as a minimal spanning tree [DAL 80] or the shortest path tree [WAL 80], not only improves the overall communication cost and the response time but also can give a more global view of the problem to the intermediate processors on the tree.

The algorithm can also be modified such that the role of the initiating processor is restricted to starting the *GVT* computation. The information gathering and final determination of the *GVT* can be done distributively by all processors once each local obtained value is broadcasted to all. It is essential only that one and only one processor starts one phase of a *GVT* computation.

2.2.4. Implementation Results

Jefferson and Sowizral have suggested a different algorithm to compute *GVT* [JES 83]. In this algorithm (algorithm GVT2), an initiating processor starts the *GVT* computation by broadcasting a START message. Upon receipt of this message, every processor will acknowledge receipt of the message and start a *GVT* computation phase. The initiator will send an END message when it receives all the acknowledgements to its START message. The processors will terminate their *GVT* computation phase when they receive the END message and then send their local minimum to the initiator. During the *GVT* computation phase, each processor computes its local minimum by minimizing over its *spt* and the unacknowledged messages. Their algorithm requires three broadcasts interleaved with two echoes as opposed to 2 broadcasts and one echo in the previous algorithm (algorithm GVT1). However, their approach does not require the definition of a mode and therefore its inclusion in message acknowledgements. In their algorithm, after the first and before the third broadcast, there is a real time interval during which all processors are busy computing *GVT*. The minimum of the local minima values during this shared *GVT* computation period guarantees a correct result. Because of the extra broadcast-echo pair algorithm GVT2 takes longer to compute *GVT* than algorithm GVT1.

The two algorithms (GVT1 and GVT2) were implemented and were shown to be working and producing satisfactory results by tests. The time consumed by either of the two algorithms was small. The implementation results of Chapter 6 show that often less than 1% of the total running time was spent on the *GVT* computation. Clearly, by increasing the frequency of *GVT* computations, this fraction will be increased. However, with a frequency that was sufficient for the application, the *GVT* computations showed to be a small fraction of the total run time of the system. The results did confirm the initial observation that the algorithm GVT1 responds faster to the *GVT* computation request. Table 2-1 gives the relative speed of the two algorithms that were obtained by several runs of the two for different simulation tests. The column numbers are the number of processors involved in the test. The other entries (relative speed) r is given by

$$r = \frac{a-b}{a}$$

where a and b are the average response times of GVT2 and GVT1 respectively. The tests were done on the simulation of models of the Game of Life for several board sizes. The details of the simulation and the implementations appear in Chapter 6.

Table 2-1. The Relative Response Time of the Two algorithms.

	2	3	4	5
test 1	0.44			
test 2	0.46			
test 3	0.59			
test 4	0.20	0.40	0.54	0.10
test 5	0.37	0.40	0.47	0.21

2.3. Optimal Rollback Order

We have mentioned that a single rollback can propagate among the processors and cause some others to rollback too. One processor may need to rollback more than once if the cancellation messages are not received in the proper order. Figure 1-2 in the previous chapter shows this effect.

Considering that each rollback involves some processing time and may involve transmission of several messages to cancel obsolete events, it is desirable to reduce the number of rollbacks to one for each affected processor. The following algorithm finds the ***optimal rollback order,*** in which each processor rolls back directly to its final rollback point. For example, in Figure 1-2, if processor 4 receives cancellation message for e_2 before any other messages, it can roll back directly to c_1, without performing three rollbacks to c_7, c_5 and c_1.

In this section, we study the Rollback method in an environment in which it is practical to introduce a global coordinator. The simulation process remains distributed and the coordinator supervises the rollback operations among all affected processors when and only when a rollback is necessary. The coordinator receives enough information about inter-submodel events to detect the occurrence of a rollback and coordinate the rollback operations. It finds affected processors and reports ***rollback times*** to them. In other words, the coordinator finds the optimal order in which the processors need to roll back to avoid further rollbacks. This way, the problem that was mentioned with processor 4 in Figure 1-2 does not occur. Note that some affected processors may have not simulated the obsolete event(s). In that case the term ***rollback time*** refers to the earliest time the processor has received an obsolete event from another affected processor and all those events received after that time must be removed from the event queue.

The coordinator does not have any information about the object assignment. To the coordinator, an interaction takes place between two processors rather than two objects and processors rollback rather than objects. Clearly, by supplying more information to the coordinator, it is possible to track the times to which objects need to roll back.

First, we view the problem of finding the affected processors and the rollback times as a graph problem. Let **G** be the ***simulation history graph*** if it is a labeled multigraph, with K vertices as the processors. A directed edge *(p,q)* exists in **G** iff *p* has transmitted an event message to *q*. The edge is labeled with the simulation time of that event message. This multigraph is different from the communication graph defined in Section 1.2. In this graph, edges develop as simulation progresses. Given that a processor *rproc* needs to roll back to s_r, the optimal rollback order can be obtained by the following algorithm.

Algorithm rollback order (a)

Input:**G** - a simulation history graph, *rproc* - a processor id, s_r - simulation time to which *rproc* should be rolled back.

Output:a list of affected processors and their rollback times.

```
begin
    construct a marked labeled multigraph H by removing from G all edges
    with labels less than $s_r$ and subsequently removing the subgraphs
    disconnected from rproc;
    for $i=1$ to $K-1$ do
        for every acyclic directed path P in H of length $i$ from iproc to $q$
        such that the labels of the edges of P are in increasing order do
        begin
            add an edge (rproc,$q$) to the graph and label the edge with the
            label of last edge of P;
            mark the new edge "essential";
            if another "essential" edge ingoing to $q$ exists then
                remove the "essential" edge with the higher label;
        end;
    $L = \{ (q,l) \mid q$ is a vertex in H , $l$ is the label of the "essential" edge of $q\}$
    return $L$;
end;
```

Note that *q* does not need to rollback if it has not advanced to that time. It only needs to remove obsolete messages that were sent by those processors that have rolled back.

Each directed path that resulted in the addition of an "essential" edge, corresponds to a sequence of event messages, ordered in increasing simulation time. The path origin, *rproc*, has invalidated the transmission of the first event message in the sequence by rolling back to a simulation time prior to the event's time stamp. Clearly all other messages that were subsequently sent are also obsolete and need to be canceled. Vertices along the path are affected by as much as the label of their incoming edge on P. The edge with the lowest label gives the earliest time the processor corresponding to that vertex is affected.

In what follows we show that the algorithm correctly identifies all affected processors and their rollback times.

Proof by contradiction

Assume that the algorithm is not correct. In that case either (i) there is at least one affected processor that is not identified or (ii) there is at least one affected processor with an incorrectly computed rollback time.

Case (i) is not possible. Note that any affected processor lies on a directed acyclic path with increasing labels from *rproc*. (This path is of length $\leq K-1$ since otherwise it would contain a cycle.) Since the algorithm adds an edge to the graph for every such path and only removes one when multiple identical destination processors exist, there must remain an "essential" edge from *rproc* to every affected processor. Hence all affected processors are identified.

Considering case (ii), let p be an affected processor with an actual rollback time s where the computed rollback time s' is not correct. i.e., $s \neq s'$. Consider all directed acyclic paths from *rproc* to p. The actual and computed rollback times, correspond to different paths with different "essential" edge labels. However, this is not possible. The algorithm searches all possible such paths and selects one that has the smallest label, s' for its corresponding "essential" edge. If s is the actual rollback time, it must be the smallest such label. Hence, the two values must be equal. \

2.3.1. Implementation Details

The above method for finding the directed paths distributively for each occurrence of a rollback can be costly and inefficient. Therefore, we require that the relevant information be available to a processor which we called the coordinator. In addition, the communication medium must preserve the FCFS property.

Every time that a processor sends an event message to another processor, it also sends a message to the coordinator, carrying a tuple *(sp,dp,st)* where

$sp =$ the processor sending the event

$dp =$ the processor receiving the event

$st =$ simulation time of the event

The coordinator maintains a data structure that represents the graph in tabular form. For each processor p, the coordinator has a table of the form shown in Figure 2-3.

The first row gives the simulation time of event messages in the order they have been transmitted, i.e. one column for each outgoing edge of p. The entry corresponding to row p_i and column s_j, (p_i,s_j), gives the simulation time of the next transmitted event from p to p_i after s_j. In other words, for all i $(p_i,s_j) \geq s_j$ or $(p_i,s_j)=NULL$. The former condition indicates that no event has been transmitted from p to p_i after s_j. The coordinator can build this table as simulation progresses. With any message transmission from p_i to p_j with

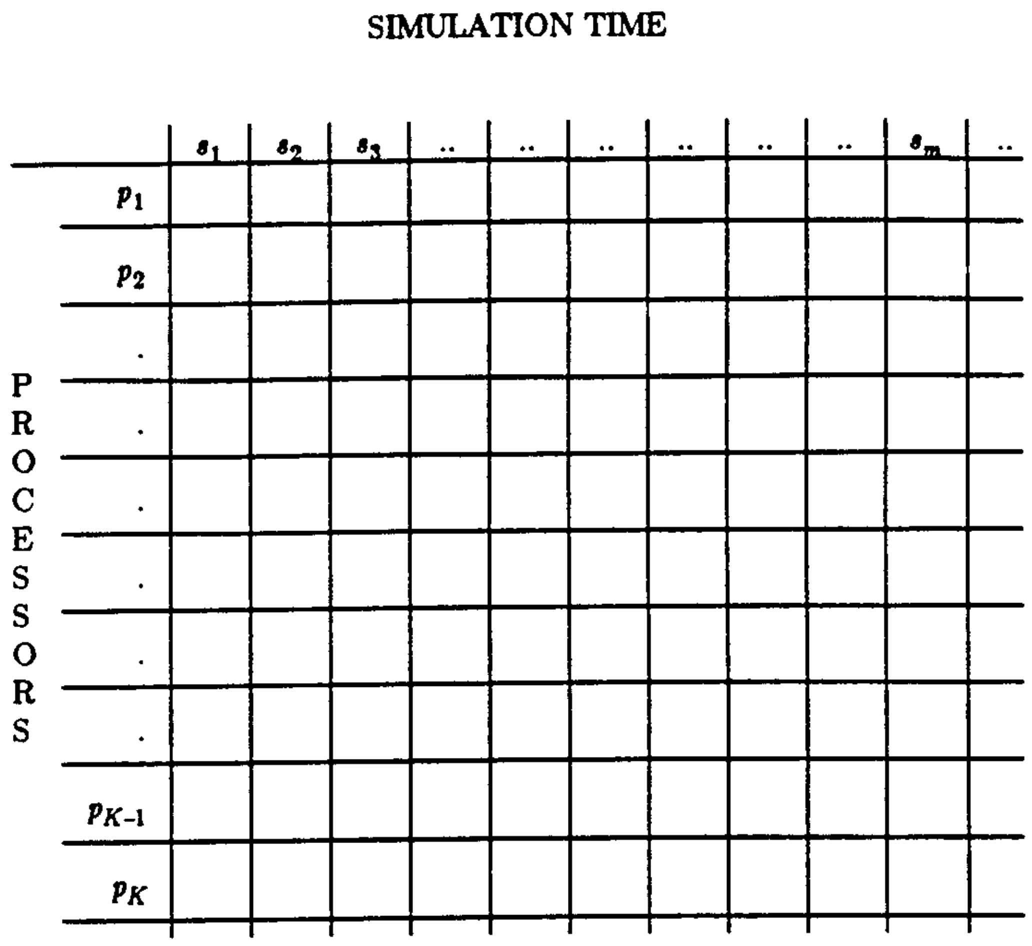

Figure 2-3. The Table for Processor p.

simulation time s, the coordinator adds a new column headed s to table of p_i. It then enters s in the entry (p_j,s), and makes all other null entries of the p_j-th row equal to s. Columns can be removed from this tables as new values for GVT are computed and also when obsolete event messages are removed.

As far as the coordinator is concerned, the simulation time of a processor is the minimum of the simulation time of its last transmitted event and last received event that caused rollback. This time, which is a *pseudo simulation time*, is a lower bound on the actual simulation time of the processor. The coordinator detects the need for a rollback if a processor receives an event with a time stamp lower than the processor's pseudo simulation time. Note that a processor may receive an event in its simulation past, although this is not detectable by the coordinator. This can occur if the processor receives an event message with with simulation time between its pseudo and actual simulation times. This event message will necessitate a rollback however, the latter type of rollback does not affect any other processor and does not produce any inconsistency in the operations.

Once the coordinator detects occurrence of a rollback, it can find the optimal rollback order with the following algorithm. Then, it can broadcast the results to all other processors to update their information. While the algorithm is being executed, the processor *rproc* that is the first rolled back processor has to wait for the results.

Algorithm Rollback Order (b)

Input : *p*, *rproc* processors; s_r, a time stamp on an event message from *p* to *rproc*, causing it to roll back to time s_r.

Output : A list (rollbacklist) of tuples *(p,q,s)*, where *p* is the processor causing a rollback, *q* is the rolled back processor and *s* is the simulation time of the event that caused rollback.

```
begin
(1)  templist = {($p$,$rproc$,$s_r$)} ;
     /* This list at any time contains information about the latest point each
     processor needs to roll back to. Initially, the list contains the first
     processor that has been rolled back and its time of rollback. The list is
     sorted in the simulation time order and at most one tuple exists for each
     processor */
```

```
(2)  while templist is not empty do
     begin
(3)      remove first entry $(p,q,s)$ from templist;
(4)      add $(p,q,s)$ to rollbacklist;
(5)      find column $s_0$ in table of $q$ with smallest $s_0$, such that $s_0 \geq s$;
(6)      for each non-null entry $(p_i,s_i)$ in column $s_0$ do
(7)          if $p_i$ is not in the rollbacklist then
             begin
(8)              insert $(q,p_i,s_i)$ in templist ;
             end;
(9)      remove entries from rollbacklist with duplicate destination processors
             leaving one with smallest simulation time;
     end;
(10) return rollbacklist ;
end algorithm rollback order;
```

2.3.2. Algorithm Correctness and Complexity

The algorithm of the previous section follows the graph theoretical idea explained earlier. As it advances it traverses the edges of the directed paths. Note that each time through the while-loop in step 2, the new edge added (if any) has a label greater than the last entry added to the rollbacklist. Therefore, the strict sequencing of labels in increasing order is maintained. In that case all the traversed paths result in an "essential" edge. On line 9, if there is more than one path ingoing to the same vertex, the one that occurs earlier in simulation time is considered. Therefore, during the i-th run of the while-loop, the templist contains the last edge of all the effective paths of length $\leq i$. Since there are at most K-1 processors that can be affected and one is found each time through the loop, templist contains at most K-i elements at the beginning of the i-th run of the while-loop 2.

The following is a worst case timing analysis of the algorithm :

The time dominant statements are those in the while-loop. During the i-th run, statement 3 has a complexity O(1), since the list is sorted. Similarly, statement 4 is of order O(1). Statement 5 is of order $O(\log X)$, where X is the maximum number of columns found in the tables after the column indicating the rollback time. Note that this is the same as the maximum number of obsolete events that were generated by one processor. Statement 6 requires a sequential search through all the entries of a column and is then O(K).

Statement 7 corresponds to another sequential search through a list of i elements and hence is of $O(i)$. Statement 8 is an insert in a sorted list of size at most $K\text{-}i$. Finally statement 9 is a sequential search through a list of $K\text{-}i$ elements. Loop 2 is executed at most $K\text{-}1$ times, and hence the overall algorithm complexity is

$$O(K^3 + K\log X)$$

where

$$K = \textit{the number of affected processors}$$

and

$$X = \textit{the maximum number of obsolete events sent by a single processor}$$

The computational complexity obtained above seems to be rather high, particularly when K is large. In fact, it is in such a situation that an optimal rollback order can be useful. When K is large, the number of affected processors can also increase and multiple rollbacks become more probable.

To see this, let

$$n(k) = \textit{the number of rollbacks needed in the worst case when } k \textit{ processors are affected by a rollback}$$

Then, $n(k)$ satisfies the following equation

$$n(k) = k \times n(k\text{-}1) + k$$

The above formula is derived as follows. Let processor *rproc* roll back and assume its rollback affects k other processors. In the worst case, all these processors have simulated the obsolete events that were sent by *rproc* and have produced other obsolete events. Let events $e_1, e_2, \ldots, e_k$ be the earlier obsolete events (in simulation time) that were sent by *rproc* to the k processors. Without loss of generality let the simulation time of these events be in the order of their indices and be denoted by $s_1, s_2, \ldots, s_k$. Suppose processor k that has received the event e_k (the latest in simulation time), receives the cancellation message first (in real time). It needs to undo everything since s_k. In the worst case, k may roll back k-1 processors which takes $n(k$-$1)$ rollbacks. Note that we are excluding *rproc,* although this processor may also be affected if it advances to the time of the obsolete messages sent by processor k.

Consider the case where any other processor i receives the cancellation for message e_i after (in real time) e_{i+1} was received by processor $i+1$ and all the obsolete events it had transmitted after e_{i+1} are canceled and affected processors are rolled back. Hence, each processor i of the k processors needs to rollback once to the time e_i and each can affect k-1 others. The formula then follows.

Given the above formula, $n(k)$ is of order $k!$. Given K Processors, one causes a rollback, one is the rolling back processor and in the worst case, the remaining $(K$-$2)$ may have to rollback. Then $O((K-2)!)$ rollbacks may be required until the rollback operation is complete if the cancellation messages are not received in the proper order.

Comparing the two worst case results, one for the computational complexity of the algorithm and the other for the possible number of rollbacks in the case of absence of such an algorithm, encourages an alternative approach to the control of the rollback operation such as the algorithm and structure suggested here.

It should be noted that O(k!) rollbacks require at least O(k!) obsolete messages. The worst case is only one of the permutations of these many messages. The probability of occurrence of such a case can be very low. Nevertheless, other undesirable cases, although not as bad, can still arise with higher probabilities. It should also be noted that the the computational cost of the algorithm does not include the overhead due to the table construction and message transmission to gather the data.

The feasibility of such an algorithm depends to a great extent on the broadcast mechanism. The information about external events needs to be broadcast quickly and efficiently. The coordinator can become a bottleneck if the rate of external events is high.

CHAPTER 3

Assignment of Objects to Processors

The problem discussed in this chapter is that of assignment of objects to processors. The objective is to improve performance and increase speed of simulation by an optimal distribution of objects among the processors. It is assumed that the simulation model has been divided into objects and the objects can be run efficiently on the processors.

Following the discussion of the previous chapter, the effective simulation time of every processor at any time *t*, is the value of *GVT* at time *t*. Therefore, faster simulation implies faster growing *GVT*. Factors affecting the rate of growth of *GVT* are:

(1) The distributed processing overhead. This includes synchronization and message processing.

(2) The overhead due to Time Warp simulation. This consists of *GVT* computation, checkpointing, rollback processing and finally amount of work that is lost as a result of rollbacks.

(3) The degree of concurrency that exists between the processors performing the simulation.

The last two factors are Time Warp-related and will be studied in order to obtain optimal object-processor configurations. The objective is then to assign objects to processors in such a way that the Time Warp overhead is reduced and concurrency is increased. In some cases, the best object assignment strategy is a dynamic one, flexible enough to respond quickly to the changes in the system. However, throughout this work, only the problem of static assignment of objects to the processors is studied. The results can be used for an initial object assignment and then complemented by a dynamic load balancing technique.

The types of Time Warp overhead that can particularly be affected by an assignment policy are the rollback processing time and the wasted simulation time. These can be reduced by reducing the number of rollbacks. Note that a rollback can occur by transmission of an event from an object with lower simulation time to one that is farther ahead. However, this is only a necessary condition. If an object is allowed to transmit events with time stamps greater than its own simulation time, receipt of an event from a slower object may not cause a rollback.

Depending on the simulation model, objects differ in their ability to cause rollback or their ability to be forced to rollback. Figure 3-1 shows the communication graph of a model with different types of objects as far as rollback is concerned. Some objects can be subject to rolling back and also cause others to rollback (nodes 3 and 4), some may need to rollback but cannot cause others to do so (node 5), some can cause rollback but cannot be subject to rollback (node 2). Note that, node 2 is only simulating events that are generated by node 1. Finally those that cannot do either (node 1).

In this work, we make no distinction among the different types of objects mentioned. That is, we assume that all objects can be subject to rolling back and can also cause others to rollback.

Objects on the same processor are simulated sequentially and cannot cause each other to rollback as long as no inter-processor event transmission occurs. Therefore, transmission of an event message between two processors not only increases the overall communication cost but also increases probability of rollback and the rollback processing overhead. Hence the communication cost between two objects is assumed to be the interaction rate or the number of event messages transmitted between the two per unit simulation time. Clearly, this measure is reasonable when the simulation model is well-behaved. We assume the system is stable enough that such an

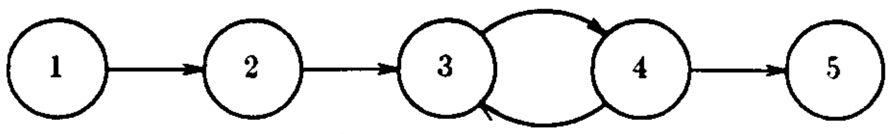

Figure 3-1. Objects with Different Rollback Capabilities

average approximates the actual interaction between objects during any unit simulation time interval.

Consider the communication graph of the simulation model. Label each edge *(i,j)* with c_{ij}, the average number of events transmitted between i and j per unit simulation time. To reduce inter-processor communication and therefore reduce possibility of rollbacks, we need to divide the nodes of this graph into fragments such that the sum of the inter fragment link labels is minimized. But first, we need the values for c_{ij}. How these values are obtained, will be discussed next. Therefore, the problem of object assignment consists of two subproblems:

(i) Given matrix $C = \{c_{ij}\}$, find an optimal assignment of objects, satisfying the requirements of lower rollback processing time and less wasted simulation time. Clearly, the best object assignment policy in this case is to assign all objects to one processor. However, in addition to the above, we like to maintain the highest level of concurrency.

(ii) Find matrix C.

In this problem, it is assumed that the processors to which the objects are assigned have equal processing powers. Furthermore, the cost of communications between every two processors in terms of delay or communication processing overhead are the same. In that case, by reducing overall traffic, the overall cost of communications in the simulator is reduced.

3.1. Optimum Object-Processor Assignment Given C

To satisfy the requirements of lower rollback processing and less wasted simulation time, we need to allocate objects on the processing elements such that the amount of communications between objects on different processors is as small as possible. As mentioned, this means placing more frequently communicating objects on the same processor. Not considering other objectives, the best we can do is to allocate all objects on one processor and therefore, reduce the inter-processor communications to zero.

To deal with the other requirement of reducing wasted simulation time, we need to balance the simulation time of the objects as much as possible. Since no assumption is made on the connectivity of the communication graph, we wish to minimize

$$| \, LVT_i(t) - LVT_j(t) \, | \quad \textit{for all objects } i \textit{ and } j \textit{ and time } t$$

where $LVT_i(t)$ is the ***simulation time*** or the *Local Virtual Time* of object i at real time t. Looking at the system over a long time period t, we have

$$LVT_i(t) = t\gamma_i$$

where γ_i is the simulation rate of object i, i.e. unit simulation time progress per unit real time.

We therefore, wish to minimize the difference in simulation rates of the objects. But the simulation rate of an object i on processor k is equal to the simulation rate of the processor and is

$$\Gamma_k = \frac{\mu_k}{\sum_{i=1}^{N} \lambda_i \beta_{ik}}$$

where

$$\frac{1}{\mu_k} = \textit{average real time processing of processor k per simulation event}$$

$$\frac{1}{\lambda_i} = \textit{average simulation time per event of object i}$$

$$N = \textit{total number of objects}$$

$$\beta_{ik} = 1 \quad \textit{if object i is assigned to processor k}$$

$$= 0 \quad \textit{otherwise}$$

Note that in our study an object can only be assigned to one processor. This may not be true in general when multiple processors are responsible for each object for fault tolerance. This issue is beyond our discussion.

We have

$$\frac{\mu_k}{\lambda_i} = \textit{the rate at which simulation time of object i grows on processor k}$$

$$\textit{in the absence of rollback}$$

The simulation rate of processor k is inversely proportional to $\sum_{i=1}^{N} \frac{\lambda_i}{\mu_k}\beta_{ik}$. To minimize the difference in simulation rates of the processors and therefore minimize wasted simulation time, we need to minimize

$$\left| \frac{1}{\sum_{i=1}^{N} \frac{\lambda_i}{\mu_k}\beta_{ik}} - \frac{1}{\sum_{i=1}^{N} \frac{\lambda_i}{\mu_l}\beta_{il}} \right| \quad \textit{for all pairs } k,l$$

Until this point, our conclusion is that to improve performance, we need to minimize inter-processor communications and the difference in the simulation time of the processors. Both these objectives are best met in a single-processor environment. To optimize multi-processor simulation, we need to deal with concurrency requirements. Given K, the maximum number of available processors, the ideal simulation rate is

$$\Gamma_{ideal} = \frac{K\mu}{\lambda}$$

where $\lambda = \sum_{i=1}^{N} \lambda_i$ and μ is the processing rate per event of the processor in a single processor situation. The ideal simulation rate is hard to achieve. We therefore define τ to be the degree of tolerance we choose to be away from the ideal rate. The simulation rate constraints of the processors can be rewritten as,

$$\Gamma_{ideal} - \tau \leq \frac{\mu_k}{\sum_{i=1}^{N} \lambda_i \beta_{ik}} \leq \Gamma_{ideal} + \tau \qquad \textit{for all } k \leq K$$

The smaller τ, the closer the objects are in simulation time.

Summarizing the above, the problem is

$$Maximize \sum_{k=1}^{K} \sum_{i=1}^{N} \sum_{j<i} c_{ij}\beta_{ik}\beta_{jk} \tag{1}$$

$$subject\ to\ \Gamma_{ideal} - \tau \leq \frac{\mu}{\sum_{i=1}^{N} \lambda_i \beta_{ik}} \Gamma_{ideal} + \tau \qquad for\ all\ k<=K \tag{2}$$

where the variables $\beta_{ik} \in \{0,1\}$ and for any i, $\sum_{k=1}^{K} \beta_{ik} = 1$.

This a quadratic 0-1 integer programming problem with linear constraints [GYE 76], [MLT 82]. The objective function (1), minimizes the inter-processor communications. This is done when the inter-object communications within a processor is maximized. The constraints (2), minimize the difference in the average simulation time of the processors and bound this difference by 2τ. Solving this optimization problem by integer-programming methods is in general time consuming [KAM 76], [HAN 79]. In fact, the problem of finding a feasible solution to meet (2) is NP-complete [GAJ 79]. The initial objective of faster simulation is then, not achievable if too much processing time is spent on object allocation. Therefore, a faster heuristic that if not optimal gives a reasonable feasible solution is desirable.

The first question is the existence of a feasible solution for this problem. Clearly, this depends on τ. In the algorithms below, an initial value for τ is given to be an arbitrary percentage of the defined measure for ideal simulation rate. This value is then increased, if no feasible solution can be obtained. It will be shown later that a better initial value for τ can be obtained by running the last algorithm presented below. Another approach is to perform a binary search in the range of the values of τ. That is, when an initial value proves to be infeasible, then a rather large value of τ is tested for feasibility. Once a feasible value is obtained, with a binary search approach the range between feasible and infeasible values is tightened until a satisfactory result is obtained.

The number of available processors is an input to the following algorithms. In the first two algorithms, it is possible to obtain a result with smaller number of available processors. However, the last algorithm only produces object-processor configuration that uses the maximum number available.

3.1.1. Heuristic Algorithm 1

The algorithm presented here, considers the primal form of the optimization problem (1). The algorithm groups objects into fragments in such a way that the inter-object communications within a fragment is maximized. In the process of grouping the objects, the constraints in (2) are checked for satisfaction. The approach is similar to the minimal spanning tree algorithm of Kruskal, [KRU 56].

In order to optimize on the computational complexity of this algorithm, the Union-Find algorithm of Tarjan [AHU 75] was used. The Union-Find algorithm finds the two sets containing two elements and performs a union on the two in the least costly way.

The algorithm gives the optimal solution when the communication graph corresponds to the graph of an equivalence relation with the sum of the loads assigned to the object of each class are as close as possible.

```
begin Algorithm H1
done = false ;
sort the links of the graph in decreasing order of cost, $c_{ij}$ ;
while not done do begin
    for $n=1$ to $N$ do fragment($n$) = object $n$ ;
    while all links are not considered do begin
        remove the largest link $(i,j)$ from the sorted list ;
        if (fragment of $i \neq$ fragment of $j$) and
        ((fragment of $i \cup$ fragment of $j$) does meet the upper bound in (2))
        then /* fragment of $i$ = the fragment containing object $i$ */
            fragment of $i$ = fragment of $i \cup$ fragment of $j$ ;
            fragment of $j$ = NULL ;
    end
    if number of non-null fragments $> K$ then
        combine fragments such that the bounds in (2) are satisfied ;
    if number of non-empty fragments $\leq K$ then
        done = true ;
    else increase $\tau$ or perform a binary search;
    /* At this point, the grouping of objects has not been successful. The
    constraints (2) are slightly perturbed and a new run of the algorithm is
    resumed */
end
end Algorithm H1
```

3.1.2. Heuristic Algorithm 2

This algorithm divides the communication graph of the objects into two subgraphs (fragments) in such a way that the inter-fragment communication is minimized. Each subgraph that does not meet the upper bound in (2) is then treated as a graph and the division process continues until all fragments meet the upper bound in (2). The lower bound constraint is then checked and the fragments with low loads are combined to meet the lower bound. Similar to the previous algorithm, if the process does not succeed with a particular value for r, it is increased and the process is repeated. To divide the graph into two fragments with minimum inter-fragment communication, the mincut-maxflow algorithm is used [EVE 79], [EDK 72], [DIN 70] and [FOF 62]. In fact, the optimization problem (1) without constraints in (2) and $K=2$ is equivalent to the maxflow-mincut problem [PIR73a], [PIR73b], [PIR 74], [RSH 79] and [STO 77]. Note that if the objects on the source side are assigned to one processor and those on the sink side are assigned to the other, the communication cost between two processors is minimized.

Similar to the previous algorithm, the resulting configuration is only bounded above by K, the maximum number of available processors; it does not guarantee all K processors will be utilized.

```
begin Algorithm H2
done = false ;
while not done do
    begin
    fragments = { graph } ;
    if there is a fragment X exceeding the upper bound on load then
        begin
        use min-cut algorithm to divide X into two fragments Y,Z ;
        fragments = fragments - X + Y + Z ;
        end
    if number of fragments $\leq K$ then done = true ;
    else
        begin
        combine the fragments such that total number is reduced and the
        upper bound requirements are met;
        if number of fragments $> K$ then
            increase $\tau$ or perform a binary search;
        end
end Algorithm H2
```

3.1.3. Heuristic Algorithm 3

This algorithm works on the constraints (2) of the optimization problem. It first divides the objects into K fragments such that the load is evenly distributed. It then tries to exchange objects in different partitions so as to minimize the communication cost.

The initial problem is then to divide a set of N elements ($N=$ number of objects) into K sets in such a way that the difference in the sum of the elements of every pair of sets is minimized. That is, given a set $(\lambda_1, \lambda_2, \cdots, \lambda_N)$, divide the set into K subsets $P_k, k=1,\ldots,K$ such that

$$\left| \sum_{\lambda_i \text{ in } P_k} \lambda_i - \sum_{\lambda_j \text{ in } P_l} \lambda_j \right| \textit{ for all pairs } k \textit{ and } l$$

is minimized. This problem was studied by Karmarker and Karp, [KAK 82]. The approach taken in the following algorithm is to assign the largest element to the set with smallest sum first. The algorithm initially sorts the elements of the set. It then assigns the first K largest elements to K sets. Starting from largest in the remaining elements, elements are added one by one to the set with lowest sum. The idea is to increase the sum of the elements in the smallest set by the largest number possible.

Unlike the first two algorithms, this algorithm is constrained by the number of available processors. If all different configurations are needed, we need to run the algorithm for different values of K.

This algorithm could be used as a pre-processing test for the first two algorithms. The result can tell us whether it is at all possible to meet the constraints in (2). It is possible that even before the *exchange* operation in the Algorithm 3., the sets do not meet the bounds in (2). This implies infeasibility of the solution with the choice of r. r should be increased until a feasible solution is obtained. Clearly this algorithm does not guarantee an optimal solution for r.

```
begin Algorithm H3
sort the loads ($\lambda_i$) in decreasing order;
assign the first $K$ to $K$ sets ($S_1,S_2,...,S_K$);
while all objects are not considered do
    begin
    take next element;
    add element to the set $S_K$;
    sort the sets in decreasing order;
    end;
/* At this point the objects are partitioned into sets in such a way that the
sum of the loads of objects in each partition is close to each other. */
do until no improvements can be made or time is over
    exchange objects such that the inter-fragment communication cost is
    reduced and the load constraint (2) is not disturbed ;
end Algorithm H3
```

3.2. Approximate Evaluation of C

The second part of the object assignment problem is to evaluate the matrix $C = \{c_{ij}\}$, where

c_{ij} = *average number of events transmitted between i and j*

per unit simulation time.

These values can be obtained in several ways. One approach is to perform a flow analysis of the *behavior module*. This is the module that is activated when the object is being simulated. The data can also be obtained from the user or a short sample run of the simulation.

Each data acquisition method by itself, may provide a poor estimate of the data. Therefore, an approach that combines all the three methods has been chosen.

Initially, a flow analysis of the behavior module is performed. The data obtained by the flow analysis of the behavior module of object i, consists of

(i) The destination objects of the generated events by i. In the case where the flow analysis does not produce an explicit result, the destination is assumed to be the set of all possible neighbors of the object. This set can be provided by the user or a short sample run of the simulation.

(ii) The average number of transmitted events to any destination, i.e. N_{ij} = Number of events from i to j per unit simulation time. Similarly, some of these values may not be explicitly derived from the flow analysis. In this case, user or the sample run data can be used as an estimate.

(iii) Average simulation time advance of the object for each behavior module activation, s_i.

Having this information, the communication graph of the object is constructed. This graph is further modified as follows to obtain an open queuing network model. Let

$$N'_{ij} = \frac{N_{ij}}{s_i} \quad \textit{for all } i \textit{ and } j$$

If

$$\sum_{\text{all } j} N'_{ij} > \sum_{\text{all } j} N'_{ji}$$

then create a source node *source* and let *(source,i)* have the label

$$N'_{0i} = \sum_{\text{all } j} N'_{ij} - \sum_{\text{all } j} N'_{ji}$$

Otherwise, if

$$\sum_{\text{all } j} N'_{ij} < \sum_{\text{all } j} N'_{ji}$$

then create a node *sink* and label the link *(i,sink)*, with the value

$$N'_{i0} = \sum_{\text{all } j} N'_{ji} - \sum_{\text{all } j} N'_{ij}$$

Figure 3-2 shows the ingoing and outgoing links of object i with the labels mentioned.

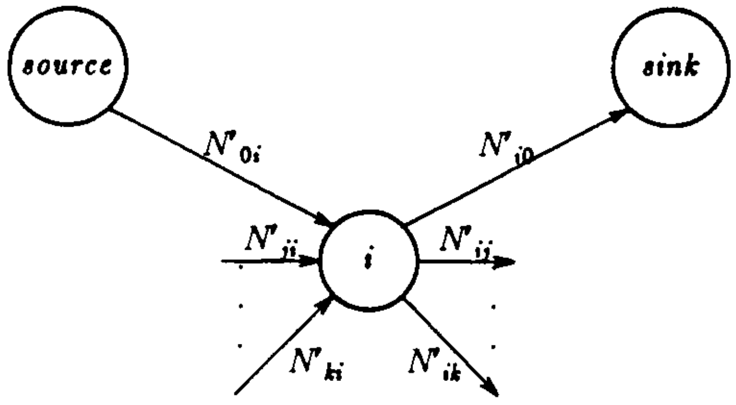

Figure 3-2. Object i with the Incident Links.

Let also

$$Total_i = \sum_{j=0}^{K} N'_{ij}$$

Then, we define

$$r_{ij} = \textit{probability that receipt of an event message by } i \textit{ will result in generation of an event to } j$$

$$= \frac{N'_{ij}}{Total_i}$$

Now assuming that the arrivals from *source* to any object i forms a poisson process with rate r_{0i}, we can use the following equations [KLE 75], to obtain λ_i, the rate of simulation event arrivals to object i.

$$\lambda_i = r_{0i} + \sum_{j=1}^{K} \lambda_j r_{ji} \quad \textit{for all } i$$

Having computed λ_i's, then

$$w_{ij} = \textit{number of events from } i \textit{ to } j \textit{ per unit simulation time}$$

$$= \lambda_i r_{ij} \qquad \textit{for all } i \textit{ and } j$$

and finally,

$$c_{ij} = w_{ij} + w_{ji}$$

3.3. Implementation Results

The three heuristic algorithms of this chapter were programmed and tested on different test cases. The object-processor assignment results for variable number of processors of two test cases appear in this section.

Case 1, is a graph model, shown in Figure 3-3. The labels on the edges give the values c_{ij}.

The *load* of an object is considered to be the total number of event messages received and sent by that object per unit simulation time. i.e., $\sum_{for\ all\ j} c_{ij}$

The results of running the programs for the algorithms H1 and H3 for 2-5 processors appear in tables 3-1 to 3-8. In these tables K denotes the number of available processors, *Total elapsed time* is the cpu time in milliseconds to

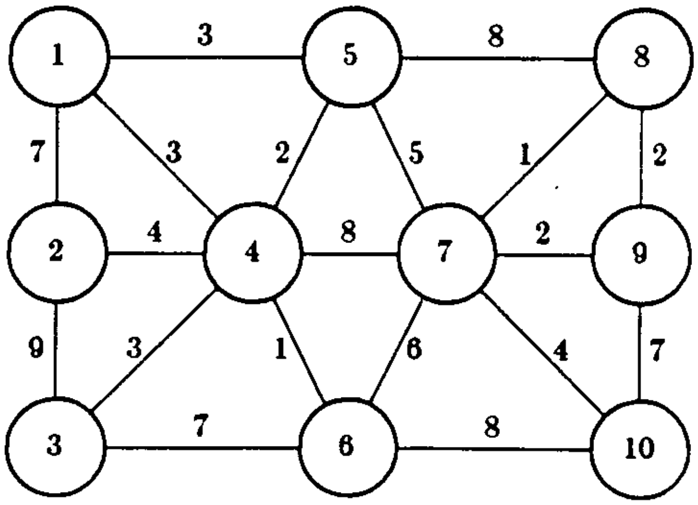

Figure 3-3. The Communication Graph of Test Case 1.

obtain the results, *Number of iterations* is the number of times τ had to be incremented for a feasible solution, *Max (Min) load* is the allowed upper (lower) bound on the load given by the constraints in (2), *Cost matrix* gives the average number of event messages transmitted between two processors per unit simulation time, $p_\tau = \frac{\tau}{\Gamma_{ideal}}$ denotes the percentage that the average simulation rate is away from the ideal rate. Finally

$$p = \textit{interaction rate between processors}$$
$$= \frac{\sum_{for\ all\ i,j,k,l} c_{ij}(1-\beta_{ik})(1-\beta_{jl})}{\sum_{for\ all\ i,j} c_{ij}}$$

where β_{ik} is defined as before.

Tables 3-1 and 3-2 show that H1 and H3 produced the same results when $K=2$. In other cases, i.e. when $K>2$, the results obtained were different. H1's running time is considerably lower than H3. H3 shows to spend a high fraction of time on the part where objects on different processors are exchanged to produce lower interaction rates. H1 produces results that give lower inter-processor communication but have a wider gap in the simulation rates of processors. The results obtained by both algorithms show that the cases of $K>2$ for this model are rather undesirable cases for the distributed simulation. The interaction rates between the processors are high. This is due to the fact that the communication graph cannot be divided into more than two fragments in such a way that the resulting fragments are still weakly interacting.

For this test case, algorithm H2 tends to create a large number of fragments, each with one or two nodes. This is because the mincut in the graph is one that separates one node from the graph. Subsequently, the program spend considerable time in the process of combining these small fragments into larger ones. The results that were obtained are superior only to an arbitrary assignment policy.

Table 3-1. Results of H1, K=2, Case 1.

K=2, Total elapsed time = 3
Number of iterations = 1, Max load = 99, Min load = 81

Processor	Objects	Total Load	Cost Matrix	
p_1	1,4,5,7,8	89	60	29
p_2	2,3,6,9,10	91	29	62

p = 0.32, p_r = 0.02

Table 3-2. Results of H3, K=2, Case 1.

K=2, Total elapsed time = 15
Number of iterations = 1, Max load = 99, Min load = 81

Processor	Objects	Total Load	Cost Matrix	
p_1	1,4,5,7,8	89	60	29
p_2	2,3,6,9,10	91	29	62

p = 0.32, p_r = 0.02

Table 3-3. Results of H1, K=3, Case 1.

Number of processors = 3, Total elapsed time = 6
Number of iterations = 3, Max load = 78, Min load = 48

Processor	Objects	Total load	Cost matrix		
p_1	1,2,3	52	32	13	7
p_2	4,5,7,8	76	13	48	15
p_3	6,9,10	52	7	15	30

$p = 0.38$, $p_r = 0.26$

Table 3-4. Results of H3, K=3, Case 1.

Number of processors = 3, Total elapsed time = 12
Number of iterations = 1, Max load = 66, Min load = 54

Processor	Objects	Total load	Cost matrix		
p_1	1,2,4,9	65	28	20	17
p_2	3,6,10	60	20	30	10
p_3	5,7,8	55	17	10	28

$p = 0.54$, $p_r = 0.08$

Table 3-5. Results of H1, K=4, Case 1.

Number of processors = 4, Total elapsed time = 5
Number of iterations = 2, Max load = 54, Min load = 25

Processor	Objects	Total load	Cost matrix			
p_1	1,2,3	52	32	10	3	7
p_2	4,7	47	10	16	8	13
p_3	5,8	29	3	8	16	2
p_4	6,9,10	52	7	13	2	30

$p = 0.47$, $p_r = 0.57$

Table 3-6. Results of H3, K=4, Case 1.

Number of processors = 4, Total elapsed time = 15 Number of iterations = 1, Max load = 50 , Min load = 40						
Processors	Objects	Total load	Cost matrix			
p_1	1,7,9	50	4	11	14	21
p_2	3,4	40	11	6	10	13
p_3	5,6	40	14	10	0	16
p_4	2,8,10	50	21	13	16	0
p = 0.94, p_r = 0.11						

Table 3-7. Results of H1, K=5, Case 1.

Number of processors = 5, Total elapsed time = 5 Number of iterations = 2, Max load = 43.2, Min load = 20.8							
Processor	Objects	Total load	Cost matrix				
p_1	1,5,8	42	22	7	5	0	8
p_2	2,3	39	7	18	7	7	0
p_3	4	21	5	7	0	1	8
p_4	6,10	41	0	7	1	16	17
p_5	7,9	37	8	0	8	17	4
p = 0.66, p_r = 0.41							

Table 3-8. Results for H3.

Number of processors = 5, Total elapsed time = 13 Number of iterations = 1 , Max load = 39.6, Min load = 32.4							
Processor	Objects	Total load	Cost matrix				
p_1	1,2	33	14	10	9	0	0
p_2	4,5	39	10	4	3	1	21
p_3	3,10	38	9	3	0	22	4
p_4	6,9	33	0	1	22	0	10
p_5	7,8	37	0	21	4	10	2
p = 0.88; p_r = 0.01							

The second test is the simulation of a version of the Game of Life, briefly described in Chapter 6. The board size is 8 $\times$ 8. Each square is a simulation object and the communication graph is shown in Figure 3-4. The squares are numbered from 1 to 64 in row major order. In the Figure 3-4, only the objects that interact with others are shown.

The results of the second test case for algorithms H1 and H3 for 2-5 processors appear in tables 3-9 to 3-16. The numbers within each square give the processor number to which the object representing that square is assigned.

Similar to the previous case, for two processors both algorithms H1 and H3 produce the same results. When $K=3$ and $K=5$, H1 divides the graph into fragments in such a way that the inter-processor interaction is lower than the results of H3. However, H3 balances the load more evenly between the processors. When $K=4$, unexpectedly, H3 gives a better result overall.

In all cases, H1's running time is considerably lower than H3. Most of H3's effort is spent in the phase where objects are exchanged to produce looser coupled fragments. It can also be seen that for $K>3$, the coupling is rather tight and the interaction rate is higher than desirable.

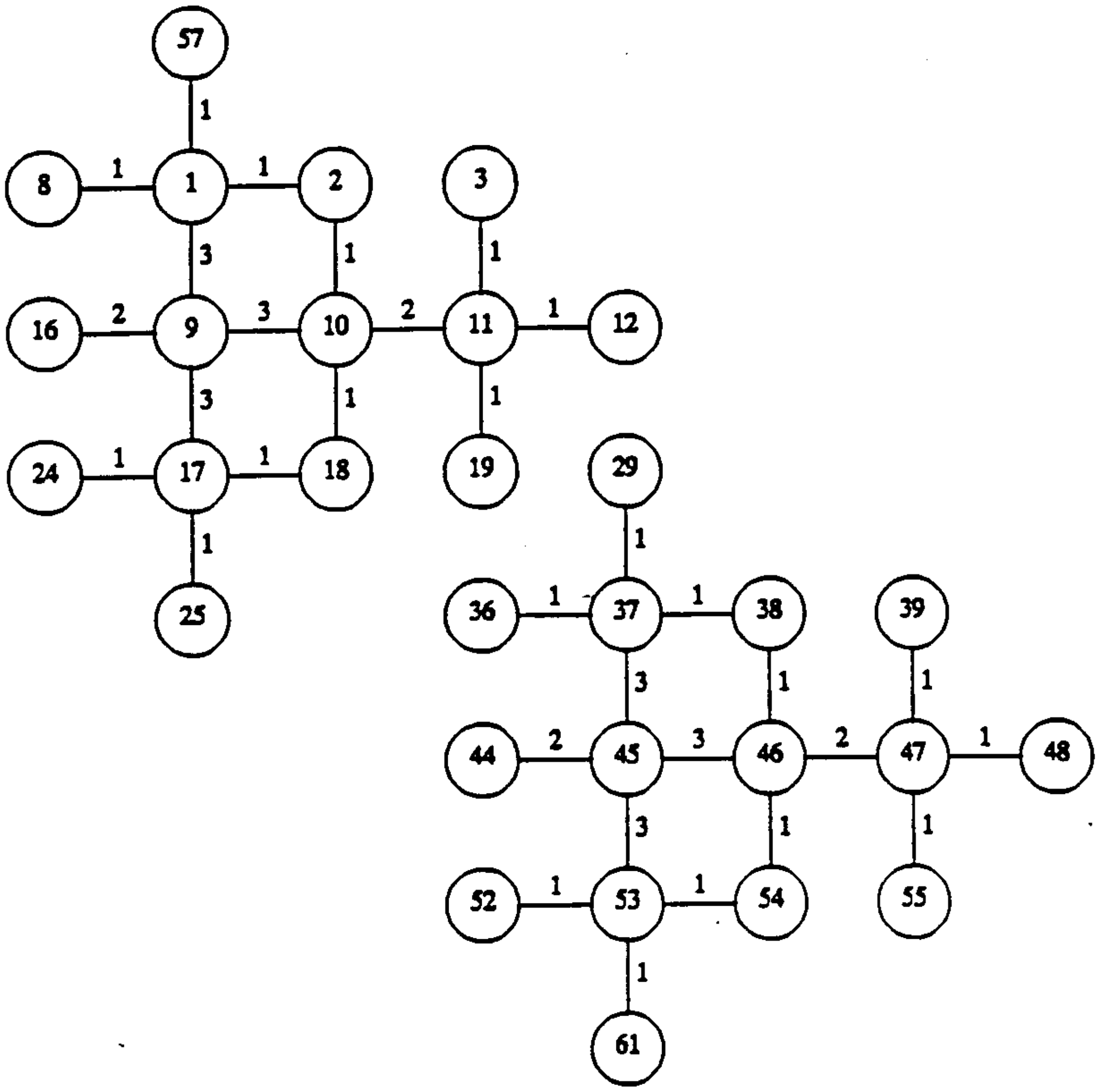

Figure 3-4. The Communication Graph for the Test Case 2.

Table 3-9. Results of H1, K=2, Case 2.

2	2	2	1	2	2	1	2
2	2	2	2	1	2	2	2
2	2	2	1	1	2	2	2
2	1	1	2	1	2	1	1
2	2	1	1	1	1	1	1
2	2	1	1	1	1	1	1
1	2	2	1	1	1	1	1
2	1	2	2	1	1	1	2

Total elapsed time = 21, Number of iterations = 1 Max load = 52.8, Min load = 43.2			
Processor	Total load	Cost	matrix
p_1	48.17	48.17	0
p_2	48.17	0	48.17
p = 0, p_T = 0			

Table 3-10. Results of H3.

2	2	2	2	1	1	2	2
2	2	2	2	2	1	2	2
2	2	2	1	1	2	2	2
2	1	1	2	1	2	1	1
2	2	1	1	1	1	1	1
2	2	1	1	1	1	1	1
1	2	2	1	1	1	1	1
2	1	2	2	1	1	1	2

Total elapsed time = 422, Number of iterations = 1 Max load = 52.8, Min load = 43.2			
Processor	Total load	Cost	matrix
p_1	48.17	48.17	0
p_2	48.17	0	48.17
p = 0, p_T = 0			

Table 3-11. Results of H1, K=3, Case 2.

2	1	1	1	1	1	1	1
2	2	1	1	1	1	1	2
2	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	3	1	1	1
1	1	1	3	3	3	1	1
1	1	1	3	3	1	1	1
2	1	1	1	1	1	1	1

Total elapsed time = 30, Number of iterations = 1 Max load = 35, Min load = 29				
Processor	Total load	Cost	matrix	
p_1	30.34	12	9	9
p_2	33	9	24	0
p_3	33	9	0	24
$p = 0.12, p_r = 0.06$				

Table 3-12. Results of H3, K=3, Case 2.

3	3	1	1	2	2	1	3
3	1	1	1	1	3	2	3
1	1	1	3	3	1	2	1
1	2	3	3	2	1	2	2
1	3	3	2	2	2	3	1
2	2	1	2	2	1	3	3
3	3	1	2	2	3	3	2
3	2	1	3	2	3	1	2

Total elapsed time = 433, Number of iterations = 1 Max load = 35, Min load = 29				
Processor	Total load	Cost	matrix	
p_1	32	18	4	10
p_2	31.17	4	26	1
p_3	33	10	1	22
$p = 0.31, p_r = 0.03$				

Table 3-13. Results of H1, K=4, Case 2.

2	2	1	1	2	2	1	2
3	3	1	1	1	2	2	1
3	2	1	1	1	2	2	2
3	1	1	2	1	2	1	1
2	2	1	1	1	1	2	1
2	2	1	2	4	4	2	2
1	2	2	4	4	1	2	1
2	1	2	2	1	1	1	2

Total elapsed time = 28, Number of iterations = 1 Max load = 26.4, Min load = 21.6					
Processor	Total load	Cost matrix			
p_1	23.17	12	0	4	7
p_2	23.17	0	12	7	4
p_3	25	4	7	14	0
p_4	25	7	4	0	14
$p = 0.45, p_r = 0.04$					

Table 3-14. Results of H3, K=4, Case 2.

1	1	2	2	1	2	3	1
4	2	2	2	2	1	1	4
4	4	2	3	3	4	4	4
4	3	4	1	3	4	3	3
4	4	3	3	3	3	1	4
2	2	1	3	3	1	1	1
3	4	4	2	2	2	1	3
1	1	2	2	2	1	3	4

Total elapsed time = 490, Number of iterations = 1 Max load = 26.4, Min load = 21.6					
Processor	Total load	Cost matrix			
p_1	25	16	2	4	3
p_2	25	2	16	3	4
p_3	23.17	4	3	16	0
p_4	23.17	3	4	0	16
$p = 0.33, p_r = 0.04$					

Table 3-15. Results of H1, K=5, Case 2.

1	1	2	1	1	1	4	1
4	4	2	2	1	1	1	4
2	2	2	1	1	1	1	2
2	1	1	1	1	1	1	1
1	1	1	1	1	3	3	1
1	1	1	5	5	3	3	3
1	1	1	5	5	3	3	1
1	1	1	1	2	1	1	1

Total elapsed time = 26, Number of iterations = 1 Max load = 21, Min load = 17						
Processor	Total load	Cost	matrix			
p_1	18.34	10	0	1	4	3
p_2	19	0	12	0	6	1
p_3	19	1	0	14	0	4
p_4	20	4	6	0	10	0
p_5	20	3	1	4	0	12
$p = 0.5$, $p_\tau = 0.04$						

Table 3-16. Results of H3, K=5, Case 2.

1	1	4	5	3	4	1	1
5	2	4	4	4	1	2	2
2	2	4	3	2	4	4	2
2	3	1	2	3	2	1	3
4	4	3	3	3	3	5	1
2	2	1	1	4	1	5	5
3	4	4	3	3	3	5	3
1	1	2	2	3	1	3	4

Total elapsed time = 399, Number of iterations = 1 Max load = 21, Min load = 17						
Processor	Total load	Cost	matrix			
p_1	19.08	6	1	2	5	5
p_2	19.08	1	8	0	2	8
p_3	20.08	2	0	12	6	0
p_4	19.09	5	2	6	6	0
p_5	19.01	5	8	0	0	6
$p = 0.6$, $p_\tau = 0.04$						

CHAPTER 4

Analysis of the Rollback Method

In this chapter, a mathematical model of the Rollback method for distributed simulation is presented and analyzed. The results of the analysis are then compared with simulation results. Simulations are also performed on less constrained version of the mathematical model. The mathematical analysis is restricted to the case where two processors are performing the simulation and the interaction rate between the processors is low. The case of more than two processors are studied by simulation.

The mathematical model considers the collection of objects assigned to each processor as a single sequential process. In this case, once an object rolls back all objects on that processor need to roll back too. Therefore, throughout this chapter the rollback operation is performed on a processor and not only on one object residing on that processor. This assumption is equivalent to assigning one object to one processor.

4.1. The Performance Model

Given K processors, the simulation model is divided into weakly interacting submodels, each being simulated by one processor. As mentioned, each submodel corresponds to a set of one or more objects that are simulated sequentially. The processors are assumed to be dedicated to simulation, each with a never empty queue of events. We further make the following assumptions :

(1) The simulation time of each submodel is updated instantaneously at the beginning of processing an event.

(2) The real time to initialize each submodel (i.e. the real time until the beginning of processing the first event) and the real time to process an event for each submodel are iid exponential with mean $\frac{1}{\mu}$. Thus, μ is the speed of each processor in events per unit real time.

(3) The simulation time advances are iid exponential with mean $\frac{1}{\lambda_i}$ for submodel i. Thus, λ_i is the simulation rate for submodel i in events per unit simulation time.

(4) Each event for submodel i independently generates an event for another submodel with probability p_i . Any submodel is equally likely to be the receiver of the generated event. Later it will be assumed that $p_i = p$ *for all* $1 \leq i \leq K$.

(5) The real time to transmit a message between two processors is a constant *d*. This constant is initially assumed to be negligible.

(6) The real time to perform rollback is a constant multiple of the average time needed to simulate the wasted simulation time. The constant multiplier, *r*, is initially assumed to be zero.

(7) Checkpointing occurs at the beginning of processing every event and therefore checkpoints are perfect. In addition, this process takes negligible processing time. The assumption of perfect checkpoints can be relaxed in the case where $r>0$. Note that in the case of non-zero *r*, the ***simulate forward*** period can be included in the rollback processing period.

(8) The GVT computation does not interfere with the simulation process.

(9) The real time to process an arriving message is negligible.

(10) Processors do not sent inter-submodel events that belong to their simulation future. i.e. each event is transmitted when the simulation clock of the processor reaches the simulation time of the event. This assumption can be relaxed in the case where the communication delay in the simulator is not zero.

4.2. The Performance Measure

The performance measure is the speedup which is defined to be the ratio of the useful simulation time per unit real time in a K-processor system over the same quantity in a uni-processor system when all processors have the same speed μ (see assumption 2 for the definition of μ). To express the speedup in terms of system parameters, let us first define the following variables :

$$s_i(t) = \textit{simulation time of submodel } i \textit{ at time } t$$

$$\gamma_i = \lim_{t \to \infty} \frac{s_i(t)}{t} \qquad \textit{assuming the limit exists}$$

Then

$$\textit{useful simulation time at real time } t = \min_{1 \le i \le K} (s_i(t)) = GVT(t)$$

Let

$$e = \lim_{t \to \infty} \frac{\textit{useful simulation time at real time } t}{t} .$$

Therefore, for K processors

$$e_{K\text{-}processor} = \min_{1 \le i \le K} (\gamma_i)$$

Note that if there are no intersubmodel events (p_i=0) or no rollbacks, then $\gamma_i = \frac{\mu}{\lambda_i}$. Now if the entire model were simulated on a single processor, then there would be no rollbacks. For this processor the rate of events per unit simulation time is $\sum_{1 \le i \le K} \lambda_i$ and therefore

$$e_{single\ processor} = \frac{\mu}{\sum_{1 \le i \le K} \lambda_i}$$

The speedup is defined as :

$$\rho_K = \frac{e_{K\text{-}processor}}{e_{single\ processor}}$$

When K=2, then

$$\rho = \frac{min(\gamma_1, \gamma_2)}{\frac{\mu}{(\lambda_1 + \lambda_2)}} \tag{1}$$

In the case where $p_1 = p_2 = 0$, then :

$$\rho = (\lambda_1 + \lambda_2)\ \min(1/\lambda_1, 1/\lambda_2)$$

$$= 1 + a \quad when\ \lambda_2 = a\lambda_1 \ \ , a \le 1$$

Clearly this quantity is maximized when $\lambda_1 = \lambda_2$, i.e. the submodels have identical simulation rates.

4.3. The Analysis of the Two-processor Model

In this section the performance of the two-processor system is analyzed and compared with the simulation results. Depending on the two simulator parameters communication delay, *d* and rollback processing multiplier, *r*, the mathematical analysis are performed slightly different. Subsections 4.3.1 - 4.3.3, each present a different case.

4.3.1. Negligible Communication Delay and Rollback Processing Time (d=r=0)

4.3.1.1. The Analysis

Rollback occurs if a processor receives an event with a simulation time lower than that of the processor. According to general assumption 7, the processor receiving such an event, rolls back to the simulation time of the event and resumes operation. With zero communication delay and rollback processing time, this is a point where the two processors synchronize their simulation clocks. Note that if objects are allowed to generate or schedule events in their simulation future, rollback does not necessarily synchronize the simulation clocks of the two processors on which the two objects reside. In the present model, the assumption is that the objects cannot schedule events in their simulation future. Let

$N(t)$ = *the number of rollbacks by real time t* and

t_k = *real time at which k-th rollback occurs*

and

$Y_k = t_k - t_{k-1}$

The sequence $\{Y_k : k \geq 1\}$ are iid and can be considered as the time between renewals of the renewal process $\{N(t) : t \geq 0\}$.

The simulation time of each processor immediately after the k-th rollback is given by :

$s^+(t_k) = min(s_1(t_k), s_2(t_k))$

The sequence $\{s^+(t_k) - s^+(t_{k-1}) : k \geq 1\}$, $(t_0 \equiv 0)$, is iid and $s^+(t_k) - s^+(t_{k-1})$ is the effective simulation time of each processor during the real time between k-1st and k-th rollback (k-th renewal). This term can be considered as the reward of the renewal process defined above and we have

$$\begin{aligned} \gamma_i &= \lim_{t \to \infty} \frac{s_i(t)}{t} \\ &= \frac{E[X_1]}{E[Y_1]} \qquad \textit{if } E[Y_1] \textit{ is finite} \\ &= \frac{E[min(s_1(t_1), s_2(t_1))]}{E[Y_1]} \end{aligned} \tag{2}$$

Next, the expected values in (2) are evaluated. Let

T_k = *real time of the k-th intersubmodel event*

$\alpha_k = i$ *if the k-th intersubmodel event was generated by submodel i*

Then $\{T_k - T_{k-1} : k \geq 1\}, (T_0 \equiv 0)$ is iid exponential with mean $\dfrac{1}{\mu p_1 + \mu p_2}$. Also $\{\alpha_k : k \geq 1\}$ is iid and $P\{\alpha_k = i\} = \dfrac{p_i}{p_1 + p_2}$. Note that $\{T_k - T_{k-1}\}$ and $\{\alpha_k\}$ are independent. Now let

k^* = *the number of intersubmodel events up to and including the event that caused the first rollback*

and let

$s'_i(t)$ = *simulation time of submodel i at real time t in the absence of rollback.*

Then

$$t_1 = T_{k^*} \qquad \textit{and}$$

$$k^* = \min\{k \geq 1 : (\alpha_k = 1 \text{ and } s'_1(T_k) \leq s'_2(T_k))\, or\, (\alpha_k = 2 \text{ and } s'_2(T_k) \leq s'_1(T_k))\}$$

Figure 4-1 gives a diagram of simulation time of the processors as a function of real time. In this Figure, the second inter-submodel event creates a rollback.

Since k^* is independent of $T_k - T_{k-1}$ for all k, then

$$\begin{aligned} E[t_1] &= E[k^*]\, E[T_1] \\ &= \frac{E[k^*]}{\mu(p_1 + p_2)} \end{aligned} \tag{3}$$

If we define $C_k = s'_1(T_k) - s'_2(T_k)$ then

$$\begin{aligned} k^* = \min\{k \geq 1 : (\alpha_k = 1 \text{ and } C_k \leq 0)\ &or \\ (\alpha_k = 2 \text{ and } C_k \geq 0)\ &\} \end{aligned}$$

Considering the disjoint events

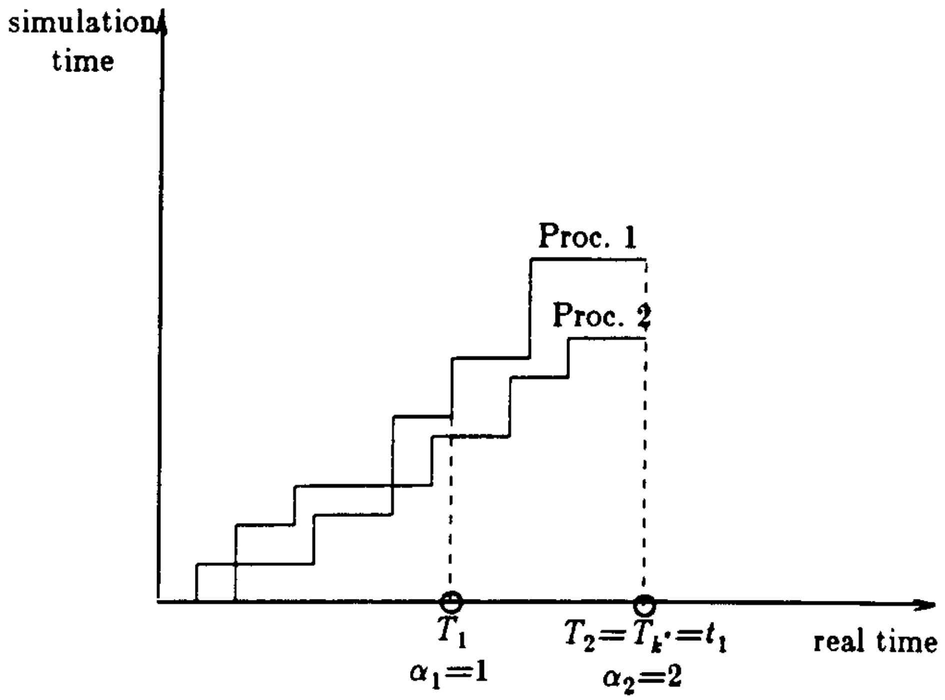

Figure 4-1. Simulation Time of the Processors.

$$A_{k1} = \{\alpha_k=1 \text{ and } C_k>0\} \quad \textit{and}$$

$$A_{k2} = \{\alpha_k=2 \text{ and } C_k<0\}$$

then

$$P\{k^*>k\} = P\{\bigcap_{j=1}^{k} A_{j1} \bigcup A_{j2}\}$$

$$= P\{\bigcup_{i_1=1}^{2} \cdots \bigcup_{i_k=1}^{2} (\bigcap_{j=1}^{k} A_{ji_j})\}$$

$$= \sum_{i_1=1}^{2} \cdots \sum_{i_k=1}^{2} P\{\bigcap_{j=1}^{k} A_{ji_j}\} \tag{4}$$

At this point, we need to make the independence assumption that $\{\alpha_k\}$ and $\{C_k\}$ are independent. In fact the two sequence are not independent, since $\alpha_k=1$ (*or* 2) implies that C_k has a positive (or negative) jump at T_k. However if p_1 and p_2 are small enough, this jump should be small relative to C_k and the assumption is reasonable (p_1 and p_2 are small if the rate of interaction between submodels is small). Note that with the independence assumption, the effect of the positive (or negative) jump of C_k at T_k is ignored. The processor that causes the rollback at that time, increases its own simulation time for the amount of that jump. This will reduce the probability of rollback and also the amount of wasted simulation time. Therefore, it is expected that the analytic result for the speedup be lower than the speedup that can be obtained as p grows. This effect will be shown later when the analytic and simulation results are compared.

Now let $B_{j1} = \{C_j > 0\}$ and $B_{j2} = \{C_j < 0\}$. Then with the independence assumption

$$P\{\bigcap_{j=1}^{k} A_{ji_j}\} = \prod_{j=1}^{k} P\{\alpha_j = i_j\}\, P\{\bigcap_{j=1}^{k} B_{ji_j}\} \qquad (5)$$

To make the analysis simpler and obtain an explicit result for the speedup, we assume that $p_1 = p_2 = p$, so that

$$P\{\alpha_j = i_j\} = 1/2$$

Then from (4) and (5)

$$P\{k^* > k\} = 2^{-k} \sum_{i_1=1}^{2} \cdots \sum_{i_k=1}^{2} P\{\bigcap_{j=1}^{k} B_{ji_j}\}$$

$$= 2^{-k}$$

Hence $E[k^*] = 2$ and from (3)

$$E[t_1] = \frac{1}{\mu p} \tag{6}$$

At this point, we need to calculate the useful simulation time during one renewal cycle , that is

$$\begin{aligned} \min(s_1(t_1), s_2(t_1)) &= \min(s'_1(t_1), s'_2(t_1)) \\ &= \frac{s'_1(t_1) + s'_2(t_1) - |s'_1(t_1) - s'_2(t_1)|}{2} \\ &= \frac{s'_1(T_k{}^*) + s'_2(T_k{}^*) - |C_k{}^*|}{2} \end{aligned} \tag{7}$$

The sequence $\{s'_i(T_k) - s'_i(T_{k-1}) : k \geq 1\}$ is iid and k^* is independent of $s'_i(T_{k+1}) - s'_i(T_k)$ *for* $k \geq k^*$. Hence from Wald's equation [ROS 70],

$$\begin{aligned} E[s'_i(T_k{}^*)] &= E[s'_i(T_1)]E[k^*] \\ &= 2E[s'_i(T_1)] \end{aligned}$$

The simulation time of submodel i at the time of the first intersubmodel event can be written as

$$s'_i(T_1) = \sum_{n=1}^{N} \varphi_{ni} X_{ni}$$

where

N = *Total number of events up to and including the first intersubmodel event*

$\varphi_{ni} = 1$ *if the n-th (inter or intra) submodel event was generated by i*

$= 0$ *otherwise*

X_{ni} = *Simulation time advance of processor i if it generated the n-th (inter or intra) submodel event*

With the assumption that $p_1 = p_2 = p$ we have

$$E[e^{-ss'_i} \mid N] = E[e^{-s\sum_{n=1}^{N}\varphi_{ni}X_{ni}}]$$

$$= \left(1/2\frac{\lambda_i}{\lambda_i + s} + 1/2\right)^N$$

The condition on N can be removed having $P\{N=k\} = (1-p)^{k-1}p$. Then

$$E[e^{-ss'_i}] = \frac{p(2\lambda_i + s)}{2p\lambda_i + (1+p)s}$$

Furthermore,

$$E[s'_i(T_1)] = -\frac{d}{ds}E[e^{-ss'_i}] \quad \textit{when } s=0$$

Then

$$E[s'_i(T_1)] = \frac{1}{2p\lambda_i}$$

and

$$E[s'_i(T_k{}^*)] = \frac{1}{p\lambda_i} \tag{8}$$

At this point we need to derive $E[|C_{k^*}|]$ to determine (7).

$$E[e^{-sC_{k^*}}] = \sum_{k=1}^{\infty} E[e^{-sC_{k^*}} | k^*=k] P\{k^*=k\}$$

$$= \sum_{k=1}^{\infty} E[e^{-sC_k} | [\bigcap_{j=1}^{k-1} (A_{j1} \cup A_{j2})] \cap (A'_{k1} \cup A'_{k2})]$$

$$P\{[\bigcap_{j=1}^{k-1} (A_{j1} \cup A_{j2})] \cap (A'_{k1} \cup A'_{k2})\}$$

where

$$A_{j1} = \{\alpha_j=1 \text{ and } C_j>0\} \ , \ A_{j2} = \{\alpha_j=2 \text{ and } C_j<0\}$$

and

$$A'_{k1} = \{\alpha_k=1 \text{ and } C_k<0\} \ , \ A'_{k2} = \{\alpha_k=2 \text{ and } C_k>0\}$$

Due to the independence assumption for $\{\alpha_k\}$ and $\{C_k\}$;

$$E[e^{-sC_{k^*}}] = \sum_{k=1}^{\infty} E[e^{-sC_k} | [\bigcap_{j=1}^{k-1} (B_{j1} \cup B_{j2})] \cap (B_{k2} \cup B_{k1})] * 2^{-k}$$

$$P\{[\bigcap_{j=1}^{k-1} (B_{j1} \cup B_{j2})] \cap (B_{k2} \cup B_{k1})\}$$

where $B_{j1} = \{C_j>0\}$ and $B_{j2} = \{C_j<0\}$. But for each j, $(B_{j1} \cup B_{j2})$ is the certain event, hence

$$E[e^{-sC_{k^*}}] = \sum_{k=1}^{\infty} 2^{-k} E[e^{-sC_k}] \tag{9}$$

C_k can be written as :

$$C_k = \sum_{j=1}^{k} (C_j - C_{j-1}) \ , \quad C_0 = 0$$

where $\{C_j - C_{j-1} : j \geq 1\}$ is iid. Letting $\Psi(s) = E[e^{-sC_1}]$, we can rewrite (9) as

$$E[e^{-sC_{k^*}}] = \sum_{k=1}^{\infty} 2^{-k} (\Psi(s))^k$$

$$= \frac{\Psi(s)}{2-\Psi(s)}$$

But C_1 is the difference in simulation times of the two processors at the point of the first intersubmodel event, so

$$C_1 = \sum_{n=1}^{N} (\varphi_{n1}X_{n1}-(1-\varphi_{n1})X_{n2})$$

With the assumption that $p_1=p_2=p$

$$\Psi(s) = \sum_{k=1}^{\infty} E\left[e^{-s(\varphi_{n1}X_{n1} - (1-\varphi_{n1})X_{n2})}\right]^k (1-p)^{k-1}p$$

$$= \sum_{k=1}^{\infty} \left[1/2\left(\frac{\lambda_1}{\lambda_1+s}+\frac{\lambda_2}{\lambda_2-s}\right)\right]^k (1-p)^{k-1}p$$

$$= \frac{p[2\lambda_1\lambda_2+(\lambda_2-\lambda_1)s]}{-2s^2+(\lambda_2-\lambda_1)(p+1)s+2p\lambda_1\lambda_2} \qquad (10)$$

Hence

$$E[e^{-sC_k^*}] = \frac{p(\lambda_1\lambda_2+s(\lambda_2-\lambda_1)/2)}{-2s^2+s(\lambda_2-\lambda_1)(1+p/2)+p\lambda_1\lambda_2} \qquad (11)$$

For the case that $\lambda_1=\lambda_2=\lambda$ then

$$E[e^{-sC_k^*}] = \frac{p\lambda^2/2}{(p\lambda^2/2-s^2)}$$

Therefore C_k^* has density $\frac{\lambda\sqrt{p/2}}{2}e^{-\lambda\sqrt{p/2}|t|}$ and

$$E[|C_k^*|] = \frac{1}{\lambda\sqrt{p/2}} \qquad (12)$$

It follows now from (1),(2),(6),(7),(8) and (12) that, if $p_1=P_2=p$ and $\lambda_1=\lambda_2=\lambda$, then

$$\rho = 2(1-\sqrt{p/2}) \qquad (13)$$

For the case of $\lambda_1 \neq \lambda_2$, the roots of the denominator of (11) have been numerically evaluated and used to compute $E[|C_{k^*}|]$ and hence ρ. The results appear in the next section.

4.3.1.2. Numerical and Simulation Results

In Figure 4-2, the speedup ρ is plotted as a function of p, the interaction probability. The results are plotted for different values of the parameter $a = \frac{\lambda_2}{\lambda_1}$. The upper curve is the numerical result of equation (13), i.e. a=1. Note that if $a \neq 1$ the speedup drops off more slowly as p increases than if $a=1$. Also, if $p \leq .05$, reasonable speedup (i.e. close to 1.5) is obtained even if $a=.5$. Furthermore, when p is close to zero, the speedup is close to $1+a$.

In order to analyze the model, an independence assumption was made in section 4.3.1 that is expected to be reasonable if p is small. Therefore the model without the independence assumption was simulated in order to test the effect of this assumption. Simulation results in the form of 95% confidence intervals are also shown in Figure 4-2. Each confidence interval was obtained from 10 independent runs of a simulation whose duration was 7000 events (inter and intrasubmodel events). It can be seen that when $p \leq .08$ the approximations appears to be a good one. Furthermore, the simulation results tend to give a higher speedup than the analytic results as p grows. This is because the independence assumption ignores the effect of the last event in simulation time of the processor that causes the rollback. The simulation of this event reduces the probability of rollback and also the amount of wasted simulation time. Consequently, the speedup is greater without the assumption.

In Figure 4-3, the simulation results for the speedup is plotted for different values of p and a. Note that $a=1$ corresponds to perfect balance of load on the processors. The load balancing is most effective when p is small. In the case when p is large (close to 1), only one processor is doing useful work at any time and hence the load balancing is not a factor any more.

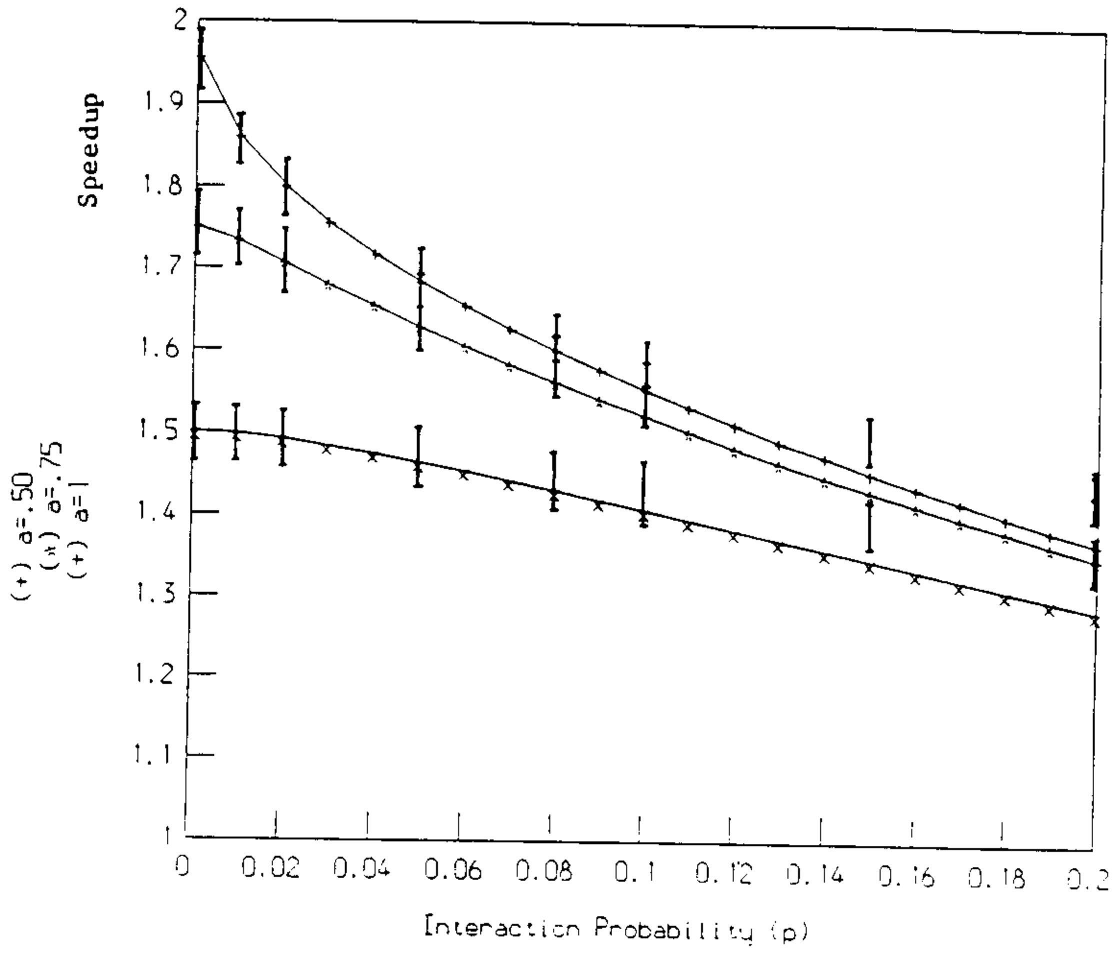

Figure 4-2. Analytic and Simulation Results for the Speedup ($d = r = 0$).

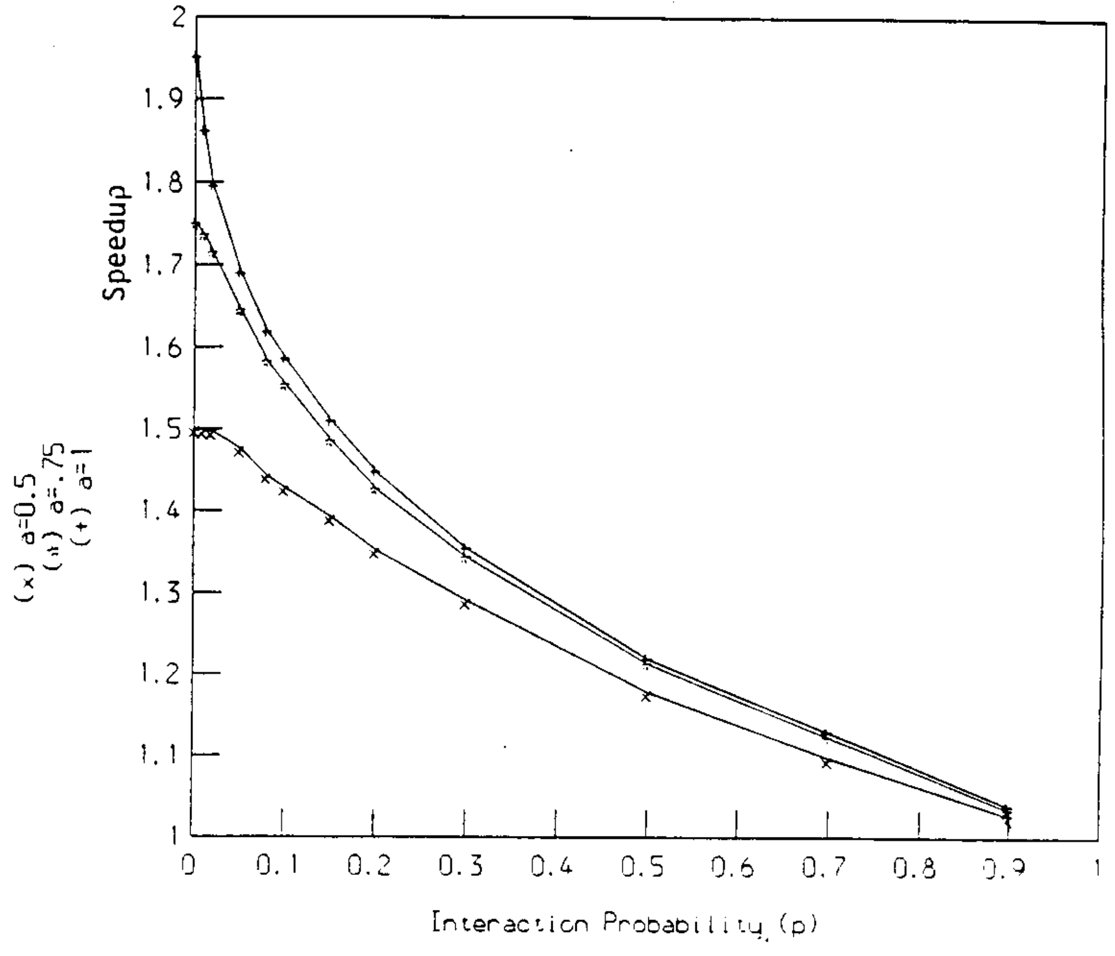

Figure 4-3. Simulation Results for the Speedup (*d*=*r*=0).

4.3.2. Negligible Communication Delay and Non-zero Rollback Processing Time

4.3.2.1. The Analysis

In this section, the two processor model is analyzed when the assumption about negligible rollback processing time is relaxed. In this modified model, it is assumed that the real time to perform rollback is a fraction $r > 0$, of the real time required on the average to simulate the ***wasted simulation time***. The analysis is carried out for the case when $p_1 = p_2 = p$ and $\mu_1 = \mu_2 = \mu$.

Let

t_n^b = *the real time at which the n-th rollback begins*

t_n^e = *the real time at which the n-th rollback ends*

then

$$Y_n = (t_n^e - t_{n-1}^e)$$

will be called the n-th cycle of operation of the processors, (Figure 4-4).

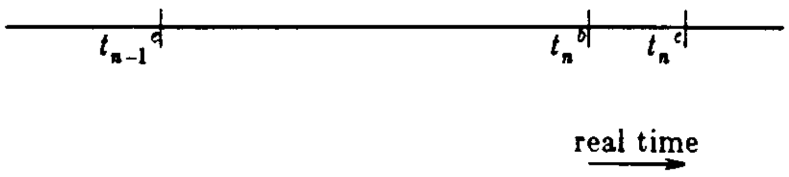

Figure 4-4. The n-th Cycle of the System.

Let also

$$\Delta_n = s_1(t_n^e) - s_2(t_n^e)$$

be the difference in simulation times of the processors at the end of the n-th cycle, The cycle durations $\{Y_i : i \geq 1)$ are not independent of each other. Note that Δ_n depends on $t_n^b - t_n^e$, that is a portion of Y_n. In fact Δ_n is the simulation time advance of the processor that caused rollback during the real time the other processor is performing the rollback operation, i.e. the real time interval of length $t_n^e - t_n^b$. Hence Δ_n depends on Y_n and it also affects Y_{n+1}, the duration of the next cycle..

Similar to the previous case, let

$s'_i(t) =$ *simulation time of submodel i at real time t, in the absense of rollbacks*

$T_k =$ *real time of the k-th inter submodel event after time* t_{n-1}^e

and

$\alpha_k = i$ *if the k-th inter submodel event after* t_{n-1}^e *was generated by submodel i*

Then given $\Delta_{n-1} = \delta$, we have

$$t_n^b - t_{n-1}^e = T_{k^*(\delta)}$$

where

$$k^*(\delta) = \min\{k \geq 1 : (\alpha_k = 1, s'_1(T_k) + \delta \leq s'_2(T_k)) \\ or(\alpha_k = 2, s'_2(T_k) \leq s'_1(T_k) + \delta)\}$$

We also have $\{T_k - T_{k-1} : k \geq 1\}$ are iid. exponential with mean $\frac{1}{2\mu p}$, $\{\alpha_k : k \geq 1\}$ are iid. and $Pr\{\alpha_k = i\} = 1/2$ and finally $\{T_k - T_{k-1} : k \geq 1\}$ and $\{\alpha_k : k \geq 1\}$ are independent.

Letting

$$C_k = s'_1(T_k) - s'_2(T_k)$$

it follows that

$$k^*(\delta) = \min\{k \geq 1 : (\alpha_k = 1, C_k \leq -\delta) or (\alpha_k = 2, C_k \geq -\delta)\}$$

With the assumption that $\{\alpha_k\}$ and $\{C_k\}$ are independent, similar to the case where $r=0$,

$$P\{k^*(\delta) > k\} = 2^{-k}$$

Therefore, $k^*(\delta)$ has the same distribution as $k^* = k^*(0)$. Hence $t_n{}^b - t_{n-1}{}^e$ is distributed as T_{k^*}, from equation (6)

$$E[t_n{}^b - t_{n-1}{}^e] = \frac{1}{\mu p}$$

Now, let X_n be the useful simulation time advance during the n-th cycle. Then X_n is distributed as

$$\min(s'_1(T_{k^*}) + \delta, s'_2(T_{k^*})) \quad \text{if} \quad \delta \geq 0, \qquad or$$

$$\min(s'_1(T_{k^*}), s'_2(T_{k^*}) - \delta) \quad \text{if} \quad \delta < 0$$

In the case where $\delta \geq 0$

$$X_n = \min(s'_1(T_{k^*}) + \delta, s'_2(T_{k^*}))$$

$$= \frac{s'_1(T_{k^*}) + \delta + s'_2(T_{k^*}) - |C_{k^*} + \delta|}{2}$$

We have

$$E[s'_i(T_{k^*})] = \frac{1}{p\lambda_i}$$

Similar to the case where r=0,

$$E[e^{-sC_{k^*}}] = \frac{p[\lambda_1\lambda_2 + s(\lambda_1-\lambda_2)/2]}{-2s^2 + s(\lambda_2-\lambda_1)(1+p/2) + p\lambda_1\lambda_2}$$

When $\lambda_1 = \lambda_2 = \lambda$, we have

$$E[e^{-sC_{k^*}}] = \frac{p\lambda^2/2}{p\lambda^2/2-s^2}$$

and hence C_{k^*} has density $\frac{\lambda\sqrt{p/2}}{2}e^{-\lambda\sqrt{p/2}|t|}$.

Considering the above, $E[|C_{k^*(\delta)}+\delta|]$

$$= \frac{\lambda\sqrt{p/2}}{2}\left[\int_{-\infty}^{-\delta} -(t+\delta)e^{\lambda\sqrt{p/2}t}dt + \int_{-\delta}^{0}(t+\delta)e^{\lambda\sqrt{p/2}t}dt + \int_{0}^{\infty}(t+\delta)e^{-\lambda\sqrt{p/2}t}dt\right]$$

$$= \frac{\lambda\sqrt{p/2}}{2}\left[\int_{-\infty}^{0} -(t+\delta)e^{\lambda\sqrt{p/2}t}dt + 2\int_{-\delta}^{0}(t+\delta)e^{\lambda\sqrt{p/2}t}dt + \int_{0}^{\infty}(t+\delta)e^{-\lambda\sqrt{p/2}t}dt\right]$$

$$= \frac{\lambda\sqrt{p/2}}{2}\left[\int_{0}^{\infty}(t-\delta)e^{-\lambda\sqrt{p/2}t}dt + 2\int_{-\delta}^{0}(t+\delta)e^{\lambda\sqrt{p/2}t}dt + \int_{0}^{\infty}(t+\delta)e^{-\lambda\sqrt{p/2}t}dt\right]$$

$$= \frac{\lambda\sqrt{p/2}}{2}\left[\frac{2}{\lambda\sqrt{p/2}}\int_{0}^{\infty}t\lambda\sqrt{p/2}e^{-\lambda\sqrt{p/2}t}dt + 2\int_{0}^{\delta}(-t+\delta)e^{-\lambda\sqrt{p/2}t}dt\right]$$

$$= \frac{\lambda\sqrt{p/2}}{2}\left[\frac{2}{(\lambda\sqrt{p/2})^2} + 2e^{-\lambda\sqrt{p/2}\delta}\int_{0}^{\delta}ye^{\lambda\sqrt{p/2}y}dy\right]$$

$$= \frac{1}{\lambda\sqrt{p/2}} + \lambda\sqrt{p/2}e^{-\lambda\sqrt{p/2}\delta}\frac{1}{\lambda\sqrt{p/2}}\left\{e^{\lambda\sqrt{p/2}y}\left(y-\frac{1}{\lambda\sqrt{p/2}}\right)\right\} \qquad \textit{for } y \in (0,\delta)$$

$$= \delta + \frac{e^{-\lambda\sqrt{p/2}\delta}}{\lambda\sqrt{p/2}}$$

Therefore,

$$E[X_n] = 1/2\left\{E[s'_1(T_{k^*})] + E[s'_2(T_{k^*})] + \delta - E[|C_{k^*}(\delta)+\delta|]\right\}$$

$$= \frac{1}{p\lambda} + \frac{\delta - \left(\delta + \frac{e^{-\lambda\sqrt{p/2}\delta}}{\lambda\sqrt{p/2}}\right)}{2}$$

$$= \frac{1}{p\lambda} - \frac{e^{-\lambda\sqrt{p/2}\delta}}{2\lambda\sqrt{p/2}} \qquad for\ \delta \geq 0$$

Similarly, when $\delta < 0$, we have

$$E[X_n] = \frac{1}{p\lambda} - \frac{e^{\lambda\sqrt{p/2}\delta}}{2\lambda\sqrt{p/2}}$$

Hence

$$E[X_n] = \frac{1}{p\lambda} - \frac{e^{-\lambda\sqrt{p/2}|\delta|}}{2\lambda\sqrt{p/2}} \tag{14}$$

>From general assumption 6, we have $t_n^e - t_n^b$ is distributed as $\frac{r\lambda}{\mu}|C_{k^*} + \delta|$.

Therefore,

$$E[t_n^e - t_n^b] = \frac{r\lambda}{\mu}\left(|\delta| + \frac{e^{-\lambda\sqrt{p/2}|\delta|}}{\lambda\sqrt{p/2}}\right)$$

But, the duration of the n-th cycle Y_n satisfies

$$E[Y_n] = E[t_n^e - t_n^b] + E[t_n^b - t_{n-1}^e]$$

$$= \frac{1}{\mu p} + \frac{r\lambda}{\mu}\left(|\delta| + \frac{e^{-\lambda\sqrt{p/2}|\delta|}}{\lambda\sqrt{p/2}}\right) \tag{15}$$

Now if $\lambda_1 = \lambda_2 = \lambda$, then the identity of the subsystem that caused the rollback does not matter and

$$E[|\Delta_n| \,|\, t_n^e - t_n^b = \tau] = \frac{\mu\tau}{\lambda}$$

and hence

$$E[|\Delta_n| \mid \Delta_{n-1} = \delta] = \frac{\mu}{\lambda}E[t_n^e - t_n^b]$$

$$= r\left(|\delta| + \frac{e^{-\lambda\sqrt{p/2}|\delta|}}{\lambda\sqrt{p/2}}\right) \tag{16}$$

Removing the conditions in (14), (15) and (16), we have

$$E[X_n] = \frac{1}{p\lambda} - \frac{E[e^{-\lambda\sqrt{p/2}|\Delta_{n-1}|}]}{2\lambda\sqrt{p/2}} \tag{14'}$$

$$E[Y_n] = \frac{1}{\mu p} + \frac{r\lambda}{\mu}\left(E[|\Delta_{n-1}|] + \frac{E[e^{-\lambda\sqrt{p/2}|\Delta_{n-1}|}]}{\lambda\sqrt{p/2}}\right) \tag{15'}$$

$$E[|\Delta_n|] = r\left[E[|\Delta_{n-1}|] + \frac{E[e^{-\lambda\sqrt{p/2}|\Delta_{n-1}|}]}{\lambda\sqrt{p/2}}\right] \tag{16'}$$

Assuming that $\Delta_n \rightarrow \Delta$ in distribution as n $\rightarrow\infty$, from (16'), we have

$$E[|\Delta|] = r\left(E[|\Delta|] + \frac{E[e^{-lamp|\Delta|}]}{\lambda\sqrt{p/2}}\right) \tag{17}$$

A proof for the correctness of this assumption appears in Appendix A. Using the inequalities,

$$1 - \lambda\sqrt{p/2}E[|\Delta|] \leq E[e^{-\lambda\sqrt{p/2}|\Delta|}] \leq 1$$

the following can be derived

$$E[|\Delta|] \leq r\left(E[|\Delta|] + \frac{1}{\lambda\sqrt{p/2}}\right)$$

$$E[|\Delta|] \leq \left(\frac{r}{1-r}\right)\frac{1}{\lambda\sqrt{p/2}} \tag{18}$$

Also we have,

$$E[|\Delta|] \geq r\left(E[|\Delta|] + \frac{1}{\lambda\sqrt{p/2}} - E[|\Delta|]\right)$$

$$= \frac{r}{\lambda\sqrt{p/2}} \tag{19}$$

Then we have,

$$E[X] = \lim_{n\to\infty} E[X_n] = \frac{1}{p\lambda} - \frac{E[e^{-\lambda\sqrt{p/2}|\Delta|}]}{2\lambda\sqrt{p/2}}$$

$$= \frac{1}{p\lambda} - \left(\frac{1-r}{2r}\right)E[|\Delta|]$$

and

$$E[Y] = \lim_{n\to\infty} E[Y_n] = \frac{1}{\mu p} + \frac{r\lambda}{\mu}\left(\Delta|] + \frac{E[e^{-\lambda\sqrt{p/2}|\Delta|}]}{\lambda\sqrt{p/2}}\right)$$

$$= \frac{1}{\mu p} + \frac{\lambda}{\mu}E[|\Delta|]$$

Hence

$$\frac{E[X]}{E[Y]} \geq \frac{\frac{1}{\lambda}\left(\frac{1}{p} - \frac{1}{2\sqrt{p/2}}\right)}{\frac{1}{\mu}\left(\frac{1}{p} + \frac{r}{1-r}\frac{1}{\sqrt{p/2}}\right)}$$

$$= \frac{\mu}{\lambda}\left(\frac{1-\sqrt{p/2}}{1+\frac{r}{1-r}\sqrt{2p}}\right)$$

and using (19),

$$\frac{E[X]}{E[Y]} \le \frac{\frac{1}{\lambda}\left(\frac{1}{p}-\frac{1-r}{2\sqrt{p/2}}\right)}{\frac{1}{\mu}\left(\frac{1}{p}+\frac{r}{\sqrt{p/2}}\right)}$$

$$= \frac{\mu}{\lambda}\left(\frac{1-(1-r)\sqrt{p/2}}{1+r\sqrt{2p}}\right)$$

At this point, we assume that the system is ergodic in the sense that

$$\lim_{t\to\infty}\frac{\min(s_1(t),s_2(t))}{2} = \frac{E[X]}{E[Y]} \qquad a.s.$$

The formal proof is given in Appendix A. Then

$$2\left(\frac{1-\sqrt{p/2}}{1+\frac{r}{1-r}\sqrt{2p}}\right) \le \rho \le 2\left(\frac{1-(1-r)\sqrt{p/2}}{1+r\sqrt{2p}}\right) \qquad (20)$$

As mentioned, r is the ratio of the real time to rollback s seconds of simulation time to the real time needed to advance s seconds of simulation time. It is shown in Appendix A that $r<1$ is required for ergodicity.

Recall that the general assumption 7 requires that the checkpoints be perfect. In that case, the rollback operation does not include the "simulate forward" step. In the case where $r>0$ this assumption can be relaxed and the "simulate forward" period can be considered part of the rollback operation. This implies larger values for r, that is workable as long as $r<1$.

4.3.2.2. Numerical and Simulation Results

In Figures 4-5 to 4-7 the bounds on the speedup given by equation (20) is plotted as a function of p, the interaction probability. The different Figures correspond to different values of r, the rollback processing multiplier. The area between two curves in each Figure, is the area in which the speedup is expected to be.

In the same Figures, the confidence intervals obtained by the simulation of a similar model are presented. The simulation model excludes the independence assumption and has the same characteristics described in section 4.3.2.2. As can be seen, in comparison with the case $r=0$, the independence assumption is reasonable for smaller range of p. Furthermore, the range for p reduces as r grows. This behavior can be expected. Note that r multiplied by the wasted simulation time gives the equivalent of rollback processing time in simulation time. With the independence assumption, the wasted simulation time is greater than the case without it. As r grows, the amount of wasted simulation time in the performance becomes more effective. Therefore, the analytic results confirm with those of simulation for smaller values of p.

In Figure 4-8, the simulation results are presented for different values of r. It can be observed that including r in the model slightly changes the speedup (even r=.5). However, if checkpointing is not done frequently, the "simulate forward" period will become large. This corresponds to increasing value of r, possibly more than 1. It was shown in Appendix A that for $r>1$ the system becomes unstable, in the sense that more time will be spent on rollback processing than on simulation. Hence, checkpointing needs to be done frequently enough to avoid such a case and also not use excessive memory.

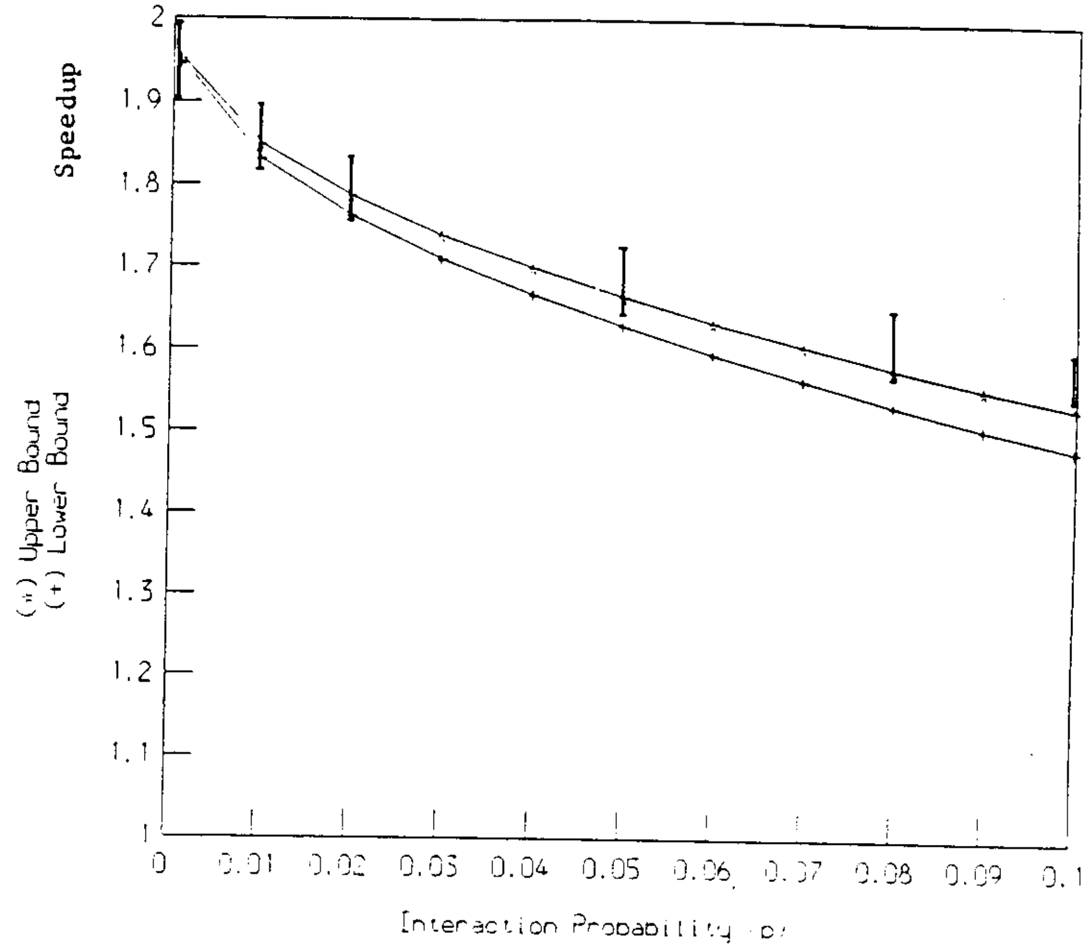

Figure 4-5. Analytic and Simulation Results for the Speedup (d=0,r=.1).

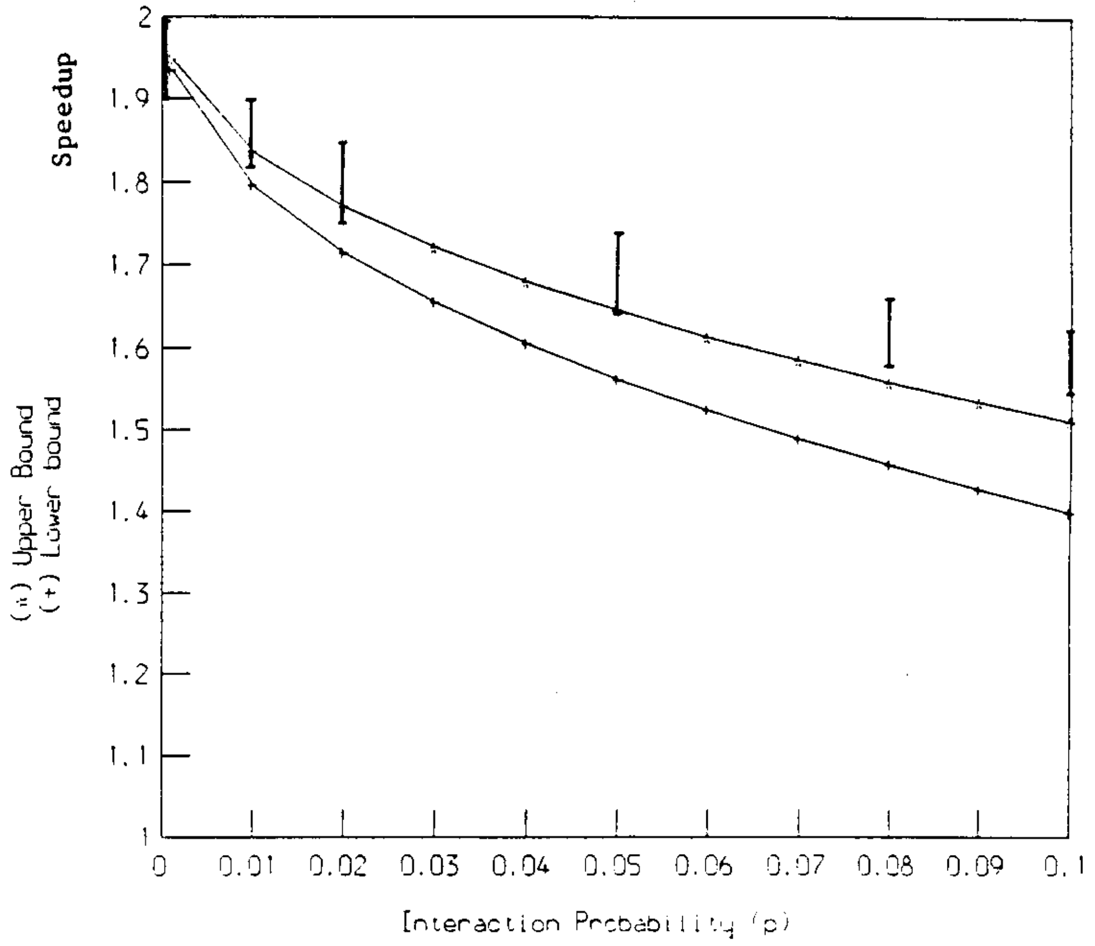

Figure 4-6. Analytic and Simulation Results for the Speedup (d=0,r=.2).

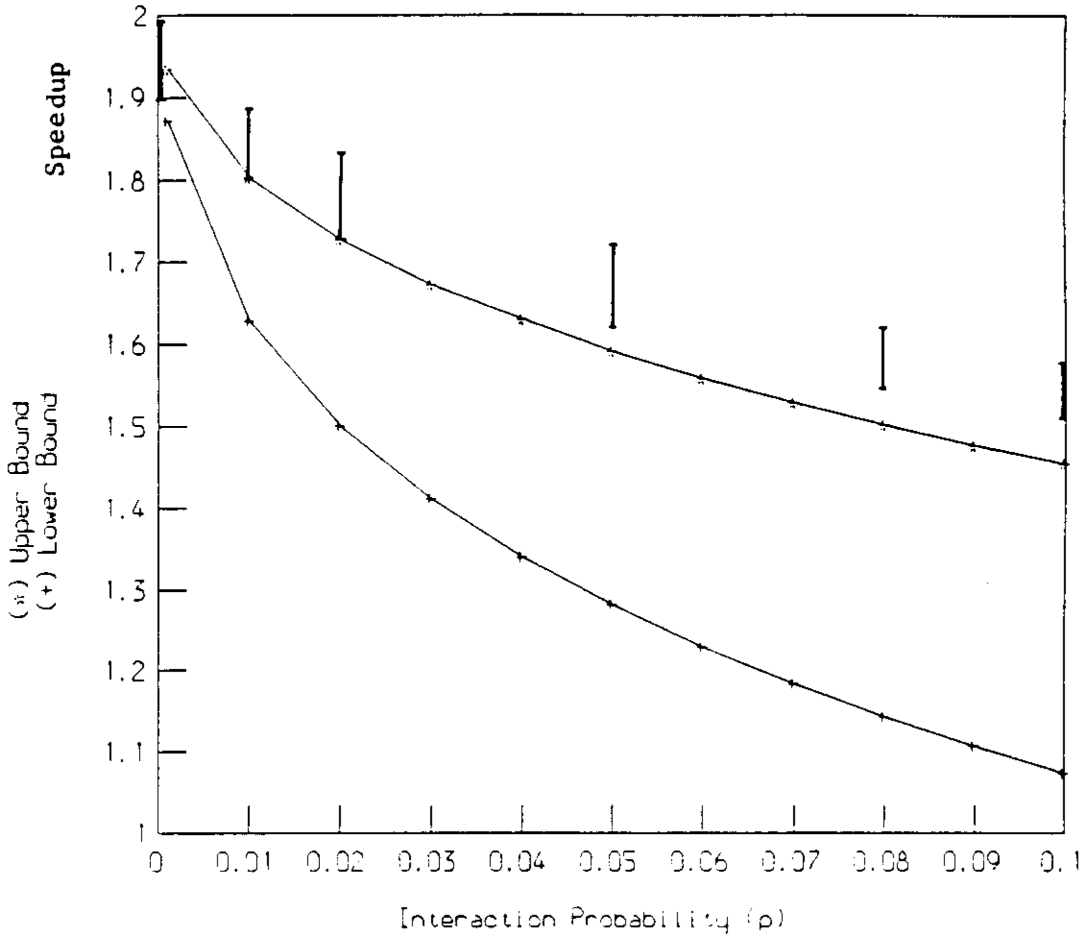

Figure 4-7. Analytic and Simulation Results for the Speedup ($d=0, r=.5$).

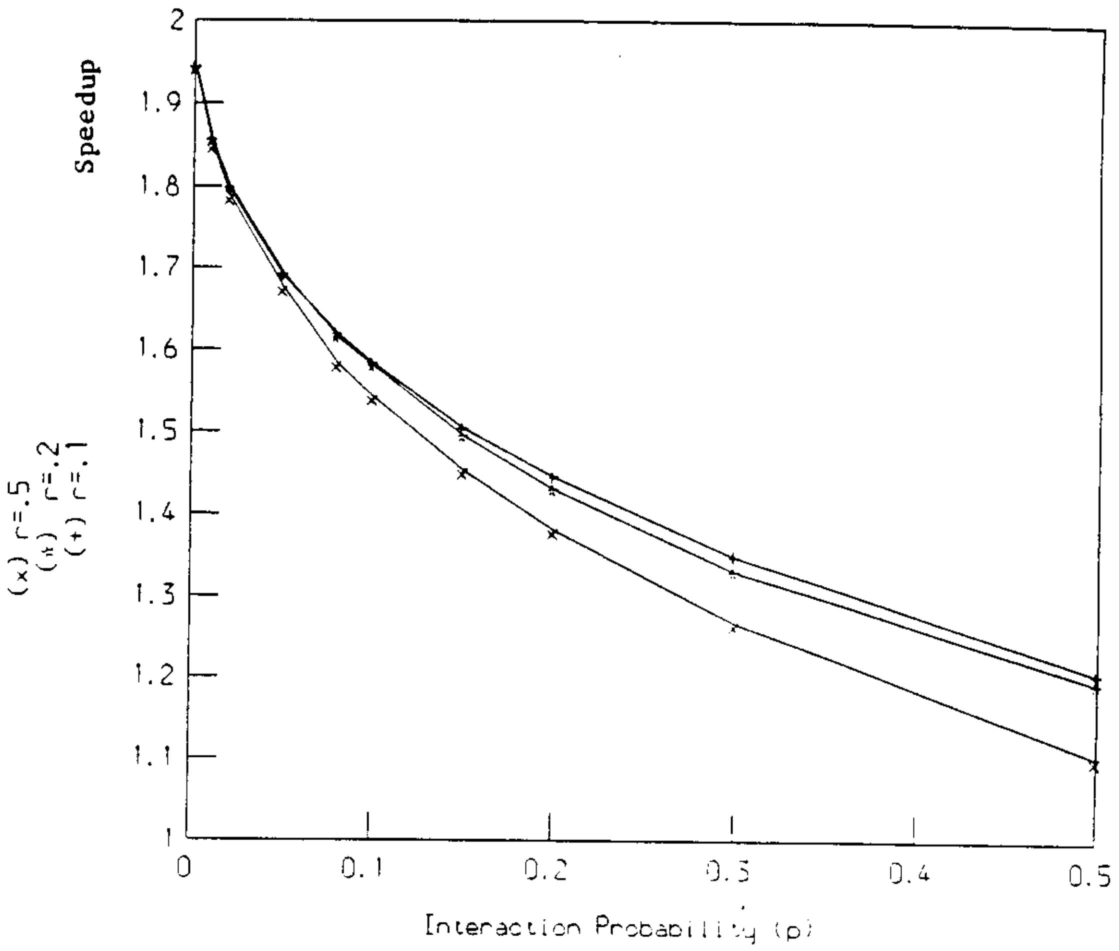

Figure 4-8. Simulation Results for the Speedup (d=0,r≥0).

4.3.3. Non-zero Communications Delay and Rollback Processing Time ($d>0, r>0$)

4.3.3.1. The Analysis

In this section both the assumption of negligible communication delay and the assumption of negligible rollback processing time are removed from the model.

In the new model, it is assumed that

(1) There is a non-zero communication delay between the processors performing the simulation. The communication delay is assumed to be a constant d.

(2) The rollback processing factor r is greater than zero.

Considering the above, let

t_n^g = *real time at which the event that causes the n-th rollback is generated*

t_n^b = *real time at which rollback operation begins,*

t_n^e = *real time at which the n-th rollback operation ends*

As before, let (t_{n-1}^e, t_n^e) be the n-th cycle and its length be denoted by Y_n, i.e. $Y_n = t_n^e - t_{n-1}^e$. Figure 4-9 shows the n-th cycle. Also, let

$$\Delta_n = s_1(t_n^e) - s_2(t_n^e)$$

then

$|\Delta_n|$ = *difference in sim. times of processors at the end of n-th cycle*

= *sim. time advance of the processor that caused the n-th rollback* and

$$X_n = \min\left(s_1(t_n{}^e), s_2(t_n{}^e)\right) - \min\left(s_1(t_{n-1}{}^e), s_2(t_{n-1}{}^e)\right)$$

$$= \textit{useful simulation time during the n-th cycle}$$

Note that

$$\min\left(s_1(t_n{}^e), s_2(t_n{}^e)\right) \geq \min\left(s_1(t_n{}^g), s_2(t_n{}^g)\right)$$

The equality holds if $d=0$. In the case of non-zero d, it is possible that the processor that caused rollback did not have the smaller simulation time at the time it transmitted the event. The receiving processor may be behind in simulation time at $t_n{}^g$, but has advanced ahead of the time of the event while the message is in-transit. Hence

$$X_n \geq X_n$$

$$= min(s_1(t_n{}^g), s_2(t_n{}^g)) - min(s_1(t_{n-1}{}^e), s_2(t_{n-1}{}^e))$$

If $\Delta_{n-1} \geq 0$ then $s_1(t_{n-1}{}^e) \geq s_2(t_{n-1}{}^e)$ and

$$X_n = \min(s_1(t_n{}^g) - s_1(t_{n-1}{}^e) + \Delta_{n-1}\ ,\ s_2(t_n{}^g) - s_2(t_{n-1}{}^e)) \tag{21.a}$$

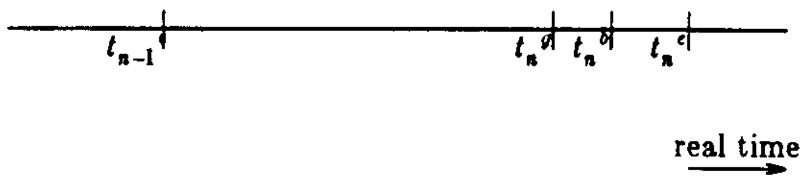

Figure 4-9. The n-th Cycle of Operation.

Otherwise, if $\Delta_{n-1} \leq 0$ then $s_1(t_{n-1}^e) \leq s_2(t_{n-1}^e)$ and

$$X_n = \min(s_1(t_n^g) - s_1(t_{n-1}^e)\ ,\ s_2(t_n^g) - s_2(t_{n-1}^e) - \Delta_{n-1}) \tag{21.b}$$

Let also T_k, α_k and $s'_i(t)$ be defined as before.

But $t_n^g - t_{n-1}^e$ is distributed as $T_{k^*_n(d)}$ where

$$k^*_n(d) = \min\{k \geq 1 : (\alpha_k = 1, s'_1(T_k) + \Delta_{n-1} \leq s'_2(T_k + d))$$
$$or\ (\alpha_k = 2, s'_2(T_k) \leq s'_1(T_k + d) + \Delta_{n-1}\}$$

Then

$$E[t_n^g - t_{n-1}^e] = E[T_{k^*_n(d)}] = \frac{E[k^*_n(d)]}{2\mu p} \tag{22}$$

Also

$$s'_i(T_k + d) = s'_i(T_k) + D_i$$

where D_i is the simulation time advance of subsystem i during a real time period of length d.

Now assume that D_i is a constant equal to its mean, i.e. assume that

$$D_i = D = \frac{\mu d}{\lambda} \quad i = 1,2$$

and let

$$C_k = s'_1(T_k) - s'_2(T_k)$$

Then

$$k^*_n(d) = \min\ \{k \geq 1 : (\alpha_k = 1, C_k + \Delta_{n-1} \leq D)\ or\ (\alpha_k = 2, C_k + \Delta_{n-1} \geq -D)\}$$
$$= \min\ \{k \geq 1 : (-D \leq C_k + \Delta_{n-1} \leq D)\ or\ (\alpha_k = 1, C_k + \Delta_{n-1} < -D)\ or$$

$$(\alpha_k=2, C_k+\Delta_{n-1}>D)\}$$

Given that $\Delta_{n-1} = \delta$, then

$$k^*(d,\delta) = \min\{k\geq 1 : (-D\leq C_k+\delta\leq D) \text{ or } (\alpha_k=1, C_k+\delta<-D) \text{ or }$$
$$(\alpha_k=1, C_k+\delta>D)\}$$

Now consider the disjoint events

$$A_{k1} = \{\alpha_k=1, C_k+\delta>D\} = \{\alpha_k=1\}\bigcap B_{k1}$$
$$A_{k2} = \{\alpha_k=2, C_k+\delta<-D\} = \{\alpha_k=2\}\bigcap B_{k2}$$

We then have

$$P\left\{k^*(d,\delta)>k\right\} = P\{\bigcap_{j=1}^{k}\left(A_{j1}\bigcup A_{j2}\right)\}$$
$$= \sum_{i_1=1}^{2}\cdots\sum_{i_k=1}^{2} P\left(\bigcap_{j=1}^{k} A_{ji_j}\right)$$

Here we make the same assumption that we made in the case where d=0. We assume that $\{\alpha_k\}$ and $\{C_k\}$ are independent, it then follows that

$$P\{k^*(d,\delta)>k\} = 2^{-k}\sum_{i_1=1}^{2}\cdots\sum_{i_k=1}^{2} P\left(\bigcap_{j=1}^{k} B_{ji_j}\right)$$
$$= 2^{-k}P\left\{\bigcap_{j=1}^{k}(B_{j1}\bigcup B_{j2})\right\}$$
$$= 2^{-k}P\left\{\bigcap_{j=1}^{k}\left(C_j+\delta \notin [-D,D]\right)\right\}$$

Therefore

$$E[k^*(d,\delta)] = 1 + \sum_{k=1}^{\infty} 2^{-k}P\left\{\bigcap_{j=1}^{k}\left(C_j+\delta \notin [-D,D]\right)\right\} \tag{23}$$

To evaluate the term within the summation, note that $C_j = C_{j-1}+C_j-C_{j-1}$ where C_j-C_{j-1} is independent of C_{j-1} and is distributed same as C_1. Let $C = C_j-C_{j-1}$. According to (10), we have

$$\psi(s) = E[e^{-sC}] = \frac{p\lambda^2}{p\lambda^2-s^2}$$

Let

$$p_j = P\{C_j+\delta \notin [-D,D] \mid C_i+\delta \notin [-D,D],\ 1\leq i\leq j-1\}$$

$$= \int_{-\infty}^{-D} + \int_{D}^{\infty} P\{C_{j-1}+C+\delta \notin [-D,D] \mid C_i+\delta \notin [-D,D],\ 1\leq i\leq j-1, C_{j-1}+\delta=y\}$$

$$dP\{C_{j-1}+\delta\leq y \mid C_i+\delta \notin [-D,D],\ 1\leq i\leq j-1\}$$

Let

$$f(y)dy = dP\{C_{j-1}+\delta\leq y \mid C_i+\delta \notin [-D,D],\ 1\leq i\leq j-1\}$$

Then

$$p_j = \int_{-\infty}^{-D} + \int_{D}^{\infty} P\{C+y \notin [-D,D]\}f(y)dy$$

$$= \int_{-\infty}^{-D} + \int_{D}^{\infty} P\{C \notin [-D-y,D-y]\}f(y)dy$$

But from (11), C has density $\frac{\lambda\sqrt{p}}{2}e^{-\lambda\sqrt{p}|t|}$. It follows that

$$P\{C+y \notin [-D,D]\} \geq P\{C \notin [-D,D]\}$$

Hence

$$p_j \geq P\{C \notin [-D,D]\}$$

$$= e^{-\lambda\sqrt{p}D} = e^{-\mu d\sqrt{p}}$$

Therefore,

$$P\{\bigcap_{j=1}^{k}(C_j+\delta) \notin [-D,D]\} = \prod_{j=1}^{k} p_j \geq (e^{-\mu d\sqrt{p}})^k$$

Hence from (23),

$$E[k^*(d,\delta)] \geq 1+\sum_{k=1}^{\infty}\left(\frac{e^{-\mu d\sqrt{p}}}{2}\right)^k$$

$$= \frac{2}{2-e^{-\mu d\sqrt{p}}} \tag{24}$$

Note that if d=0 then $P\{\prod_{j=1}^{k}(C_j+\delta \notin [-D,D])\} = 1$ and hence

$$E[K^*(0,\delta)] = 2 \tag{25}$$

that agrees with the previous results.

At this point we look at the useful simulation time within a cycle. Note that $s_i(t_n^g)-s_i(t_{n-1}^e)$ is distributed as $s_i(T_{k^*(d,\delta)})$. It follows from (21) that

$$X_n = \min(s'_1(T_{k^*(d,\delta)})+\delta, s'_2(T_{k^*(d,\delta)})) \qquad \delta\geq 0$$

$$= \min(s'_1(T_{k^*(d,\delta)}), s'_2(T_{k^*(d,\delta)})-\delta) \qquad \delta<0$$

$$= \frac{s'_1(T_{k^*(d,\delta)})+s'_2(T_{k^*(d,\delta)})+|\delta|-|C(d,\delta)|}{2} \tag{26}$$

where

$$C(d,\delta) = s'_1(T_{k^*(d,\delta)})-s'_2(T_{k^*(d,\delta)})+\delta$$

$$= C_{k^*(d,\delta)}+\delta$$

We also have

$$E[s'_i(T_{k^*(d,\delta)})] = \frac{E[k^*(d,\delta)]}{2p\lambda} \tag{27}$$

It follows from (27) and (28) that

$$E[X_n(\delta)] \geq E[X_n(\delta)]$$

$$= \frac{E[k^*(d,\delta)]}{2p\lambda} + \frac{|\delta|}{2} - \frac{E[|C(d,\delta)|]}{2} \tag{28}$$

Using 22,

$$E[Y_n(\delta)] = \frac{E[k^*(d,\delta)]}{2\mu p} + E[t_n{}^e(\delta) - t_n{}^g(\delta)]$$

Also $|\Delta_n|$ is distributed as $s'_i(t_n{}^e - t_n{}^g)$ i=1,2. So

$$E[|\Delta_n(\delta)|] = \frac{\mu}{\lambda} E[t_n{}^e(\delta) - t_n{}^g(\delta)]$$

and

$$E[Y_n(\delta)] = \frac{E[k^*(d,\delta)]}{2\mu p} + \frac{\lambda}{\mu} E[|\Delta_n(\delta)|] \tag{29}$$

Now the wasted simulation time is given by

$D + s_2(t_n{}^g) - s_1(t_n{}^g)$ *if 1 causes rollback*

$D + s_1(t_n{}^g) - s_2(t_n{}^g)$ *if 2 causes rollback*

That is , wasted simulation time is

$$\leq D + |s_1(t_n{}^g) - s_2(t_n{}^g)|$$

with equality if d=0.

$$= D + |\Delta_{n-1} + s'_1(T_{k^*(d,\delta)}) - s'_2(T_{k^*(d,\delta)})|$$

Hence, the expected real time to perform the rollback operations satisfies the following:

$$E[t_n^e(\delta)-t_n^b(\delta)] \le \frac{r\lambda}{\mu}\Big(D+E[|C(d,\delta)|]\Big)$$

Therefore, the expected difference in simulation time of the two processors at the end of the n-th cycle is given by

$$E[|\Delta_n(\delta)|] \le D(1+r)+rE[|C(d,\delta)|] \tag{30}$$

with equality when d = 0.

But

$$\begin{aligned} E[|C(d,\delta)|] &= E[|C(d,\delta)| \mid C(d,\delta) \in [-D,D]]P\{C(d,\delta) \in [-D,D]\} \\ &\quad + E[|C(d,\delta)| \mid C(d,\delta) \notin [-D,D]]P\{C(d,\delta) \notin [-D,D]\} \\ &\le D + E[|C(0,\delta)|] \end{aligned} \tag{31}$$

With a similar approach to that of the case (r=d=0), we can show that $k^*(0,\delta)$ has a geometric distribution with parameter $\frac{1}{2}$ as does k^*. Therefore, $C_{k^*}(0,\delta)$ has the same distribution as C_{k^*}, i.e. it has density $\frac{\lambda\sqrt{p/2}\,e^{-\lambda\sqrt{p/2}|t|}}{2}$.

Using the density, it is straight forward to show that

$$\begin{aligned} E[|C(0,\delta)|] &= E[|C_{k^*(0,\delta)}+\delta|] \\ &= |\delta| + \frac{e^{-\lambda\sqrt{p/2}|\delta|}}{\lambda\sqrt{p/2}} \end{aligned} \tag{32}$$

It follows from (28), (31) and (32) that

$$\begin{aligned} E[X_n(\delta)] \ge E[X_n(\delta) &\ge \frac{E[k^*(d,\delta)]}{2p\lambda} - \frac{D}{2} - \frac{e^{-\lambda\sqrt{p/2}|\delta|}}{2\lambda\sqrt{p/2}} \\ &\ge \frac{E[k^*(d,\delta)]}{2p\lambda} - \frac{D}{2} - \frac{1}{2\lambda\sqrt{p/2}} \end{aligned} \tag{33}$$

It also follows from (30)-(32) that

$$E[|\Delta_n(\delta)|] \leq D(1+2r) + r\left(|\delta| + \frac{e^{-\lambda\sqrt{p/2}|\delta|}}{\lambda\sqrt{p/2}}\right)$$

$$\leq D(1+2r) + r\left(|\delta| + \frac{1}{\lambda\sqrt{p/2}}\right) \tag{34}$$

Hence

$$E[|\Delta_n|] \leq D(1+2r) + rE[|\Delta_{n-1}|] + \frac{r}{\lambda\sqrt{p/2}} \tag{35}$$

We now assume that $\lim_{n\to\infty} E[|\Delta_n|] = E[|\Delta|]$ exists and is finite. The proof is similar to the previous case given in Appendix A. Then from (15)

$$E[|\Delta|] \leq \frac{D(1+2r) + \frac{r}{\lambda\sqrt{p/2}}}{1-r} \tag{36}$$

Also assuming that

$$\lim_{n\to\infty} E[k^*_n(d)] = E[k^*(d)] \quad \textit{exist}$$

$$\lim_{n\to\infty} E[X_n] = E[X] \quad \textit{exists}$$

and

$$\lim_{n\to\infty} E[Y_n] = E[Y] \quad \textit{exists}$$

Then we have from (13)

$$E[X] \geq \frac{E[k^*(d)]}{2p\lambda} - \frac{D}{2} - \frac{1}{2\lambda\sqrt{p/2}}$$

From (29) and (36), it follows that

$$E[Y] \leq \frac{E[k^*(d)]}{2\mu p} + \frac{\lambda}{\mu}\left(\frac{D(1+2r) + \frac{r}{\lambda\sqrt{p/2}}}{1-r} \right)$$

Assume (similar to the previous case),

$$\lim_{t\to\infty} \frac{\min(s_1(t), s_2(t))}{t} = \frac{E[X]}{E[Y]} \qquad a.s.$$

Then ρ, the speedup factor is given by

$$\rho = 2\frac{\lambda}{\mu}\frac{E[X]}{E[Y]}$$

$$\geq 2\left[\frac{\frac{E[k^*(d)]}{2p} - \frac{\lambda D}{2} - \frac{1}{2\lambda\sqrt{p/2}}}{\frac{E[k^*(d)]}{2p} + \frac{\lambda D(1+2r) + \frac{r}{\lambda\sqrt{p/2}}}{1-r}} \right]$$

Using the lower bound in (4), it follows that

$$\rho \geq 2\left(\frac{\frac{1}{2-e^{-\mu d\sqrt{p}}} - \frac{p\mu d}{2} - \sqrt{p/2}}{\frac{1}{2-e^{-\mu d\sqrt{p}}} + \frac{p\mu d(1+2r) + r\sqrt{2p}}{1-r}} \right) \tag{37}$$

Note that the two parameters of the simulator μ, the event processing rate and d, the communication delay appear together in all the terms. μd is the average number of events that can be simulated during the interval of transmission of a message. Thus, to makeup for the loss in the speedup that results from greater values of d, we need to reduce μ. i.e. reduce the speed of processing.

If d=0, then

$$\rho \geq 2\left(\frac{1-\sqrt{p/2}}{1+\frac{r\sqrt{2p}}{1-r}}\right) \tag{38}$$

that agrees with (20).

Note that in the case, $d=r=0$ (equation (13)), the speedup is independent of the event processing rate μ and event generation rate, λ, as long as the two processors are homogeneous. The only factor that is affecting the performance is p, the interaction probability. In the above case, $d>0$, $r>0$, the speedup is still independent of the simulation parameter, λ. In this case, the two simulator parameters d and μ affect the performance and the two always appear together. Recall that the term μd is the average number of simulated events during a message transmission interval. It may appear at first that by increasing the processing power of the processors, i.e. μ, the speedup will also be increased. The equation (38) shows the contrary. Increasing μ has the same effect as increasing d, and this, as will be shown in the next section reduce the speedup. Therefore, in order to increase the performance, the term μd has to be reduced for fixed p.

The analysis in this section assumes that in the simulator a delay d exists between the time to send and receive a message. As a result, the speedup is reduced. The reduction is due to the fact that the receiving processor is still simulating during the time d. Once it receives the message, on the average it has a higher simulation clock than it would have in the case where $d=0$. This increase in simulation time, increases probability of rollback and therefore reduces the speedup.

Now consider the case where there is a similar time difference between the time to send and receive a message in the simulation model also. This time difference can be the result of future scheduling of events or a communications delay in the system that is being simulated. We denote this time difference by d_s. In this case, the final speedup result of equation (37) can be approximated by replacing d with $(d-d_s*\frac{\lambda}{\mu})$, assuming this term is not negative. Note that the effect of the simulator delay d, was the average amount of the simulation time progress that can be made during this time. i.e., since the receiving processor can on the average advance $d*\frac{\mu}{\lambda}$ while the message is in-transit, the probability of rollback and the amount of wasted simulation time increases accordingly. Here, we have subtracted the equivalent delay of the simulation model in real time from the delay in the simulator. Therefore, if in the simulation model a similar delay in interaction between submodels exist, the effect of the delay in the simulator is smaller.

4.3.3.2. Numerical and Simulation Results

In Figures 4-10 to 4-13 the effects of non-zero rollback processing time and communication delay are shown by plotting ρ versus p from (37) for different values of r and d. The curves are lower bounds for the speedup. On the same Figures, the results of the simulation of a similar model without the small interaction probability assumption are also shown. Note that the lower bound is tight and close to the simulation results for very small values of p. The analytic lower bound diverges from the simulation results as $p \rightarrow .1$. This is due to two factors (i) The independence assumption and (ii) the approximations that were made during the derivation of (37). These approximations are more reasonable when p is small. The values selected for d are relative to the event processing rate μ.

Figure 4-14 gives the simulation results for several values of d. The simulation results are obtained under the same simulation parameters given in 4.3.2. It can be seen that the increase in d, severly reduces the performance. As mentioned before, the delay d will be less effective in the speedup if a similar delay exists between submodels of the simulation model.

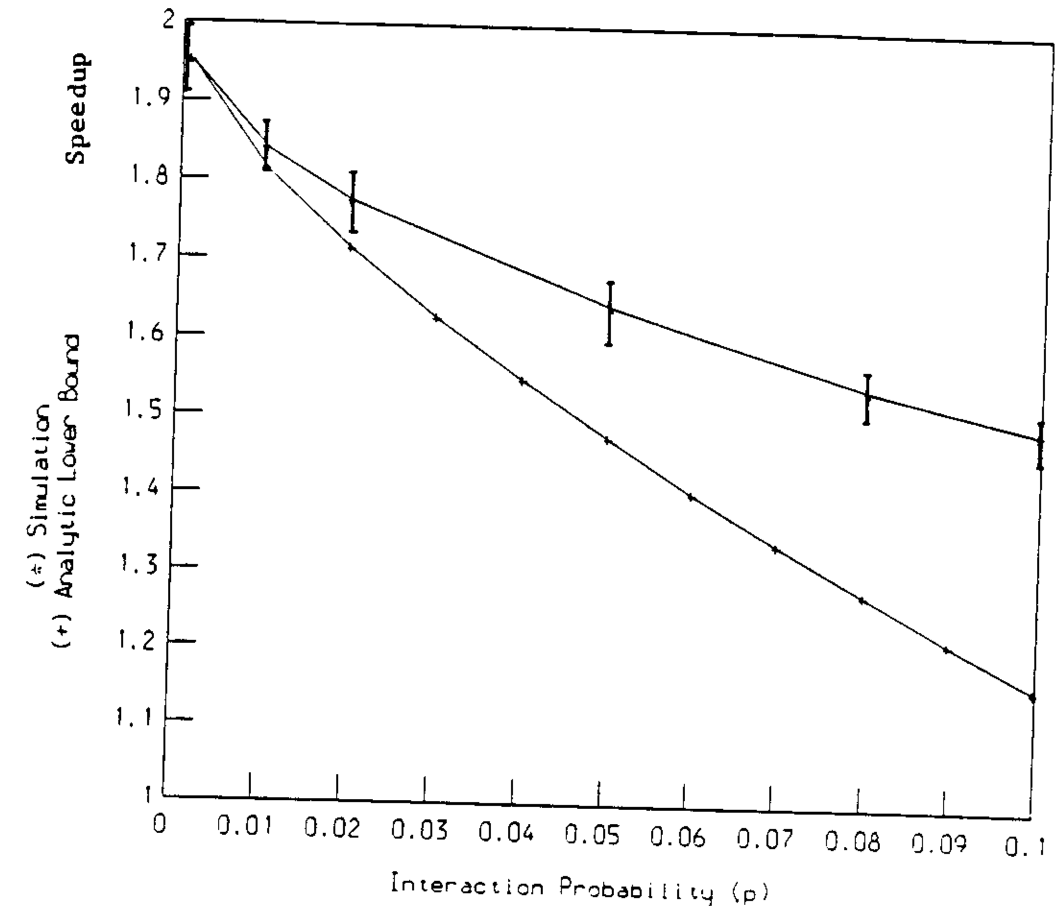

Figure 4-10. Analytic Lower Bound and Simulation Results for the Speedup,

$(d=\mu, r=0)$.

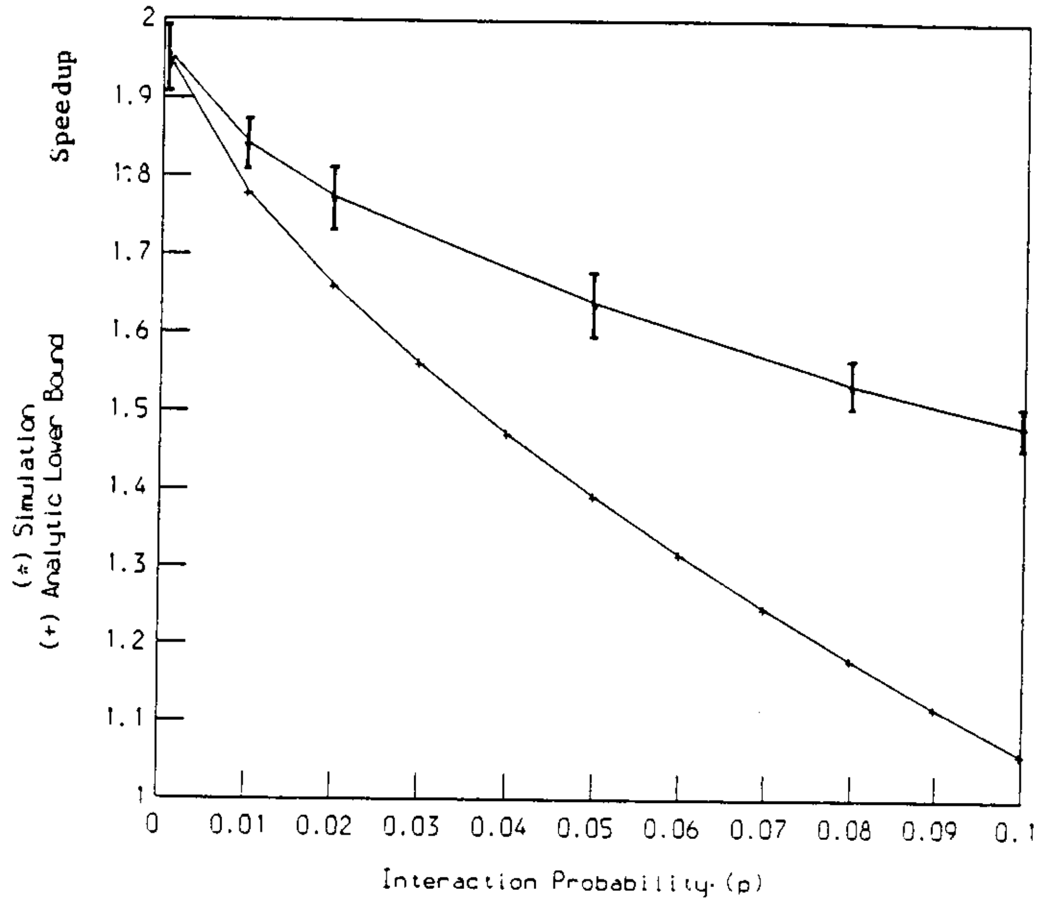

Figure 4-11. Analytic Lower Bound and Simulation Results for the Speedup, $(d=\mu, r=.1)$.

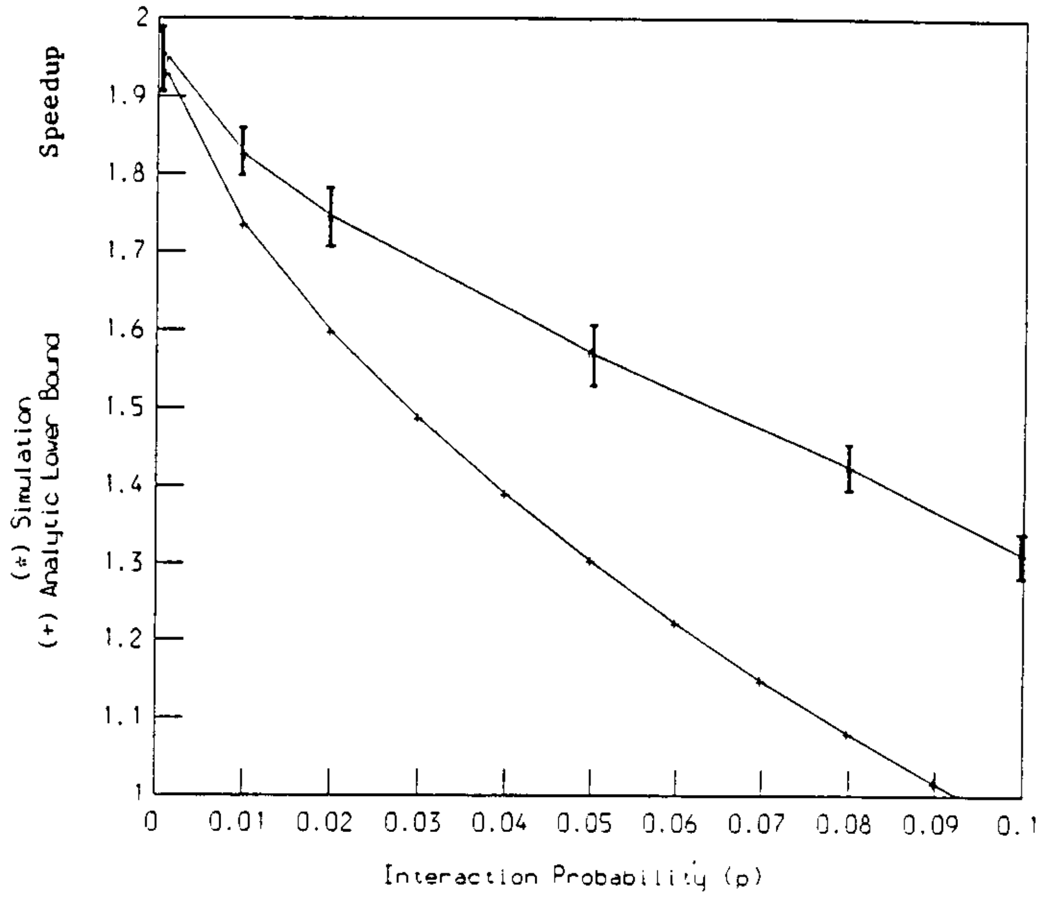

Figure 4-12. Analytic Lower Bound and Simulation Results for the Speedup,

$(d=\mu, r=.2)$.

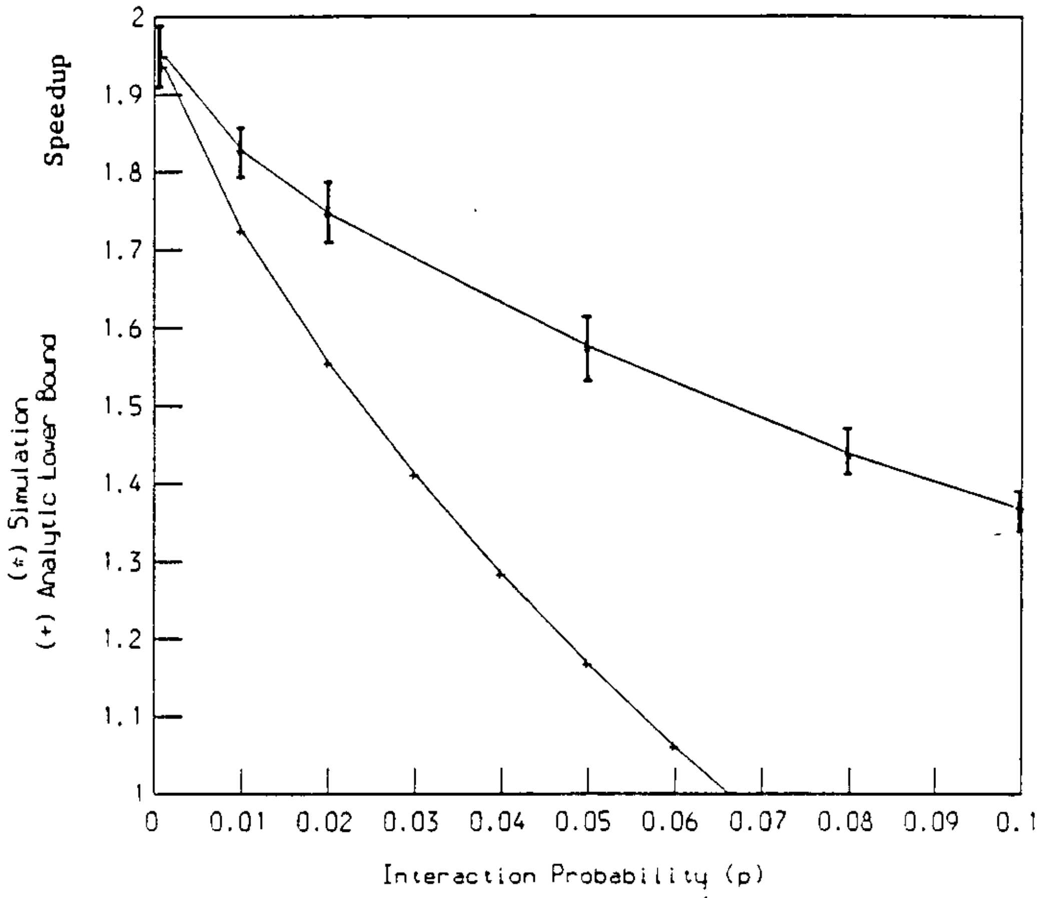

Figure 4-13. Analytic Lower Bound and Simulation Results for the Speedup,

$(d=2\mu, r=.1)$.

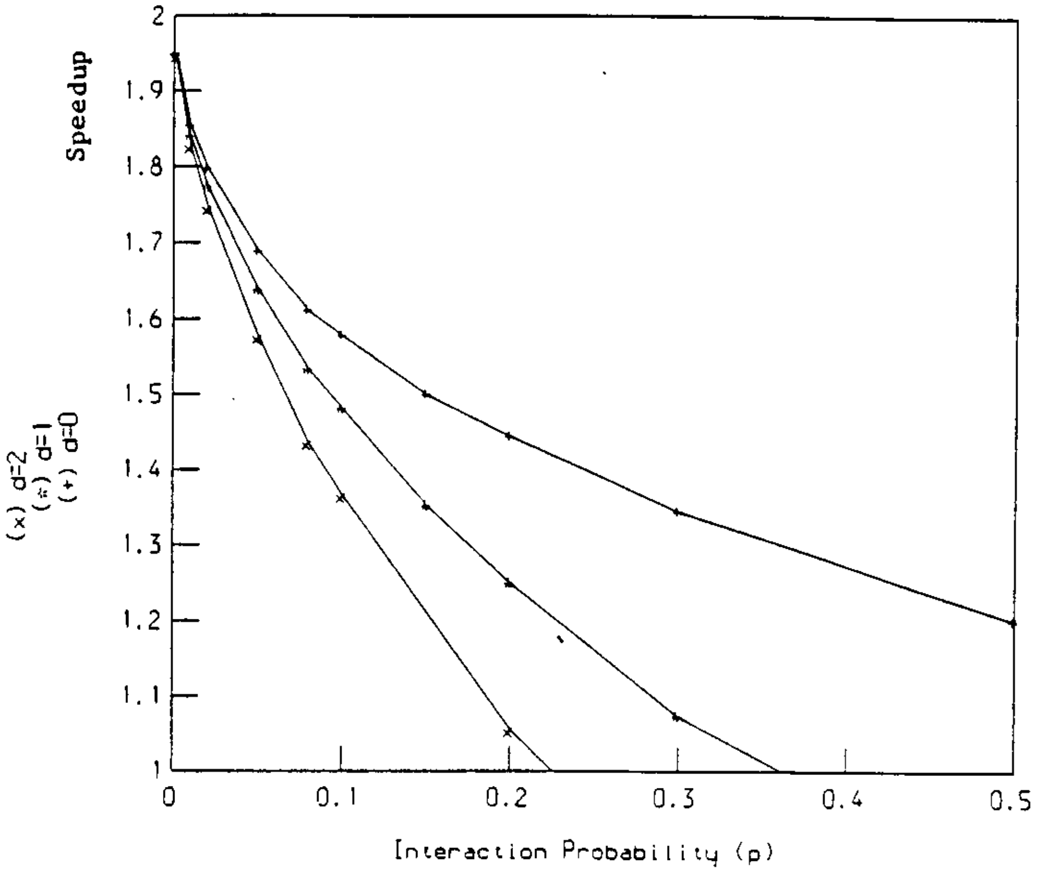

Figure 4-14. Simulation Results for the Speedup ($d \geq 0, r = .1$).

4.4. More than Two Processors

In this section the simulation results of the model of this chapter for more than two processors are presented. The model assumes that the processors and the submodels are statistically identical. Furthermore, every generated event is an inter submodel event with probability p, and any submodel is equally likely to receive an inter-submodel event.

Figure 4-15 shows the multi-processor speedup as a function of the interaction probability when the rollback processing time and the communications delay is zero. In Figure 4-16 the speedup is plotted in the case where the communications delay is equal to the time to simulate an event and the rollback processing multiplier, r is equal to 0.1. As it can be seen reasonable speedup can be obtained when p is small.

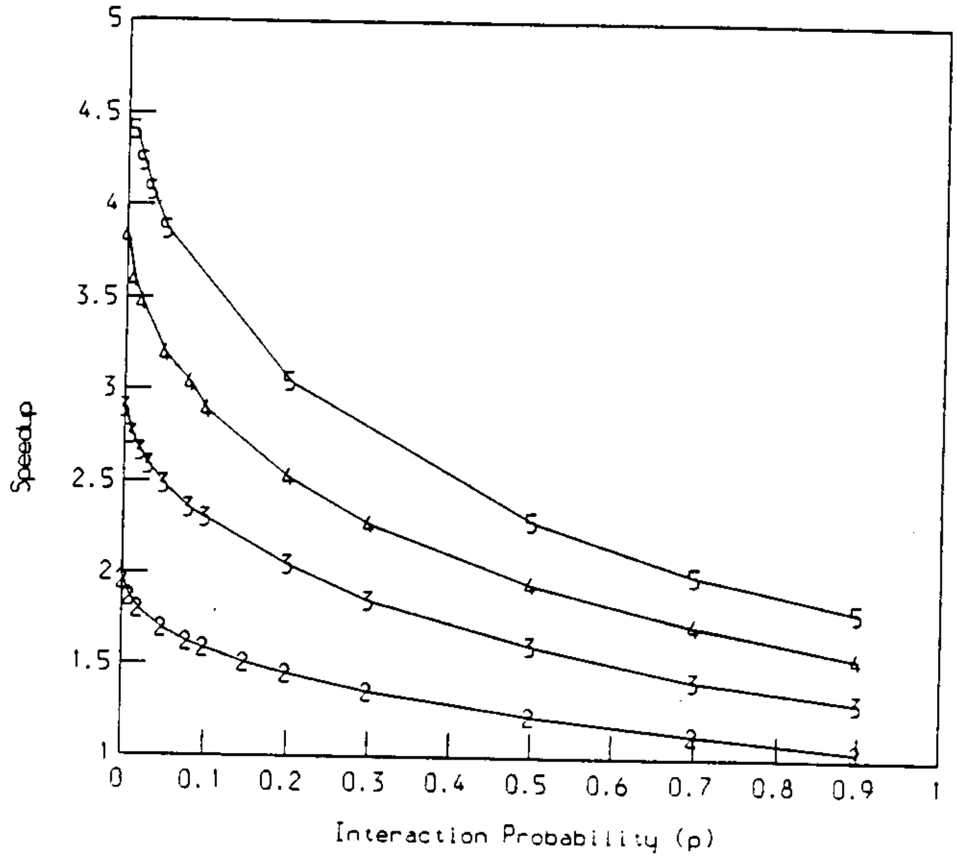

Figure 4-15. The Simulation Results for the Speedup with Multiple processors, (d=r=0).

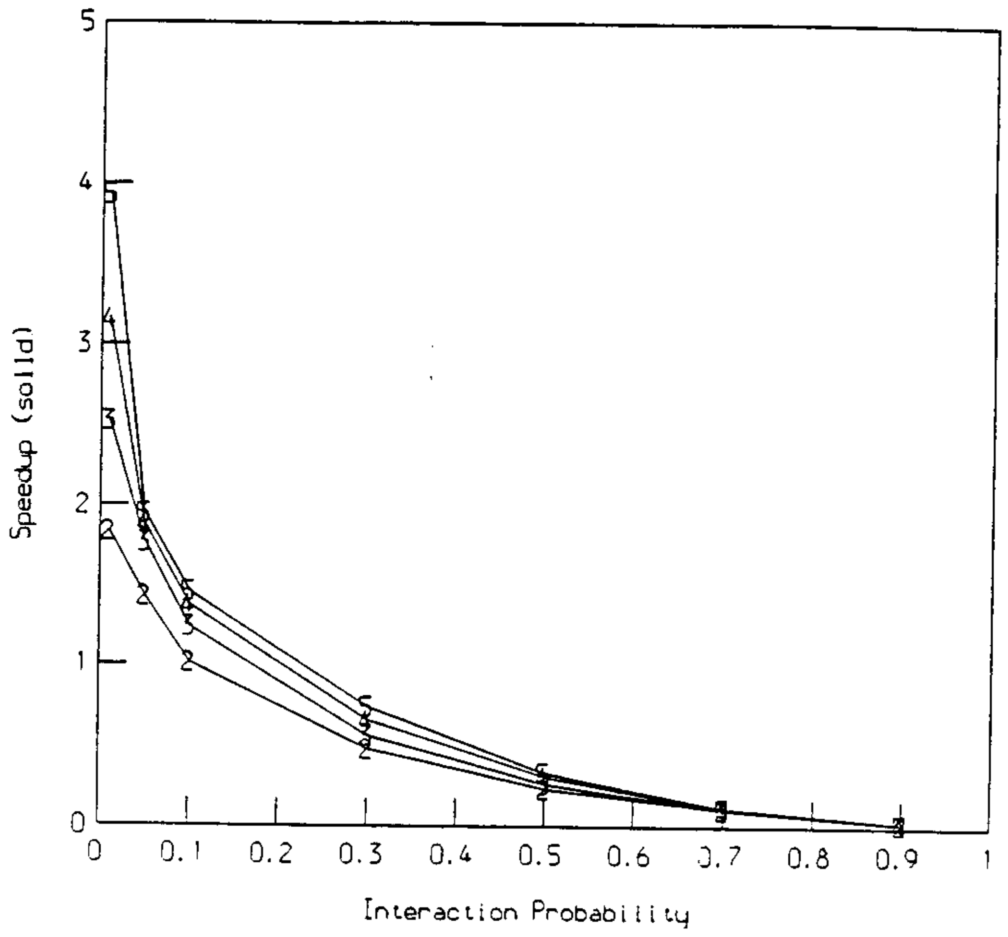

Figure 4-16. The Simulation Results for the Speedup with Multiple processors,

$(d=\mu, r=.1)$.

CHAPTER 5

Analysis of the Prediction Method

In this chapter, a model of the Link Time Algorithm for distributed simulation is analyzed. The model is based on an extension of this method with a prediction function [CHM 79].

Recall that with this method of simulation, every processor can safely simulate up to the minimum link time of its incoming links which we call the *safe simulation period.* The *link time* was defined to be the simulation time of the earliest non-simulated event message transmitted on that link. Obviously, the safe simulation period cannot be computed if one or more of the incoming links do not contain a non-simulated event message. In that case, the processor has to wait for an event message to arrive on the empty link(s). The waiting can cause deadlock.

To prevent deadlock, processors send null messages to their neighbors giving a lower bound on the simulation time of the next departing event from that processor. This information can be used by the waiting processor to compute a safe simulation time. In general, this approach is only workable if the simulation time between two generated events on a processor is bounded below by a non-zero value ϵ. In Chapter 1, it was shown that when ϵ is small, the overhead can be very large.

The extension of the prediction facility to the model, enables a processor to give information about a more distant future to its neighbors. This approach uses the characteristics of the simulation model and assumes that certain future steps of simulation are predictable by the processor. The predictions can be used by the neighbors to compute a safe simulation period, possibly farther ahead in simulation time.

Consider a simulation model in which the times of occurrence of some inter-submodel events can be predicted in advance. If the events are not canceled or modified then the predictions are called *exact*. The assumption is that each processor can predict the time of occurrence of the next inter-submodel event. Obviously, exact predictions are the most desirable. However, an acceptable property of the prediction is that the predicted time if not exact, be a non-zero lower bound on the time the event actually occurs. In the latter case, the processor receiving the lower bound can safely simulate to that time. If the event does not occur at the predicted time then nothing needs to be undone and only a new prediction is required.

An example of a simulation model where some events can be predicted is the queuing network model given in Figure 5-1. In this Figure, let queue i be simulated by processor P_i, $1 \leq i \leq 5$. It is described below how predictions can be done in different ways in this model.

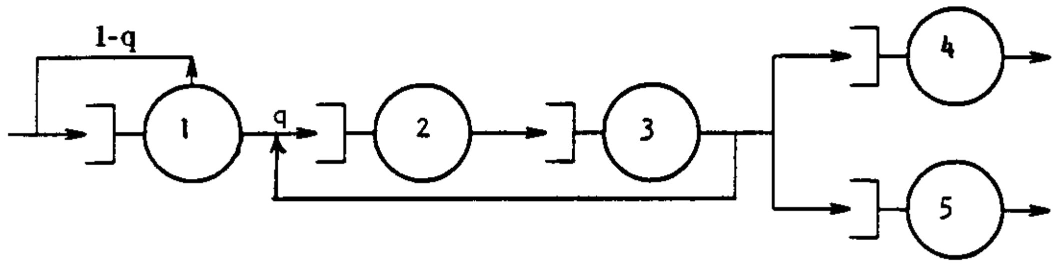

Figure 5-1. Example of a Simulation Model with Possibility of Predictions.

Given an arrival of a customer C to queue 1, P_1 can schedule or predict a departure for C given its unfinished work and its queuing discipline. The prediction is exact if the queuing discipline is FCFS. However, if a different discipline such as LCFS preemptive exists, then the best prediction that can be made is to give a lower bound on the service time of the customers. Having P_1's prediction, P_2 can also make a prediction on the time of its next departure to P_3. The correctness of this prediction depends on P_3's next event. Therefore, these two processors need to exchange predictions to agree on the next inter-submodel event(s). P_3's neighbors can benefit from the fact that it can send customers to more than one queue. Hence, whenever P_3 sends an event or a prediction to a neighbor, it can also send a prediction to the others informing that no departure events will be scheduled prior to the time of that event.

In general, making predictions may not be possible or practical. The goal of this analysis is to determine the potential speedup that may be obtained in the cases where making predictions is possible. With this analysis, a theoretical upper bound on the performance of this method of simulation is obtained. It will be seen that unless predictions are made for a time far in advance, the speedup can be very low.

5.1. The Performance Model

The assumption is that a processor i can predict s_i, the simulation time of its next departing event assuming that there are no arrivals in the mean time. This prediction is exact unless the processor receives an inter-submodel event with a simulation time less than s_i after it has made the prediction. In this case, processor i can make a new prediction only after it simulates the inter-submodel event. Therefore, the processors operate as follows :

(a) Make a prediction for the time of the next departing event.

(b) Broadcast the predictions to all the processors that can be affected by transmission of an event from that processor. With respect to the communication graph of the model, all processors that lie on a directed path originating from the processor sending the event can be affected. Note that no particular broadcast mechanism is assumed. It is possible to send the predictions to the immediate neighbors only. They will in turn pass the revised prediction to their neighbors and so on.

(c) When predictions from all the processors are received, compute $s_{\min}$, the time of the earliest predicted event.

(d) Simulate to $s_{\min}$. At that time the processors sending and receiving the event can make new predictions. Broadcast the new predicted times and goto (c). Note that in this case, we are assuming that the communication graph is fully connected.

Considering the above, we further make the following assumptions :

(1) The simulation model is divided into submodels, each being simulated by one processor.

(2) The simulation time of each submodel is updated instantaneously at the beginning of processing an event.

(3) The real time for each processor to simulate an event is iid exponential with mean $\frac{1}{\mu}$.

(4) The simulation time advances are iid exponential with mean $\frac{1}{\lambda}$ for each submodel. Thus, λ is the simulation rate for each submodel in events per unit simulation time.

(5) The real time to transmit a message between two processors is a constant d. This constant is initially assumed to be zero.

(6) The real time to make a prediction is negligible. This assumption can be relaxed in the case where $d>0$. In this case, the time to make a prediction can be included in the delay to transmit a message.

(7) The message processing time is negligible.

(8) Predictions are independent variables, based on the assumption that every generated event at a processor will be an inter-submodel event with probability p. Therefore, the predicted simulation time until next

departure from a processor is iid exponential with mean $\frac{1}{\lambda p}$.

5.2. The Performance Measure

Similar to the analysis of Chapter 4, the performance measure is the speedup that is defined to be the ratio of the useful simulation time per unit real time in a K-processor system over the same quantity in a uni-processor system when all processors have the same speed μ. Also let,

$$s_i(t) = \textit{simulation time of submodel } i \textit{ at time } t, \textit{ and}$$

$$e = \lim_{t \to \infty} \frac{\textit{useful simulation time at real time } t}{t}$$

Note with this method of simulation, all the work is useful. Then

$$e_{single\ processor} = \frac{\mu}{K\lambda}$$

where

$$\lambda = \textit{simulation time per event of a submodel}$$

and

$$e_{K\text{-}processor} = \lim_{t \to \infty} \frac{\min\limits_{i \le K}\left(s_i(t)\right)}{t}$$

Hence

$$\rho = \frac{e_{K\text{-}processor}}{e_{single\ processor}}$$

$$= \lim_{t \to \infty} \frac{\min\limits_{i \le K}\left(s_i(t)\right)\frac{K\lambda}{\mu}}{t}$$

But, the numerator is equivalent to the real time used by all processors to simulate the useful simulation time. In this model, unlike the one in Chapter 5, all the simulation time advance is useful. Therefore, the ***useful real time*** of a processor will be considered as the total time spent on simulation. Hence

$$\rho = \lim_{t \to \infty} \frac{\sum_{i=1}^{K} \textit{Useful real time of k-th processor in } t}{t}$$

Ideally, this measure is K when the processors are perfectly synchronized and do not need to wait for event arrivals.

5.3. The Two-processor Analysis

In this section an analysis of the two-processor model is presented. Figure 5-2 gives the communication graph of the simulation model and a diagram illustrating one possible scenario for the operation of the two processors during the interval (0,t) in real time.

The solid arrows in the time diagram represent transmission of event messages from one processor to the other (inter-submodel events). The dashed arrows denote the transmission of a message carrying a prediction

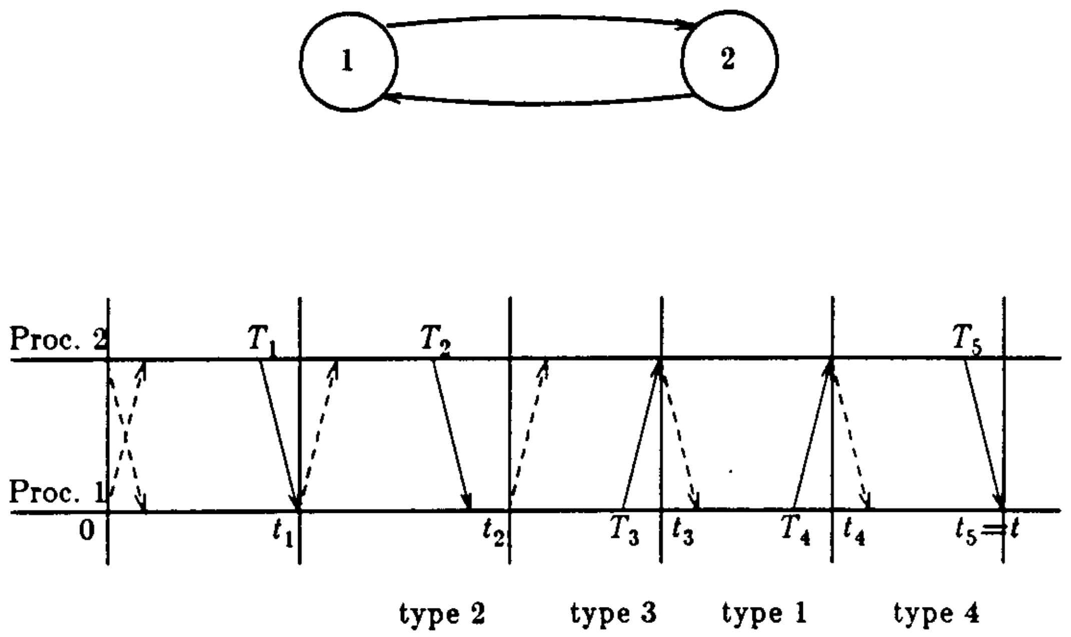

Figure 5-2. The Time Diagram and Communication Graph of the Operation of Processors.

generating an inter-submodel event, sends its predicted time for the next departure along with the event message. Let, T_i denote the real time at which the i -th inter-submodel event is generated. Note the assumption that a processor transmitting an event message has the same simulation time as that of the event at the time of transmission.

Referring to Figure 5-2, at real time zero, the processors exchange their predictions which is the simulation time of their next inter-submodel event (or departure), given no arrivals from outside. They start simulating after computing the safe simulation period to be the time period until the earlier prediction. At real time T_1, processor 2 that predicted an earlier departure, generates and transmits the first inter-submodel event. Processor 2 also sends its prediction for the simulation time of the next departure with the event message. At t_1, processor 1 that has already completed simulation of the previous safe period, receives the event message and the new prediction. Subsequently, it sends its own predicted time. Since processor 1 has both predicted values at t_1, it can immediately start simulating to the time of the next inter-submodel event. However, processor 2 can start simulating in the new interval only after it has received its neighbor's prediction. Then during an interval the following takes place :

(i) making predictions

(ii) determining the simulation time of the next inter-submodel event (departure)

(iii) simulating to the time of the next departure

(iv) transmitting the inter-submodel event

These steps are repeated until the simulation process terminates. Note that at the end of each interval, the two processors have the same simulation time.

Another point to consider about the model is that once a processor makes a prediction, it can improve on that prediction only after it has either sent or simulated an inter-submodel event. This property can be seen in Figure 5-2. Processor 1 cannot immediately respond to processor 2's prediction sent at T_2. First, it has to simulate the safe simulation period and the event and then produce its prediction at t_2.

Using the above model, the real time axis can be divided into four different types of intervals shown in Figure 5-2. The intervals start and end at the points of transmission of prediction messages by processors receiving the inter-submodel events. The type of each interval depends on the source (s) of two consecutive inter-submodel events. Without loss of generality, we define the interval types as follows :

[1] Interval formed by two consecutive departures of simulation events from processor 1.

[2] Interval formed by two consecutive arrivals of simulation events to processor 1.

[3] Interval formed by an arrival to and a following departure from processor 1.

[4] This interval is formed by a departure from, followed by an arrival to processor 1.

Let

$$t_i = \textit{real time at which the i-th interval ends} \ \ (t_0 \equiv 0)$$

$$Y_i = \textit{length of the i-th period} = t_i - t_{i-1}\,, \quad i \geq 1$$

$$N(t) = \textit{number of completed intervals by time } t, \textit{ and}$$

$$r_k^i = \textit{useful real time of processor k within the i-th interval}$$

The variables Y_i are not independent since the length of each interval depends on the sending processor of the inter-submodel event of the previous interval. However, at t_i, the duration of the next interval can be specified independently of the past. Therefore, $\{N(t) : t \geq 0\}$ is a markov renewal process [ROS 70]. Let us define an embedded (possibly delayed) renewal process to be

$$\{\bar{N}(t) : t \geq 0\} = \textit{Number of completed intervals of some type (say 1) by time } t$$

Furthermore, a *cycle* is defined to be the real time between two renewals. Also, let

$$\tilde{r}_k{}^i = \textit{Useful time of processor k in cycle i}$$

Then The speedup can be written as

$$\rho = \lim_{t \to \infty} \frac{\sum_{k=1}^{2} [\sum_{i=1}^{N(t)} \tilde{r}_k{}^i + \epsilon_k(t)]}{t}$$

$$= \sum_{k=1}^{2} \lim_{t \to \infty} \frac{\sum_{i=1}^{N(t)} \tilde{r}_k{}^i + \epsilon_k(t)}{t}$$

where

$$\epsilon_k(t) = \textit{useful time of } k \textit{ during } (t_{N}(t), t)$$

Hence

$$\rho = \frac{\sum_{k=1}^{2} E[\tilde{r}_k{}^1]}{E[\textit{length of a cycle}]} \tag{1}$$

Let I_i be the length of an interval of type i then with homogeneous processors,

$$E[\textit{length of a cycle}] = \sum_{i=1}^{4} E[I_i]$$

Note that because the processors are homogeneous, all the four types of intervals are equally likely to occur within a cycle. .

We therefore need to compute the expected length of each interval and the expected useful real time during each interval. Let

$$r_{kj} = \textit{useful real time of processor k in an interval, given processor j generates}$$
$$\textit{the next inter-submodel event}$$

Then

$$I_1 = \max(2d+r_{11}, r_{21})$$

$$I_2 = max(r_{12}, 2d+r_{22})$$

$$I_3 = d+max(r_{11}, r_{21})$$

$$I_4 = d+\max(r_{12}, r_{22}) \qquad (2)$$

Due to the homogeneity of processors,

$$r_{11} = r_{22} = r_1$$

and

$$r_{12} = r_{21} = r_2$$

Therefore,

$$I_1 = I_4 \ , \ I_2 = I_3$$

Now

$$r_1 = \sum_{i=1}^{M+1} x_i$$

where M is the number of intra-submodel events of processor k and has a poisson distribution with parameter $\lambda(1-p)$ and x_i is the real time to simulate the i- th event during an interval by a processor and $\{x_i : i \geq 1\}$ are iid exponential with mean $\frac{1}{\mu}$. We also have

$$r_2 = \sum_{i=1}^{M} x_i$$

In the latter case, the processor only simulates intra-submodel events during the interval.

Let $h_1(r)$ be the conditional density for the real time to simulate events by a processor within one interval given that processor generated the inter-submodel event at the end of that interval (density for r_1). Since r_1 is the sum of $M+1$ exponential variables, given $M = m$, we have

$$h_1(r) | M=m = \frac{\mu(\mu r)^m}{m!} e^{-\mu r}$$

But

$$Pr\{M=m | y=\textit{predicted simulation time}\} = \frac{(\lambda(1-p)y)^m}{m!} e^{-\lambda(1-p)y} \qquad m \geq 0$$

The event $\{M_k=m \textit{ during } y \textit{ predicted simulation time}\}$ is independent of which processor will generate the next inter-submodel event (see [HES 82],page 80). Hence unconditioning on m, we have

$$h_1(r) = \mu\ e^{-\lambda(1-p)y}\ e^{-\mu r} \sum_{m=0}^{\infty} \frac{(\lambda(1-p)y)^m}{m!} \frac{(\mu r)^m}{m!}$$

and

$$\begin{aligned} H_1(r) &= Pr\{r_1 \leq r\} \\ &= 1 - e^{-\lambda(1-p)y}\ e^{-\mu r} \sum_{m=0}^{\infty} \frac{(\lambda(1-p)y)^m}{m!} \sum_{i=0}^{m} \frac{(\mu r)^i}{i!} \end{aligned} \tag{3}$$

Similarly, if we let $h_2(r)$ to be the density for r_2 then given $M_k = m$,

$$\begin{aligned} h_2(r) | M=m &= \frac{\mu(\mu r)^{m-1}}{(n-1)!} e^{-\mu r} && m \geq 1 \\ &= 0 && m=0 \end{aligned}$$

Unconditioning on m,

$$h_2(r) = e^{-\lambda(1-p)y} u_0(r) + \mu e^{-\lambda(1-p)y}\, e^{-\mu r} \sum_{m=1}^{\infty} \frac{(\lambda(1-p)y)^m}{m!} \frac{(\mu r)^{m-1}}{(m-1)!}$$

where u_0 is the unit impulse function. The above formula shows that with probability $e^{-\lambda(1-p)y}$ the processor receiving the inter-submodel event, will not be advancing during the interval. Defining $H_2(r)$ to be the distribution function for r_2, we have

$$H_2(r) = 1 - e^{-\lambda(1-p)y}\, e^{-\mu r} \sum_{m=1}^{\infty} \frac{(\lambda(1-p)y)^m}{m!} \sum_{i=0}^{m-1} \frac{(\mu r)^i}{i!} \tag{4}$$

But,

$$\begin{aligned} E[r_1 \mid y] &= (E[M] + 1)E[X_i] \\ &= \frac{1+\lambda(1-p)y}{\mu} \end{aligned} \tag{5}$$

and similarly,

$$E[r_2 \mid y] = \frac{\lambda(1-p)y}{\mu} \tag{6}$$

where y is the predicted simulation time within an interval and is exponentially distributed with mean $\frac{1}{(2\lambda p)}$. Unconditioning on y,

$$\begin{aligned} E[r_1] &= \frac{1+p}{\mu p} \\ E[r_2] &= \frac{1-p}{\mu p} \end{aligned} \tag{7}$$

5.3.1. Case of Negligible Communication Delay (d=0)

At this point let d, the communications delay be equal to zero. The case of non-zero communication delay will be considered later. The equations in (2) can thus be written as

$$I = I_i = max(r_1, r_2) \quad for\ all\ i \tag{8}$$

In this case the renewals occur at the beginning of each interval. i.e the cycle includes one interval. Let $H_{\max}(r) = Pr\{I \leq r\}$, then

$$\begin{aligned}
H_{\max}(r) &= H_1(r)H_2(r) \\
&= 1 - e^{-\lambda(1-p)y}\, e^{-\mu r} \sum_{m=0}^{\infty} \frac{(\lambda(1-p)y)^m}{m!} \sum_{i=0}^{m} \frac{(\mu r)^i}{i!} \\
&\quad - e^{-\lambda(1-p)y}\, e^{-\mu r} \sum_{n=1}^{\infty} \frac{(\lambda(1-p)y)^n}{n!} \sum_{j=0}^{n-1} \frac{(\mu r)^j}{j!} \\
&\quad + e^{-(2\lambda(1-p))y}\, e^{-(2\mu)r} \left[\sum_{m=0}^{\infty} \frac{(\lambda(1-p)y)^m}{m!} \sum_{i=0}^{m} \frac{(\mu r)^i}{i!} \right] \\
&\quad \left[\sum_{n=1}^{\infty} \frac{(\lambda(1-p)y)^n}{n!} \sum_{j=0}^{n-1} \frac{(\mu r)^j}{j!} \right]
\end{aligned}$$

Therefore,

$$\begin{aligned}
E[I|y] &= E[r_1|y] + E[r_2|y] \\
&\quad - e^{-(2\lambda(1-p))y} \sum_{m=0}^{\infty} \frac{(\lambda(1-p)y)^m}{m!} \sum_{n=1}^{\infty} \frac{(\lambda(1-p)y)^n}{n!} \sum_{i=0}^{m} \frac{\mu^i}{i!} \sum_{j=0}^{n-1} \frac{\mu^j}{j!} \int_0^{\infty} e^{-(2\mu)r} r^{i+j}\, dr \\
&= E[r_1|y] + E[r_2|y] \\
&\quad - e^{-(2\lambda(1-p))y} \left[\sum_{m=0}^{\infty} \frac{(\lambda(1-p)y)^m}{m!} \sum_{i=0}^{m} \frac{(\lambda(1-p)y)^n}{n!} \right] \sum_{i=0}^{m} \frac{\mu^i}{i!} \sum_{j=0}^{n-1} \frac{\mu^j}{j!} \frac{(i+j)!}{(2\mu)^{i+j+1}}
\end{aligned}$$

$$= \frac{1}{\mu p} - \frac{p}{2\mu} \sum_{m=0}^{\infty} \sum_{n=1}^{\infty} \frac{(m+n)!}{m!n!} \left(\frac{1-p}{2} \right)^{m+n} \sum_{i=0}^{m} \sum_{j=0}^{n-1} \frac{(i+j)!}{i!j!} (\frac{1}{2})^{i+j} \tag{9}$$

The proof for convergence of the infinite sum appears in the Appendix B.

From (1)-(9) the speedup can be written as

$$\begin{aligned} \rho &= \frac{E[r_1] + E[r_2]}{E[I]} \\ &= \frac{E[r_1] + E[r_2]}{E[r_1] + E[r_2] - \Delta} \\ &= \frac{\frac{1}{\mu p}}{\frac{1}{\mu p} - \Delta} = \frac{1}{1 - \mu p \Delta} \end{aligned} \tag{10}$$

where

$$\Delta = \frac{p}{2\mu} \sum_{m=0}^{\infty} \sum_{n=1}^{\infty} \frac{(m+n)!}{m!n!} \left(\frac{1-p}{2} \right)^{m+n} \sum_{i=0}^{m} \sum_{j=0}^{n-1} \frac{(i+j)!}{i!j!} (\frac{1}{2})^{i+j} \tag{11}$$

The numerical results of (10) appear in the next section.

5.3.2. Numerical and Simulation Results

In Figure 5-3, the speedup ρ is plotted as a function of interaction probability p, from equation (10). The results are obtained by numerically evaluating Δ. The upper bound on the computation error is 10^{-5}. The derivation for an upper bound on the error appears in Appendix B.

5.3.2. Numerical and Simulation Results

In Figure 5-3, the speedup ρ is plotted as a function of interaction probability p, from equation (10). The results are obtained by numerically evaluating Δ. The upper bound on the computation error is 10^{-5}. The derivation for an upper bound on the error appears in Appendix B.

To verify the analytic model, a simulation of a similar model is performed. The results of the simulation in the form of 95% confidence intervals also appear in Figure 5-3. The simulation results are obtained from 10 independent runs of the simulation program for the duration of 7000 simulation events. It can be seen that the two results agree and reasonable speedup (close to 1.5) can be obtained for $p < 0.15$.

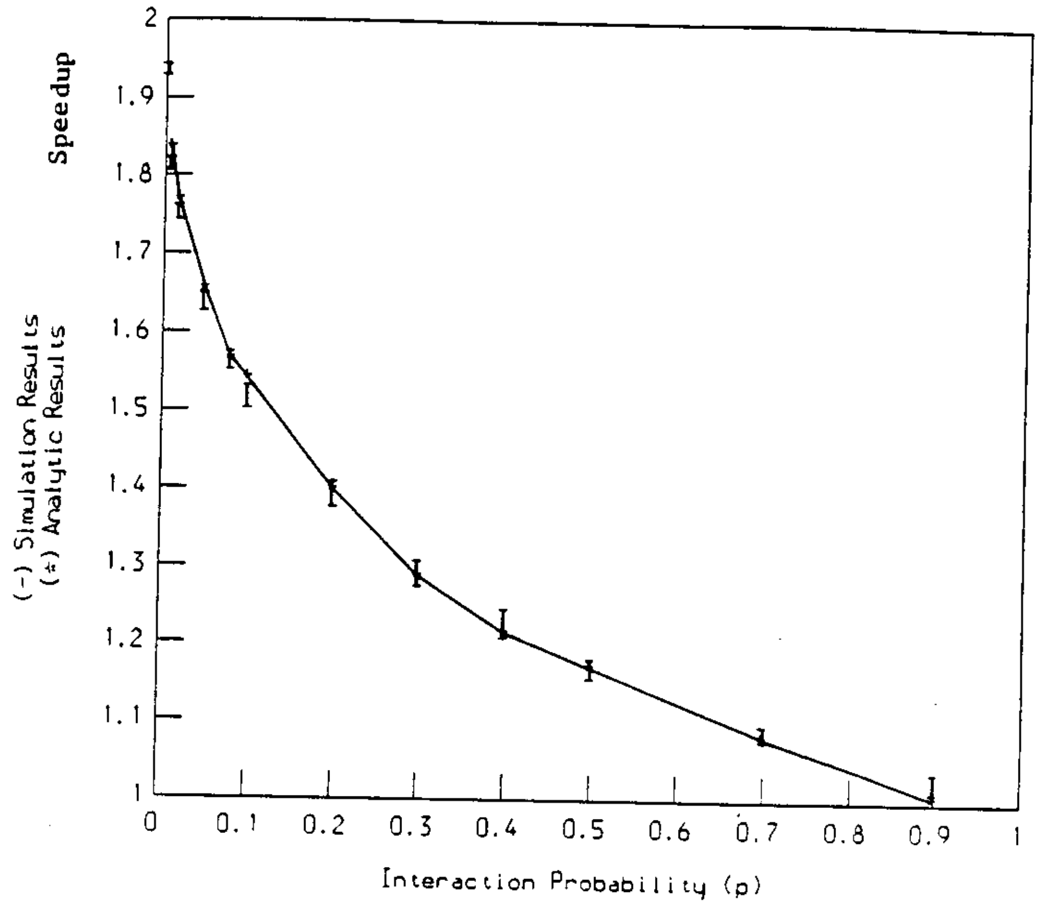

Figure 5-3. Analytic and Simulation Results for the Speedup (d=0).

5.3.3. Case of non-zero Communication Delay (d>0)

In this case the assumption of homogeneous processors still holds. We can obtain an upper bound on the performance as follows. From (2)

$$I_1 = I_2 = \max(2d+r_1,r_2)$$

$$I_3 = I_4 = d + max(r_1,r_2)$$

Hence

$$E[I_1] = E[I|d=0] + d$$

$$E[I_2] \leq E[I|d=0] + 2d \qquad (12)$$

With assumption of homogeneity, any interval is equally likely to occur, hence

$$\rho \geq \frac{E[r_1] + E[r_2]}{1.5d+E[r_1]+E[r_2]-\Delta} \qquad (13)$$

$$= \frac{1}{1.5\mu pd+1-\mu p\Delta}$$

where Δ is given by (11).

Figures 5-4 and 5-5 show the numerical evaluation of (13) as a function of p, the interaction probability for different values of d. The upper bound on the computation error is 10^{-5}. On the same Figures the simulation results appear. The simulation results for the speedup for different values of d are shown on Figure 5-6. Similar to the model of Chapter 4, the speedup degrades severely as d grows. Furthermore, the bound obtained by analysis becomes loose as p increases.

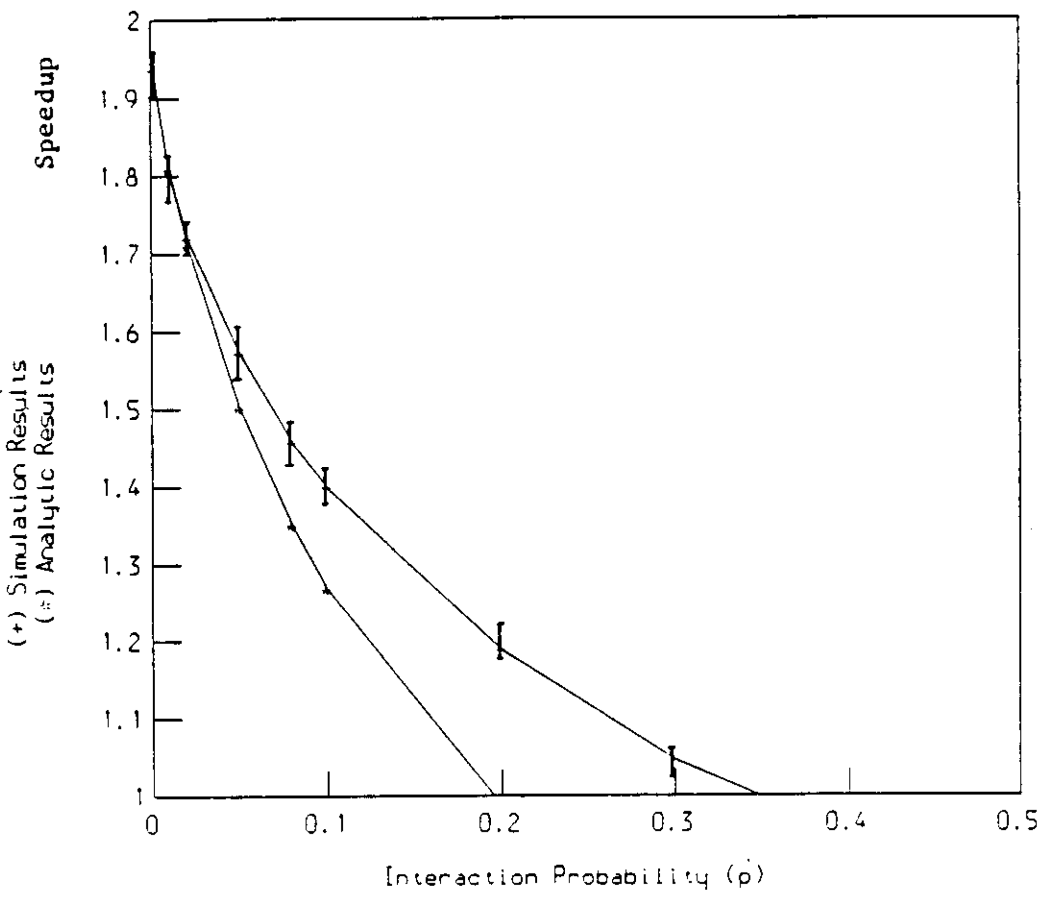

Figure 5-4. Analytic Lower Bound and Simulation Results for the Speedup,

$(d=\mu)$.

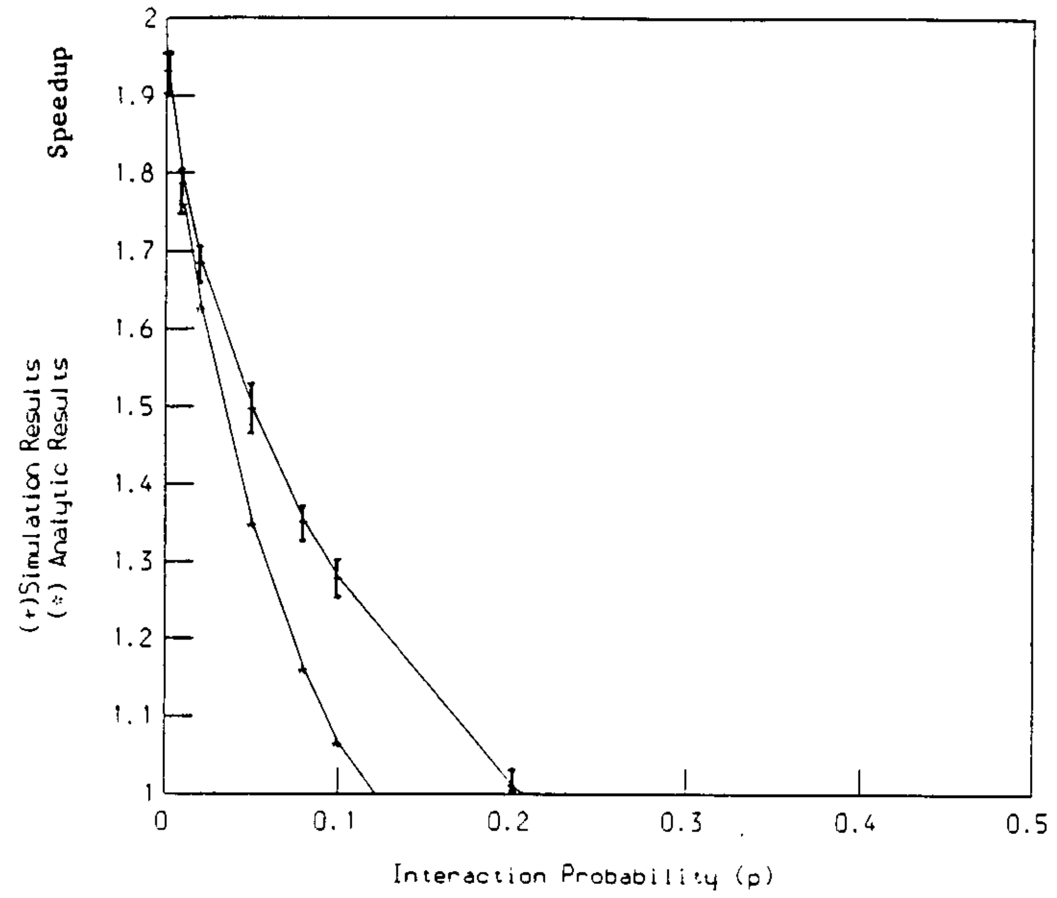

Figure 5-5. Analytic Lower Bound and Simulation Results for the Speedup,

(d=2μ).

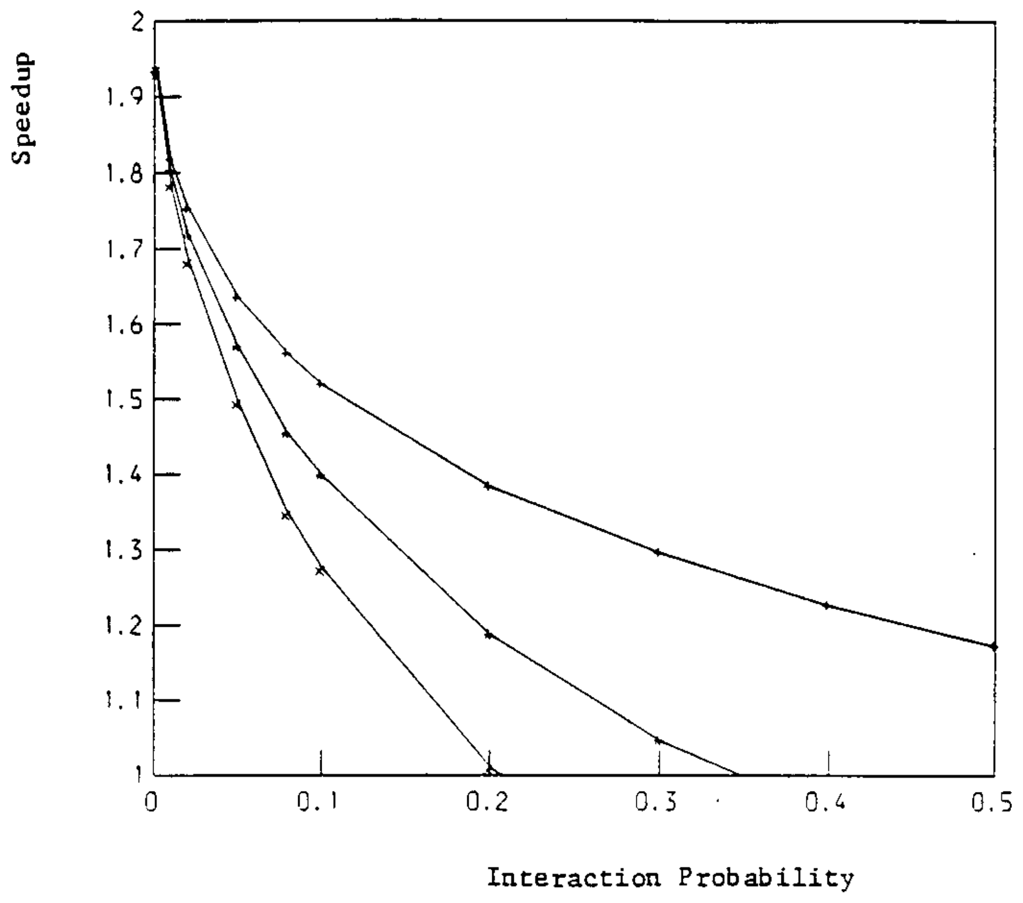

Figure 5-6. Simulation Results for the Speedup ($d>0$).

5.4. More than Two Processors

In the case where more than two processors are performing the simulation, a processor k can be affected by an inter-submodel event if there is a directed path of length 1 or more from the processor generating that event to k in the communication graph. Hence, even if the event is not directly sent to k, it could still be affected by it. In this section, we make the assumption that the communication graph is strongly connected, so there is a directed path between any two processors. With this assumption, processors have to broadcast their predictions. Then each can safely simulate to the time of the event that is predicted to occur earliest. At that time, the processors sending and receiving the predictions will make new predictions and broadcast the predicted values. A new simulation interval starts when processors compute the time of the next inter-submodel event and start simulating to that time. Figure 5-7 gives the time diagram of a possible scenario for the operation of three processors.

It can be seen that similar intervals to those of the two-processor model can be defined. Without loss of generality, we define the intervals to be of the following types :

[1] Interval formed by two consecutive departures of simulation events from processor 1.

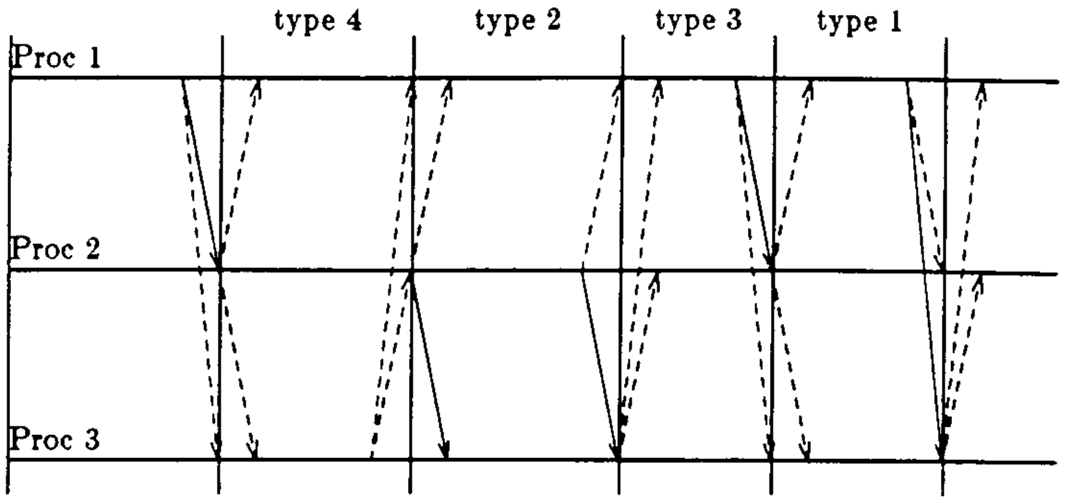

Figure 5-7. A Time Diagram for the Operation of Three Processors.

[2] Interval formed by two consecutive departures of simulation events from any processor(s) except 1.

[3] Interval formed by a departure from any processor except 1 followed by a departure from 1.

[4] Interval formed by a departure from processor 1 followed by a departure from any processor except 1.

Unlike the two-processor case, the frequency of occurrence of these intervals are not equal. Also, $\{N(t) : t>0\}$, is not a renewal process. Recall that N(t) is the number of completed intervals by time t. Note that only two processors are necessarily synchronized at the end of the interval. One or more of the remaining processors may have not completed simulating the predicted time.

Now, if we modify the model such that all processors have to complete the predicted time in a cycle, before a new cycle starts, then $\{N(t) : t>0\}$, is a renewal process. In this modified model when d=0, the speedup is

$$\rho_{K\text{-processors}}(modified) \geq \frac{\sum_{i=1}^{K} E[R_i]}{E[\max_{1\leq i\leq K}(R_i)]}$$

where

$$R_i = \textit{useful real time for processor } i \textit{ in an interval}$$

One of the variables $R_i, 1\leq i\leq K$ is the useful real time of the processor that generated the inter-submodel event at the end of that interval has a distribution given by (3). The remaining R's have a distribution given by (4). So

$$\rho_{K\text{-processors}}(modified) = \frac{E[r_1] + (K\text{-}1)E[r_2]}{E[\max_{i\leq K\text{-}1}(r_1, r_2^i)]}$$

where r_1's distribution is given by (3) and r_2^i is the i-th variable in the order statistics $\{r_2^k : k=1,...,K\text{-}1\}$ with a parent distribution given by (4).

Clearly, the speedup in the original model is greater than that in the modified model. The processors are not forced to synchronized at the end of each interval and therefore may not need to wait as much for the others to complete. Hence,

$$\rho_{K\text{-processors}} \geq \frac{E[r_1] + (K-1)E[r_2]}{E[\max\limits_{i \leq K-1}(r_1, r_2^i)]}$$

$$= \frac{\frac{1}{\mu p}}{E[\max\limits_{i \leq K-1}(r_1, r_2^i)]}$$

An explicit formula cannot be obtained for the denominator of the above formula. However, the model has been simulated and the results appear in Figure 5-8. The simulation results for the case where the d is non-zero appears in Figure 5-9.

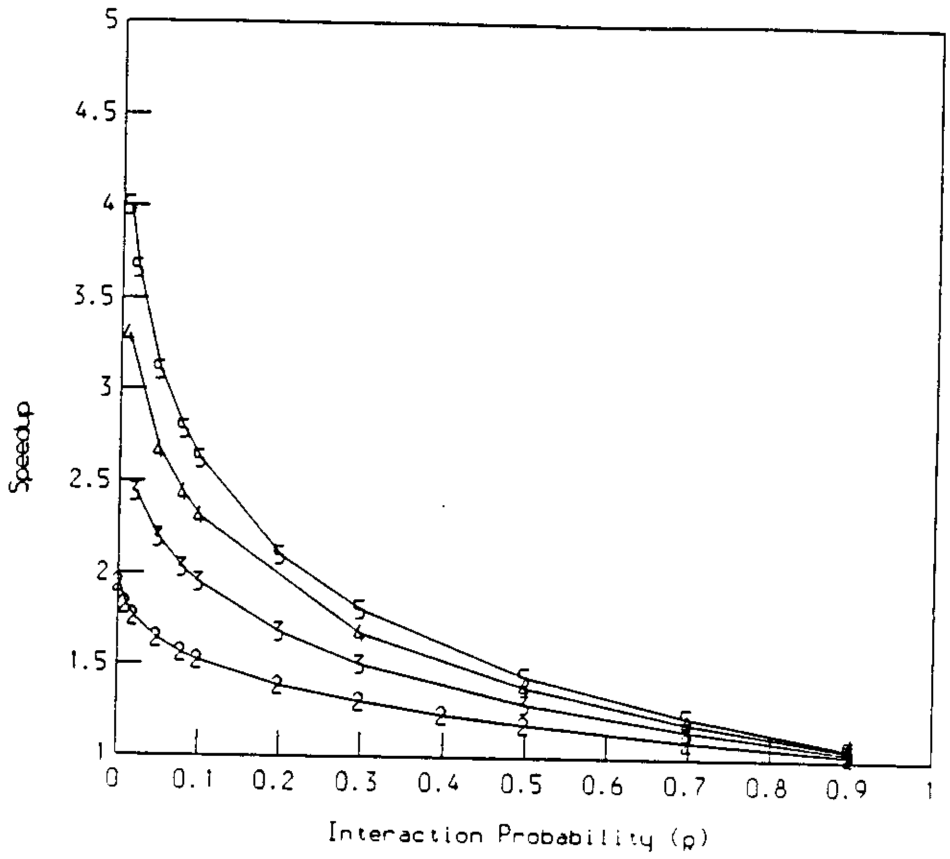

Figure 5-8. Simulation Results for the Speedup with Multiple Processors (d=0).

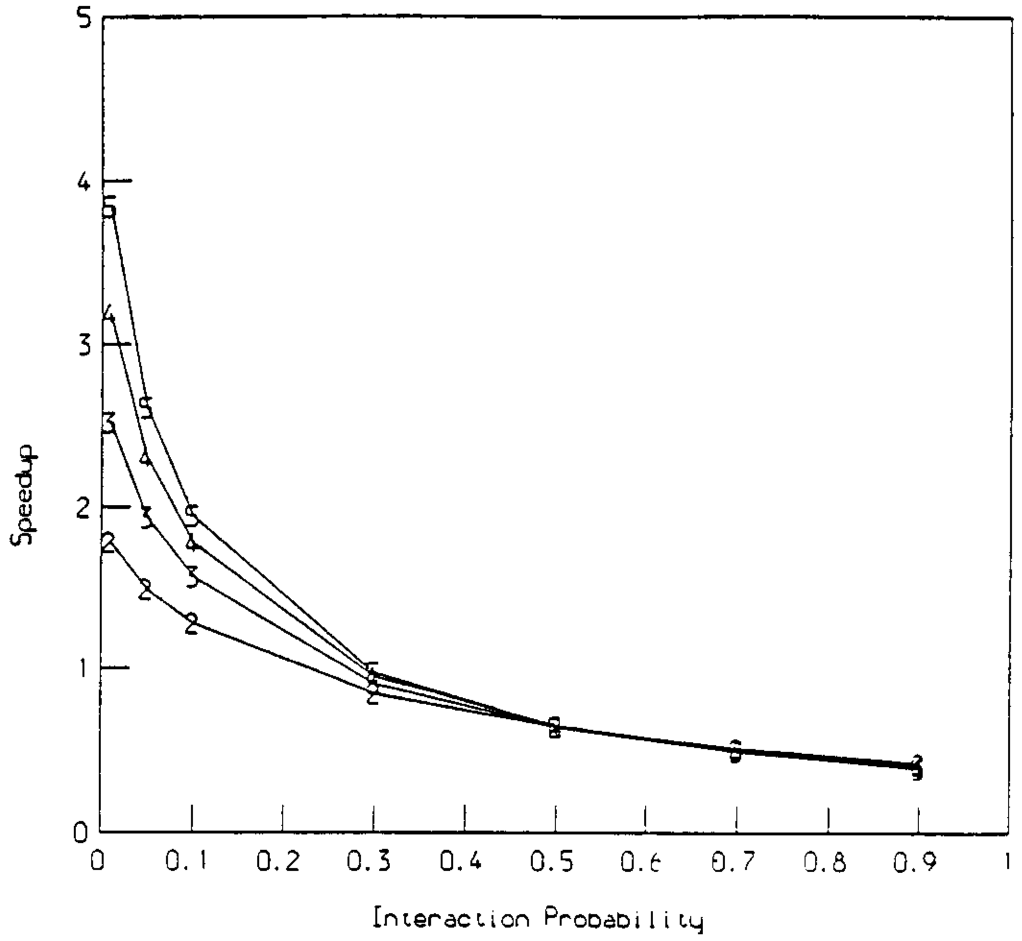

Figure 5-9. Simulation Results for the Speedup with Multiple Processors ($d=\mu$).

5.5. Comparing the Two Models

The analysis of Chapters 4 and 5 of the two methods for distributed simulation (Chapters 4 and 5), give us some insight into the effects of different system parameters on the speedup. The models are statistically similar and both are simplifications of reality. For example in the model for the Rollback method, the storage requirement was not included and in the second model (Chapter 5), making predictions was assumed to be possible and furthermore they were exact. The idea is to see whether any speedup can be obtained under these simplified conditions and if so, how would other parameters affect the results. It was shown that considerable speedup can be achieved under some conditions. Furthermore, in the absence of those conditions even the simple model does not perform efficiently. The results of the Rollback method will be confirmed with implementation results in the next chapter.

In comparison with each other, Figures 5-10 to 5-12 present the analytic and simulation results of the 2-processor model of each method. Figure 5-10 gives the speedup obtained by analysis of the two methods in region $p<.08$, which is the range where the independence assumption of the Rollback model is reasonable. The two curves are fairly close, with the curve for the Rollback method slightly higher at all points.

Figure 5-11 shows the simulation results of the two methods for all values of p when the communication delay is negligible. Finally, in Figure 5-12, the speedup is plotted as a function of interaction probability from simulation of the case where $d=\mu$. Here also the two curves are close with the curve for the speedup of the Rollback model ahead of the one for the Prediction model.

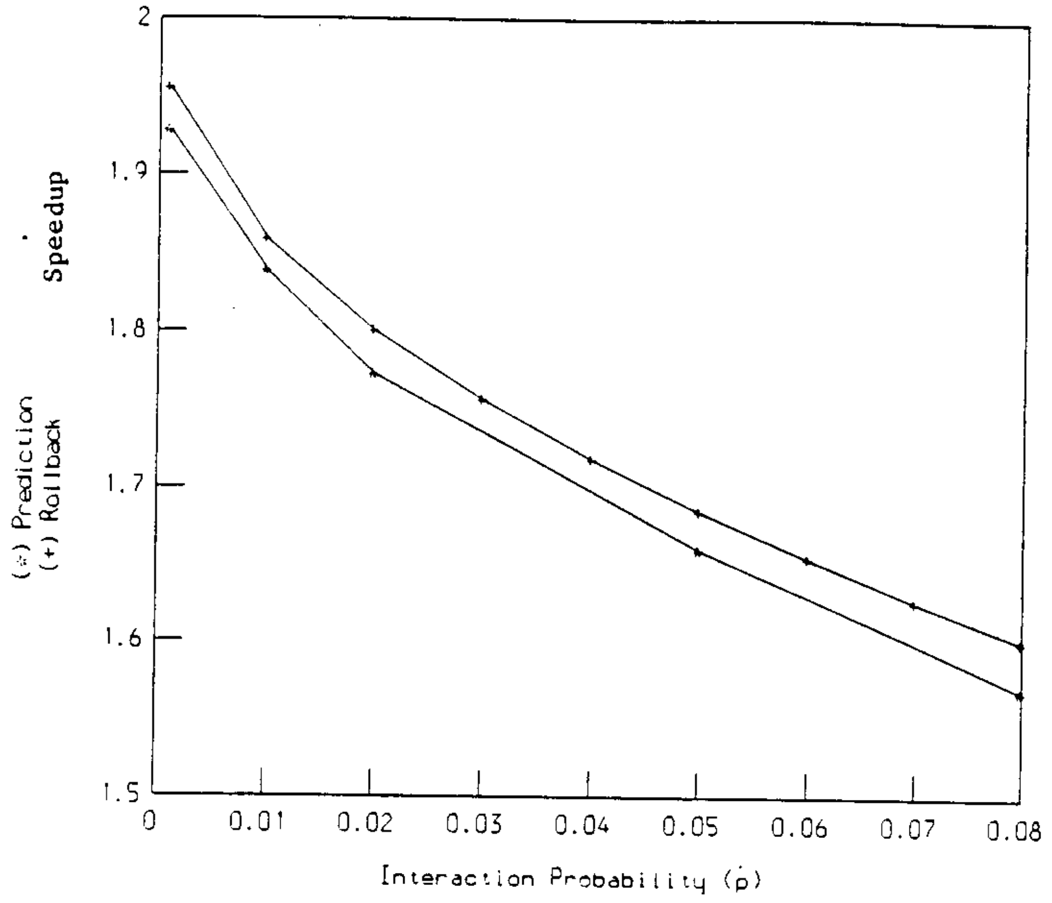

Figure 5-10. Analytic Results for the Speedup of the Two Methods ($K=2, d=0$).

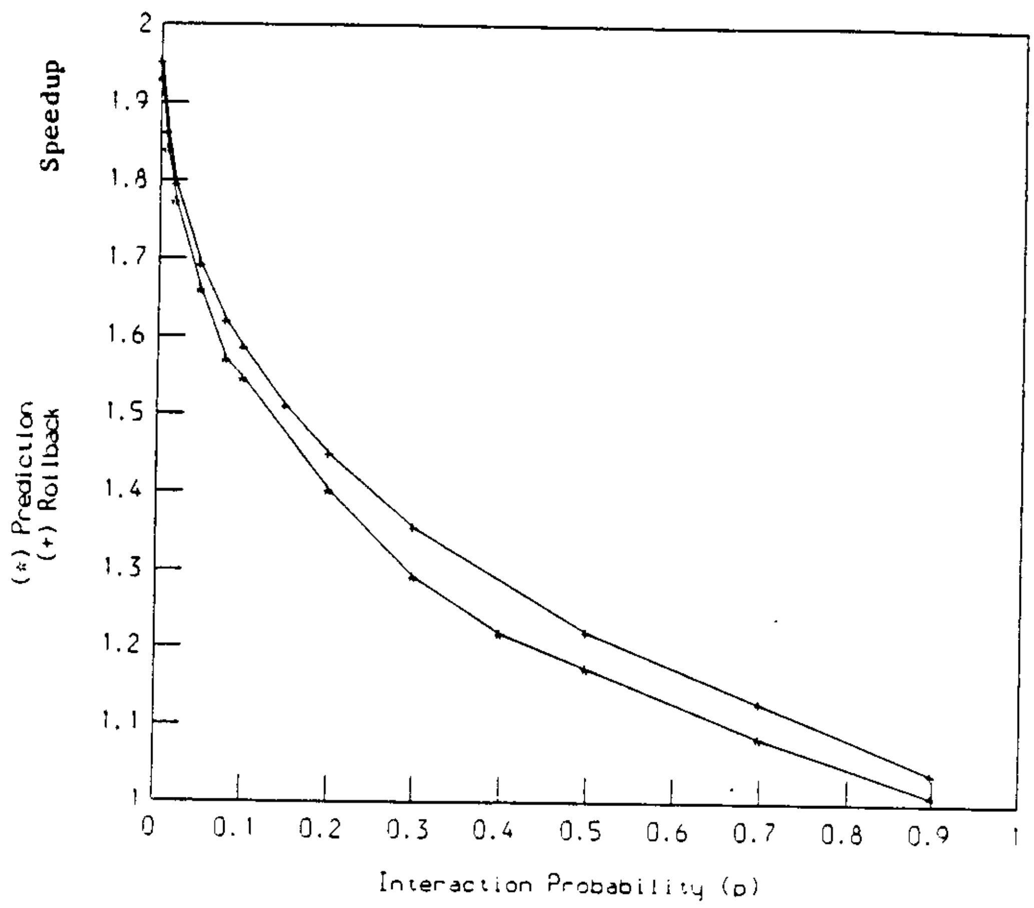

Figure 5-11. Simulation Results for the Speedup of the Two Methods,

(K=2,d=0).

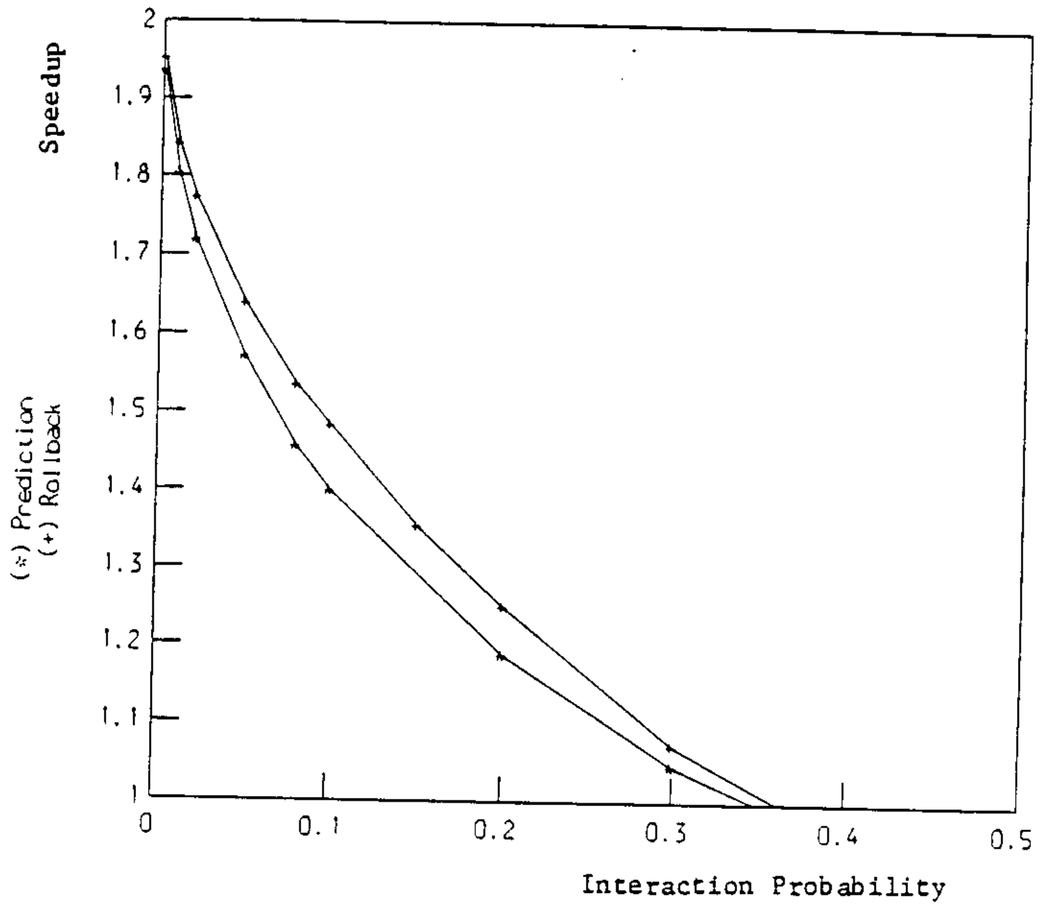

Figure 5-12. Simulation Results for the Speedup of the Two Methods,

$(K=2, d=\mu)$.

CHAPTER 6

Implementation Results

In this chapter the results of implementation of the Time Warp method are presented. This method for distributed simulation was implemented at the Rand Corporation and is running on a network of five Xerox 1100 machines (Dolphin) and a Vax 750. The Dolphins run a version of Interlisp programming environment and are used for simulation. The Vax is used to gather statistics about the system. The code which is written in GLISP was developed by Henry Sowizral and David Jefferson.

The user can request one or more processors to perform the simulation. Depending on the availability, all or fewer processors are assigned to perform the task. The processors are homogeneous in terms of software and almost similar in terms of hardware. The available physical and virtual memory of all processors except one is 2304 and 5655 pages of 512 bytes respectively. The single rather larger processor in terms of storage has physical and virtual memory of 3072 and 6707 pages. The processors are independent except at some points at which one acts as an initiator. The processor on which the user starts up the system initiates the operations on all others by loading the simulation module. The processors interact with each other at the points of transmission of events. Another operation which needs an initiator is the GVT computation. At the beginning of each GVT computation phase, one processor becomes the initiator and leads the computation. Nevertheless, as far as the simulation process is concerned, all processors are similar and no global control exists.

The termination criteria is a desired value for GVT. Once the GVT reaches this value the simulation stops and the required statistics are transmitted to the statistics gatherer. Also, throughout the simulation process, the intermediate statistics are transmitted.

Several simulation application programs have been implemented. The application that has been selected for the results that follow is the simulation of a mathematical game, called the Game of Life [GAR 70]. The rules of this game are briefly described below.

A checkerboard is used as the game board. Initially, some of the squares on the board are marked. The basic idea is to change the squares of the board at each generation depending on its neighboring squares. A square has eight neighbors, horizontally, vertically and diagonally. A new generation is created with the following rules :

(1) Survivals - A marked square remains marked for the next generation if two or three of its neighbors are marked.

(2) Deaths - A marked square with four or more marked neighbors dies (is unmarked) from overpopulation. A marked square with one or none

marked neighbors dies from isolation.

(3) Births - A non-marked square with exactly three neighbors is marked.

The changes for all the squares for one generation occur simultaneously.

Using the above rules with modifications in some cases, the Game of Life for different board sizes has been simulated. The simulation model consists of objects, each being a square of the Life Board. These squares are assigned to the processors with different object assignment strategies. The processors simulate their objects and transmit messages between neighboring objects if they do not reside on the same processor. The simulation time advances are deterministic and are equal to the time between two consecutive generations of the Life Board.

The simulation program was run on one to five processors for boards of sizes 12x12 and 16x16. Dividing the Life Board into 4x4 arrays, the initial state of the board consists of marked squares in positions (2,1), (2,2) and (2,3) of each array. The results are shown in the following sections. The first section contains a version which is called the Slow Life and the second is the Fast Life. The difference is in the way the system was run. For the slow version, the Time Warp code was run in interpret mode and therefore the operation is an order of magnitude slower than the fast version that was run in compiled mode. This way, the effects of heavy loads on the system could be observed. It should be noted that the simulation program was always run in interpret mode and only the message processing, rollback processing, GVT computation and statistical routines were compiled or interpreted depending on the version. Another difference between the two versions is that in the slow case one of the processors is simulating the Life Board itself. It needs to get the state of all the objects at each generation and transmit the whole picture to an output device. This was creating a bottleneck and therefore slowing down all processors. In the fast version the object Life Board was removed from the simulation and all output was sent to the statistics gatherer.

6.1. Slow Version

In Figure 6-1 , the GVT is plotted as a function of real time for 1-5 processors. This graph is the result of the simulation of a 12x12 Life Board. The simulation here and most other cases were run for 30 generations. The lower curve that is similar to a step function is the simulation time advance of a single processor when all the overhead due to distributed processing is absent. It can be seen that the highest performance is given by the four-processor simulation. However, the speedup that is obtained from 2-4 processor simulation is proportional to the number of processors. In other words

$$\rho_4 \approx 4/3\rho_3 \approx 2\rho_2$$

In the 5-processor simulation, the performance starts to degrade and even goes below the 2-processor simulation (not shown in Figure 6-1). This behavior occurs because (a) the interaction frequency is high and (b) one of the processors falls behind at some point. Another processor that is farther ahead is continuously feeding the slower processor with simulation messages. The flow of messages further reduces the simulation rate of the slower processor. This phenomena that occurred several time during different application runs, indicated the necessity for a control mechanism that will control the flow of messages at the simulation level.

Table 6-1, gives some statistics about this test case for 2-5 processors. As it can be seen, approximately 30% of the elapsed computation time was spent on

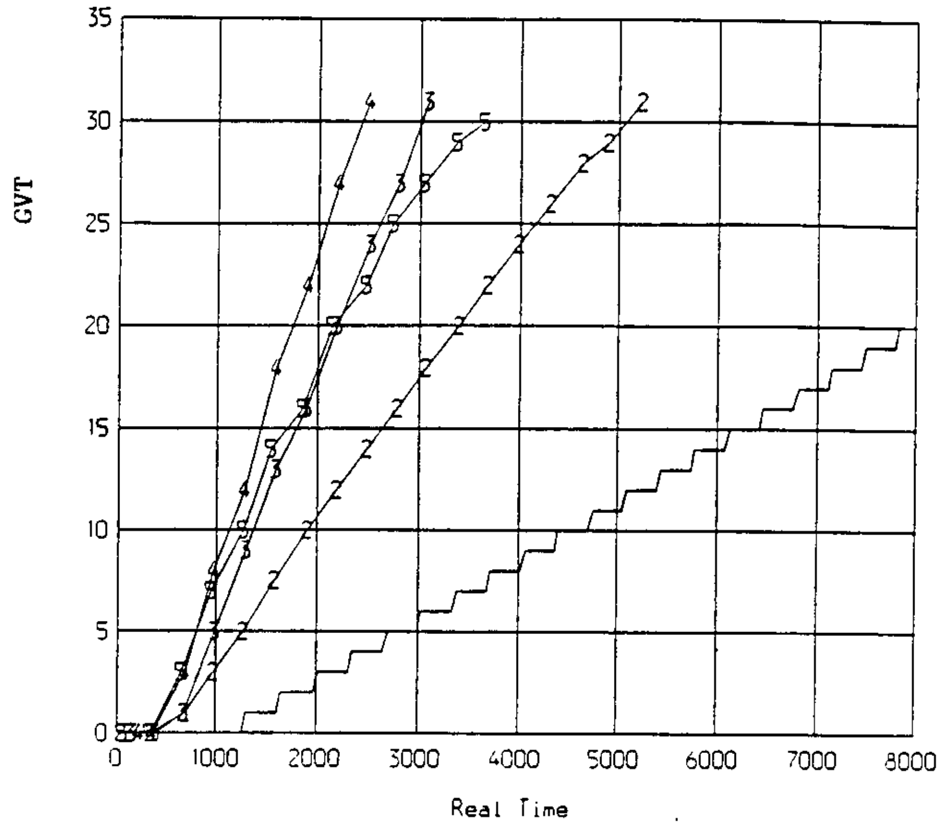

Figure 6-1. The GVT Advance for 1-5 Processors, Life 12x12 (Slow).

event processing. Furthermore, the Time Warp overhead such as GVT processing, rollback processing and scavenging utilized less than 3% of the total time. The remaining time was spent on scheduling, state saving, message processing and generating statistics. Scavenging is the process of removing obsolete entries from the queues, each time a new value for GVT is computed.

Figure 6-2 gives the processor's simulation time of this test case in the 2-processor simulation. The curve marked with a "*" is the curve for GVT. The other two curves are the simulation time curves of the two processors. The occurrences of rollback can be seen at the points of sudden negative jumps in the upper curve. The processor that rolls back, catches up with its previous simulation time rather quickly. This is due to the fact that only a subset of the objects are rolled back. After the rollback occurs, the processor may start simulation of those objects that have not been affected by the rollback.

Table 6-1. Statistics of Slow Version of Life 12x12.

	2	3	4	5
Interaction Frequency	0.14	0.13	0.3	0.34
Fraction of Time in Event Processing	0.35	0.36	0.32	0.30
Fraction of Time in Rollback Processing	0.001	0	0.005	0.004
Fraction of Time in GVT Processing	0.01	0.01	0.004	0.003
Fraction of Time Scavenging	0.02	0.02	0.01	0.02

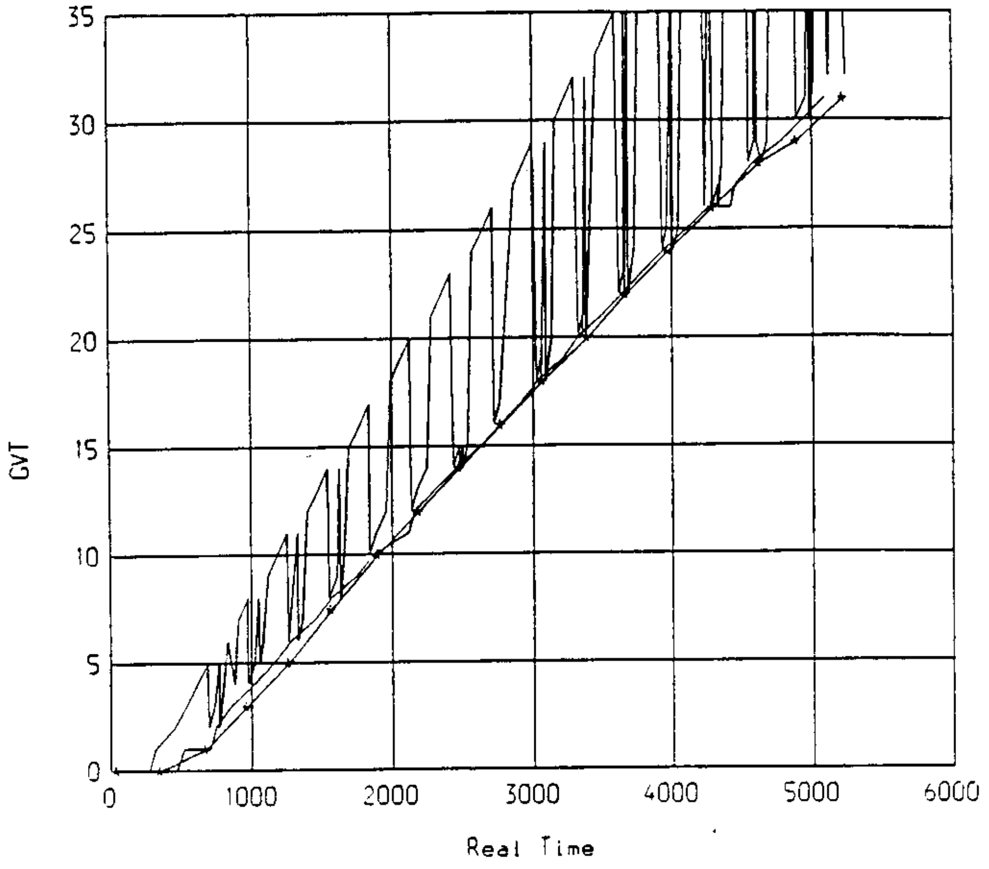

Figure 6-2. Processors' Simulation Time Advance in 2-processor Simulation, Life 12x12 (Slow).

In Figure 6-2, it can also be seen that the GVT curve closely follows the simulation time of the slower processor. This shows the relative tight lower bound on GVT that is obtained by the GVT algorithm (Chapter 2). At some points, the GVT curve crosses the lower simulation time curve. This is due to the infrequent reports of the local simulation times or infrequent savings of states.

Figure 6-3 shows the simulation times of three processors in the 3-processor simulation of the same test case. Once again the GVT curve marked with a "*", closely follows the simulation curve of the slower processor. The occurrence of rollback can be seen in one of the processors farther ahead. Another relatively fast processor does not have to rollback, since it does not need to interact with the slower processor.

Finally, Figure 6-4 shows the simulation times of four processors in the 5-processor simulation. For a more clear picture, the simulation time of one processor is deleted from the Figure. As it was mentioned before, one of the processors falls behind and its simulation rate also declines as time goes by. Note that, although other processors are far ahead, the overall simulation time cannot proceed.

The other board size that was tested is one with 16x16 squares. The results of this simulation for 1-5 processors is shown in Figure 6-5. The Figure shows the curves for GVT advance (simulation time in 1-processor case) as a function of elapsed real time. The speedup that can be obtained by 2-4 processors can be seen. However, the relative speedup of 4-processor over 3-processor simulation is negligible. 5-processor simulation once again starts to degrade. Its performance becomes so low, that at the end of the run, it is hardly advancing. It appears that the slowest processor is continuously processing messages sent by faster ones.

Table 6-2 gives the statistics about this simulation run. Similar to the previous case, approximately 30% of the time is spent on event processing. The high interaction frequency of the 5-processor simulation results to some extent to its degraded performance.

Another result that was desirable was the effect of object assignment on the performance. Initially, the above simulation runs were obtained by assigning squares (objects) to the processors in an arbitrary order. The performance for 2-processor simulation was similar to what appears in the two Figures 6-1 and 6-5. However, for more than two processors, the performance was much lower than those in the Figures. Consequently, the objects were divided between the processors, using algorithm H1 of Chapter 3. The resulting performance increases considerably for 3 and 4 processors. Figure 6-6 gives the GVT curves for 3-processor simulation of Life 16x16 with the two different object assignment strategies.

Table 6-2. Statistics of the Slow Version of Life 16.

	2	3	4	5
Interaction Frequency	0.09	0.17	0.14	0.35
Fraction of Time in Event Processing	0.33	0.33	0.34	0.30
Fraction of Time in Rollback Processing	≈0	0.001	≈0	0.004
Fraction of Time in GVT Processing	0.02	0.01	0.01	0.01
Fraction of Time Scavenging	0.02	0.02	0.01	0.02

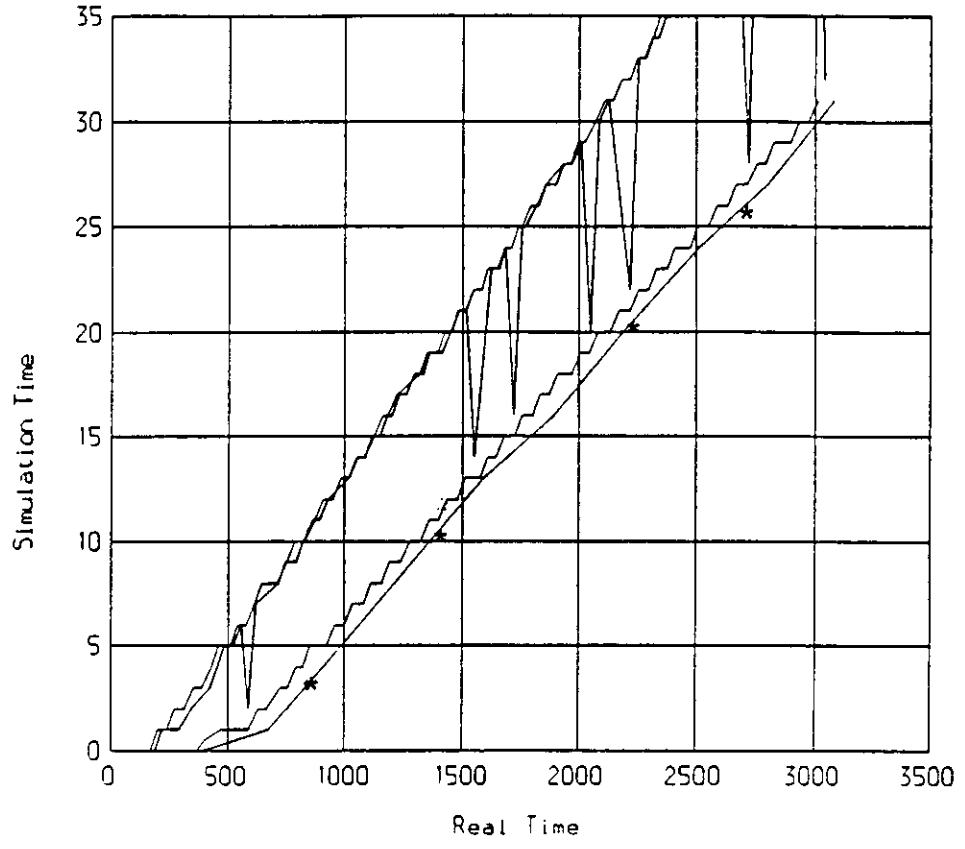

Figure 6-3. Processors' Simulation Time Advance in 3-processor Simulation, Life 12x12 (Slow).

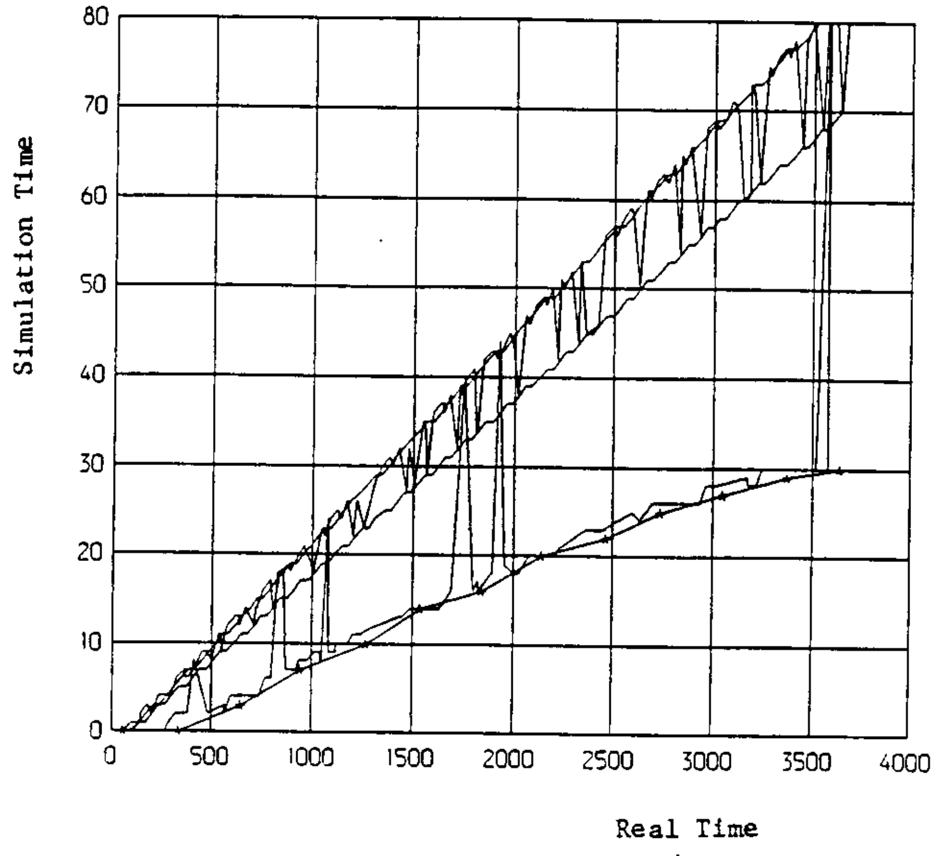

Figure 6-4. Processors' Simulation Time Advance in 5-processor Simulation, Life 12x12 (Slow).

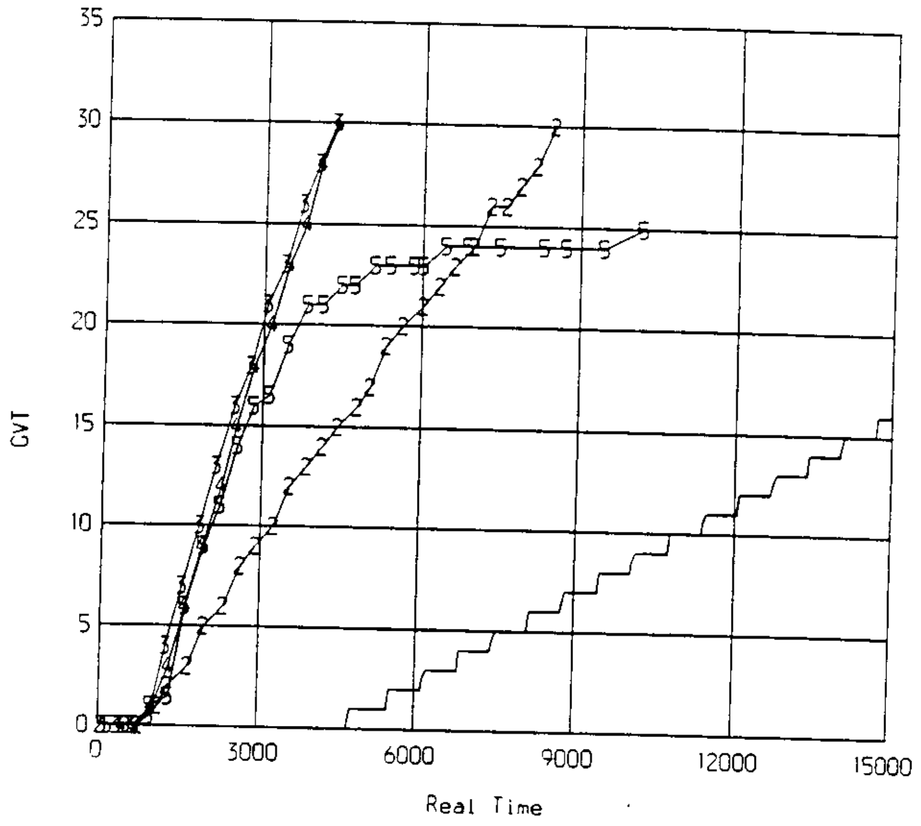

Figure 6-5. The GVT Advance for 1-5 Processors, Life 16x16 (Slow).

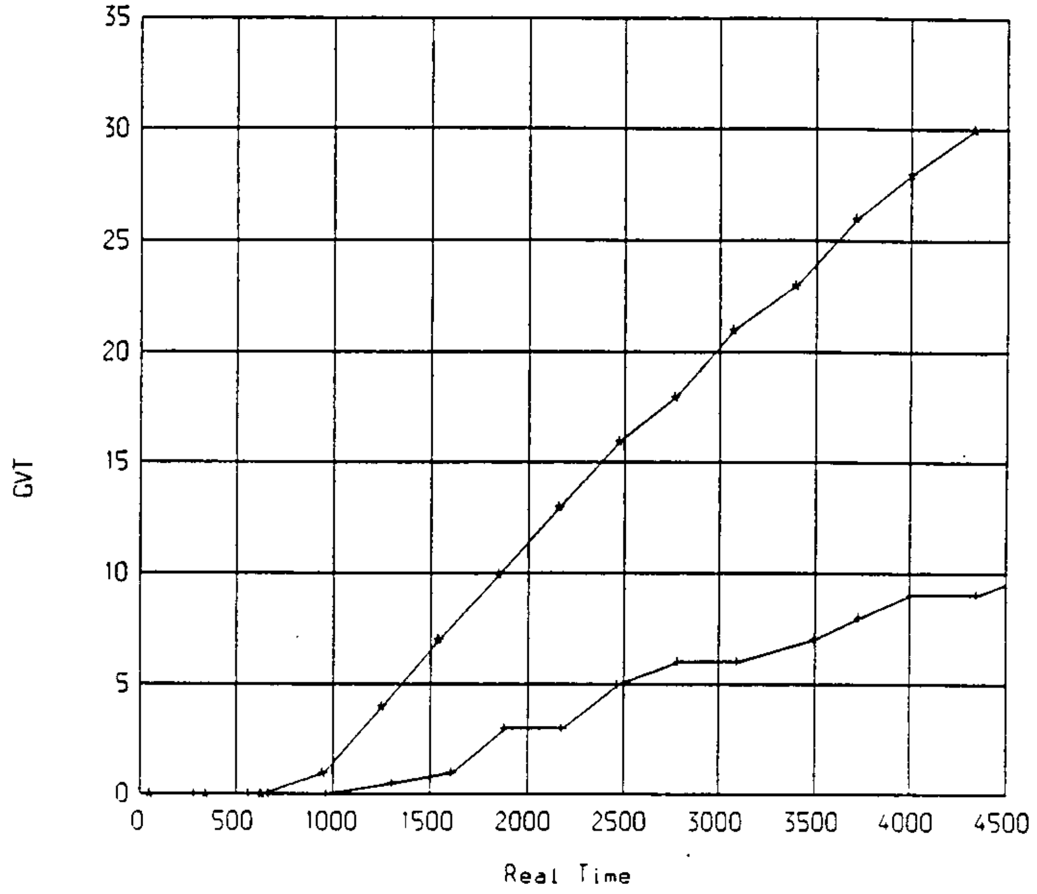

Figure 6-6. The GVT Advance for 3 Processors with and without Algorithm H1.

6.2. Fast Version

As described before, the runs in this section involve the same application of Game of Life, except that the code was run in the faster compiled mode and a bottleneck was removed from the system by removing the Life Board object. The difference is that the loads of the objects on the processors are considerably less. In Figure 6-7, the results of simulation runs for a fast Game of Life of size 12x12 is shown. The curves are the plots for GVT of 1 to 5 processor simulation. Unlike the slow version, the speedup grows with the number of processors. The 5-processor simulation does not show the same degradation that appeared in the previous case. Note that the bottleneck is removed from the model and hence message overflow does not create a problem. Nevertheless, the increase in performance is not at the rate of increase of number of processors. In fact, the relative speedup of 3-processor simulation is highest. It happens that with this particular board size, the objects can be divided more efficiently (less interaction) between processors.

Another observation is the difference in the speedups of various multi-processor simulations in the two slow and fast versions. The speedup in the slow version is considerably higher than the speedup in the fast version for the same number of processor. This is resulted from the heavy load of the simulation on the processor in single processor simulation. In the slow version, the one processor can hardly tolerate the load. Clearly, this results in a high speedup when several processors are utilized. In the fast version, the single processor is capable to run the whole system. Nevertheless, in this case a positive speedup can be obtained. Table 6-3 gives the statistics about this run.

The simulation time of the processors in the 3-processor simulation of the above case (Life 12x12) is given in Figure 6-8. The operation of the processors seem to be quite smooth with no (possibly very few) rollbacks. The GVT curve is relatively farther away from the minimum simulation time curve. This is due to infrequent computation of GVT. Note that GVT computation frequency is in the elapsed real time units. In this case, the simulation progresses much faster than previous version and therefore, the GVT computation is not run as frequently (in simulation time). The frequency of GVT runs is increased for the next test case that also happens to be a fast run.

Table 6-3. Statistics of the Fast Version of Life 12x12.

	2	3	4	5
Interaction Frequency	0.1	0.05	0.19	0.27
Fraction of Time in Event Processing	0.46	0.44	0.37	0.39
Fraction of Time in Rollback Processing	0.001	0	≈0	0.001
Fraction of Time in GVT Processing	0.01	0.02	0.003	0.001
Fraction of Time Scavenging	0.002	0.006	0.003	0.001

A model similar to the above with a Life Board of size 16x16 was also run. The GVT curves appear in Figure 6-9 and the statistics are presented in Table 6-4. The speedup is clearly increasing with number of processors. However, the relative speedup of 4-processor simulation is superior in this case. The reason is similar to the previous case. The Life Board of this size can be more efficiently divided into four processors. In fact, the above two tests confirm the result of the analytic model that the lower the interaction probability the higher is the speedup.

In Figure 6-10, the simulation time advance of processors in the 3-processor simulation of this case is plotted. Frequent occurrences of rollback can be observed in this Figure. Comparing the two Figures 6-8 and 6-10, the difference in the behavior of 3-processor simulation in two different cases can be observed. This and the speedup results obtained indicate the usefulness of an object assignment strategy that does not necessarily use the maximum number of available processors.

Table 6-4. Statistics of the Fast Version of Life 16x16.

	2	3	4	5
Interaction Frequency	0.03	0.1	0.05	0.17
Fraction of Time in Event Processing	0.36	0.43	0.47	0.45
Fraction of Time in Rollback Processing	≈0	0.001	≈0	0.001
Fraction of Time in GVT Processing	0.04	0.02	0.03	0.03
Fraction of Time Scavenging	0.01	0.01	0.006	0.002

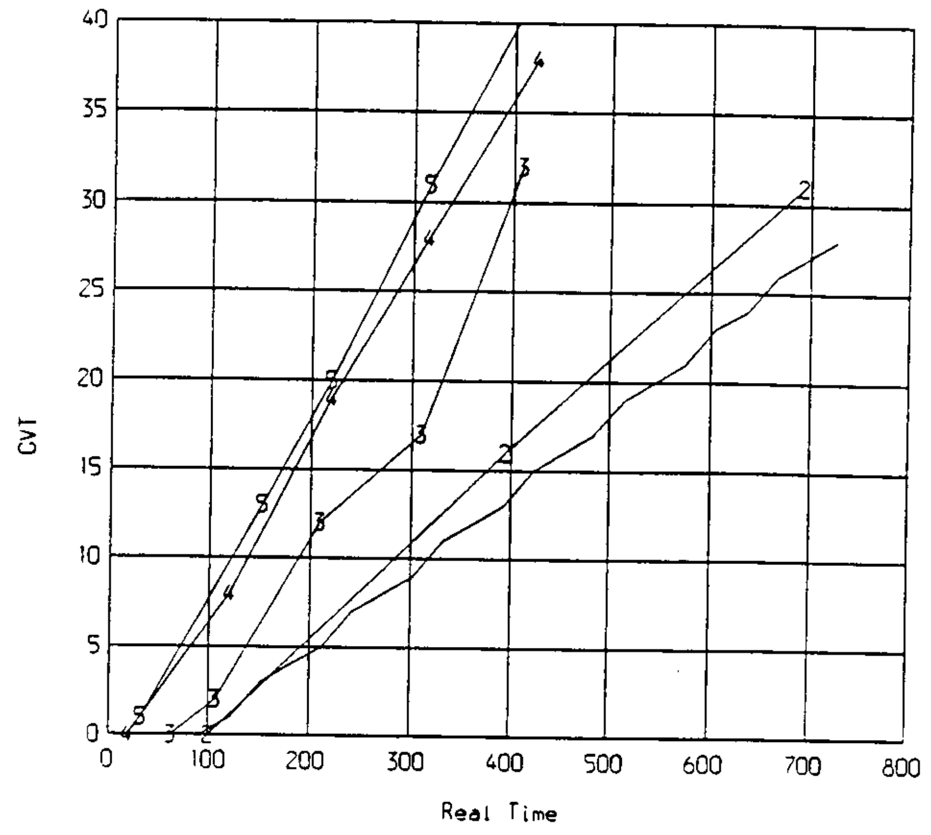

Figure 6-7. The GVT Advance for 1-5 Processors, Life 12x12 (Fast).

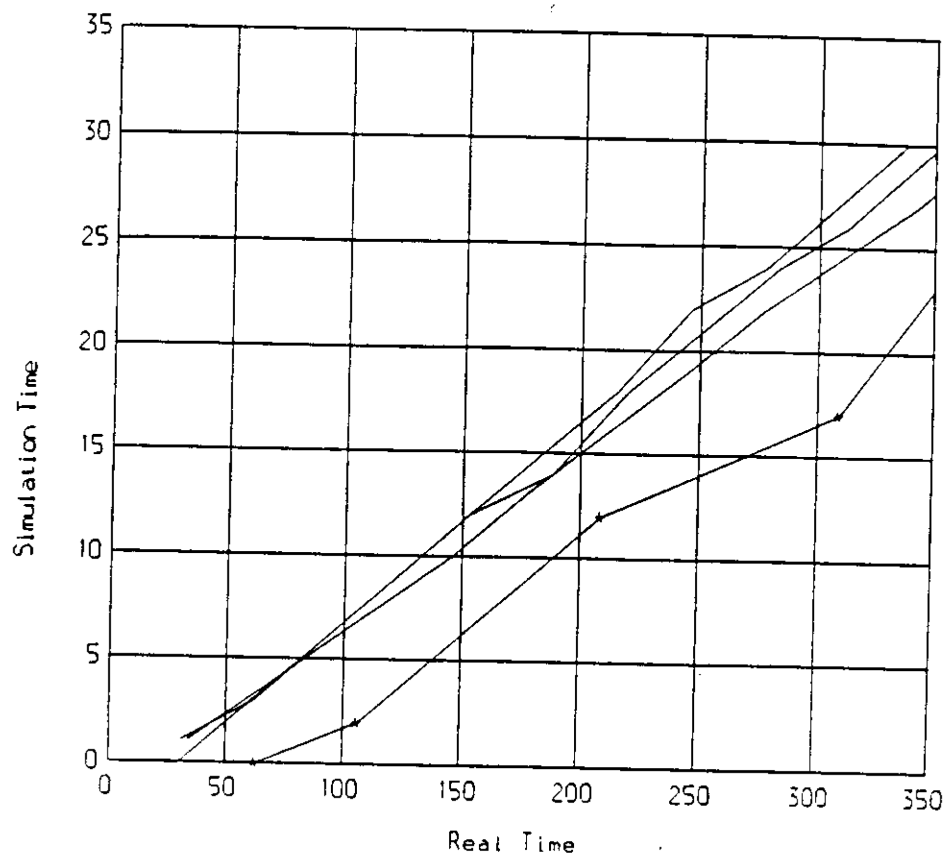

Figure 6-8. Processors' Simulation Time Advance in 3-processor Simulation, Life 12x12 (Fast).

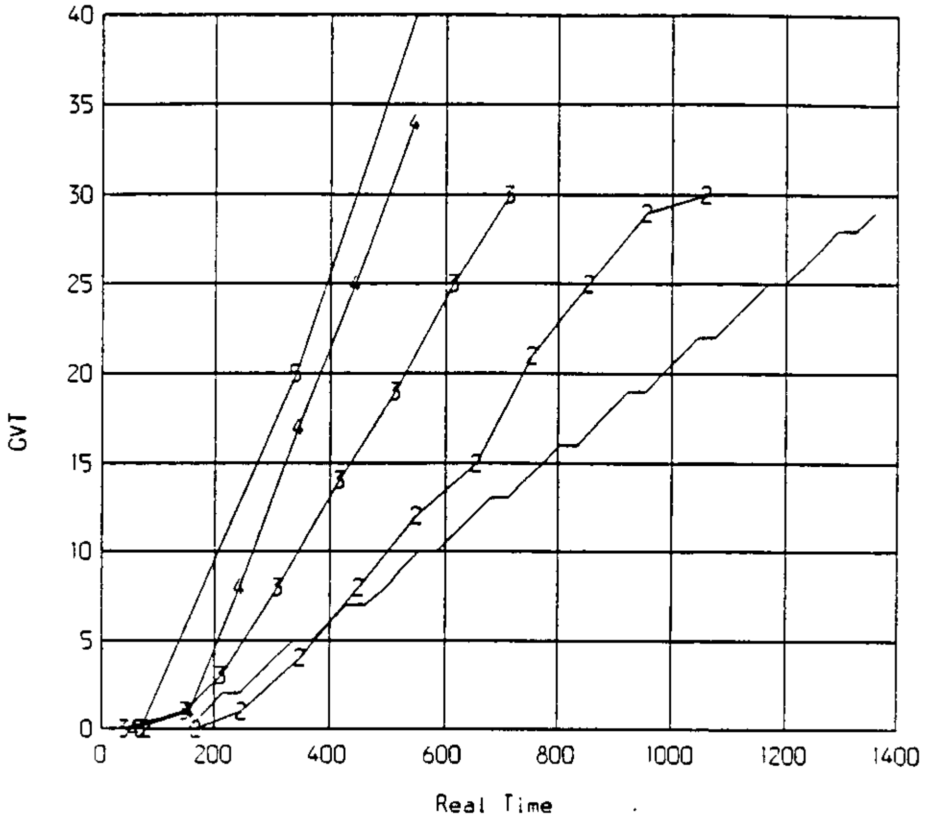

Figure 6-9. The GVT Advance for 1-5 Processors, Life 16x16 (Fast).

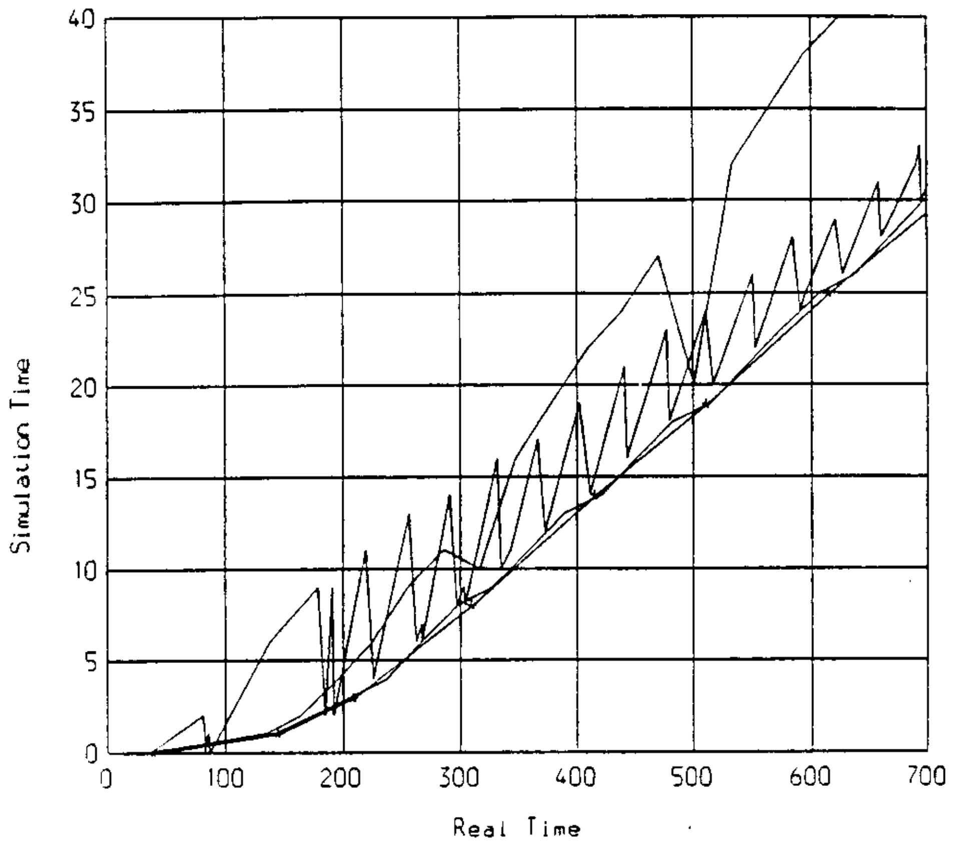

Figure 6-10. Processors' Simulation Time Advance in 3-processor Simulation, Life 16x16 (Fast).

CHAPTER 7

Conclusions

7.1. Summary of Contributions

The content of this dissertation is an extension to the existing work in the area of distributed simulation. The contributions are in the performance analysis of two distributed simulation methods and presenting algorithms for effective utilization of one of these methods. The general conclusion derived from mathematical analysis, simulation and implementation is that distributed simulation is workable. Furthermore, under certain conditions, it is superior to the conventional single-processor simulation. The conditions are not impossible or too restrictive. Apart from large simulation applications that can hardly be handled by single processors, smaller models can still benefit from distributed processing and produce results faster. The following is a summary of results obtained from this work.

7.1.1. Performance Analysis

The two most effective factors in the performance measure were shown to be the interaction probability and the communication delay in the simulator. The results show that the simulation model must be decomposable into weakly interacting submodels. High interaction rates increase the overhead and reduce

7.2. Suggestions for Future Research

Distributed simulation is a new and growing area in distributed processing. Some problems are common with other distributed systems and existing solutions are workable. Nevertheless, due to some of its particular aspects, more efficient solutions may be obtained by subject related research. Following are some topics that is suggested for future research.

(i) Performance Analysis - In the area of performance analysis the models need to be extended to remove some of its restrictions. Furthermore, models to test the effects of checkpointing frequency on the performance can produce a valuable lead in this subject.

(ii) Load Balancing - Dynamic load balancing techniques are required to complement the static strategies.

(iii) Fault Tolerance - Although in simulation fault tolerance may not be as vital as some other applications, the existence of a fault tolerant distributed simulation system can save considerable time and processing power. Note that because of the inherent property of the Rollback method, inclusion of fault tolerance may be simpler than other applications.

(iv) Object Relocation - Portability of objects or submodels dynamically is a prerequisite for the two previous suggested topics. Note that relocating an object does not only affect the "to" and "from" processors but can have a the speedup. Similarly, large values for the communication delay in the simulator reduces the speedup unless it is matched with a similar delay in the system to be simulated. A simulator with negligible communication delay between the processors (shared memory) was shown to produce the highest speedup.

The Rollback method was shown to be mathematically stable as long as the rollback processing factor is less than 1. i.e. the real time to perform rollback over some simulation time period (wasted simulation time), is strictly smaller than the real time used to simulate that simulation time period. As can be noted, this requirement is reasonable and not restrictive.

It was also shown that the prediction method is workable when the predictions correspond to distant future. A model that requires frequent predictions corresponds to one with high interaction rates between submodels.

The two methods for distributed simulation studied in this work were shown to produce a comparable speedup in the cases where the storage requirements of the Rollback method were not considered and the predictions in the Link Time method were assumed to be perfect.

7.1.2. Algorithms

A distributed algorithm was presented that correctly finds a lower bound on the simulation time of the slowest processor. It was furthermore shown by

implementation that the algorithm requires little processing time and produces fairly tight bounds. This algorithm can be generalized to any distributed system to obtain global information about the system such as termination.

Load balancing was also shown to be an important part of a distributed simulation method. The algorithms presented provide a reasonable static load balancing strategy. The effectiveness of one of the algorithms was shown by implementation results.

7.1.3. Implementation

The implementation results did in general agree with the previous analytic and simulation results, although the models were different. The effects of high interaction probability and communication delay were seen to be significant in the speedup. Furthermore, some overhead of the Rollback method such as rollback processing and GVT computation time were shown to be small.

These results provide us with some ideas for future research. Load balancing and flow control were shown to be necessary for the distributed system to be running efficiently. In addition, frequency of checkpointing that can directly affect storage requirements need to be studied further.

7.2. Suggestions for Future Research

Distributed simulation is a new and growing area in distributed processing. Some problems are common with other distributed systems and existing solutions are workable. Nevertheless, due to some of its particular aspects, more efficient solutions may be obtained by subject related research. Following are some topics that is suggested for future research.

(i) Performance Analysis - In the area of performance analysis the models need to be extended to remove some of its restrictions. Furthermore, models to test the effects of checkpointing frequency on the performance can produce a valuable lead in this subject.

(ii) Load Balancing - Dynamic load balancing techniques are required to complement the static strategies.

(iii) Fault Tolerance - Although in simulation fault tolerance may not be as vital as some other applications, the existence of a fault tolerant distributed simulation system can save considerable time and processing power. Note that because of the inherent property of the Rollback method, inclusion of fault tolerance may be simpler than other applications.

(iv) Object Relocation - Portability of objects or submodels dynamically is a prerequisite for the two previous suggested topics. Note that relocating an object does not only affect the "to" and "from" processors but can have a global effect.

(v) Combination of the two - The two methods for distributed simulation studied here have their own merits. Designing a system that combines both ideas can be a viable alternative. For example, the idea of the Link Time can be used to control the flow of messages from the faster to the slower processor in the Rollback method.

(vi) Extension - Extending the Time Warp method to other distributed system can produce an alternative in the way systems are designed. Some work is already underway in the area of distributed databases.

REFERENCES

[AHU 75] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1975.

[ATN 78] Athreya, K.B. and Ney, P., "A New Approach to the Limit Theory of Recurrent Markov Chains", Trans. of the American Mathematical Society, Vol. 245, Nov. 1978, pp. 493-501.

[BEG 80] Bernstein, P.A. and Goodman, N., "Fundamental Algorithms for Concurrency Control in Distributed Database Syatems", Computer Corp. of America, Tech. Report CCA-80-05, Feb. 1980.

[BRY 77] Bryant, R.E., "Simulation of Packet Communication Architecture Computer Systems", MIT LCS Technical Report TR-188, Nov. 1977.

[BRY 79] Bryant, R.E., "Simulation on Distributed system", Proc. First International Conference on Distributed Systems, Huntsville, Alabama, Oct. 1979, pp. 544-552.

[CEG 82] Christopher,T., Evans, M., Gargega, R.R. and Leonhardt, T., "Structure of a Distributed Simulation System", The 3rd. International Conference on Distributed Computing Systems, Miami, Oct. 1982.

[CHL 80] Chu, W.W., Holloway, L.J., Lan, M. and Efe, K. "Task Allocation in Distributed Data Processing", IEEE. Trans. on Computers, Nov. 80, pp. 57-69.

[CHM 78] Chandy, K.M. and Misra, J., "A Non-trivial Example of Concurrent Processing : Distributed Simulation", Dept. of Computer Science, University of Texas at Austin, TR-82, Sept. 1978.

[CHM 79a] Chandy, K.M., Holmes, V. and Misra, J. "Distributed Simulation of Networks", Computer Networks, 3 (1979), pp. 105-113.

[CHM 79b] Chandy, K.M. and Misra, J., "Distributed Simulation : A Case Study in Design and Verification of Distributed Programs", IEEE Tran. on Software Eng., Vol. SE-5, No. 5, Sept. 1979, pp. 440-452.

[CHM 81] Chandy, K.M. and Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", Comm. of the ACM, Vol. 24, No. 11, April 1981, pp. 198-206.

[CML 83] Chandy, K.M. and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems",???.

[COO 81] Cooper, M.W., "A survey of Methods for Pure Nonlinear Integer Programming", Management Science, Vol. 27, No. 3, March 81.

[DAL 77] Dalal, Y.K., "Broadcast Protocols in Packet switched Computer Networks", PhD. thesis, Stanford University, April 1977.

[DAV 69] David, H.A., "Order Statistics", 2nd. Edition, Wiley and Sons, 1962.

[DIS 80] Dijkstra,E.W. and Scholten, C.S., "Termination Detection for Diffusing Computations", Information Processing Letters, August 1980, pp. 1-4.

[DIN 70] Dinic, E. A., "Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation", Soviet Math. Dokl., Vol 11, 1970, pp. 1277-1280.

[DOO 53] Doob, J.L., "Stochastic Processes", John Wiley and Sons, 1953.

[EDK 72] Edmunds, J. and Karp, R. M., "Theoretical Improvements in Arithmetic Efficiency for Network Flow Problems," J.ACM, Vol. 19, 1972, pp. 248-264.

[EVE 79] Even, S., "Graph Algorithms", Computer Science Press, 1979.

[FOF 62] Ford, L.R., and Fulkerson, D.R., "Flows in Networks", Princeton University Press, 1962.

[FRA 80] Frances, N. "Distributed Termination", ACM-TOPLAS 2,1,1980.

[GAJ 79] Garey, M.R. and Johson, D.S., "Computers and Intractability", W.H. Freeman and Co., San Fransisco, 1979.

[GAR 70] Gardner, M., "Mathematical Games", Scientific American, Oct. 1970, pp. 120-123.

[GYE 76] Gylys, V.B. and Edwards, J.A., "Optimal Partitioning of Workload for Distributed Systems", Digest of Papers, COMPCON 76, Sept. 76, pp. 353-357.

[GOR 69] Gordon, G., "System Simulation", Prentice Hall, 1969.

[HAN 79] Hansen, P., "Methods of Nonlinear 0-1 Programming", Annals of Disc. Math., 5,1979, pp. 53-70.

[HAR 52] Harris, T.E., "The Existence of Stationary Measures for Certain Markov Processes",Proc. Third Berkeley Symposium on Mathematical Statistics and Probability, 1954-1955, Vol II, University of Calif Press, Berkeley and Los Angeles, pp. 113-124.

[HEM 82] Heyman, D.P. and Matthew, J.S., "Stochastic Models in Operations Research", Vol. 1, Mc Graw-Hill, 1982.

[JES 82] Jefferson, D. and Sowizral, H., "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control", Rand Report, N-1906-AF, Dec 1982.

[JEF 84] Jefferson, D. et al., "Implementation of Time Warp on the Caltech Hypercube", SCS Conference on Distributed Simulation, Jan. 1985.

[JES 83] Jefferson, D. and Sowizral, H., "Fast Concurrent Simulation Using the Time Warp Mechanism, Part II: Global Control", Rand Report, Aug 1983.

[KAM 76] Kannan, R. and Monma, C., "On the Computational Complexity of Integer Programming Problems", Institute fur Operations Research, Universitat Bonn.

[KAK 82] Karmarker, N. and Karp, M., "The Differencing Method of Set Partitioning",Report No. UCB/CSD 82/113, Computer Science Division (EECS), University of California, Berkeley, Dec. 1982.

[KLE 75] Kleinrock, L., "Queueing Theory", Vol. 1, Wiley and Sons, 1975.

[KRU 56] Kruscal, J.B., "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", Proc. Amer. Math. Soc. 7:1, 1956, pp. 48-50.

[LMM 81] Lavenberg, S.S., McNair, E.A., Mrkowitz, H.M., Sauer, C.H., Shedler, G.S. and Welch, P.D., "Computer Performance Modeling Handbook", Academic Press, 1981.

[MIC 82] Misra, J. and Chandy, K.M., "Termination Detection of Diffusing Computations in Communicating Sequential Processes", ACM Tran. on Programming Languages and Systems, 4,1, Jan. 1982, pp. 37-43.

[MLT 82] Ma, P.R., Lee, E.Y and Tsuchiya, M., "A Task Allocation Model for Distributed Computing Systems", IEEE. Trans. on Computers, Vol. C-31, No. 1, Jan. 82, pp. 41-47.

[ORE 71] Orey, S., "Lecture Notes on Limit Theorems for Markov Chain Transition Probabilities", Van Nostrand, 1971.

[PEA 80] Peacock, J.K., "Distributed Simulation Using a Network of Processors", Dept. of Computer Science, University of Waterloo, CCNG T-Report T-87, Jan. 1980.

[PIR 73a] Picard, J.C. and Ratliff, H.D., "Minimal Cost Cut Equivalent Networks", Management Science, Vol. 19, No. 9, May 1973, pp. 1087-1092.

[PIR 73b] Picard, J.C. and Ratliff, H.D., "A Graph Theoretic Equivalence for Integer Programs", Operations Research, Vol. 21, No. 1, 1973, pp. 261-269.

[PIR 74] Picard, J.C. and Ratliff, H.D., "Minimum Cuts and Related Problems", Networks, 5, 1974, pp. 357-370.

[PWM 79a] Peacock, J.K., Wong, J.W. and Manning, E.G., "Distributed Simulation Using a Network of Processors", Computer Networks, Vol. 3, No. 1, Feb. 1979, pp 44-56.

[PWM 79b] Peacock, K.J., Wong, J.W. and Manning, E., "Synchronization of Distributed Simulation Using Broadcast Algorithms", Proc. Fourth Berkeley Conference on Distributed Data Management and Computer Networks, San Fransisco, Aug. 1979.

[PWM 79c] Peacock, K.J., Wong, J.W. and Manning, E., "Distributed Approach to Queuing Network Simulation", Proc. Winter Simulation Conference, San Diego, Dec. 79.

[REV 75] Revuz, D. "Markov Chains", North Holland Publishing, 1975.

[RLT 78] Randell, B., Lee, P.A. and Treleaven, P.C., "Reliability Issues in Computing system Design", Computing Surveys, Vol. 10, No. 2, June 78.

[ROS 70] Ross, S.M., "Applied Probability Models with Optimization Applications", Holden-Day, San Fransisco, 1970.

[RSH 79] Rao, G.S., Stone, H.S. and Hu, T.C., "Assignment of Tasks in a Distributed Processor System with Limited Memory", IEEE. Tran. on Computers, Vol. C-28, No. 4, April 79.

[SEE 79] Seethalakshmi, M., "Performance Analysis of Distributed Simulation", M.S. Report, 1979, Dept. of Computer Science, University of Texas at Austin.

[STO 77] Stone, H.S.,"Multiprocessor Scheduling with the Aid of Network Flow Algorithms",IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, Jan. 1977, pp. 85-93.

[TWE 74] Tweedie, R.L., "Sufficient Conditions for Ergodicity and Recurrence of Markov Chains on a General State Space", Stoch. Processes and Their Applications, 3, 1975, pp. 385-403.

[WAL 80] Wall, D.W., "Mechanisms for Broadcast and Selective Broadcast", PhD. thesis, Stanford University, Tech. Report No. 190, June 1980.

APPENDIX A

Recall that in the analysis of chapter 4, a cycle of operation was defined as the interval between the completion times of two consecutive rollback operations.

That is

$$Y_n = t_n^e - t_{n-1}^e$$

$$= \textit{the length of the n-th cycle (real time)}$$

where

$$t_n^e = \textit{the real time at which the n-th rollback operation ends}$$

We further defined the following variables

$$X_n = \textit{useful simulation time advance during the n-th cycle}$$

$$X(t) = \textit{useful simulation time at time } t$$

$$|\Delta_n| = \textit{the difference in simulation time of the two processors at } t_n^e$$

$$t_n^b = \textit{real time at which the n-th rollback begins}$$

and the speedup was defined as

$$\rho \propto \lim_{t \to \infty} \frac{X(t)}{t}$$

where the constant of proportionality is the simulation rate, $\frac{\mu}{\lambda}$.

In what follows, a proof is presented that

$$\lim_{t \to \infty} \frac{X(t)}{t} = \frac{E[X_1]}{E[Y_1]} \tag{A1}$$

when the two expected values exist and are finite.

Note that the above is not obvious, since $\{X_n, n \geq 1\}$ are dependent random variables. The same applies to $\{Y_n, n \geq 1\}$. Both variables depend on the simulation time difference of the processors at t_{n-1}^c,i.e. $|\Delta_{n-1}|$, which in turn depends on Y_{n-1}.

PROOF

The proof consists of two parts. First, we need to prove that the processes $\{X_n\}$ and $\{Y_n\}$ are ergodic processes. Next, the ergodic theorem for stationary processes is used to prove the equality (A1).

Part 1

According to chapter 4,

$$Y_n = Y_{1n} + Y_{2n}$$

where Y_{1n} is distributed exponentially with mean $\frac{1}{\mu p}$ and Y_{2n}, the real time to perform rollback, is a function of $|\Delta_{n-1}|$. Hence, Y_n is a function of $|\Delta_{n-1}|$. Similarly, X_n, the useful simulation time during Y_n is a function of $|\Delta_{n-1}|$. Therefore, the proof for ergodicity of processes X_n and Y_n reduces to the proof for ergodicity of $|\Delta_n|$.

$\{|\Delta_n|, n \geq 1\}$ is a homogeneous markov process defined at points t_n^c and the random variables Δ_n are in $\mathbf{R}^+$ (R^+). Note that we only consider the absolute value of the time difference. Hence, the conditional value δ is always non-negative. Let

$$p(\delta,A) = p^1(\delta,A) = Pr\{|\Delta_n| \in A \,|\, |\Delta_{n-1}| = \delta\}$$

be the transition probability of the process.

To show that $|\Delta_n|$ is ergodic, we need to prove that

(i) $|\Delta_n|$ is φ-irreducible [ATN 78], [DOO 53], [HAR 52], [ORE 71], [REV 75]. i.e.there is a σ-finite measure φ such that for any A with $\varphi(A)>0$,

$$p^n(\delta,A) = Pr\{|\Delta_n| \in A \,|\, |\Delta_0|=\delta\} > 0 \quad \textit{for any } \delta \textit{ in } \mathbf{R}^+$$

(ii) $|\Delta_n|$ satisfies the following theorem by Tweedie [TWE 75].

Theorem Let $\{X_n\}$ be a φ-irreducible Markov chain on a topological space (X,F) If {p(x,.)} is strongly continuous, a sufficient condition for X_n to be ergodic is the existence of a K $\in$ $\boldsymbol{X}$ and a non-negative measurable function g on $\boldsymbol{X}$ such that

$$(a) \quad \int_X p(x,dy)g(y) \leq g(x)-1, \quad x \textit{ notm } K$$

$$(b) \quad \int_X p(x,dy)g(y) = \lambda(x) \leq B < \infty \quad \textit{for } x \in K \textit{ and fixed } B$$

A transition probability p(x,.) is strongly continuous if for every A $\in$ $\boldsymbol{F}$, p(x,A) is a continuous function in x.

(i) Irreducibility

The transition probability in this case is

$$p(\delta,A) = Pr\{\sum_{i=0}^{n} u_i \in A\}$$

where u_i is iid. exponential with rate λ and n is the number of poisson arrivals in an interval of length $r\lambda/\mu|C_k^* + \delta|$ and C_k^* has distribution $\frac{1}{\lambda\sqrt{p/2}} e^{-\lambda\sqrt{p/2}|t|}$.

The irreducibility condition, therefore holds.

(ii) Positive recurrence

To prove the positive recurrence, we need to prove that the conditions of the above theorem holds. The strong continuity of the transition probability follows from its definition. We therefore need to prove that a set K and a measurable function g(.) exists, such that the conditions are true.

Let g(y)=y, then

$$\int_{R^+} p(\delta,dy)y = E[|\Delta_n| \; |\Delta_{n-1}|=\delta]$$

$$= r[\delta + \frac{e^{-\lambda\sqrt{p/2}\delta}}{\lambda\sqrt{p/2}}]$$

where the last equality follows from (16) of chapter 4.

$$\leq r\delta + \frac{r}{\lambda\sqrt{p/2}}$$

Now, the condition (a) of the theorem holds if

$$r\delta + \frac{r}{\lambda\sqrt{p/2}} \leq \delta - 1$$

i.e. if

$$\delta \geq \frac{r+\lambda\sqrt{p/2}}{\lambda\sqrt{p/2}(1-r)} \quad \textit{for } r<1$$

Taking $K = \{\delta : \delta \leq \frac{r}{\lambda\sqrt{p/2}(1-r)}\}$, condition (a) of theorem holds.

Also

$$E[|\Delta_n|] \leq r(\delta + \frac{1}{\lambda\sqrt{p/2}})$$
$$\leq \frac{r(1+\lambda\sqrt{p/2})}{(1-r)\lambda\sqrt{p/2}} \quad \textit{for } \delta \in K$$

Letting B to be the RHS of the above inequality, the condition of (b) also holds. This completes the proof that the $|\Delta_n|$-process is ergodic and hence $|\Delta_n| \rightarrow |\Delta|$ in distribution as $n \rightarrow \infty$. Furthermore, the distribution is finite. Consequently, the Y_n and X_n processes are also ergodic.

Part 2

Given that Y_n and X_n are ergodic, it is shown that

$$\lim_{t \rightarrow \infty} \frac{X(t)}{t} = \frac{E[X_1]}{E[Y_1]} .$$

We have

$$X(t_{n(t)}) \leq X(t) \leq X(t_{n(t)+1})$$

where

$$n(t) = sup\{n : t_n{}^{c} \leq t\}$$

then

$$\frac{X(t_{n(t)})}{t_{n(t)}}\frac{t_{n(t)}}{t_{n(t)+1}} \le \frac{X(t)}{t} \le \frac{X(t_{n(t)+1})}{t_{n(t)}}$$

Letting $t \rightarrow \infty$,

$$\lim_{t\rightarrow\infty} \frac{X(t)}{t} = \lim_{n\rightarrow\infty} \frac{X(t_n^c)}{t_n^c}$$

$$= \frac{\sum_{i=1}^{n} X_i}{\sum_{i=1}^{n} Y_i}$$

$$= \frac{1/n \sum_{i=1}^{n} X_i}{1/n \sum_{i=1}^{n} Y_i}$$

Using the ergodic theorem for stationary processes,

$$\lim_{n\rightarrow\infty} RHS = \frac{E[X]}{E[Y]}$$

and the proof is complete.

APPENDIX B

1. Convergence

This is a proof for the convergence of the infinite sum in chapter 5, equation (13). We have

$$\Delta = \frac{p}{2\mu}\sum_{m=0}^{\infty}\sum_{n=1}^{\infty}\frac{(m+n)!}{m!n!}\left(\frac{1-p}{2}\right)^{m+n}\sum_{i=0}^{m}\sum_{j=0}^{n}\frac{(i+j)!}{i!j!}(\frac{1}{2})^{i+j}$$

and

$$E[I_1] = \frac{1}{\mu p} - \Delta$$

where I is the length of the interval. Let

$$The\ sum\ = \sum_{m=0}^{\infty}\sum_{n=1}^{\infty}\frac{(m+n)!}{m!n!}\left(\frac{1-p}{2}\right)^{m+n}\sum_{i=0}^{m}\sum_{j=0}^{n-1}\frac{(i+j)!}{i!j!}\left(\frac{1}{2}\right)^{i+j}$$

Consider the two inner summations, and let

$$k=i+j$$

Then

$$\sum_{i=0}^{m}\sum_{j=0}^{n-1}\frac{(i+j)!}{i!j!}\left(\frac{1}{2}\right)^{i+j} \leq \sum_{k=0}^{m+n-1}\sum_{j=0}^{k}\frac{k!}{(k-j)!j!}\left(\frac{1}{2}\right)^{k} \cdot\ \textit{with equality if } m=n=\infty$$

$$= m+n$$

Therefore,

$$The\ sum \leq \sum_{m=0}^{\infty}\sum_{n=1}^{\infty}(m+n)\ \frac{(m+n)!}{m!n!}\left(\frac{1-p}{2}\right)^{m+n}$$

Similarly, let

$$k=m+n \ ,$$

then

$$\begin{aligned}
\textit{The sum} &\leq \sum_{k=1}^{\infty} \sum_{n=1}^{k} k \frac{k!}{(k-n)!n!} \left(\frac{1-p}{2}\right)^{k} \\
&= -2 \sum_{k=1}^{\infty} \left(\frac{1-p}{2}\right) \frac{d}{dp} \left(\sum_{n=1}^{k} \frac{k!}{(k-n)!n!} \left(\frac{1-p}{2}\right)^{k}\right) \\
&= -\left(1-p\right) \sum_{k=1}^{\infty} \frac{d}{dp} \left((1-p)^{k}-1\right) \\
&= -\left(1-p\right) - \frac{d}{dp} \sum_{k=1}^{\infty} (1-p)^{k} \\
&= -\left(1-p\right) \frac{d}{dp} [\frac{1}{p}-1] \\
&= \left(\frac{1-p}{p^2}\right)
\end{aligned}$$

The RHS is finite as long as $p \neq 0$, so "The sum" is converging for non-zero values of p, the interaction probability. Hence

$$\Delta \leq \frac{(1-p)}{2p\mu}$$

and

$$\begin{aligned}
E[I_1] &= \frac{1}{p\mu} - \Delta \\
&\geq \frac{2-p}{p\mu} \qquad (d=0)
\end{aligned}$$

As p -> 0 , $E[I_1] \rightarrow \infty$ as is expected. Also $E[I_1] = \frac{1}{\mu}$ when p=1. i.e. I_1 is the real time to simulate one event when p=1.

2. Upper Bound on the Error

The following is the computation for an upper bound on the error resulted from computing a finite sum for Δ .

Let M and N be the upper limits in the computation of Δ for the two infinite sums. Then

$$\sum_{m=0}^{\infty} \sum_{n=1}^{\infty} = \sum_{m=0}^{M} \sum_{n=1}^{N} + \sum_{m=M+1}^{\infty} \sum_{n=1}^{N} + \sum_{m=0}^{\infty} \sum_{n=N+1}^{\infty}$$

The error is the sum of the last two terms, ie.

$$\begin{aligned} error &= \sum_{m=M+1}^{\infty} \sum_{n=1}^{N} + \sum_{m=0}^{\infty} \sum_{n=N+1}^{\infty} \\ &\leq \sum_{m=M+1}^{\infty} \sum_{n=0}^{\infty} + \sum_{m=0}^{\infty} \sum_{n=N+1}^{\infty} \end{aligned}$$

Taking M and N to be equal, then

$$error \leq 2 \sum_{m=M+1}^{\infty} \sum_{n=0}^{\infty}$$

Therefore, the error introduced in Δ is

$$\begin{aligned} &\leq \frac{p}{\mu} \sum_{m=M+1}^{\infty} \sum_{n=0}^{\infty} \frac{(m+n)!}{m!n!} \left(\frac{1-p}{2}\right)^{m+n} \sum_{i=0}^{m} \sum_{j=0}^{n} \frac{(i+j)!}{i!j!} \left(\frac{1}{2}\right)^{i+j} \\ &\leq \frac{p}{\mu} \sum_{m=M+1}^{\infty} \sum_{n=0}^{\infty} \frac{(m+n)(m+n)!}{m!n!} \left(\frac{1-p}{2}\right)^{m+n} \\ &\leq \frac{p}{\mu} \sum_{k=M+1}^{\infty} \sum_{n=0}^{k} \frac{kk!}{(k-n)!n!} \left(\frac{1-p}{2}\right)^{k} \\ &= \frac{p}{\mu} \left(\frac{1-p}{2}\right) \frac{d}{dp} \left[-2 \sum_{k=M+1}^{\infty} (1-p)^k \right] \end{aligned}$$

$$= \frac{p}{\mu}\left(1-p\right)\frac{d}{dp}\left[-(\frac{1}{p}-\frac{1-(1-p)^{M+1}}{p})\right]$$

$$= -\frac{p}{\mu}\left(1-p\right)\frac{d}{dp}\left[\frac{(1-p)^{M+1}}{p}\right]$$

$$= \frac{1}{\mu}\left(1-p\right)\frac{(M+1)p(1-p)^{M}+(1-p)^{M+1}}{p}$$

Now, depending on p, we need to select an M such that the term on the RHS is smaller than the accepted error.