APPLICATION OF PARALLEL PROCESSING FOR OBJECT ORIENTED DISCRETE EVENT SIMULATION OF MANUFACTURING SYSTEMS

By

HEMANT C. BHUSKUTE

Bachelor of Engineering
University of Bombay
Bombay, India
1986

Master of Manufacturing Systems Engineering
Oklahoma State University
Stillwater, Oklahoma
1989

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
DOCTOR OF PHILOSOPHY
May, 1993

APPLICATION OF PARALLEL PROCESSING FOR OBJECT ORIENTED DISCRETE EVENT SIMULATION OF MANUFACTURING SYSTEMS

Thesis Approved:

Joe H. Mize
Thesis Adviser

Kenneth E Case

[illegible]

Manjunath Kamath

[illegible]

Thomas C. Collins
Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to all those who have helped during my doctoral studies. First and the most of all I would like to thank the chairman of my doctoral committee, Dr. Joe Mize for the time and effort that he has invested in my research and education. He has been a true friend and a great motivator. I would also like to thank each of the members of my doctoral committee for their impact on my education and research. Dr. Kenneth Case provided an example of an excellent educator, specifically his exceptional teaching style. Dr. Kamath and Dr. Schuermann have provided numerous invaluable suggestions especially at the research proposal stage. Dr. Teague installed the C++ compiler onto the Intel iPSC/2 Hypercube and supplied the specific instructions for using the compiler.

I would also like to thank my parents, Mr. Chandrashekhar and Mrs. Shailaja Bhuskute. Their ever present support and encouragement has been a source of strength. Thank you for instilling in me a solid foundation of moral strength and an everlasting quest for knowledge. I would also like to thank my aunt and uncle Deepak and Rohini Phadke for their emotional support and guidance during my early years in the United States.

Among friends and colleagues, I would like to thank my former roommates Ashok Ramchand and Sanjay Srinivasan, my best friend David Sebert, and the friends in the CIM Center Manoj Duse and Jagannath Gharpure for their wonderful friendship. I would like to extend my special appreciation to my former boss Millett O'Connell and his wife Heidi for their hospitality and friendship.

For assistance in accomplishing this research, I would like to extend my sincere appreciation to the organizations providing funding for my graduate study. The AT&T Foundation and Oklahoma Center for the Advancement of Science and Technology (OCAST) provided significant financial and equipment support for the CIM Center where much of this work was performed. The IE&M department provided support for all the five years of graduate study in the form of research and teaching assistantships.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
	Motivation for Research	1
	Overview of the Dissertation	4
II.	PROBLEM STATEMENT	6
	Introduction	6
	Simulation of Discrete Systems	7
	Simulation in Manufacturing Context	8
	Object Oriented Modeling	9
	Concurrent Object Oriented Programming	12
	Parallel Discrete Event Simulation	14
	Problem Description	17
	Unanswered Questions	22
III.	LITERATURE REVIEW	24
	Introduction	24
	Concurrent Object Oriented Programming	24
	Parallel Processing and Parallel Computer Architecture	28
	Parallel Discrete Event Simulation	33
	Processor Synchronization Mechanisms	38
IV.	STATEMENT OF RESEARCH	42
	Research Goal	42
	Research Objectives	42
	Research Scope and Limitations	44
	Research Contributions	45
V.	RESEARCH METHODOLOGY	47
	Research Plan	47
	Selection of The Manufacturing System	52
	Selection of Submodel Network Topologies	54

Chapter Page

VI. OBJECT ORIENTED REPRESENTATION 58

Introduction 58
C++ Class Library 58
Experimental Setup 64

VII. EVALUATION OF CONCURRENT OOP CONSTRUCTS 67

Introduction 67
Evaluation of ConcurrentSmalltalk Constructs 67

VIII. SUBMODEL CREATION METHODOLOGY 76

Introduction 76
Analysis of Parallel Discrete Event Simulation 76
Submodel Creation Methodology 79

IX. COMMUNICATION PROTOCOL DESIGN 83

Introduction 83
Design of Communication Protocols 83

X. EXPERIMENTATION RESULTS 92

Introduction 92
Simulation Verification 92
Experimentation Setup 95
Experimentation Results 96
Submodel Network Topology - Independent Clusters (E1) 100
Submodel Network Topology - Tandem (E2) 104
Submodel Network Topology - Fork (E3) 108
Submodel Network Topology - Join (E4) 112
Submodel Network Topology - Fork + Join (E5) 116
Comparison of Communication Protocols (E6) 121
Summary of Experimental Results 127

XI. SUMMARY, CONCLUSIONS AND RECOMMENDATIONS 130

Introduction 130
Research Summary 130
Research Contributions 135
Recommendations for Future Research 136

Chapter Page

BIBLIOGRAPHY 138

APPENDIXES*

APPENDIX A - C++ CODE FOR PARALLEL DISCRETE EVENT SIMULATION ENVIRONMENT

APPENDIX B - SIMULATION INPUT, OUTPUT, TRACE, MODEL, AND ALLOCATION TABLE FILES

APPENDIX C - VERIFICATION OF SIMULATION ENVIRONMENT USING SLAM II

APPENDIX D - VERIFICATION OF PARALLEL IMPLEMENTATION OF THE SIMULATION ENVIRONMENT

APPENDIX E - C++ CODE FOR INPUT FILE CREATION PROGRAMS

APPENDIX F - SIMULATION EXPERIMENTATION AND OTHER UNIX SHELL SCRIPT FILES

APPENDIX G - ANALYSIS OF VARIANCE OF EXPERIMENTAL DATA

* Available in the School of Industrial Engineering and Management Library at Oklahoma State University.

LIST OF TABLES

Table Page

1. Composition of Intel iPSC/2 Hypercube Cx or Sx Node 32
2. Production and Consumption Times.. 73
3. Verification of the PDES Environment 94
4. Experimentation Identification 95
5. Cluster Topology (E1) ANOVA Summary... 100
6. Cluster Topology (E1) Table of Means 102
7. Tandem Topology (E2) ANOVA Summary 104
8. Tandem Topology (E2) Table of Means... 106
9 Fork Topology (E3) ANOVA Summary 109
10. Fork Topology (E3) Table of Means 110
11. Join Topology (E4) ANOVA Summary 113
12. Join Topology (E4) Table of Means 114
13. Fork+Join Topology (E5) ANOVA Summary.. 116
14. Fork+Join Topology (E5) for Forward Communication Protocol (C=0) Table of Means 117
15. Fork+Join Topology (E5) for F+B Communication Protocol (C=1) Table of Means 117
16. Tandem Topology (E6) ANOVA Summary 123
17. Tandem Topology (E6) for Forward Communication Protocol (C=0) Table of Means 124
18. Tandem Topology (E6) for F+B Communication Protocol (C=1) Table of Means 124

LIST OF FIGURES

Figure Page

1. On-line Simulation Support for Production Control 2
2. Derivation of Five Submodel Configurations from a System Model........ 19
3. Speedup and Efficiency of the Parallel Implementation of PDES 20
4 Optimality of the Total Load on Each Processor in PDES 22
5. Actor Code for "range-prod" Function 26
6. Concurrent Computation Structure of the Factorial Problem Execution... 27
7. Representation of the System in Space-Time Rectangle 34
8 Space-Time Rectangle for Different PDES Techniques........................... 36
9. A Generalized Manufacturing System Model used for Experimentation.. 53
10. Types of Networks Considered for Experimentation 55
11. Thirty Two Node Network for "Clusters" and "Tandem" Topologies 55
12. Networks for "Join", "Fork" and "Fork+Join" Topologies 57
13. Modeling Objects of the Manufacturing System.................................... 59
14. PDES Model Execution on "n" Hypercube Nodes 65
15. Bounded Buffer in a Producer-Consumer System 68
16. Program Code for Class BoundedBuffer 69
17. Program Code for Class Producer 70
18. Program Code for Class Consumer 71
19. Execution Analysis of the Producer-Consumer System Model 74
20. Uniprocessor Discrete Event Simulation 77
21. Submodel Simulation Execution Logic 79

Figure Page

22. Manufacturing System Model Example 84

23. Processing of the Arrival Message 85

24. C++ Code for "forward" Communication Protocol 87

25. C++ Code for "forward+backward" Communication Protocol 89

26. Execution Performance Comparison of Communication Protocols 90

27. Manufacturing System Example for Environment Verification 93

28. Shell Script Execution Sequence 97

29. Speedup Curves for "Cluster" Topology (E1) 102

30. Efficiency Curves for "Cluster" Topology (E1) 103

31. Speedup Curves for "Tandem" Topology (E2) 107

32. Efficiency Curves for "Tandem" Topology (E2) 107

33. Speedup Curves for "Fork" Topology (E3) 111

34. Efficiency Curves for "Fork" Topology (E3) 111

35. Speedup Curves for "Join" Topology (E4) 114

36. Efficiency Curves for "Join" Topology (E4) 115

37. Speedup Curves for "F+J" Topology Using "forward" Protocol (E5) 119

38. Speedup Curves for "Fork+Join" Topology Using "f+b" Protocol (E5) 119

39. Efficiency Curves for "F+J" Topology Using "forward" Protocol (E5) 120

40. Efficiency Curves for "F+J" Topology Using "f+b" Protocol (E5) 120

41. Speedup Curves for "Tandem" Topology Using "forward" Protocol (E6) 125

42. Speedup Curves for "Tandem" Topology Using "f+b" Protocol (E6) 125

43. Efficiency Curves for "Tandem" Topology Using "forward" Protocol (E6) 126

44. Efficiency Curves for "Tandem" Topology Using "f+b" Protocol (E6) 126

CHAPTER I

INTRODUCTION

Motivation for Research

Discrete-event simulation is often characterized as a "What-If" tool, capable of giving detailed answers to questions of the form "What will be the effect on the system performance if this change is made?". It provides a framework to make detailed design decisions before the system becomes operational. Recently simulation has become increasingly attractive as today's faster computers, electronic data collection facilities and integrated databases allow relatively quick access to an enormous amount of data typically needed for complex simulation models. Thus, discrete event simulation is becoming one of the most important design and analysis tools for complex manufacturing systems. However, discrete event simulation is not without its disadvantages. An elaborate model generation process and long execution times are some major issues that hinder widespread usage of discrete event simulation as a tool for system analysis and design.

Traditionally, simulation has been used in an off-line mode to support "once per design" decisions. The choices made then are established for the life of the system. Execution times for complex simulation models are frequently measured in minutes or hours. If many alternative system configurations are evaluated or search based optimization is pursued, total lead time to arrive at an optimal decision may run in weeks or months. Thus, discrete event simulation in an off-line mode is still a problem.

Interest is growing in expanding the range of application of simulation, using it in a more on-line role to support regular and repetitive decision making throughout the operational life of a manufacturing system [Rogers 91]. This type of repetitive decision making can be regarded as being performed by a hypothetical manufacturing controller (management or production control department in reality) operating on the manufacturing system. Figure 1 depicts a simplified framework for production control of a manufacturing system.

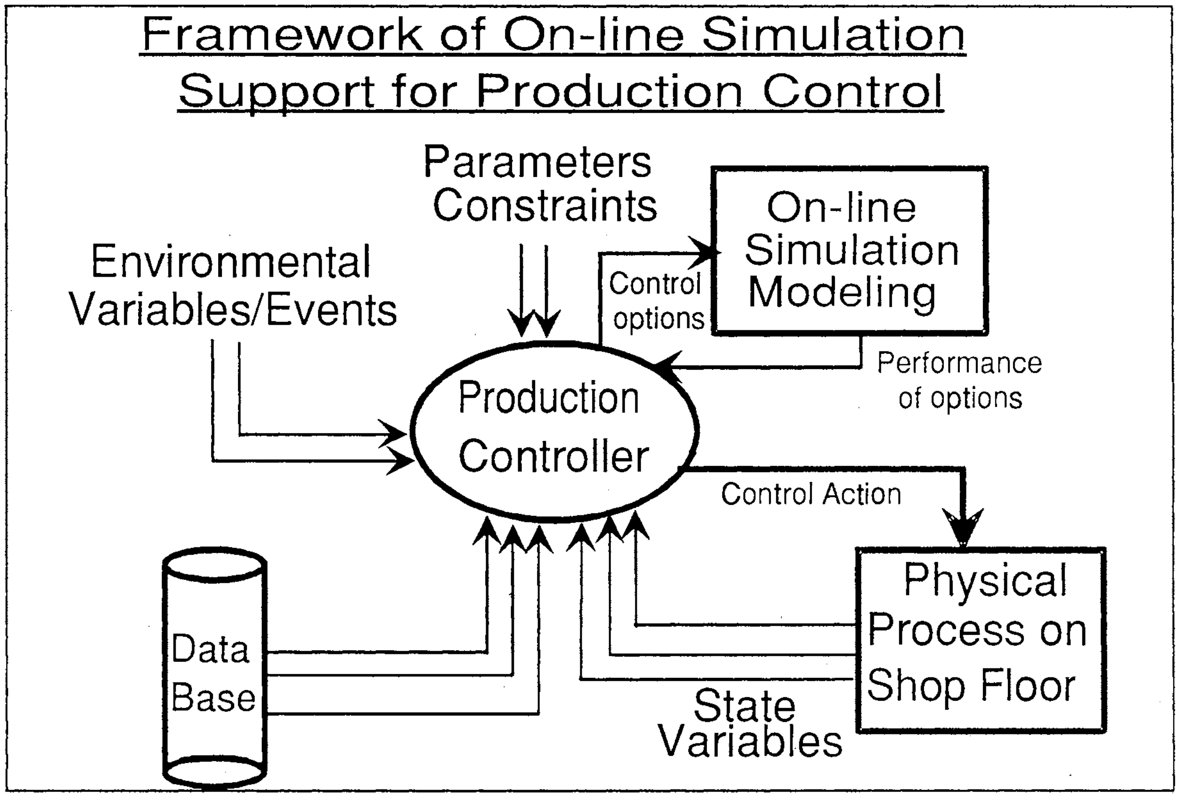
Framework of On-line Simulation Support for Production Control
Parameters Constraints
On-line Simulation Modeling
Environmental Variables/Events
Control options
Performance of options
Production Controller
Control Action
Data Base
Physical Process on Shop Floor
State Variables

Figure 1. On-line Simulation Support for Production Control

A management information system reports the values of internal state variables, external environmental variables, parameters, and constraints and other database information to the production controller. On the basis of this information the production controller controls the manufacturing system. Since the state of the system is constantly changing, the controller must take action in a reasonably short time. An extended lag in the control action can be devastating to the manufacturing system performance as the current solution may quickly become outdated. This requires quick evaluation of alternative system decisions to arrive at an optimal system configuration or control scheme. However, in reality discrete event simulation takes long model execution time. This means that discrete event simulation can be a sound candidate for evaluating alternative real time control strategies, only if the simulation execution times of the system models are considerably reduced. Reduction of the simulation time of manufacturing system models is one of the primary motivations of this research. Over the last thirteen years researchers have tried several approaches to improve the computational performance of discrete event simulation. Some of the distinct approaches include vectorization techniques by Chandak and Browne [Chandak 1983], functional decomposition approach by Comfort [Comfort 1984], execution of independent trials on different processors by Biles et al. [Biles 1985], fast simulation approach by Chen et al. [Chen 90] which replaces the event calendar by recursive mathematical system equations, and distributed simulation. This research proposes to use distributed simulation, which takes advantage of parallel processing technology, to simulate a discrete event model on multiple processors. The distributed simulation paradigm utilizes the inherent parallelism in manufacturing systems to construct smaller "chunks" of the model that can be executed concurrently on parallel processors. Thus, the simulation execution time is reduced by applying a number of processors to simultaneously execute the discrete event simulation model.

As the manufacturing system is reconfigured, the simulation analysis requires a quick and accurate corresponding change in the simulation model of the system. This process is usually repetitive and entails continuous model updating and maintenance. If the model is written in traditional programming languages, the updating process can be quite cumbersome. What is needed is a superior modeling paradigm which allows quick but structured changes to the model. The specification of a concurrent, object oriented paradigm is a secondary motivation for the research. The proposed research uses object oriented modeling concepts that allow very quick model changes based on the corresponding system reconfigurations [Beaumariage 90].

In retrospect, the proposed research addresses two fundamental issues, rapidly modifiable system models and fast discrete event simulation model execution schemes. It considers an object oriented modeling paradigm for quickly reconfigurable models and distributed processing for faster simulation model execution.

Overview of the Dissertation

The remainder of this dissertation is presented in ten chapters, and a bibliography. Chapter II provides a detailed description of the problem statement. Chapter III reviews the literature relevant to this research. This includes literature related to object oriented modeling of manufacturing systems, distributed simulation, concurrent object oriented programming, and processor synchronization mechanisms. Chapter IV presents the statement of research by outlining the research goals and objectives. Chapter V discusses the research methodology including research plan, performance measures and experimental scenarios. Chapter VI provides a brief overview of the object oriented modeling environment created for the research experimentation. Chapter VII evaluates concurrent object oriented programming constructs for parallel discrete event simulation. Chapter VIII discusses the methodology for submodel creation, and Chapter IX presents the process of designing communication protocols. Chapter X focuses on the results of the simulation experiments. Chapter XI is the summary and conclusions chapter that summarizes the research effort and suggests directions for further research. The seven appendices provide supporting material including listings of computer programs and explanation for the choices and discrepancies in several areas.

CHAPTER II

PROBLEM STATEMENT

Introduction

As explained in Chapter I the fundamental focus of this research is to take advantage of emerging technologies such as parallel processing and object oriented modeling, to create a simulation environment that fulfills two major requirements of an on-line simulation support system for the production control of manufacturing systems. This research will not only demonstrate the viability of the application of parallel processing for discrete event simulation, but also attempt to assess the efficiency aspects of parallel discrete event simulation.

A clear and precise determination of the problem statement for this research involves understanding issues primarily from two areas. Firstly, the exact computational requirements of the simulation effort and secondly, the parallel implementation complexities of the computation. The determination of exact computational requirements involves understanding the properties of discrete systems, simulation specifications for discrete systems, nature of manufacturing systems, and the use of simulation for obtaining important performance measures of manufacturing systems. In the implementation area, use of object oriented modeling for representation of manufacturing system models, concurrent object oriented programming constructs for taking advantage of concurrencies in the simulation applications, implementation complexities of parallel discrete event simulation, and communication requirements of distributed computing systems, are addressed in the following sections of this chapter.

Simulation of Discrete Systems

A system is defined as a collection of items from a circumscribed sector of reality that is the object of study or interest [Pritsker 79]. A model is a description of a system. Models are used primarily for describing, designing, and analyzing real world systems. Abstract models represent the system in terms of mathematical-logical variables, equations and relationships. The state of a system is defined as the collection of variables necessary to describe a system at a particular time. The behavior of a system over time is described by a sequence of state transitions. When state transitions occur only at discrete points in time the system is said to be a discrete system. Simulation of discrete systems is typically known as discrete event simulation. The behavior of discrete systems such as manufacturing is a result of highly concurrent and independent, cooperative, or contentious activities of their components. Although the modeling and simulation principles can be equally effectively used for any type of discrete system, for the purpose of this research only manufacturing systems have been considered.

In simulation modeling of a manufacturing system, it is necessary to develop an adequate formalism in which various concurrent activities and interactions of the manufacturing system components can be expressed naturally. At the same time, the model must be executable as a concurrent computer program [Yonezawa 88]. This strategy improves the analysis efficiency as the concurrent paradigm provides close resemblance to the truly concurrent real world systems and concurrent execution enables faster solution. Object oriented languages offer a natural way for the modeling of systems, but these models are typically created for nonconcurrent execution. For example in Smalltalk-80 [Goldberg 89] the concurrency is emulated via instances of class "Process" and a single virtual machine processor. There are two main problems considered for the purpose of this research. Firstly, the inadequacy of many contemporary modeling techniques to naturally model the cooperative concurrent processes in a discrete system and, secondly, the nonparallel execution of discrete system models.

Simulation in Manufacturing Context

A manufacturing enterprise is an excellent example of a system. It is a collection of interdependent elements (physical components, information components and control / decision policies) that work together to achieve a set of common but continuously changing organizational goals. A wide range of alternative techniques is available for the design and analysis of manufacturing systems. The spectrum of alternatives ranges from analytical modeling to direct experimentation on the real system. Today's complex manufacturing systems are not easily describable in terms of analytical models and direct experimentation on the real system is typically costly. This usually makes simulation the analysis tool of choice. Law [Law 86] has pointed out the following three fundamental reasons for the simulation of manufacturing systems:

(1) Determining resource requirements, such as number and type of machines, material handlers and support equipment, factory layout, location and sizes of the work-in-process buffers, manpower requirements, evaluation of capital investments, etc.

(2) Performance evaluation, which typically includes throughput analysis, bottle-neck station analysis or makespan analysis.

(3) Operational policy evaluation, which involves the comparative analysis of a number of policies or procedures designed to solve the same problem. These policies are generally in the area of production control or scheduling, WIP inventory levels, FMS cell control polices, etc.

Simulation modeling of a discrete system requires conceptual frameworks or paradigms to guide the modeler in creating a valid representation of a system in the form of a model. A number of such paradigms or "world views" are available today. A survey paper [Derrick 89] describes thirteen different conceptual frameworks or paradigms. Of the four groups of discrete event simulation paradigms described in the paper, only two are useful from this research point of view. They are the classical or historical framework and the new emerging paradigms.

In the classical or "conventional" approach the process of model building involves the following steps:

(1) Definition of the problem statement, context, symptoms, model purpose, etc.

(2) Determination of system boundary, level of abstraction, state variables, internal model structures, experimental design, etc.

(3) Model coding using typical general purpose languages such as GPSS and SLAM, then verification and validation.

(4) Analysis of results and further experimentation.

In this inherently top down approach, the system boundary, the level of abstraction, model coding and validation are directly derived from a specific problem and context. This may cause several difficulties in modifying, altering or changing simulation models of the same system to address a variety of problems or alternative configurations of the same system.

A different approach to model building is to determine the level of abstraction, system boundary and validation even before the model building process begins. This arrangement requires a significant effort up front that pays off later because the model can be modified easily to solve multiple future problems or system reconfigurations. This approach typically uses an Object Oriented Paradigm to generate the system models, and hence it is called Object Oriented Modeling.

Object Oriented Modeling

The advent of object oriented programming (OOP), a paradigm in which all the program variables are represented as "objects", appears to be a significant advancement towards the development of multiple use, general purpose models [Pratt 1991]. In OOP an *object* is a collection of private data (instance and class variables) and behaviors (methods). There are four key concepts in OOP languages: encapsulation, message passing, dynamic binding and inheritance. Encapsulation is the confinement of data in modules, typically the objects, and restriction of access only to the pre-defined methods. In OOP, procedure calls are typically known as messages. In response to a message, the object executes the requested method (if appropriate) and returns another message, if so desired. Since each procedure or method has to belong to an object, it is required to be referenced through the object. This means that the procedure to be invoked is object specific and hence determined not at compilation time, but at run time. This is known as dynamic binding. This behavior of method execution is significantly different from the familiar "function calls" in conventional non-OOP languages. Inheritance allows the definition of a class of objects to be made by indicating that the new class is just like an existing class but different only in the specified way. The new classes inherit the complete behavior of their super classes along with the additional traits defined for them in their own methods. This process precludes the necessity of rewriting of code for new classes to emulate the behavior similar to existing classes. These four properties of OOP languages can be effectively utilized to create multiple use, general purpose models as explained below.

The object oriented approach to modeling decomposes the system based on the concept of object. Instead of factoring the system based on the modules that denote functions, the system model is structured around objects corresponding to those that exist in reality. The modeling process in the object oriented paradigm starts with the creation of objects or databases of objects in the manufacturing system under study. Typically the OOM approach uses bottom up model building [Pratt 1992]. That is, the objects at the lowest level in the hierarchical structures are defined first. These primary objects (called the modeling primitives) are then used to define other coupled objects in the system.

Thus the entire system model is created by using the primitives as building blocks. Unlike the conven-tional approach there are no supporting guidelines (such as problem statement) to arrive at an adequate level of abstraction or definition of primitives. In order to have maximum flexibility it is desired that the objects should be defined at their lowest possible level so that all the behaviors of potential interest are captured. For example, in manufacturing systems, physical entities such as machines could be described by using atoms and molecules as the primitive objects. However, such an extreme level of primitives may not offer any extra advantages and in fact may turn out to be a big problem in terms of memory and run time. Hence, the primitives are normally set at the level which captures the required "observable behavior". Furthermore, if for any reason it is required to break the primitive down one or more levels, the object oriented paradigm can still take care of such situations by appropriate modifications in the class hierarchy. The simulation and statistics collection logic is typically handled by specially created simulation and statistics collection classes. Even in the context of OOM a number of world views can be used to create the model simulator classes.

With respect to simulation modeling there are several advantages of using the object oriented paradigm. Several researchers [Adiga 89] [Beaumariage 90] have enumerated these advantages in great depth. In general, OOP can create reusable models. Quick modifications and reusability of software code is further improved by class inheritance and by combining primitives to form coupled models. The readily available database of modeling primitives and coupled models is an added attraction.

There are several important points to remember when dealing with object oriented programming. Reusability of code is meaningful only if the generated code is good. Errors in a reused portion of code can have disastrous effect on the simulation execution. Reusable models are not easy to create. There could be a number of model consistency problems if the previous modeling approach cannot accommodate the behavior of a new subsystem. Reusability almost forces the user not to look into the implementation details of a primitive and hence a badly designed primitive, though an irritant, cannot be easily rewritten as there may be a number of ties with other classes.

In typical object oriented languages like Smalltalk-80 [Goldberg 89] message passing has similar semantics to a procedure call, and the computation of the message is done sequentially. The entire Smalltalk-80 environment is based on the paradigm of virtual machine, which sequentially executes the queued processes [Lalonde 1991]. In order to describe a problem such as discrete event simulation that contains concurrency, a notion of process has been introduced. A process is created by sending a fork message to a block context. However, this decision eventually imposes upon the programmer the cumbersome task of modeling the problem in two different level modules: objects and processes. This impairs the description and understandability. If it is desired to take advantage of the concurrency found in discrete event simulation the problem has to be modeled as a set of cooperating processes. Hence, discrete event simulation can be modeled more naturally if the objects are not only maintained as self contained modules but also as units of concurrent execution. This idea will be further explored in the following section.

Concurrent Object Oriented Programming

Concurrent object oriented programming is a programming and modeling methodology in which a system is modeled as a collection of concurrently executable program modules, called objects, that interact with one another by sending messages. As Yonezawa et al. [Yonezawa 88] has pointed out, the motivation for the research on this metho-dology stems from the need to design powerful yet flexible computer software systems that satisfy the ever growing demand of computer systems to solve more complex problems and provide more sophisticated services required in today's information intensive society. He further adds that even though powerful and faster hardware is being produced at a reasonable cost, a traditional use of this hardware will not create computer systems that are capable of satisfying such demands. What is required is the exploitation of parallelism by using a large number of computing agents created from multiple computers that make these computing agents work cooperatively. Exploitation of parallelism is very attractive, but it is not an easy task. The major difficulty arises from the fact that the modeling systems require a wider variety of interactions and a higher degree of concurrency among their system components. The central issue in exploitation of parallelism is what and whose activities should be carried out in parallel and how such concurrent activities should interact with one another. In designing software systems that exploit parallelism, it is required to find how the system model should be broken into components that can be activated in parallel and how to provide coordination between these components. The decomposition should be naturally modular. These natural modeling and modularity concepts fit exactly into the object oriented paradigm discussed in the previous section. Thus, object oriented, concurrent programming combines the concepts of objects and the exploitation of parallelism to create a paradigm that can be effectively used for representation and modeling of concurrent systems.

In uniprocessor object oriented programming, a problem is modeled as a set of cooperating objects and is solved by exchanging messages among objects. In concurrent programming, a problem can modeled as a set of cooperating processes [Yokote 88]. Therefore, object oriented computing and concurrent programming have a very similar structure; objects correspond to the processes, and message passing corresponds to inter-process communication. A process is not necessarily a self-contained module. However, from the viewpoint of modular programming a process is created as a self-contained unit of concurrent execution. Thus, similar to object oriented programming, the object oriented concurrent programming paradigm treats everything as an object which is also a self contained process. In general, this arrangement adds to the overhead of process scheduling, which can be minimized by detecting most of the static dependencies at compilation time. Thus, the object oriented concurrent programming paradigm unifies objects and processes.

Parallel Discrete Event Simulation

Before discussing the issues related to parallel discrete event simulation, it is imperative to explain one major property of physical systems called the causality principle. In simple words, the causality principle states that the future cannot affect the past. All physical systems obey the causality principle. In a discrete system, if an event has some effect on another event then the former must always occur before the later; in other words, cause must precede the effect. Events having no direct or indirect relationships do not require such sequencing constraints and they need not occur in a prescribed order. The order in which the simulator processes the events must obey the causality principle. For example, if event A occurring at time 7 has some influence on event B occurring at time 15 then the simulator must process event A before event B. Thus, for the purpose of execution, it is important to maintain a proper sequence of events. In cases where the events are not processed by the simulator in a correct sequence, the causality principle gets violated and the simulation produces erroneous results.

In uniprocessor application, causality is easily ensured by the ordering of events in increasing simulation time sequence and always following the rule that the event having the smallest occurence time is processed next. The simulation program repeatedly removes the next event from the event list and calls the procedure that models the changes induced by that event. This procedure updates the state variables to reflect the new change of state and schedules any new events as needed. This process is repeated until the event list is empty or an "end of simulation" event is processed. Thus, it is quite easy to satisfy the causality principle in uniprocessor implementation. On the contrary, in parallel implementation it is much more difficult to avoid violation of the causality principle because many events are executed concurrently. Preservation of the causality principle is one of the root problems in a successful parallel implementation of a discrete event simulation program.

Parallel Discrete Event Simulation (PDES) is the execution of discrete event simulation on a parallel processor. This requires partitioning of the simulation model into distinct units, which are executed on different processors. Thus, a global model of the system is partitioned into a collection of smaller local models.

In a uniprocessor simulation program, time in the physical system is modeled by a global variable simulation time or clock. In parallel simulation, this single clock variable has to be replaced by a distributed clock and at the same time the partial ordering of the events imposed by causality in the physical system should not be violated. Two approaches have been developed to satisfy the above requirement. In the first, a global clock is used to ensure that all the processes advance together in lock step, making it a time driven simulation. The global clock process waits until all the activities at the current time are completed. The clock is then advanced to the next time step. It is easy to see that this method guarantees that the causality principle is not violated, however, its usefulness is quite limited to situations in which a number of events have the same event times. Otherwise, most processes would lie idle while waiting for the simulation clock to advance. The second approach, referred to as event driven method, provides each logical process its own local clock. Each logical process is responsible for advancing its own clock as the simulation proceeds. The clocks in different processes are advanced at completely different rates, which eliminates the need for a process to wait for the processes to which it is not directly or indirectly related. Thus, this approach eliminates the problem associated with the time driven simulation.

In the event driven approach, each process can receive a time stamp message or event from several other processes. To maintain the local causality, it is necessary to ensure that these messages are processed in increasing time stamp order. The local causality requires non-decreasing event order only within a process, but it does not constrain a collection of related processes to collectively process messages in non-decreasing event order. It allows any logical process to get ahead of another as long as the sequencing constraints within each process are not violated.

Adherence to local causality in each process is sufficient to ensure global causality if all interactions between processes are only through time stamp messages. This is because a violation in causality can only occur when an event A with time stamp T_A has a direct effect on event B with time stamp $T_{B,}$, where $T_A < T_B$, but the simulator erroneously processes B before it processes A. In a simulation program, event A can affect event B in two ways:

1. Event A causes the creation of event B
2. Event A modifies the state variables used by event B

In the first case, event B cannot be processed before event A because event B is created only while executing event A. The second case suggests that events A and B should belong to the same process because the state variables are local to each process and cannot be accessed by other processes. If local causality is assured then it is again not possible for the events to be processed out of order. Therefore, adherence to local causality ensures that no causality is violated globally. It is important to note that the above statement is valid only if all the interactions between the processes are via event messages, and the processes do not have any shared global variables.

Simultaneous satisfaction of the local causality constraints for all the processes which are advancing in simulation time at different rates lead to a problem called processor 'deadlock'. The deadlocking situation occurs when in order to maintain local causality, process A waits for process B which is waiting for process A itself. Thus both processes wait for each other indefinitely. Under these circumstances the two concerned processes are said to be deadlocked. As mentioned above, the concurrent satisfaction of the local causality constraints for all the processes impose a deadlock problem. Deadlock situations should either be avoided, or detected and eliminated. In the literature, a number of researchers [Chandy 81] [Chandy 89b] [Renolds 82] have proposed a variety of algorithms to perform deadlock-free discrete event simulation. These techniques fall into three major categories [Chandrasekaran 87]:

1] Avoid deadlock by generating "NULL" messages to distribute the simulated time across the neighboring processes determined a priori by the event dependencies.

2] Allow the deadlock to occur, but provide a mechanism to detect and recover from deadlock situations.

3] Avoid deadlock by allowing processes to process the events on any non-empty queue, regardless of the number of other input queues that are empty. This can lead to violation of local causality, so an additional rollback mechanism is provided to undo erroneous computations and return to some point before the causality constraint was violated, giving the process another chance to perform the computation correctly.

As mentioned before, a number of schemes have been developed to deal with the problems involved in the implementation of parallel discrete event simulation. The exact details of some of them are considered in the literature review chapter.

The above sections have provided a general introduction to various aspects of the problem domain. The next section attempts to describe the specific problem statement and outline the solution requirements.

Problem Description

This research is specifically focused at the demonstration of the viability of parallel processing for discrete event simulation of manufacturing systems. It is designed to obtain more insight into an efficient execution performance of PDES. As the modeling of manufacturing systems is not the primary focus of this research, a relatively simple model of manufacturing system has been chosen for the purpose of this research. The exact specification of this model is described in Chapter V. As established earlier, quick reconfiguration and high reusability of modeling elements are some of the essential requirements of the simulation model. This concept of model reusability in object oriented modeling and simulation environments is well documented in the literature [Adiga 89] [Basnet 90]. In fact, several researchers [Adiga 89] [Pratt 91] have demonstrated the ability of the object oriented modeling paradigm to create rapidly modifiable and reusable simulation models. Therefore, an object oriented paradigm is used for implementing the modeling and simulation environment. The choice of C++ is further justified as it is the only object oriented language currently available on the Intel iPSC/2 computer, a parallel processing computer at the Oklahoma State University. Details of C++ object oriented implementation of modeling and simulation environment are outlined in Chapter VI.

The Intel iPSC/2 computer is a distributed memory MIMD (Multiple Instructions Multiple Data) machine in which the inter-processor communication is performed via message passing between the individual processors. Therefore, parallel discrete event simulation is achieved by dividing the simulation model of a manufacturing system into several submodels and allowing each processor to execute a single submodel.

Figure 2 depicts a number of ways in which these submodels can be created for a particular manufacturing system. Each distinct way results in a single configuration of the submodels that collectively constitute the entire model. The flow of parts (work flow items) between the machines belonging to two different submodels creates communication messages between the processors. Thus, combined part routings of a manufacturing system and thereby the manufacturing system network topology influence the communication pattern between the processors. Typically, all parts in a manufacturing system have their own independent process routings. By superimposing all the part routings, a generalized routing network for the entire manufacturing system can be created. On the basis of this network, a specific network of submodels depicting their interprocessor communication patterns can be derived such that each node of the network represents a single submodel and the communication between two submodels is represented by an arc between the corresponding nodes of the network. This network of submodels is defined as a "submodel network".

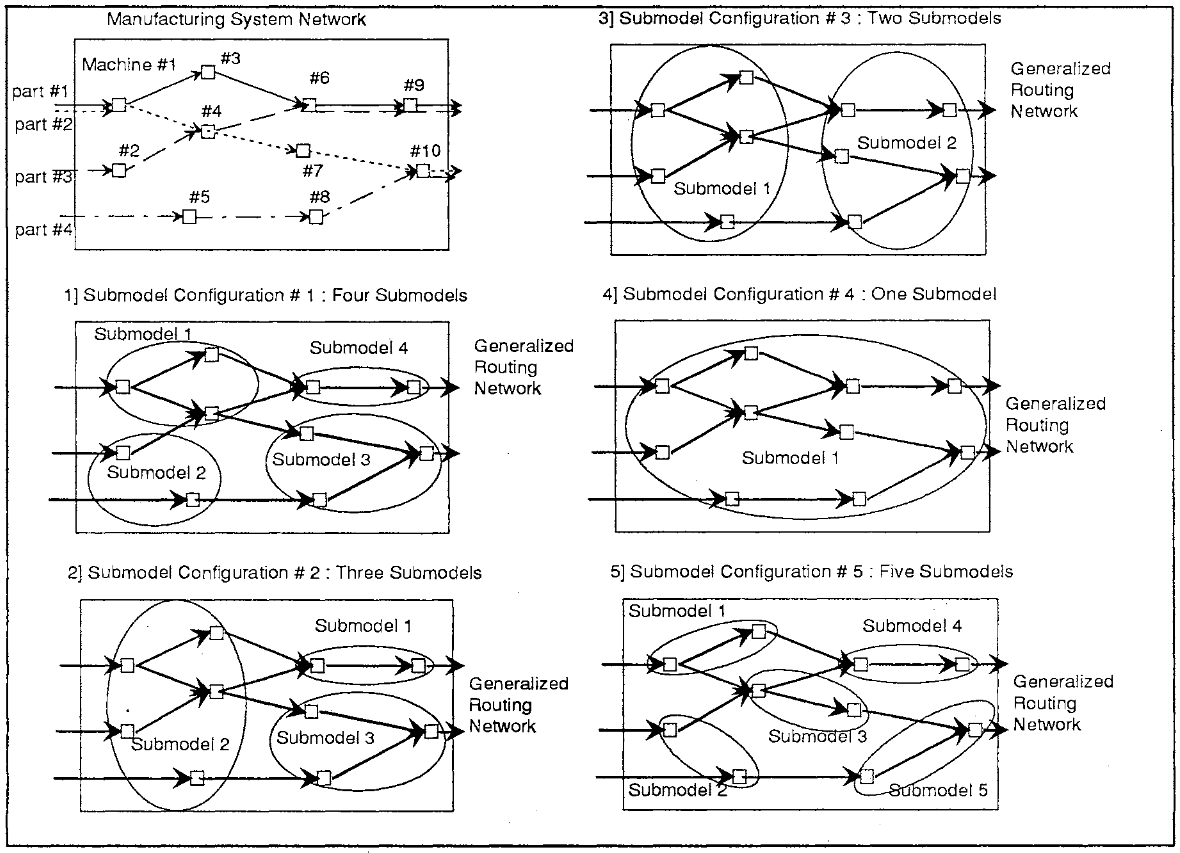
Manufacturing System Network
Machine #1
#2
#3
#4
#5
#6
#7
#8
#9
#10
part #1
part #2
part #3
part #4
1] Submodel Configuration # 1 : Four Submodels
Submodel 1
Submodel 2
Submodel 3
Submodel 4
Generalized Routing Network
2] Submodel Configuration # 2 : Three Submodels
Submodel 1
Submodel 2
Submodel 3
Generalized Routing Network
3] Submodel Configuration # 3 : Two Submodels
Submodel 1
Submodel 2
Generalized Routing Network
4] Submodel Configuration # 4 : One Submodel
Submodel 1
Generalized Routing Network
5] Submodel Configuration # 5 : Five Submodels
Submodel 1
Submodel 2
Submodel 3
Submodel 4
Submodel 5
Generalized Routing Network

Figure 2. Derivation of Five Submodel Configurations from a System Model

In implementing parallel discrete event simulation, each submodel of the submodel network is simulated by a single processor. This arrangement allows certain processors to go ahead of others in simulated time. That is, a processor may be executing a future event when an event in its past arrives from another processor. Under these circumstances the past event may execute differently because of the earlier execution of the future events. This situation allows the future to affect the past and therefore violates the causality principle. Unless special mechanisms typically known as "processor synchronization mechanisms" are provided to trap such inconsistencies, the simulation results would be erroneous. Hence, provision of a suitable processor synchronization mechanism is another problem that must be resolved for a successful implementation of a parallel discrete event simulation.

Although communication synchronizes the two submodel simulation processes running on two distinct processors, it adds an unnecessary communication burden to these processors, thereby degrading their execution performance. The performance of a software application is generally measured in speedup and efficiency factors. Figure 3 depicts the definitions of speedup and efficiency.

$$\text{Speed up} = \frac{\text{execution time for uni-processor implementation of the problem}}{\text{execution time for a parallel implementation}}$$

$$\text{Efficiency} = \frac{\text{speedup}}{\text{number of processors used in the parallel implementation}}$$

Figure 3. Speedup and Efficiency of the Parallel Implementation of DES

The actual simulation execution load on each processor of the Intel iPSC/2 computer is called the "computation load". As explained earlier, the parallelization of discrete event simulation also requires each processor to communicate with other processors. This is called the "communication load". A parallel application such as parallel discrete event simulation gives better execution performance if the total or combined communication and computation load on each processor is minimized. The communication load on the computer is drastically influenced by the way in which the submodels are created. Three major ways in which the submodels influence the communication between processors are:

1] Simulation of the movement of work flow items between the machines under different submodels of the manufacturing system requires that the corresponding work flow item objects are moved across the processors simulating the corresponding submodels.

2] If the flow of the work flow item objects between processors becomes cyclic the simulation process can potentially deadlock. The only way to avoid or break a deadlock is to add a substantial amount of communication between processors. This communication is generally provided by deadlock detection and recovery algorithms [Bain 88].

3] If two submodels under different processors share a common resource such as a material handler, the causality constraints require the execution of an event on one processor to inquire about the status of the shared resource from the other processor. This introduces additional communication between the processors.

The submodel creation is a key process that influences the communication patterns of the parallel discrete event simulation and hence the speed and the efficiency of an application. Therefore, the major problem at the submodel creation stage is to minimize the communication load while attempting to use as many processors as possible, i.e., to create as many submodels as possible. Since a modeler can obtain several different configurations of the submodels from a single manufacturing system model, the modeler has to choose the submodel configuration that gives the best execution performance. At one end of the spectrum, a uniprocessor application requires virtually no communication but shifts the entire computation load to a single processor, thereby producing suboptimal results in most cases. On the other end of the spectrum, if a large number of processors are used the communication overload can become overwhelming. Thus, either end of the spectrum only results in a suboptimal performance. As depicted in Figure 4 optimality exists somewhere in between.

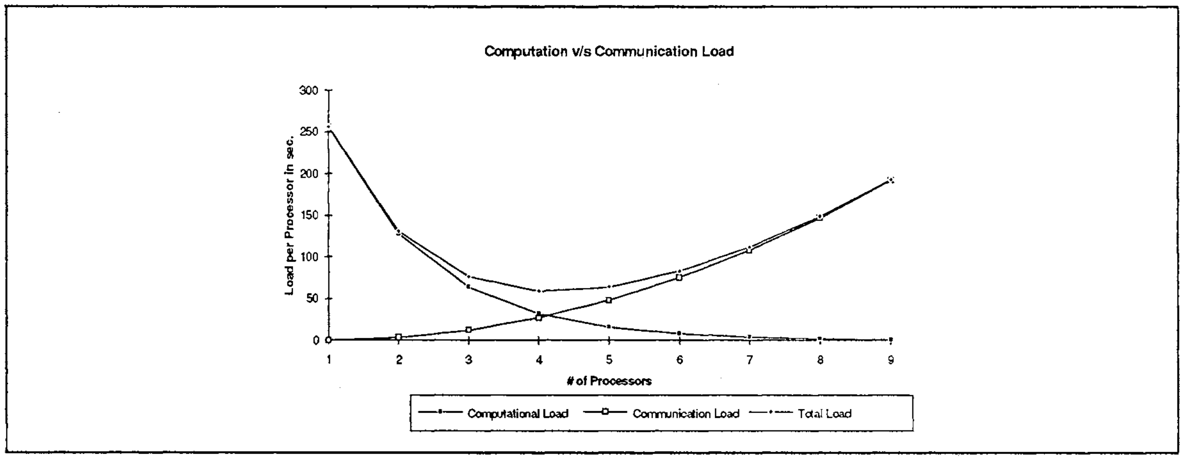

Figure 4. Optimality of the Total Load on each Processor in PDES

Unanswered Questions

The above explanation of the problems involved in object oriented modeling and parallel implementation of discrete event simulation of manufacturing systems poses a number of unanswered questions. Among the unanswered questions are:

1] What factors influence the efficiency of parallel discrete event simulation of manufacturing systems?

2] What impact do these factors have on a parallel implementation of the discrete event simulation of a manufacturing system?

3] What constructs must an object oriented concurrent programming language provide to achieve process synchronization for a parallel application?

4] How can deadlocking situations be avoided in parallel implementation of discrete event system simulations?

5] Can a test be developed to detect if a manufacturing system possesses the characteristics that can lead to a deadlock during parallel implementation of a discrete event simulation of that system?

☆ 6] How can the random number generation process be effectively handled so that it can result in identical simulation results regardless of uniprocessor or multi-processor implementation ?

This research seeks to address these questions and gain insight into a methodology for answering them. Some of the above questions are discussed in the distributed simulation literature. Brief reviews of pertinent research articles are provided in the next chapter.

CHAPTER III

LITERATURE REVIEW

Introduction

This chapter contains a formal introduction of the literature related to object oriented modeling and parallel discrete event simulation (PDES) of manufacturing systems. This review consists of four major areas: concurrent object oriented programming, parallel processing and parallel computing architecture, parallel discrete event simulation, and processor synchronization mechanisms. There is an enormous amount of literature available in all of the above areas. The majority of research in the last two areas has been concentrated on the computer or communication system models. The similarity between the computer/communication systems and manufacturing systems validates the use of this research in the manufacturing systems context.

Concurrent Object Oriented Programming

The inclusion of concurrent object oriented programming aspects in the context of this research comes from the fact that concurrent OOP not only provides modeling constructs in terms of objects, but also takes advantage of the concurrencies in real systems to create models that can be implemented on parallel processing machines. The literature relevant to this research includes research efforts in the development of concurrent OOP languages such as "ABCL/1" by Yonezawa et al. [Yonezawa 88], "ConcurrentSmalltalk" by Yokote et al. [Yokote 88], "POOL-T" by America [America

88], "Orient/84K" by Ishikawa et al. [Ishikawa 88], and "Actor" languages by Agha et al. [Agha 88a].

From a historical perspective, concurrent programming concepts first appeared in the literature in 1977, when Kahn and McQueen [Kahn 77] developed the constructs of streams to capture functional systems. Brock and Ackerman [Brock 81] added the inter-stream ordering information in order to make the 'stream' model more suitable for concurrent computation. Pratt [Pratt 82] formalized the generalized theory of processes in terms of sets of partially ordered multisets (pomsets) of events. This generalized process model is compatible with the laws of concurrent processing formulated by Hewitt et al. [Hewitt 77]. Hoare [Hoare 78] proposed a language for concurrency called CSP, based on sequential processes. In CSP the communication between the processes is synchronous. Along similar lines, the actor model with the existence of a mail system was introduced to enable asynchronous communication between processes. The relevant concepts of the actor model are explored in the following brief summary of the research article by Agha et al. [Agha 88b].

The authors suggest that providing a mechanism for dealing with shared resources, dynamic reconfigurability, and inherent concurrency are the fundamental considerations in designing the actor language. An *actor* is a computational agent that carries out its actions in response to the incoming communication messages. Even though an actor is analogous to an abstract concept of an object, it is distinctly different from an object, in the sense that it also encompasses in itself the notion of process. Unlike the objects created in traditional OOP languages such as Simula [Birtwistle 79] and Smalltalk [Goldberg 89], an actor can transform its behavior dynamically without necessarily being constrained by the restrictions imposed by its membership to its class. In general, an actor can send a message to itself or other actors, create more actors, or specify a replacement behavior to pipeline its actions. All actors in a system carry out their actions concurrently. In response to a communication, an actor may send several communications to other actors. The creation of new actors increases the amount of concurrency feasible in the system. In actuality, an actor is a tuple consisting of a mail address and a current behavior. The mail address is associated with a mail queue for incoming messages. The current behavior is specified in terms of local state functions. An actor system consists of actors along with a set of unprocessed tasks. A configuration is a snapshot of an actor system. A solution to the problem is obtained as the actor system starts at the initial configuration and proceeds through the intermediate configurations to end the processing at the final configuration. The number of actors in the system typically grows as the concurrency in the solution algorithm is dynamically exploited by the computing actors in the intermediate configurations. For example, the function evaluating the factorial of an integer can be implemented as a product of ranges that will be evaluated concurrently. The authors have presented the actor code for the implementation of a factorial algorithm in actor language, as depicted in Figure 5.

```
(defFunction range-prod (lo hi)
        (if (= lo hi)
           (then lo)
           (else (let ((mid (/ (+ hi lo) 2)))
                 (* (range-prod lo mid) (range-prod (+ 1 mid) hi))))))
```

Figure 5. Actor Code for "range-prod" Function

Each actor containing a range-prod function with 'lo' and 'hi' values, creates two more actors with the corresponding two new parameters as 'lo', '(hi+lo)/2' and '(hi+lo)/2', 'hi' and waits for these two actors to send back their respective products. Each of these two actors, in turn, creates two more actors with corresponding 'lo' and 'hi' values. This process of creating actors continues until the current actor has the same values in the 'hi' and 'lo' variables. This structure of the concurrent factorial computation is depicted in Figure 6.

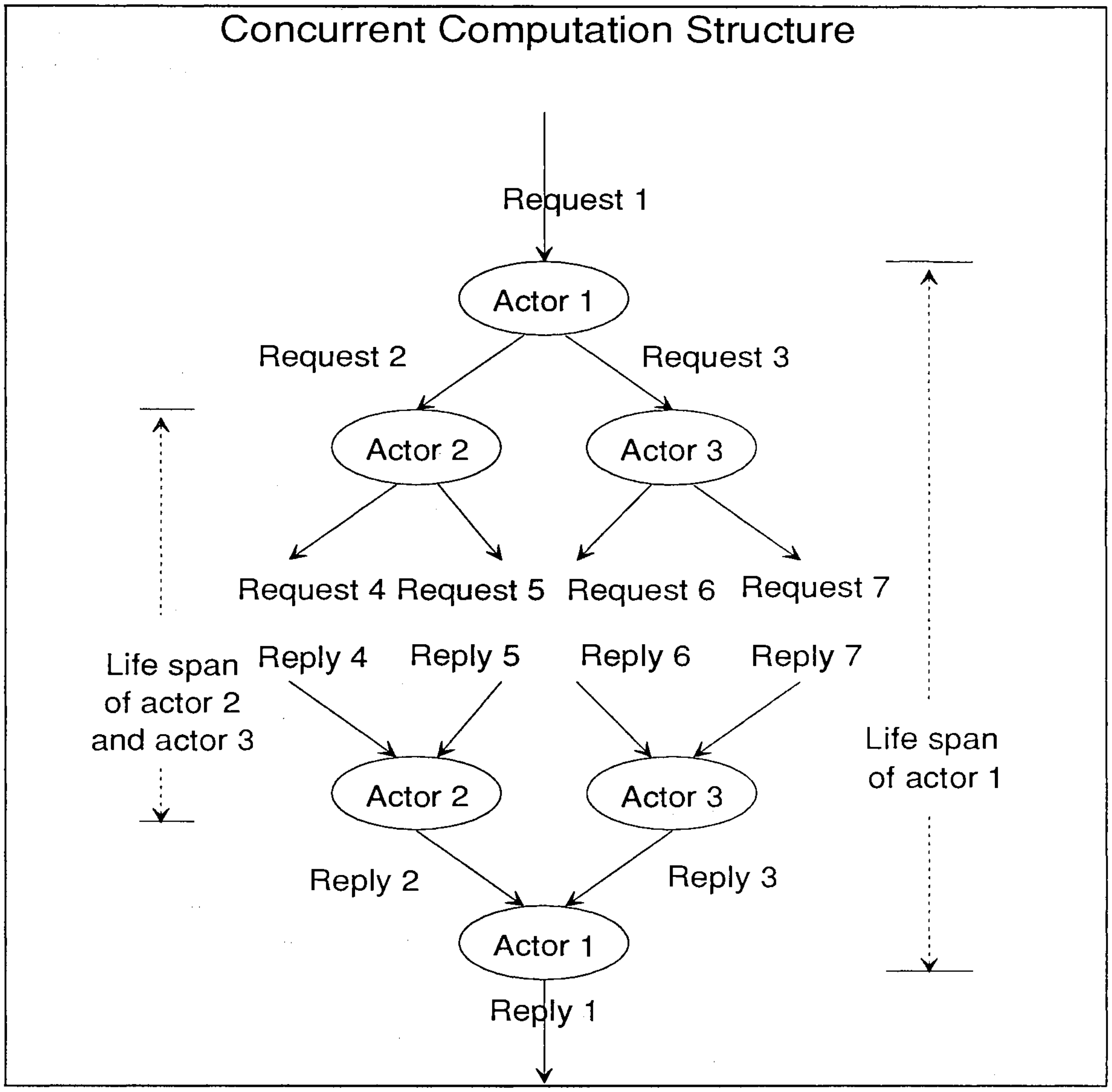

Figure 6. Concurrent Computation Structure of the Factorial Problem Execution

As the replies to its requests are obtained, the actor multiplies the two replies and creates a response message for its parent actor. As the computation progresses upwards in the hierarchy, the excess child actors are destroyed, eventually maintaining only a single actor containing the answer.

In general, the ability of actors to create new actors (the customers) and distribute the work is well suited for fine-grain message passing parallel computers having thousands of small processors with low communication latency. The root compiler of the actor language 'Acore' provides the mechanisms for inter-processor communication and other memory management tasks. 'Acore' allows a number of actors under each 'worker' that executes the actor computation on a single processor. In this way, the actors can be easily distributed on the available processors of a parallel computer.

In retrospect, the Concurrent Object Oriented language Actor provides for dynamic growth and reconfigurations of actors that exploit the parallelism in an open concurrent system in order to effectively implement it on a variety of parallel computers.

Parallel Processing and Parallel Computer Architecture

This section of the literature review provides a brief introduction to parallel processing and parallel architecture, specifically the architecture of the Intel hypercube computer. The majority of the following description is obtained from books by Aki [Aki 89] and Bustard [Bustard 88].

In a typical personal computer, there is only one processing unit, the microprocessor chip. If a computer has more than one processing unit, it is called a parallel computer. The need for parallel computing arises from the fact that in many real time applications, the solution to a problem requires that an enormous amount of computation be performed in a very short period of time. The basic idea behind parallel processing is to divide the given problem into a number of distinct sub-problems that can be solved simultaneously, each on a different processor. The results are then combined to produce the answer to the original problem. This is a radical departure from the algorithmic model of computation, designed for sequential uniprocessor machines.

Over the last forty years uniprocessor computers have achieved a dramatic increase in computation speeds. The main reason for this dramatic increase was the availability of faster electronic hardware components. The computer hardware moved from relays, to vacuum tubes, to transistors, to integrated circuit (ICs) chips, to small and large scale integration (LSI) chips, and then to very large scale integration (VLSI) chips. Unfortunately, it is perceived that this trend may soon come to an end. The limiting factor is simply the law of physics that gives the speed of light in a vacuum. The only way around this problem is to use parallelism in the problem statement and simultaneously solve the sub-problems crafted out of the original problem.

Any computer, sequential or parallel, operates by executing instructions on the supplied data. A stream of instructions (algorithm) tells the computer what to do at each step. A stream of data (the input to the algorithm) is affected by these instructions. Depending on the number of streams of data or instructions, four groups of computers are often defined [Aki 1989]. They are:

1. <u>Single Instruction Single Data Stream (SISD) :</u> This computer does not exhibit any parallelism. A typical personal computer is a good example of this type.
2. <u>Multiple Instruction Single Data Stream (MISD) :</u> The parallelism is achieved by letting the processors simultaneously do different things on the same data stream. Applications of this type include object classification problems.
3. <u>Single Instruction Multiple Data Stream (SIMD) :</u> The parallelism is achieved by dividing the data into multiple streams that are processed simultaneously using the same instructions on multiple processors. Applications of this type include matrix multiplication problems.

4. Multiple Instruction Multiple Data Stream (MIMD) : Computers of this type are the most general and most powerful of the four groups. They can handle parallelism via multiple data and/or multiple instruction streams. They can mimic the behavior of all the other groups.

MIMD machines are further classified as shared memory or distributed memory machines. The shared memory machines share a common memory, and synchronize their operations through this memory. The distributed memory machines have separate individual memory and therefore, they require direct processor communication for synchronization.

MIMD computers allow the execution of asynchronous algorithms. Asynchronous algorithms are difficult to design, evaluate and implement. An asynchronous algorithm is a collection of processes, of which some or all are processed on different processors. The execution of an asynchronous algorithm starts with the creation of computational tasks, or processes, to be performed. Once a process is created, it must be executed on a processor. If a processor is available, the process is assigned to the processor that performs the computation specified by the process. Otherwise, the process is queued and waits for a processor to become free. When a processor completes execution of a process, it checks the process queue. If a process is waiting, it is selected for execution. This process creation and execution continues until all the processes are executed, or any process instructs the processor to stop the execution of the asynchronous algorithm.

In evaluating a parallel algorithm for a given problem, it is typically compared to it's equivalent single processor implementation. A good indication of the quality of a parallel implementation is the speedup it produces. The speedup is defined as

$$\text{Speed up} = \frac{\text{execution time for uni-processor implementation of the problem}}{\text{execution time for a parallel implementation}} \qquad (1)$$

Another indicator of the quality of a parallel implementation is the utilization of each processor or it's efficiency. The efficiency is defined as

$$\text{Efficiency} = \frac{\text{speedup}}{\text{number of processors used in the parallel implementation}} \tag{2}$$

In controlling a parallel computer, a number of tasks must be managed. Typically, the operating system provides additional constructs that manage the memory, inter-processor communication, I/O communication, etc. These tasks are highly dependent on the architecture of the parallel computer. Hence, the following paragraph describes Intel's iPSC/2 architecture, operating system and interconnection synchronization mechanism.

Intel's iPSC/2 concurrent supercomputer is a cost effective solution for large scale applications. In an iPSC/2, a large number of processors or nodes work concurrently on parts of a single problem. An iPSC/2 system consists of computing nodes, I/O nodes, peripherals, and a front end processor called the host. A node is a processor (Intel 80386 chip) and memory combination. Each node runs the NX/2 operating system, uses message passing to communicate to other nodes, and can access both the host file system and the iPSC/2 concurrent file system.

The front-end processor is called the System Resource Manager (SRM). The SRM runs the UNIX System V operating system, augmented with iPSC system extensions and TCP/IP networking software which links the remote work stations and the SRM.

A typical iPSC system application has a host program that runs on the host and a node program that runs on a group of allocated nodes called a 'cube'. The host program runs under the UNIX operating system as one or more processes. It usually initializes the application, provides any necessary human interface, and loads the node program onto the nodes. Each node executes the node program. Typically a node program performs calculations, exchanges the synchronization messages with the other nodes, and sends data back to the host.

The iPSC/2 system at Oklahoma State University has 32 CX nodes connected in the form of a hypercube of dimension 5 (2^5). Even though all the nodes in an iPSC system are not fully connected, a node can send a message directly to any other node in the network without affecting the processing of the nodes in the connecting path. This is achieved via Direct-Connect Module (DCM). The composition of each CX node is described in table 1 below.

TABLE 1

COMPOSITION OF INTEL IPSC/2 HYPERCUBE CX OR SX NODE

Processor	**Memory**	**Numeric Processing**
386 microprocessor	16 Mega bytes of memory	80387 or Waiteck SX processor

The operating system of the iPSC/2 provides a number of user interface commands, such as, *getcube, cubeinfo, relcube, load, attachcube*, etc., to let the user manipulate the resources available on the system. It also provides a variety of synchronous and asynchronous message passing calls. They are synchronous and asynchronous send and receive.

Synchronous send and receive : (csend() / crecv()): A synchronous send means that the program executing the send waits until the send is complete. A synchronous receive means that the program executing the *receive* waits until the message arrives in the specified buffer.

Asynchronous send and receive : (isend() / irecv()): An asynchronous send means that the program executing the *send* does not wait until the send is complete. To make sure that a *send* is complete the *msgwait()* command is used. An asynchronous receive means that the program executing the *receive* initiates the receipt of a message, and when the information is required by the process, uses *msgwait()* to block further execution of the process until the *receive* is complete.

Parallel Discrete Event Simulation

One of the earliest publications in the area of distributed simulation included a case study by Chandy and Misra [Chandy 79]. This case study contains one of the pioneering efforts in the determination of requirements, constraints and program design of distributed simulation. Since then, researchers have provided a number of approaches to solve the distributed simulation problem. Chandy and Sherman [Chandy 89a] have proposed a unified framework called space-time for describing the discrete event simulation problem. This framework provides a foundation for the classification of the numerous research methods and helps identify new research directions in distributed simulation. They have explained the space-time framework with the help of a system containing two billiard balls in a friction free cylinder. Figure 7 shows the space-time rectangular representation of the system. The two identical balls, initially placed at opposite ends of the cylinder, travel towards each other with equal speed. The space-time region of interest is a rectangle where x goes from 0 to L (length of the cylinder) and t goes from 0 to T (required simulation time).

The simulation of the system is a method of filling in the space-time rectangle. As seen in Figure 7, the two balls collide with each other and return back to collide with the end wall. The cylinder surface is assumed frictionless, and hence, the balls follow their back and forth linear motion indefinitely. As the simulation time is continuously increasing, the back and forth motion in space (i.e. along the x axis) results in a zig-zag motion in the space-time rectangle.

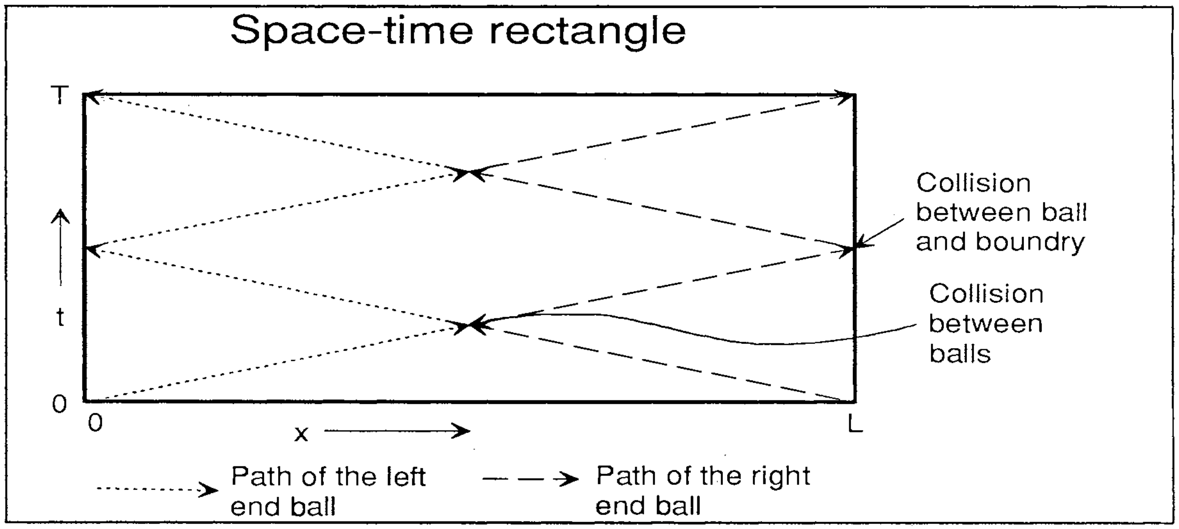
Space-time rectangle
T
t
0
0
x
L
Collision between ball and boundry
Collision between balls
Path of the left end ball
Path of the right end ball

Figure 7. Representation of the System in Space-Time Rectangle

Figure 8 explains the four most prominent approaches to simulation. Figure 8 (a) is the *Time Driven Simulation* in which the program determines the behavior of the system at all times in an interval $[0,T^*]$ and then proceeds to evaluate the system behavior at $T^*+\varepsilon$ for some arbitrarily small ε (most continuous system simulations are performed in this manner). Figure 8(b) shows a different case where the time step ε is much larger than the earlier case. Typically ε is selected to be some important event in the system (such as collision between balls). This is the *Event List Approach* to discrete event simulation. A distributed simulation is characterized by the partitioned space-time rectangle as depicted in Figure 8(c). Each processor evaluates or fills the inner details of its partition. In the conservative distributed simulation technique, for each partition in space, the program determines the behavior of the system at all times from T_i to T_i+S_i as shown in Figure 8(d). Higher values of S_i make the simulation run more efficiently. In an optimistic distributed simulation the system behavior is estimated from T^* to T_i+S_i. If these estimates are found to be incorrect, they are corrected by a roll-back and recovery method. If all the estimates are shown to be correct for some interval T^*+u, then the time is advanced to T^*+u. Both distributed simulation methods use many processing agents that significantly reduce the time required to fill the space-time rectangle or the simulation execution time.

The authors further explain that achieving a successful implementation of discrete event simulation involves a careful partitioning of the system space-time diagram. The space-time diagram contains a vertical line for each process in the system. The synchronizing messages passed between the processes are represented by horizontal arrows connecting the vertical processes. Typically, a collection of processes are run on each processor, which means that a tandem arrangement of processors partition the space-time rectangle into vertical strips. Each strip contains a number of vertical lines, corresponding to the processes it contains.

A number of researchers (such as Fujimoto [Fujimoto 89], Jones et al. [Jones 86], Nevison [Nevison 90], etc.) have devoted their attention to measuring the performance of several strategies for distributed simulation on a number of parallel processing computers.

Comparisons of a number of distributed simulation strategies require the development of proper performance measures. Fujimoto [Fujimoto 89] has developed a number of performance measures to compare the deadlock avoidance and deadlock

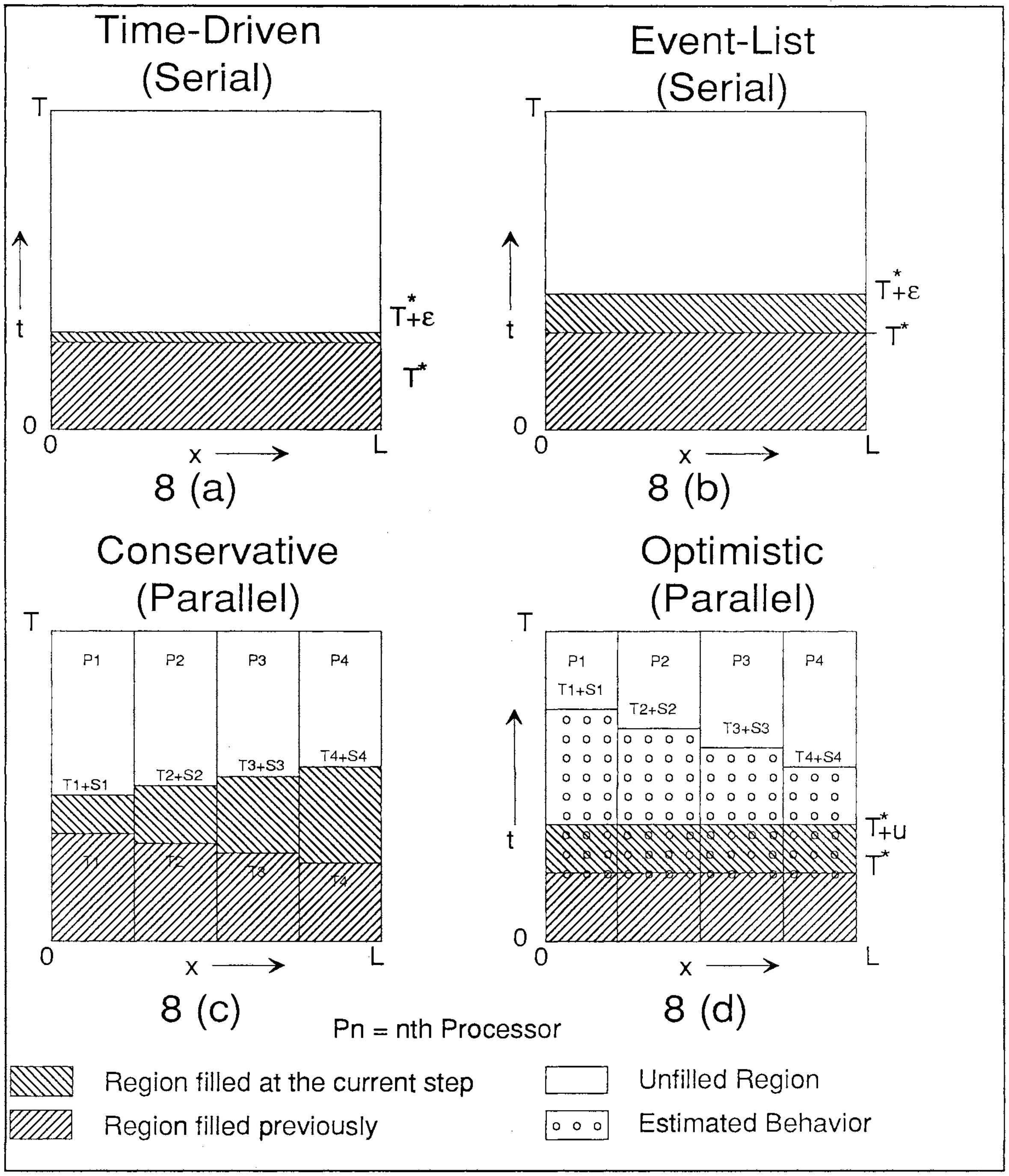

Figure 8. Space-Time Rectangle for Different PDES Techniques

detection and recovery techniques. He used speedup, "null" message ratio, and the deadlock ratio as his performance measures. He used a BBN Butterfly™ multiprocessor for the distributed simulation testbed. In the deadlock avoidance technique, when a process is blocked, it communicates its lookahead function value to its neighboring processes in order to avoid deadlock. In the deadlock detection and recovery technique, a global counter is maintained that holds the number of unblocked processes. In a deadlock situation the counter becomes zero. The recovery process locates the message with smallest time stamp in the system and restarts the corresponding blocked process. The author has performed an extensive empirical performance evaluation of distributed simulation programs and provided the following conclusions.

1. The lookahead ability of logical processes plays a critical role in determining the efficiency of the deadlock detection and recovery algorithms. This is due to the fact that the processes must spend an extensive amount of time waiting if their lookahead ability is poor.
2. Deadlock detection and recovery simulation for moderate to high messages containing different types of logical processes can be adversely affected by a small number of processes that exhibit poor lookahead capability.
3. Networks containing inherently poor lookahead properties, e.g. prioritized queues, are ill suited for both algorithms.
4. Distributed simulation, using deadlock avoidance or deadlock detection and recovery algorithms, is a viable approach to speeding up work loads containing a moderate to high degree of parallelism.
5. In order to judge which performance enhancement approach will be most effective for a particular situation, the programmer must be intimately familiar with the simulation application.

Processor Synchronization Mechanisms

In the parallel simulation literature, a considerable amount of research effort has been directed toward the development of time synchronization mechanisms. Developing a sound synchronization mechanism is vital to successful implementation, because the speedup factor is almost entirely dependent on the network communication load. To date, little empirical data is available for measuring the performance of specific implementations of these algorithms, even though a number of studies have been reported (for example: [Seethalakshmi 79] or [Reed 85]). A notable exception is the recent work on queueing network simulations by Reed, Malony, and McCredie [Reed 90]. The performance of a specific implementation of a time synchronization algorithm is significantly dependent on the architecture of the parallel computer. The experiments involving iPSC/2 have not been explored in detail. From this perspective, a couple of references [Bain 88] and [Davis 90] lie close to the context of this research. The following paragraphs discuss two important research papers having network algorithm implementation on Intel iPSC/2.

William Bain and David Scott [Bain 88] from Intel Science Computers have proposed a conservative deadlock avoidance algorithm. They explain the simple technique (suggested by [Chandy 81]) of feeding time synchronization information to all successors, when an event message is sent to one successor. This technique has a disadvantage of generating an enormous amount of message traffic. The *demand driven* technique requires a process to request time information from its predecessors before advancing its time. This approach minimizes message traffic. Using this approach, an algorithm can be developed to detect and avoid deadlocks in the network.

The authors have outlined a general, deadlock free algorithm for demand driven synchronization. The algorithm guarantees small fixed size messages but it may require a number of requests for the exchange of time information to advance a process's time. A brief explanation of the algorithm is provided in the next paragraph.

Notation:

n :	Number of processes (one on each processor)
t_i :	Distributed clock time of process 'i'
p_i :	The predecessor process of process 'i'
s_i :	The successor process of process 'i'
l_i [j]:	The earliest time that an event can arrive from jth predecessor of process 'i'

When an event t_m arrives at process 'q' at the head of the queue for channel 'j', the process updates the value of l_i[j] (also known as the channel time) to t_m. Before the process can consume and process the contents of the message, it must assure that no message, with time less than t_m, will arrive from any other predecessors. This is the essential constraint of time progressive synchronization. If all processor times exceed t_m, i.e. for all values of j, l_i[j] is greater than or equal to t_m, the event message is consumed immediately. Otherwise, the process requests the channel times from all processors having channel times less than t_m, by placing an entry with the predecessor 'id' and requested time t_m in the *request queue.* The predecessor replies in one of the following three ways:

YES : Which indicates that the predecessor channel has reached the request time, and the new channel time is passed back as a reply.

NO : Which indicates that the predecessor channel has not reached the request time and another request must be made.

RYES : (a reflected 'yes'), which indicates that the channel has conditionally reached the requested time. The RYES answer is provided to detect the cycles within the connection graph and thereby avoid deadlock in time synchronization.

When a process receives a time request from one of its successors, it checks the process 'id' and request time to match any request already in the request queue. If an entry is encountered a RYES message is sent to the requesting successor, indicating that a cycle has been encountered. If the process has not reached the requested time, then it makes time requests to all of its predecessors with lower channel times than the requested times. In order to detect cycles in the connection graph, it places the originating processor 'id' in the time request message of its own process 'id'.

Thus, this algorithm provides a time synchronization strategy for a parallel discrete event simulation running on a number of processors. As the authors have demonstrated, the algorithm can effectively handle deadlock situations, with RYES type messages. The paper is specifically important from this research point of view, because the algorithm has been implemented on an Intel iPSC/2 machine.

Davis et al. [Davis 90] from Virginia Polytechnic Institute have proposed a conservative "null" message algorithm to carry out a discrete event simulation on an Intel iPSC/2. They have investigated the effects of network topologies on the speedup factor. The paper presents a method for creating a distributed event list and describes conditions under which the algorithm can be expected to efficiently provide significant speedup of a discrete event simulation. The distributed event list algorithm uses a method of interprocessor communication and synchronization based on the null message algorithm for distributed simulation proposed by Chandy and Misra [Chandy 81].

The authors have derived an ideal speedup calculation for a parallel implementation of a discrete event simulation. They further state that under ideal conditions, (i.e. assuming zero communication load) the theoretical speedup is equal to N^2. This means, for a sequential event list implementation, theoretically, superlinear speedup is possible. They further explain that this theoretical speedup is in general not attainable because of the communication overhead.

The basic experimentation in their research involves simulating 32 submodels, each on a single processor. Each submodel has 2 tandem M/M/1 queues with only about 10% of the entities flowing out of the submodel. The remaining entities are terminated. The total input of entities constitutes entities created in each submodel and the entities arriving from the predecessor's submodels. The implementation involves 3 types of network configurations of submodels; viz. tandem, balanced feedforward, and unbalanced feedforward. The system is simulated on 2, 4, 8, 16, and 32 parallel processing nodes.

From the above experimentation, the authors have concluded that:

1. The theoretical speedup factor for parallel simulation using the event list algorithm is greater than N (the number of processors).
2. The strategy of sending extra "stimulus" null messages, and the "Null" messages with a timeout between sends are only marginally effective.
3. The attainable speedup is highly dependent on the absence of feedback loops in the logical system topology.
4. The tightly coupled logical processes complicates the assignment problems.

The knowledge base available in the field of parallel discrete event simulation (PDES) is narrowly focused on designing processor synchronization mechanisms for a number of deadlocking and non-deadlocking situations. Very few studies [Fujimoto 1989] have comprehensively summarized the effect of pertinent factors on PDES. The proposed research attempts to provide insight into PDES by observing the effects of a number of factors that influence PDES. The literature review described in this chapter provides the basic foundation in determining the factors that influence PDES and the effects of these factors on PDES. This research uses the RYES type message passing mechanism as a foundation for designing the processor synchronization schemes. The next chapter provides a detailed list of goals and objectives of the research.

CHAPTER IV

STATEMENT OF RESEARCH

Research Goal

The motivation for this research effort is to explore the concepts, problems, and implementation design difficulties associated with the application of parallel processing technology for the simulation of discrete event systems. The primary goal of this research is to identify and quantify the factors that significantly influence an effective parallel implementation of simulations of object oriented models of manufacturing systems. Based on the knowledge and information obtained from the above investigation, a secondary goal of this research is to suggest specific guidelines for any such parallel implementations.

Research Objectives

To accomplish the goal, the following research objectives have been identified:

OBJECTIVE 1 - Concurrent Object Oriented Modeling

Evaluate the constructs of concurrent object oriented programming languages and environments in the context of their applicability to parallel discrete event simulation. Concurrent Object Oriented Languages provide the necessary constructs to concurrently execute programs on parallel architecture machines. Under this objective a number of constructs mentioned in the literature review will be studied to create an in-depth understanding of the concepts involved in process synchronization.

OBJECTIVE 2 - Submodel Creation Logic

Develop strategies for creating submodels by logically clustering the simulation objects present in an Object Oriented model of a manufacturing system. These submodels will each be executable as an independent process running on a single processor. The external arrivals to the portion of the system emulated by the submodel are the communication messages between the corresponding processors. Therefore, the submodel creation will be based on (1) dependencies of the simulation object processes (2) communication overhead and (3) number of available processors. The attempt will be to break up the main model into as many submodels as the available number of processors, by considering the event dependencies that minimize the communication between processors.

OBJECTIVE 3 - Communication Strategy Design

Develop strategies for effectively achieving communication between Intel iPSC/2 processor nodes. In the context of an event based simulation approach, the event scheduling logic will keep track of the communication messages the processor needs to receive before the next event is scheduled. On the basis of the generalized network topology of the manufacturing system, the event scheduling logic will be modified. This will require the development of several communication strategy designs, one for each network topology.

OBJECTIVE 4 - Performance Analysis via PDES Implementation

Create and implement several PDES models on the Intel iPSC/2 hypercube computer in order to carry out performance analysis of the PDES of the experimental manufacturing system. The performance analysis of the manufacturing system involves determination of the influences of the selected experimental design factors, i.e., network topology, system size, and number of available processors on the PDES performance measures, viz. speedup and efficiency. The performance measures will be collected under all the desired system configurations given in the experimental design.

OBJECTIVE 5 - Methodology for Achieving a Successful PDES

Develop a comprehensive methodology for achieving PDES on a parallel computer with distributed memory architecture. On the basis of the investigation described in objective 4, develop a comprehensive methodology that takes into account the influences of each factor, the submodel creation strategies, and communication mechanism designs, for achieving a successful Parallel Discrete Event Simulation model.

Research Scope and Limitations

During the course of this research, the author has come across numerous innovative, intelligent, pertinent, and tangential research issues, that lie fairly outside the scope of this research. Typically, these issues are inclined towards the field of computer science. The author has neither the background nor the inclination to pursue these leads. For example, "The management of PDES of a system having a network with deadlocking topology" has been one of the most popular research topics in the literature.

The primary assumption of this research is that a distributed memory parallel computer is employed for parallel discrete event system simulation. In a distributed memory computer, the communication between the processors is completely handled by messages passed across the processors. There is no global memory. The submodel creation strategies and communication mechanism designs are based entirely on this fundamental assumption.

The scope of this research is limited to manufacturing systems. The manufacturing systems that are modeled as a part of the investigation are gross simplifications of the real world systems. For example, material handling, bounded work-in-process buffers between machines, alternate product routings, machine tools, customer order arrivals, sophisticated system controllers, etc., are not included in the models under this investigation. Thus, this research is a preliminary analysis of the execution of PDES solely based on the generalized product routings in the manufacturing system.

Another severe limitation of this research is that during a simulation the deadlocking situations are avoided either by careful selection of non-deadlocking product flow routings, or by capturing the deadlocking portions of the generalized routing network within a submodel. This arrangement keeps the deadlocking portion under the control of a single processor, thereby avoiding deadlocking situations during the parallel execution.

This investigation is conducted on an Intel iPSC/2, a parallel processing machine with its 32 nodes connected in the form of a hypercube. Therefore, the nature of investigation is limited to those aspects which are compatible with the characteristics and structure of this particular parallel machine.

Research Contributions

The major contribution anticipated from this research is the conceptualization and validation of the methodology created to guide a user in obtaining Object Oriented Discrete Event Manufacturing System Simulation Models that can be executed on a parallel processing computer. For practitioners who model manufacturing systems, the development of this methodology offers significant rewards in the following two areas. First, by analyzing the manufacturing system topologies and the factors that influence the model execution performance of a parallel implementation, decisions regarding the appropriateness of parallel implementation, suitability of the number of processors, and submodel creation strategies, can be made a priori to the actual implementation. Second, the detailed processor communication design for a corresponding network topology, made available by this research, can be used as a basis for designing the required processor communication mechanisms.

Other significant contributions anticipated from this research are:

- Determination of the factors that have the potential to influence the execution performance of a parallel discrete event simulation of manufacturing systems.
- Demonstration of the viability of parallel implementation of discrete event simulation of manufacturing systems via a modified event scheduling technique.
- Provision of empirical data for further research in the areas of parallel discrete event simulation of discrete event systems.

CHAPTER V

RESEARCH PLAN AND METHODOLOGY

Research Plan

In order to achieve the goals and objectives outlined in Chapter Four, this research is carried out in the phases detailed below. Phase I finalizes the performance measures, experimental conditions, experimental factors, and other design methodologies. Phase II proposes a manufacturing system model that can be used to implement the desired experimental conditions, or to set the levels of the experimental factors at the desired values. This model is expected to emulate any needed experimental condition, without undergoing a structural change. Phase III proposes verification of the basic simulation model and the statistics collection routines, by creating an equivalent model in the general purpose simulation language, SLAM II [Pritsker 86]. In this phase, a validation of the simulation model is not attempted, because the experimental manufacturing system, though realistic, is purely hypothetical. Phase IV attempts to implement and execute the discrete event simulation model of the manufacturing system on a single node of the Intel iPSC/2. This yields the uniprocessor simulation execution time, which is used as a baseline for speedup calculations. Phase V establishes a number of design strategies used to achieve an effective split of the global model into smaller chunks, each executed on a single processor. Phase VI is devoted to designing a number of processor synchronization mechanisms to accomplish a distributed simulation of the desired discrete event simulation model. Phase VII achieves successful implementations of the designs proposed in both Phases V and VI. Phase VIII verifies that the parallel discrete event simulation of the manufacturing system has been implemented successfully under all the experimental conditions. It provides the results and conclusions of the experiment. A design methodology which accomplishes parallel discrete event simulation is expected as one of the important outcomes of this phase.

PHASE I - Finalize Experimentation Details

Phase I finalizes the experimentation details such as performance measures and experimental design factors that will be used during parallel discrete event simulation experimentation. The performance measures for parallel implementation of discrete event simulation is derived by comparing it to its equivalent single processor implementation. A good indication of the quality of parallel implementation is its speedup. The speedup is defined as

$$\text{Speedup} = \frac{\text{execution time for uniprocessor implementation of the problem}}{\text{execution time for a parallel implementation}} \qquad (3)$$

Another indicator of the quality of parallel implementation is the utilization of each processor or its efficiency. The efficiency is defined as

$$\text{Efficiency} = \frac{\text{speedup}}{\text{number of processors used in the parallel implementation}} \qquad (4)$$

Of the above two performance measures, this research will consider speedup as the performance measure of primary importance. Measured efficiency is only used as a threshold limit below which the implementation performance is not desired.

The goal of this experimentation is to investigate the effect of experimental factors on the performance of parallel discrete event simulation. Among several factors that affect the performance of PDES, the size of the manufacturing system, the load (utilization of all machines) on the manufacturing system, the number of processors used for parallel implementation, and the communication protocol design are the prominent factors selected for experimentation. The former two are non-controllable factors because they are automatically decided by the manufacturing system being modeled. The latter two can be controlled by selecting their most suitable levels. These experiments will be repeated for a variety of submodel network topologies thereby observing the effects of experimental factors on different network topologies.

The following network topologies are selected for the PDES experimentation.

The Submodel Network Topologies

1: Submodel Network with "Independent Clusters" of nodes.

2: Submodel Network with "Tandem" arrangement of nodes.

3: Submodel Network with "Fork" topology of nodes.

4: Submodel Network with "Join" topology of nodes.

5: Submodel Network with both "Join" and "Fork" topologies of nodes.

In order to develop a preliminary understanding of the functional relationship of the above four factors with the performance measure, an experimental design is set up as described below;

1. The Size of the Manufacturing System Model (2 Levels):

Level-1: Small size with 32 machines and 32 parts.

Level-2: Large size with 256 machines and 32 parts.

2. The Load on the Manufacturing System (2 Levels):

Level-1: Low load with average utilization of each machine around 0.4.

Level-2: High load with average utilization of each machine around 0.8.

3. The Number of Processors Used for PDES Implementation (6 Levels):

Level-1: Uniprocessor implementation.

Level-2: Two-processor implementation.

Level-3: Four-processor implementation.

Level-4: Eight-processor implementation.

Level-5: Sixteen-processor implementation.

Level-6: Thirty two-processor implementation.

4. Communication Protocol (2 Levels):

Level-1: "Forward" protocol.

Level-2: "Forward+Backward" or "Demand Driven" protocol.

Each simulation is performed with a fixed simulation time interval and three replications of each combination of the levels of the factors. This leads to (2x2x6x2x3 = 144 simulations) for each network topology. For a given network topology the interprocessor communication messages are frequently well balanced so that all the processors are allocated approximately the same amounts of both communication and execution load. Two levels of communication protocols differently affect the execution performance under unbalanced and infrequent communication messages. This obviously can not be tested by the above experimental design. A new experiment with a specific manufacturing system routing structure that produces infrequent communication messages is designed to handle this situation.

PHASE II - The Manufacturing System Model

The second phase deals with the modeling process. In this phase, a suitable manufacturing system is specified. An object oriented model of this manufacturing system is developed to take advantage of the modeling flexibility and reusability provided by the object oriented paradigm. More discussion on this new experiment is provided under "Selection of Manufacturing System" section.

PHASE III - Verification of the PDES Environment

In order to verify the simulation execution and statistics collection routines, an equivalent SLAM II system model is developed. Exactly matching statistics collection numbers can be obtained by substituting deterministic input values for the various distributions in the model. This verification is vital for testing the simulation logic, the part creation process, machine operations, and the accuracy of routing implementation. The demonstration of simulation logic verification is supplied in Chapter X.

PHASE IV - Implementation of the Simulation Model on the iPSC/2

In this phase, the simulation model is executed on a single processor. This is an essential step to overcome the incompatibilities between the version of C++ used for the model development on a PC and AT&T's C++ version available on the Intel iPSC/2. Verification of a uniprocessor implementation is provided in Chapter X.

PHASE V - Design Strategies for Submodel Creations

In this phase, a design strategy that allows a global model fragmentation into smaller submodels is developed. Using this strategy a global model is divided into pieces of smaller submodels and each one is executed by a single processor node of the Intel iPSC/2. Event dependencies of the processes pertaining to a submodel, communication overhead of the processor network, volume of the inter-processor communication, etc., are some of the important factors that influence the design strategy.

PHASE VI - Design Strategies for the Synchronization Mechanisms

In this phase, a strategy will be designed to establish proper synchronization among the iPSC/2 nodes. As explained before, each node is assigned a set of machine processes. During the simulation, the submodels communicate with each other via message passing. If the simulation process on a submodel has to wait for a message from another submodel on a different node, the simulation logic should detect this condition and stop the simulation until the node receives the message. Typically, a variety of algorithm designs can be created for this detection, to halt the simulation execution on the node, and to take an appropriate action when the required message is received. Since the allocation of the processes to the nodes does not change dynamically, the communication patterns in the network can be determined a priori. This helps in tailoring a strategy to exactly satisfy the individual requirements of each node.

PHASE VII - Implementation of Phases V and VI

In this phase, the strategies prepared for the submodel creation and communication mechanism designs are implemented. The implementation of the designs developed in Phase V and Phase VI involves creation and testing of the simulation code. The verification of the multi-processor implementation of discrete event simulation of the target manufacturing system is provided in Chapter X.

PHASE VIII - Implementation of the Experimentation

In this phase, the simulation model of the manufacturing system is executed several times (each with different experimental conditions) to obtain the results for the entire experimentation. These results will be verified with those obtained from the corresponding uniprocessor implementation. This phase represents termination of the research activity and presentation of the results in the final format.

Selection of the Manufacturing System

A hypothetical manufacturing system having 'M' number of machines, and processing 'N' number of part types is selected for the purpose of experimentation. A generalized model of this system is shown in Figure 9. Depending on the routing specifications, each part can follow a distinct route through the system before converting itself into a final product. There is an input buffer at each machine. Work flow generators, which serve the same purpose as 'Create' nodes in SLAM II, schedule arrivals of parts with a given interarrival distribution. Appropriate statistics, such as queue

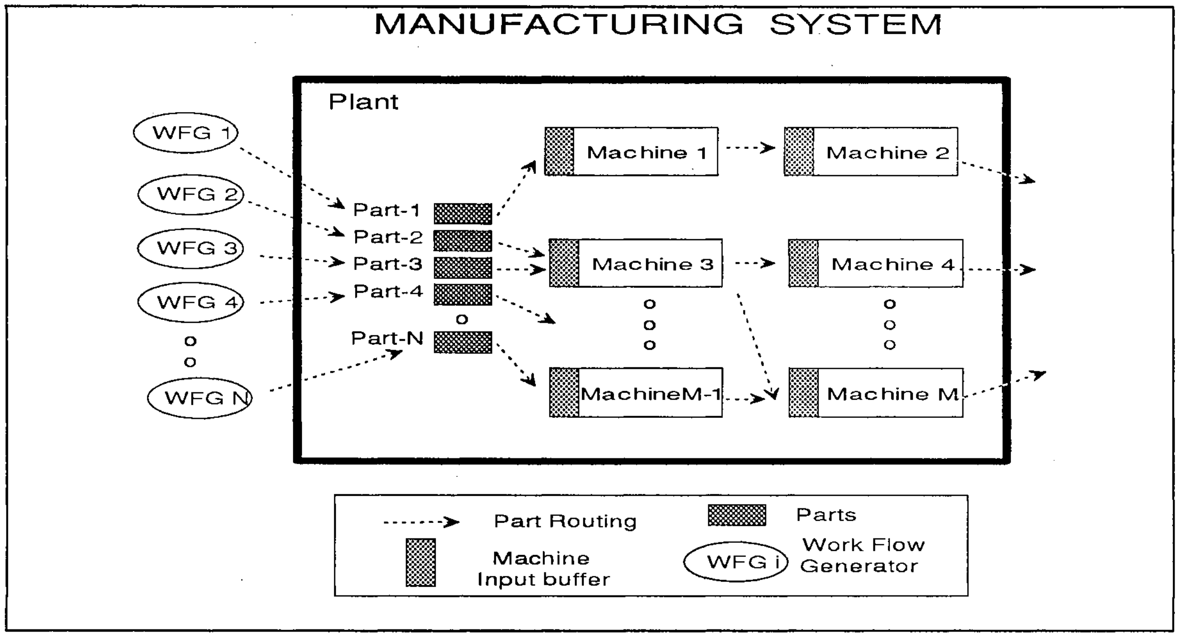

Figure 9. A Generalized Manufacturing System Model used for Experimentation

length, machine utilization, time-in-system, etc., are collected at the required locations in the system. This manufacturing system model can be used to emulate all five proposed submodel network topologies. The size of the model can be changed by simply specifying new input values for 'M', and 'N' and by specifying the desired routings. Thus, this manufacturing system model is simple but quite convenient for this research. A C++ object oriented simulation model of this generalized manufacturing system is provided in Chapter VI.

Selection of Submodel Network Topologies

The purpose of this section is to describe the submodel network topologies selected for parallel discrete event simulation. Figure 10 depicts the five levels of submodel network topologies that are selected for the implementation.

1] Network with "Independent" or "Disjoint" submodel nodes

A submodel network topology that has no arcs or communication between any two submodels falls under this category. A simulation model of a cellular manufacturing system with completely separate product lines, can be sub-divided such that each product line becomes a single submodel. Under this scenario, the machines belonging to the same product line do not interact with the machines from the other product lines, and therefore require no communications between the submodels. Figure 11 depicts a 32 disjoint node network created in conjunction with a 32 processor simulation implementation. For small size models each node simulates a single machine whereas, for a large size model containing 256 machines, each node simulates an eight machine cluster that forms a single part routing.

2] Network with "Tandem" arrangement of submodel nodes

A submodel network topology that has a unidirectional tandem configuration of the submodels falls under this category. Figure 11 depicts a 32 node tandem network created in conjunction with a 32 processor simulation implementation. For small size models each node simulates a single machine whereas, for a large size model containing 256 machines, each node simulates 8 machines. To maintain balanced communication patterns for all nodes, tandem routing for each part has machines from two consecutive nodes.

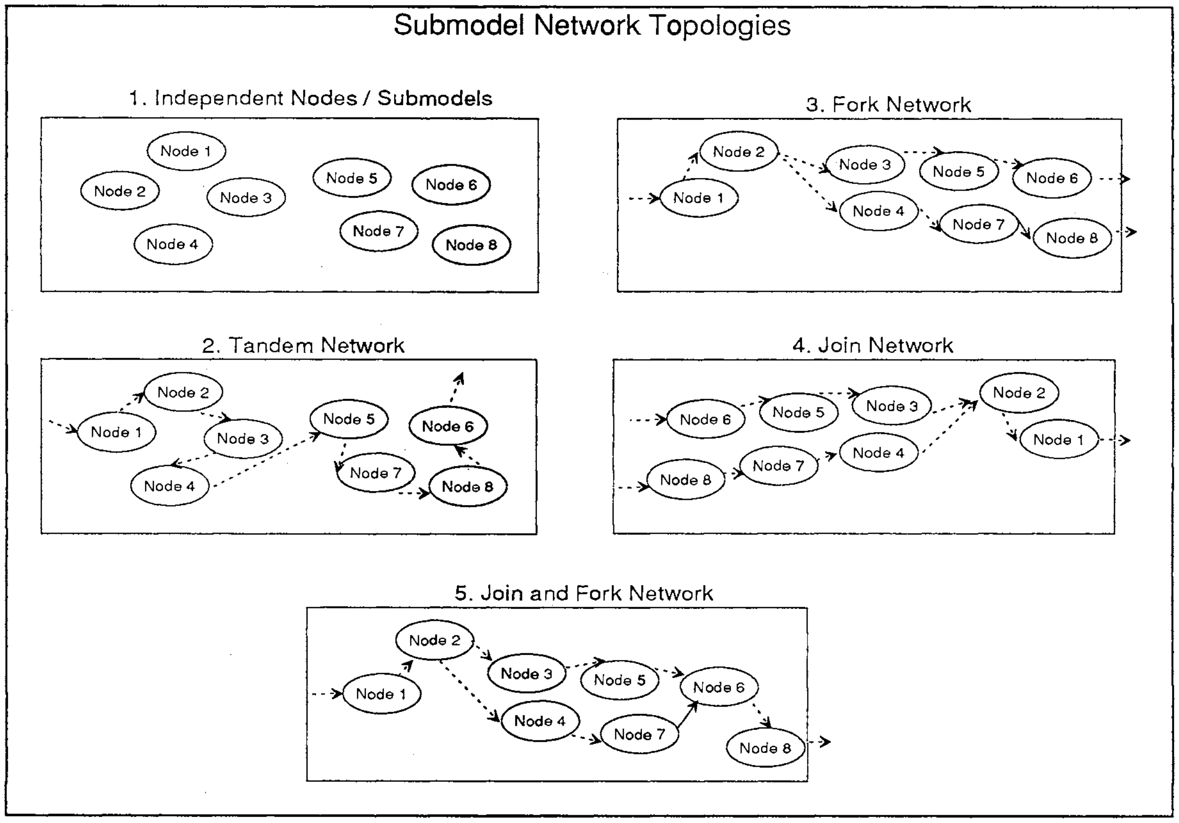

Figure 10. Types of Networks Considered for Experimentation

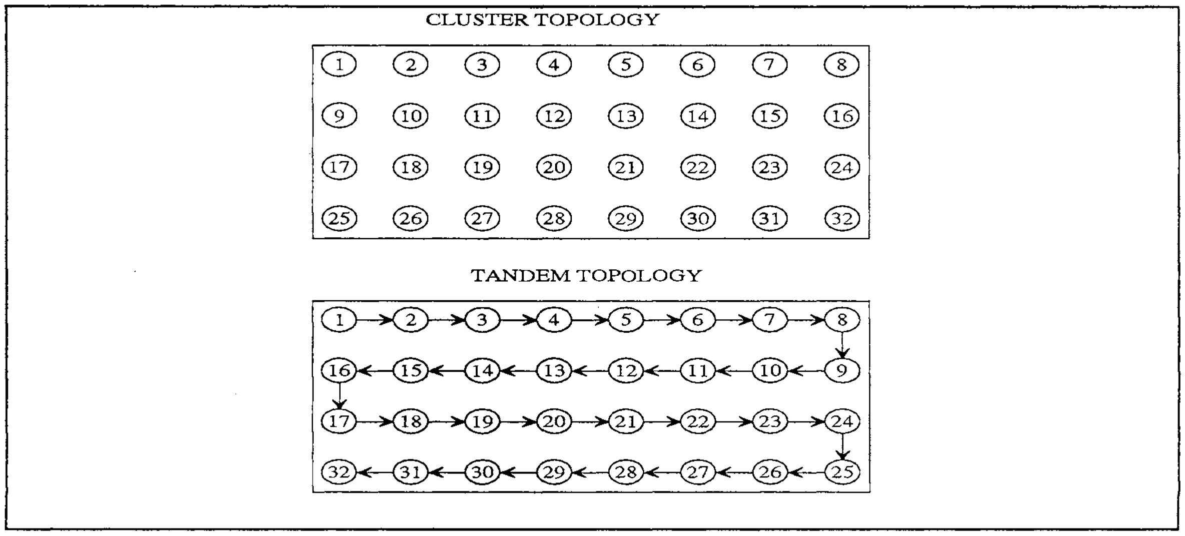

Figure 11. Thirty Two Node Network for "Clusters" and "Tandem" Topologies

3] Network with communications that form 'Fork' topology of submodel nodes

A submodel network with some of the submodel nodes connected in such a way that each of these submodels has a single input communication and multiple (two) output communications, falls under this category. Figure 12 depicts a 32 node "Fork" network created in conjunction with a 32 processor simulation implementation. For small size models each node simulates a single machine whereas, for a large size model containing 256 machines, each node simulates 8 machines. To maintain balanced communication patterns for all nodes, the fork network structure is derived by use of tandem routings for several parts, each extending from the root node over to a single forked branch.

4] Network with communications that form 'Join' topology of submodel nodes

A submodel network with some of the submodel nodes connected in such a way that each of these submodels has multiple input communications and a single output communication, falls under this category. Figure 12 depicts a 32 node "Join" network created in conjunction with a 32 processor simulation implementation. For small size models each node simulates a single machine whereas, for a large size model containing 256 machines, each node simulates 8 machines. To maintain balanced communication patterns for all nodes, the "Join" network structure is derived by use of tandem routings for several parts, each extending from a single joining branch over to the joined node.

5] Network with both 'Join' and 'Fork' topology of submodel nodes

A submodel network with some of the submodel nodes connected to form a "Join" topology and other submodel nodes connected to form a "Fork" topology falls under this category. Figure 12 depicts a 32 node tandem network created in conjunction with a 32 processor simulation implementation. For small size models each node simulates a single machine whereas, for a large size model containing 256 machines, each node simulates 8 machines. To maintain balanced communication patterns for all nodes, the "Fork + Join" network structure is derived by use of two tandem routings each originating at the same node then forking out to two different nodes and then joining back at the next node.

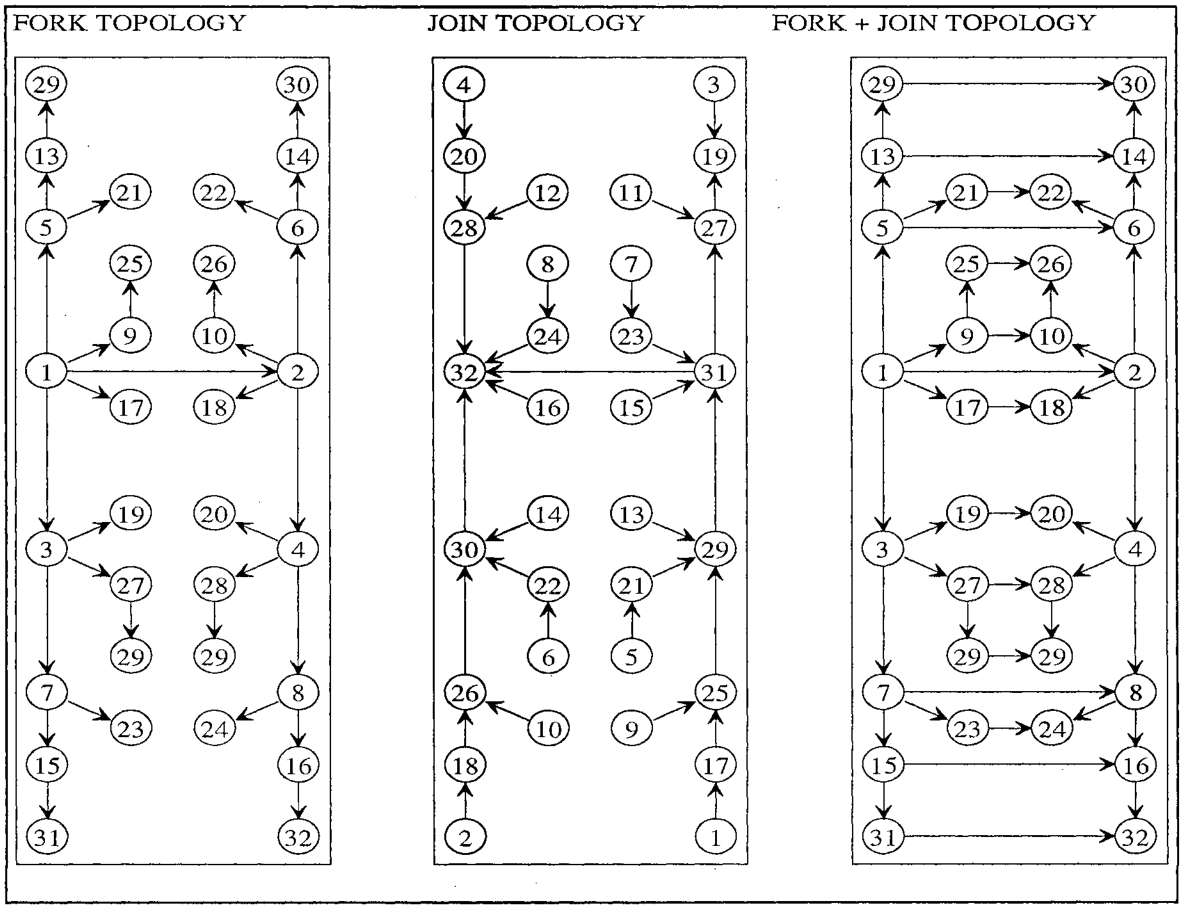

Figure 12. Networks for "Join", "Fork" and "Fork+Join" Topologies

This completes the outline of the research plans and methodology. The following chapters contain the description of the experimentation environment, experimental results and conclusions.

CHAPTER VI

OBJECT ORIENTED REPRESENTATION

Introduction

This chapter describes an object oriented representation of the target manufacturing system selected for the experimentation. It also provides information about the simulation experimental set up in the next section. C++ is chosen to be the implementation language for object oriented representation of the manufacturing system. An object oriented model of the manufacturing system consists of a variety of classes of modeling primitives such as machine, work flow item, work flow generator, routing, etc., which are directly abstracted from their respective real world counterparts. The detailed object library described below also contains a variety of simulation support objects such as random number generator, simulation, statistics collection, etc., that provide the code for simulation execution, random variate generation, and statistics collection. These two types of objects together constitute a parallel discrete event simulation environment. C++ object class library code resides in 'cnode.c' file listed in the Appendix A. The following section briefly describes the C++ object class library.

C++ Class Library

The purpose of this section is to describe the C++ object class library for the parallel discrete event simulation environment. The object oriented model of the generalized manufacturing system described in an earlier chapter constitutes a number of classes dealing with the modeling primitives, the simulation logic, and the statistics collection. Figure 13 depicts the important modeling objects, their instance variables, and their class hierarchy in the simulation environment. A brief explanation of the important classes is given below:

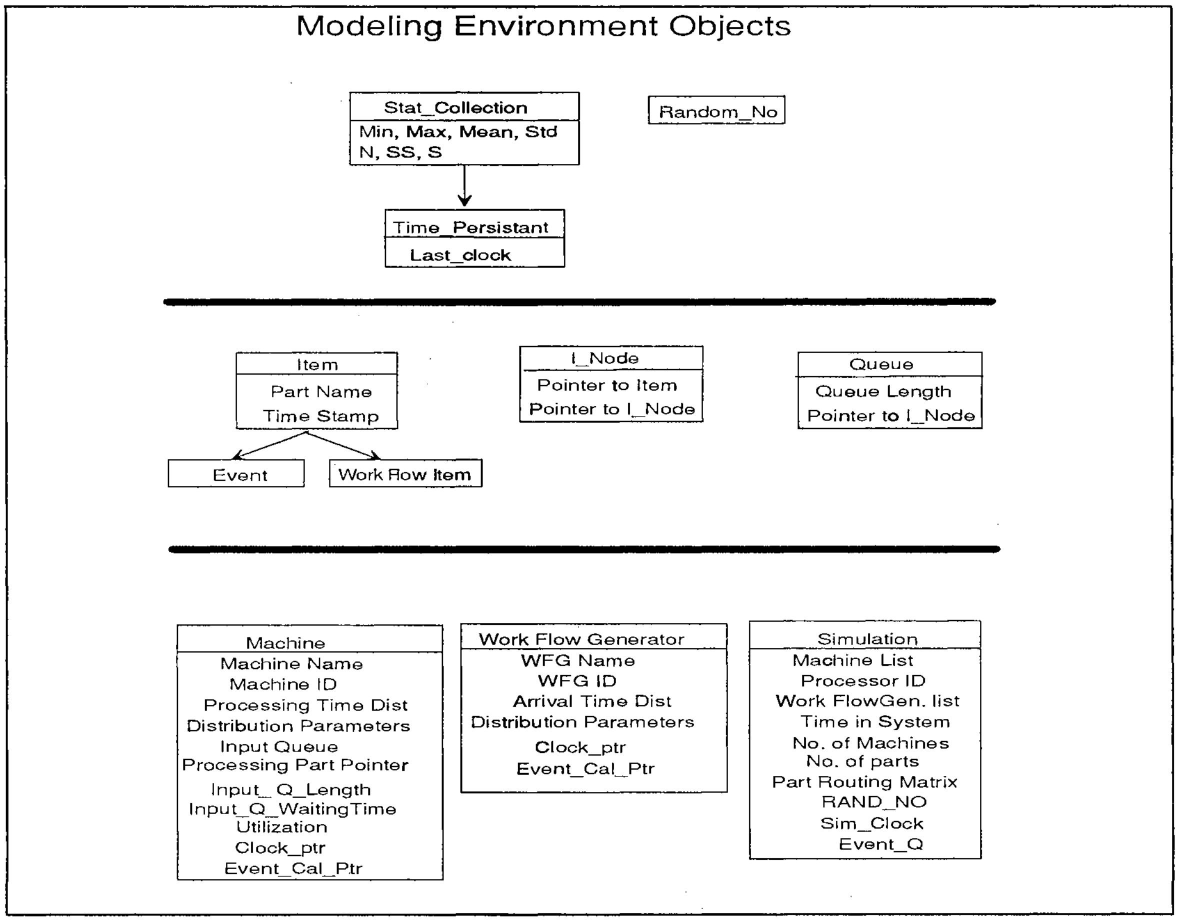

Figure 13. Modeling Objects of the Manufacturing System

Stat Collection: This class collects the observation based statistics and maintains the minimum, the maximum, the average, and the standard deviation of a data stream.

'min', 'n', 'max', mean', 'std' are some of the important instance variables defined for an instance of this class. Following are the important member functions of this class.

'Collect(double x)': This member function collects the value of 'x' as a new observation of the data stream and calculates the minimum, maximum, cumulative sum and the cumulative sum of squares for the data stream.

'print_result(char* statistics)': This function prints to the output file the statistics collection results that include, number of observations, the maximum, the minimum, the average, standard deviation along with the title provided by the character pointer 'statistics'.

Time Persistent: This is a subclass of the "Statistics Collection" class. It collects time persistent variables such as utilization, and queue length. Besides the member functions inherited from its superclass, the Time Persistent class stores previously collected values in the instance variable 'last_clock', so that the time persistent statistics can be collected. This class overrides the super class 'Collect(double x)'by 'Collect(double clock, double new_value)' member functions because the time persistence calculations require both the new observation and the observation collection time.

Random_No: This class generates both random variates and random numbers. It is designed to operate in multiple random number stream mode. That is, a different random number stream can be assigned to each stochastic process in the simulation. 'Random_No' class requires a seed, a distribution type, and appropriate parameters. After generating the random variate the new seed value is sent back to the requesting object, such as the work flow generator or a machine. The requesting object will supply this new seed for the next random variate request. Following are the important member functions of this class.

'next_seed(long seed)': This member function receives a seed value, uses Park and Miller [Park 1988] random number generator algorithm, and returns the next seed.

'next(char D, double param[], long& seed)': This member function receives a distribution type, the distribution parameters, and a seed value. It uses 'next_seed' function to obtain a new seed value and generates the appropriate random variate specified by distribution type 'D' and parameter vector 'param[]'.

Item: This class provides an abstract structure to both its subclasses. It offers 'Part Name' and 'Time Stamp' as two instance variables to its subclasses 'Event' and 'WFI'. Instances of 'Item' are queued by an instance of class 'Queue', thereby allowing the instances of its subclasses to be queued in an event calendar or machine input queues.

I_Node: This structure holds a pointer to the instance of class 'Item' and a pointer to an instance of the next entity queued in a 'Queue' object. Thus 'Queue' is made up of a linked list of the instances of structure 'I_Node'.

Queue: This class consists of a linked list of the instances of structure 'I_Node'. It queues instances of class 'Item' by using structure 'I_Node'. Following are the important member functions of this class.

'ADD(Item *ani)': This member function receives an instance of class 'Item" and, adds it to the queue. It uses 'Time_Stamp' value to determine the queue discipline such that the lowest 'Time Stamp' value items are queued at the front of the queue.

'REMOVE()': This function removes the first item from the front of the queue.

Event: This class holds the attributes of events that are scheduled on the event calendar. It acquires its behavior from its superclass 'Item' through inheritance.

WFI: This class holds the attributes of parts that move through the manufacturing system. Class 'WFI' also obtains its behavior from its superclass 'Item' through inheritance. It has additional instance variables, 'Start_Time' that keeps track of the simulation entering time of a work flow item, 'At_Step' which holds the routing step number, and 'Serial' which stores the serial number.

Machine: This class models the behavior of a machine. Its instance variables are 'Input_Queue' an instance of class 'Queue', 'P_Dist' that stores processing time distribution, 'P_Param[3]' which has the distribution parameters, 'Status' which maintains the status of the machine, and 'seed' that contains the current random number seed. A number of instance variables such as 'I_Q_L' (input queue length), 'I_Q_WT' (input queue waiting time), 'Util' (machine utilization), and 'Blk' (blocking) are provided for the statistics collection. Two instance variables, '*blocking_mc' (containing the pointer to the machine blocking this machine) and '*BM' (containing a list of pointers to the machines that are blocked by this machine), help accomplish the blocking operations and unblocking of the machines. Following are the important member functions of this class.

'accept(Item *awfi)': This function accepts a work flow item from the previous machine or a work flow generator. If the machine is idle the processing of the work flow item begins immediately. Otherwise, the work flow item is added to the input queue.

'start_process()': This function removes a work flow item from the input queue and begins its processing. If this machine has been blocking some other machine, the processing at that machine is restarted by sending a 'restart_blocked_process' message.

'end_process()': This function returns a processed work flow item that can be sent to the next processing machine in its routing.

'block_process()': This function blocks the processing on the current machine if the input queue of the machine down stream is full.

'restart_blocked_process()': This function restarts a blocked process by sending its processed part to the blocking machine and accepting a new part from the input queue.

WFG: This class provides a behavior similar to a 'Create' node in SLAM II. It creates instances of class 'WFI' according to an arrival distribution and enters them into the simulation.

Simulation: This class controls the entire simulation process. It also provides the messages that communicate to other instances of this class on different processors. It holds references to all the machines and work flow generators under its control. Its instance variables supply routings, an event calendar, a random number generator and several statistics collection objects. It primes up the event calendar by sending an initialization message to all the work flow generators and then schedules the events on the event calendar. In multiple processor implementation, class 'Simulation' terminates a work flow item at the end of its routing, or sends it to the processor containing the next machine in its routing. Following are the important member functions of this class.

'ReceiverSender()': This function identifies the predecessor and successor processors for the current processor. This member function is executed before the beginning of simulation so that the processor communication patterns are deter-mined a priori.

'Perform()': This function performs the actual simulation. It contains the communication protocol for inter processor communication. It identifies when an event can be scheduled without violating the causality principle and then schedules the event using 'ScheduleAnEvent()' function.

'ScheduleAnEvent()': This function removes an event from an event calendar and schedules it using either FinishUp(int machineNo)' or 'Arrival(int partNo) function.

'FinishUp(int machineNo)': This function determines whether a work flow item can be moved to the next machine in its routing sequence. If it can be moved (i.e. when the next machine is not blocked) the work flow item is moved to the appropriate machine, else the previous machine is sent a 'block_process' message to block the processing of the current work flow item. It the next machine is on a different processor, the work flow item is sent to that processor using 'SendMessage(awfi,AP[next_mc])' message.

'SendMessage(awfi,AP[next_mc])': This function creates a message from the work flow item description and sends it to the processor stored in 'AP[next_mc]'.

'ReceiveMessage()': This function receives a message from a predecessor process-or and creates a work flow item object from it. This work flow item object is stored in an 'InputBuffer' and an event corresponding to the arrival of the work flow item is scheduled on the event calendar.

Experimental Setup

This section describes the experimental setup provided for conducting a variety of experiments on the Intel iPSC/2 hypercube. As mentioned in the third chapter, the hypercube computer consists of a SRM (System Resource Manager) and a set of nodes connected in the form of a hypercube. A typical application run on the hypercube consists of two programs, one which runs on the SRM machine and a second that runs on a selected set of nodes. The SRM program is typically known as a 'host' program, whereas the program running on a node is called the 'node' program.

The host and node programs of this application reside respectively in 'chost.c' and 'cnode.c' files. Compilation of these programs requires linking a number of library files and setting a number of compiler options which are listed in a file called 'makefile' provided in Appendix B. Figure 14 depicts the execution host program. As described in Figure 14, the host program reads a simulation model from 'modelxxx.des' input file, and the submodel allocation information from 'allocxx.tab' file. Upon reading these two files, the host program creates a number of input files named 'pdesxx.inp' for the node program. It also loads the node program onto the specified number of nodes. Each node program accesses its corresponding input file to obtain its assigned submodel. Each node program then executes the assigned submodel, writes the output in 'pdesxx.out' files and the execution trace in 'pdesxx.tra' file. Finally these output files are assimilated in a single 'amxxxpxx.exp' file. The output of a node program contains the machine related statistics such as 'utilization', 'input queue waiting time', etc., and part related statistics such as 'time in system'. Each node program also records its submodel execution time and sends it to the host program which finds the maximum execution time and displays it on the screen.

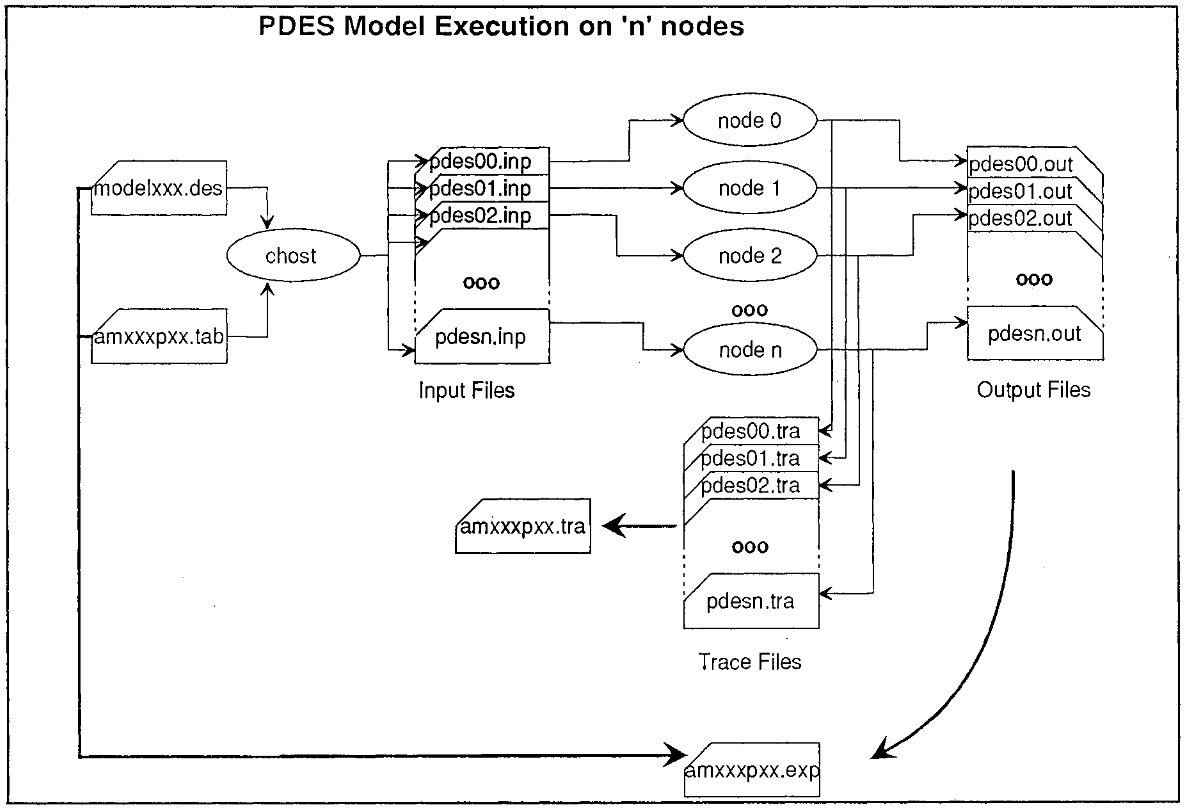

Figure 14. PDES Model Execution on "n" Hypercube Nodes

Shell script 'experiment' receives from the user the model file name, the allocation file name, and the host file name. Then it executes the host file and collects the output in '*.exp' file and the trace in '*.tra' file.

In multiple processor implementation, several node programs simultaneously run on a number of processors. In order to obtain non-erroneous simulation results a proper communication protocol must be provided in the node program. The communication protocols confirm that the causality principles are not violated and allow the simulation on each processor to safely proceed towards completion. The following two chapters first look at the communication protocol constructs provided in the concurrent object oriented programming languages and then supply the protocol design for the implementation of the parallel discrete event simulation on the Intel iPSC/2 hypercube computer.

CHAPTER VII

EVALUATION OF CONCURRENT OOP CONSTRUCTS

Introduction

In order to accomplish the first research objective, this chapter attempts to evaluate a number of concurrent object oriented programming constructs in the context of their applicability to parallel discrete event simulation.

Evaluation of "ConcurrnetSmalltalk" Constructs

A typical object oriented concurrent programming language provides a number of concurrent constructs, which can be used to model concurrent processes. For the purpose of illustration the constructs provided in "ConcurrentSmalltalk" [Yokote 88] are explained below. In "ConcurrentSmalltalk" there are two major constructs that activate and synchronize the objects:

Construct 1 : (&) This asynchronous method call sends a message to the receiver object and the program proceeds to the next line without waiting for a reply message.

Construct 2 : (||) This construct specifies that the receiver object will return a reply and it will continue to execute. That is, after evaluation of this expression the sender and receiver objects are executed concurrently.

To evaluate the effectiveness of these constructs, their execution is analyzed in the context of an example. A Producer-Consumer problem is selected for this purpose.

The development of an object oriented model of this problem in "ConcurrentSmalltalk" and its execution analysis is explained below.

This example attempts to produce a concurrent object oriented model of the simple producer-consumer system depicted in Figure 15. The system consists of a producer who produces items, a consumer who consumes these items and a bounded buffer between the producer and the consumer. This problem is modeled by defining three main classes of objects; a producer, a consumer and a bounded buffer. Each of these objects has an independent process which can be synchronized by the concurrent constructs discussed above. Figures 16, 17, and 18 on the following pages display the "ConcurrentSmalltalk" code of the system model.

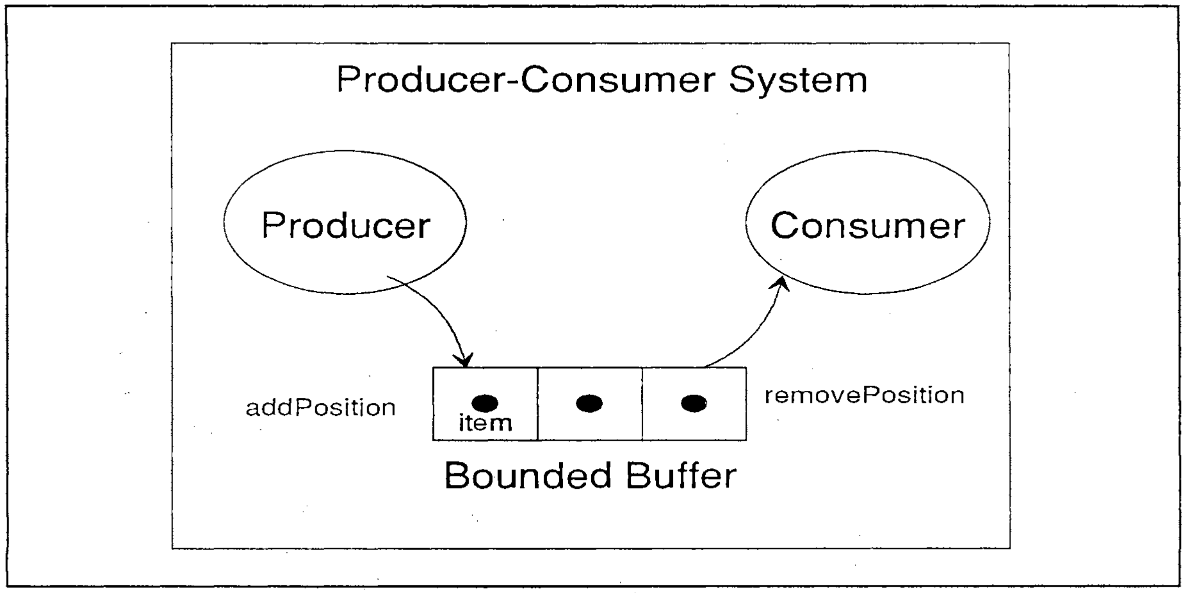

Figure 15. Bounded Buffer in a Producer-Consumer System

```
Object atomic Subclass: #BoundedBuffer
        instanceVariableNames: 'buffer size max read write wait'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Producer-Consumer'

BoundedBuffer methodsFor: 'initializing'
Setup: n
        buffer := Array new: n.
        max := n.
        size := 0.
        removingPosition := 1.
        addPosition := 1.

BoundedBuffer methodsFor: 'accessing'
deposit: anitem
        wait notNil ifTrue: [ wait run&. wait := nil ].   " If somebody is waiting then release it and make wait = nil "
        size = max ifTrue: [wait := thisContext sender receiver. ^#full ].   " If buffer is full, make producer wait "
        ||buffer at: addPosition put: anitem.    " Note the execution continues after replying with anitem. "
        size := size + 1.
        addPosition := addPosition \\ max+1.
remove
        wait notNil ifTrue: [ wait run&. wait := nil ].   " If somebody is waiting then release it and make wait = nil "
        size = 0 ifTrue: [wait := thisContext sender receiver. ^#empty ].   " If buffer is empty, make consumer
                                                                                wait "
        ||buffer at: removePosition.    " Note the execution continues after replying with anitem. "
        size := size - 1.
        removePosition := removePosition \\ max+1.

BoundedBuffer class methodsFor: 'instance creation'
new: max
        | newBuffer |
        newBuffer := super new.
        newBuffer setup: max.
        ^newBuffer

BoundedBuffer class methodsFor: 'example'
example
        " BoundedBuffer example."
        | buffer producer consumer |
        buffer := BoundedBuffer new: 10.
        producer := Producer new: buffer name: #PRODUCER.
        consumer := Consumer new: buffer name: #CONSUMER.
        producer forever&.
        consumer forever&.
```

Figure 16. Program Code for Class BoundedBuffer

```
Object Subclass: #Producer
        instanceVariableNames: 'buffer save myName'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Producer-Consumer'

Producer methodsFor: 'initializing'
Set: aBuffer name: aName
        buffer := aBuffer.
        myName := aName.

Producer methodsFor: 'private'
makeAnItem
        " This code generates items at an interval of time given by an interarrival distribution. "

Producer methodsFor: 'accessing'
deposit: anitem
        | rv |
        rv := buffer deposit: anItem.
        rv = #full ifTrue: [ save := anItem ^#full ]
                ifFalse: [^anItem]

forever
        | rv |
        [true] whileTrue: [ rv := self deposit: self makeAnItem    " Generate an Item and deposit it into the
                                                                        bounded buffer"
                        rv = #full ifTrue: [ ^#full ]]    " rv specifies if buffer is full or not; if full, the execution
                                                                terminates "

run
        | rv |
        rv := self deposit: save.    " Deposit the saved item first "
        rv = #full   ifTrue: [ ^#full ]    " rv specifies if buffer is full or not; if full, the execution terminates "
                ifFalse [ self forever]    " if not full run forever "

Producer class methodsFor: 'instance creation'
new: buffer name: aName
        | newProducer |
        newProducer := self new.
        newProducer set: buffer name: aName.
        ^newProducer
```

Figure 17. Program Code for Class Producer

```
Object Subclass: #Consumer
          instanceVariableNames: 'buffer myName'
          classVariableNames: ''
          poolDictionaries: ''
          category: 'Producer-Consumer'

Consumer methodsFor: 'initializing'
Set: aBuffer name: aName
          buffer := aBuffer.
          myName := aName.

Consumer methodsFor: 'private'
consume: anItem
          " This code consumes items at an interval of time given by a consumption distribution. "

Consumer methodsFor: 'accessing'
remove
          | anItem |
          anItem := buffer remove.
          anItem = #empty ifTrue: [^#empty ]     "If the bounded buffer is not empty return the removed item"
                    ifFalse: [^anItem]

forever
          | anItem |
          [true] whileTrue: [ anItem := self remove.    " Remove an item from the bounded buffer"
                              anItem = #empty ifTrue: [^#empty ]    "If the bounded buffer is empty stop execution"
                                                ifFalse: [self consume: anItem]    " else, consume the item "

run
          self forever    " Restart the consumption process forever; restarted by the bounded buffer"

Consumer class methodsFor: 'instance creation'
new: buffer name: aName
          | newConsumer |
          newConsumer := self new.
          newConsumer set: buffer name: aName.
          ^newConsumer
```

Figure 18. Program Code for Class Consumer

Class BoundedBuffer has two important methods defined; *deposit:* and *remove.* Method *deposit:* is executed when the Producer object sends a *deposit:* message to the BoundedBuffer class. During its execution, method *deposit:* first checks whether the bounded buffer is full. If it is full, Producer is kept waiting and #full is returned to terminate the activities of the Producer. Otherwise, the item is stored in the bounded buffer, and size and addPosition variables are updated. The *deposit:* method also checks whether the customer object is waiting. If it is waiting, the activities of the consumer are restarted by sending a *run* message to it. Method *remove* terminates the activities of the consumer object and makes the consumer wait, when the bounded buffer empties.

The object code of Producer class is depicted in Figure 17. Method *makeAnItem* creates the items that are sent to the bounded buffer. Method *forever* is executed as follows: it produces new items by sending *makeAnItem* to itself, and then deposits these items into the bounded buffer by sending a *deposit:* message to the BoundedBuffer object. If the reply indicates that the bounded buffer is full, the execution of method *forever* is terminated by the BoundedBuffer object. When the Consumer object removes an item from the buffer, method *forever* is restarted upon receiving a *run* message from the BoundedBuffer object.

The object code of Consumer class is depicted in Figure 18. There are two important methods described for the Consumer object. Method *consume:* removes an item from the bounded buffer and delays it for the consumption time determined from a consumption probability distribution. Method *forever* is executed as follows: it removes an item from the bounded buffer by sending a *remove:* message to the BoundedBuffer object. If the reply indicates that the bounded buffer is empty the activities of the consumer are terminated. Otherwise, the consumer consumes an item by sending message *consume:* to itself. When the Producer object adds an item to the buffer, method *forever* is restarted in response to a *run* message received from the BoundedBuffer object.

The producer and consumer objects have their independent processes. Under normal circumstances (that is when the bounded buffer is neither completely full nor empty), both processes operate concurrently (on different processors if the program has been implemented on a parallel processing machine). The execution process of the

Producer object creates items and sends them to the buffer until the process is terminated when the bounded buffer becomes full. This process is restarted as soon as the Consumer object creates an empty space in the bounded buffer by removing an item. The execution process of the Consumer object consumes an item and terminates it from the system until the consumption process is stopped when the bounded buffer becomes empty. This process is restarted as soon as the Producer object makes the bounded buffer non-empty by depositing an item. These processes run forever until they are both terminated when the desired simulation ending time is reached.

The following paragraphs explain the concurrent execution Producer-Consumer system. In this example, it is assumed that the bounded buffer can hold a maximum of three items. The producer takes five milliseconds to execute method *makeAnItem,* and the consumer takes 30 milliseconds to execute the *consume:* method. These times (in milliseconds) are respectively defined to be item production time and item consumption time. Table 2 depicts sample production and consumption times for eight items.

TABLE 2

PRODUCTION AND CONSUMPTION TIMES

Item Number	Production Time (msec)	Consumption TIme (msec)
1	5	30
2	5	5
3	5	5
4	5	5
5	20	25
6	5	5
7	5	5
8	5	5

Refer to Figure 19 for a detailed explanation of the execution of the producer-consumer system model. As the simulation begins the consumer process checks the bounded buffer and finds it empty. This terminates the consumer process.

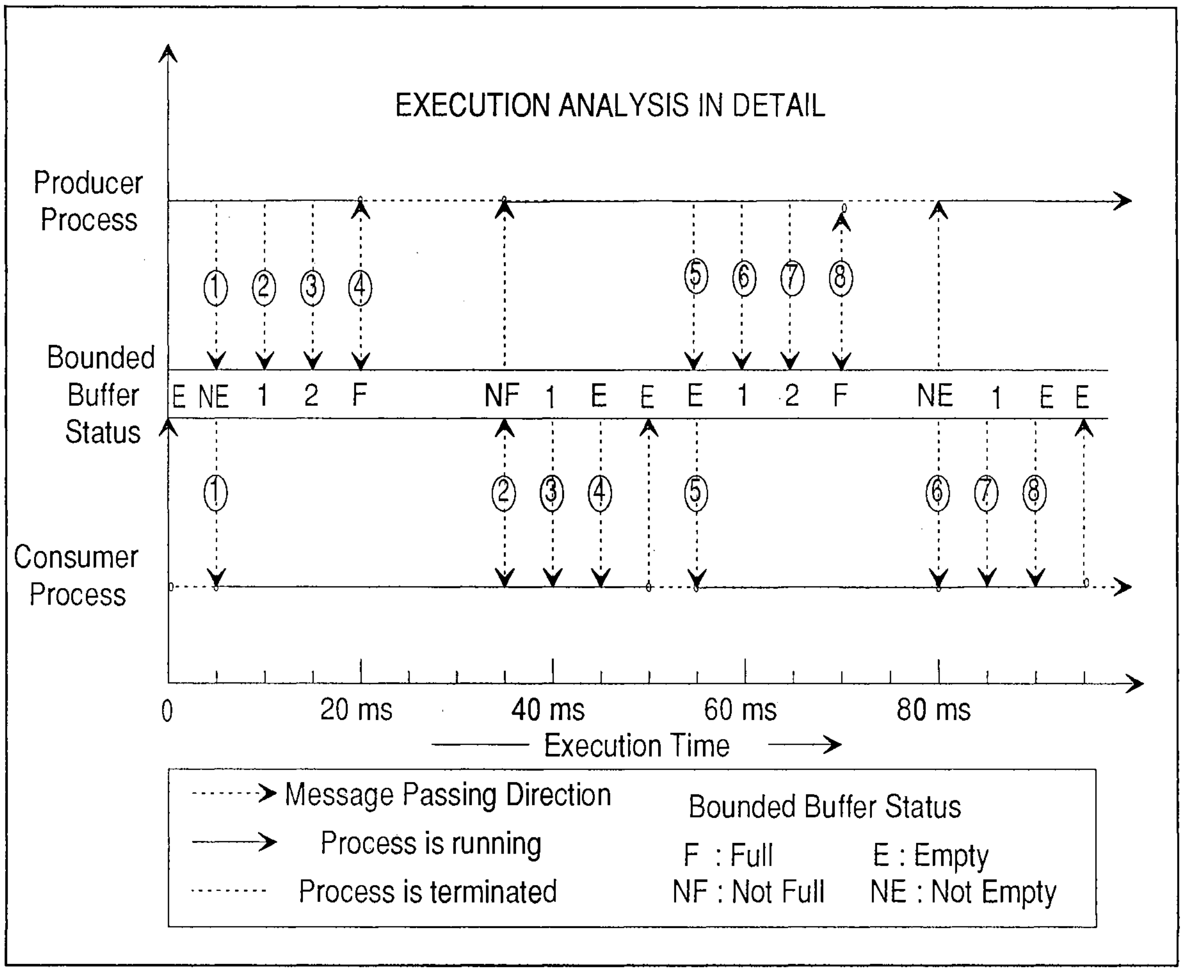

Figure 19. Execution Analysis of the Producer-Consumer System Model

The producer makes an item and puts it into the bounded buffer at time 5 msec. The producer process continues as the bounded buffer is not full yet. As soon as the first item enters the buffer, the consumer process is restarted. The consumer removes the item from the buffer and begins consuming it. Meanwhile, the producer makes three more items and loads them into the buffer to make it full at 20 milliseconds. At this time the producer process is terminated. The producer process is restarted when the consumer picks up the second item at 35 milliseconds. Thus, the producer and consumer processes operate intermittently until the simulation is run for the desired period.

Construct 1 provides a fork for concurrent execution of sender and receiver objects, where as construct 2 is useful in situations where the information requested by the sender is readily available but it also requires recalculation of the internal states. By acknowledging the reply to the request by sender object using construct 2, the receiver object can concurrently execute the updating internal states along with the sender object execution. This is unlike a normal function return call in C where the execution of the current function is automatically terminated as soon as the return statement is executed. These two constructs along with several other synchronization methods such as, receive, receiveAnd:, receiveOr:, etc., form a complete set of concurrent constructs that allow the user to take advantage of inherent concurrency in the problem domain.

In conclusion, object oriented languages like ConcurrentSmalltalk provide highly efficient constructs that can exploit even extremely fine grain concurrency in the application domain without programming rigorous synchronization mechanisms. This analysis of ConcurrentSmalltalk constructs has been very useful in stimulating ideas for developing software communication protocol designs explained in Chapter IX.

CHAPTER VIII

SUBMODEL CREATION METHODOLOGY

Introduction

The purpose of this chapter is to develop a submodel creation methodology based on the analysis of parallel implementation programs for discrete event simulation. In order to achieve this objective, the next section provides a detailed description of the execution of discrete event simulation both in case of uniprocessor and multi-processor implementation. In the following section, a set of factors that influence the submodel creation process are identified, and a general strategy for submodel creation is specified. It is important to note that this strategy is highly influenced by the message passing architecture of Intel iPSC/2 hypercube and its applicability is limited to parallel implementation of discrete event simulation on distributed memory computers.

Analysis of Parallel Discrete Event Simulation

The purpose of this section is to analyze both uniprocessor and parallel implementation of discrete event simulation. It provides better understanding of the necessity of communication, contents of the messages, message passing and receiving mechanisms, etc. for a successful PDES implementation. An arrangement of a typical PDES application is outlined below.

Simulation program arrangement for a uniprocessor simulation execution is explained in Figure 20 on the next page. The simulation program developed for the purpose of this research is similar to a general purpose uniprocessor discrete event simulation framework developed by Mitrani [Mitrani 1982]. It consists of an event calendar which enqueues and schedules simulation events, and a number of functions or procedures that respectively hold event execution code for simulating their respective events. These events update system states, collect statistics, and schedule the new events onto the event calendar. With this arrangement of discrete event simulation program, the "causality" constraints [Chandy 1979] are maintained during the simulation execution. The event calendar acts as a single entity that accepts entire system wide events, and maintains proper ordering of the events. This arrangement results in an error free execution of the simulation.

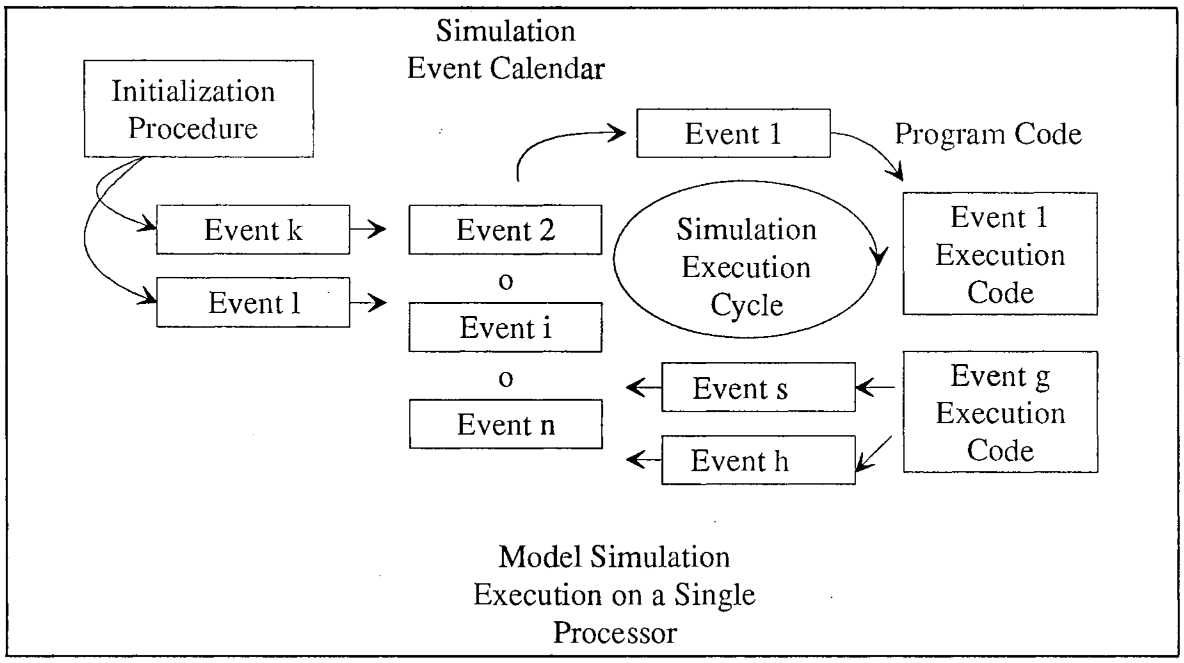

Figure 20. Uniprocessor Discrete Event Simulation

For the purpose of this research, the PDES is achieved by breaking up the manufacturing system model into a number of distinct submodels. Each submodel is then executed on a single processor. This arrangement of the simulation program for parallel implementation of discrete event simulation of manufacturing systems is depicted in Figure 21. As depicted in Figure 21, the execution of a submodel on a single processor involves an event calendar that schedules events happening within the boundary of the submodel, and a set of procedures that contain the code for the events scheduled inside the current submodel. The submodel simulation logic also incorporates mechanisms for scheduling events arriving from the predecessor submodels, and the event code procedures for sending new events to the successor submodels. Thus, simulation execution of each submodel running on a single processor has its own event calendar which simulates the events within the boundary of the submodel, while the dependence of the events between the submodels is reflected by the transfer of event across the submodels.

If the submodels are completely independent (i.e. the cause for the scheduling of an event always lies inside the submodel) there are no incoming and outgoing events or communications, and therefore this arrangement does satisfy the causality principle leading to a successful PDES implementation. However, in general all submodels are not independent. At least some of them are influenced by the events happening in some other submodels. The outcome of such events can modify the sequence of events in the dependent submodel. This can create situations which violate the causality principle and produce erroneous results. However, such situations can be avoided by providing proper synchronizing mechanisms between the respective submodels.

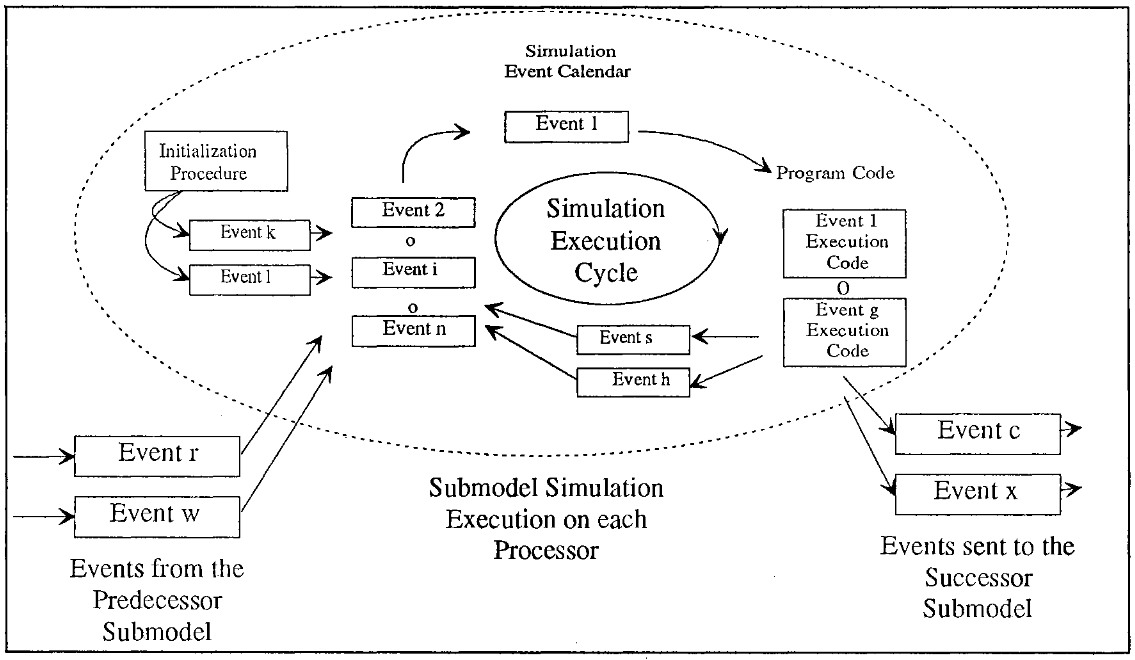

Figure 21. Submodel Simulation Execution Logic

Submodel Creation Methodology

The purpose of this section is to develop a comprehensive methodology for creating submodels from a single manufacturing systems model. There are two major guiding principles for such a methodology. First, for an efficient parallel processing application the goal is to minimize the inter-processor communication. As the independent submodels do not require communication between their respective processors, independent submodels make PDES implementation very efficient. Secondly, in case of dependent submodels, specific topology of the submodel network can create 'deadlock' situations [Chandy 1981]. Even though there are several mechanisms currently available for deadlock detection and recovery, almost all of them provide only marginal speedups [Reed 1988]. And hence for the purpose of submodel creation, topologies that have a potential for 'deadlock' are avoided. Under these two major guidelines submodel creation methodology for parallel discrete event simulation is specified below.

The main focus in submodel creation is to achieve maximum speedup. From knowledge of parallel processing principles, it is evident that the speedup can be improved by using larger and larger number of processors (submodels), and also by minimizing the communication overload on the processors. The communication overload on each processor can be obtained in terms of the number of communication messages between the processors. The evaluation of this number requires definition of the following notation.

Notation:

m	: Number of machines in the system
n	: Number of parts in the system
p	: Number of processors used for parallel implementation ($p < m$)
a_i	: Average number of parts of part type 'i' flow during the simulation run
$X_{i,j,k}$	: Equals one if 'i'th part type moves from 'j'th to 'k'th machine, Otherwise it equals zero
S_l	: Submodel 'l'; a set of machines belonging to 'l'th submodel
Tc	: Average communication time for a single message
Te	: Average computation time for executing a single event
C_K	: Number of communication messages received by machine "K"

As each part moves from one machine to another machine in its routing, it generates messages between two machine processes thereby creating communication overhead. The communication overhead corresponding to each machine can be calculated as;

$$\text{Communication overhead for machine "K"} = C_K = \sum_{j=1}^{m} \sum_{i=1}^{n} a_i \cdot X_{ijK} \tag{5}$$

For two machines belonging to a single submodel, these messages are the events scheduled on the event calendar and therefore do not produce communication overhead. Therefore the total communication overhead on a single processor can be obtained as;

$$\text{Communication overhead for submodel '1'} = \sum_{K \in S_1} C_K - \sum_{k \in S_1} \sum_{j \in S_1} \sum_{i=1}^{n} a_i \cdot X_{ijk} \tag{6}$$

The first term calculates the messages that would be sent from all the machine processes in the manufacturing system if each submodel contained only one machine process. In the second term machine j and k are such that they both belong to a single submodel S_1. The second term subtracts the messages that would be sent inside the submodel. If it is assumed that each incoming message results in execution of a single event, then the main objective of the submodel creation process is;

$$\text{Min}\left\{ \underset{\forall S_l}{Max} \left\{ \text{Tc} \cdot \left[\sum_{k \in S_1} C_k - \sum_{k \in S_1} \sum_{j \in S_1} \sum_{i=1}^{n} a_i \cdot X_{ijk} \right] + Te \cdot \left[\sum_{k \in S_1} C_k \right] \right\} \right\} \tag{7}$$

Subject to a number of constraints given below;

1] $p \le m$; beyond this value of 'p' the overhead associated with the synchronization becomes enormous.

2] Avoid feedback loop structures in the submodel network because such situations lead to 'deadlock' during simulation execution.

3] Balance the total communication plus the execution overload on each processor executing a single submodel.

4] Take advantage of independent clusters of machines by assigning them to a single submodel thereby reducing the communication overload to zero.

5] Avoid "Join" topology of submodel network, because this topology requires the simulation event scheduling logic to wait for more event messages before it decides to proceed with the simulation execution.

Evaluation of this type of optimization is not easy. A straightforward approach is to use the above guidelines to come up with alternative submodel configurations and then evaluate the objective function for each configuration. The configuration which gives the least value for the objective function is the desired submodel configuration among those considered.

As submodels reside on different processors, synchronization between the submodels results in communication between the respective processors. Interprocessor communication can be achieved by a variety of communication strategies explained in Chapter III. The strategy used for the purpose of this research is explained in the following chapter.

CHAPTER IX

COMMUNICATION PROTOCOL DESIGN

Introduction

This chapter presents an overview of the communication protocol design process. For a general parallel processing application the communication protocols are designed by understanding the distribution of the problem computation over the processors, and the resulting communication requirements of each processor. As presented in Chapter III, for parallel discrete event simulation applications, a number communication protocols are available in the literature. A majority of protocols are proposed to effectively overcome 'deadlock' situations. But, from execution performance perspective, the above protocols for a 'deadlock' situation still give very marginal improvements in speedup and therefore are yet not very efficient approaches [Reed 1988]. And therefore, while creating submodels, this research considers only non-deadlocking situations. In general, non-deadlocking situations may preclude the user from further dividing a submodel into smaller pieces. It may appear that this restricts the improvement in speedup, but one must keep in mind that if the submodel is further divided the 'deadlock' situation will add enormous amount of communication resulting in a net decline in speedup.

Design of Communication Protocols

The submodel creation strategies carefully produce submodels without any 'deadlock' potential. And therefore, the communication protocols designed for the purpose of this research are not required to handle the 'deadlock' situations. This not only simplifies the design process, but also provides highly efficient interprocessor communication. For the purpose of implementation, two distinct communication protocols have been designed, viz. <u>forward</u> and <u>forward+backward</u>. Before discussing these communication protocols, the communication requirements during parallel implementation of discrete event simulation must be established. The following example explains the inter-processor communication process.

Figure 22 depicts a manufacturing system model divided into three submodels each containing two machines. There are two parts, P1 and P2 having two distinct routings as shown in Figure 22.

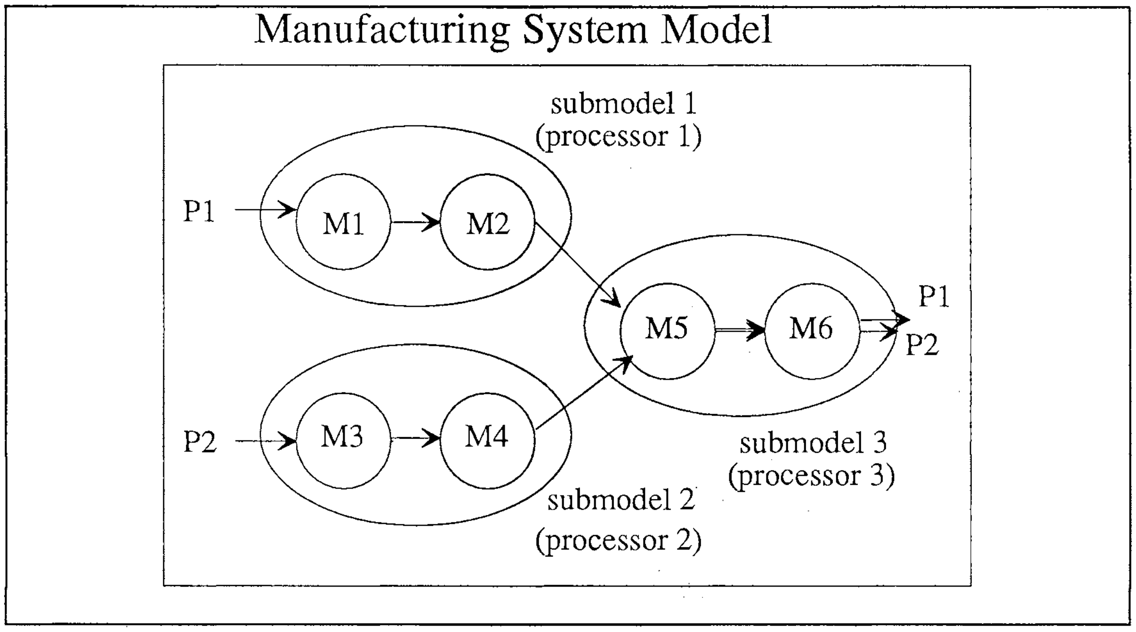

Figure 22. Manufacturing System Model Example

During the course of simulation, as part P1 finishes processing at machine M2, the next machine (M5) in its routing is not available on the current processor. Hence, the part P1 must travel to Processor 3 to simulate its operations on machine M5. This is achieved by sending a message containing part P1 to the submodel on processor 3. If these communication messages are sent and accepted without proper synchronization the simulation execution on processor 3 may violate the "Causality" principle and may produce erroneous results. The proposed protocols provide logical constructs for proper synchronization of processors. A brief explanation of each protocol is presented below.

Forward Protocol: This protocol is a collection of a number of constructs depicted in Figure 23. It consists of an input queue called 'InputBuffer' that enqueues the arriving parts, and an array called "ChannelTime" that holds the simulation clock times on the predecessor processors.

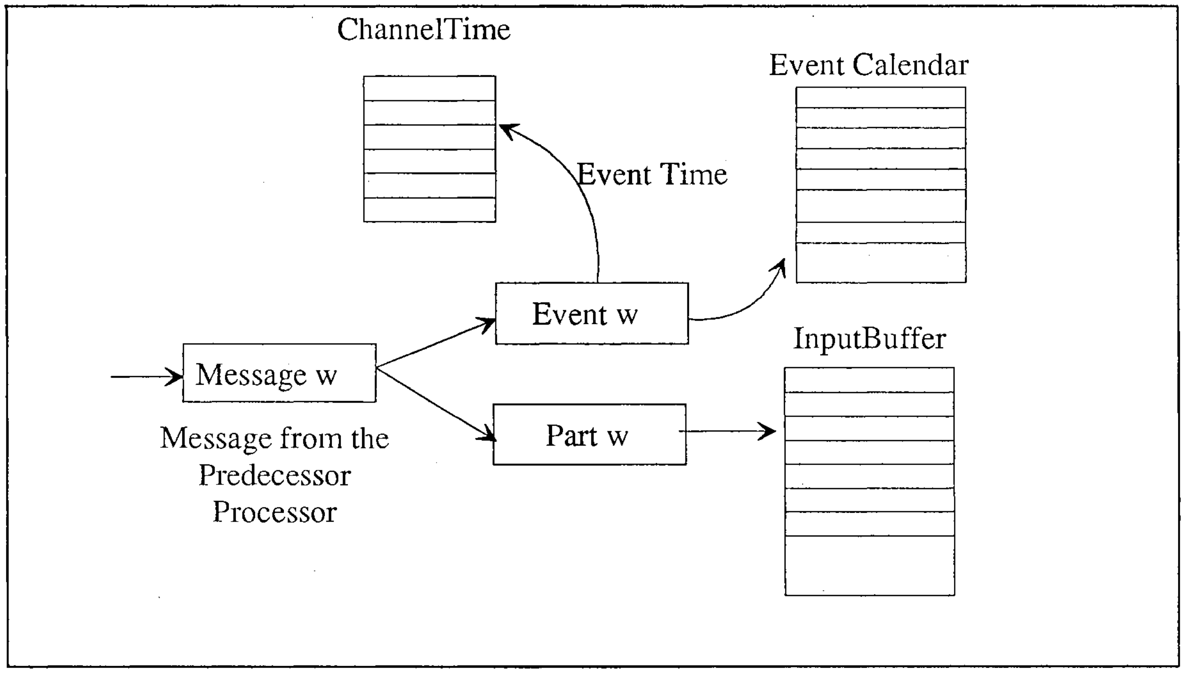

Figure 23. Processing of the Arrival Message

As a message containing a part arrives from the predecessors it is stored in the message buffer of the processor. The communication protocol realizes this arrival when statement "iprobe()" is executed. The arrived message is then received in the "Rsync" data structure and the corresponding part is recreated as an instance of class WFI (work flow item). This part is stored in the "InputBuffer" and the corresponding event is scheduled onto the event calendar. The arrived message also contains the simulation clock time on the predecessor processor. This information is used to update "ChannelTime" array. If the current simulation clock time is smaller than the minimum of the "ChannelTime" array the simulation is safely continued. If it is greater than or equal to the minimum of the 'ChannelTime" array the simulation is suspended until a new message containing new channel time makes the minimum channel time greater than the simulation clock. This arrangement conserves the "Causality" principle. The exact algorithm for the forward protocol is depicted in Figure 24.

Member function "perform" is responsible for the entire simulation execution. Function "ReceiverSender()" checks if the current processor has any predecessors or successors. If there are predecessors, then the simulation must synchronize with its predecessor by receiving messages, otherwise function "Simulate" (which operates without input synchronization messages) is invoked. Under the first case, "ReceiveMessages" function awaits to receive a synchronization message (message type 1). When such message is received the "UpdateChannelTime" function updates the corresponding channel time and the "ChannelTimeMin" variable. Variable "ChannelTimeMin" holds the time value up to which the current simulation can safely proceed. When the "ChannelTimeMin" increases the simulation end time, no input messages are expected at the input and therefore the program uses the "Simulate" function to proceed with the simulation. Member function "Perform" schedules an event if the current simulation clock time is less than "ChannelTimeMin", otherwise it waits for incoming synchronization messages. Function "SendMessage" creates a synchronization message by using the part details and the current simulation clock time. It then sends the synchronization message to the designated processor and deletes the part object.

```
void Simulation::Perform()
{ReceiverSender();
 if (receiver)
  { ChannelTimeMin = 0;
    ReceiveMessage();
    while(Sim_End_Time > ChannelTimeMin)
     {while(Sim_Clock < ChannelTimeMin)
       {ScheduleAnEvent(); } // End while
      ReceiveMessage();
     } // End of while Sim_End_Time
  }; // End of if receiver
 Simulate();
}; // End Perform

void Simulation::SendMessage(WFI* new_part,
int node)
  { // Create a Message from the received WFI
  WFImsg.id = new_part->ID;
  WFImsg.serial = new_part->Serial;
  WFImsg.step = new_part->At_Step;
  WFImsg.time = new_part->Time_Stamp;
  WFImsg.start_time = new_part->Start_Time;
  WFImsg.pid = my_node;

csend(1,&WFImsg,sizeof(WFImsg),node,0);
     delete new_part;
}; // Send Message Ends
```

```
void Simulation::ReceiveMessage()
{crecv(1,&WFImsg,sizeof(WFImsg));
UpdateChannelTime(WFImsg.pid,WFImsg.time);
  // Add the arrived part to the inputBuffer
WFI* new_part;
new_part = new WFI(WFImsg.id,WFImsg.time,
WFImsg.start_time,WFImsg.step,WFImsg.serial);
InputBuffer.ADD(new_part)
  // Add the event to the event queue
Event* new_event;
new_event = new
Event(PARTS+100,WFImsg.time);
Event_Q.ADD(new_event);
}; // End of Receive Message
void Simulation::UpdateChannelTime(int
channel, double time)
{int k;
 double min;
 ChannelTime[channel] = time;
 min = Sim_End_Time +100;
 for(k=0;k<receiver;k++)
 {if (ChannelTime[Ch[k]] < min)
   { min = ChannelTime[Ch[k]]; }; // End of if
 }  // End of for
 ChannelTimeMin = min;
}; // End of UpdateChannelTime
```

Figure 24. C++ Code for "forward" Communication Protocol

The simulation environment using Forward communication protocol waits for the incoming messages, and simulates the model until the "Causality" constraints are not violated (i.e. until the "Simulation Clock" < "ChannelTimeMin"). The simulation process then stops until a new input message is received. At the end of the simulation (of the submodel) the predecessor processors flag their successors that they should not expect any incoming communication. The simulation process uses the "simulate" function for the purpose of simulation after it receives the flags from all its predecessors.

Forward+backward Protocol : This protocol contains all the "Forward" communication constructs plus a number of added member functions depicted in Figure 25. In its forward portion it behaves identical to the communication using the "Forward" function. However, the backward portion is responsible for "Null" type messages [Chandy 1979] discussed in Chapter III. This demand driven backward portion of the protocol sends synchronization message to the bottleneck predecessor when the current processor is waiting for an input communication message. When the predecessor sends back its simulation clock time the current processor updates the "ChannelTimeMin" and progresses its simulation up to this simulation time. When all the predecessors have completed the simulation of their respective submodels, the current processor uses the "simulate" function. This added module in the communication protocol is implemented by using the following functions depicted in Figure 25.

Member function "Sync_Request()" sends a synchronization request to the bottleneck predecessor. Using "iprobe()" function it checks to see if a message is already waiting to be received in the communication buffer. If a message is present then it is received using "ReceiveMessage" function, otherwise by using "Find_R_Channel()" function the bottleckneck predecessor is identified, and a synchronization request message is sent to the bottleneck predecessor. The simulation processing halts until the bottleneck predecessor replies with its clock time. Since it is purely an asynchronous communication the count of number of messages received from the bottleneck predecessor is matched with the number of messages it has sent. If these two numbers do not match there are some messages in transit and therefore the channel time is not updated, and the simulation process waits until the transit message are received.

```
int Simulation::Find_R_Channel()
 {int min_Ch,k;
  double min;
  min_Ch = 0;
  min = Sim_End_Time +100;
  for(k=0;k<receiver;k++)
  {if (ChannelTime[Ch[k]] < min)
    { min = ChannelTime[Ch[k]];
      min_Ch = Ch[k];
    }; //End of min if
  } // End of for
  return min_Ch;
 }; //End of Find_R_Channel

void Simulation::Reply_Sync_Request()
 {int NID;
  if(iprobe(2))
  {crecv(2,&Rsync,sizeof(Rsync));
    NID = Rsync.Nid;
    Ssync.Nid = my_node;
    Ssync.time = Sim_Clock;
    Ssync.Nms = SM[NID];
    csend(3,&Ssync,sizeof(Ssync),NID,0);
   }; //End of if iprobe
 }; //End of Reply_Sync_Request
```

```
void Simulation::Sync_Request()
 {int ch;
  long WT;
  WT = mclock();
  while(1)
  {Reply_Sync_Request();
   if (iprobe(1))
    {ReceiveMessage();
     break;
    }
   if (mclock()-WT > 10)
    {ch = Find_R_Channel();
     if (ChannelTime[ch] < Sim_End_Time)
      {Rsync.Nid = my_node;
       csend(2,&Rsync,sizeof(Rsync),ch,0);
       crecv(3,&Ssync,sizeof(Ssync));
       if (Ssync.Nms == RM[Ssync.Nid])
        { UpdateChannelTime(Ssync.Nid,
           Ssync.time);
         }; //End of if
       break;
      }; //End of if Channel_Time
    }; //End of if mclock
   }; //End of while (1)
 }; //End of Sync_Request
```

Figure 25. C++ Code for "forward+backward" Communication Protocol

Function Reply_Sync_Request() replies to the synchronization request from a successor processor. It simply sends the current simulation clock time value, and the number of synchronization messages sent to the requesting processor, through the synchronization message. Function "Find_R_Channel" determines the bottleneck predecessors on which the simulation on the current processor is waiting for synchronization messages.

The simulation environment using Forward+backward communication protocol simulates the model until the "Causality" constraints are not violated (i.e. until the "Simulation Clock" < "ChannelTimeMin"). If there are no incoming messages available in the communication buffer, it synchronizes with the bottleneck predecessor so that it can continue the simulation process. The major difference between the two protocols is the demand of communication by the successor to its bottleneck predecessor. If all the processors are balanced in terms of incoming communication messages and the frequency of communication is relatively high, then both strategies will give almost the same execution performance. Otherwise, the "Forward+backward" will perform better than the "Forward" protocol. This can be seen from the example depicted in Figure 26.

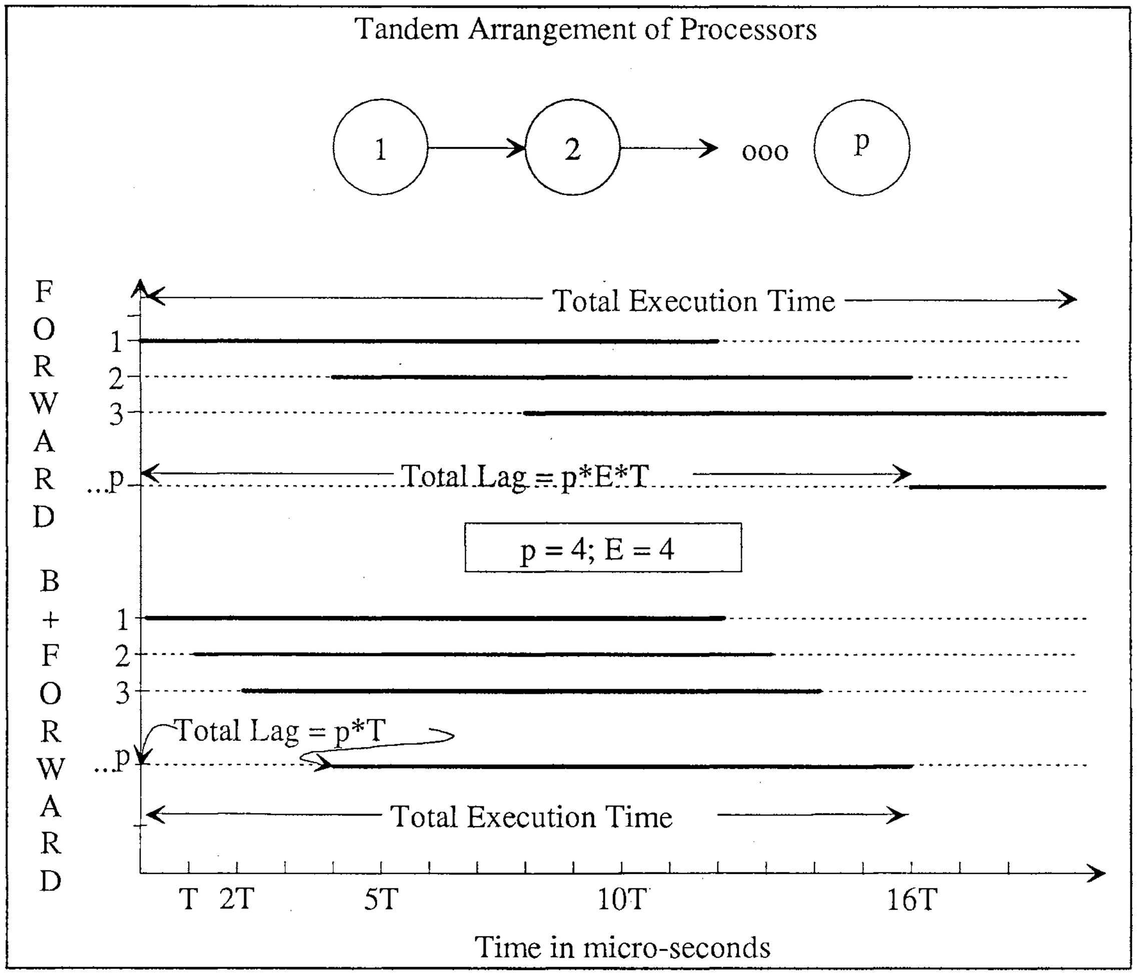

Figure 26. Execution Performance Comparison of Communication Protocols

Figure 26 depicts a tandem arrangement of 'p' processors, each having a single simulation submodel. Each submodel communicates with the successor after every 'E' internal events. If it takes 'T' microseconds to execute an event, then the external communication is sent every 'E*T' microseconds. The graph in Figure 26 depicts the execution performance of both protocols. Since the "Forward" protocol does not have any "demand" mechanism, the execution of each processor incrementally staggers by "E*T", resulting in a net "p*E*T" extra execution time. On the contrary, "Forward+backward" protocol can demand a synchronization message from its bottleneck predecessor, thereby enabling each processor to obtain the simulation clock time from its predecessor. This leads to substantially high overlap of their execution. Ideally, (with instantaneous synchronization) it is "p*T" instead of "p*E*T", and therefore the speedup factor can be substantially improved by using "Forward+backward" communication protocol. An experimental comparison of these two communication protocols is provided in Chapter X.

CHAPTER X

EXPERIMENTATION RESULTS

Introduction

This chapter presents the experimentation setup and the experimental results. The experimentation is accomplished by implementing the methodology outlined in Chapter V within the object oriented framework described in Chapter VI.

Verification of the Simulation Environment

This section presents the process of verification of the PDES environment designed for the experimentation. The verification process involves simulating a manufacturing system by using both the SLAM II simulation environment and the newly developed parallel discrete event simulation (PDES) environment. The SLAM II simulation output is then compared to the output obtained from the PDES environment. Verification of the PDES environment is primarily focused on the accuracy of event scheduling logic, statistics collection, and simulation results. In order to get an exact match between the two simulation results, the stochastic variables in the model such as arrival distributions, processing time distributions, etc., are specified to be deterministic constants. The manufacturing system depicted in Figure 27 is selected for verification of the PDES environment. This system has 5 machines and 3 part routings. The routings are deterministic and the parts do not require material handling for the movement between the machines. The arrival rates of the parts and the processing times at the machines are deterministic. This system is modeled using both SLAM II and the PDES environment. Both the PDES and the SLAM II models for this manufacturing system are outlined in Appendix C. Appendix C also provides the output results obtained from the simulations of these models.

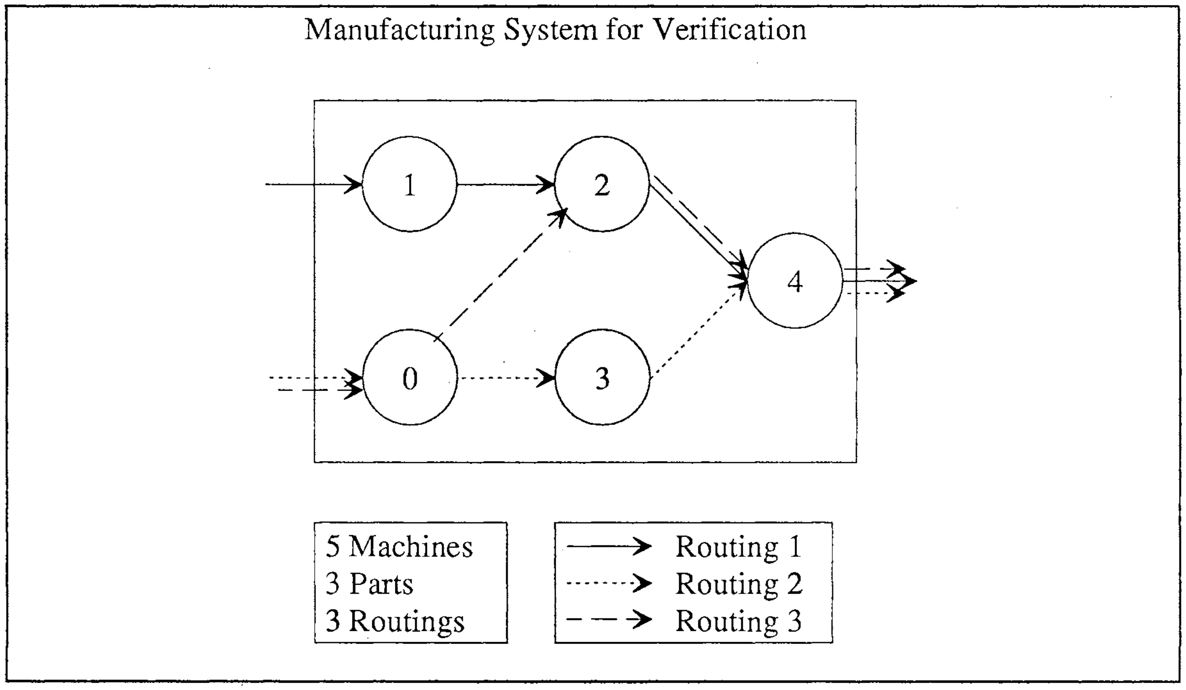

Figure 27. Manufacturing System Example for Environment Verification

Table 3 on the next page exhibits queue length, utilization, and waiting time statistics for all machines and time in system statistics for each part in the system. Queue length, utilization, and time in system figures for both environments have a very close match except for the minor differences in the last digit due to rounding off by the SLAM II environment. The difference in the waiting time statistics is attributed to the difference in the statistics collection mechanisms for the two environments. These differences are explained in Appendix C. This proves the correctness of the PDES environment designed for parallel discrete event simulation of manufacturing systems.

TABLE 3

VERIFICATION OF THE PDES ENVIRONMENT

System Element	Statistics	SLAM II	PDES Simulator
Machine 0	Utilization	1.000	1.000
	Q Length Avg.	10.526	10.526
	Q Length Std	5.783	5.782
	Wait Time Avg.	22.452	23.307
Machine 1	Utilization	1.000	1.000
	Q Length Avg.	6.686	6.685
	Q Length Std	3.889	3.888
	Wait Time Avg.	19.713	20.000
Machine 2	Utilization	0.950	0.950
	Q Length Avg.	3.15	3.149
	Q Length Std	1.894	1.894
	Wait Time Avg.	10.185	10.333
Machine 3	Utilization	0.651	0.650
	Q Length Avg.	0.020	0.020
	Q Length Std	0.140	0.139
	Wait Time Avg.	0.143	0.142
Machine 4	Utilization	0.910	0.910
	Q Length Avg.	10.399	10.399
	Q Length Std	6.975	6.974
	Wait Time Avg.	28.958	29.500
Part Type 0	Time in Sys. Avg.	50.6	50.600
	Time in Sys. Std.	22.2	22.154
	Time in Sys. Max	81.0	81.000
	Time in Sys. Min.	23.0	23.000
Part Type 1	Time in Sys. Avg.	50.0	50.000
	Time in Sys. Std.	26.9	26.879
	Time in Sys. Max	16.0	16.000
	Time in Sys. Min.	84.0	84.000
Part Type 2	Time in Sys. Avg.	55.3	55.333
	Time in Sys. Std.	23.4	23.352
	Time in Sys. Max	30.0	30.000
	Time in Sys. Min.	76.0	76.000

Experimentation Setup

The experimental design described in Chapter V specifies several simulation experiments. Table 4 describes a list of experiments and then corresponding experiment IDs used throughout the remainder of this dissertation. The first five experiments are designed to simulate five submodel network topologies of the submodel network. The sixth experiment, E6 is specifically designed to reveal the differences between the two communication protocols that are not evident in earlier experiments. In each of the above six experiments there are two levels of "Manufacturing System Size", "Communication Protocol", and "Manufacturing System Load" and six levels of the "Number of Processors" factor. Three simulation experiment replications are performed for each combination of factors. This results in one hundred and forty four simulations under each experiment. Each simulation input file further requires a new set of manufacturing system description parameters and random number seeds. This enormous amount of information is created by executing input file creation programs supplied in Appendix E.

TABLE 4

EXPERIMENTATION IDENTIFICATION

ID	Network Topology	Protocols (#s)	Size (#s)	Processors (#s)	Load (#s)	Reps (#s)	# of Sim Runs
E1	Clusters	F, B+F (2)	S, L (2)	1,..,32 (6)	L, H (2)	3	144
E2	Tandem	F, B+F (2)	S, L (2)	1,..,32 (6)	L, H (2)	3	144
E3	Fork	F, B+F (2)	S, L (2)	1,..,32 (6)	L, H (2)	3	144
E4	Join	F, B+F (2)	S, L (2)	1,..,32 (6)	L, H (2)	3	144
E5	Fork+Join	F, B+F (2)	S, L (2)	1,..,32 (6)	L, H (2)	3	144
E6	Tandem	F, B+F (2)	S, L (2)	1,..,32 (6)	L, H (2)	3	144

Input files creation programs generate twelve input files that are executed by using two levels of communication protocols. The input files specify the machines, parts, routings, arrival rate parameters, processing time parameters, and multiple random number seeds. Each experiment (144 simulations) is performed by executing the "mruns" shell script through "MRUNS" command. Figure 28 depicts the sequence of shell script execution. The "MRUNS" command processes the "mruns" shell script in the background. This enables the user to logout after "MRUNS" is executed. "mruns" script creates an "outputfile" output file which contains 144 execution time observations, each corresponding to a simulation run. Shell script "mruns" executes "12MODELS" file to create 12 input files. It then repeatedly executes "EXP" script to perform simulations for both large and small manufacturing systems on 1, 2, 4, 8, 16, and 32 processors. "EXP" script executes "exp32" and "exp256" scripts to accomplish small and large system size simulations respectively. Script "exp32" copies "model032.des" input file to the "model.des" file and the appropriate allocation file to "alloc.tab" file. It then executes "chost" program to accomplish the simulation as explained by the experimental setup in Chapter VI. Script "MAKEexp" uses "spades" and "COF" scripts to delete the unwanted "pdes.*" files and generate the final "*.exp" output file which contains the model, the input, and the output information. Appendix F supplies all the shell scripts used during the process of experimentation.

Experimentation Results

This section provides the results of the six experiments depicted in Table 4. The first five of the above experiments have relatively high frequency of interprocessor communication. The interprocessor communication is designed to be infrequent for the last experiment in order to illustrate the differences between the two communication protocols. Frequency of communication is a result of the ratio of the number of events that lead to interprocessor communication, to the total number of events scheduled by the event calendar. Therefore, the frequency of communication can be modified by varying the routing structure or the arrival distribution parameters.

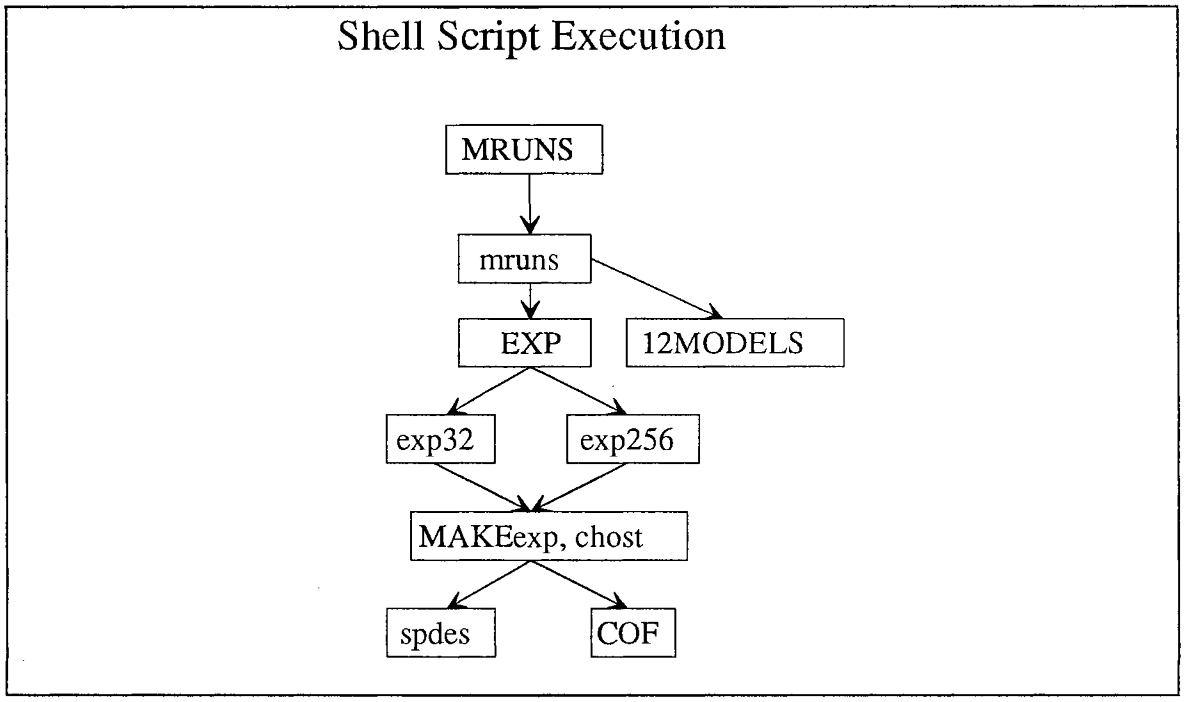

Figure 28. Shell Script Execution Sequence

The exact description of each experiment, its experimental factors and the levels of each experimental factor is provided in Chapter V. Simulation execution time and speedup are the primary performance measures used for the analysis of parallel discrete event simulation. Analysis of each experiment consists of presentation of the output in a table similar to the design of experiments table, conducting the analysis of variance test (ANOVA) on the experimental data, and the interpretation of the behavior of the performance variable within the scope of the experiment. The SAS output of the

ANOVA for all experiments are provided in Appendix G. Headings L, S, P of the first three columns of each table displaying results of an experiment correspond to the three factors in the experimental design, viz. Manufacturing System Load, Manufacturing System Size, and Number of Processors used for the implementation. Manufacturing System Load levels 0 and 1 correspond to low load and high load respectively. Manufacturing System Size levels 0 and 1 respectively correspond to small size and large size. Number of processors levels 0, 1, 2, 3, 4, and 5 correspond to 1, 2, 4, 8, 16, and 32 processors respectively.

In order to understand the relationship of the factors affecting simulation execution, an approximate mathematical model of the simulation execution process has been developed. This model neglects communication delays and attempts to explain the behavior of the execution time of the PDES application over all combinations of the levels of experimental factors. Following is a brief description of this model.

<u>Notation:</u>

N	: Number of events scheduled during entire simulation
n	: Average queue length of the event calendar
Te	: Time for executing a single event
Ts	: Time for performing a single comparison on the event calendar
Ti	: Time for simulation startup and finish
T	: Execution time per processor

For single processor calculation, the main execution time components are, the time for simulation startup and finish, and the time for event execution and event queueing during the entire simulation. Placing a newly scheduled event at an appropriate place on an event calendar with average length 'n' requires "n/2" average comparisons. This leads to the following equation for total simulation execution time for single processor application.

$$T = Ti + N\,(\,Te + \frac{n}{2}\cdot Ts) \qquad (8)$$

For "p" processors, there are "N/p" number of events scheduled on each processor, the average event calendar length becomes "n/p", and the startup and finish time also becomes "Ti/p". This leads to the following Equation for total simulation execution time for "p" processor application.

$$T = \frac{Ti}{p} + \frac{N}{p}\,(\,Te + \frac{n}{2\cdot p}\cdot Ts) \qquad (9)$$

Therefore the expected speedup is;

$$\text{Speedup} = \frac{Ti + N\,(\,Te + \frac{n}{2}\cdot Ts)}{\frac{Ti}{p} + \frac{N}{p}\,(\,Te + \frac{n}{2\cdot p}\cdot Ts)} \qquad (10)$$

Equation (10) can be simplified as;

$$\text{Speedup} = \frac{p}{1 - X\,(\,1 - \frac{1}{p})} \qquad (11)$$

Where "X" is defined as;

$$X = \frac{N\cdot n\cdot Ts}{2\cdot(Ti + N\,(\,Te + \frac{n}{2}\cdot Ts))} \qquad (12)$$

And the efficiency becomes;

$$\text{Efficiency} = \frac{1}{1 - X\,(\,1 - \frac{1}{p})} \qquad (13)$$

Equations (10) - (13) will be used for providing the explanation for the behavior of speedup and efficiency for all the following experiments.

Submodel Network Topology - Independent Clusters (E1)

This experiment consists of the simulation of a manufacturing system with independent "Clusters" of machines. Table 5 is an "Analysis of Variance" (ANOVA) summary table that furnishes the variance analysis of the important factors and their interactions. Factor communication protocol is not found to be significant because the "independent cluster" topology of processors does not require interprocessor communication during execution of the simulation. System load, system size, number of processors used for simulation, and their higher order interactions are found to be significant. System load and system size directly alter the computational requirements of a PDES application. They alter the number of events executed during the simulation, i.e. the value of "N" in Equation 9. It is evident from Equation 9 that the number of processors used for PDES application "p" is also a significant factor. This explains the influences of these factors on the execution time of a PDES application. The higher order interactions can also be explained by referring to Equation 9.

TABLE 5

CLUSTER TOPOLOGY (E1) - ANOVA SUMMARY

Factor	df	OSL*	$\alpha = 0.01$
Commu. Protocol (C)	1	0.1874	Do not Reject
System Load (L)	1	0.0001	Reject
System Size (S)	1	< 0.0001	Reject
# of Processors (p)	5	< 0.0001[1]	Reject
(L X S)	1	0.0001	Reject
(L X p)	5	0.0001	Reject
(S X p)	5	< 0.0001	Reject
(L X S X p)	5	0.0001	Reject

* OSL - Observed Significance Level

Before explaining the higher order interactions it is important to note that, if the output is dependent on multiplication or division of two independent variables, then their interaction is significant as the simple effect (the difference in response between two levels of a factor at a combination of levels of other factors) will be always dependent on the value of the other variable. System load influences the number of events scheduled during simulation, "N". System size influences both the event calendar length "n" and the value of "N". The number of processors used for PDES implementation is defined to be "p" by the above notation. This explains why factor "number of processors" (p) has interaction with system load (L) and system size (S) as depicted in Table 5. Similarly the third term in Equation 9 involves multiplication of three variables which depend on the factors (p), (L), (S) resulting in a three way interaction among them.

As the factor communication protocol is not significant, the "Table of Means" can be obtained by averaging over all (two) levels of the communication protocol. This results in Table 6 as the "Table of Means" with 24 means out of 144 observations. The execution times depicted in these tables are the result of 3 replications of each combination of the significant experimental factors averaged over two levels of communication protocol. The simulation execution times in Table 6 are expressed in milliseconds.

Table 6 depicts the speedup and efficiency values for each combination of the three significant factors. The effect of changing the number of processors on the speedup value at each combination of system load and size is explained by Figures 29 and 30. Figure 29 depicts a graph of speedup against the number of processors used for parallel implementation. An inspection of these figures yields the following observations:

1] The speedup increases as the number of processors is increased. This is explained by observing Equation 11. Variable "p" in the numerator of the speedup equation makes speedup increase as more processors are used for parallel implementation.

TABLE 6

CLUSTER TOPOLOGY (E1) - TABLE OF MEANS

			Exe. Time	Speed Up	Efficiency				Exe. Time	Speed Up	Efficiency
L	S	p	Mean (msec)	Mean	Mean	L	S	p	Mean(msec)	Mean	Mean
0	0	1	98590	1.00	100.00%	1	0	1	120916	1.00	100.00%
0	0	2	42159	2.34	116.93%	1	0	2	49771	2.43	121.47%
0	0	4	19562	5.04	126.00%	1	0	4	22526	5.37	134.20%
0	0	8	10096	9.77	122.07%	1	0	8	11250	10.75	134.35%
0	0	16	6531	15.10	94.35%	1	0	16	7042	17.17	107.32%
0	0	32	5451	18.09	56.52%	1	0	32	5664	21.35	66.71%
0	1	1	809752	1.00	100.00%	1	1	1	1267016	1.00	100.00%
0	1	2	305914	2.65	132.35%	1	1	2	435794	2.91	145.37%
0	1	4	127792	6.34	158.41%	1	1	4	168310	7.53	188.20%
0	1	8	58093	13.94	174.24%	1	1	8	72268	17.53	219.15%
0	1	16	29538	27.41	171.34%	1	1	16	34777	36.43	227.70%
0	1	32	18115	44.70	139.69%	1	1	32	20170	62.82	196.30%

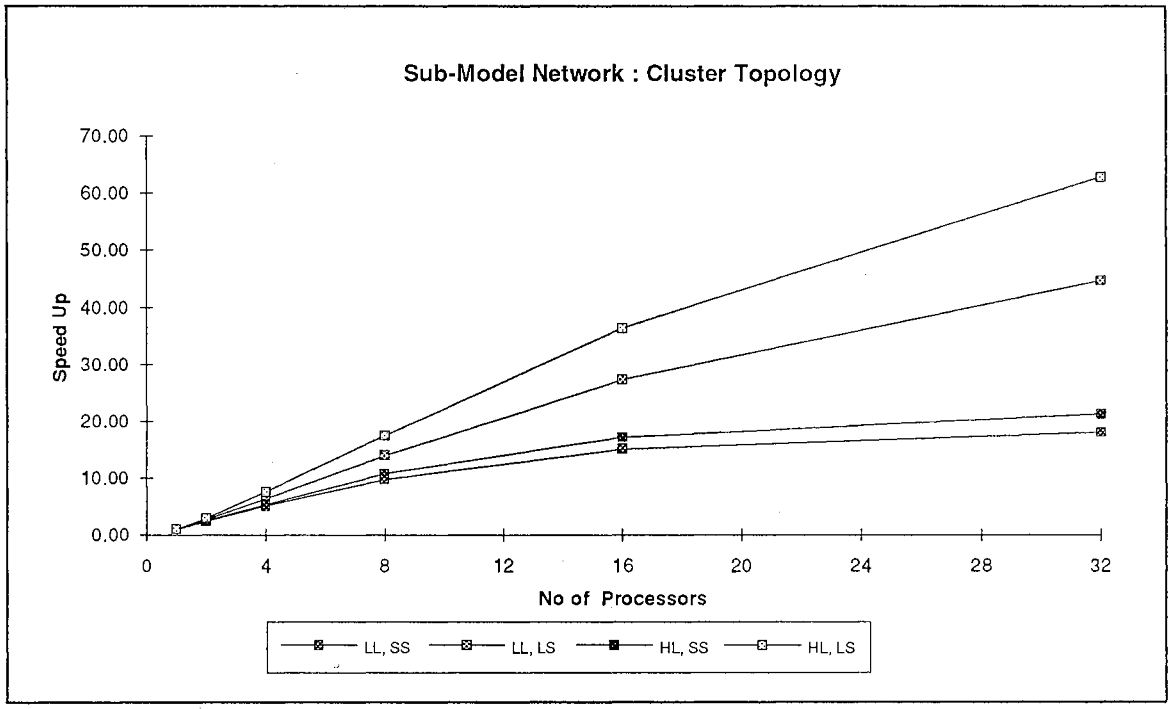

Figure 29. Speedup Curves for "Cluster" Topology (E1)

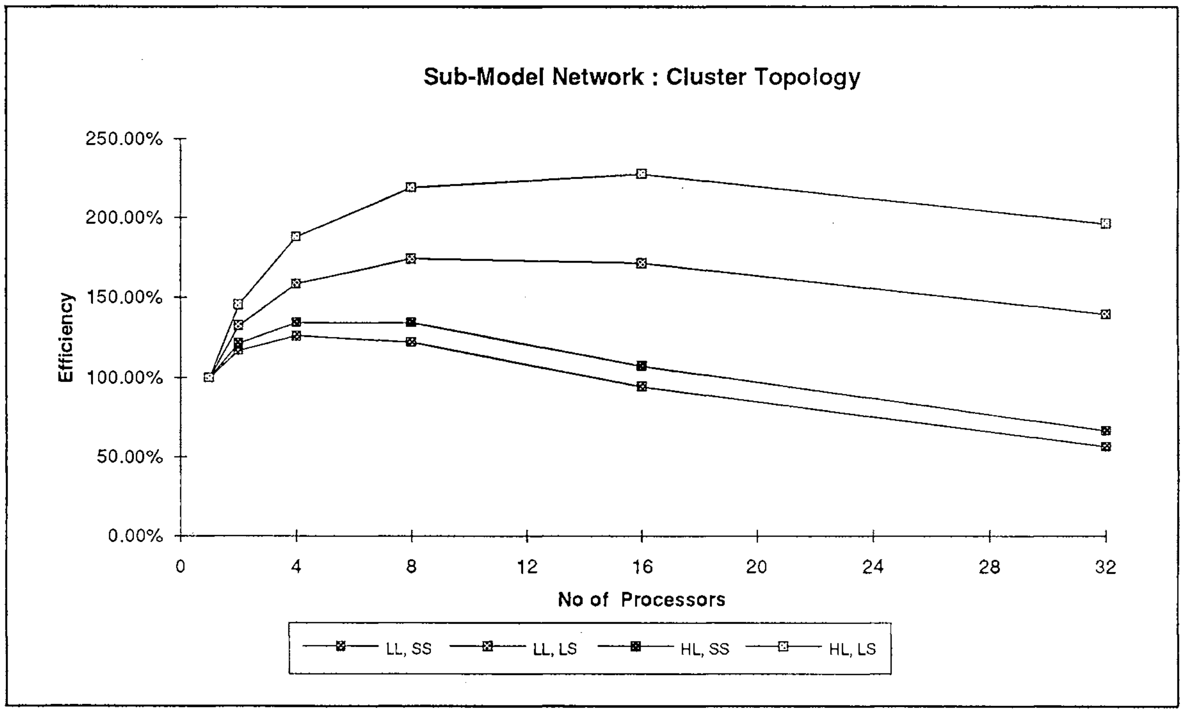

Figure 30. Efficiency Curves for "Cluster" Topology (E1)

2] The higher the system load the better the speedup. This can be explained by Equations 11 and 12. Higher system load increases "N" thereby increasing the value of "X" in Equation 12. It is clear from Equation 11 that an increase in "X" would result in greater speedup.

3] The higher the system size, the better the speedup. This can also be explained by Equations 11 and 12. Higher system size increases both "n" and "N" thereby increasing "X" in Equation 12. It is clear from Equation 11 that an increase in "X" would result in higher speedup.

4] High system load and large system size case gives highest speedup.

6] Low system load and small system size case gives lowest speedup.

7] As the number of processors is increased, at first the efficiency curves climb, but beyond 8 processors they slowly decline.

It is important to note that the above Equations (8 to 13) are gross approximations for obtaining PDES execution performance measures and therefore would not accurately predict the performance measures in each case. They are used to explain the dominant relationships between the experimental factors.

Submodel Network Topology - Tandem (E2)

This experiment consists of simulations of a manufacturing system with "Tandem" network of submodels. Table 7 is an ANOVA summary table that furnishes the variance analysis of the important factors and their interactions.

TABLE 7

TANDEM TOPOLOGY (E2) - ANOVA SUMMARY

Factor	df	OSL	$\alpha = 0.01$
Commu. Protocol (C)	1	0.0293	Do not Reject
System Load (L)	1	0.0001	Reject
System Size (S)	1	< 0.0001	Reject
# of Processors (p)	5	< 0.0001	Reject
(L X S)	1	0.0001	Reject
(L X p)	5	0.0001	Reject
(S X p)	5	< 0.0001	Reject
(L X S X p)	5	0.0001	Reject

System load, system size, number of processors used for simulation, and their higher order interactions are found to be significant. The explanation for the significance of these factors and their interactions is identical to the explanation provided for experiment E1. However, a major difference between experiments E1 and E2 is that experiment E2 involves interprocessor communication during simulation execution and therefore communication protocols have a potential to be a statistically significant factor. In fact at $\alpha = 0.05$ it becomes significant. Low communication frequency can very easily make communication protocol a significant factor. In the next paragraph Equations (14)-(16) are developed for explaining the influence of communication load on the parallel implementation of the simulation.

The interprocessor communication during simulation execution causes an additional delay in the execution process. If "Tc" is the time required for a single communication, each processor transacts "C" communications during simulation execution, and "Tw" is the total waiting time for communication during the simulation, then the speedup and efficiency Equations (4) and (5) should be modified as shown below;

$$\text{Speedup} = \frac{p}{1 - X\left(1 - \frac{1}{p}\right) + Y \cdot p} \qquad (14)$$

Where Y is defined as;

$$Y = \frac{Tc \cdot C + Tw}{Ti + N\left(Te + \frac{n}{2} \cdot Ts\right)} \qquad (15)$$

and the efficiency becomes;

$$\text{Efficiency} = \frac{1}{1 - X\left(1 - \frac{1}{p}\right) + Y \cdot p} \qquad (16)$$

In experiment E2, as factor communication protocol is not significant, the "Table of Means" can be obtained by averaging over all (two) levels of the communication protocol. This results in Table 8 as the "Table of Means" with 24 means out of 144 observations.

TABLE 8

TANDEM TOPOLOGY (E2) - TABLE OF MEANS

			Exe. Time	Speed Up	Efficiency					Exe. Time	Speed Up	Efficiency
L	S	p	Mean	Mean	Mean		L	S	p	Mean	Mean	Mean
0	0	1	149145	1.00	100.00%		1	0	1	187711.8	1.00	100.00%
0	0	2	68540.67	2.18	108.80%		1	0	2	82499.83	2.28	113.76%
0	0	4	34672.17	4.30	107.54%		1	0	4	40594.33	4.62	115.60%
0	0	8	19520.5	7.64	95.51%		1	0	8	21925.17	8.56	107.02%
0	0	16	12869.67	11.59	72.43%		1	0	16	13467.5	13.94	87.11%
0	0	32	10524	14.17	44.29%		1	0	32	11057.5	16.98	53.05%
0	1	1	1445868	1.00	100.00%		1	1	1	2302272	1.00	100.00%
0	1	2	569338.8	2.54	126.98%		1	1	2	827256.2	2.78	139.15%
0	1	4	247882.8	5.83	145.82%		1	1	4	330412.2	6.97	174.20%
0	1	8	116340.5	12.43	155.35%		1	1	8	146349.7	15.73	196.64%
0	1	16	60395	23.94	149.63%		1	1	16	73186.33	31.46	196.61%
0	1	32	34986.67	41.33	129.14%		1	1	32	40706.5	56.56	176.74%

Table 8 depicts the speedup and efficiency values for each combination of the three significant factors. Equations 14, 15, and 16 are used for explaining the behavior of speedup and efficiency curves. As the interprocessor frequency is relatively high, total waiting time ""Tw"" for communication is minimal. Thus, for "Tandem" topology the value of "Y" becomes very small as compared to the rest of the terms in the denominator and therefore the term "Y*p" can be omitted from the denominator. The effect of changing the number of processors on the speedup and efficiency values at each combination of the system load and the system size is explained by Figures 31 and 32. An inspection of these figures yields the following observations.

1] The speedup increases as the number of processors is increased. This is explained by observing Equation 14. The "p" in the numerator makes speedup increase as the number of processors are increased.

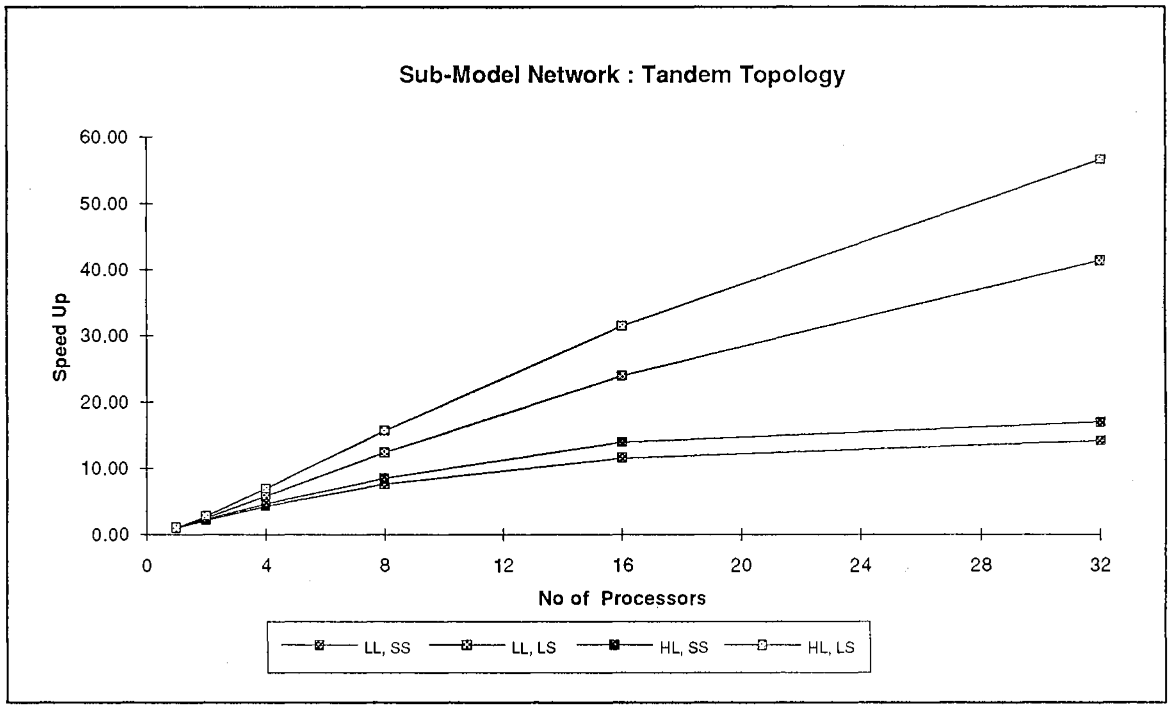

Figure 31. Speedup Curves for "Tandem" Topology (E2)

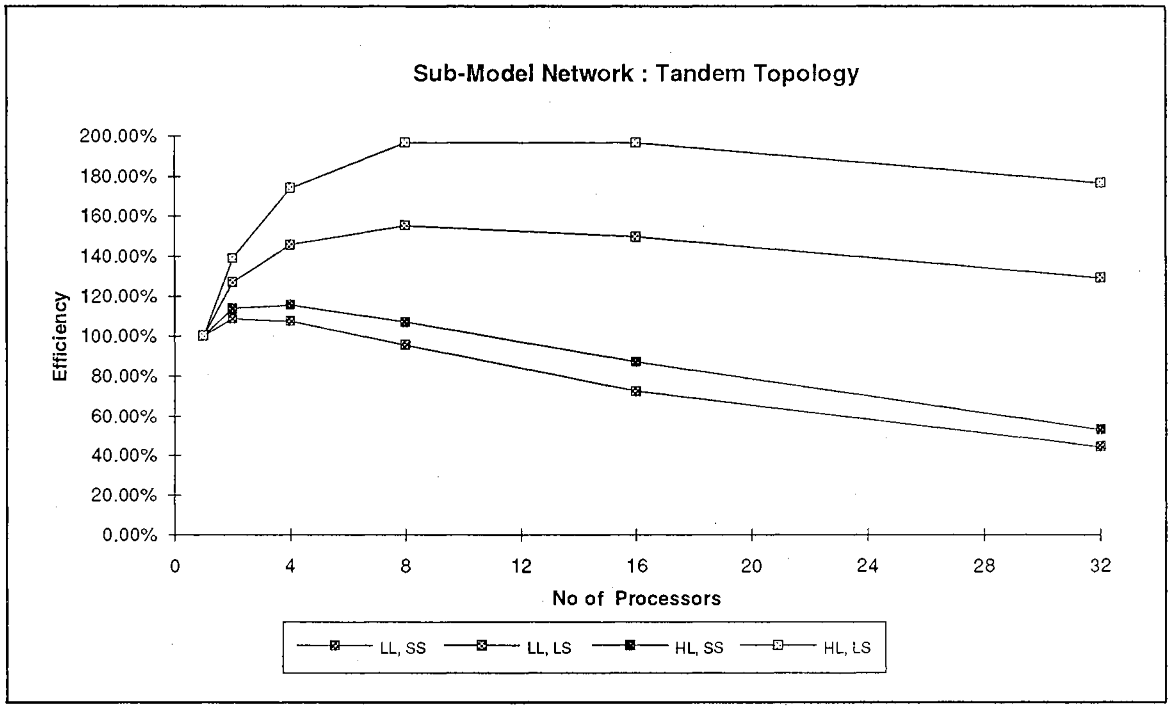

Figure 32. Efficiency Curves for "Tandem" Topology (E2)

2] The higher the system load, the better the speedup. This can be explained by Equations 14 and 15. Higher system load increases "N" thereby increasing "X" in the Equation 12. It is clear from Equation 14 that an increase in "X" would result in higher speedup.

3] The higher the system size, the better the speedup. This can be explained by Equations 11 and 12. Higher system size increases both "n" and "N" thereby increasing "X" in Equation 12. It is clear from Equation 11 that an increase in "X" would result in higher speedup.

4] High system load and large system size case gives highest speedups.

6] Low system load and small system size case gives lowest speedups.

7] As the number of processors is increased, at first the efficiency curves climb but beyond 8 processors they slowly decline.

Submodel Network Topology - Fork (E3)

This experiment consists of the simulation of a manufacturing system with "Fork" network of submodels. Table 9 is an ANOVA summary table that furnishes the variance analysis of the important factors and their interactions. In this experimentation, factors system load, system size, number of processors used for simulation, and their higher order interactions are found to be significant. The explanation for the significance of these factors and their interactions is identical to the explanation provided for experiment E1. A major difference between experiments E1 and E3, is that experiment E3 involves interprocessor communication during simulation execution, and therefore factor communication protocol has a potential to be a statistically significant factor. However, as the frequency of communication is relatively high throughout the experimentation it does not become a significant factor. Low communication can very easily cause communication protocol to be a significant factor.

As factor communication protocol is not significant, the "Table of Means" for experiment E3 can be obtained by averaging over all (two) levels of the communication protocol. This results in Table 10 as the "Table of Means" with 24 means out of 144 observations.

TABLE 9

FORK TOPOLOGY (E3) - ANOVA SUMMARY

Factor	df	OSL	$\alpha = 0.01$
Commu. Protocol (C)	1	0.0544	Do not Reject
System Load (L)	1	0.0001	Reject
System Size (S)	1	< 0.0001	Reject
# of Processors (p)	5	< 0.0001	Reject
(L X S)	1	0.0001	Reject
(L X p)	5	0.0001	Reject
(S X p)	5	< 0.0001	Reject
(L X S X p)	5	0.0001	Reject

Table 10 depicts the speedup and efficiency values for each combination of the three significant factors. Equations 14, 15, and 16 are used for explaining the behavior of speedup and efficiency curves. As the interprocessor frequency is relatively high, total waiting time "Tw" for communication is minimal. Thus, for "Fork" topology value of "Y" becomes very small as compared to the rest of the terms in the denominator and therefore term "Y*p" can be omitted from the denominator. The effect of changing the number of processors on the speedup and efficiency values at each combination of system load and size are explained by Figures 33 and 34. An inspection of these figures yields the following observations.

1] The speedup increases as the number of processors is increased. This is explained by observing Equation 14. The "p" in the numerator makes speedup increase as the number of processors is increased.

2] The higher the system load, the better the speedup. This can be explained by Equations 14 and 15. Higher system load increases "N" thereby increasing "X" in Equation 12. It is clear from Equation 14 that an increase in "X" would result in higher speedups.

3] The higher the system size, the better the speedup. This can also be explained by Equations 11 and 12. Higher system size increases both "n" and "N" thereby increasing "X" in Equation 12. It is clear from Equation 11 that an increase in "X" would result in higher speedups.

TABLE 10

FORK TOPOLOGY (E3) - TABLE OF MEANS

			Exe. Time	Speed Up	Efficiency
L	S	p	Mean	Mean	Mean
0	0	1	147805.5	1.00	100.00%
0	0	2	71544.5	2.07	103.30%
0	0	4	35703.33	4.14	103.50%
0	0	8	23029.83	6.42	80.23%
0	0	16	17716.17	8.34	52.14%
0	0	32	16993.5	8.70	27.18%
0	1	1	1372184	1.00	100.00%
0	1	2	562948.2	2.44	121.87%
0	1	4	249852.5	5.49	137.30%
0	1	8	136602	10.05	125.56%
0	1	16	94564	14.51	90.69%
0	1	32	81648.5	16.81	52.52%
1	0	1	184953.8	1.00	100.00%
1	0	2	85019.83	2.18	108.77%
1	0	4	41431.17	4.46	111.60%
1	0	8	25939.33	7.13	89.13%
1	0	16	19727	9.38	58.60%
1	0	32	18506.33	9.99	31.23%
1	1	1	2178261	1.00	100.00%
1	1	2	802924.2	2.71	135.65%
1	1	4	327408.3	6.65	166.33%
1	1	8	169148.7	12.88	160.97%
1	1	16	112975.8	19.28	120.50%
1	1	32	95488.67	22.81	71.29%

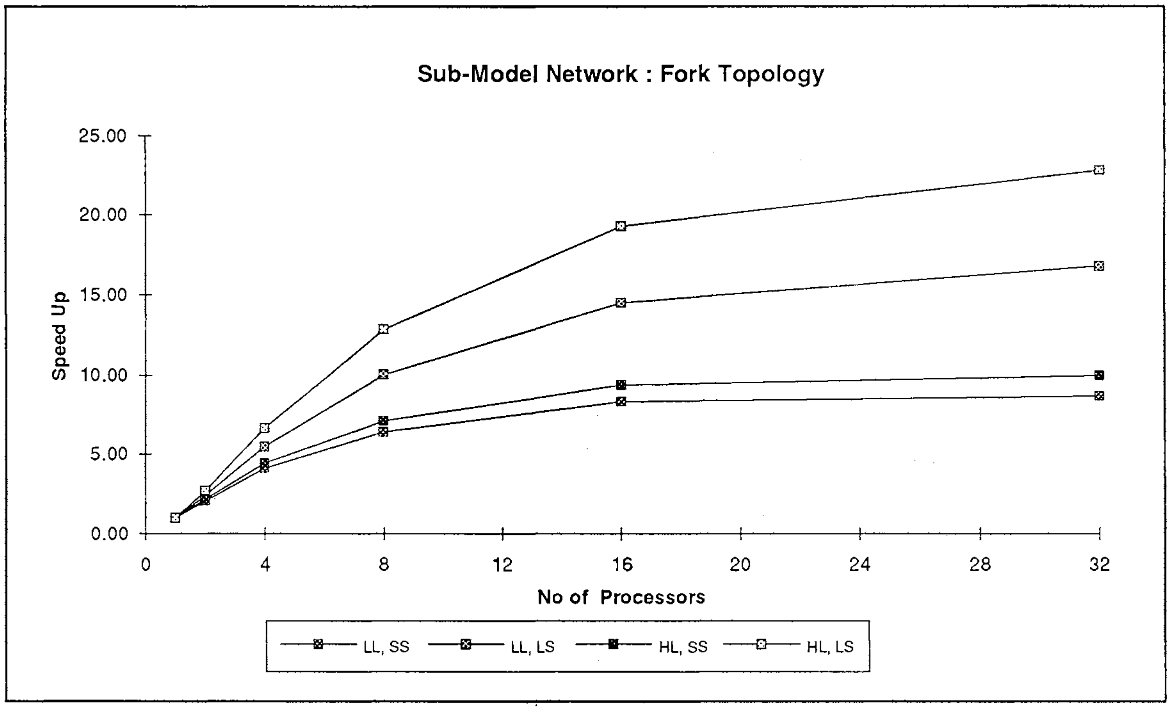

Figure 33. Speedup Curves for "Fork" Topology (E3)

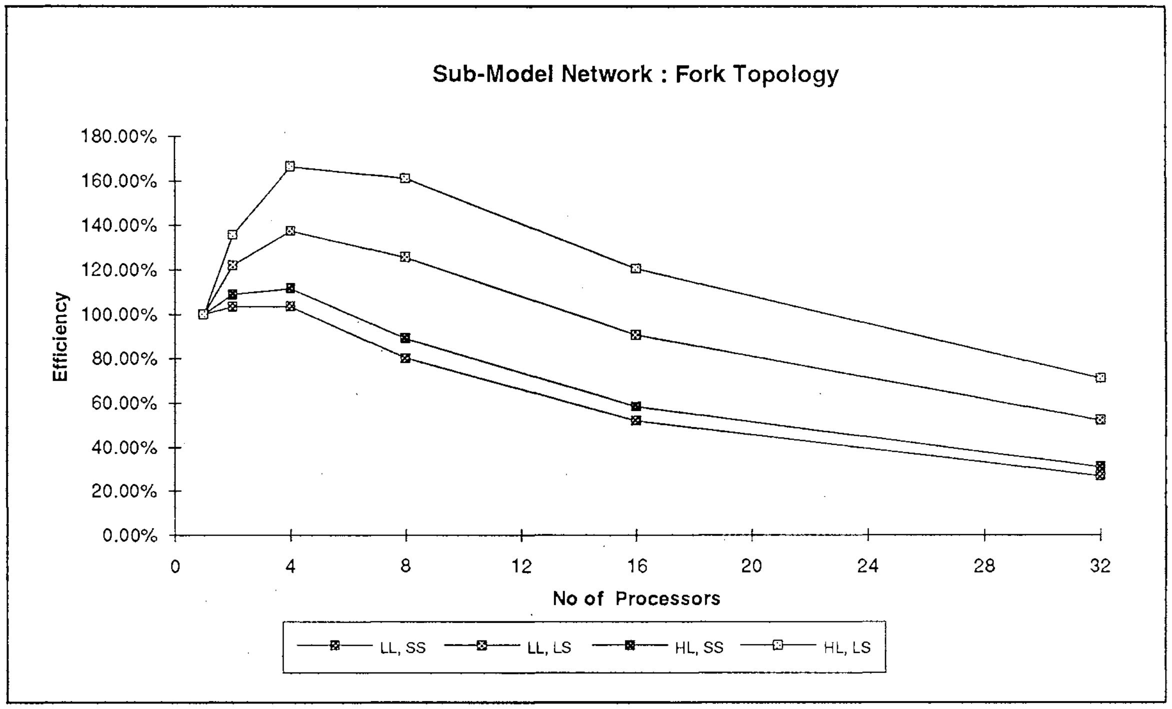

Figure 34. Efficiency Curves for "Fork" Topology (E3)

4] High system load and large system size case gives highest speedups.

6] Low system load and small system size case gives lowest speedups.

7] As the number of processors is increased, at first the efficiency curves climb but beyond 4 processors they slowly start falling.

8] For 32 and 16 processor implementations the speedup values are substantially lower than the corresponding speedups for experiments E1 and E2.

Submodel Network Topology - Join (E4)

This experiment consists of the simulation of a manufacturing system with "Join" network of submodels. Table 11 is an ANOVA summary table that furnishes the variance analysis of the important factors and their interactions. In this experimentation factors system load, system size, number of processors used for simulation, and their higher order interactions are found to be significant. The explanation for the significance of these factors and their interactions is identical to the explanation provided for experiment E3. A major difference between experiments E3 and E4, is that unlike E3 in experiment E4 each processor must wait for a message from all its predecessor processors for synchronization during the simulation execution, and therefore an unbalanced communication between the predecessors can given far better performance by "forward+backward" protocol. This would result in the communication protocol being a statistically significant factor. However, as the communication patterns are well balanced in this case, both protocols perform equally well and communication protocol is not a significant factor.

As factor communication protocol is not significant, the "Table of Means" for experiment E4 can be obtained by averaging over all (two) levels of the communication protocol. This results in Table 12 as the "Table of Means" with 24 means out of 144 observations. Table 12 depicts the speedup and efficiency values for each combination of the three significant factors. Equations 14, 15, and 16 are used for explaining the behavior of speedup and efficiency curves. As the interprocessor frequency is relatively high, the total waiting time "Tw" for communication is minimal. Thus, for "Join" topology the value of "Y" becomes very small as compared to the rest of the terms in the denominator and therefore the term "Y.p" can be omitted from the denominator.

TABLE 11

JOIN TOPOLOGY (E4) - ANOVA SUMMARY

Factor	df	OSL	$\alpha = 0.01$
Commu. Protocol (C)	1	0.2148	Do not Reject
System Load (L)	1	0.0001	Reject
System Size (S)	1	< 0.0001	Reject
# of Processors (p)	5	< 0.0001	Reject
(L X S)	1	0.0001	Reject
(L X p)	5	0.0001	Reject
(S X p)	5	< 0.0001	Reject
(L X S X p)	5	0.0001	Reject

The effect of changing the number of processors on the speedup and efficiency values at each combination of system load and size are explained by Figures 35 and 36. An inspection of these Figures yield the following observations.

1] The speedup increases as the number of processors is increased. This is explained by observing Equation 14. The "p" in the numerator makes speedup increase as the number of processors is increased.

TABLE 12

JOIN TOPOLOGY (E4) - TABLE OF MEANS

			Exe. Time	Speed Up	Efficiency				Exe. Time	Speed Up	Efficiency
L	S	P	Mean	Mean	Mean	L	S	P	Mean	Mean	Mean
0	0	1	147769	1.00	100.00%	1	0	1	183770.8	1.00	100.00%
0	0	2	71129	2.08	103.87%	1	0	2	84720.5	2.17	108.46%
0	0	4	35427.17	4.17	104.28%	1	0	4	41132.17	4.47	111.70%
0	0	8	22361	6.61	82.60%	1	0	8	25001.33	7.35	91.88%
0	0	16	18462.33	8.00	50.02%	1	0	16	19951.83	9.21	57.57%
0	0	32	17211.67	8.59	26.83%	1	0	32	18436.83	9.97	31.15%
0	1	1	1367297	1.00	100.00%	1	1	1	2159951	1.00	100.00%
0	1	2	560468.3	2.44	121.98%	1	1	2	797642	2.71	135.40%
0	1	4	250259.7	5.46	136.59%	1	1	4	326478.7	6.62	165.40%
0	1	8	136733.8	10.00	125.00%	1	1	8	168557.8	12.81	160.18%
0	1	16	94812.17	14.42	90.13%	1	1	16	112704.3	19.16	119.78%
0	1	32	82066.83	16.66	52.06%	1	1	32	95771.33	22.55	70.48%

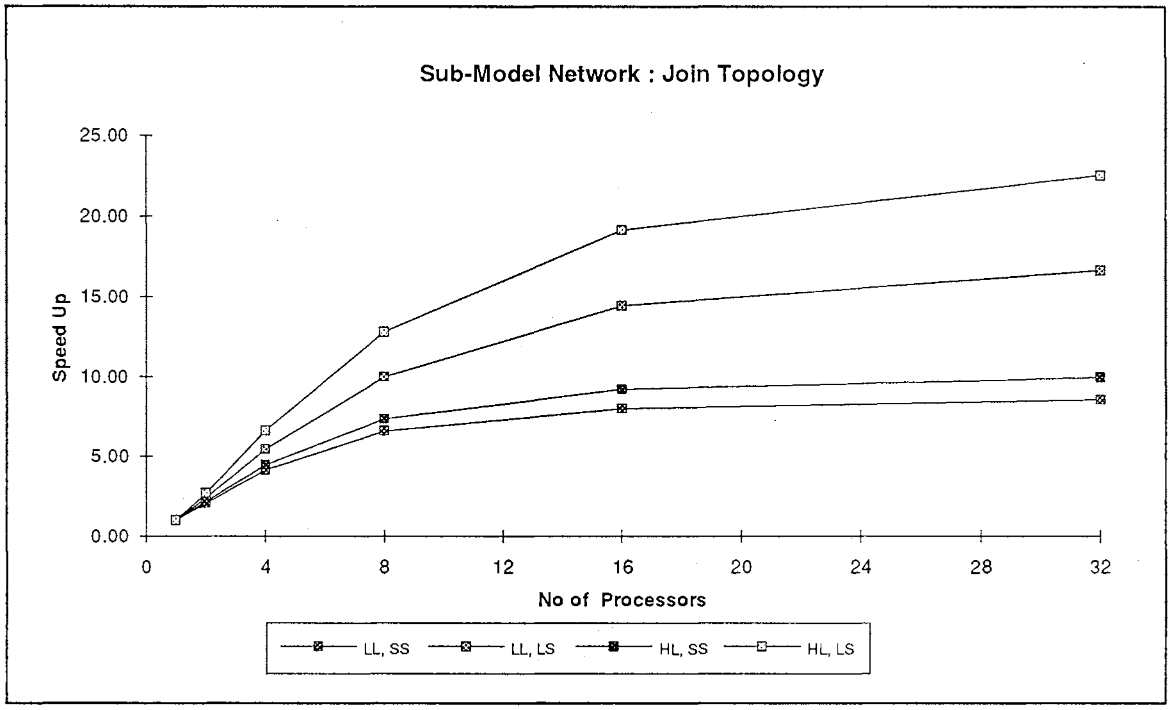

Figure 35. Speedup Curves for "Join" Topology (E4)

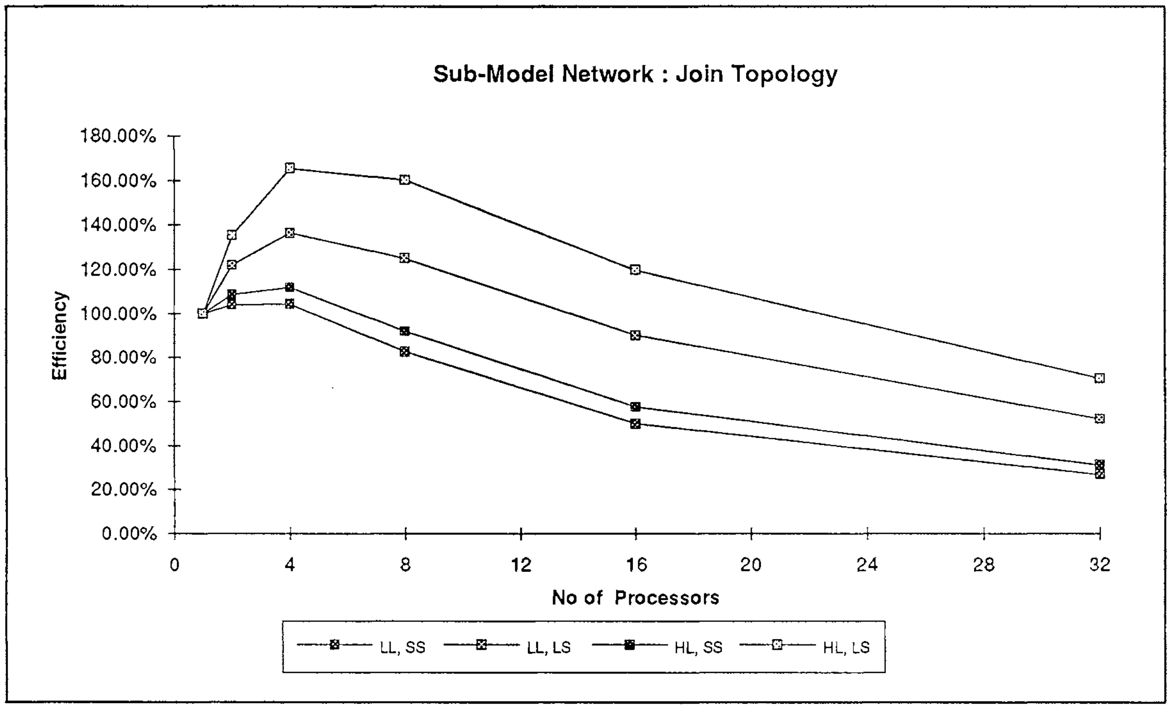

Figure 36. Efficiency Curves for "Join" Topology (E4)

2] The higher the system load, the better the speedup. This can be explained by Equations 14 and 15. Higher system load increases "N" thereby increasing "X" in Equation 12. It is clear from Equation 14 that an increase in "X" would result in higher speedups.

3] The higher the system size, the better the speedup. This can also be explained by Equations 11 and 12. Higher system size increases both "n" and "N" thereby increasing "X" in Equation 12. It is clear from Equation 11 that an increase in "X" would result in higher speedup.

4] High system load and large system size case gives highest speedups.

6] Low system load and small system size case gives lowest speedups.

7] As the number of processors is increased, at first the efficiency curves climb but beyond 4 processors they slowly decline.

Submodel Network Topology - Fork + Join (E5)

This experiment consists of the simulation of a manufacturing system with "Fork and Join" network of submodels. Table 13 is an ANOVA summary table that furnishes the variance analysis of the important factors and their interactions. In this experimentation factors communication protocol, system load, system size, number of processors used for simulation, and their higher order interactions are found to be significant. A major difference between the results of this experiment and the earlier experiments, is that in this experiment communication protocol and its interaction with the number of processors are also found to be statistically significant. In experiment E5 at higher values of number of processors (8, 16, 32) each processor has multiple input and output channels. Under this situation "forward+backward" protocol unnecessarily creates null messages that increase the communication load on the processors, thereby increasing the simulation execution time. Although "forward" protocol involves waiting for the incoming messages, this protocol gives better performance as the communication between the processors is balanced and frequent.

TABLE 13

FORK+JOIN TOPOLOGY (E5) - ANOVA SUMMARY

Factor	df	OSL	$\alpha = 0.01$
Commu. Protocol (C)	1	0.0001	Reject
System Load (L)	1	0.0001	Reject
System Size (S)	1	< 0.0001	Reject
# of Processors (p)	5	< 0.0001	Reject
(C X p)	5	0.0001	Reject
(L X p)	5	0.0001	Reject
(L X p)	5	0.0001	Reject
(S X p)	5	< 0.0001	Reject
(L X S X p)	5	0.0001	Reject

TABLE 14

FORK+JOIN TOPOLOGY (E5) FOR FORWARD COMMU. PROTOCOL (C=0)
TABLE OF MEANS

			Exe. Time	Speed Up	Efficiency				Exe. Time	Speed Up	Efficiency
L	S	P	Mean	Mean	Mean	L	S	P	Mean	Mean	Mean
0	0	1	193723.7	1.00	100.00%	1	0	1	245363	1.00	100.00%
0	0	2	93354	2.08	103.76%	1	0	2	112156	2.19	109.38%
0	0	4	60938.33	3.18	79.48%	1	0	4	69399.67	3.54	88.39%
0	0	8	36815.67	5.26	65.77%	1	0	8	41912.33	5.85	73.18%
0	0	16	28437	6.81	42.58%	1	0	16	31518	7.78	48.66%
0	0	32	27299.67	7.10	22.18%	1	0	32	30602.67	8.02	25.06%
0	1	1	2011063	1.00	100.00%	1	1	1	5261728	1.00	100.00%
0	1	2	809519	2.48	124.21%	1	1	2	1750748	3.01	150.27%
0	1	4	373600.3	5.38	134.57%	1	1	4	761613	6.91	172.72%
0	1	8	204211.3	9.85	123.10%	1	1	8	328556	16.01	200.18%
0	1	16	146683.3	13.71	85.69%	1	1	16	184715.3	28.49	178.04%
0	1	32	134101.3	15.00	46.86%	1	1	32	158424.7	33.21	103.79%

TABLE 15

F+J TOPOLOGY (E5) FOR DEMAND DRIVEN COMMU. PROTOCOL (C=1)
TABLE OF MEANS

			Exe. Time	Speed Up	Efficiency				Exe. Time	Speed Up	Efficiency
L	S	P	Mean	Mean	Mean	L	S	P	Mean	Mean	Mean
0	0	1	194938.3	1.00	100.00%	1	0	1	246259.3	1.00	100.00%
0	0	2	93924.67	2.08	103.77%	1	0	2	113443.3	2.17	108.54%
0	0	4	147321.7	1.32	33.08%	1	0	4	151990.3	1.62	40.51%
0	0	8	69844	2.79	34.89%	1	0	8	72341.67	3.40	42.55%
0	0	16	28736.67	6.78	42.40%	1	0	16	31778.33	7.75	48.43%
0	0	32	56310.33	3.46	10.82%	1	0	32	43296.33	5.69	17.77%
0	1	1	2005755	1.00	100.00%	1	1	1	5281959	1.00	100.00%
0	1	2	809628.3	2.48	123.87%	1	1	2	1766641	2.99	149.49%
0	1	4	461254.3	4.35	108.71%	1	1	4	846732	6.24	155.95%
0	1	8	202595.3	9.90	123.75%	1	1	8	354349.3	14.91	186.33%
0	1	16	147065.7	13.64	85.24%	1	1	16	185463	28.48	178.00%
0	1	32	134343.3	14.93	46.66%	1	1	32	159262.7	33.17	103.64%

As factor communication protocol is significant, in the "Table of Means" for experiment E5, the execution time means are not averaged over all (two) levels of the communication protocol. This results in Table 14 and 15 as the "Table of Means", each with 24 means out of 144 observations.

Tables 13 and 14 depict the speedup and efficiency values for "forward" and "forward+backward" communication protocol at each combination of the other three significant factors. Equations 14, 15, and 16 are used for explaining the behavior of speedup and efficiency curves. As the interprocessor frequency is relatively high, total waiting time "Tw" for communication is minimal but "forward+backward" protocol produces excessive null messages thereby increasing "C', and eventually the value of "Y". Thus, for "Fork+Join" topology the term "Y*p" dominates the rest of the terms in the denominator. This gives consistently lower speedups for "forward+backward" protocol. The interaction between the processors can be further explained by the product "Y*p" term which determines the execution time for multiprocessor implementation. The effects of changing the number of processors on the speedup and efficiency values at each combination of system load and size are depicted by Figures 37 and 38 for the "forward" protocol and Figures 39 and 40 for the "forward+backward" protocol. Inspection of these figures yield the following observations.

1] The speedup increases as the number of processors is increased. This is explained by observing Equation 14. The "p" in the numerator makes speedup increase as the number of processors is increased. For "forward+backward" protocol the two curves with factor manufacturing system size at its lower level, the speedup decreases when the number of processors is increased from 2 to 4. This can be explained as follows. At two processors the topology of the submodel network is "Tandem", but as the number of processors is increased from 2 to 4 the processor network becomes a real "Fork+Join" topology network. This along with excessive null messages from the "forward+backward" protocol give a dip in the

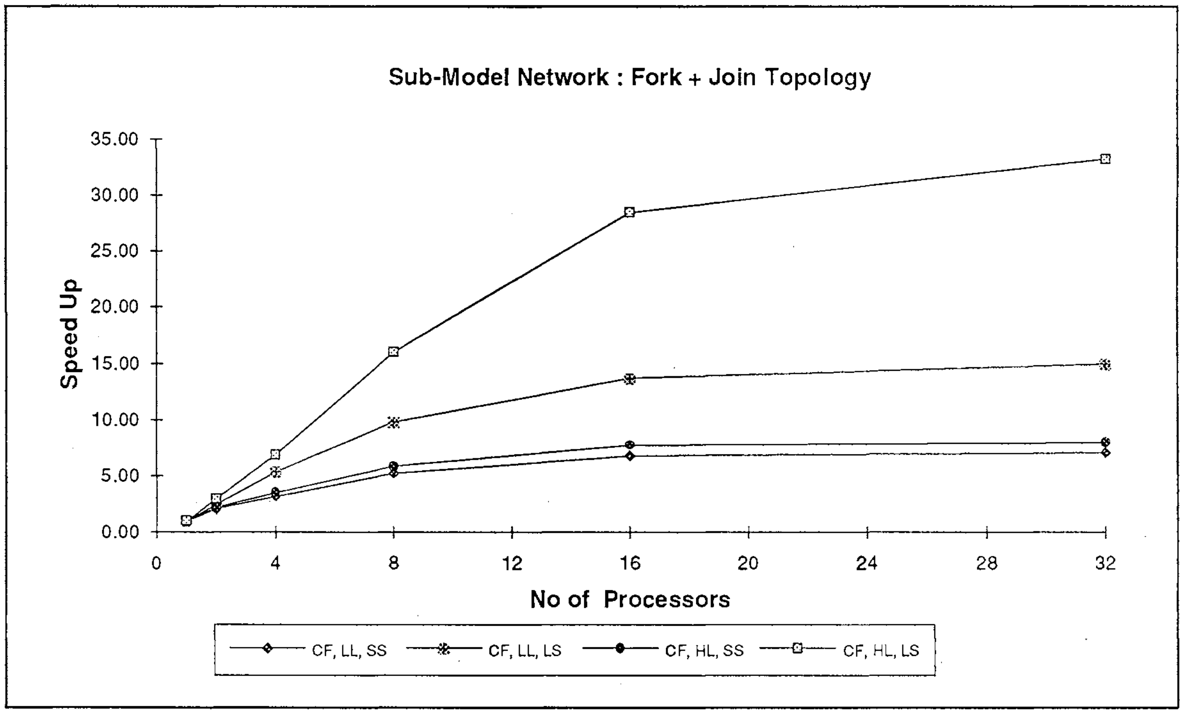

Figure 37. Speedup Curves for "Fork+Join" Topology Using "forward" Protocol (E5)

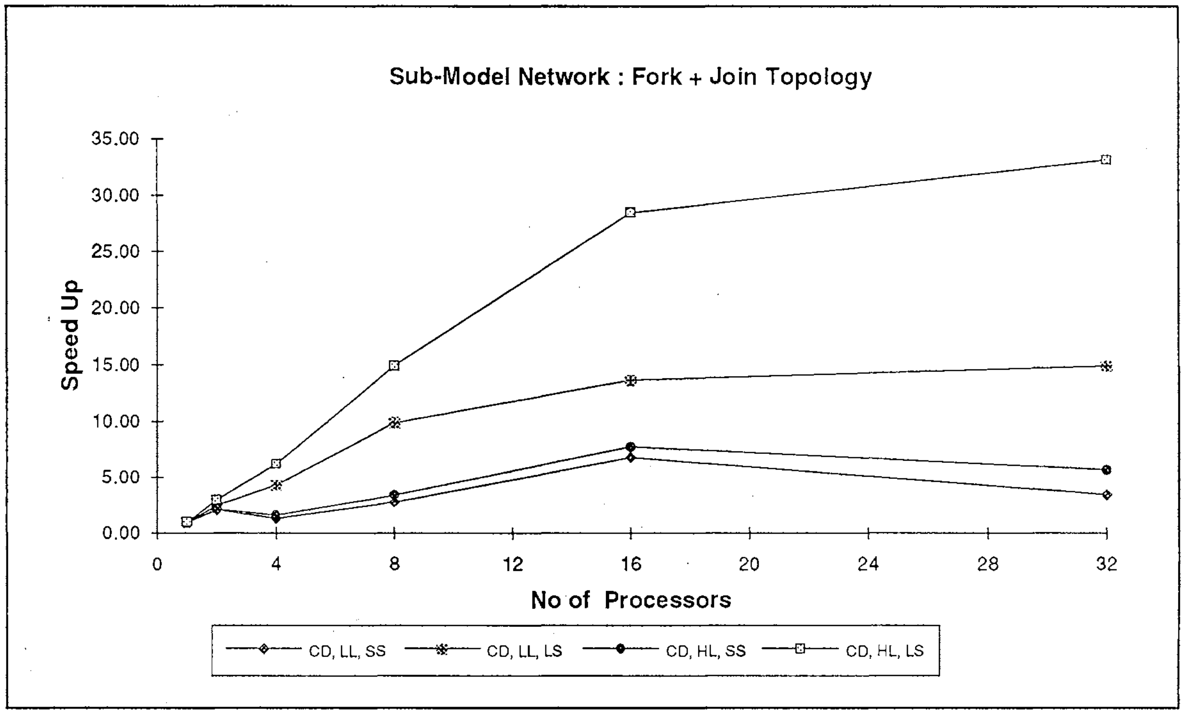

Figure 38. Speedup Curves for "Fork+Join" Topology using "f+b" Protocol (E5)

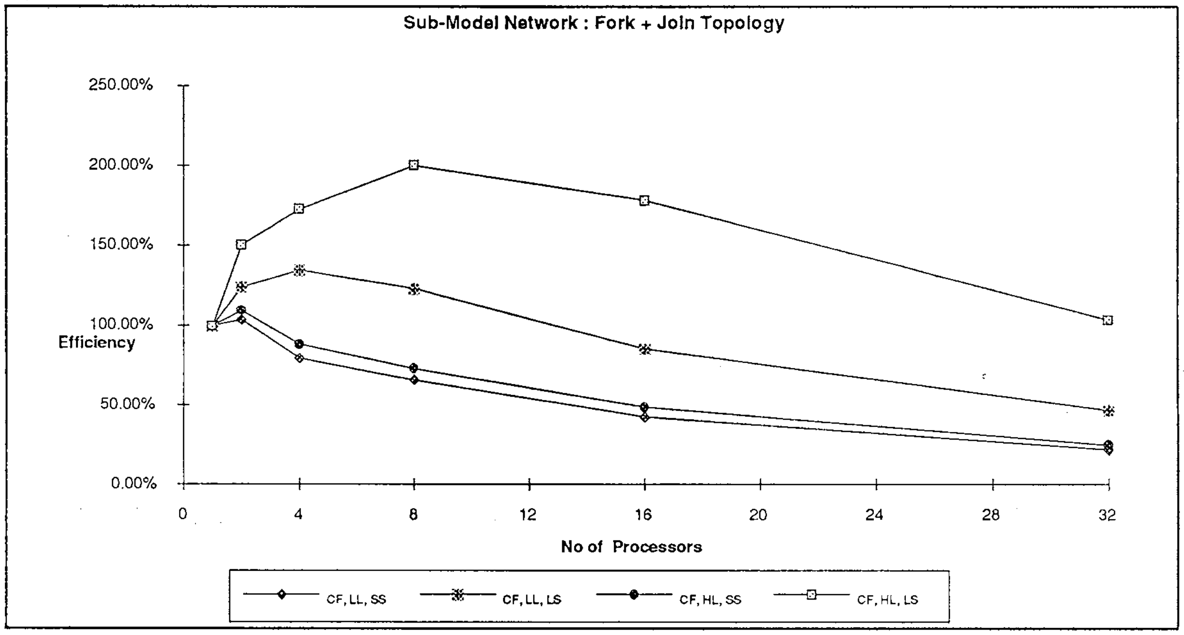

Figure 39. Efficiency Curves for "Fork+Join" Topology Using "forward" Protocol (E5)

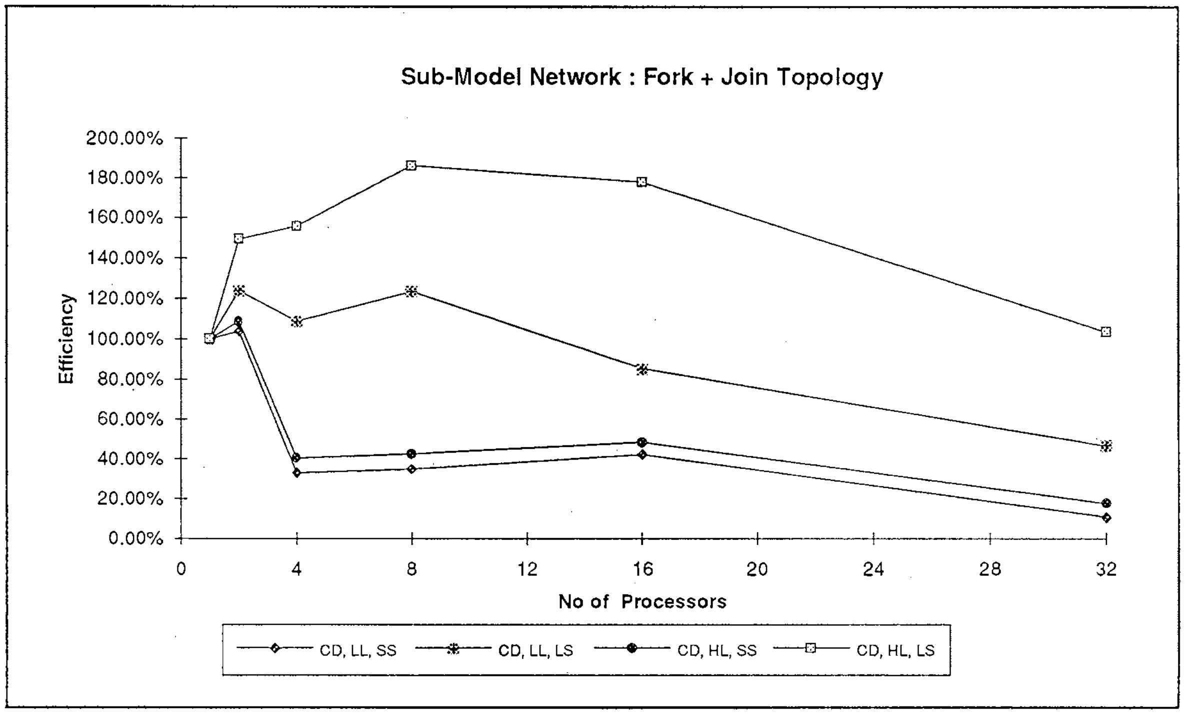

Figure 40. Efficiency Curves for "Fork+Join" Topology Using "f+b" Protocol (E5)

performance at 4 processors. At 32 processors the computation load on each processor is extremely small causing even more frequent null message requests and thereby resulting in performance degradation.

2] The higher the system load, the better the speedup because the curves with high system load are higher than the corresponding low system load curves. This can be explained by Equations 14 and 15. Higher system load increases "N" thereby increasing "X" in Equation 12. It is clear from Equation 14 that an increase in "X" would result in higher speedups.

3] The higher the system size, the better the speedup because the curves with high system size are higher than the corresponding low system size curves. This can also be explained by Equations 11 and 12. Higher system size increases both "n" and "N" thereby increasing "X" in Equation 12. It is clear from Equation 11 that an increase in "X" would result in higher speedup.

4] High system load and large system size case gives highest speedup.

5] Low system load and small system size case gives lowest speedup.

6] The effect of a change in system load at large system size is higher than that at small system size. This explains the statistically significant interaction of the two factors, system size and system load.

7] As the number of processors is increased, at first the efficiency curves climb but for large size beyond 8 processors and for small size beyond 2 processors they slowly decline.

Comparison of Communication Protocols (E6)

This experiment consists of the simulation of a manufacturing system with "Tandem" network of submodels. Table 16 is an ANOVA summary table that furnishes the variance analysis of the important factors and their interactions. In this experimentation, factors communication protocol, system load, system size, number of processors used for simulation, and their higher order interactions are found to be significant. A major difference between the results of this experiment and the earlier tandem experiment, is that in this experiment communication protocol and its two and three way interactions with "P" and system load are also found to be statistically significant. In experiment E6 the frequency of message passing between processors is designed to be low. Unlike the "forward" protocol, the "forward+backward" protocol has mechanisms for creating additional null messages (synchronization demand messages) that can synchronize each processor thereby reducing the waiting for incoming messages. Once the synchronization is received the processor is free to continue simulation execution until the simulation time reaches the channel time acquired from the predecessors. The explanation for the significance of other factors and their interactions is identical to the respective explanations for the earlier "Tandem" experiment, E2. In experiment E6 at higher values of number of processors (8, 16, 32) and small manufacturing system size the computational load per processor is very low, therefore even null message synchronization of "forward+backward" protocol gives only a marginal improvement. On the contrary, at large size of manufacturing system the computational load per processor is relatively high leading to a superior performance by "forward+backward" protocol.

As the factor communication protocol is significant, the execution times in "Table of Means" for experiment E6 are not averaged over all (two) levels of the communication protocol. This results in Tables 17 and 18 as the "Table of Means", each with 24 means out of 144 observations. Table 17 depicts the speedup and efficiency values for forward communication protocol at each combination of the other three significant factors. Equations 14, 15, and 16 are used for explaining the behavior of speedup and efficiency curves. As the interprocessor frequency is extremely low, the total waiting time for communication "Tw" is high but "forward+backward" protocol produces null messages to reduce "Tw", thereby reducing the value of "Y". Therefore "forward+backward" protocol gives consistently higher speedups. At small system size this effect is offset by the fact that for eight or more processors the computational load per processor is very small and reduction in "Tw" is accompanied by the increase in the number of synchronization messages "C" thereby nullifying the increase in the value of "Y".

TABLE 16

TANDEM TOPOLOGY (E6) - ANOVA SUMMARY

Factor	df	OSL	$\alpha = 0.01$
Commu. Protocol (C)	1	0.0001	Reject
System Load (L)	1	0.005	Reject
System Size (S)	1	< 0.0001	Reject
# of Processors (p)	5	< 0.0001	Reject
(C X S)	1	0.0001	Reject
(C X p)	5	0.0002	Reject
(L X p)	5	0.0001	Reject
(S X p)	5	< 0.0001	Reject
(C X S X p)	5	0.0084	Reject
(L X S X p)	5	0.0003	Reject

The effects of changing the number of processors on the speedup and efficiency values at each combination of system load and size are explained by Figures 41 and 42 for the "forward" protocol and Figures 43 and 44 for the "forward+backward" protocol. Inspection of these figures yield the following observations.

1] The speedup increases as the number of processors is increased. This is explained by observing Equation 14. The "p" in the numerator makes speedup increase as the number of processors are increased.

TABLE 17

TANDEM TOPOLOGY (E6) FOR FORWARD COMMU. PROTOCOL (C=0)
TABLE OF MEANS

			Exe. Time	Speed Up	Efficiency				Exe. Time	Speed Up	Efficiency
L	S	P	Mean	Mean	Mean	L	S	P	Mean	Mean	Mean
0	0	1	87058	1.00	100.00%	1	0	1	98753	1.00	100.00%
0	0	2	64766.67	1.34	67.21%	1	0	2	70061.33	1.41	70.48%
0	0	4	54604.33	1.59	39.86%	1	0	4	57354.33	1.72	43.05%
0	0	8	49969.67	1.74	21.78%	1	0	8	51620.33	1.91	23.91%
0	0	16	48221	1.81	11.28%	1	0	16	49583	1.99	12.45%
0	0	32	47802.67	1.82	5.69%	1	0	32	48606.67	2.03	6.35%
0	1	1	592112	1.00	100.00%	1	1	1	804181.7	1.00	100.00%
0	1	2	410277	1.44	72.16%	1	1	2	468147	1.72	85.89%
0	1	4	333667.7	1.77	44.36%	1	1	4	332026.7	2.42	60.55%
0	1	8	294681.3	2.01	25.12%	1	1	8	270306	2.98	37.19%
0	1	16	278209.3	2.13	13.30%	1	1	16	243706	3.30	20.62%
0	1	32	270490	2.19	6.84%	1	1	32	231189.7	3.48	10.87%

TABLE 18

TANDEM TOPOLOGY (E6) FOR DEMAND DRIVEN COMMU. PROTOCOL (C=1)
TABLE OF MEANS

			Exe. Time	Speed Up	Efficiency				Exe. Time	Speed Up	Efficiency
L	S	P	Mean	Mean	Mean	L	S	P	Mean	Mean	Mean
0	0	1	87497	1.00	100.00%	1	0	1	99598.33	1.00	100.00%
0	0	2	46179.67	1.89	94.74%	1	0	2	50341.67	1.98	98.92%
0	0	4	29353.33	2.98	74.52%	1	0	4	30799.67	3.23	80.84%
0	0	8	22987	3.81	47.58%	1	0	8	23478.33	4.24	53.03%
0	0	16	23699.67	3.69	23.07%	1	0	16	24814.33	4.01	25.09%
0	0	32	30403.33	2.88	8.99%	1	0	32	35031.67	2.84	8.88%
0	1	1	597482.7	1.00	100.00%	1	1	1	810984	1.00	100.00%
0	1	2	266302.3	2.24	112.18%	1	1	2	329383.7	2.46	123.11%
0	1	4	139866	4.27	106.80%	1	1	4	160353.7	5.06	126.44%
0	1	8	86272.33	6.93	86.57%	1	1	8	94832	8.55	106.90%
0	1	16	65832	9.08	56.72%	1	1	16	69412.33	11.68	73.02%
0	1	32	60654.67	9.85	30.78%	1	1	32	64449.33	12.58	39.32%

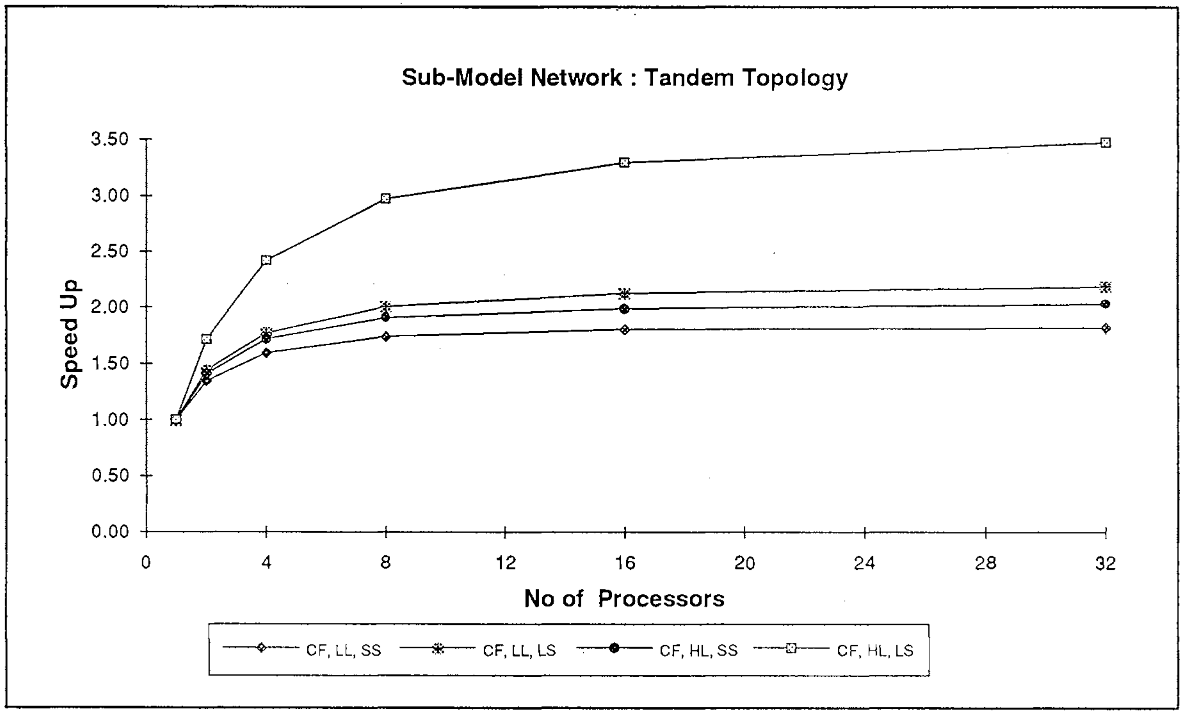

Figure 41. Speedup Curves for "Tandem" Topology Using "forward" Protocol (E6)

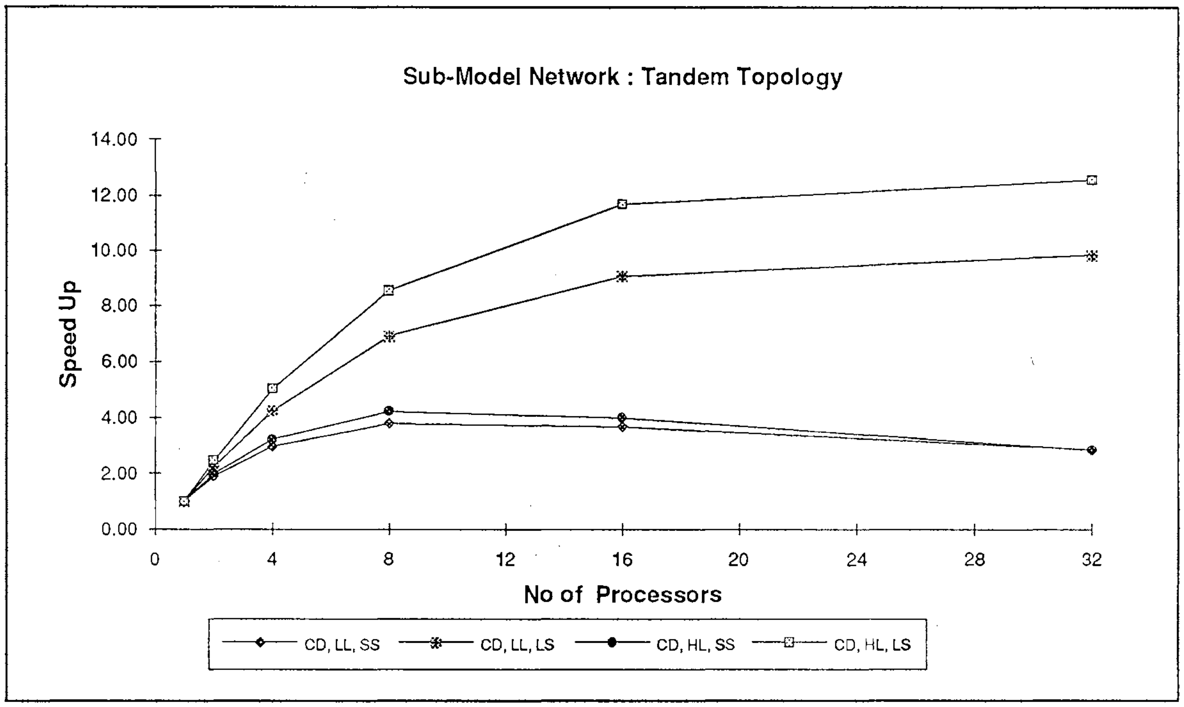

Figure 42. Speedup Curves for "Tandem" Topology Using "f+b" Protocol (E6)

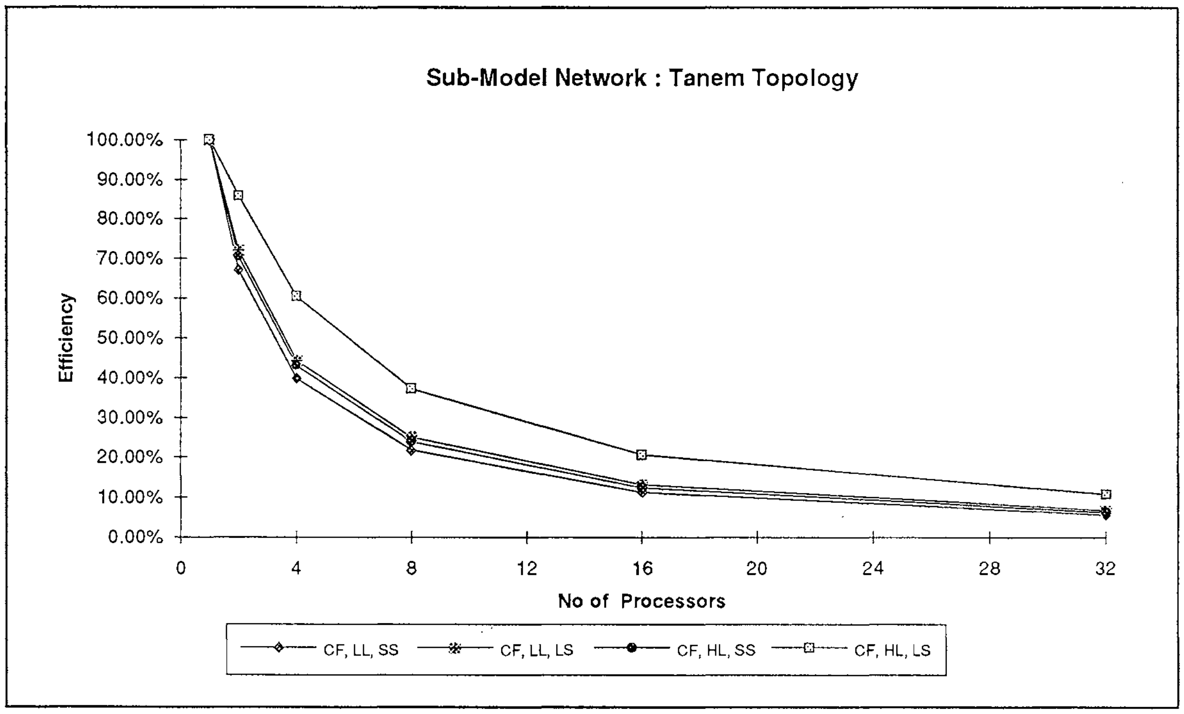

Figure 43. Efficiency Curves for "Tandem" Topology Using "forward" Protocol (E6)

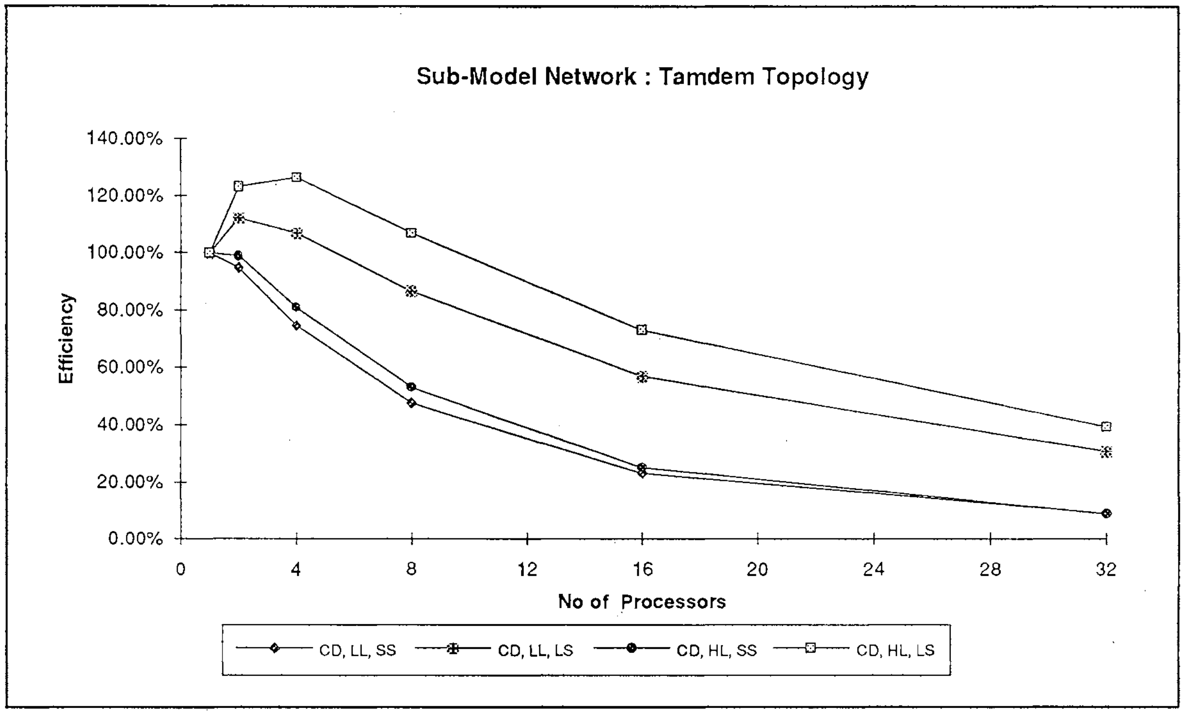

Figure 44. Efficiency Curves for "Tandem" Topology Using "f+b" Protocol (E6)

2] The higher the system load, the better the speedup because the curves with high system load are higher than the corresponding low system load curves. The explanation for this is identical to that of the earlier "Tandem" experiment (E2).

3] The higher the system size, the better the speedup because the curves with high system size are higher than the corresponding low system size curves. The explanation for this is identical to that of the earlier "Tandem" experiment (E2).

4] High system load and large system size case gives highest speedup.

5] Low system load and small system size case gives lowest speedup.

6] The effect of a change in system load at large system size is higher than that at small system size. This explains the statistically significant interaction of the two factors, system size and system load.

7] For "forward+backward" protocol, as the number of processors is increased, at first the efficiency curves climb but for large size beyond 4 processors and for small size beyond 2 processors they slowly decline. On the contrary, all efficiency curves for "forward" strategy result in a monotonically decreasing curve.

Summary of Experimentation Results

Detailed observations of each experiment has been provided in the above six sections. This section attempts to summarize the entire research experimentation involving the above experiments. It not only identifies a common behavior among the topologies, but also provides conclusions about the relative performance of the experimental submodel network topologies. The commonalties in execution performance of several topologies are provided in the next paragraph. The paragraph following it describes the differences in the execution performance.

Among all topologies, the speedup generally improved as the number of processors is increased. However, the efficiency figures initially improve (going beyond

100%) and then constantly decline. The encouraging fact is that for large systems over a significant range for number of processors, superlinear speedups (efficiency > 100%) have been obtained. Execution of the event calendar being a searching type application, the execution time is proportional to (N) the average event calendar length. In these parallel implementations the superlinear speedups are observed because of two reasons, viz. the new discrete event simulation algorithm which substantially reduces the original event calendar length, and the availability of a number of processors. The new discrete event simulation algorithm saves the extra searches through the event calendar, thereby providing a major reduction in the computation. In other words, by using the new simulation algorithm a system can be simulated on a single processor as a collection of processes (each corresponding to a single submodel) and a sublinear speedup can be achieved. The second sublinear component of speedup is a result of using more than one processor to simulate these processes. As in the case of the current experimentation, when both the sublinear speedup components are combined they give a net superlinear speedup as a product of two sublinear speedup components. Both of these sublinear speedup components depend very heavily on the synchronization requirements between all the system submodels. And therefore, as the processor assignment approaches one machine process on each processor (or one machine per submodel) the performance improvement is very marginal. Further, as each process essentially is a single thread of computation, below this ratio, (i.e., the ratio of logical processes to the number of processors, ratio < 1) it is almost impractical to further distribute the computation as logical dependencies between variables require extensive synchronization. This makes the above scheme almost impractical. A ratio of 8 (where most of the efficiency curves for small system peaked) is typically favored. The sixth experiment establishes the necessity of "forward+backward" or "demand driven" protocol. As the frequency of the communication is also dependent on the nature of the manufacturing system in terms of its routing structure, for a general system frequent interprocessor synchronization is not guaranteed. Therefore, despite either insignificant improvement or even slight decline in performance when used with frequent and balanced synchronization requirements, "forward+backward" communication is preferred for a general case.

One of the major aspects of the comparative analysis of five topologies is the inherent communication requirements of each topology. "Independent Clusters" topology requires no interprocessor communication and produces the best results. In "Tandem" topology, as a processor requires communication only from its single predecessor, it is only affected by the frequency of communication. It therefore comes second in performance. "Fork" is similar to "Tandem" in the sense that the input communication still comes from a single source, but this source has to provide communication to a number of processors and therefore the individual frequency of communication between the source and its predecessor gets several fold smaller than the "Tandem" topology. This is reflected in the loss of speedup (from 56.56 to 22.81). Besides frequent synchronization, the "Join" topology also requires balanced communication from the various incoming channels and the problem only gets worse for a large number of incoming channels. "Fork+Join" topology is a very general topology. Because of its structural features, it has much larger computational requirements than the other manufacturing system models. Therefore a true comparison between this topology and others is not attempted. The main motivation in experimenting with "Fork+Join" topology was to see if this type of system poses any additional problems in implementation and analysis. Under the experimental conditions "Fork+Join" gives almost linear speedups with large size system. In general "Fork+Join" topology should be expected to perform somewhere in between its "Fork" and "Join" components.

In conclusion, the research experimentation results and a thorough analysis of the research findings are provided in this chapter. The next chapter provides general guidelines for accomplishing an efficient PDES, and outlines future research directions.

CHAPTER XI

SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

Introduction

This chapter presents concluding thoughts about this research effort. It includes a summary of the research in light of the proposed objectives, contributions to the body of knowledge in parallel discrete event simulation, and recommendations for additional research or new research directions.

Research Summary

The goal of this research was to analyze the factors that significantly influence an effective parallel implementation of the simulation of object oriented models of manufacturing systems. To accomplish this goal five research objectives, each addressing a different aspect of the problem, were established. The sections below review the accomplishments in each of these objectives.

Concurrent Object Oriented Modeling

The first research objective was the evaluation of concurrent object oriented modeling constructs for parallel discrete event simulation. The entirety of Chapter VII dealt with this objective. In this chapter it was established that the concurrent object oriented programming constructs provide very powerful means for creating concurrently executable instructions. Being object oriented, they create encapsulated objects that have the properties of distributed systems. These properties provide the logical separation of two objects for their concurrent evaluation. Concurrent object oriented programming constructs provide an automatic synchronization of the processes without using semaphores or other types of synchronization procedures. The user is relieved from the mundane task of coping with the synchronization among processors. However, it is important to note that the user must have a very clear understanding of "which construct to use?" and "where?". That is the user must be aware of the types of inherent concurrencies in the programming application such as simulation and must appropriately use the available concurrent construct.

The above summary has accomplished the objective of the evaluation of concurrent object oriented programming constructs for its applicability to the parallel discrete event simulation.

Submodel Creation Logic

The second objective was the formulation of the submodel creation logic. Chapter VIII is solely devoted to accomplishing this objective. In the context of this research, several guiding principles were used for specifying the submodel creation logic and the optimization function. Firstly, an efficient parallel processing application minimizes the interprocessor communication. Secondly, in case of dependent submodels, previous research [Reed 1988] has shown that even though there are several mechanisms currently available for deadlock detection and recovery, almost all of them provide only marginal speedups. And hence for the purpose of submodel creation, topologies that have a potential for 'deadlock' are avoided. Thirdly, it is also important to note that the speedup can be improved by using more and more processors. Based on these three principles the procedure for developing submodel creation logic or allocation of the machine processes to the processors was derived.

Using the procedure described above, the objective of formulating the submodel creation logic was accomplished.

Communication Strategy Design

The third objective was the design of a communication strategy for the interprocessor communication between Intel iPSC/2 hypercube nodes. Chapter IX is solely devoted to accomplishing this objective. As the submodel creation strategies carefully produce submodels without any 'deadlock' potential, the communication protocols designed for the purpose of this research are not required to handle the 'deadlock' situations. This not only simplifies the design process, but also provides highly efficient interprocessor communication. For the purpose of implementation, two distinct communication protocols were designed, viz. forward and forward+backward. The key for developing the communication protocols is to develop the communication requirements for parallel execution of discrete event simulation. For complex dependencies among the events on different processors (such as blocking machines where the parts arrive to the blocked machine from another processor, or the communication patterns with a potential for "deadlock", etc.) a suitable protocol must be tailored for the required use. By abstracting the event dependencies, a researcher can develop generalized concurrent object oriented programming constructs that can create a general communication protocol.

Using the procedure described above, the objective of designing a communication protocol for the parallel implementation was accomplished.

Performance Analysis via PDES Implementation

The fourth objective was the analysis of the performance of parallel discrete event simulation application. Chapter X is solely devoted to accomplishing this objective. The experimental design described in Chapter V specified a total of six simulation experiments. The first five experiments were designed to simulate five submodel network topologies of the submodel network. The sixth experiment was specifically designed to reveal the differences between the two communication protocols that are not evident in earlier experiments. In each of the above six experiments there were two levels of "Manufacturing System Size", "Communication Protocol", "Manufacturing System Load", and six levels of the "Number of Processors" factor. Three simulation experiment replications were performed for each combination of factors. An approximate mathematical model of the execution process was also developed. This model was used to explain the behavior of the performance of the simulation applications over several combinations of the experimental factors. Specific conclusions of these experiments are provided in Chapter X. These conclusions also helped in developing the methodology for an efficient parallel discrete event simulation.

Using the procedure described above, the objective of analyzing the performance of parallel discrete event simulation application was accomplished.

Methodology for Achieving a Successful PDES

The fifth objective was the creation of a methodology for an efficient parallel discrete event simulation of a manufacturing system. This methodology is supplied in the next paragraph.

A typical manufacturing system simulation effort requires the evaluation of multiple scenarios or control policies and the selection of the best scenario or control policy among the prespecified set. For obtaining sound statistical confidence in the simulation results each scenario is further replicated several times. That is, for "S" scenarios and "r" replications there are a total of "S*r" simulations required for identifying the best control policy among the available set. If the experimental design of such a study consists of "F" factors each with "L" levels, then there would be a total of "F*L" possible scenarios and "F*L*r" simulations.

One of the simplest ways of achieving faster evaluation of the total set of experimentation scenarios is to create a batch file or host program that distributes these

"F*L*r" simulations on "F*L*r" nodes or processors of the parallel processing computer; each processor executing a single simulation. This arrangement takes advantage of very coarse grain concurrency in the parallel simulation application and quickly results in a speedup of "F*L*r". In this arrangement the internal complexities of event dependencies do not affect the performance of parallel simulation application because each simulation execution is still a uniprocessor application.

To take advantage of finer grain concurrency in the simulation application, execution of each simulation is further distributed on multiple processors by dividing the simulation model into submodels and executing each submodel on a single processor. And since execution of each model is further distributed this arrangement further improves speedups. If by using "p" processors we can make a single simulation run "S" times faster, a net speedup of "F*L*r*S" can be achieved by using a total of "F*L*r*p" processors. However, it is important to note that maintenance of the "causality" constraints of each submodel add massive communication overhead which depends on the complexity of the event dependencies among the submodels and therefore the resultant speedup depends on dependency or topology of the interprocessor communication patterns. These patterns are highly influenced by the submodel creation process. Chapter VIII provides a submodel creating methodology that can help the user in properly allocating the machine processes to the submodels. Complex dependencies of the events between submodels requires highly sophisticated communication protocols that can add excessive communication. The "forward" and "forward+backward" communication protocols provided in Chapter IX can be used as a model for developing more complex protocols. Specifically the user should avoid the communications having a potential of "deadlock" or feedback communication patterns.

Typical manufacturing systems such as cellular, tandem lines, flow lines, flexible manufacturing systems, job shop systems, etc., can be effectively simulated by using their inherent routing topology. For a cellular manufacturing system, each cell can be considered as a submodel and therefore can result in a disjoint submodel network of independent clusters of machines. As explained in Chapter X, this arrangement can result in superlinear speedup. A "tandem" line can be simulated by using a "tandem" network of submodels, a flow line can be crafted as a "Tandem" submodel, or "Fork" or Join" or a combination of the two topologies. The key here is to create as many submodels as possible until the interprocessor communication between submodels becomes overwhelming.

Research Contributions

As explained in the first chapter, widespread use of discrete event simulation as an analysis tool is hindered because of enormous computational requirements of a simulation effort. This research therefore focused on achieving faster execution of simulation models by using multiple processors. There are very few published results available in this area. One of the major intended contributions of this research was to create empirical data for parallel discrete event simulation of manufacturing systems. The vehicle used to demonstrate the parallel implementation was an object oriented modeling environment on Intel iPSC/2 hypercube parallel processor.

The completion of the research objectives as documented in the previous section makes the following contributions to the area of advanced simulation modeling of manufacturing systems within Industrial Engineering:

1] Determination of the factors that have the potential to influence the execution performance of a parallel discrete event simulation of manufacturing systems.

2] Demonstration of the viability of parallel implementation of discrete event simulation of manufacturing systems via a modified event scheduling technique.

3] Specification of a submodel creation methodology.

4] Development of a mathematical function for analyzing the parallel implementation of discrete event simulation of manufacturing systems.

5] Development of a design methodology for interprocessor communication protocols.

6] Development of a comprehensive methodology achieving an efficient implementation of a parallel discrete event simulation of manufacturing systems.

7] Provision of empirical data for further research in the areas of parallel discrete event simulation of discrete event systems.

Recommendations for Future Research

As a result of the research conducted in this study, the following recommendations are made for additional research in this area.

Parallel Processing Architectures

The findings for this research are valid for parallel implementations on distributed memory message passing architectures such as the Intel iPSC/2 hypercube. Similar methodologies for an efficient implementation on other parallel architectures can be developed.

Concurrent Object Oriented Programming Constructs

While accomplishing the first objective, this research realized the importance of concurrent object oriented constructs. Currently these constructs cannot be readily used as an integral part of simulation objects. Complex dependencies of simulation objects can be abstracted to create abstract concurrent discrete event simulation objects which have synchronization mechanisms built in. Then the user can create the manufacturing system objects as subclasses of these abstract concurrent objects. These concurrent objects would be similar to the "Actor" [Agha 88] objects described in Chapter II.

Submodel Creation Logic

While accomplishing the second objective, a submodel creation methodology and the objective function for selecting the optimal process allocation arrangement was developed. This methodology is designed for simplistic interprocessor communication requirements. A detailed objective function can be developed for more complex communication patterns created by the complex event dependencies. This methodology can also be formulated as a linear programming optimization problem that can be solved to obtain the best submodel designs or process allocation arrangement.

Design of Communication Protocols

Researchers in the computer science area have taken some major strides in this problem specific domain. Most researchers devote their attention to developing algorithms that can withstand "deadlock" situations. More research is needed in understanding the event dependencies involved in typical manufacturing operations such as hierarchical control, material handling, blocking and balking, machine breakdowns and repair, etc.

BIBLIOGRAPHY

Adiga, S. (1989), "Software Modeling of Manufacturing Systems: A Case for an Object Oriented Programming Approach," *Analysis, Modeling, and Design of Modern Production Systems*, A. Kusiak and W.E. Wilhelm, Eds., J.C. Baltzer A.G., Basel, Switzerland.

Agha, G. and C. Hewitt (1988a), "Actors: A Conceptual Foundation for Concurrent Object Oriented Programming," *Research Directions in Object Oriented Programming*, MIT Press, Cambridge, MA.

Agha, G. and C. Hewitt (1988b), "Concurrent Programming Using Actors," *Object Oriented Concurrent Programming*, MIT Press, Cambridge, MA.

America, P. (1988), "POOL-T: A Parallel Object Oriented Language," *Object Oriented Concurrent Programming*, MIT Press, Cambridge, MA.

Aki, S. (1989), *The Design and Analysis of Parallel Algorithms,* Prentice-Hall, Inc., Englewood Cliffs, NJ.

Bain, L.W. and D.S. Scott (1988), "An Algorithm for Time Synchronization in Distributed Discrete Event Simulation," *Proceedings of the SCS Multiconference on Distributed Simulation,* B. Unger and D. Jefferson Eds., SCS, San Diego, CA, 30-33.

Basnet, C., P. Farrington, D. Pratt, M. Kamath, C. Karacal, and T. Beaumariage (1990), "Experiences in Developing an Object-Oriented Modeling Environment for Manufacturing Systems," *1990 Winter Simulation Conference Proceedings,* O. Balci, R. Sadowski, and R. Nance, Eds., IEEE, Piscataway, NJ, 477-481.

Beaumariage T. (1990), "Investigation of an Object Oriented Modeling Environment for the Generation of Simulation Models," Ph.D. Thesis, Department of Industrial Engineering and Management, Oklahoma State University, Stillwater, OK.

Biles, W. (1985), "Statistical Considerations in Simulation on a Network of MicroComputers," *Winter Simulation Conference Proceedings,* IEEE, Piscataway, NJ, 388-393.

Birtwistle, G.M. (1979), *Discrete Event Modeling on Simula,* Springer-Verlag New York Inc., New York, NY.

Brock, J.D. and W.B. Ackerman (1981), "Scenarios: A Model of Non-determinate Computation," *Formalization of Programming Concepts*, Springer-Verlag, 252-259.

Bustard, D., J.Welder, and J. Welsh (1988), *Concurrent Program Structures*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Chandak, A. and J.C. Browne (1983), "Vectorization of Discrete Event Simulation," In*Proceedings of the 1983 International Conference on Parallel Processing,* 359-361.

Chandrasekaran, U. and S. Sheppard (1987), "Discrete Event Distributed Simulation - A Survey," In *Proceedings of the SCS Multiconference on Methodology and Validation,* O. Balci Eds., SCS, San Diego, CA, *19,* 1, 32-37.

Chandy, K.M. and J. Misra (1979), "Distributed Simulation : A Case Study of Design and Simulation of Distributed Programs," *IEEE Transactions on Software Engineering, SE-5,* 5, 440-452.

Chandy, K.M. and J. Misra (1981), "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM, 24,* 4, 198-206.

Chandy, K.M. and R. Sherman (1989a), "Space Time and Simulation," *Proceedings of the SCS Multiconference on Distributed Simulation,* B. Unger and R. Fujimoto Eds., SCS, San Diego, CA, *21,* 2, 53-57.

Chandy, K.M. and R. Sherman (1989b), "The Conditional Event Approach to Distributed Simulation," *Proceedings of the SCS Multiconference on Distributed Simulation,* B. Unger and R. Fujimoto Eds., SCS, San Diego, CA, 21, 2, 93-99.

Chen L. and C. Chen (1990), "A Fast Simulation Approach for Tandem Queueing Systems," *1990 Winter Simulation Conference Proceeding,* O. Balci. R.P. Sadowaski, and R.E. Nance Eds., WSC, IEEE, Piscataway, NJ, 539-546.

Comfort, J.C. (1984), "The Simulation of a Master Slave Event Set Processor," *Simulation, 42,* 3, 117-124.

Davis IV, N.J., D. Mannix, W.H. Shaw, and T.C. Hartrum (1990), "Distributed Discrete-Event Simulation Using Null Message Algorithm on Hypercube Architectures," *Journal of Parallel and Distributed Computing, 8*, 349-357.

Derrick J.E., O. Balci, and R.E. Nance (1989), "A Comparison of Selected Conceptual Frameworks for Simulation Modeling," *1989 Winter Simulation Conference Proceeding,* E.A. MacNair, K.J. Musselman, and P. Heidelberger Eds., WSC, IEEE, Piscataway, NJ, 711-718.

Fujimoto, R.M. (1989), "Performance Measurements of Distributed Simulation Strategies," *Transactions of The Society of Computer Simulation*, O.A. Palusinski and P. Luker Eds., SCS, San Diego, CA, *6*, 2, 89-132.

Goldberg, A., and D. Robson (1989), *Smalltalk 80: The Language,* Addison-Wesley, Reading, M.A.

Hewitt, C.E. and H. Backer (1977), "Laws for Communicating Parallel Processes," *Proceedings of 1977 IFIP Congress,* 987-992.

Hoare, C.A.R. (1978), "Communicating Sequencial Processes," *Communications of the ACM, 21,* 8, 666-677.

Ishikawa, Y. and Tokoro M. (1988), "Orient84/k: An Object Oriented Concurrent Programming Language for Knowledge Representation," *Object Oriented Concurrent Programming*, MIT Press, Cambridge, MA.

Jones, D.W. (1986), "An Empirical Comparison of Priority-Queue and Event-Set Implementations," *Communications of the ACM, 29,* 4, 300-311.

Kahn, K. and D. MacQueen (1977), "Coroutine and Networks of Parallel Processes," *Information Processing 77: Proceedings of IFIP*, Academic Press, 993-998.

Law, A.M. (1986), "An Introduction to Simulation: A Powerful Tool for Analyzing Complex Manufacturing Systems," *Industrial Engineering* 18, 5, 46-63.

Lalonde, W.R. and J.Pugh (1991),*Inside Smalltalk: Volume II*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

McCarthy, J. (1959), "Recursive Functions of Symbolic Expressions and their Computation by Machine," *Memo 8*, MIT.

Mitrani, I. (1982), *Simulation Techniques for Discrete Event System,* Cambridge University Press, Cambridge, UK.

Nevison C. (1990), "Parallel Simulation of Manufacturing Systems: Structural Factors," *Proceedings of the SCS Multiconference on Distributed Simulation,* D. Nicol Eds., SCS, San Diego, California, 22, 2, 17-22.

Park, S.K. and K.W. Miller (1988), "Random Number Generators: Good Ones are Hard to Find," *Communications of the ACM, 31,* 10,1192-1201.

Pratt, V.R. (1982), "On the Composition of Processes," *Proceedings of the 9th Annual Conference on Principles of Programming Languages 1982.*

Pratt, D.B., P.A. Farrington, C.B. Basnet, H.C. Bhuskute, M. Kamath, J.H. Mize (1991), "A Framework for Highly Reusable Simulation Modeling: Separating Physical, Information, and Control Elements," In *Proceedings of the 24th Annual Simulation Symposium*, A.H. Rutan (Ed.), IEEE Computer Society Press, 254-261.

Pratt, D.B. (1992), "Development of a Methodology for Hybrid Metamodeling of Hierarchical Manufacturing System Within a Simulation Framework," Ph.D. Thesis, Department of Industrial Engineering and Management, Oklahoma State University,Stillwater, OK.

Pritsker, A.A.B. (1979), "Compilation of Definitions of Simulation,"*SIMULATION*, 33,1,61-63.

Pritsker, A.A.B. (1986), *Introduction to Simulation and SLAM II,* Third Edition, Halsted Press, New York, NY.

Reed, D.A. (1985), "Parallel Discrete Event Simulation: A Case Study," *18th Annual Simulation Symposium,* IEEE Computer Society Press, 95-107.

Reed, D.A. and A. Maloney (1988), "Parallel Discrete Event Simulation: The Chandy Misra Approach,"In *Proceedings of SCS Multiconference on Distributed Simulation,* B. Unger and D. Jefferson Eds.,SCS, San Diego, CA, 8,13.

Reynolds, P.F. (1982), "A Shared Resource Algorithm for Distributed Simulation," *Proceedings of the 9th Annual Symposium on Computer Architecture,* Austin, TX, *9,* 3, 259-266.

Rogers, P. and M.T. Flanagan (1991), "On-Line Simulation For Real-Time Scheduling Of Manufacturing Systems," *Industrial Engineering, 23,* 12, 37-40.

Seethalakshmi, M. (1990), "A Study and Analysis of Performance of Distributed Simulation," M.S. Thesis, University of Texas at Austin, Austin, TX.

Yokote, Y. and M.Tokoro (1988), "Concurrent Programming in ConcurrentSmalltalk," *Object Oriented Concurrent Programming*, MIT Press, Cambridge, MA.

Yonezawa, A. , E. Shibayama, T. Takada, and Y. Honda. (1988), "Modeling and Programming in Object Oriented Concurrent Language ABCL/1," *Object Oriented Concurrent Programming*, MIT Press, Cambridge, MA.

VITA

Hemant C. Bhuskute

Candidate for the Degree of

Doctor of Philosophy

Thesis: APPLICATION OF PARALLEL PROCESSING FOR OBJECT ORIENTED DISCRETE EVENT SIMULATION OF MANUFACTURING SYSTEMS

Major Field: Industrial Engineering and Management

Biographical:

Personal Data: Born in Pune, India, September 29, 1964, the son of Chandrashekhar S. and Shailaja C. Bhuskute.

Education: Graduated from Ramnarayan Ruia Junior College, Bombay in May 1982; received Bachelor of Science Degree in Electrical Engineering from Bombay University in May 1986; received Master of Manufacturing Systems Engineering Degree from Oklahoma State University in July, 1989; completed the requirements for the Doctor of Philosophy degree at Oklahoma State University in May 1993.

Professional Experience: Manufacturing Systems Engineer, Fabricut Inc., from May 1988 to January 1989; Teaching Assistant, School of Industrial Engineering and Management, Oklahoma State University, from August 1989 to May 1990; Research Associate, School of Industrial Engineering and Management, Oklahoma State University, from June 1990 to present.