DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/88D-14

Distributed Discrete-Event Simulation Using Variants of the Chandy-Misra Algorithm on the Intel Hypercube

THESIS

David Louis Mannix
Captain, USAF

AFIT/GCS/ENG/88D-14

AFIT/GCS/ENG/88D-14

DISTRIBUTED DISCRETE-EVENT SIMULATION USING VARIANTS OF THE CHANDY-MISRA ALGORITHM ON THE INTEL HYPERCUBE

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Information Systems

David Louis Mannix, B.S.
Captain, USAF

December, 1988

Acknowledgments

I'd like to express sincere thanks to my thesis advisor, Captain Nathaniel J. Davis, USA for his guidance, understanding, and insightful commentary, and to the other members of my thesis committee, Dr. Thomas C. Hartrum and Captain Wade H. Shaw, USA, for their expert advice and support. I'd also like to thank Captain Brian J. Donlan, USAF for his help and advice. Thanks are also in order for Messrs. Bruce Clay and Rick Norris for their superb operational support. Most of all, though, I'd like to thank my wife, [redacted] for being my support system while at the same time enduring the separation of a "grad school widow" without complaint. I can only hope that she knows the extent of my gratitude for her encouragement, support, and confidence, which could never be fully expressed in words.

David Louis Mannix

Table of Contents

Page

Acknowledgments ii

Table of Contents iii

List of Figures vi

List of Tables vii

Abstract viii

I. Introduction 1-1

1.1 Simulation Overview 1-2

1.2 Distributed Processing Overview 1-4

1.3 Problem Statement 1-6

1.4 Scope 1-8

II. Literature Review 2-1

2.1 Parallelism in the Simulation Process 2-1

2.1.1 Application Level Parallelism 2-1

2.1.2 Support Function Distribution 2-3

2.1.3 Model Function Distribution 2-4

2.2 Distributed Simulation Algorithms 2-5

2.2.1 Chandy-Misra Null Message Algorithm 2-6

2.2.2 Bryant's Infinite Buffers Algorithm 2-8

2.2.3 Peacock et al - Link Time/Blocking Table Algorithms 2-12

Page

2.2.4 Chandy-Misra Deadlock Detection and Recovery Algorithm 2-13

2.2.5 SRADS Algorithm 2-17

2.2.6 Optimistic Algorithms - Time Warp 2-18

2.3 Performance Studies of Distributed Simulation 2-21

2.3.1 Performance Measurement 2-22

2.3.2 Factors Affecting Performance 2-23

2.3.3 Chronology of Performance Studies 2 25

2.4 Summary 2-29

III. Description of Algorithms 3-1

3.1 The Physical System 3-2

3.1.1 Definitions 3-3

3.1.2 The Physical Process 3-4

3.1.3 Event-Oriented View of the Physical Process . 3-6

3.1.4 Message-Passing 3-10

3.1.5 Predictability 3-11

3.2 The Logical System 3-12

3.2.1 The Logical Process 3-13

3.2.2 Logical Process I/O 3-20

3.2.3 Properties of LP Communication 3-23

3.2.4 Summary of Algorithm 3-24

3.2.5 Deadlock in the Logical System 3-25

3.2.6 Null Message Variants 3-35

3.2.7 Bounds on Required Memory 3-39

IV. Performance Analysis 4-1

4.1 Performance Characteristics of the Event List 4-1

Page

4.1.1 The Linear List Implementation 4-1

4.1.2 Theoretical Bounds on Speed-Up 4-2

4.2 Empirical Studies 4-7

4.2.1 Methodology 4-8

4.2.2 Empirical Results 4-13

4.3 Summary 4-33

V. Conclusions 5-1

5.1 Summary 5-1

5.1.1 Null Message Strategies 5-2

5.1.2 Assignment Heuristics 5-2

5.2 Assessment of the Distributed Event List Algorithm . . 5-3

5.2.1 Advantages of the Distributed Event List Algorithm 5-3

5.2.2 Disadvantages of the Distributed Event List Algorithm 5-4

5.3 Recommendations for Further Research 5-5

Bibliography BIB-1

Vita VITA-1

List of Figures

Figure | Page

3.1. Non-Cyclic System of PP's with Cyclic Mapping 3-19

3.2. Set of Logical Processes Subject to Cyclic Deadlock 3-27

3.3. Set of Logical Processes Subject to Acyclic Deadlock 3-30

4.1. Balanced Flow Queueing Sub-model 4-9

4.2. Network of Sub-models in Tandem Topology 4-14

4.3. Speed-Up for Tandem Topology 4-15

4.4. Network of Sub-models in Feed-Forward Topology 4-17

4.5. Speed-Up for Feed-Forward Topology 4-18

4.6. Tandem Logical System with Pseudo-Feedback 4-20

4.7. Speed-Up Effect of Spin Loop 4-22

4.8. Tandem Speed-Up, Time-out Nulls, 0 Spin Loop 4-25

4.9. Balanced Feed-Forward Speed-Up, Time-out Nulls, 0 Spin Loop . 4-26

4.10. Balanced Feed-Forward Speed-Up, Stimulus Nulls, 1000 Spin Loop 4-28

4.11. Unbalanced Feed-Forward Speed-Up, Stimulus Nulls, 0 Spin Loop 4-29

4.12. Tandem Topology with Single Loop 4-32

List of Tables

Table		Page
4.1.	Insertion into Linear List with Exponential Inter-Event Times . .	4-3
4.2.	Effect of Pseudo-Feedback Loops on Tandem Model	4-20
4.3.	Summary of Factors for Null Message Strategy	4-23
4.4.	Assignment Strategies for Feed-Forward Topology	4-31
4.5.	Effect of Assignment, Balanced Feed-Forward, 8 Processors . . .	4-31
4.6.	Effect of Assignment, Unbalanced Feed-Forward, 8 Processors . .	4-32
4.7.	Assignment Strategies for Tandem Topology w/ Single Loop . . .	4-32
4.8.	Effect of Assignment, Tandem with Single Feedback, 8 Processors	4-33

AFIT/GCS/ENG/88D-14

Abstract

The goal of distributed simulation is to speed up simulation by distributing a simulation model's execution over multiple processors. This thesis reviews existing methods for distributed simulation, and introduces an algorithm for distributed discrete-event simulation, the *distributed event list* algorithm, based on the Chandy-Misra algorithm, with an event list, similar to that used in sequential simulation, at each logical process. Null messages are used for deadlock avoidance. The algorithm is described, and is shown to require a bounded amount of memory at each logical process.

A performance analysis of the distributed event list algorithm is performed. In the analytical portion, a linear event list implementation is shown to be of super-linear time complexity in relation to events simulated. This time complexity implies theoretical speed-up of greater than N for a simulation distributed over N processors. This result contradicts a commonly-held view of the existence of a bound of N on attainable speed-up.

Empirical studies evaluate the performance of the distributed event list algorithm under a variety of conditions. Speed-up greater than N are shown to be achievable for certain topologies of simulation models, confirming the time complexity analysis. The topology of the simulation model is shown to greatly affect the attained speed-up. Simulation networks with directed cycles exhibit extremely poor performance, in agreement with previous performance studies of the Chandy-Misra algorithm.

Alternate strategies for sending the Null messages used for deadlock avoidance are compared. Results show that for tandem and feed-forward topologies, a certain level of Null messages are beneficial to speed-up.

The problem of assigning a given simulation model to a set of logical processes is addressed. It is seen that topology of the logical system plays a critical role in the effectiveness of an assignment strategy.

DISTRIBUTED DISCRETE-EVENT SIMULATION USING VARIANTS OF THE CHANDY-MISRA ALGORITHM ON THE INTEL HYPERCUBE

I. Introduction

Distributed discrete event simulation has been a topic of intense research interest since the late 1970's. The principal goal of distributed simulation is to reduce the time needed to perform a simulation by spreading its execution over multiple processors. This is accomplished by exploiting the parallelism inherent to discrete-event simulation.

The time required to execute large simulation models on even the fastest sequential computers has limited the use of simulation in several application domains, such as large-scale digital logic systems, simulation of weather patterns, and military applications such as strategic defense and conventional battle management. Distributed simulation is one possible way of extending the usefulness of simulation into these and other areas.

While algorithms for distributed discrete-event simulation have existed for a number of years, little had been accomplished until very recently in the task of evaluating and refining these algorithms, perhaps due to the non-availability of cost-effective distributed computing systems. Early performance studies of distributed simulation algorithms were accomplished by using uniprocessor computers to simulate distributed computing systems [CM79].

The widespread availability of relatively cheap microprocessors has enabled the development of less costly multiprocessor computing systems for the research community and for general use, resulting in new opportunities and incentives for the development of distributed simulation methods [Hei86]. Recent research has focused on performance evaluation of the existing algorithms, based on empirical studies performed on various distributed architectures [Fuj88, RM88, RMM88].

1.1 Simulation Overview

Simulation is a mathematical tool that allows the user to modify and experiment with a system when it would otherwise be impractical to do so. Simulation has been defined by Banks and Carson as "the imitation of the operation of a real-world process or system over time" [BC84] and by Shannon as "the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behavior of the system or of evaluating various strategies for the operation of the system" [Sha75].

In a defining a simulation model, the modeler seeks to capture the essential behavior of some physical system to a certain level of abstraction. The modeler implements the model by constructing a simulation computer program. In doing so, the model must be defined in terms of functions that are computable, necessitating the adoption of a particular paradigm or view of the physical systems to be simulated. The modeler can implement the model in a general-purpose programming language such as FORTRAN or Ada, or by using a *specialized simulation* language such as GASP [Pri74], SLAM II [Pri86], and many others. Simulation languages save a large amount of programming effort by providing pre-defined, standard functions for controlling a simulation, calculating statistics, etc., allowing the modeler to concentrate on the abstraction of the simulated system [LK82].

Discrete-event simulation refers to simulation of a system whose state changes only at a countable number of points in time. The majority of discrete-event simulation models are stochastic in nature, having one or more random variables as inputs. The outputs of such models are also random variables, and statistical methods must be used in their analysis [Sha75].

Most discrete-event simulation is characterized as being event-driven, where an event is an instantaneous occurrence which may change the state of a system [BC84]. The event-driven method of discrete-event simulation advances simulated time in irregular intervals defined by the time of occurrence of each simulated event. An alternate discrete-event simulation method, time-driven discrete-event simulation, advances simulated time in regular intervals and simulates each event when the time of its occurrence has been reached [LK82].

In a typical sequential implementation of the event-driven approach to simulation, an "event list" is maintained of events that have been scheduled to occur.

The simulation advances by simulating the imminent event in the event list; i.e., the event with the earliest simulation time associated with it. This event is removed from the event list and the simulation clock is advanced to the time of the event. Simulating an event may change the values of variables that describe the state of the system, and, in addition, may cause new events to be added to the event list [BC84]. Discrete-event simulations are generally designed to terminate when the simulation clock reaches a certain value, or when the number of occurrences of some event reaches a pre-defined limit [LK82].

1.2 Distributed Processing Overview

There is no consensus concerning the exact definition of a distributed processing system. Is the "distribution" necessarily geographic in nature, or does it simply refer to the logical distribution of processing? J.P. Verjus prefers a strict definition based on geographic distribution of processing resources, characterizing a distributed system as "a set of separate sites ... interconnected by communications channels" [PV83]. While some definitions classify distributed processing as a form of parallel processing, Hwang and Briggs maintain a distinction. They assert that "parallel processing and distributed processing are closely related," although as data communications technology advances, "the distinction between parallel and distributed processing becomes smaller and smaller" [HB84].

One distinctive feature of distributed systems seems to be that no shared memory is allowed [Sch82]. A distributed system has been defined by D. Herman as "a set of co-operating processes installed on a multiprocessor architecture without a common memory [PV83]." The preceding definition of a distributed system shall be used for the remainder of this paper, since it appears to describe the essential features of a distributed system without the arbitrary constraint of geographic separation.

A distributed program, then, is a collection of cooperating (or communicating) processes. Because one of the goals of a distributed system is to increase the throughput over that available in a non-distributed system, it is desirable to exploit the maximum amount of concurrency. This is accomplished by allowing the processes to run asynchronously, communicating only as required [Sch82]. Hence, synchronization of communicating processes is a major part of distributed computing algorithms.

C.A.R Hoare has introduced a notation and paradigm for the specification and operation of *communicating sequential processes* [Hoa78]. His paradigm is designed to allow proofs of correctness, to limit memory requirements, and to eliminate non-determinism to the maximum extent possible [Hoa85]. Many others, most notably Dijkstra, have developed specific algorithms relating to the correct operation of communicating processes [Lam78, DS80, Sch82].

In partitioning a task into a system of communicating sequential processes, several design criteria are relevant. Defining the processes in such a way as to balance the processing load will maximize throughput, assuming homogeneous processors and excluding interprocessor communications. On the other hand, interprocess communication carries with it an overhead cost. While we would expect linearly-increasing throughput with additional processors, communication overhead eventually leads to the *saturation effect*. This effect, similar to "thrashing" in early memory paging systems, can cause throughput to decrease with each additional processor applied to a problem [CH*80].

The throughput advantage of a parallel computation is often quantified by the *Speed-up factor*, which is the ratio of the time taken to execute a computation on a single processor to the time taken to execute the same computation in the parallel system [Fuj88]. The *efficiency* of a distributed implementation is described by the ratio of Speed-up to N, where N is the number of concurrent processes in the distributed system. If the same algorithm is used in both the single and multiple processor computations, perfect efficiency is considered to be 1.0, corresponding to a Speed-up factor of N [Hei86].

1.3 Problem Statement

In a simulation of sufficient size and complexity to merit the use of distributed computing techniques, it is likely that the system to be modelled will consist of a larger number of components, or physical processes, than there are processors available in the distributed computing system. Even when this is not the case, it is preferable in some instances to distribute a simulation over a subset of available processors, as in Heidelberger's method of concurrently performing simulation replications, achieving speed-up equal to the number of processors [Hei86].

In cases such as these, it is desirable to have an efficient algorithm for simulating multiple physical processes on a single processing node. A natural candidate algorithm is the *future events list* method used in sequential discrete-event simulation. The concept of combining a conservative distributed simulation algorithm with an event list at each logical process is not a new one. Bryant's algorithm for distributed simulation included such a fusion of distributed and sequential simulation concepts [Bry79]. Authors of subsequently published algorithms, however, generally have chosen either to extend the message-passing paradigm within each logical process, or not to address the issue.

In addition, many algorithms for distributed discrete-event simulation require the assumption of a particular "world-view" or paradigm of the physical system being simulated. However, we may not wish to modify our abstraction of the physical system in order to accommodate the implementation of our simulation model on a particular computing system. A desirable method for distributed simulation will allow distributed simulation with a physical system paradigm based on the widely-used *event-oriented* view, easing the problems associated with parallelizing a simulation model that has been developed for sequential simulation.

It is the goal of this thesis to present such a method for distributed discrete-event simulation, the *distributed event list* algorithm, and to describe the conditions under which the algorithm can be expected to efficiently provide significant speed-up of a discrete-event simulation.

An event list structure at each logical process ensures chronological execution of events. It will be shown that under certain conditions with a simulation distributed over N processors, the distributed event list algorithm can yield speed-up greater than N. Speed-up equal to the number of processors has often been asserted in the literature to be the maximum achievable speed-up for a distributed simulation [Hei86, RMM88].

The distributed event list algorithm uses a method of interprocess communication and synchronization based on the *Null message* algorithm for distributed simulation proposed by Chandy and Misra [CM79]. Several variants on the basic communications method are presented. The effects of the topology of physical process interconnection and the computational intensity of event processing on the resultant speed-up are explored.

1.4 Scope

This thesis includes a review of methods for performing distributed discrete-event simulation in Chapter 2, and develops a particular method, the distributed event list algorithm with several variants, based on the conservative synchronization protocol proposed by Chandy and Misra, in Chapter 3. An analytical performance analysis of this algorithm is presented and supported with empirical studies in Chapter 4. Chapter 5 summarizes the results of the analysis and studies to provide insight into the conditions under which the algorithm can be expected to provide significant Speed-up, including comparisons among the variants.

II. Literature Review

2.1 Parallelism in the Simulation Process

Kaudel identifies three kinds of parallelism in the simulation process, each of which can, at least theoretically, be exploited to speed up simulation. *Application level parallelism* consists of executing multiple simulation trials concurrently. *Support function distribution* utilizes parallel processing in the computations required by simulation overhead functions, while retaining the overall sequential nature of the simulation. *Model function distribution* consists of the spatial decomposition and parallelization of a single simulation model. Of the three, model function distribution has by far received the most attention in distributed simulation research. Kaudel asserts that these three approaches could be applied simultaneously to a discrete-event simulation [Kau87].

2.1.1 Application Level Parallelism Application level parallelism takes advantage of the multiple trials that are generally performed in simulation experiments. Execution of a stochastic simulation model is normally replicated in order to obtain reasonable confidence interval estimates of output parameters [LK82]. In addition, simulation experiments are usually conducted for the purpose of comparing two or more alternatives for a system's operation. A model is constructed for each alternative and run (with each model execution replicated as above) [Sha75]. Application level parallelism achieves speed-up by concurrently executing the individual trials on separate processors. Running N replications concurrently, the theoretical speed-up factor is N, achieving what is considered "perfect" efficiency, excluding the negligible overhead of the control functions [Hei86].

Biles et al discuss distributing simulation at the application level using a hierarchical network of microcomputers, combining application level parallelism and support function distribution [BDO85]. In Biles' method, a "tree" configured network of microcomputers is used, with the individual simulation trials performed at the lower levels of the tree and statistical analysis, optimization, and control functions performed at successively higher levels.

Heidelberger provides a theoretical basis for computing the relative efficiency of distributing each simulation trial over multiple processors versus executing parallel independent replications, given a stochastic simulation [Hei86]. Heidelberger's analysis shows that certain statistical considerations, such as initialization bias, are important in determining the optimal combination of model function distribution and application parallelism for achieving the desired accuracy in the minimum time.

The uncomplicated approach of concurrently executing independent simulation trials has received scant attention in distributed simulation literature, possibly because it is so straightforward, and a product of statistical considerations rather than computer science. Kaudel comments as late as 1987 that the use of application level parallelism is untested [Kau87]. Application level parallelism appears to offer a significant source of concurrency for simulation, although its utility naturally depends on the specific application involved.

2.1.2 Support Function Distribution Support function distribution exploits the parallelism available by functionally distributing a simulation. Model- independent support functions, such as accumulating statistics, managing the event set, generating pseudorandom numbers, etc., often comprise a large portion of the required computation in a simulation, up to 80% in one simulation analyzed by J. Comfort [Com82]. Comfort achieved a maximum speed-up factor of 1.4 for one benchmark using a PDP-11 host with three Motorola MC68000 microprocessor systems serving as a pipelined event set processor. The addition of more processors to the pipeline had negligible effect [Com82]. In additional studies, Comfort achieved speed-ups from 1.5 to 1.8 on a system consisting of a PDP-11 host processor and three MC68000 systems added to provide priority queue manipulation, state accounting, and event set processing [Com83]. A method of pipelined event list processing for a shared memory multiprocessor has been proposed by D. Jones, but hasn't been implemented [Jon86a].

Support function distribution is certainly limited by the inherent lack of parallelism in traditional simulation algorithms. Comfort's results indicate that nothing resembling an N-fold speed- up can be expected through support function distribution [Com83]. Jones comments that, hypothetically, support function distribution could be used in conjunction with other forms of parallelism to increase speed-up

[Jon86a]. Kaudel notes that the modest speed-ups achieved with support function distribution might prove it to be a less efficient alternative to the other distributed simulation methods [Kau87].

2.1.3 Model Function Distribution The vast majority of distributed simulation literature deals with methods of distributing the event routines of a simulation model over multiple processors. Indeed, most authors do not address any other possibilities for distributed simulation. Model function distribution exploits the parallelism of the simulated system to determine which events in the system can be simulated concurrently. Kaudel notes that distribution of model functions generally implies the homogeneous distribution of simulation support functions as well [Kau87]. For the remainder of this paper, *distributed simulation* is synonymous with the model function distribution of simulation.

The traditional sequential approach to simulation does not lend itself well to parallelization because of its frequent manipulation of a single data structure, the *future events list* [CM81, RMM88]. The future events list (or simply *event list*) provides an ordering of all events in a simulation, enforcing a totally sequential view of the behavior of a physical system that typically exhibits some degree of concurrency.

To achieve the maximum parallelization of a simulation, the ordering of events is restricted to that dictated by *event dependencies* within the simulated system. If an event B depends on event A, then event A must be simulated before event B.

However, if the two events are independent, they may be simulated in any order, or concurrently. The dependency relation formalizes our intuitive understanding of the order in which events have to occur in a simulated system. Over an entire simulation, the dependency relationship forms an irreflexive partial order of the simulation's events [Mis86]. This was recognized by Lamport, who devised a synchronization protocol that preserves event dependencies within systems of distributed processes [Lam78].

Algorithms for distributing a simulation model over multiple processors typically partition the simulation among several communicating processes, in which each process models a portion of the overall system and communicates with other processes by passing messages. These processes are most often referred to as *logical processes* [CM79] or *objects* [BJ85]. It is interesting to note that an object-oriented view of sequential simulation using message-passing has previously been implemented, due to software engineering considerations, by several simulation languages, most notably *Simula-67* [DN66].

2.2 Distributed Simulation Algorithms

Distributed simulation algorithms differ as to their methods of interprocess synchronization and degree of centralized control. Most distributed simulation algorithms use some method of blocking and unblocking the execution of the logical processes for inter- process synchronization [CM79, Bry79, Rey83], although an alternate method, to be discussed later, has been proposed [Jef85]. When process blocking is used to enforce event ordering, there is the potential problem of deadlock occurring among logical processes, unless provisions have been built into the algorithm to preclude deadlock.

2.2.1 Chandy-Misra Null Message Algorithm K. Chandy and J. Misra proposed one of the original algorithms for distributed discrete-event simulation in [CM79]. In their *Null Message* algorithm, processes communicate exclusively through *messages* to other processes; there is no shared memory or central controlling process. The individual processes run asynchronously, each executing the same algorithm to ensure inter-process synchronization. The algorithm uses "Null" messages to avoid process deadlock, a problem inherent to this algorithm.

In describing their Null Message algorithm, Chandy and Misra define a model of physical systems based on the concept of communicating *Physical Processes* (or PP's) [CM79]. A simulated system consists of a finite number of physical processes, which represent components of the system and communicate exclusively through messages, each message associated with a point in time.

Messages in the physical system are of the form (t, m), where t is the time of message transmission and receipt, and m is the message content. Messages are sent between any two PP's in order of increasing t value.

Chandy and Misra discuss two important properties of physical systems: *realizability* and *predictability*. All physical systems have the property of realizability, which states that "A message sent by a PP at time t is a function of its initial state, t, and the messages it has received up to and including t." In addition, physical systems have the property of predictability. Predictability ensures that "the output of any PP up to any time t can be computed given the initial state of the system" [Mis86].

A physical system of N PP's can be simulated by constructing a simulator consisting of N asynchronous *Logical Processes* (LP's), in which LP_i simulates PP_i. In the logical system, there is a communications channel from LP_i to LP_j if and only if PP_i sends messages to PP_j in the physical system. Messages are assumed to be transmitted correctly using an unspecified communications protocol [CM79].

Process synchronization is accomplished by permitting a process to advance its simulation clock only when it is certain that the process will receive no messages with a timestamp value less than the new time on the process simulation clock. The t value of the last message transmitted over a channel is its *channel clock* value. An LP may advance its simulation clock to the minimum of all its incoming channel clocks. This ensures that no message will arrive to cause an incorrect sequence of events at any LP.

The basic algorithm as described is subject to the problem of process *deadlock* [CM79, CM81]. In a cyclic network, a cycle of LP's with the same simulation time may occur, so that in effect each LP waits for a message that only it can provide. This deadlock may be avoided by requiring at least one LP in each cycle to have some positive delay time between receiving a message and sending a message [CM79].

Another type of deadlock [CM79], may occur even in acyclic networks using this algorithm. In order to advance its simulation clock, a LP must wait for messages on all incoming channels whose channel clock values are equal to the value of the LP simulation clock. If no messages ever arrive on one or more of these channels, the LP is blocked, and can not progress in its simulation.

Null messages are the mechanism used to avoid this possibility of deadlock. A Null message notifies an LP not to expect a "real" message over a channel up to a given point in time.

Its effect is to advance the channel clock of the channel on which it is sent, allowing the recipient LP to advance its clock. The Null message has no other effect on the state of the system, as the message content is "Null."

Chandy and Misra offer proof of correctness of their Null message algorithm by proving "1) chronology of the (message) tuple sequence, 2) correctness of every tuple sequence at any point in simulation, 3) absence of deadlock, and 4) termination of simulation" [CM79].

2.2.2 Bryant's Infinite Buffers Algorithm
R. E. Bryant developed a conservative algorithm for distributed simulation, concurrently with, but independent of, the

early work of Chandy and Misra [Bry79]. Bryant's algorithm is similar to the *Null Message* algorithm in many respects. There are, however, significant differences in the communications protocols and time advance mechanisms. The appellation "Infinite Buffers" [Rey83] was given to Bryant's algorithm because there is no way of knowing a priori the amount of memory that a process in the simulation may require. This is due to the absence of flow control in the inter-process communications protocol [Kau87].

Bryant's algorithm is based on a paradigm of autonomous processes, which, as in the *Null Message* algorithm, communicate through timestamped messages. Messages between two processes are assumed to arrive in the order that they were sent, but not necessarily in the order that the corresponding messages would be sent in the physical system. Process interactions are represented by timestamped "stimulus" messages sent between processes.

Bryant's algorithm, unlike the *Null Message* algorithm, queues up the incoming stimulus messages in a future events list within each logical process. This feature potentially allows complex physical processes to be encapsulated within a single logical process. An event in Bryant's algorithm consists of three steps:

1. A stimulus message is received (or dequeued);
2. a new state of the process is computed based on the old state and the nature and simulation time of the stimulus; and

3. some number (possibly zero) of stimulus messages are sent to other processes and possibly to the sending process itself [Bry79].

Stimulus messages are the method by which the simulation changes its state, but they do not serve as a time advance mechanism. Each process in the simulation maintains its own simulation clock, and can simulate, in time order, any stimulus messages with timestamp value less than or equal to the value of the process clock. Bryant's algorithm utilizes two mechanisms for advancing the process simulation clocks - "time incrementation" and "time acceleration" [Bry79].

Time incrementation uses a special type of message called an increment message that does not simulate a message sent in the physical system, but instead conveys synchronization information, similar to the operation of a null message in Chandy and Misra's algorithm [CM79]. Because stimulus messages might not be sent in chronological order, they can't be used for synchronization. When a process receives an increment message over some channel, the process "knows" it will receive no earlier stimulus messages over that channel, and can increment its channel clock to the time of the increment message. The logical process updates its own clock in a manner similar to that used in the *Null Message* algorithm, simulates pending events, and sends its own increment message over all of its output channels.

Thus, increment messages avoid acyclic deadlock in a manner similar to the way null messages perform in the Chandy-Misra algorithm [CM79]. Cyclic deadlocks are avoided by requiring that for every set of processes in a cycle, at least one has a *delay* > 0, where *delay* is the minimum possible simulated time difference between receipt of a stimulus and the resulting output stimulus.

Time acceleration is an additional synchronization mechanism that expedites time advance within cyclic portions of the simulation. Peacock et al noted that significantly high message overhead and inefficiency may result when cycles of processes in a distributed simulation network are left to iteratively increment their simulation clocks by the minimum values, characterized as the "*pseudo-time-driven effect*" [PWM79a].

Time acceleration requires an analysis of the system's interconnections a priori, in order to identify the cyclic portions. The simulation is partitioned into a set of equivalence classes, where the members of each class form a cycle within the simulation network, or the class consists of an individual process which belongs to no cycles. For each cycle in the simulation network, time acceleration identifies the earliest possible time of the next event in the cycle, and advances the clocks of all processes in the cycle to that time.

Time acceleration is implemented by arbitrarily selecting a process in each cyclic class to send out periodic test messages to all processes in its class to which it has a channel. The test messages circulate through the communications channels of the class. Each process that the test message passes through updates it, if necessary, so that the test message always contains the time of the earliest potential event of any of the processes in its class that it has encountered. When all test messages have returned to the sending process, the sending process takes the minimum of the times contained in the messages. The test process then sends a wave of "set" messages to the processes in its class, ordering them to advance their clocks to the new time.

Certain processes in a simulation are identified as source processes, which send stimulus messages but do not receive any. Termination of a simulation is accomplished by causing each source process to send an increment message with timestamp of infinity once it has sent all of its stimulus messages. When a process has received such a message over each of its incoming channels, it will receive no more stimulus messages. The process simulates all pending events, sends out infinity increment messages of its own, and finally terminates.

2.2.3 Peacock et al - Link Time/Blocking Table Algorithms J.K. Peacock, J. Wong, and E. Manning, after collaboration with Chandy and Misra, published a version of the Null Message algorithm called the *Link Time* algorithm [PWM79b]. Peacock et al don't address required memory in their algorithm; however, there is no mention of processes blocking due to send operations [PWM79b]. The Link Time algorithm can then be considered almost equivalent to the *Null Message* algorithm, without the flow control provisions of the latter.

Peacock et al outline several other conservative algorithms in [PWM79b], including their *Blocking Table* algorithm. In this algorithm, every process maintains a list of other processes that can reach it by a path of empty links, and could therefore cause its next event to be preempted. No detailed version of the algorithm was provided. However, updating the blocking tables for each node appears to be a complicated and arduous task.

A method for "tight" event-driven simulation, in which a global simulation time is enforced and the next event over all processes is found and simulated, as well as several methods for time-driven discrete-event simulation were also addressed by Peacock et al [PWM79b].

2.2.4 Chandy-Misra Deadlock Detection and Recovery Algorithm In [CM81], Chandy and Misra published a distributed simulation algorithm based on deadlock detection and recovery, developed as an alternative to their *Null Message* algorithm, which had been shown to incur a large message overhead in simulations with feedback. The simulation detects deadlock in a distributed manner [MC82], and a central process, the controller, issues instructions to the appropriate processes to initiate deadlock recovery. The new algorithm was constructed using the same simulation paradigm of message-passing physical processes as used in the *Null Message* algorithm [CM79, Mis86:described above].

Chandy and Misra characterize simulation as being one of a class of problems in which phases of the problem may be solved in parallel, but the phases themselves must be performed in sequence, yielding "a sequence of parallel computations [CM81]." In this structure, synchronization is required only at phase interfaces. The termination of a phase manifests itself as a deadlock in the simulation problem, so that the algorithm for each logical process consists of the iteration of the following sequence: **1) Parallel Phase** - Simulate until deadlock occurs; **2) Phase Interface** - Initiate computation to break the deadlock [CM81].

In the parallel phase, logical processes (or LP's) behave similarly to the asynchronous logical processes defined in the Null Message algorithm [CM79], except of course that they only transmit non-null messages. The communication protocol assumes bounded buffers of an arbitrary size, and a LP attempting to send a message will be blocked if the recipient process's input buffer is full. Chandy and Misra note that this protocol can be used to ensure that the sum of the memory used by the all of the LP's is equivalent to that used by a conventional sequential simulation [CM81] (at some cost to speed-up, naturally).

Deadlock detection is performed in a distributed manner using an algorithm based on the work of Dijkstra and Scholten in "Termination Detection for Diffusing Computations" [DS80]. Dijkstra and Scholten's algorithm deals with a generalized "diffusing" computation, in which computation in the distributed system is started by an outside process, the environment, sending a single message to an initiating process. When a process receives its first message, it is then free to send messages to other processes. When a process terminates, it sends signals back along the channels from which it has received messages, constrained by the number of signals it has received from processes to which it has sent messages. The protocol guarantees that termination will be detected by a signal sent from the initiating process to the environment, within a finite number of steps after all processes have terminated [DS80].

Misra and Chandy's termination detection scheme [MC82] modifies Dijkstra and Scholten's algorithm to work under the constraints of C.A.R. Hoare's protocol for communicating sequential processes [Hoa78, Hoa85]. Dijkstra and Scholten assume that processes may freely send messages [DS80]; Hoare's protocol adapted by Misra and Chandy assumes that a process must wait to send if the receiving process is not waiting to receive [MC82]. As a result, an idle process can not determine the cause of its own idleness unless it first queries its neighbors. The implementation of the termination detection scheme uses two types of signals: A signals, corresponding to signals defined in [DS80], and B signals, used by a process to determine the waiting status of its connected processes, and thus its own status [MC82].

When the controller process (the *environment*) has received a deadlock detection notice, it sends a signal to each LP_i to compute and output the best lower bound W_{ij} for the time of next message transmission over each of its output channels (i,j). This is accomplished in n iterations (n being the number of LP's), as described in [CM81]. For each LP_i, the iterative computation allows the values W_{ki} for all input channels (k, i) computed in iteration m to be used in computing W_{ij} during iteration $m + 1$, where $1 \leq m < n$.

After computing W for all output channels, each LP must report its resumable status to the controller. A LP is defined to be resumable if the set of channels it is *waiting on* is different from the set that it was waiting on when deadlock occurred. After reporting to the controller, each LP is free to continue Phase 1 computations [CM81].

The authors of this algorithm have since proposed alternative methods for deadlock detection. In [CM83], Chandy and Misra, along with L. Haas, describe a communications deadlock detection protocol based on the concept of the *dependent set*.

The dependent set of a process is the set of processes such that a message from a member of the dependent set will cause the process to resume execution. Any process, upon finding itself idle relative to its computational work, initiates a query to all members of its dependent set. If the process receives replies equal to the number of queries it sent, then it is deadlocked. Only processes that are idle take action in response to a query. When an idle process receives a valid query, it propagates it to each member of its dependent set. When a process not the query process receives replies from each member of its dependent set, it issues a reply to the process which queried it. This algorithm is somewhat flexible in that a time-out may be used to delay a query computation until the process has been idle for some arbitrary time [CM83].

Misra has proposed and proven a deadlock detection scheme based on a *marker*, a special message that continuously travels a path that takes it once through every LP in the simulation network [Mis86]. Additional message channels may be required to create the marker path. In the marker algorithm, any LP that receives the marker has the obligation to send it on its way when the LP becomes idle. Each LP maintains a flag to signify whether or not the LP has received or sent any messages since the last visit by the marker. The marker detects deadlock if, with n LP's in the network, it visits n consecutive LP's that have not communicated since the marker's last visit. The marker can also be used in deadlock recovery. For that purpose, the marker can record the minimum next event time of all the idle LP's it has visited and the identity of the corresponding LP. Upon detection of deadlock, the marker traverses to and restarts this LP by advancing the LP clock to the next event time. Markers can also be tailored for individual networks. By analyzing the simulation network prior to execution, additional markers can be set up to circulate within subnetworks that are prone to deadlock [Mis86].

2.2.5 SRADS Algorithm P. Reynolds introduced an algorithm for distributed simulation called the *Shared Resource Algorithm for Distributed Simulation (SRADS)* based on the concept of *active logical processes* [Rey82, Rey83]. In most distributed simulation methods, logical processes are relatively passive, unable to perform unless driven by messages sent by other processes. In the *SRADS* algorithm, when a process needs to read or write a message, it attempts to access a shared facility, which is the storage point for messages between a set of two or more LP's. The sequencing requirement and associated protocol guarantees that to access the shared facility, an LP must have the least simulation time of any LP connected to that shared facility.

Each shared facility holds only a single message, so it is possible for a message to be overwritten without ever being read.

Although it is deadlock-free, the *SRADS* algorithm is limited in that it assumes that the processes in the underlying physical system are synchronized [Rey82]. The assumption is that a reading process "knows" to poll the shared facility for a message at a particular simulation time. If this is not the case, as in many simulations, an LP may read an outdated message, resulting in an incorrect simulation [NR84].

Reynolds and D. Nicols subsequently proposed an extension to *SRADS* called the *appointment* [NR84], which is a message that a writing process sends to a reading process, providing the largest known lower bound on the time that a message will be sent to the shared facility. When the reader reaches the appointment time, it is blocked until the writer gives it a signal to unblock. This does prevent the reader from simulating too far and then reading a message from the past, although it isn't clear what, if any, advantages *SRADS* plus appointments offers over other blocking algorithms, given its inherent limitations.

2.2.6 Optimistic Algorithms - Time Warp The distributed simulation algorithms described above have been characterized as "conservative" [Rey83] or "pessimistic" algorithms [RMM88]. *Conservative* algorithms are those which preserve the proper sequence of events throughout the entire simulation [Rey83]. In other words, at any time during execution of the simulation, the order in which events have been simulated at any LP is correct and preserves event dependencies, although the logical processes have not necessarily simulated the physical system up to the same point in time. Conservative algorithms can be considered pessimistic, since they imply that if LP's were left to run asynchronously, the frequency of event ordering conflicts would be high, and it is therefore more efficient to prevent such conflicts than to correct them as they occur.

In 1982, D. Jefferson and H. Sowizral introduced an interesting alternative to conservative distributed simulation algorithms, the *Time Warp* algorithm [JS85], based on the concept of *virtual time*, which Jefferson analogizes to virtual memory [Jef85]. In contrast to the blocking-advance mechanisms used in conservative algorithms, *Time Warp* includes a local time rollback mechanism for synchronization. Processes, called objects, execute asynchronously, sending messages without regard to potential synchronization conflicts. When an existing synchronization conflict comes to light, a rollback occurs and portion of the simulation must be re-executed. Because the algorithm assumes that synchronization conflicts won't occur often, *Time Warp* can be considered an *optimistic* algorithm.

Each *Time Warp* process maintains a local virtual clock, set to the timestamp of the last message read by the process. Messages received are placed in an input queue by increasing order of message timestamp. A process continues to read from its input queue and calculate output messages until it reads a message with a timestamp less than its local virtual clock. The process must then roll back its local virtual clock to the value of the newly- received message timestamp. This implies that all messages already sent to other processes during the rolled-back period must somehow be cancelled, and all events re-simulated with the addition of the new message. Local virtual time can not, then, be considered a measure of actual progress in a simulation, as it may be rolled back to a previous point [JS85].

Re-simulating an interval is simple, in that recently input messages and previous values of the state variables are maintained by each process. Message cancellation is accomplished through the use of *antimessages*. When a process outputs a message, it saves a copy for possible later use as an antimessage. If a message must be cancelled due to rollback, the roll-back process need only send the corresponding antimessage. If a matching message and antimessage meet in the input queue of a process, they are mutually annihilated, cancelling the message. If the message has already been processed, receipt of the antimessage causes the process to roll back to the time of the message, which is removed from the input queue, and processing continues. Of course, this rollback may spawn antimessages itself, so that a single rollback may initiate a "ripple" of rollbacks and antimessages through the simulation [Jef85].

Processes would be subject to rollback to the beginning of the simulation and would therefore have to store their entire message histories, if it were not for *Global Virtual Time*. Global Virtual Time is calculated as the minimum of all local virtual clocks and the send timestamps of all messages that have been sent but not processed. Events that happened prior to Global Virtual Time are irrevocably committed, so that processes must only maintain their message histories from Global Virtual Time onward. Note that Global Virtual Time can be used as a measure of a simulation's progress, as it is non-decreasing [Jef85].

Time Warp has several advantages. As an algorithm it is relatively simple and quite elegant. *Time Warp* doesn't require that messages be received by a process in any order to maintain correctness of the simulation. Processes may be created and destroyed in the course of the simulation.

This flexibility does come at a cost, however. The memory requirements of *Time Warp* are quite high. All messages generated since Global Virtual Time are stored twice – once in the sending process and once in the receiving process. This requirement for memory may be traded-off against the cost of computing Global Virtual Time more frequently. Finally, there is the computational time wasted by rolling processes back. Jefferson and Sowizral claim that the amount of processing time that a process spends on roll-backs in *Time Warp* would be spent blocked in an equivalent simulation using conservative algorithms [JS85]. It seems, though, that the processor hosting a blocked process could still perform useful simulation support computations. Processing time spent on roll-backed events is simply lost.

2.3 Performance Studies of Distributed Simulation

A decade has elapsed since the first algorithms for distributed simulation were published, and there is still only a small knowledge base derived from empirical testing of these algorithms. This is to a certain extent attributable to the lack of available multiprocessor computing systems for research use. To partially overcome this handicap, some performance studies of discrete-event simulation have been accomplished using uniprocessor systems, ironically, to simulate the operation of distributed simulation systems [CM79, Ree85]. The large number of factors affecting the performance of distributed simulation has also hindered the development of heuristics to guide the use of distributed algorithms for simulation.

2.3.1 Performance Measurement The primary metric used to measure the effectiveness of distributed simulation is the ***speed-up*** factor. Speed-up conveys the throughput advantage of performing a computation over multiple processors. Speed-up is often expressed as a function of the number of processors, and is said to be *linear* if Speed-up increases linearly with the number of processors. The formula for Speed-up factor is T_1/T_n, where T_n is the time to execute the simulation over n processors. There are, however, multiple definitions for T_1 in use in distributed simulation literature.

If one is interested in the absolute speed advantage gained by a distributed simulation over a sequential algorithm such as the event list, then T_1 would be defined as the time to execute the simulation using the sequential implementation [Fuj88]. If one wishes to emphasize the marginal effect of each additional processor on the performance of a given distributed algorithm, then T_1 can be defined to be the time to execute the distributed algorithm on a system of one processor [RM88, RMM88]. Both of these definitions can be commonly found in the literature.

Although Speed-up is the metric of primary concern, in certain cases, intermediate metrics may be desirable to quantify a certain effect that may impact the attainable Speed-up. Fujimoto defines the *null message ratio* to be the number of null messages processed in an implementation of the *Null Message* algorithm divided by the number of "real" messages processed. The *deadlock ratio* is the number of messages processed divided by the number of deadlocks that occur in a deadlock detection and recovery algorithm.

2.3.2 Factors Affecting Performance The factors that may affect the performance of a distributed simulation algorithm are extremely diverse, arising from the nature of the system being simulated, the model constructed of the system, and the hardware and software environment under which the distributed simulation is performed.

2.3.2.1 Structure of the Simulated System Early distributed simulation studies recognized that the structure of the simulated system has a significant effect on performance [PWM79a]. The topology of interconnections between LP's has been studied as a variable in almost every study of distributed simulation to date. A related factor, the routing probability of messages, has been shown to affect performance in some algorithms [PWM79a]. The distribution of the message timestamp increment, which is the difference in the timestamp value of a message input by a PP and the timestamps of the resulting output messages, has been shown to be significant. The value of *lookahead*, the lower bound on timestamp increment, has been shown to be especially significant in the performance of distributed simulation algorithms [Fuj88].

2.3.2.2 Simulator Workload The amount of computation performed by a LP between message input and output, also called the *CPU grain* has been shown to positively correlate to the resulting speed-up, [Fuj88, DR83], although the effect of varying the grain appears to be interrelated with the effects of topology [RM88].

Also of great interest is the workload balance, which seeks to even out the computational load among the available processors. Workload balancing is a topic of interest within the distributed processing community because better workload balance implies higher throughput, excluding the costs of *interprocessor communication (IPC)*. Chu et al describe a *saturation effect* in which the overhead of IPC increases to the point that throughput decreases with the application of each additional processor to the problem [CH*80]. Hence, in the decision process of partitioning a task and allocating the sub-tasks to processors, a trade-off exists between workload balance and IPC. Chu et al note that finding an optimal allocation of tasks is computationally complex, and that it is generally more efficient to use a sub-optimal allocation guided by heuristics [CH*80].

Machine architecture partially determines methods available for task allocation. In shared-memory systems, either *static allocation*, where tasks are permanently assigned to processors, or *dynamic allocation*, where idle processors take work from a queue of processes, can be used [RM88]. In loosely-coupled message-passing systems, static allocation is used; dynamic allocation is considered impractical due to communication overhead.

2.3.2.3 Implementation Factors The software implementation (e.g. the compiler used, the efficiency of the code) and the machine architecture (e.g. the amount of memory, the communication overhead, the existence of shared memory) need to be controlled, and their effects normalized, before direct comparisons can be made among the results of performance studies of distributed simulation. Failure to adequately do so may result in comparative judgements made on the basis of implementation factors, rather than on the merits of the respective algorithms. The ability to normalize the results of previous performance studies eliminates the need to replicate these studies in comparatively evaluating some new method.

2.3.3 Chronology of Performance Studies Early performance studies of distributed simulation were typically concerned with demonstrating the feasibility of some particular algorithm. Peacock et al [PWM79a] performed experiments with their version of the *Null Message* algorithm on a network of microcomputers. Their results indicate that the topology of the simulation model is a major factor in the algorithm's performance, and that topologies with many cycles may suffer from poor performance. This study was limited in scope; only a few simple topologies were distributed over different numbers of nodes; no other parameters were varied. Similar experiments, described in [CM79], were conducted by M. Seethalakshmi using a uniprocessor simulation of a distributed system. These studies show that the *Null Message* algorithm may not be suitable for simulations of all topologies. In particular, Seethalakshmi's results appear to have motivated the development of Chandy and Misra'a Deadlock Detection and Correction algorithm [CM81].

D. Reed, in [Ree85], performed a uniprocessor simulation of several topologies with varying populations of messages, using both the Null Message [CM79] and Chandy-Misra Deadlock Detection and Recovery [CM81] algorithms. His results supported the findings of [PWM79a] concerning the effect of feedback cycles on the performance of the *Null Message* algorithm. Low message populations were also implicated in causing poor performance of the *Null Message* algorithm. The conclusion drawn was that deadlock detection and recovery would engender less overhead than deadlock avoidance, especially where there is much feedback and a small message population.

In more recent research, R. Fujimoto evaluated the performance of the *Null Message* or "Deadlock Avoidance" and Deadlock Detection and Recovery algorithms on a BBN Butterfly shared memory multiprocessor under a variety of controlled conditions [Fuj88]. A symmetric toroid topology with 64 and 16 logical processes was used for the experiments. The experiments assumed that a good workload balance could be found, so that in most experiments, symmetric logical processes were used.

Important performance considerations were shown by Fujimoto to include the quality of the lookahead function for timestamp incrementation in the deadlock avoidance algorithm, and a sufficiently large message population in the case of the deadlock detection and recovery algorithm. In the latter, an "avalanche effect" was found, such that performance dramatically improved when message traffic reached a certain level. An increase in the message population required to start the avalanche effect resulted from introducing a process with an asymmetric timestamp distribution function. Otherwise, asymmetry did not significantly effect performance. Dynamic process allocation was possible because of the shared memory architecture of the simulation testbed, but was generally shown to hurt performance compared to the static allocation, most likely due to the "perfect" load balancing in the simulator [Fuj88].

D. Reed, A. Malony, and B. McCredie analyzed the performance of both the Deadlock Avoidance and the Deadlock Detection and Recovery algorithms on a Sequent Balance 21000 shared-memory multiprocessor [RMM88]. Their experiments evaluated distributed simulations of several queueing network topologies using the *RESQ* paradigm for queueing models, including simple tandem, feed-forward (forked), cyclic, central server, and cluster (nested feedback) networks. Reed et al point out that studies of simple topologies do not reflect typical simulation problems, while complex topologies, though realistic, increase the difficulty in finding the causes of poor performance in a distributed simulation. In this study, the effects of dynamic node allocation were studied in combination with a variety of message workloads.

The results of this study were significantly more negative than those of Fujimoto [Fuj88]. Speed-up approaching N was not achieved even in the tandem case with more than a few nodes. This is surprising, especially considering that the Speed-up factor was calculated using the distributed algorithm executed on one processor (which invariably takes more time than an event list implementation) as baseline, yielding an artificially high speed-up. A possible explanation given was that some of the poor performance may be due to machine architecture constraints such as bus and memory contention. Performance was said to have improved noticeably when PP's were dynamically assigned to processors for the cluster topology. Assumptions that may have affected the results include limited lookahead in conjunction with the deadlock avoidance algorithm, and no a priori knowledge of the simulation network.

Reed and Malony conducted further studies in which the amount of computation at each node, the "spin loop," was varied to observe the effects on central server and cluster network topologies, considered to be the most representative of typical simulation models [RM88]. The central server model distributed with deadlock recovery showed no effect from increased spin loop, because the presence of the "bottleneck" central server led to regularly occurring deadlock. In contrast, the same topology with deadlock avoidance yielded performance improvement when the spin interval was increased. Simulating the cluster network topology with deadlock recovery showed, as with the central server, the relative independence of Speed-up and spin interval. In a surprising result, the cluster model with deadlock avoidance suffered decreased Speed-up as spin interval was increased, in direct opposition to the central server model results. This can be explained when one considers the large number of forks in the cluster network, and that large time intervals between message transmissions lead to a "decoupling" of the LP's, and increased Null message overhead. In this case, supposedly distinct factors, topology and spin interval, were actually interrelated in their effects on distributed simulation performance [RM88].

2.4 Summary

The majority of the algorithms for distributed simulation have been in existence since the late 1970's and are therefore quite mature; most are provably correct and relatively straightforward. Distributed simulation algorithms have been divided into two categories: Conservative and optimistic algorithms. Performance studies have been conducted on both types of algorithms, considering many of the factors that are expected to significantly affect performance. Some research data is available to be used in the formation of heuristics for applying these algorithms. Additional performance studies are required, attempting to normalize the effects of the implementation factors, so that algorithm performance can be estimated and compared over a wide range of conditions.

III. Description of Algorithms

The algorithms under study are distributed simulation algorithms built on a paradigm of physical systems based on the message-passing paradigm of Chandy and Misra, while incorporating concepts of traditional *event-oriented* simulation. The physical system may be simulated by a distributed logical system using the conservative synchronization algorithm proposed by Chandy and Misra [CM79], slightly modified with a prediction function, and utilizing an event list, similar to that used in traditional sequential simulation, at each *Logical Process (LP).*

These algorithms are of interest because it may often be necessary or desirable to simulate a system of N components or *Physical Processes (PP's)* with a distributed computing system consisting of fewer than N processors. Extending the message-passing paradigm of Chandy and Misra to create multiple processes within a single processor introduces the problem of process scheduling, along with complicating deadlock resolution. The goal is to find an efficient method for simulating multiple PP's within a single LP.

A natural candidate algorithm for simulating a complex physical process within a single logical process is the event list method used in sequential simulation. The linear nature of the event list with respect to time ensures that the events within each LP are simulated in strictly chronological order. The chronological ordering provided by a monolithic event list, combined with the interprocessor synchronization method of Chandy and Misra, guarantee that event dependencies are preserved. The Chandy–Misra Null Message algorithm is easily extensible to accommodate an event list implementation at each logical process.

This chapter presents the message–passing paradigm of physical systems proposed by Chandy and Misra [CM79], modifies it to conform to an "event–oriented" view, and applies the resulting paradigm to a logical system for distributed discrete–event simulation. The problem of deadlock in the logical system and the use of Null Messages for deadlock avoidance are discussed. Alternative strategies for sending Null messages are presented. These variants, which may enhance the speed of the logical system's execution under certain circumstances, will be evaluated in Chapter 4.

3.1 The Physical System

Chandy and Misra have defined a paradigm of physical systems based on the concept of message–passing PP's in [CM79]. While this paradigm maps well into a logical system for distributed simulation, it does not capture all aspects of the time and state relationships in a physical system that may be of interest. To do so, the Chandy–Misra physical system paradigm can be reconciled with some "traditional" concepts of discrete–event system modelling. While the message–passing model of physical processes seems to obviate concepts such as events, for example, events can be defined implicitly in terms of the state changes of individual physical processes.

The concept of the *message* can be re- stated in terms of *event dependencies* among PP's [Mis86].

3.1.1 Definitions It has been observed by R. Nance that the evolution of simulation has resulted in the existence of many differing paradigms of physical systems, such as *event-oriented* and *process-oriented* views, and "the inability to generalize the simulation modelling task [Nan81]." This state of affairs has, in turn, resulted in considerable imprecision in the terminology of simulation. Nance has proposed a set of basic definitions to clarify the time and state relationships in the simulation of physical systems [Nan81]. The following definitions are incorporated into the description of the physical system:

- An *Object* is anything that can be characterized by one or more *Attributes* to which values are assigned.
- A *System* is comprised of objects and relationships among objects.
- The *State* of an Object is the enumeration of all attribute values of that object at a particular instant.
- An *Event* is a change of object state, occurring at an instant, that initiates an activity precluded prior to that instant.
- An event is *Determined* if the only condition on event occurrence can be expressed strictly as a function of system time. Otherwise, the event is *Contingent.*

- An *Object Activity* is the state of an object between two events describing successive state changes for that object.

3.1.2 The Physical Process In describing their Null Message algorithm, Chandy and Misra define a model of physical systems based on the concept of communicating physical processes [CM79]. A simulated system consists of some finite number N of physical processes (or PP's) which represent the components of an actual physical system. The PP's communicate with each other exclusively through messages. A message in the physical system can be described as a *tuple* of the form (t, m), where t is the time of transmission and m is the message content. If PP_i sends messages to PP_j, there exists a *message arc* [CM79] (i,j) between PP_i and PP_j. Messages are transmitted instantaneously.

In the physical process paradigm, each PP is described in terms of messages sent and messages received. The sequence of messages transmitted over arc (i,j) is of the form $s_{ij} = \{(t_1, m_1), \cdots, (t_K, m_K)\}$ where K is the finite number of messages transmitted over (i,j) during the simulation period $[0, Z]$. Note that $0 < t_1 < \cdots < t_K \leq Z$, reflecting the chronological order of message transmission. Chandy and Misra define the *message history* of arc (i,j) at time t, $h_{ij}(t)$, as the sub-sequence of s_{ij} denoting the message tuples transmitted over arc (i,j) up to and including time t. The state of PP_i at time t is represented by a set of state variables $\Psi_i(t)$. In the Chandy-Misra paradigm, the existence of some function b_i is assumed, such that:

$$\Psi_i(t+) = b_i(\Psi_i(t),\ INFORM_{ki}(t)\,\forall k) \tag{3.1}$$

where $t+$ is the instant in time directly after time t, and $INFORM_{ki}(t)$ is the information sent across arc (k, i) at time t, possibly NULL (signifying no event), or an event message (t, m) [CM79]. By applying 3.1 over an interval $[t, t')$, where $t < t'$, the function B_i can be derived [CM79]:

$$\Psi_i(t') = B_i(t', \Psi_i(t), h_{ki}(t, t')\,\forall k) \tag{3.2}$$

where $h_{ki}(t, t')$ is the sequence of messages over (k, i) in the time interval $[t, t')$. The existence of B_i means that the state of PP_i at time t can be computed by knowing the state of the PP_i at some previous time, and all messages that have been received in the interval [CM79].

In certain instances it is possible to predict with certainty the messages sent by a PP_i up to some future time t' given $h_{ki}(t)$ for all k. For example, consider a single–server, single–queue queueing process with one input arc corresponding to a service arrival, and one output arc corresponding to a service completion. With a constant inter–service time of 10 with no pre–emption, the output of the process can be deduced up to time $t + 10$, where t is the time that a job entity begins service. The value $t' - t$, associated with each arc in the physical system, is called the *Lookahead* function at time t for arc (i, j), or $L_{ij}(t)$ [CM79]. Lookahead is computed with the following function, which is also assumed to be computable [CM79]:

$$L_{ij}(t) = F_{ij}(\Psi_i(t),\ INFORM_{ki}(t)\,\forall k) \tag{3.3}$$

Lookahead value L_{ij} provides the bound on prediction of the future output of PP_i over (i,j), calculated by function C_{ij}:

$$h_{ij}(t, t + L_{ij}(t)) = C_{ij}(\Psi_i(t), INFORM_{ki}(t)\,\forall k) \tag{3.4}$$

which is assumed to be computable [CM79].

For each PP_i, there exists a set of points in time t_e in the interval $[0, Z]$ such that $\Psi_i(t_e)$ is different from $\Psi_i(t_e+)$, where t_e is the instant in time preceding time t_e+. These points in time are events, in accordance with the above definition. Taking the set of instants in time at which Ψ_i changes and the values of Ψ_i at those times yields the state "trajectory" of PP_i, the relation between Ψ_i and simulation time t, $STRAJ(\Psi_i(t), t)$, where $0 \leq t \leq Z$ [Zei76].

By 3.1, the receipt of a message at PP_i is an event at PP_i. It is apparent from 3.4 that a message transmission at PP_i also reflects a change in Ψ_i. Instants in time where changes in Ψ_i occur are not explicitly defined in the message-passing paradigm unless they are associated with a message transmission or reception.

3.1.3 Event-Oriented View of the Physical Process In the message-oriented paradigm discussed above, the state of PP_i, Ψ_i, is a function of its initial value and the messages received by PP_i over the simulation period. While this is a valid paradigm of physical systems, this view does not emphasize the changes in Ψ_i over time. In the event-oriented view, changes in Ψ_i, i.e., events, have primacy. The physical process can be defined terms of events as follows:

Physical process i is described as the tuple

$$PP_i = \langle M_i, E_i, \Psi_i, N_i, B_i, D_i \rangle$$

where M_i is the set of incoming message event types
E_i is the internal event type
Ψ_i is the state variable set for PP_i
N_i is the internal event generating function
B_i is the set of state transform functions
D_i is the message generating function

3.1.3.1 External Events A message received at PP_i from PP_k is an event of type M_{ki} associated with message channel (k,i), where $M_{ki} \in M_i$. Such a message transmitted at time t in the physical system is described by the tuple (t, m_{ki}). abbreviated $m_{ki}(t)$. For each $M_{ki} \in M_i$, $\exists B_{ki} \in B_i$, providing a mapping such that: $B_{ki} : \Psi_i \times M_{ki} \longrightarrow \Psi_i$. Given a message $m_{ki}(t)$ and $\Psi_i(t)$, the values of the state variables of PP_i at time t, then:

$$\Psi_i(t+) = B_{ki}(\Psi_i(t), m_{ki}(t)) \tag{3.5}$$

where $t+$ is the instant in time subsequent to t. Note that B_{ki} is a computable function from the assumption made previously that 3.1 is computable.

3.1.3.2 Contingency and Time Advance A message event received at PP_i initiates an activity of some duration (possibly infinite), during which Ψ_i is invariant. It may be possible to calculate the value t_a, where t_a is the length of time that PP_i will remain in $\Psi_i(t)$ if no external events occur (the "time advance" value)

[Zei76]. The value t_a is calculated with function A_i where:

$$t_a = A_i(\Psi_i(t)), where\, t_a \geq 0 \tag{3.6}$$

If t_a is not defined, then Ψ_i is a *passive* state. Note that A_i is a computable function if 3.2 is computable, which is assumed, taking $h_{ki}(t_{in}, Z) = NULL$ for all k. Time $\acute{t} = t + A_i(\Psi_i(t))$, the simulation time of the next scheduled internal event at PP_i, can then be computed.

Chandy and Misra do not permit any PP_i to send event messages to itself, since "that effect can be achieved by a process looking at its own computation" [CM79], presumably by applying 3.4 over all output lines (i, j) to determine the next event message sent by PP_i, and then using 3.2 to update Ψ_i. However, this procedure hides the true nature of the state change. The time and state relationship is made explicit by computing, at any simulation time t, an internal event of type E_i, $e_i(\acute{t})$, where $\acute{t} = t + A_i(\Psi_i(t))$, $t \leq \acute{t}$. If 3.2 is indeed computable, as is assumed, then $e_i(\acute{t})$ is derivable from $\Psi_i(t)$. The computable function N_i can be defined such that:

$$e_i(\acute{t}) = N_i(\Psi_i(t), t) \tag{3.7}$$

If $\Psi_i(t)$ is a passive state, then $N_i(\Psi(t), \hat{t}) = (Z, NULL)$ for any $\hat{t} \geq t$.

If $\Psi_i(t)$ is not passive, then the next internal event $NEV_i(t) = N_i(\Psi_i(t), t)$ can be scheduled on a contingent basis. Whether the event actually occurs is dependent upon event messages received in the interval $[t, \acute{t})$. If $NEV_i(t) = e_i(\acute{t})$ and message $m_{ki}(\hat{t})$ arrives at PP_i, $t < \hat{t} \leq \acute{t}$, then $\Psi_i(\hat{t}) = b_{ki}(\Psi_i(t), m_{ki}(\hat{t}))$ and the new next internal event is $NEV_i(\hat{t}) = N_i(\Psi_i(\hat{t}), \hat{t})$ (which may be the same as $NEV_i(t)$). If $NEV_i(t) = e_i(\acute{t})$, then $\Psi_i(\acute{t})$ is calculated by:

$$\Psi_i(\acute{t}) = B_i(\Psi_i(t), e_i(\acute{t})) \tag{3.8}$$

3.1.3.3 Simultaneous Events Multiple event messages arriving at PP_i can lead to the occurrence of simultaneous events at PP_i, as can an event message arriving at the same instant as a scheduled activity–ending internal event. The treatment of simultaneous events that affect the same set of state variables has not received a great deal of attention in simulation literature. In most applications of discrete–event simulation, simultaneous events are either independent or dependencies can be safely ignored [Par69]. This is not always the case, however, and in attempting to formulate a general paradigm for simulation, this issue must be addressed.

The concept of *transitory states* is introduced from [Zei76]. These are states between two events, such that the activity delineated by those events has zero duration. A transitory state at time t at PP_i is $\Psi_i^n(t)$, where n is the number of events that have already occurred at PP_i at time t. With N events $E_n(t), n = 1, \cdots, N$ occurring at time t, where $E_n \in M_i \cup E_i$, $\Psi_i(t)$ will take on N transitory states:

$$\Psi_i^0(t) \xrightarrow{E_1(t)} \Psi_i^1(t) \xrightarrow{E_2(t)} \cdots \xrightarrow{E_{N-1}(t)} \Psi_i^{N-1}(t)$$

$$\textit{and } \Psi_i^N(t) \xrightarrow{E_N(t)} \Psi_i^0(t+)$$

Note: while Ψ_i is no longer a function strictly of time, the function argument notation will be used for consistency to identify the time reference of a value of $\Psi_i^n(t)$. $\Psi_i(t)^+$ denotes the state of PP_i subsequent to a $\Psi_i(t)$, which is either at time t or time t+.

Events $m_{ki}(t)$ and $m_{gi}(t)$ are said to be independent if:

$b_{ki}(b_{gi}(\Psi_i, m_{gi}(t)), m_{ki}(t)) = b_{gi}(b_{ki}(\Psi_i, m_{ki}(t)), m_{gi})$.

In other words, the same Ψ_i will result from simulating the two events in arbitrary order. If these events are not independent, an ordering relation at PP_i, $ORDER_i$, is included in the description of PP_i, such that if (M_{ki}, M_{gi}) in $ORDER_i$, then $m_{ki}(t)$ must be simulated before $m_{gi}(t)$ to yield a correct result. Failure to consider this possibility may result in the equivalent of a "race" condition in digital logic [Par69]. Unfortunately, it will be seen that the distributed event list algorithm can not prevent this condition from occurring in many instances.

3.1.4 Message-Passing A message in the physical system at time t can be viewed as a manifestation of the dependency relation, d, over the set of all events in the physical system, such that if $d(E_x, E_y)$, the occurrence of event E_x implies the occurrence of E_y. This relation can be defined over the subdomain $\{E_i\} \times M_j$ for every message arc (i, j) by some function $D_{ij} \in D_i$, where:

$$m_{ij}(t) = D_{ij}(\Psi(t), e_i(t)),\ D_{ij} \in D_i \tag{3.9}$$

If a message m_{ij} is determined by event e_i as above, then e_i is the *prompting event* of m_{ij}. By convention, a message may only be transmitted at the instant that its prompting event has occurred. No contingent messages are permitted. For any message over (i, j), no time elapses between the internal event at PP_i that initiates the message and the corresponding message event at PP_j. A message transmission of zero duration may seem counter-intuitive, but it is important to remember that a message reflects the simultaneous change of state of interdependent physical processes: it is not itself a physical message.

3.1.5 Predictability The dependency relation d over the events in the physical system forms an irreflexive partial order (irreflexive, so that no event depends upon itself). Intuitively, the dependency relation d reflects the order in which events must occur in the system. Pairs of events for which d is not defined may be simulated concurrently, or in arbitrary order [Mis86].

Suppose a cycle of PP's exists, such that for PP_i, $i = 1, \cdots, n$, message arcs $(i, (i+1) mod\, n)$, $i = 1, \cdots, n$ exist. If every event message $m_{ij}(t)$, $j = (i+1) mod\, n$, initiates an activity of duration 0, at the conclusion of which an event message $m_{jk}(t)$, $k = (j+1) mod\, n$ is sent, then there exists a circular definition such that

$d(m_{ij}(t), m_{jk}(t))$, $i = 1, \cdots, n$, $j = (i+1) mod\, n$, $k = (j+1) mod\, n$ exists,

and hence $d(m_{ij}(t), m_{ij}(t))$ exists (due to transitivity). As a result, each message in the cycle depends upon itself, creating a situation of non-deterministic inputs to each PP in the cycle.

The property of systems which precludes the previous situation is called *predictability*. Predictability ensures that for each cycle in the system of PP's, there is at least one PP_i such that $A_i(\Psi_i(t)) > 0$, $0 < t < Z$. Chandy and Misra claim that all physical systems have the property of predictability [CM79]. It is assumed in this paper that all physical systems that may be simulated possess the property of predictability.

3.2 The Logical System

A system of N PP's can be simulated by constructing a system of M Logical Processes (or LP's), with $M \leq N$, where each LP simulates a disjoint set of one or more PP's, called a ***sub-model***, and each sub-model contains at least one PP. LP_i simulates the events of its sub-model S_i in chronological order using an event list data structure and associated operations, similar to those used in sequential simulation, to ensure the correct simulation order of events over the simulation time interval $(0, Z)$.

Message events transmitted between physical processes in different sub-models are simulated by messages sent between LP's. LP_i and LP_j communicate in the logical system if and only if some $PP_m \in S_i$ communicates with some $PP_n \in S_j$ in the physical system. In this case a *message channel* $[i, j]$ exists between LP_i and LP_j.

3.2.1 The Logical Process The state of sub-model S_i as simulated within LP_i is described by the set of variables $STATE_i$, which is $\cup\{\Psi_n \mid \forall PP_n \in S_i\}$, whose structure is simulation dependent and not relevant to the operation of the distributed algorithm. The value of any Ψ_n within $STATE_i$ is only changable by the operation $EVENT_i$, declared as $EVENT_i(event, eventlist, state) \longrightarrow state, eventlist$. $EVENT_i$ encapsulates the external and internal state transition and message-generating functions of every PP within S_i. The effect of $EVENT_i(STATE_i, e)$ on the value of $STATE_i$ for any event e is assumed to properly simulate the effect of event e occurring at its physical process.

The *event notice* is the primary data item of interest in discrete-event simulation. An event notice contains information that determines a possible change in the state of the simulation.

A data item of type event_notice can be described as follows:

```
type event_notice is record of
  t : time; simulation time of event
  e : event_data;
end record;
```

where, for event E, E.t is the invariant simulation time of the event, and E.e, of type event_data, includes the information which determines the manner in which the state of the simulation will change when some state transition function, also determined by the event_data e, is applied.

3.2.1.1 The LP Event List Event messages sent to any PP within S_i are placed, in order of t value, in the LP_i's event list, $EVNTLIST_i$. Within the upper bound on T_i set by the synchronziation constraint $LOOK_i$, discussed below, the simulation time at LP_i, T_i, is advanced to the time of the first event in $EVNTLIST_i$, and the event is removed from the event list and simulated. The event list, then, is a priority queue ordered by event time. Events may be ordered within time of occurrence by some secondary ordering function. The simulation of each event may cause future events to be scheduled on the event list, and the transmission of event messages to other LP's. Because an event list orders events chronologically, events may be simulated in proper order by simply removing the first event from the list and simulating its effects. Contingent events may be pre–empted (cancelled), by searching the event list sequentially for the event, which is removed from the list if found.

The event_list is defined as follows:

```
structure event_list is
  INSERT(event,event_list) ⟶ event_list;
  CANCEL(event,event_list) ⟶ event_list;
  FINDNEXT(event_list) ⟶ event;
  GETNEXT(event_list) ⟶ event, event_list;
  LENGTH(event_list) ⟶ integer;
end structure;
```

where INSERT(E,L) places event E in event_list L in increasing
    order of event time and returns the modified event_list L.
  CANCEL(E,L) searches event_list L to find event E, and deletes E from L
    if found
  FINDNEXT(L) returns the value of the first event in
    event_list L, without modifying L.

GETNEXT(L) removes the first event from event_list L and returns its value, and the modified event_list L.
LENGTH(L) returns the integer value equal to the number of events in event_list L.

Methods of implementing the event_list structure are not elaborated upon here. An ordered linear list is the most common "traditional" implementation, providing event insertion and cancellation in $O(n)$ time and next event retrieval in constant time. An overview of some newer methods, with empirical comparisons, is found in [Jon86b]. A discussion of the performance effects resulting from the choice of event list implementation is found in Chapter 4.

3.2.1.2 LP Time Advance Mechanism To ensure chronological ordering of all events within S_i, event messages received at LP_i from other logical processes over every message channel $[k, i]$ need to be considered. It will be shown that it is possible to calculate a lower bound on the simulation time of the next message that will be received at LP_i over any message channel $[k, i]$. This value, $LOOK_i$, can be derived from information transmitted from each LP_k.

A variable T_{ki} for each message channel $[k, i]$ is declared, such that T_{ki} equals $E_{ki}.t$, where E_{ki} is the last message to have been transmitted over $[k, i]$. T_{ki} is the *channel clock* value for $[k, i]$ [CM79]. Recall that the t values of messages transmitted over arc (i, j) in the physical system are monotonic increasing and bounded by Z [CM79]. Because LP_k in the logical system may send messages originating from multiple PP's in S_k to S_i at LP_i, the t values of messages sent over $[k, i]$ in the logical system are not necessarily monotonic, but are guaranteed to be non-decreasing, by the chronological ordering of $EVNTLIST_k$. Given this, if T_{ki} is the t value of the last tuple transmitted over channel $[k, i]$ at any point during the simulation, then all messages from LP_k to LP_i have been sent up to time T_{ki}. LP_i "knows" all messages it has received up to time $LOOK_i$, where $LOOK_i = minimum\{T_{ki}, \mid \forall [k, i]\}$. LP_i can then simulate events and calculate output event messages up to and including $LOOK_i$.

It would be ideal to delay the simulation of events occurring at time $LOOK_i$ until all possible event messages occurring at this time have been read by LP_i. Doing so would ascertain the proper order of execution among simultaneous message events arriving at LP_i. However, constraining the advance of $LOOK_i$ in this way may lead to deadlock of the physical system in certain configurations of LP's, as shall be described. As a result, the order of simulation of simultaneous event messages that arrive at a LP is determined by their order of arrival in the logical system, and can not be specified *a priori*. This property of the logical system may limit the applicability of the distributed event list algorithm, unless a corrective modification to the algorithm is found. None is apparent.

3.2.1.3 Predicting Message Transmissions It is possible to calculate a lower bound on the time of the next message to be transmitted over any outgoing message channel $[i, j]$ of LP_i. As described above, a trivial lower bound may always be derived from T_{ij}, due to the property of monotonicity. A tighter bound may be computable from information in $STATE_i$, given knowledge of S_i. When this prediction is possible, the lower bound on the time of the next message over any $[i,j]$, $NEXTOUT_i$, may be calculated and appended to each outgoing message tuple. By computing $LOOK_j = min\{NEXTOUT_i \,|\, \forall[i,j]\}$, LP_j may then be able to advance T_j sooner, since $NEXTOUT_i \geq T_j$ holds.

In computing $NEXTOUT_i$, two possibilities must be considered: 1) Some event that is presently pending in $EVNTLIST_i$ may, when simulated, prompt a message over $[i,j]$. Given the time–ordering of $EVNTLIST_i$, the time of the first event in $EVNTLIST_i$ provides a lower bound on the time of such an occurrence. The value of this lower bound, $MINEVNT_i$, can be applied to every channel $[i,j]$. 2) A message event yet to arrive at LP_i may prompt a message over some $[i,j]$. To calculate a lower bound $MINABS_i$ on the time of this occurrence, information about S_i is required. The earliest that an incoming message can arrive (if the logical system is correct) is the current simulation time T_i. To compute $MINABS_i$, T_i is added to the minimum elapsed simulation time between an incoming message event M_{ki} over any $[k,i]$ and the time of a resultant message M_{ij} over any $[i,j]$, given no other messages received or events pending. This value is denoted as $MINADV_i$. $MINADV_i$ is calculable by knowledge of the range of the function A_k for each PP_k within S_i and the topology of communication of all PP's within S_i. The value of $MINADV_i$ is invariant throughout the simulation, due to the invariance of the physical system structure, and therefore need only be calculated once, *a priori*.

Calculation of $NEXTOUT_i$ is accomplished by taking the minimum of $MINABS_i$ and $MINEVNT_i$. The algorithm for the calculation of $NEXTOUT_{ij}$ is then given as:

Algorithm 3–A:
{Lower Bound on time of next message over any $[i,j]$}

```
declare $MINABS_i$ : time; {bound on time of the next
                     message output over any $[i,j]$ due to
                     an incoming message}
        $MINADV_i$ : time; {bound on the time until the next
                     message output, given
                     message received at time Ti}
        $MINEVNT_i$ : time; {bound on time of next message
                     output by due to event on
                     the LP event list}
begin

  $MINABS_i := T_i + MINADV_i$;
  if LENGTH($EVNTLIST_i$) = 0 then
    $NEXTOUT_i := MINABS_i$;
  else
    MINEVNTi := FINDNEXT($EVNTLIST_i$).t;
    $NEXTOUT_i := \min(MINABS_i, MINEVNT_i)$;
  endif:
end;
```

3.2.1.4 Logical System Predictability The property of predictability in physical systems has been discussed previously. Were it not for predictability, systems of PP's connected in a cyclic topology could not be simulated [Bry79, CM79], and it was assumed in Section 3.1.5 that all simulated systems are predictable. Un-

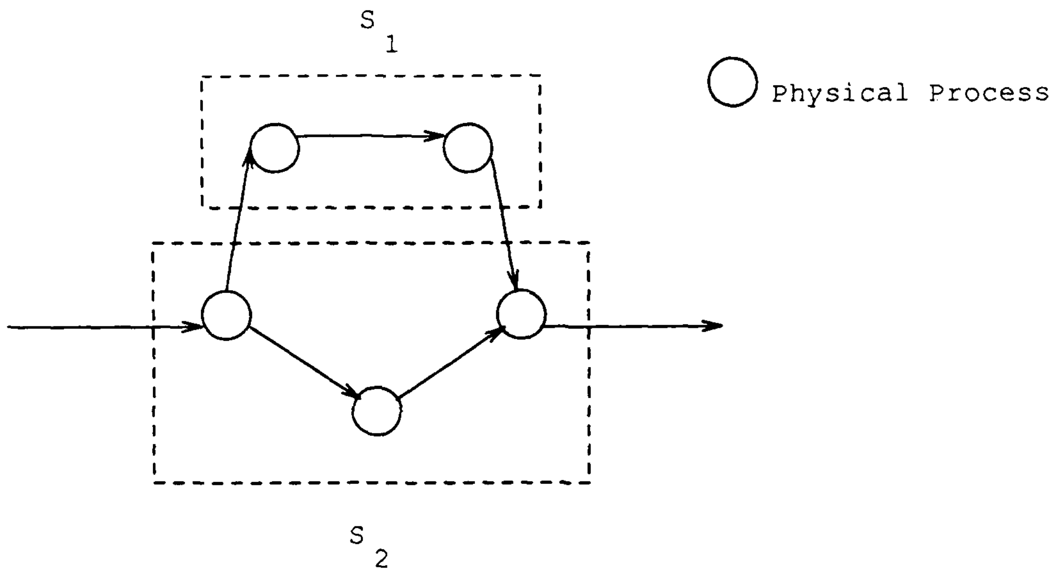


Figure 3.1. Non-Cyclic System of PP's with Cyclic Mapping

fortunately, the physical system's predictability does not guarantee the predictability of the simulating logical system in the distributed event list algorithm, since a non-cyclic set of PP's may be mapped into a cyclic set of sub–models (LP's) as depicted in Figure 3.1. In cases such as this, it is necessary to ensure that each cycle of LP's is predictable. This can be accomplished by placing constraints on the mapping of PP's onto LP's.

Previously, the only constraint on assignment of PP's to LP's has been that there be at least one PP simulated per LP. To ensure predictability of the logical system, the following condition is required;

Logical System Predictability Condition: Suppose a cycle of LP's exists, such that for LP_i, $i = 1, \cdots, n$, message channels $[i, (i+1) mod\, n]$, $i = 1, \cdots, n$ exist. For every such cycle in the logical system, there must exist at least one LP_j, such that $MINADV_j > 0$.

3.2.2 Logical Process I/O The message_tuple is the data item used to transmit event messages and the value of $NEXTOUT_i$ over $[i, j]$ in the logical system, and is defined as follows:

```
type message_tuple is record of
  m : message;
  next : time: {lower bound on simulation time of next
                message over channel}
end record;

where
  type message is
    subtype event_notice;
    subtype no_event;
  end type;
```

The subtype no_event contains only a single possible value, NULL, signifying that no event is being transmitted in the message_tuple.

The utility of a no_event message will become apparent in the following sections. The value of $M.next$ for a message_tuple M sent over channel $[k, i]$ is $NEXTOUT_k$.

3.2.2.1 Message Input A message is read into LP_i from an incoming message channel $[k, i]$ by executing the function $READ_i(k)$ at LP_i. The boolean function $MESSPENDING_i(k)$ returns TRUE if a message on channel $[k, i]$ is available to be read by LP_i, and FALSE otherwise. An event message read into LP_i is inserted immediately, in proper chronological order, into $EVNTLIST_i$ (NULL messages, to be discussed later, are read and disgarded).

If LP_i is allowed to read all event messages pending from each incoming channel at any point in the simuation, the number of messages read into LP_i at that point can not be predicted, because of the asynchronous operation of the LP's. The length of $EVNTLIST_i$ may then grow without any known upper bound, the only upper bound on the number of incoming messages being the total number of messages received by LP_i during the entire simulation. To ensure that the amount of memory needed to store incoming event messages in $EVNTLIST_i$ is known *a priori*, a mechanism for controlling the flow of messages into each LP is required.

The goal is to constrain the reading of messages into LP_i to a known level while maintaining the advance of simulation time at LP_i. This is accomplished by permitting LP_i to read messages only from the set of input channels $[\acute{k}, i]$ such that $NEXT_{\acute{k}i} = LOOK_i$ for all $\acute{k}$. Recall that $LOOK_i = min\{NEXT_{ki}\, \forall k\}$, so that $\{[\acute{k}, i] \mid \forall k\}$ is the set of channels whose NEXT values are constraining the advance of $LOOK_i$, and hence the simulation time T_i. LP_i reads pending messages, if any exist, from each channel in $[\acute{k}, i]$ until $NEXT_{\acute{k}i} > LOOK_i$. This algorithm is shown below in Algorithm 3–B, which is iterated until $LOOK_i$ is advanced. The number of messages read from a single channel is bounded by the number of messages that can possibly be sent to an LP at any single instant in simulation time, plus one. This value can be calculated from the interconnection structure of the physical system.

Algorithm 3–B:

```
declare for all $[k,i]$,
        $NEXT_{ki}$ : time {lower bound on time of next message on channel $[k,i]$}
        $M_{ki}$ : message_tuple {message retrieved from channel$[k,i]$}
    endfor;

begin

    for all $[k,i]$,
      while $MESSPENDING_i(k)$ and $NEXT_{ki} = LOOK_i$ loop
        $M_{ki} := READ_i(k)$;
        $NEXT_{ki} := M_{ki}.next$;
        if $M_{ki}.m \neq NULL$ then
          $INSERT(M_{ki}.m, EVNTLIST_i)$;
        endif;
      end loop;
    endfor;

end;
```

3.2.2.2 Message Output A message M is sent from LP_i over $[i,j]$ by performing the procedure $SEND_i(M,j)$. Event messages from LP_i may be generated when an event is simulated at LP_i. Messages are created and sent within the $EVENT_i$ procedure. Note that event messages transmitted from LP_i must have a time value t equal to that of the event that prompted the message — contingent messages are not allowed. This is in accordance with the basic Chandy-Misra algorithm and the physical system paradigm described above [CM79]. Operation $SEND_i(M,j)$ places the message M in a message tuple for transmission on channel $[i,j]$, consisting of the message itself and the value of $NEXT_{ij}$, computed using Algorithm 3–A above.

3.2.3 Properties of LP Communication A communications protocol for message-passing isn't explicitly specified in [CM79], but some assumptions are made concerning message transmission in the logical system. Messages are assumed to be transmitted correctly, and messages between any two LP's are assumed to be received in the order transmitted, within a finite time period. The algorithms discussed in this paper also operate under these basic assumptions [CM79].

The above assumptions imply some requirements for the communications protocol. In any implementation of the logical system, there is some finite amount of memory available to buffer incoming messages at each LP. To ensure the correctness of message transmissions, then, the message communication protocol must ensure that a message over channel $[i,j]$ will not be sent until LP_j has sufficient buffer space available to accommodate it. In that case, LP_i may queue outgoing messages into an output buffer of some finite size. If LP_i attempts to send a message to LP_j whose receive buffer for $[i,j]$ is full, and LP_i's send buffer for $[i,j]$ is full, LP_i can not proceed and is blocked. The ramifications of this blocking send protocol will be discussed in the following section on the resolution of process deadlock.

3.2.4 Summary of Algorithm The basic distributed event list algorithm at each LP consists primarily of a sequence of two phases, executed iteratively until simulation termination conditions have been reached:

Algorithm 3–C:

```
declare Ti : integer; {simulation time at LP i}
        for all [k,i], $NEXT_{ki}$ : time; {channel clock}
        $LOOK_i$ : time; {moving upper bound on time advance}
        $NEV_i$ : event_notice; {next event to be simulated at LP i}
        $EVNTLIST_i$ : event_list; {event list for LP i}
        $STATE_i$ : state; {set of state variables for $S_i$}
begin
  {Initialize}
  $T_i$, $LOOK_i$ := 0: $NEV := (Z, NULL)$;
  for all $[k,i]$, $NEXT_{ki}$ := 0: endfor;
  {Initialize $STATE_i$: Schedule initial events, if any}

  loop until {simulation termination condition reached}

  {Read Phase: Attempt to read messages to advance $LOOK_i$}
    until $LOOK_i > T_i$ loop
      Read messages on input channels [k,i]: {Algorithm 3–B}
      $LOOK_i$ := minimum($NEXT_{ki}$);
    end loop;

  {Event Phase: Simulate all pending events up to $LOOK_i$}
    while $LENGTH(EVNTLIST_i) > 0$ and $NEV.t \leq LOOK_i$ loop
      NEV := GETNEXT($EVNTLIST_i$): {Get next event from list}
      $T_i$ := NEV.t;
      {Simulate next event}
      $EVENT_i(NEV, EVNTLIST_i, STATE_i)$;
    end loop;

    {if no events scheduled in interval $T_i$ to $LOOK_i$, advance $T_i$}
    if $LOOK_i > T_i$ then
      $T_i := LOOK_i$;
    endif;

  end loop;
end;
```

3.2.5 Deadlock in the Logical System The basic logical system as described above is subject to the problem of logical process deadlock, as described in [CM81]. Deadlocks may occur within both cyclic and acyclic networks of LP's. A method for avoiding deadlock, the Null Message algorithm proposed by Chandy and Misra in [CM79], has been modified for use within the distributed event list algorithm, and is presented in this section.

3.2.5.1 Cyclic Deadlock Among LP's A *cyclic deadlock* may occur within a set of LP's $\{LP_k \mid k = 1, \cdots, K\}$ such that message channel $[k, (k+1) mod\, K]$, $k = 1, \cdots, K$ exists in the logical system. Such a deadlock occurs when each LP_k in the set is attempting to read from channel $[(k-1) mod\, K), k]$, and no message is in transit over $[(k-1) mod\, K, k]$, $k = 1, \cdots, K$ [CM79]. In Example 3–1 below, the set of LP's is deadlocked, and will not advance in simulation time. LP's outside of the deadlocked set that depend on event messages from any LP in this set will then eventually become blocked due to message "starvation," and so will their dependent LP's, and so on, until the entire logical system is halted.

In Example 3–1, we show that the system in Figure 3.2 (b), the set of $\{LP_i, \ LP_2\}$ is deadlocked, and will not progress in simulation time. A message in the logical system is denoted by (t,m,next).

Example 3–1 In Figure 3.2 (a), LP1 has previously sent event message (1,m12,2) to LP2. LP2 read this message, inserted (1,m12) into EVNTLIST1, advanced LOOK2 = 2, and then ended its Read Phase. LP2 simulated (1,m12), causing an internal event (2,ev2), which prompted message (2,m21). (2,m21,3) was then transmitted to LP1. At this point the next internal event scheduled at LP2 was (8,ev2). LP2, with LOOK2 = T2 = 2, could simulate ne further, and so entered a Read Phase. LP1 has read message (5,m01,6) from [0,1] in a previous Read Phase, which caused (5,m01) to be placed as the next event in EVNTLIST1. LP1 could not advance its clock to 5, being constrained by NEXT21 = 1. LP1 then entered another Read Phase, attempting to read from [2,1]. (2,m21,3) arrived, and (2,m21) was read into EVNTLIST1 as the new next event. LOOK1 was advanced to 3, and LP1 left its Read Phase. This is the status of the set of LP's as shown in Figure 3.2 (a).

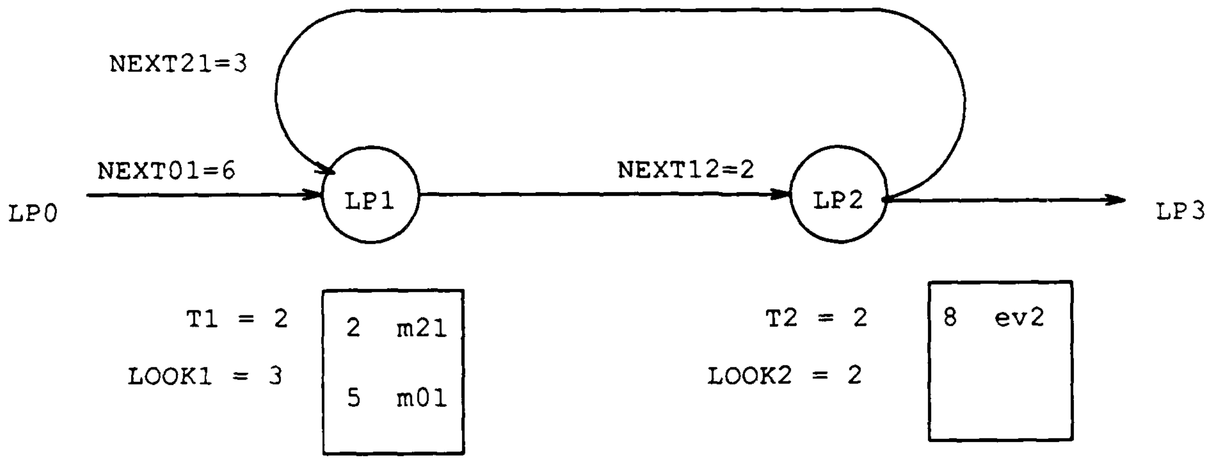
LP1 now enters its Event Phase, and with LOOK1 = 3, is able to remove (2,m21) from EVNTLIST1 and simulate it. As a result, an internal event is scheduled at time 3, which is then also simulated. The simulation of this internal event prompts event (3,m12), which is sent over [1,2] in message tuple (3,m12,4). With its next scheduled event at time 5, LP1 can simulate no more, and so enters its next Read Phase. LP2, in a Read Phase as we recall, reads (3,m12,4), advances LOOK2 = 4, and exits its Read Phase. LP2 then simulates (3,m12), causing an internal event at time 4, which prompts an event message (4,m23,5) to LP3 (outside of the cycle). With its next internal event at time 8, LP2 can not simulate further, and enters a Read Phase once again.

The cycle is now in a deadlocked state, as shown in Figure 3.1 (b). LP1 will continue to attempt to read messages from [2,1] and no other channel, since NEXT21 = 3 is constraining LOOK1 and NEXT01 = 6 is not. LP2 is in a Read Phase, attempting to read messages over [1,2] until LOOK2 is advanced. LP2 will not leave its Read Phase until it receives a message over [1,2], and LP1, as we recall, can not send any messages until it reads a message over [2,1].

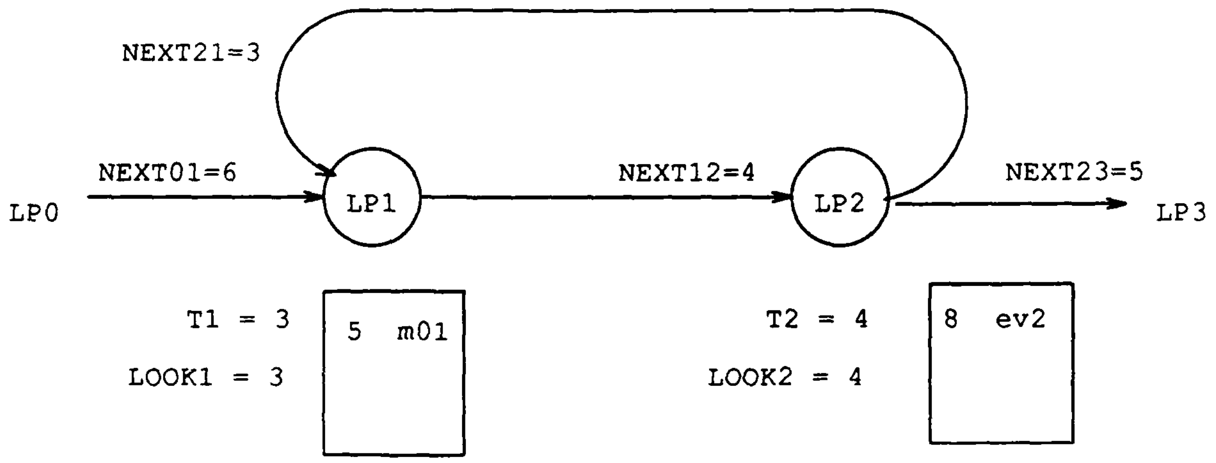
In the above example, each LP in the cycle is each waiting to receive a message exclusively from the previous LP. No message from outside the cycle of LP's is sufficient to change this condition. Therefore, the deadlock condition is permanent.

3.2.5.2 Acyclic Deadlock Among LP's Deadlock may also occur in acyclic networks of LP's under certain conditions. A "blocking send" implementation with bounded buffers as described in Section 3.2.3 can result in *acyclic deadlock* when a set of LP's are connected in a "fork–join" configuration, as in Figure 3.3 below.

In any implementation of the logical system, each LP has a finite buffer size for sending and receiving messages on each channel over which it communicates. Buffer



(a) Cyclic Set of LP's Prior to Deadlock



(b) Cyclic Set of LP's in Deadlock State

Figure 3.2. Set of Logical Processes Subject to Cyclic Deadlock

sizes are purely an implementation constraint of the logical system. *Acyclic deadlock* is a possibility when a LP may receive messages over more than one channel, but does not receive messages over all channels at equivalent rates. If LP A is waiting to read over a subset of its input channels, the other channel buffers may become filled, causing a sending LP B to become blocked. If another LP C sends sufficiently many messages to B, it will eventually become blocked, because LP B will not read, being blocked. If the channels on which LP A is waiting are dependent on messages sent by LP C, then LP A will never receive messages over that channel, since LP C is blocked, and LP A will continue to attempt to read [CM79].

In Example 3–2 below, we show that the system in Figure 3.3 (b) is deadlocked, and will not progress in simulation time. We assume that each message channel in the logical system has buffer size sufficient to hold a single message.

Example 3–2 In Figure 3.3 (a), the set of $\{LP_j,\ j = 1, \cdots, 4\}$ is not yet in a deadlocked state. We observe that LP1 had previously sent message (5,m12,6) and has received message (6,m01,7) from LP0. LP2 had previously sent (4,m23,5) to LP3. In a subsequent Read Phase, LP2 read message (5,m12,6), placed (5,m12) in EVNTLIST2, advanced LOOK2 to 6, and then ended its Read Phase. LP3, in a Read Phase with NEXT13 = NEXT13 = 3, was attempting to read over both [1,3] and [2,3]. LP3 read (4,m23,5), placing (4,m23) in EVNTLIST3. But because NEXT13 = 3, LOOK3 remained at 3, and LP3 remained in a Read Phase, now only attempting to read from [1,3], in order to advance LOOK3. This is the status of the set of LP's shown in Figure 3.3 (a).

LP2 has ended its Read Phase, and now enters the subsequent Event Phase. LOOK2 = 6, and so LP2 simulates (5,m12). As a result, an internal event is scheduled for time 6. This event is still within the "safe" simulation period, and so is simulated. The internal event prompts a message (6,m23), which is sent over [2,3] in message tuple (6,m23,7). With no more pending events before or at time 6, LP2 then enters a Read Phase. LP3, as we recall, is reading, but only over [1,3], so that message (6,m23,7) occupies buffer[2,3]. Meanwhile, LP1 has simulated (6,m01), causing an internal event at time 7. With LOOK1 = 7, this event is simulated, which prompts message (7,m12,8). LP1 then enters another Read Phase.

LP2, in its Read Phase, reads (7,m12,8). LOOK2 is advanced to 8, and LP2 begins an Event Phase. Simulating (7,m12) causes internal event (8,ev2), which prompts message (8,m23). LP2 then attempts to send (8,m23,9), but is blocked, because message (6,m23,7) still occupies buffer[2,3]. LP2 will remain blocked until LP3 reads (6,m23,7). LP1, in a Read Phase, reads (8,m01,9). A subsequent internal event at time 9 prompts message (9,m12,10) to LP2. But because it is blocked, LP2 can not read over [1,2], and (9,m12,10) occupies buffer[1,2].

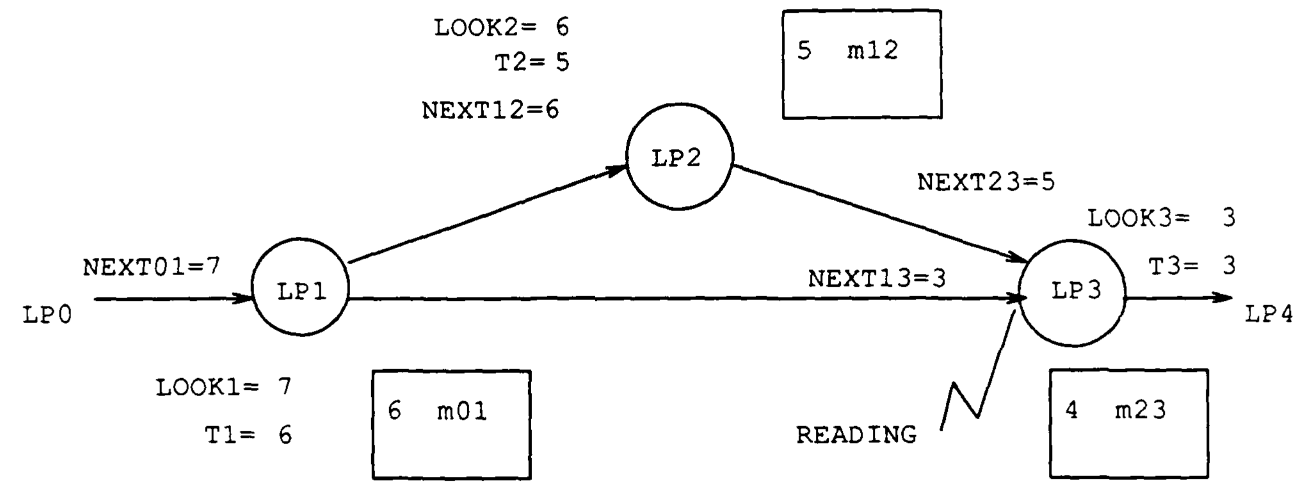
LP1, able to simulate no further, enters a Read Phase. In this phase, LP1 reads (9,m01,10), and consequently exits the Read Phase. Simulating (9,m01) cuases an internal event at time 10. This event is simulated, prompting message (10,m12,11), and a new next internal event at time 14.

Once LP1 tries to send message (10,m12,11) to LP2, the conditions required for deadlock are complete, as shown in Figure 3.3 (b). LP1 attempts to send the message over channel [1,2], but buffer[1,2] is full, so LP1 is blocked. Buffer[1,2] may only become unblocked if LP2 reads its contents, message (9,m12,10). LP2 is blocked sending over [2,3], and will only become unblocked if buffer[2,3] becomes free. For that to occcur LP3 would have to read over [2,3]. LP3 will not read over [2,3] unless it first reads over [1,3]. LP3 can not read a message over [1,3] until LP1 sends one. LP1 can not send a message over [1,3] because LP1 is blocked: hence, the deadlock cycle is complete.

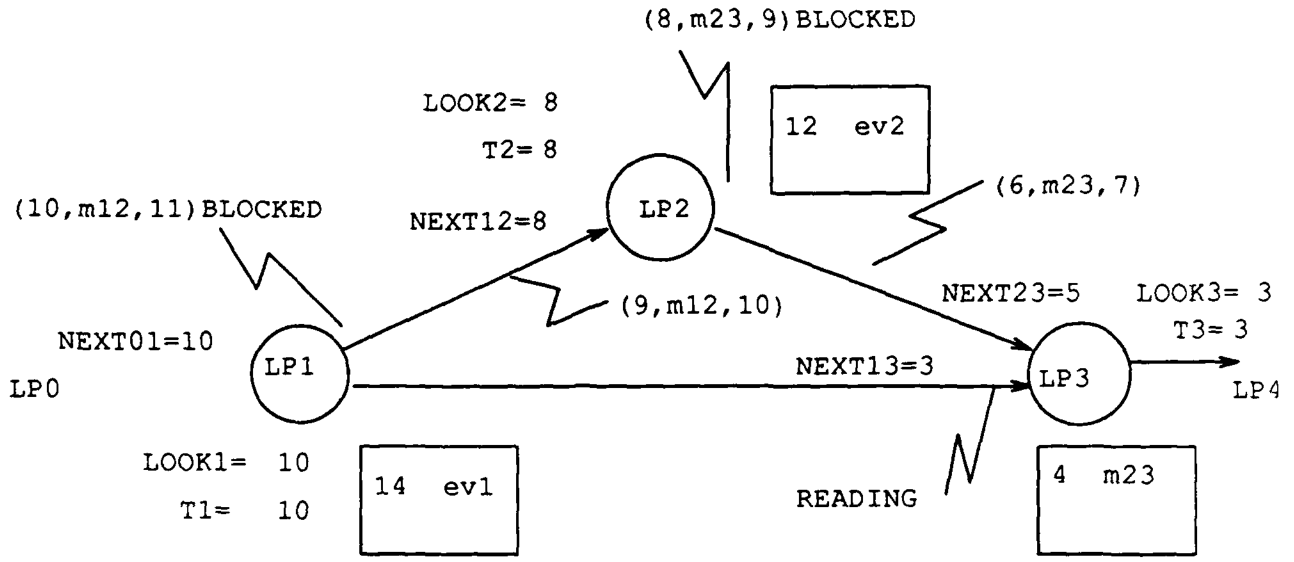
No message from any LP outside of the deadlocked set can affect the condition of the deadlock set. The deadlock condition is then permanent and the set of LP's described above will not progress in simulation time, leading to non–termination of the logical system.

3.2.5.3 Null Message Deadlock Avoidance The method of Null messages is used to avoid the process deadlock that is inherent to the basic system [CM79]. A Null message is a message transmitted with the singular purpose of advancing the channel clock of the channel on which it is sent, thus possibly allowing the recipient LP to advance its simulation clock. The contents of the message, the no–event symbol NULL, do not affect the state of simulation at the sending and receiving LP's.

The following condition is sufficient to prevent cyclic deadlock in the logical system:



(a) Set of LP's Prior to Deadlock



(b) Set of LP's in Deadlock State

Figure 3.3. Set of Logical Processes Subject to Acyclic Deadlock

Null Message Condition 1: For every $LP_i, i = 1, \cdots, N$: once LP_i exits a Read Phase, LP_i sends at least one message, either an event message or a Null message, over every output channel $[i, j]$ before entering another Read Phase.

The validity of the assertion that this condition avoids deadlock can be seen intuitively from Example 3–1 above. It can be proven inductively for any valid cyclic set of LP's by showing that it avoids the circular read condition of cyclic deadlock.

The following condition is sufficient to prevent acyclic deadlock in the logical system:

Null Message Condition 2: For every LP_i, $i = 1, \cdots, N$: once LP_i sends an event message (t, m, $next$) over message channel $[i, j]$, then LP_i has sent a message, either an event message or a Null message, such that $next_{im} \geq LAST_{ij}$ for all m, over every other output channel $[i, m]$ before attempting to send another message over $[i, j]$.

The validity of the assertion that this condition avoids deadlock can be seen intuitively from Example 3–2 above. It can be proven inductively for any valid set of LP's by showing that it avoids the read—write—read condition of acyclic deadlock.

It is not efficient to send a Null message over $[i, j]$ that has no effect on the value of $NEXTOUT_i$ at LP_j. We preclude this by maintaining at each LP_i a variable for each outgoing message channel $[i, j]$, $LAST_{ij}$, set to the value of $NEXTOUT_i$ of the message, Null or event, that was last transmitted over $[i, j]$. A Null message is only transmitted over $[i, j]$ if the computed value of $NEXTOUT_i$ to be inserted into the message is strictly greater than $LAST_{ij}$.

Note that the added constraint of $LAST_{ij}$ on the sending of a Null Message over $[i, j]$ does not violate the Null Message Conditions. In cases where $NEXTOUT_i = LAST_{ij}$ and the sending of a Null message is inhibited, then, by definition of $LAST_{ij}$, a message M_{ij} has been sent over $[i, j]$, such that $M_{ij}.next = NEXTOUT_i$. By Algorithm 3–A, either the value of T_i and the time of the next event in $EVNTLIST_i$, $MINEVNT_i$, are at present identical to those at the time M_{ij} was sent over $[i, j]$, or the value of $MINEVNT_i$ is the same and equal to $LOOK_i$ at the time M_{ki} was sent. If a Read Phase had occurred in the intervening time since the last message send, $LOOK_i$ would have advanced (Algorithm 3–C), precluding the latter condition. The value of T_i would have increased with the value of $LOOK_i$, due to the sending of M_{ij} (Algorithm 3–C), precluding the former condition. No Read Phase has then occurred since the time of the last message over $[i, j]$. A Null message is not then necessary, and Null Message Condition 1 is preserved. Null Message Condition 2 is preserved because a Null message is withheld only if some message has been sent over $[i, j]$ with next message component $next_{ij} = NEXTOUT_i$. $NEXTOUT_i \geq LAST_{im}$ for any [i,m] due to monotonicity of T_i. Then, if a Null Message is withheld over $[i, j]$, it is guaranteed that $LAST_{ij} \geq LAST_{im}$, for all m, and Null Message Condition 2 is preserved.

We can now describe an algorithm for procedure $SENDNULL_i(j)$ to send a Null message over output channel $[i,j]$ such that the Null Message Conditions are fulfilled:

Algorithm 3–D:

```
begin
  Compute $NEXTOUT_i$: {Algorithm 3–A}
    if $NEXTOUT_i > LAST_{ij}$ then
      $SEND_i((NULL, NEXTOUT_i), j)$;
      $LAST_{ij} := NEXTOUT_i$;
    endif;
  end;
```

The Null Message Conditions can be implemented at LP_i with the following straightforward algorithm:

Algorithm 3–E (1);

```
begin
  {Initialize}
  $T_i$, $LOOK_i := 0$: $NEV := (Z, NULL)$;
  for all $[k,i]$, $NEXT_{ki} := 0$: endfor;

  {Initialize $STATE_i$: Schedule initial events, if any}

  loop until {simulation termination condition reached}

    for all $[i,j]$ which did not send messages
      during previous Event Phase,
      $SENDNULL_i(j)$: {Algorithm 3–D}
    endfor;

    Perform Read Phase;
    {Attempt to read messages until advance $LOOK_i$}

    Perform Event Phase;
    {Simulate pending events up to $LOOK_i$}
```

```
      {if no events scheduled in interval $T_i$ to $LOOK_i$, advance $T_i$}
      if $LOOK_i > T_i$ then
        $T_i := LOOK_i$;
      endif;

    end loop;

end;
```

It is possible to take advantage of some of the properties of Null messages to modify the Read Phase of the LP for greater efficiency. Unlike an event message. a Null message received at LP_i over $[k, i]$ has no message content to store after its message time has been used to update $NEXT_{ki}$. It is therefore possible to read any number of Null messages over an input channel without exceeding storage bounds. Because the time associated with each successive Null message is guaranteed to be increasing (by Algorithm 3–D), only the last Null message in a sequence of Null messages in an input buffer is of interest to us. The situation may arise in which sequences of Null messages in an input buffer cause many small increments of simulation time advance. Since each time advance at LP_i leads to a Null message sent over every channel $[i, j]$, this "thrashing" from Read Phase to Event Phase intensifies the thrashing effect in recipient LP's, increasingly fragmenting the simulation time advance and clogging system buffers with superfluous Null messages. Thrashing is alleviated by modifying Algorithm 3–B so that a Null message received over $[k, i]$ will not cause reading over $[k, i]$ to end unless no more messages are pending over $[k, i]$.

3.2.6 Null Message Variants Null Messages are overhead in the logical system, contributing nothing to the actual execution of the simulation model. In order to maximize throughput, then, it might seem wise to restrict the conditions for the transmission of Null messages to the minimum required to avoid deadlock. This, it shall be seen, is often not the case.

Null messages sent over $[i,j]$ perform their function of deadlock avoidance by incrementing the value of channel clock $NEXTOUT_i$ at LP_j. If $NEXTOUT_i = LOOK_j$ before a Null message is sent over $[i,j]$, then the transmission of the Null message may allow LP_j to advance $LOOK_j$ sooner than it ordinarily could. This may be particularly beneficial if a large percentage of events in the logical system are internal to some LP, with infrequent messages between LP's.

The following algorithms are variants on the basic algorithm that change the conditions under which Null messages are sent, for the purpose of increasing the throughput of the logical system. A performance analysis and empirical comparison of these methods versus the "basic" method described above can be found in Chapter 4.

3.2.6.1 Null Message Algorithm with Stimulus Nulls Prof B. Donlan at the Air Force Institute of Technology has devised a method of sending additional Null messages purely to improve Speed-up. The extraneous Null messages, known as *stimulus* Nulls for their effect on throughput, are transmitted after the execution of a certain number of events, specified as a ratio of events to stimulus Nulls.

Algorithm 3–E (2);

```
declare EVNTCNT : integer {Number of events since last transmission of stimulus Nulls}
        NULLRATIO : integer {Number of events executed before stimulus Nulls are sent}

begin

        {Initialize}
        EVNTCNT := 0:
        loop until {simulation termination condition reached}

    {Regular Null send required to fulfill Null Message Condition}
    for all $[i,j]$
      which did not send messages during previous Event Phase,
      $SENDNULL_i(j)$: {Algorithm 3–D}
    endfor;

    Perform Read Phase;
    {Attempt to read messages until able to advance $LOOK_i$}

        {Event Phase: Simulate pending events up to $LOOK_i$}
    while $LENGTH(EVNTLIST_i) > 0$ and $NEV.t \leq LOOK_i$ loop
      NEV := GETNEXT($EVNTLIST_i$): {Get next event from list}
      $T_i$ := NEV.t;
      {Simulate next event}
      $EVENT_i(NEV, EVNTLIST_i, STATE_i)$;

      {Send Stimulus Nulls}
      EVNTCNT := EVNTCNT + 1;
      if EVNTCNT ¿ NULLRATIO then
        for all $[i,j]$,
          $SENDNULL_i(j)$;
        endfor;
        EVNTCNT := 0;
      endif;
    end loop;

    {if no events scheduled in interval $T_i$ to $LOOK_i$, advance $T_i$}
    if $LOOK_i > T_i$ then
```

```
        $T_i := LOOK_i$;
      endif;

        end loop;

end;
```

Preliminary studies have shown that stimulus Nulls can provide significantly improved Speed-up for some systems of LP's. As the stimulus null ratio is increased beyond a certain point, the incremental increases in Speed-up diminish rapidly as the Null message overhead takes its toll. The reasons for this phenomenon are discussed fully in Chapter 4.

3.2.6.2 Null Message Algorithm with Timed Nulls Another variation on the basic Null message scheme is that of transmitting Null messages only after some amount of processing time has elapsed. This algorithm was proposed by Misra in [Mis86], and can be incorporated into the distributed event list algorithm.

To employ this algorithm variant it is necessary to modify Null Message Condition 1. Rather than specifying that a message be sent over every output channel between each Read Phase, the following condition can be specified:

Null Message Condition 1a For every $LP_i, i = 1, \cdots, N$: once LP_i begins a Read Phase, LP_i may send at least one message, either an event message or a Null message, over every output channel $[i,j]$ before entering another Read Phase, and LP_i will send a message over every output channel $[i,j]$ within a finite amount of time.

This condition is fulfilled in the following algorithm for transmission of time-driven Null messages:

```
Algorithm 3–E (3): declare $clock_i$: real_time: {processor clock time at $LP_i$}
                           $NULLTIME_i$ : real_time: {specified elapsed time between
                                                      Nulls over all $[i,j]$}
                           $TIMEOUT_i$ : real_time: {time-out value for Null send}

begin
  {Initialize}
    $TIMEOUT_i := clock_i + NULLTIME_i$;
  loop until {simulation termination condition reached}

  {Read Phase: Attempt to read messages to advance $LOOK_i$}
    until $LOOK_i > T_i$ loop
      Read messages on input channels [k,i]: {Algorithm 3–B}
      $LOOK_i :=$ minimum($NEXT_{ki}$);

      {Send Nulls over each channel $[i,j]$ if the Time–out for Null send has expired}
      if $clock_i > TIMEOUT_i$ then
        for all $[i,j]$,
          $SENDNULL_i(j)$;
        endfor;
        $TIMEOUT_i := clock_i + NULLTIME_i$;
      endif;

    end loop;

  Perform Event Phase: {Simulate pending events}
```

```
        {if no events scheduled in interval $T_i$ to $LOOK_i$,
          advance $T_i$}
        if $LOOK_i > T_i$ then
          $T_i := LOOK_i$;
        endif;

    end loop;

end;
```

Naturally, the value of $NULLTIME_i$ at each LP_i should effect the efficiency of this algorithm in a given situation. The performance effects of the choice of $NULLTIME_i$ are discussed in Chapter 4.

3.2.7 Bounds on Required Memory Each LP in the logical system requires a bounded amount of storage, and the sum of the storage required by all LP's is comparable to the storage required by an equivalent sequential simulation. This property of the logical system is one shared by the original Null Message algorithm [CM79].

The set of state variables for LP_i, $STATE_i$, represents the states of all PP's in S_i. In a logical system of N LP's, $\cup\{STATE_i, i = 1, \cdots, N\}$ represents the state of the logical system. The number of PP's is bounded, and the amount of information required to capture the state of any PP must be bounded. Otherwise, the physical system could not be simulated [CM79]. The storage required to maintain the states of all PP's in the distributed system is the same as that required in an equivalent sequential simulation.

A method of controlling message input into LP_i as a way of bounding the required size of the event list $EVNTLIST_i$ has been introduced above in Section 3.2.2.1. Given the properties of the logical system, it can be shown that the required size of $EVNTLIST_i$ is bounded by a predictable amount. It is assumed that every $PP_m \in S_i$, $m = 1, \cdots, M$ may have a pending internal event at any time. Thus the number of events in $EVNTLIST_i$ is bounded by M plus the maximum number of event messages that are pending in $EVNTLIST_i$ at any point during the simulation.

From Algorithm 3–C, we know that in any single Read Phase, messages are read until $LOOK_i$ can be advanced from time T_i, and that messages read over any channel $[k, i]$ during any Read Phase have the event time component $t = T_i$, except for the last message that is read over $[k, i]$ in any Read Phase, which may have time component $\acute{t}$, $T_i \geq \acute{t} \geq LOOK_i$. The maximum number of messages that can be sent at an instant in simulation time from a given PP to another is one, by monotonicity in the physical system. Because each event message represents a message transmitted in the physical system, the number of messages that can be read into LP_i during a Read Phase is bounded by the the sum of the number of PP's that can send messages to each PP within S_i, plus one additional message for each input channel $[k, i]$. This number is known from the configuration of the physical system, without examining any PP internally.

It follows from Algorithm 3-C that any message read into LP_i has a simulation time component $t \leq LOOK_i$, and thus will be simulated during the subsequent Event Phase. Given this, the number of event messages in $EVNTLIST_i$ from outside of LP_i is bounded by the the maximum number of event messages that can be read during a single Read Phase, as described above.

Event messages can also be sent between two PP's within LP_i during an Event Phase of processing. Because each event message has the same time component as the internal event that prompted it, all event messages internal to LP_i that are prompted in a particular Event Phase are also simulated in that same phase, and before any later event is simulated, due to the chronology of $EVNTLIST_i$. The maximum number of event messages internal to LP_i pending at any point in time is then the maximum number that can be prompted in a single instant in simulation time. As in the case of external event messages, this is equal to the number of PP's in S_i that send messages to each $PP_m \in S_i$, $m = 1, \cdots, M$, due to the monotonicity of messages in the physical system.

Because the order in which internal events and message events at an instant in simulated time at LP_i are simulated can't be predicted in advance, assume that all event messages at a point in simulated time are prompted before any are simulated (worst case for bounding $EVNTLIST_i$). The maximum number of event messages pending is the sum of the maximum number of event messages from within LP_i and those read in. Combining these two quantities with the maximum number of pending internal events at LP_i gives the upper bound on the number of messages in $EVNTLIST_i$ as:

$$\sum_{m=1}^{M} |IN_m| + K + M$$

where IN_m is the set of PP's that may send messages to $PP_m \in S_i$.

It is submitted that the size of each event is bounded, as an event reflects a single change in value over a number of state variables, and the number of state variables of each PP is bounded, for reasons described previously.

The storage overhead for the operation of each LP_i, consisting of the variables declared in the preceding algorithms, is also bounded. Each declared variable is either a scalar quantity, such as $LOOK_i$, or a one-dimensional array of scalars over each input channel $[k, i]$, such as $NEXT_{ki}$, or over each output channel $[i, j]$, such as $LAST_{ij}$. Given that the number of LP's is finite, each of these arrays is bounded. Hence, the storage overhead at each LP is bounded for a given system of LP's.

Given a blocking send communications protocol as described in this chapter, the amount of storage dedicated to message buffers at each LP may be arbitrarily chosen without affecting the correctness of the logical system. The chosen buffer size, however, will generally affect the execution time of the logical system.

IV. Performance Analysis

4.1 Performance Characteristics of the Event List

The event list is the primary data structure of the distributed event list algorithm. The efficiency of the implementation of the event list at each logical process will have a major effect on the execution time of the distributed event list simulation. Similarly, the event list implementation in the sequential simulation used for comparison will affect the computed speed-up. The distributed event list algorithm will be shown to take advantage of the time complexity of event list operations to provide a speed-up factor that, in some cases, exceeds what had been thought to be the theoretical bound on achievable speed-up.

The event list is a priority queue of event notices, ordered by event simulation time. The primary operations on the event list are next event retrieval and event notice insertion, although others, such as previewing the first event and deletion of an arbitrary event from the list, may be performed. Next event retrieval removes and returns the first event notice in the event list. Event notice insertion operates on a given event notice to insert it in its proper order in the event list.

4.1.1 The Linear List Implementation Several physical implementations of the event list have evolved. The "classical" implementation is an ordered linear list, which was used in early simulation languages, such as GASP [Pri74], and is still in fairly wide use today. Its major advantage is ease of implementation. The event list used in the implementation of the distributed event list algorithm and the comparison sequential simulations is a doubly-linked linear list implemented with FORTRAN arrays. Next event retrieval with the linear list involves simply removing the event notice at the head of the list, and is therefore done in constant time. Insertion of an arbitrary event notice into the event list requires that the items in the list be searched sequentially for the correct point of insertion. In asymptotic notation, event notice insertion with a linear list is therefore accomplished in $O(L)$ time, where L is the number of events in the event list [VD75].

The dependence of event list insertion time on the number of event notices in the list at the time of insertion is illustrated by the following timing experiment. Using a linear list implementation of the event list, event lists of size 10, 100, and 1000 were created, in which the simulation time increments of events in the list were exponentially distributed. In this experiment, 100 events (with the same distribution of inter-event times) were inserted into each event list; an event was removed from the front of the list prior to each insertion to maintain constant list size. The total time to accomplish the insertions only was measured for each event list size. The results of this experiment, shown in Table 4.1, demonstrate the correlation between event list size and list insertion time.

4.1.2 Theoretical Bounds on Speed-Up This dependency of insertion time on the number of event notices in the event list has important implications for the

Length of Event List	10	100	1000
Time to Insert 100 Events (s)	0.90	4.23	37.40

Table 4.1. Insertion into Linear List with Exponential Inter-Event Times

attainable speed-up factor using the distributed event list algorithm. It has been assumed in previous literature [BJ85, Hei86] that there is a bound on the speed-up factor of any distributed simulation equal to N, where N is the number of processors used. This is a theoretical bound, neglecting communications and synchronization overhead. and has not been thought to be attainable [BJ85].

The underlying assumption behind a speed-up bound of N is that the execution time T of a simulation depends upon the number of events, E, occurring in the simulation model, and processing time T_E per event, bounded by some constant. The execution time T of the sequential simulation is then $E\,T_E$. The theoretical minimum execution time for a distributed execution of the simulation model over processors P_n, $n = 1, \cdots, N$ is then equal to

$$MAX\{E_n,\ n = 1, \cdots, N\}\, T_E$$

where E_n is the number of events occurring in the simulation sub-model at processor n. This value is minimized for a given number of processors when

$E_1 = E_2 = \cdots = E_N = E/N$. The absolute minimum execution time for a distributed simulation over N processors occurs when events are evenly distributed, and is $(E/N)T_E$. Absolute minimum speed-up can be then computed as $(ET_E)/((E/N)T_E) = N$.

The assumption of a linear relationship between the number of events in a simulation and its execution time is easily shown to be unjustified by considering the simulation overhead in a sequential simulation algorithm. Super-linear time complexity of the sequential algorithm in terms of E will be demonstrated for a linear list implementation of the event list; similar results can be shown for other event list implementations, although the super-linearity of the more efficient implementations is not as dramatic.

A non-preemptive sequential simulation algorithm is considered, such that scheduled events are never deleted from the event list prior to their removal for execution. Deletion of an arbitrary event notice from the event list is of the same time complexity as an insertion, so that omitting the possibility of the former does not weaken the argument, as will become evident.

In the general sequential event list algorithm, the basic iteration, performed once for each of E events in the simulation, is as follows: 1) Remove the first event from the event list. 2) Calculate the effects of the event, including updating the state variables of the simulation, possibly calculating new events and inserting them in proper order in the event list. The execution time of the sequential simulation can be analyzed by considering the number of events, E, and the following periods of time :

$T_c(e)$ – time required to calculate a new event e

$T_r(e)$ – time required to remove event e from the event list

$T_u(e)$ – time required to update state variables due to event e

$T_i(e)$ – time required to insert event e into the event list

Due to the nature of the linear list, $T_r(e)$ is constant for all $e = 1, \cdots, E$, and so has $O(1)$ time complexity. Assume $T_u(e)$ and $T_c(e)$ are also $O(1)$ for all e, bounded by constant values. (Because the goal of the argument is to show super-linearity of the sequential algorithm to E, this is the worst case for the argument.)

Now consider $T_i(e)$, which with a linear event list is of $O(L_e)$ time complexity, where L_e is the number of events in the event list at the time of insertion of event e. One upper bound on L_e is E, but it can be seen intuitively that it is a trivial one – since each event must be inserted into the list, there is no event that is inserted when there are already E events in the event list. Similarly, there exists only one possible event e' such that $L_{e'} = E - 1$ at the time of insertion. Because the value of L_e is bounded by $L_{e-1} + 1$, a meaningful bound on insertion time can only be obtained by considering the set of event insertions in the simulation as a whole.

The total time to insert all of the events of the simulation into the event list is bounded by a function of $\sum_{e=1}^{E} L_e$. This value, the sum of the lengths of the event list at the time of each event's insertion, is maximized when the maximum number of events are scheduled before any are removed and simulated. In such a case, $L_1 = 0$, $L_2 = 1$, and, in general, $L_e = e - 1$. The total insertion time for all events, T_I, can then be expressed in asymptotic notation as follows:

$$\begin{aligned} T_I &= O(\textstyle\sum_{e=1}^{E-1} e) \\ &= O(((E-1)\,E)/2) \\ &= O((E^2 - E)/2) \\ &= O(E^2) \end{aligned}$$

The total execution time of the sequential algorithm, T, is:

$T \;=\; \sum_{e=1}^{E}(T_c(e) + T_r(e) + T_u(e)) \;+\; T_I$ *which can be expressed in asymptotic notation as:*

$$\begin{aligned} &= \textstyle\sum_{e=1}^{E}(O(1) + O(1) + O(1)) \;+\; T_I \\ &= E\,O(1) \;+\; O(E^2) \\ &= O(E) \;+\; O(E^2) \\ &= O(E^2) \end{aligned}$$

Theoretical speed-up of the distributed event list algorithm, with sequential and distributed algorithms using a linear list, and with the same idealistic assumptions made previously, is then: $O(E^2)/O(E^2/N^2) \;=\; N^2$

As before, this value is the theoretical best speed-up, and is not, in general. attainable, due to overhead and communications costs, and the fact that it is based upon the "worst case" time complexity of the sequential algorithm. Other event list implementations of lower time complexity will have correspondingly lower theoretical speed-ups - although real execution time will decrease in comparison to the linear list implementation as the average L increases. Attained speed-up will decrease with increased overhead for the distributed simulation, naturally, and also in those cases where the linear components of the execution time, such as event calculation time, overwhelm the relative advantage in event insertion time gained by distributing the event list.

4.2 Empirical Studies

Empirical studies were conducted to gauge the performance of the variants of the distributed event list algorithm under a variety of conditions. In these studies, a family of queueing models with high degrees of inherent parallelism were simulated using each variant of the distributed algorithm. The results demonstrate the relationship between speed-up and certain characteristics of the simulation, and provide insights into the most effective strategies for transmission of Null messages under varying conditions, as well as yielding some elementary heuristics for mapping a given simulation model into a logical system.

4.2.1 Methodology Empirical performance studies were conducted on the Intel iPSC-d5 hypercube systems at AFIT. Synthetic simulation models were constructed, and executed under controlled conditions.

4.2.1.1 Simulation Workload Model The models used as the simulation workload in the empirical studies are queueing models consisting of 32 replications of a homogeneous sub-model, connected in various topologies. The basic sub-model, proposed by Prof. W. Shaw at AFIT, ensures a model with a high degree of inherent parallelism, with the additional properties of independent (entity creation) events at each sub-model and a balanced flow of messages between sub-models. The basic sub-model consists of two single-server queue PP's in tandem, with entity creation and termination processes, connected in a feedback loop as shown in Figure 4.1.

A balanced flow of entities between sub-models is achieved by controlling arrival and service rates, and routing probabilities. Exponential inter-arrival times with a mean of 1000 time units are used for the creation process at each sub-model, except where a sub-model is a "source" process with no entities arriving from any other sub-model. In that case, an exponential inter-arrival time of mean 1111.1 is used. Service times for all sub-models are biased exponential with a mean of 100 time units and a bias of 1 time unit. The bias is introduced to guarantee a non-zero service time, thus enabling lookahead predictions to be made. Figure 4.1 depicts the workload sub-model in queueing symbology.

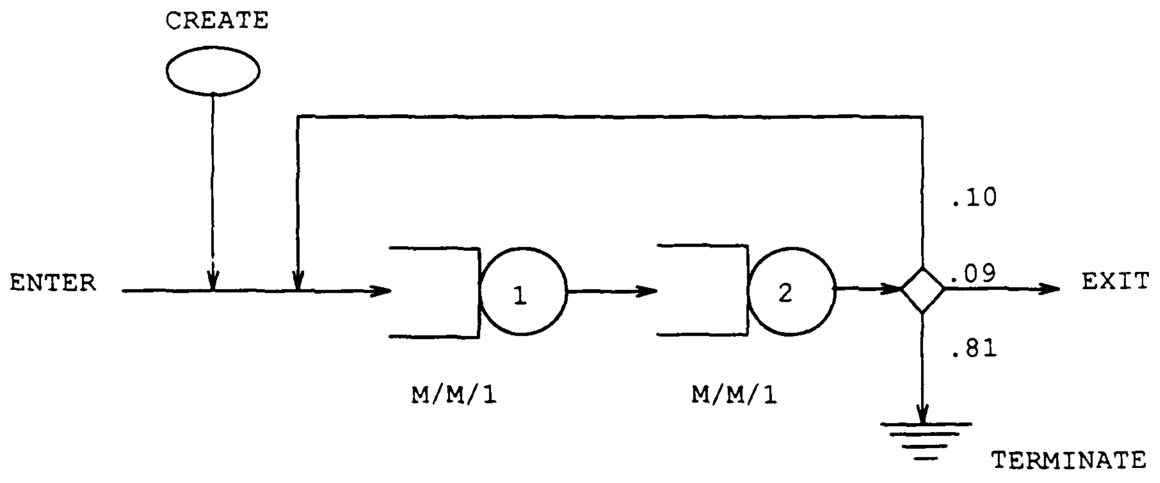


Figure 4.1. Balanced Flow Queueing Sub-model

Complex simulation models are created by connecting 32 of the basic sub-models in a topology such that an entity exiting one sub-model either enters a connected sub-model (in the form of an event message sent between LP's) or is terminated. A probability is associated with each possible transition of an entity between connected sub-models. In the empirical studies, logical systems of fewer than 32 LP's were constructed by grouping the basic sub-models to form larger sub-models in accordance with the assignment criteria in use. Note that it is possible to assign portions of a basic sub-model to different LP's. This was not done in the experiments, however, because of the tightly coupled nature of the basic sub-model, as well as the convenience of manipulating sub-models instead of individual PP's.

4.2.1.2 Experimental Environment All experiments were conducted on the Air Force Institute of Technology's two Intel iPSC-d5 Hypercube multiprocessor systems. The iPSC hypercube consists of 32 homogeneous processor nodes, each node based on an Intel 80286 CPU. The nodes communicate exclusively through message-passing (no shared memory), and are connected to each other in a 5-dimensional hypercube topology with 10 Mbit/s point-to-point serial communications channels. Nodes are also connected via global Ethernet channel to the Cube Manager, an Intel System 286/310 microcomputer [Int86b].

Due to the lack of a blocking send communication protocol in the iPSC node operating system [Int86a], the performance effects of limited buffer size were not considered. The existing non-blocking send protocol was used, and can be viewed as equivalent to a blocking send implementation in which buffer sizes are sufficiently large to eliminate the need for any LP to block while sending (for those simulations performed). The distributed algorithm was implemented as if a blocking send was in use, with Null messages sent under conditions sufficient to avoid the acyclic deadlock possibility raised by a blocking send protocol. Because unconstrained buffer size is the most general case of a system of communicating processes and these empirical studies do not claim that the achieved speed-ups are attainable in every case, the performance effects of constraining the buffer size (and resulting implications for assignment heuristics and Null message strategies) are left as topics for future research.

The implementation of the distributed event list algorithm is a modification of a package of subroutines for performing distributed discrete-event simulation developed in Ryan-McFarland (TM) FORTRAN by Prof. B. Donlan at AFIT. This implementation provides an environment for distributed simulation similar to early FORTRAN-based simulation languages such as GASP [Pri74]. In addition to the statistics collection and other simulation functions generally provided by such languages, synchronization and message-passing for distributed simulation is provided in a manner nearly transparent to the user. More information on this implementation is available from: AFIT/ENG, Wright-Patterson AFB, Ohio 45433, (Attn: Dr T. Hartrum).

The exponential random variates used were derived from (0,1)Uniform pseudo-random numbers using the simplified inverse transform method [BC84]. The generation of uniform random numbers was accomplished using the portable FORTRAN multiplicative congruential generator proposed by Schrage, with modulus $2^{31} - 1$ and multiplier 16807. The seed values for random number generation were taken from pp. 212-3 of Bratley, Fox, and Schrage, and are reported to provide non-overlapping sequences of length 131,072 [BFS83]. Independent random number streams were used for events at each sub-model, so that the simulation results for a particular model did not vary over different distributed and sequential implementations.

Empirical testing of the (0,1)Uniform random number streams was accomplished to verify that the values generated were indeed uniform and independent. A run length of 5000 numbers was tested for each of the random number streams that was used. To test the independence of the values within each stream, the up-down runs test was performed with $\alpha = .05$. To test for uniformity, the Chi-Square test was performed with $\alpha = .05$ and $k = 100$.

The sequential simulations used to calculate speed-up factors were each executed on a single processing node of one of the iPSC hypercubes, to ensure homogeneous processor speed among observations. The algorithm for sequential simulation used the same event-list implementation as the distributed event list algorithm, with all overhead associated with message-passing and distributed simulation removed.

4.2.1.3 Performance Measurement and Instrumentation All experimental runs were accomplished on all or some subset of the 32 processor nodes. To initiate a run, an identical process is loaded and started on each node of the hypercube, and a configuration message is subsequently sent from the Cube Manager to each node in the executing set of processes to inform it of its assigned sub-model of the simulation.

When all participating processes have responded with a "configuration received" message, a global "start simulation" message is sent. Processes not participating in the simulation remain in a loop, waiting for a configuration message to arrive. As each node completes its simulation, a "simulation complete" message is sent to the Cube Manager. When completion messages are received from all executing nodes, simulation statistics are collected from each node. During the experimental runs, no other processes were executed on the processing nodes.

Included in the simulation execution times, in both distributed and sequential algorithms, is the collection of certain simulation statistics. Population data was gathered on inter-arrival and service times, and the time-in-system of all entities in the queueing model. Data was collected on the length of each queue in the simulation model, including the event list. Time data was also collected for server utilization. Collection of data to calculate statistics such as these can be considered an integral part of any simulation, and was therefore included in the execution time of all simulation models, while the calculation of statistics from the data was not. In addition, the numbers of Null and event messages received and sent at each LP were accumulated in the distributed simulation runs. The additional overhead of this collection, not reflected in the equivalent sequential simulation, was found to be negligible, and was therefore ignored.

Empirical observations were made for simulation runs of 1000000 time units, to allow the underlying simulation to reach steady state conditions. All simulation runs were triple-replicated, and the mean taken as the observed value.

4.2.2 Empirical Results

4.2.2.1 Effects of Topology and Spin Loop The topology of connections among the LP's of a distributed simulation has been identified as a significant determinant of performance in performance studies of the Chandy-Misra algorithm [PWM79, RM88]. In response to this, the performance of the distributed event list

S 1
S 2
S 3
S 30
S 31
S 32

Figure 4.2. Network of Sub-models in Tandem Topology

algorithm was investigated for simulation networks of several topologies. The most basic of these is the tandem topology (See Figure 4.2), consisting of a set of M sub-models S_m, $m = 1, \cdots, M$ with probability 1.0 of entity routing from S_m to S_{m+1}, except for S_M, which routes outgoing entities into a termination process. With no possibility of deadlock, the tandem model provides little challenge to a distributed simulation algorithm, and satisfactory performance for tandem networks can be considered a "feasibility test" for a distributed simulation algorithm [RMM88].

Tandem topologies are expected to perform well in distributed simulation, so it is not surprising that significant speed-ups have been realized by applying the distributed event list algorithm to tandem networks. The attained speed-up factors, however, were somewhat higher than the number of processors in most cases. This exceeds what had previously been thought to be the theoretical bound on speed-up of a distributed simulation. These super-linear speed-ups are evidence of the super-linear effects of the event list implementation as described in Section 4.1. Figure 4.3 presents the speed-up factors achieved for the tandem topology.

A more complex family of topologies is the class of feed-forward topologies, in which entities exiting a sub-model can be routed to one of several sub-models,

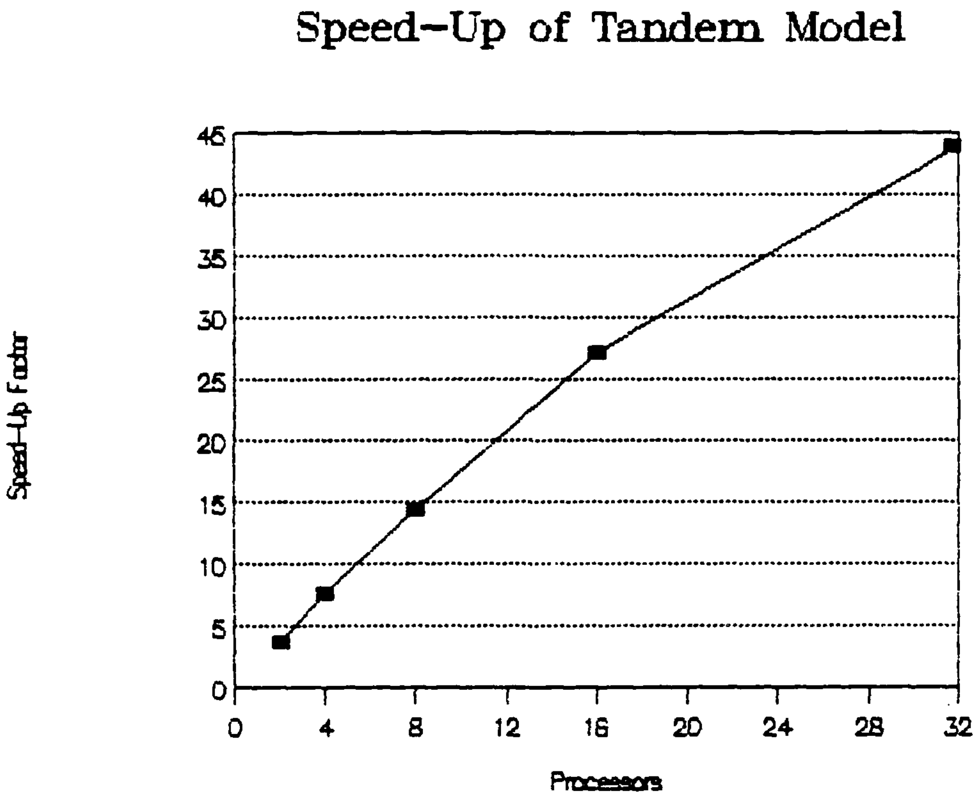


Figure 4.3. Speed-Up for Tandem Topology

with the restriction that there exist no directed cycles in the sub-model connection graph. Feed-forward networks often contain sets of sub-models in the fork-join configuration that may cause acyclic deadlock to occur. Two versions of a generalized feed-forward topology with a high degree of branching, as shown in Figure 4.4, were investigated. A "balanced" version of the feed-forward topology was evaluated, in which the entities emanating from a sub-model have an equal probability of branching to a given connected sub-model. In addition, an "unbalanced" version was investigated, in which an arbitrarily chosen path of each multiple branch was assigned a routing probability of .01, with the remaining .99 probability divided evenly over the remaining paths.

In performance studies of the Chandy-Misra algorithm, the introduction of feed-forward branching in the logical system has been shown to have an adverse impact on speed-up [RMM88]. Those results are contradicted by the performance of the distributed event list algorithm in feed-forward networks (See Figure 4.5). Both balanced and unbalanced feed-forward models achieved speed-ups consistently equal to or better than those observed in the tandem model.

The superior performance achieved for models with feed-forward topologies is partially attributable to the nature of the simulation workload model in use. A high ratio of internal events to outside communications in each sub-model make the simulation particularly amenable to a two-dimensional decomposition, and causes

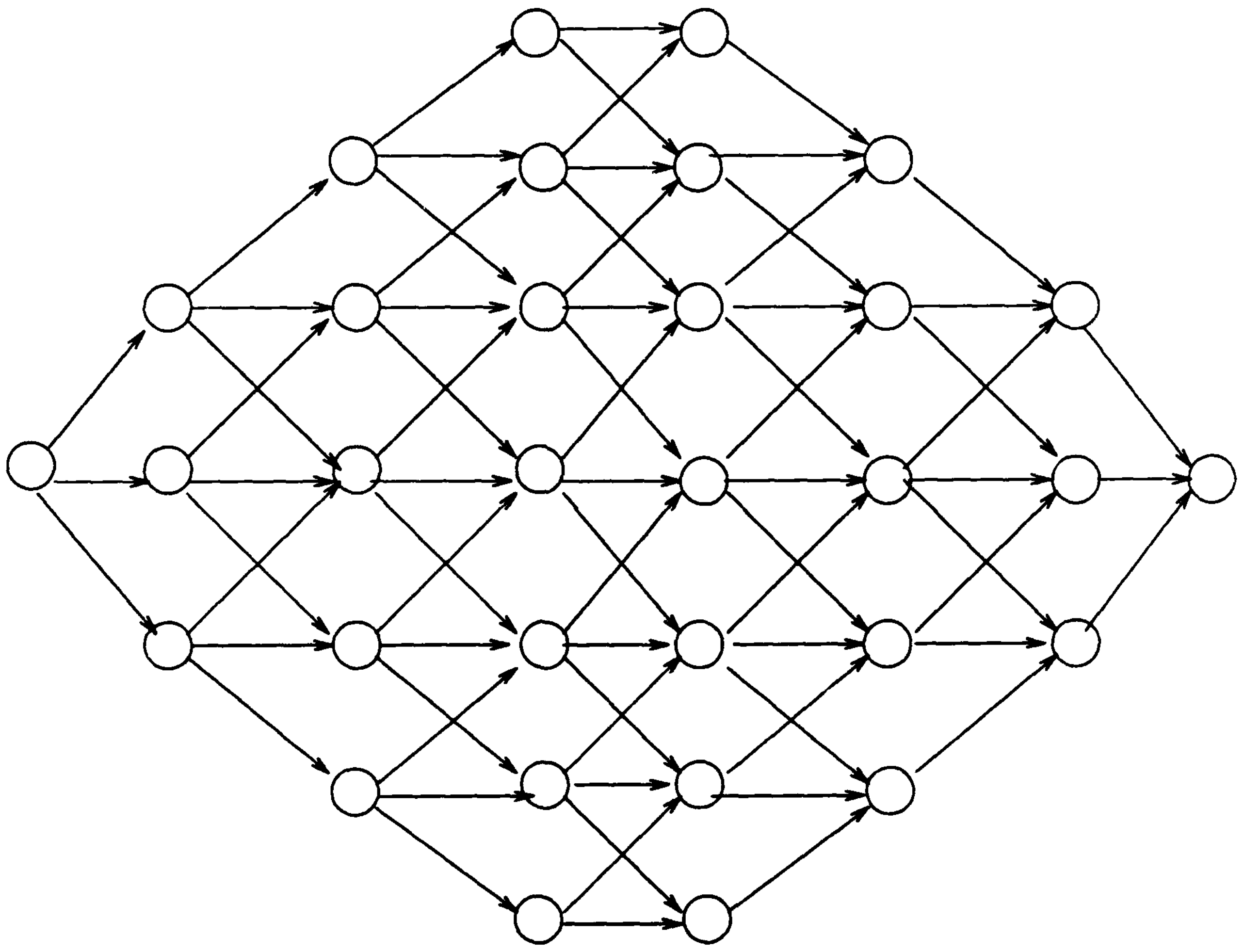


Figure 4.4. Network of Sub-models in Feed-Forward Topology

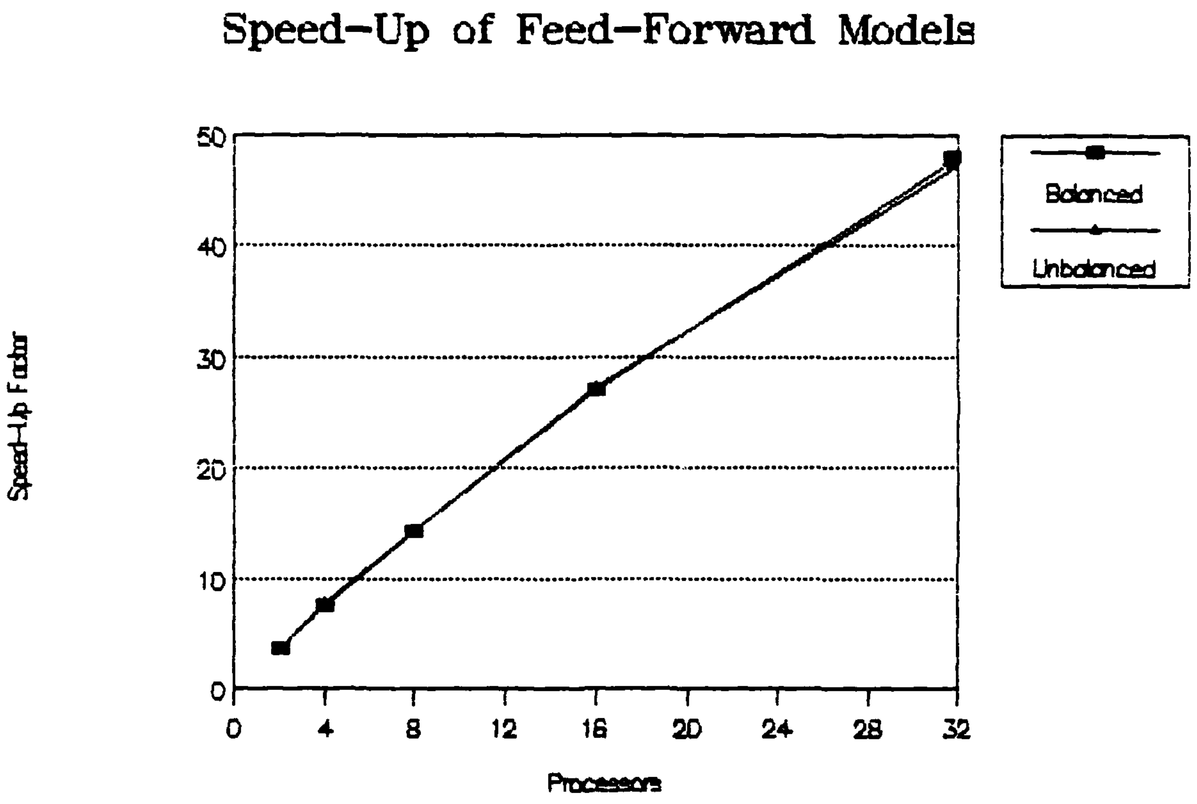


Figure 4.5. Speed-Up for Feed-Forward Topology

performance to be relatively insensitive to the fairly large numbers of Null messages transmitted in these feed-forward networks.

Another common topological class of simulation networks are those in which directed cycles or "feedback loops" appear in the sub-model connection graph. Feedback loops in a simulation topology test the cyclic deadlock avoidance mechanism of the distributed simulator. Performance of the Chandy-Misra Null Message algorithm has been shown to react negatively to the presence of feedback loops in the LP connection graph [PWM79, CM81]. This property is also present in the distributed event list algorithm with Null messages. Introducing a feedback path over a set of LP's appears to cause a time-driven effect as described in Chapter 2, in which each LP is dependent on another LP for time advance, the net result being that all LP's are constrained by the slowest LP in the cycle.

This effect was demonstrated by adding a "pseudo" feedback loop from the last LP to the first LP in a tandem logical system. The underlying simulation model was not changed, however, so that the there was no actual probability of entities feeding back. The tandem model executed as it normally would, but the logical system synchronized time advance as if a back-to-front feedback loop existed (see Figure 4.6). Differences in distributed execution time from the tandem model are then wholly attributable to the added synchronization and communication overhead of the pseudo-feedback, and execution times can therefore be directly compared. The tandem and pseudo-feedback models were executed on 32 processors (with identical

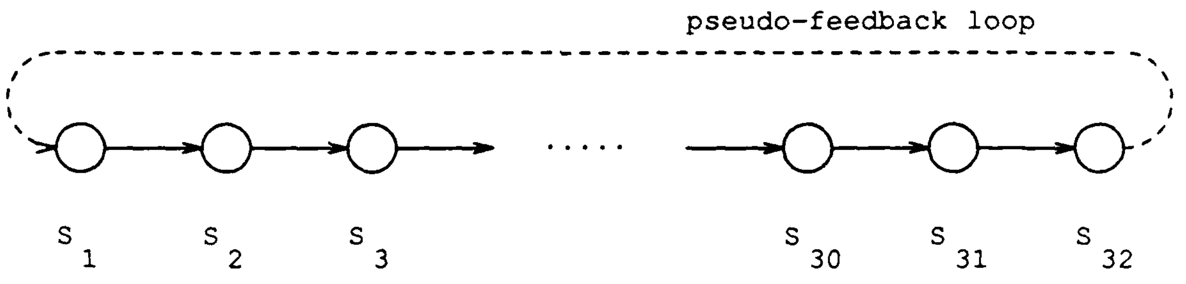


Figure 4.6. Tandem Logical System with Pseudo-Feedback

Topology	Execution Time (s)	# of Null Messages
Tandem	36.0	6422
Pseudo-feedback (1 Loop)	3601.3	1.9×10^6
Pseudo-feedback (2 Loops)	3729.2	3.0×10^6
Pseudo-feedback (3 Loops)	3963.1	4.4×10^6
Pseudo-feedback (4 Loops)	4361.48	7.0×10^6

Table 4.2. Effect of Pseudo-Feedback Loops on Tandem Model

processor assignment). The introduction of a pseudo-feedback loop in the tandem model resulted in a hundred-fold increase in execution time and an "explosion" of Null message traffic. Additional pseudo-feedback loops nested inside of the outermost loop resulted in further significant increases in execution time and the number of Null messages transmitted. These results are shown in Table 4.2.

The effects of a computational workload applied to each event of the simulation were evaluated. A computational workload, also known as the *spin loop* [Fuj88], was associated with each event in the distributed and sequential simulations. The spin loop is a totally artificial computational load; in this case each spin loop consisted of two multiply operations and a divide operation. Experiments were conducted for spin loops of 0 and 1000, for tandem and feed-forward topologies distributed over 2^i, $i = 1, \cdots, 5$ processors.

Adding a spin loop to the computation of each event had several effects. In a simulation with a low ratio of internal events to event messages at each LP, the increased amount of work done in relation to the communications overhead would tend to increase speed-up. In distributed simulations with a high internal event/communications ratio as displayed by the models in these empirical studies, other components of execution time are dominant over communications overhead. In the distributed event list algorithm, increased spin loop also causes the list insertion time to be a relatively less-important component of execution time. The constant load per event can overwhelm the super-linear advantage gained by distributing the event list.

The latter effect can be expected to be dominant in the models presented above, since achieved speed-ups have been consistently super-linear. This does turn out to be the case for both tandem and feed-forward networks, as seen in Figure 4.7 (only balanced feed-forward is shown; unbalanced feed-forward results were very similar).

4.2.2.2 Null Message Strategies The relative effectiveness of the variant strategies for transmitting Null messages was evaluated in a series of experiments. Two variants, the use of a Null Message Time-out and the addition of Stimulus Nulls,

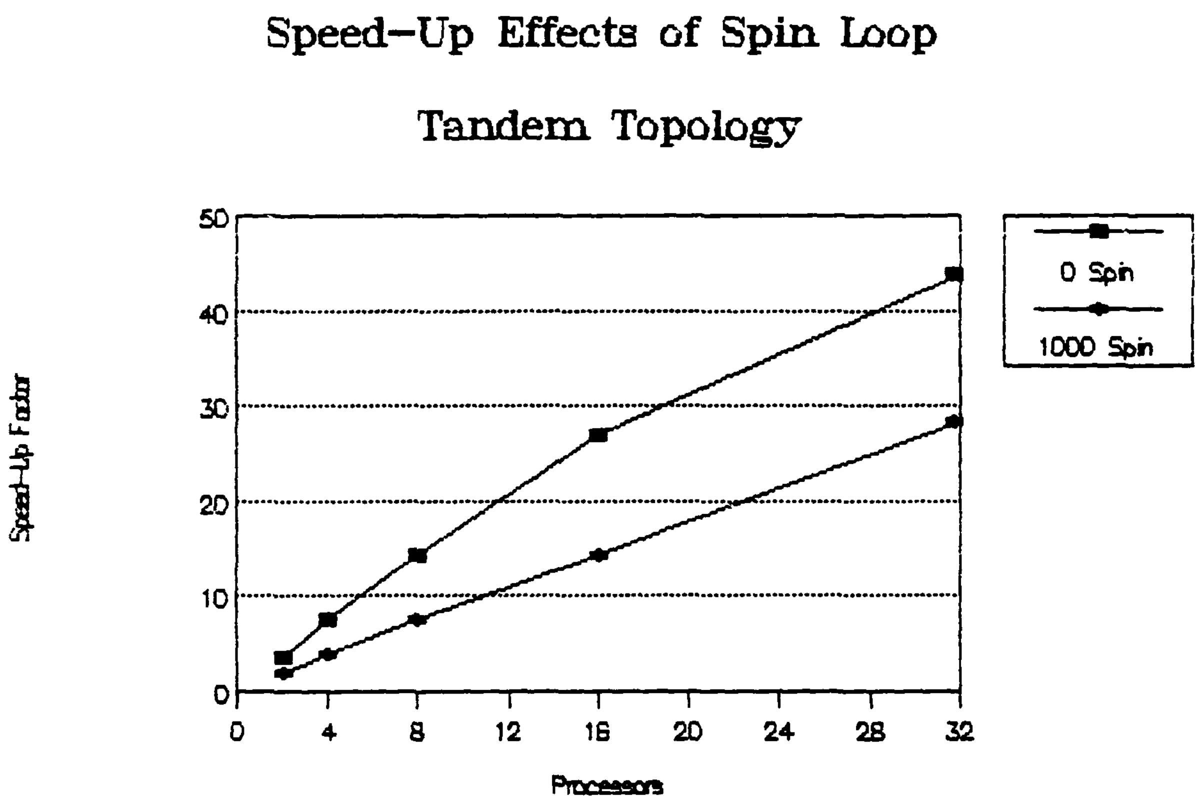


Figure 4.7. Speed-Up Effect of Spin Loop

Topology	Spin Loop	Number of Nodes	Null Messages
tandem			standard
feed-forward(balanced)	0	2^i	w/ Time-out
		$i = 1, \cdots, 5$	
feed-forward(unbalanced)	1000		w/ Stimulus Nulls

Table 4.3. Summary of Factors for Null Message Strategy

both described in Chapter 3, were evaluated in comparison with the "standard" strategy for Null message transmission. Comparisons were made for networks of various topologies, for numbers of processors $2^d, d = 1, \cdots, 5$, and with the addition of spin loop. A summary of the experimental factors is presented in Table 4.3. Note that no feedback topologies were evaluated. Preliminary analyses demonstrated that both Null message variants caused the number of Null messages generated by feedback models, already excessive, to further increase. This often led to buffer saturation and the abnormal termination of the simulation, as a result of necessary implementation compromises that had been made (discussed in Section 4.2.2). It was decided to orient the research toward other, more productive areas.

One variant method for Null message transmission is the Null message with Time-out algorithm described in Section 3.2.6.2. Recall that this algorithm variant sends Null messages after a specified amount of real clock time has passed without a simulation time advance. In the experiments, the standard Null message strategy was compared with the *Time-out algorithm* for time-out values of 25 ms, 100ms, 1000ms, and 10^6 ms.

In the case of the tandem model, the Time-out strategy was found to hurt performance in almost every instance. This is somewhat intuitive, since a tandem model with a balanced workload sends relatively few Null messages, so there is little chance of excessive Null message overhead. (In addition, the high internal /external event ratio of the workload model would appear to benefit from a certain amount of Null message traffic.) The negative effect of Time-out was most pronounced for the case of 0 spin loop,as can be seen in Figure 4.8. In these observations, increasing the Time-out value always decreased the speed-up. The decrease in speed-up was not proportional to the Time-out value, but roughly proportional to the number of Null messages eliminated by the Time-out (the limit, of course, being when no Null messages are sent). This effect is consistent with the relative insensitivity to Time-out of the simulations distributed over few nodes, since those highly-utilized processes send Null messages infrequently in the tandem model.

Because of the decreased ratio of the Time-out values to event processing times, Time-outs had less effect when a spin loop of 1000 was introduced. The effect of the Time-out was still negative in each case, however.

Feed-forward models were evaluated with the time-out algorithm, and also suffered decreased performance in comparison to the standard method. As in the tandem case, the decrease in speed-up was consistent with increased time-out, with the effect proportional to the number of Null messages eliminated by each Time-out. The negative effect of Time-out on the unbalanced feed-forward model was

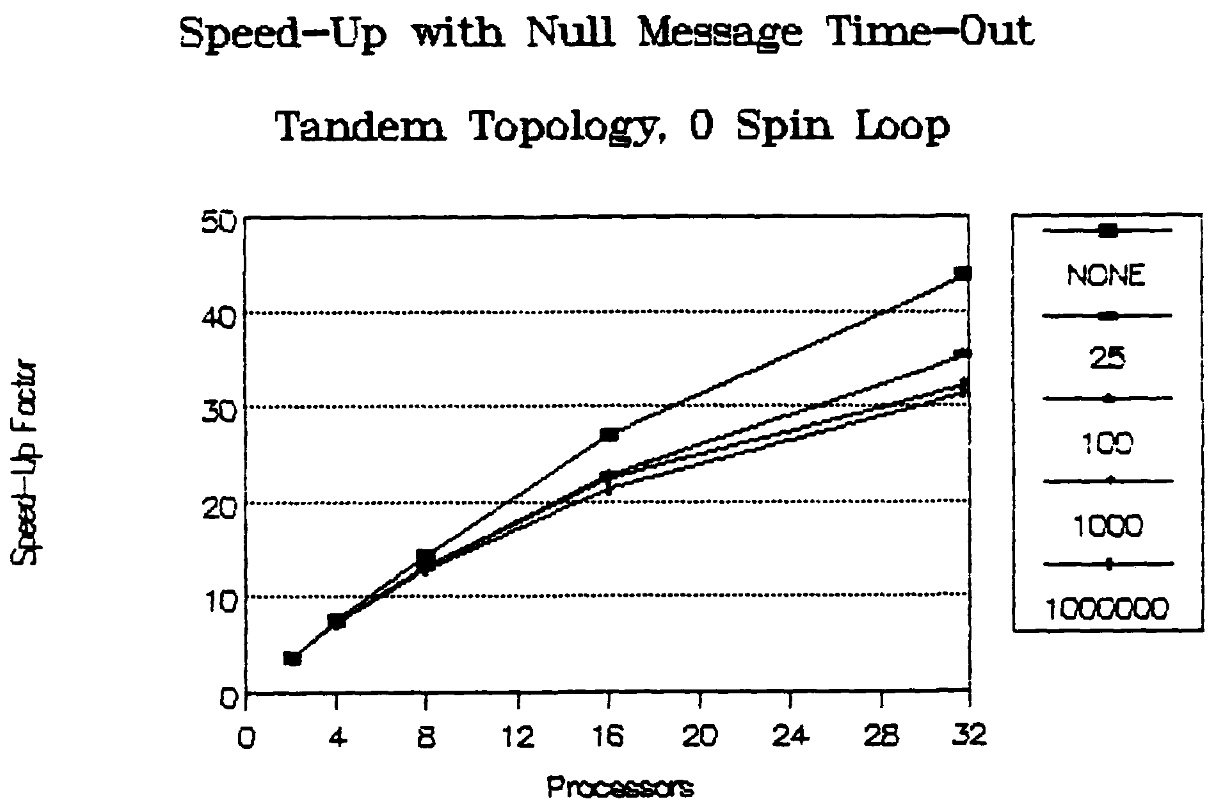


Figure 4.8. Tandem Speed-Up, Time-out Nulls, 0 Spin Loop

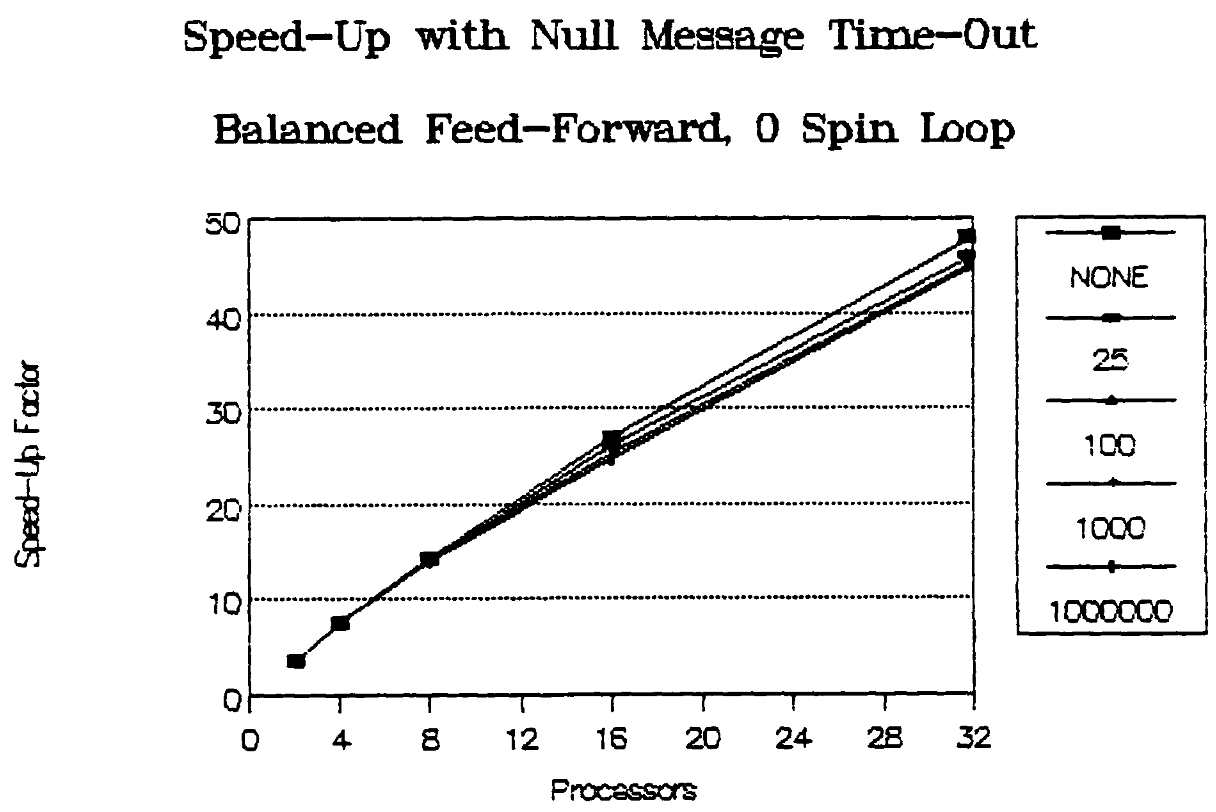


Figure 4.9. Balanced Feed-Forward Speed-Up, Time-out Nulls, 0 Spin Loop

noticeably more than that in the balanced case, which was relatively unaffected (See Figure 4.9).

The poor performance of the Time-out algorithm is also partly attributable to high internal event/communications ratio of the workload model. More positive results would be expected in applying Time-out to simulations that are "communications bound," especially those in networks with high degrees of branching.

As the sending of extra Stimulus Null messages (as described in Section 3.2.6.1) is, in some sense, the "complement" of the Time-out, the Stimulus Null method would appear to have potential, given the failure of the Time-out algorithm that has been observed. Experiments were performed in which Stimulus Nulls were transmitted over the outgoing message channels of each LP in conjunction with 10%, 1%, and 0.1% of the events simulated at that LP.

Results of these experiments show that slight increases in speed-up are regularly achievable with Stimulus Nulls, across all topologies and spin loop values. This was typified by the performance of the balanced feed-forward case with 1000 spin loop, shown if Figure 4.10.

An interesting observation is that of feed-forward networks with 0 spin loop. Here both balanced and unbalanced models showed a decrease in speed-up at 16 to 32 nodes as 10% Stimulus Nulls were applied. For these models, lower levels of Stimulus Nulls had negligible effect, as seen in Figure 4.11. Because the feed-forward models with 0 spin loop have the highest Null message traffic to begin with, the appearance is that of a "threshold" for Null messages, up to which performance improves, but after which performance begins to drop off due to the overhead of reading and sending extraneous Null messages.

The assignments of simulation models to logical processes in the above experiments were chosen so as to maintain a balanced workload among logical processes. Identical assignments were used across all observations. The logical process as-

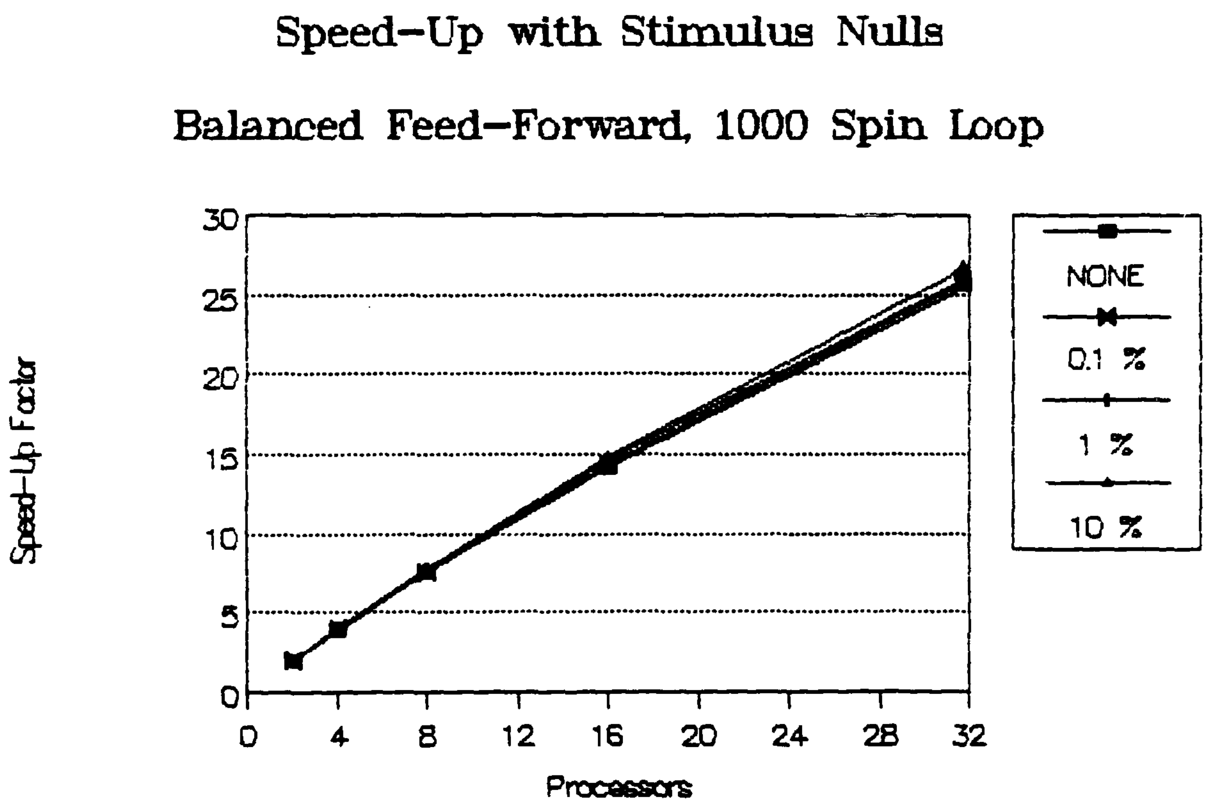


Figure 4.10. Balanced Feed-Forward Speed-Up, Stimulus Nulls, 1000 Spin Loop

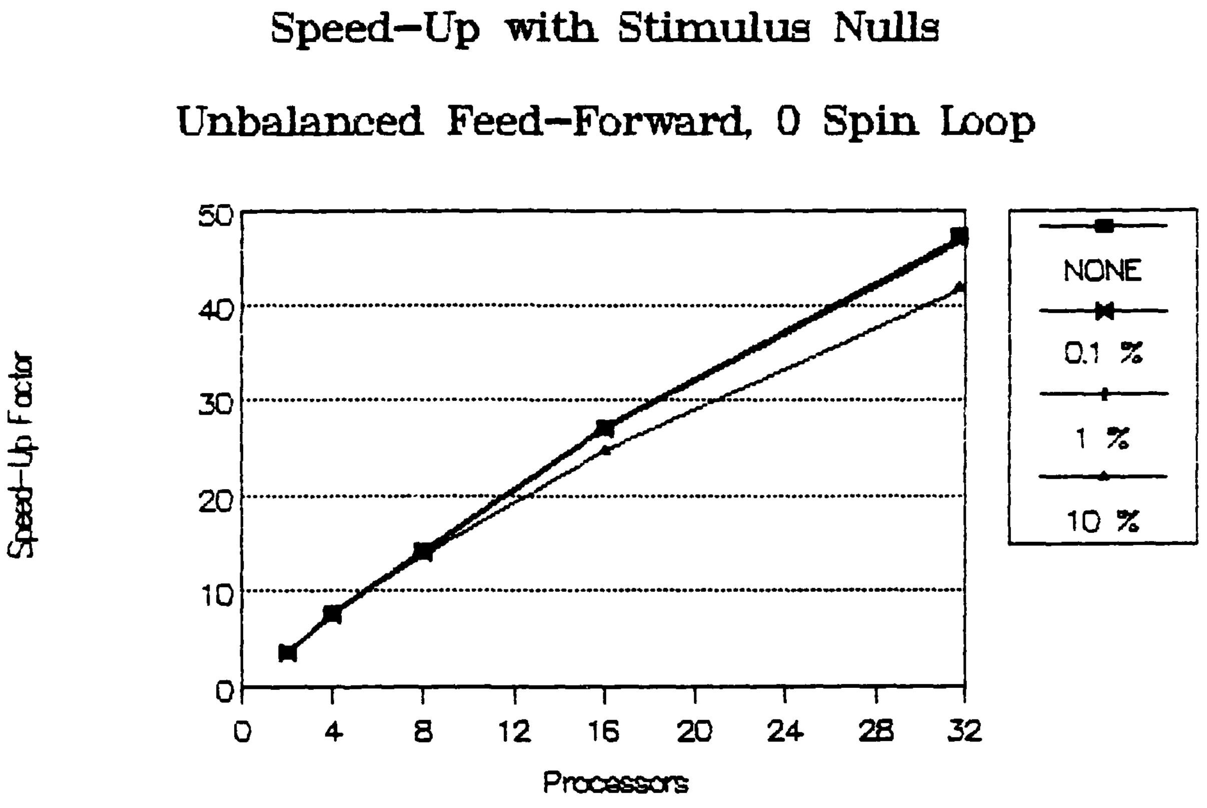


Figure 4.11. Unbalanced Feed-Forward Speed-Up, Stimulus Nulls, 0 Spin Loop

signment did not induce feedback loops in any instance. The assignment of logical processes to processors of the hypercube was done to minimize communications overhead in the tandem case and variants thereof. The processor assignment was fixed for all runs of each topology.

4.2.2.3 The Assignment Problem The assignment of processes to processors is a classic problem in distributed processing. Two conflicting factors, communications overhead and processing workload balance, must be traded-off in order to achieve an assignment that maximizes throughput. Methods for calculating an optimal assignment for a distributed application are generally computationally intractable, however [CH*80]. This may be especially applicable in the area of simulation, where the communications and processing loads of a simulation model are not well-known *a priori* (or else closed form solutions, rather than simulation, could have been used to solve the application problem). It is then necessary to resort to heuristic methods for determining a "good," rather than "best" assignment of processes to processors.

In the distributed event list algorithm, the way in which the logical process encapsulates the simulation sub-model at its processor adds a new dimension to the assignment problem. The monolithic nature of the LP time advance mechanism makes it possible for non-cyclic topologies of PP's to be mapped into cyclic networks of LP's, as shown in Chapter 3. Even simple tandem networks can be assigned in such a way. Given the poor performance demonstrated by the distributed event list

Assignment	Characteristics
Balanced Load	Vertical format; No feedback introduced; Equal number of PP's per LP
Pure Vertical	No feedback introduced; Load balance not considered; Logical system reduces to tandem

Table 4.4. Assignment Strategies for Feed-Forward Topology

Assignment	Execution Time (s)	# Null Messages	# Event Messages
Balanced Load	103.2	4288	2394
Pure Vertical	160.2	4055	3049

Table 4.5. Effect of Assignment, Balanced Feed-Forward, 8 Processors

algorithm in association with feedback loops in the logical system, the topology of the logical system must be considered in addition to the "traditional" assignment factors of workload and communication overhead.

Experiments were conducted to provide insights into effective strategies for mapping a simulation onto a given number of logical processes. In one set of these experiments, models of feed-forward topology, with both balanced and unbalanced routing, were assigned to a logical system of 8 processors, with the assignment methods shown in Table 4.4.

The results of these assignment experiments show that achieving an even workload over all logical processes is of greater importance than eliminating branching, for both balanced and unbalanced feed-forward networks. These results mesh with earlier findings of comparable performance for tandem and feed-forward topologies.

Assignment	Execution Time (s)	# Null Messages	# Event Messages
Balanced Load	103.5	4456	2307
Pure Vertical	159.6	4124	3023

Table 4.6. Effect of Assignment, Unbalanced Feed-Forward, 8 Processors

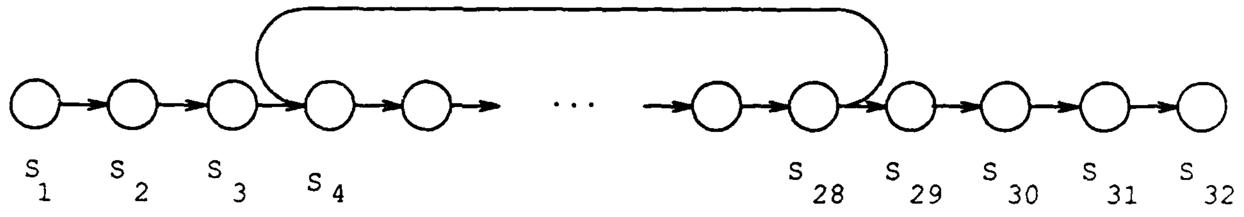


Figure 4.12. Tandem Topology with Single Loop

An experiment was conducted to gauge the effectiveness of containing an existing feedback loop within a logical process at the expense of load balancing. A new topology was introduced for this experiment, consisting of a tandem network with a single feedback loop with .01 routing probability, as shown in Figure 4.12. The configuration of this topology yields the worst possible load balance for an 8 node assignment if the feedback loop is contained within a logical process. The assignment strategies used are given in Table 4.7.

Assignment	Characteristics
Balanced Load	Equal number of PP's per LP
Loop Contained	Contain feedback loop within a single LP; Load balance not considered

Table 4.7. Assignment Strategies for Tandem Topology w/ Single Loop

Assignment	Execution Time (s)	# Null Messages	# Event Messages
Balanced Load	2997.44	7.3×10^5	685
Contain Loop	650.1	1190	655

Table 4.8. Effect of Assignment, Tandem with Single Feedback, 8 Processors

It is apparent from the Single Loop model that the elimination of feedback loops can play a more important role than load balance in the performance of the distributed event list algorithm. As shown in Table 4.8, containing the feedback loop at the cost of an unbalanced computational workload can improve execution time by a large amount.

The preceding assignment experiments suggest the essential relationships between the factors of topology and workload balance, and so may be used as starting points for heuristic solutions to the assignment problem. Finding effective heuristics for the assignment problem is a major hurdle to be overcome before the distributed event list algorithm can be considered widely applicable.

4.3 Summary

A linear event list implementation for distributed simulation has been shown to be $O(E^2)$, for E events simulated. Time complexity of event list operations of greater than $O(E)$ has been shown to imply theoretical speed-up factors of greater than N for a simulation distributed over N processors. This result contradicts a commonly held view in the literature, which asserts the existence of a bound of N on attainable speed-up. It has been shown that a speed-up bound of N is unjustified, as it ignores the time complexity of the event list overhead in the sequential simulations used for comparison.

Empirical studies have been conducted to evaluate the performance of the distributed event list algorithm under a variety of conditions. Speed-up greater than N was achieved for certain topologies of simulation models, confirming the above time complexity analysis. The topology of the simulation model was shown to greatly affect the attained speed-up. Simulation networks with directed cycles or "feedback loops" were shown to exhibit extremely poor performance, in agreement with previous performance studies of the Chandy-Misra algorithm [RMM88]. Feed-forward branching topologies, which showed poor performance relative to the tandem model in previous studies (on shared-memory machines), actually performed better than the tandem case in these experiments. This was attributed to the high inherent parallelism and low communications ratio in the models studied.

A high computational workload associated with each event was shown to lower the attained speed-up below N. This contradicted Fujimoto's results, in that the higher computation/communications ratio induced by the increased workload is expected to improve speed-up [Fuj88]. This conflict was explained as a "dampening" of the super-linear effects of distributing the event list with a constant time component associated with each event.

Alternate strategies for sending the Null messages used for deadlock avoidance were compared. Results showed that for tandem and feed-forward topologies, a certain level of Null messages were beneficial to speed-up. It was also seen that a threshold exists, above which additional Null messages are unnecessary overhead. No strategy that was evaluated was shown to improve the poor performance of simulation topologies with feedback.

The problem of assigning a given simulation model to a set of logical processes was addressed. Again, it was seen that topology played a critical role in the effectiveness of an assignment strategy. Avoiding feedback loops was shown to be of greater importance than "traditional" assignment considerations such as balancing the processing load of the logical system.

V. Conclusions

5.1 Summary

An algorithm for distributed discrete-event simulation, the distributed event list algorithm, has been described in this thesis. This algorithm uses an event list to order events at each logical process, and uses a variant of the Chandy-Misra algorithm for inter-process communication, with a prediction function added. Null messages are used to avoid process deadlock, as in the original Chandy-Misra algorithm. The distributed event list algorithm has been shown to require a bounded amount of memory at each logical process.

A study of event list implementations shows that the theoretical speed-up factor for the distributed event list algorithm is in excess of N, for a simulation distributed over N processors, where both sequential and distributed simulation event lists are implemented with a linear list. More efficient event list implementations yield lower theoretical speed-ups, but can still exceed N, which had previously been thought to be the optimum speed-up.

Empirical studies show that speed-up values greater than N can be regularly achieved for simulations with high degrees of parallelism and topologies without feedback, if the computational workload per event was small in comparison to the list insertion time. Topology is shown to be a prime factor in determining attainable speed-up of a given simulation, with the presence of feedback loops playing a critical role.

5.1.1 Null Message Strategies Several Null message strategies were evaluated in addition to the basic deadlock avoidance strategy. The strategy of sending extra "stimulus" Nulls was found to be marginally more effective than the basic strategy in tandem networks, and in cases where there was a large computational load or "spin loop" associated with each event.

The strategy of sending Null messages during the LP Read Phase with a time-out between sends did not prove to be effective. Indeed, this strategy, an attempt to reduce the number of "unnecessary" Null messages, was found to provide worse performance than the basic algorithm in every case. In networks with feedback loops, the time-out strategy had the effect opposite to that intended, causing an "avalanche" of Null messages that virtually paralyzed the logical system, leading to buffer saturation in most cases.

The most significant result of studying the variant Null message strategies was that no strategy was found to improve the poor performance of the distributed event list algorithm in the presence of feedback loops in the logical system.

5.1.2 Assignment Heuristics Empirical studies have yielded some elementary heuristics for the assignment of physical to logical processes when using the distributed event list algorithm. The following basic rules can be used for distributing a simulation over a given number of logical processes. The rules are listed in descending order of importance:

1. Avoid introducing feedback loops due to assignment of the simulation model to the logical system.
2. Contain existing feedback loops within a logical process whenever possible.
3. Achieve a balanced processing load among the logical processes.
4. If possible, reduce the amount of branching in the logical system.

5.2 Assessment of the Distributed Event List Algorithm

The distributed event list algorithm has been shown to provide a significant source of speed-up for discrete-event simulation in many circumstances. There are, however, several properties of the algorithm which may limit its applicability. Perceived advantages and disadvantages of the distributed event list algorithm in relation to other distributed simulation algorithms are enumerated below.

5.2.1 Advantages of the Distributed Event List Algorithm Some advantages of the distributed event list algorithm for distributed discrete event simulation are as follows:

- The distributed event list algorithm can provide heretofore unrealized speed-ups for simulations of certain topologies. The algorithm can be highly efficient, providing significant speed-up with few processors.

- The algorithm is based on the widely-used event-oriented view of simulation, and therefore requires no change in perspective to use, as do some other distributed simulation algorithms. Parallelization of existing sequential simulation models should then be comparatively easy with the distributed event list algorithm.

- The use of a single logical process at each processor, as in the distributed event list algorithm, facilitates checkpointing and the collection of statistics.

5.2.2 Disadvantages of the Distributed Event List Algorithm Some perceived weaknesses in the distributed event list algorithm are the following:

- Attainable speed-up is highly dependent upon the absence of feedback loops in the logical system topology, as in the original Chandy-Misra Null message algorithm. A feedback loop in the simulated system can be negated by containing it inside a single logical process. This practice, however, limits the number of processes in the logical system to the maximum number of disjoint non-cyclic subgraphs of the physical process connectivity graph.

- The distributed event list algorithm can not provide a deterministic execution of simultaneous events in many cases. Because each logical process can only predict its future message output if no event is scheduled at the current simulation time, processes are required, in order to avoid cyclic deadlock, to simulate any events scheduled for the current simulation time, rather than waiting for all possible message events at that time to arrive. Event messages with the same simulation time arriving at a logical process from two different processes are simulated in the arbitrary order of their arrival.

- The tightly-coupled nature of the logical process complicates the assignment problem. As demonstrated in Chapter 3, a logical system may have a radically different topology than the underlying physical system, perhaps introducing feedback loops where none exist in the simulated system. The aforementioned poor performance of cyclic systems and the necessity of maintaining logical system predictability (Chapter 3) make topology an important assignment consideration.

5.3 Recommendations for Further Research

Recommendations for additional research focus on ameliorating some of perceived weaknesses of the distributed event list algorithm. These weaknesses keep the distributed event list algorithm from possessing the qualities of robustness required for a generally-applicable distributed simulation algorithm.

A variant of the distributed event list algorithm was developed that replaces the Null message algorithm for deadlock avoidance with a Marker algorithm for deadlock detection and recovery, as proposed by Misra [Mis86]. This variant should be evaluated as a possible remedy for the poor cyclic performance of the present algorithm. Improving the performance of the distributed event list algorithm in networks with feedback would ease the assignment problem as well. Other possible algorithms for deadlock detection and recovery are outlined in [CM81].

Comprehensive assignment heuristics should be developed, with the eventual goal of integration into an automated system for physical-to-logical system mapping. The elementary heuristics provided above, such as avoiding inducing loops in the logical system, would be relatively straightforward to automate. More challenging would be the automation of in-depth assignment heuristics, utilizing information internal to the physical system to estimate a near-optimal assignment based on the estimated computation and communications load for a given simulation.

A grave weakness in the distributed event list algorithm is its lack of capacity in dealing with simultaneous events. There is no obvious solution to this problem. Unless the algorithm can be modified so that the order of execution of simultaneous events can be ascertained, the distributed event list algorithm will remain unsuitable for those applications, such as digital logic simulation, in which the order of execution of simultaneous events significantly affects the resulting system state.

Bibliography

BCS4. Jerry Banks and John S. Carson. *Discrete Event Simulation*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

BDO85. William E. Biles, Cheryl M. Daniels, and Tamilea J. O'Donnell. Statistical considerations in simulation on a network of microcomputers. In *Proceedings of the 1985 Winter Simulation Conference*, pages 388–393, December 1985.

BFS83. Paul Bratley, Bennett L. Fox, and Linus E. Schrage. *A Guide to Simulation*. Springer-Verlag, New York, NY, 1983.

BJ85. Orna Berry and David Jefferson. Critical path analysis of distributed simulation. In Paul Reynolds, editor, *Distributed Simulation 1985*, SCS, La Jolla CA, 1985.

Bry79. Randal E. Bryant. Simulation on a distributed system. In *Proceedings of the 1st Int'l Conference on Distributed Computing Systems*, pages 544–552, October 1979.

CH*80. W. W. Chu, L.J. Holloway, et al. Task allocation in distributed data processing. *Computer*, 13(11):57–69, November 1980.

CM79. K. Mani Chandy and Jayadev Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5):440–452, September 1979.

CM81. K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

CM83. K. Mani Chandy and Jayadev Misra. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

Com82. John Craig Comfort. The design of a multiprocessor based simulation computer - i. In *Proceedings of the Fifteenth Annual Simulation Symposium*, pages 17–33, March 1982.

Com83. John Craig Comfort. The design of a multiprocessor based simulation computer - ii. In *Proceedings of the Sixteenth Annual Simulation Symposium*, pages 197–209, 1983.

DN66. Ole-Johan Dahl and Kristen Nygaard. Simula - an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, September 1966.

DR83. David L. Davidson and Paul F. Reynolds Jr. Performance analysis of a distributed simulation algorithm based on active logical processes. In *Proceedings of the 1983 Winter Simulation Conference*, pages 263–264, December 1983.

DS80. Edsger W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

Fuj88. Richard M. Fujimoto. Performance measurements of distributed simulation strategies. In *Distributed Simulation 1988*, SCS, La Jolla CA, 1988.

HB84. Kai Hwang and Faye A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York NY, 1984.

Hei86. Philip Heidelberger. Statistical analysis of parallel simulation. In *Proceedings of the 1986 Winter Simulation Conference*, pages 290–295, 1986.

Hoa78. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

Hoa85. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs NJ, 1985.

Int86a. Intel. *iPSC Dynamic Loader Manual*. Intel Scientific Computers, Beaverton, Oregon, November 1986. Order Number 310103-002.

Int86b. Intel. *iPSC System Overview Manual*. Intel Scientific Computers, Beaverton, Oregon, November 1986. Order Number 310610-001.

Jef85. David Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

Jon86a. Douglas W. Jones. Concurrent simulation: an alternative to distributed simulation. In *Proceedings of the 1986 Winter Simulation Conference*, pages 417–423, 1986.

Jon86b. Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

JS85. David Jefferson and Henry Sowizral. Fast concurrent simulation using the time warp mechanism. In *Distributed Simulation 1985*, SCS, La Jolla CA, 1985.

Kau87. Fred J. Kaudel. A literature survey on distributed discrete event simulation. *Simuletter*, 18(2):11–21, June 1987.

Lam78. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

LK82. Averill M. Law and W. David Kelton. *Simulation Modelling and Analysis*. McGraw-Hill, New York NY, 1982.

MC82. Jayadev Misra and K.M. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 4(1):37–43, January 1982.

Mis86. Jayadev Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

Nan81. Richard E. Nance. The time and state relationship in simulation modeling. *Communications of the ACM*, 24(4):173–179, April 1981.

NR84. David M. Nicol and Paul F. Reynolds, Jr. Problem oriented protocol design. In *Proceedings of the 1984 Winter Simulation Conference*, pages 471–474, November 1984.

Par69. David L Parnas. On simulating networks of parallel processes in which simultaneous events may occur. *Communications of the ACM*, 12(9):519–531, September 1969.

Pri74. A. Alan B. Pritsker. *The GASP IV Simulation Language*. John Wiley & Sons, New York NY, 1974.

Pri86. A. Alan B Pritsker. *Introduction to Simulation and SLAM II*. Systems Publishing Corp., West Lafayette IN, 1986.

PV83. Y. Paker and J.P. Verjus, editors. *Distributed Computing Systems*. Academic Press, New York NY, 1983.

PWM79a. J. Kent Peacock, J.W. Wong, and Eric G. Manning. A distributed approach to queueing network simulation. In *Proceedings of the 1979 Winter Simulation Conference*, pages 399–406, December 1979.

PWM79b. J. Kent Peacock, J.W. Wong, and Eric G. Manning. Distributed simulation using a network of processors. *Computer Networks*, 3:44–56, 1979.

Ree85. Daniel A. Reed. Parallel discrete event simulation: a case study. In *Proceedings of the Eighteenth Annual Simulation Symposium*, pages 95–107, 1985.

Rey82. Jr. Reynolds, Paul F. A shared resource algorithm for distributed simulation. In *Proceedings of the Ninth Annual Int'l Computer Architecture Conference*, pages 259–266, April 1982.

Rey83. Jr. Reynolds, Paul F. Active logical processes and distributed simulation: an analysis. In *Proceedings of the 1983 Winter Simulation Conference*, pages 263–264, December 1983.

RM88. Daniel A. Reed and Allen D. Malony. Parallel discrete event simulation: the chandy-misra approach. In *Distributed Simulation 1988*, SCS, La Jolla CA, 1988.

Vita

David L. Mannix was born in [redacted] He graduated in 1980 [redacted] High School i[redacted] He attended Miami University, Oxford, Ohio, graduating in May 1984 with a B.S. in Systems Analysis, while obtaining his commission as a Second Lieutenant, U.S. Air Force, through the ROTC program. He was subsequently assigned to Aeronautical Systems Division, Air Force Systems Command, Wright-Patterson AFB, Ohio, under the Deputy Comander for Strategic Systems. In this assignment, he managed the development of safety-of-flight-critical flight control software, prior to entering the Air Force Institute of Technology in June 1987.

[redacted]

RMM88. Daniel A. Reed, Allen D. Malony, and Bradley D. McCredie. Parallel discrete event simulation using shared memory. In *Distributed Simulation 1988*, SCS, La Jolla CA, 1988.

Sch82. Fred B. Schneider. Synchronization in distributed programs. In *ACM Transactions on Programming Languages and Systems*, pages 179–195, April 1982.

Sha75. Robert E. Shannon. *Systems Simulation: The Art and Science.* Prentice-Hall, Englewood Cliffs NJ, 1975.

VD75. Jean G. Vaucher and Pierre Duval. A comparison of simulation event set algorithms. *Communications of the ACM*, 18(4):223–230, April 1975.

Zei76. Bernard P. Zeigler. *Theory of Modelling and Simulation.* John Wiley and Sons, New York, NY, 1976.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/88D-14	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering — 6b. OFFICE SYMBOL *(If applicable)* AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS *(City, State, and ZIP Code)* Air Force Institute of Technology Wright-Patterson AFB, OH 45433	7b. ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING/SPONSORING ORGANIZATION (See Block 8c) — 8b. OFFICE SYMBOL *(If applicable)* JTFPMO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* Joint Tactical Fusion Program Management Office 1500 Planning Research Drive McLean, Virginia	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
(See Block 19)

12. PERSONAL AUTHOR(S)
David L. Mannix, B.S., Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
MS Thesis	FROM ______ TO ______	1988 December	140

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	07	
12	05	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Distributed Data Processing, Computerized Simulation
Distributed Simulation, Parallel Simulation

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Title: Distributed Discrete-Event Simulation Using Variants of the Chandy-Misra Algorithm on the Intel Hypercube

Thesis Advisor: Nathaniel J. Davis IV, CPT, USA
Assistant Professor of Electrical Engineering

[illegible]
10 Jan 89

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)*	22c. OFFICE SYMBOL
Nathaniel J. Davis IV, CPT, USA	(513) 255-5533	AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

The goal of distributed simulation is to speed up simulation by distributing a simulation model's execution over multiple processors. This thesis reviews existing methods for distributed simulation, and introduces an algorithm for distributed discrete-event simulation, the *distributed event list* algorithm, based on the Chandy-Misra algorithm, with an event list, similar to that used in sequential simulation, at each logical process. Null messages are used for deadlock avoidance. The algorithm is described, and is shown to require a bounded amount of memory at each logical process.

A performance analysis of the distributed event list algorithm is performed. In the analytical portion, a linear event list implementation is shown to be of super-linear time complexity in relation to events simulated. This time complexity implies theoretical speed-up of greater than N for a simulation distributed over N processors. This result contradicts a commonly-held view of the existence of a bound of N on attainable speed-up.

Empirical studies evaluate the performance of the distributed event list algorithm under a variety of conditions. Speed-up greater than N are shown to be achievable for certain topologies of simulation models, confirming the time complexity analysis. The topology of the simulation model is shown to greatly affect the attained speed-up. Simulation networks with directed cycles exhibit extremely poor performance, in agreement with previous performance studies of the Chandy-Misra algorithm. Alternate strategies for sending the Null messages used for deadlock avoidance are compared. Results show that for tandem and feed-forward topologies, a certain level of Null messages are beneficial to speed-up. The problem of assigning a given simulation model to a set of logical processes is addressed. It is seen that topology of the logical system plays a critical role in the effectiveness of an assignment strategy. [illegible]