

**ANALYTIC MODELING METHODOLOGY FOR EVALUATING  
THE PERFORMANCE OF DISTRIBUTED,  
MULTIPLE-COMPUTER SYSTEMS**

**Alexander Kapelnikov**

**November 1987  
CSD-870061**

UNIVERSITY OF CALIFORNIA

Los Angeles

Analytic Modeling Methodology for Evaluating the  
Performance of Distributed, Multiple-Computer Systems

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

by

Alexander Kapelnikov

1986

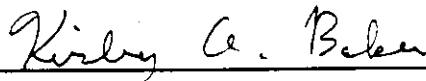
This research was partially sponsored by the Advanced Research Projects Agency, Department of Defense, under Contract No. F29601-87-K-0072, Very Large Distributed Information Processing Systems.

© Copyright by

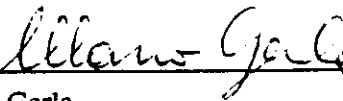
Alexander Kapelnikov

1986

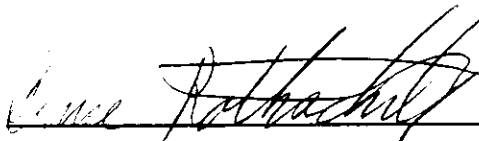
The dissertation of Alexander Kapelnikov is approved.



Kirby Baker



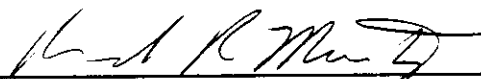
Mario Gerla



Bruce Rothschild



Milos D. Ercegovac, Committee Co-Chair



Richard Muntz, Committee Co-Chair

University of California, Los Angeles

1986

## DEDICATION

This dissertation is dedicated to the fond memory of my grandmother, Dina Krizeman, who laid the foundation for my intellectual development.

## Table of Contents

	page
Abstract .....	xii
<b>1 INTRODUCTION .....</b>	<b>1</b>
1.1 Overview of Distributed Systems .....	2
1.2 Need for Performance Prediction .....	5
1.3 Current Methods .....	6
1.4 Motivation for a New Methodology .....	7
1.5 Organization of Thesis .....	9
<b>2 DISCUSSION OF DISTRIBUTED SYSTEMS .....</b>	<b>10</b>
2.1 Need for Parallel Processing .....	10
2.2 General Principles .....	11
2.3 Architectural Variations .....	13
2.3.1 Task Allocation Policies .....	14
2.3.2 Synchronization Schemes .....	15
2.3.3 Classification of Distributed Systems .....	16
2.3.4 Examples of Different Categories .....	17
2.4 Program Variations .....	18
2.4.1 Task Attributes .....	18
2.4.2 Structural Variations .....	19
2.4.3 Examples of Parallel Algorithms .....	21
2.5 Design Issues .....	23
<b>3 THE MODEL .....</b>	<b>26</b>
3.1 Physical Domain Model .....	26
3.1.1 The P/C Subnetwork .....	27
3.1.2 The M/U Subnetwork .....	29
3.1.3 The Black Box Construct .....	32
3.2 Program Domain Model .....	36
3.2.1 Computation Control Graphs .....	37
3.2.2 Classes of Tasks .....	43
3.2.3 Segmentation: Hierarchical Perspective .....	44
3.2.4 Examples of Computation Control Graphs .....	45
<b>4 APPROXIMATE ANALYTIC SOLUTION .....</b>	<b>50</b>
4.1 Example of a Model .....	51
4.1.1 System Description .....	51
4.1.2 Physical Domain Model .....	54
4.1.3 Program Domain Model .....	56
4.2 Decomposition Approximation .....	58
4.3 Markov Process Construction .....	63
4.3.1 State Space Description .....	64
4.3.2 Generation of a Markov Process from a Computation Control Graph .....	72
4.3.2.1 Markov Process Generation Algorithm .....	72

4.3.2.2	State Space Reduction .....	76
4.3.3	State Transition Rates .....	80
4.4	Solving Markov Processes .....	82
4.4.1	Exact vs. Approximate Techniques .....	82
4.4.2	Acyclic Processes and Symbolic Solution .....	84
5	OPTIMIZATIONS AND HEURISTICS .....	86
5.1	Segments: Compact Description .....	87
5.2	Sequential Combination .....	88
5.3	EXCLUSIVE-OR Combination .....	90
5.4	Parallel Combination .....	92
5.4.1	Computing Mean Execution Time .....	92
5.4.1.1	Definition of Notation .....	93
5.4.1.2	Estimating Average Throughputs of Tasks .....	94
5.4.1.3	Estimations .....	95
5.4.1.4	Computing (Average) First Complete Execution .....	96
5.4.2	Computing the Distribution of Parallelism .....	98
5.4.3	Parallel Combination of Several Segments .....	99
5.4.4	Parallel Combination of Sequential Combinations .....	99
5.5	Looping Constructs .....	103
5.5.1	Simple Loops .....	103
5.5.1.1	Computing Loop Mean Time Execution .....	105
5.5.1.2	Loop Combination Distribution of Parallelism .....	114
5.5.1.2.1	Definition of Notation .....	114
5.5.1.2.2	Derivation of Average Amount of Time in Phase I .....	116
5.5.1.2.3	Derivation of Average Amount of Time in Phase II .....	117
5.5.1.2.4	Average Fractions of Time during Loop L .....	118
5.5.2	Complex Loops .....	118
5.6	Other Heuristics .....	124
5.6.1	Hierarchical Combinations .....	124
5.6.2	Multi-Chain Models .....	125
5.6.3	Process Arrivals .....	126
5.7	Degree of Aggregation vs. Accuracy .....	127
6	CASE STUDY A: .....	129
6.1	Simulation Environment .....	130
6.1.1	Overview of RESQ .....	130
6.1.2	Simulation Models .....	131
6.2	Application Description .....	133
6.2.1	The Algorithm .....	133
6.2.2	Implementation A .....	135
6.2.3	Implementation B .....	137
6.2.4	Implementation C .....	141
6.3	Experimentation with Different Architectures .....	143
6.3.1	Tightly-Coupled System .....	143
6.3.1.1	System Description .....	144
6.3.1.2	Physical Domain Model .....	146
6.3.1.3	Numerical Results .....	146

6.3.2 Loosely-Coupled System -- Centralized Synchronization .....	149
6.3.2.1 System Description .....	149
6.3.2.2 Physical Domain Model .....	151
6.3.2.3 Numerical Results .....	151
6.3.3 Loosely-Coupled System -- Distributed Synchronization .....	154
6.3.3.1 System Description .....	154
6.3.3.2 Physical Domain Model .....	156
6.3.3.3 Numerical Results .....	156
6.4 Discussion of Results .....	159
<b>7 CASE STUDY B: SIGNAL PROCESSING APPLICATION .....</b>	<b>162</b>
7.1 Application Description and Analysis .....	162
7.1.1 Single Instance of Process P .....	163
7.1.2 Random Process Arrivals .....	166
7.2 Evaluation of a Simple System .....	168
7.2.1 System Description .....	168
7.2.2 Physical Domain Model .....	170
7.2.3 Numerical Comparisons .....	173
7.3 Evaluation of a Sophisticated Architecture .....	176
7.3.1 System Description .....	176
7.3.2 Physical Domain Model .....	178
7.3.3 Numerical Comparisons .....	182
7.4 Interpretation of Results .....	185
<b>8 SUMMARY AND CONCLUSIONS .....</b>	<b>187</b>
8.1 Summary of the Methodology .....	187
8.2 Application Considerations .....	189
8.3 Comparisons with Other Methods .....	192
8.4 Suggestions for Further Research .....	194



## ACKNOWLEDGEMENTS

This research was supported in part by the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

I wish to express my appreciation to my Doctoral Committee, Professors Kirby Baker, Milos Ercegovac, Mario Gerla, Richard Muntz and Bruce Rothschild. I am especially grateful to my thesis advisors, Professors Richard Muntz and Milos Ercegovac, for their continued guidance, support and stimulating discussions throughout the course of this research.

I would also like to acknowledge the cooperation Professor Michel Melkanoff in allowing me the unrestricted access to the CAD/CAM Laboratory and the use of its computing facilities for running numerous simulation experiments.

## VITA

- June 27, 1961 — Born, Odessa, U.S.S.R.
- 1974 — emigrated to U.S., established permanent residency
- 1981 — B.S., Mathematics-Computer Science,  
— University of California, Los Angeles
- 1982 — M.S., Computer Science  
— University of California, Los Angeles
- 1986 — Ph.D., Computer Science  
— University of California, Los Angeles

## ABSTRACT OF THE DISSERTATION

Analytic Modeling Methodology for Evaluating the  
Performance of Distributed, Multiple-Computer Systems

by

Alexander Kapelnikov

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1986

Professor Richard Muntz, Co-Chair

Professor Milos Ercegovic, Co-Chair

In this dissertation, we describe an analytic modeling methodology for evaluating the performance of distributed, multiple-computer systems. The concepts and techniques of this methodology are useful for the approximate analysis of a wide range of distributed computing environments and communication networks. The main strategy of our approach is to segregate, as much as possible, the model of the “logical” behavior of an application (a program or a process) from the model of its underlying execution environment. For representing program behavior, graph-based techniques are used, while extended queueing networks are utilized for modeling system architectures. The solutions of both types of models are combined to estimate the performance of a distributed system in executing some selected applications.

To illustrate the practical application of the methodology introduced in this dissertation and provide an indication of its expected accuracy level, we have included two case studies.

## CHAPTER 1

### INTRODUCTION

After computers (and other machines for automating and improving some of our human functions) had become accepted by industrial and business communities, their original manufacturers immediately adopted the computer model championed by *von Neumann*, one of the foremost pioneers of computer science. Subsequent producers of computers and related products followed in the footsteps of their predecessors and continued this trend of centrally controlled, single-processor systems with updateable memory. Even though remarkable advances in technology, particularly in electronics and manufacturing techniques, resulted in phenomenal increases in speed and decreases in size and cost of components, the general architectural principles remained basically unchanged [BAC78]. Perhaps, computer manufacturers felt comfortable with the organization of the von Neumann architecture, since it closely paralleled *human organizations* which had proven track records, such as those commonly found in large corporate environments. For that and other reasons, this traditional design approach was loyally followed for almost three decades.

However, no matter how rapidly our technology progresses, the basic laws of physics (as currently accepted) limit the maximum speed attainable by a single-processor computer to a very large, *but still finite*, value. Therefore, during the past decade, there has been a growing interest in investigating feasible alternatives to the von Neumann design. Particular consideration has been given to designs with *distributed* control and *multiple* processing units, capable of concurrent, asynchronous operation. Since the execution of a program requires the cooperation of these processing units, and cooperation entails *communication* (similar to that in human or-

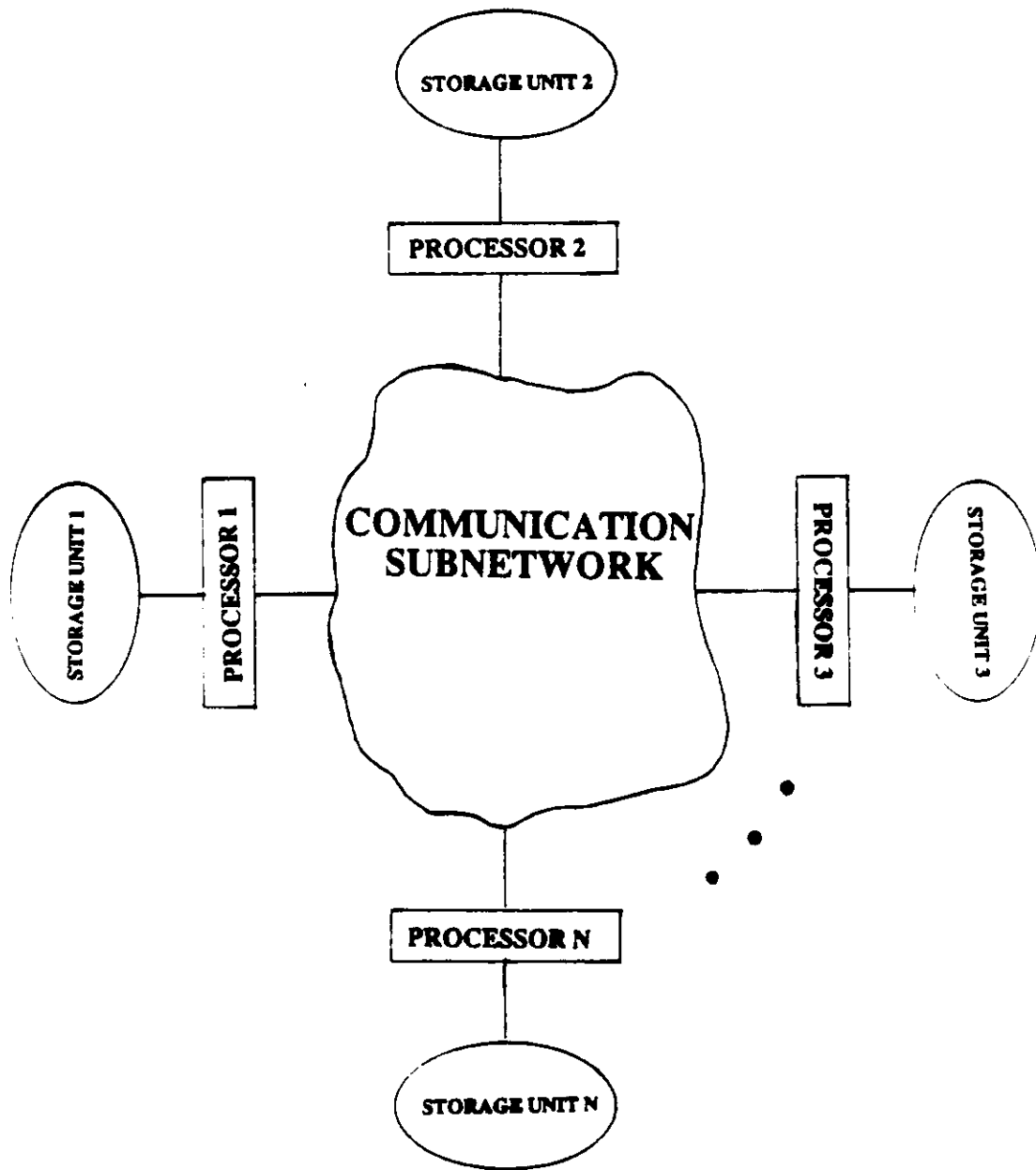
ganizations), there must be a mechanism to facilitate the exchange of pertinent information among processors. A proper choice of this communications mechanism is one of the most important issues in the design of such systems. Figure 1.1 is a conceptual diagram of a generic, distributed, multiple-computer system. This system consists of  $N$  nodes which are interconnected through some communication subnetwork. Each node is comprised of a processor and a storage unit. The size and sophistication of each individual processor can range from those of a DMA controller to those of a general-purpose, mainframe computer. The storage units can also vary considerably in terms of capacity and technology used. The possibilities for the communication subnetwork are virtually limitless (as will be shown in Chapter 2).

This thesis addresses the problem of analytic performance evaluation of distributed system architectures. In this introductory chapter, we shall first describe some applications of the distributed system concepts and illustrate the need for performance prediction. We shall then review some of the other currently available performance analysis tools. Finally, we shall evaluate those tools, identify their shortcomings and discuss what improvements we plan to achieve with our modeling methodology.

## 1.1 Overview of Distributed Systems

The architectural concepts presented above can be applied at almost any level of the computer systems hierarchy -- from the design of a circuit in a VLSI chip all the way to the design of an intercontinental computer network. In this section, we shall discuss their application at two of the more pertinent of these levels.

First, we consider the level of self-contained, general-purpose computers. *Data-flow architectures* serve as good examples of the application of *non-von Neumann* concepts at this level. A typical data-flow computer consists of a number of processing units, which are interconnected by data buses, cross-bar switches or other



**Figure 1.1 Generic Distributed Multi-Computer System**

means and one or more memory modules (usually of the associative kind), which are used to perform the matching of instruction operands into executable sets. This class of computers has been under investigation by many research groups since the early 1970s. Most of the pioneering work in this field had been performed by Dennis and his colleagues at MIT [ARV80, DEN83]. Several detailed simulators and prototypes of such computers have been constructed at different universities and used to obtain performance measurements and experiment with various design alternatives. These simulated computers are usually programmed using *data-flow languages*, which are a subset of the class of functional languages [ACK82]. In a data-flow machine, an instruction is ready to be executed as soon as all of its operands are available. The major advantages of such languages are that they are free of side-effects and exploit all of the parallelism available in a program. However, a serious drawback is the generally large instruction size, due to the overhead involved in explicitly specifying all instruction sequencing. Note that, in *procedural* languages used with von Neumann computers, most instruction sequencing is implicitly defined by the ordering of instructions in a program.

We then turn our attention to systems composed of a set of autonomous (and possibly heterogeneous) processors, each of which can fully function in a stand-alone fashion. These processors are usually interconnected through an external communication network and mutually cooperate to provide a pool of distributed computing power. These kinds of systems are typically used for transaction-oriented processing such as accessing and updating distributed data bases and in applications requiring a high degree of security and/or fault-tolerance [FIN84]. One of the major advantages of applying distributed design concepts at this level is the resource-sharing capability, which results in reducing both cost, by not having to replicate the complete functionality of a system at each site, and system response time, by having idle processors “assist” overloaded ones. The multi-processor Cm\* machine built at the Carnegie-Mellon University is a good representative of this type of system



[DEM82].

## **1.2 Need for Performance Prediction**

Even from the brief discussion presented above, we can infer that there is a countless number of design alternatives for developing any distributed system. Some of the more important design issues are: the configuration, size and technology of the interconnection network; distribution of system functionality among its processing units; allocation of workload (program tasks) to processors; how to synchronize execution of tasks; the size of individual tasks; where to store various data sets. A designer of a distributed system is presented with the dilemma of properly resolving these issues in order to meet certain objectives. These objectives may include such considerations as average execution times of certain types of programs, utilizations of individual processing units, communication and synchronization overhead, cost/performance ratios of different system configurations, modularity, and user-friendliness. It is usually not until the final stages of system development that it is possible to determine whether or not the original objectives have been met. Thus, an incorrect choice, in deciding on any one of the numerous issues, can result in a very costly and time consuming re-design and re-development effort.

Given the aforementioned considerations, one can appreciate the importance of being able to predict the eventual performance of a system during its design process or primeval stages of development, so that any design flaws can be detected and corrected without great expense. With proper performance prediction tools, a designer should be able to gradually "pilot" the design into meeting all of the required objectives.

### 1.3 Current Methods

Currently available performance prediction methods for distributed systems fall into two general categories. Methods of the first category employ *simulation* tools to construct and run a detailed model of a system. Usually general-purpose simulation packages, such as UCLA's SARA System (Graph Model of Behavior) [VER82], IBM's RESQ [SAU81b], or PAWS [INF81], are used, which have built-in facilities for gathering and analyzing performance statistics. Different packages can vary considerably in their modeling primitives, model definition languages, and performance measurement facilities. Thus, a model usually has to be re-designed and re-implemented in order to run in a different simulation environment. It is up to the modeler to determine what level of detail to implement in a model and how to properly abstract the pertinent characteristics of the actual system being evaluated. In deciding on the latter issues, one must consider what performance measures are being sought and what accuracy level is required.

In some cases, general-purpose packages may be intolerably slow in simulating very detailed models or not be equipped to provide all of the desired performance measures. For these reasons, several research groups have developed their own, special-purpose, hardware and software simulators to model specific distributed systems [FIN84, THO81]. A given special-purpose simulator can usually represent only a particular system architecture, although some are parameterized to be able to model different configurations of the same basic design.

The second category of performance analysis tools comprises approximate *analytic* techniques. Each such technique is generally applicable only to a narrow range of system architectures and specific types of program structures. These methods employ either standard queueing network models or graph models, which trace system states during program execution, e.g., Petri Nets [PET81]. The techniques based only on queueing networks, such as the one proposed in [HEI83],

currently suffer from the inability to explicitly represent interdependencies of tasks in a program. Such methods usually utilize external, Poisson arrival streams, with “heuristically” chosen customer arrival rates, to represent new tasks being enabled (i.e., spawned by already executed tasks). Procedures for computing numerical values for those rates vary with different modeling applications. In order to make a model more accurate, state-dependent arrival rates and dynamic class changes by queuing network customers can be used.

Most of the techniques based purely on graph models, such as Stochastic Petri Nets [MOL81], suffer from being very closely tied to a specific system architecture. In other words, the model of the program behavior is intimately linked with the model of the execution environment. Thus, a minor architectural change may require a new model to be constructed and solved. Another limitation of most graph-based methods is that their state space size usually grows combinatorially with respect to the size of the particular configuration being modeled, thus, limiting their practical applicability.

#### **1.4 Motivation for a New Methodology**

Both simulation and analytic categories of currently available performance prediction tools suffer from shortcomings which limit their range of practical application. Using general-purpose simulation packages involves not only a significant cost of developing a model, but also a large amount of expensive CPU time for every run of the simulation. Also, even if a single parameter is changed in a model, a complete, new set of simulation runs is required to determine the new performance statistics. Furthermore, the type of a computing environment necessary to support most general-purpose simulation packages is usually very sophisticated and expensive. Special-purpose simulators are generally more efficient, but their development is very costly and each is able to model only a specific system architecture.

The major limitation of most currently available analytic methods is that an individual technique is generally applicable to modeling only a very narrow range of systems. Methods based purely on queueing network models (e.g., the procedure described in [HEI83]) lack general procedures for “translating” the precedence relationships between tasks in a program into customer arrival streams and selecting proper arrival rates for those open customer chains. Methods based exclusively on graph models usually cannot capture all of the pertinent details of the program execution environment without generating excessively complex graphs. Also, such models can quickly become unmanageable by increasing the configuration size of the system being modeled.

Thus, with currently available performance analysis tools, one must sacrifice either computational efficiency, as with general-purpose simulators and some graph-based methods, or generality, as with special-purpose simulators, queueing network models, and other graph-based techniques. Our goal in this research has been to develop a new, general modeling framework for efficiently evaluating the performance of the broad class of distributed, multiple-computer systems, which enables a modeler to control the cost of solving a model according to the level of accuracy desired. The major premise of our methodology is that the graph-based methods are best suited for modeling precedence relationships between tasks in a program, while techniques based on queueing network models are best suited for representing the details of the execution environment. Therefore, by segregating the model of the program behavior from the model of the system architecture, we intend to exploit the advantages of both queueing networks and graph models where they are most beneficial.

## 1.5 Organization of Thesis

In this first chapter, we have presented a general overview of distributed systems, demonstrated the need for performance prediction and motivated the development of our modeling methodology. Chapter 2 will elaborate upon distributed design principles, describe and exemplify variations in program structures and system architectures and discuss design issues pertinent to achieving high system performance.

In Chapter 3, we will present the modeling portion of our methodology and describe in detail each of its two major components: the physical domain model and the program domain model. Chapter 4 constitutes the analysis and solution portion of our methodology, i.e., it details the procedure for solving the type of models developed in Chapter 3. In Chapter 5, we will demonstrate some heuristic techniques which can be used to improve the efficiency of the procedure presented in the preceding chapter and to also extend the range of application of our methodology.

The following two chapters will show how our methodology can be applied to the practical problem of modeling and analyzing actual distributed systems and parallel implementations of programs or transactions. In particular, in Chapter 6, we will study the effects of concurrency control in distributed computations and, in Chapter 7, we will evaluate the performance of a signal processing application implemented on a multiprocessor system. We will also analyze the accuracy of our methodology, compare its solutions with those obtained from simulation, and discuss the implications of the results of those experiments.

Chapter 8 will conclude this thesis by reviewing the whole methodology, identifying its main research contributions, discussing how to effectively apply its modeling and solution procedures, and providing guidelines for conducting further research of the yet unresolved issues.

## CHAPTER 2

### DISCUSSION OF DISTRIBUTED SYSTEMS

Before describing our modeling philosophy and solution methodology for distributed systems, we will present a discussion of these systems. This will help emphasize the importance of distributed processing, especially its impact on the industrial and academic computing communities, and the magnitude of the complexity involved in designing and modeling multiple-computer environments. After motivating the need for parallel computation, we will review the general principles underlying distributed architectures, discuss different system configurations, consider variations in certain properties of programs (or processes), and, finally, identify important design issues pertaining to the development of such systems.

#### 2.1 Need for Parallel Processing

As already mentioned in Chapter 1, conventional, centralized computer architectures will not be able to meet the ever-increasing data processing and problem solving needs of academic institutions, industrial organizations, and governmental agencies. For instance, future signal processing applications, such as missile tracking systems, will require billions of operations per second. This kind of throughput is not physically realizable by a single processor, no matter what kind of technology (real or conceptual) is used for producing logic elements. Even the processing capacity of CRAY-2, one of the fastest vector supercomputers currently available, which has an elementary clock cycle of 4 nanoseconds, is orders of magnitude away from this figure. On the other hand, the recent technological advances in the miniaturization of digital circuitry have made it possible for a significant amount of computer

processing power to reside on a single silicon chip. These advances will inevitably lead to an explosion in the production of inexpensive and ultra-small processors [COF79].

The considerations presented above make distributed, multiple-computer systems both economically feasible and necessary to satisfy current and future computing needs. Aside from performance issues, there are other important factors which make distributed processing desirable, e.g., fault-tolerance and system cost. Distributed architectures are inherently more fault-tolerant than centralized ones. Namely, individual processing elements can be repaired, serviced, and replaced without affecting the rest of the system; sensitive computations can be performed in parallel by several processors and individual results can then be compared with each other; data can be distributed among different storage facilities and critical information can be replicated. Computing costs can also be reduced by such systems: hundreds or even thousands of microprocessors, when properly connected together, can outperform a mainframe while having a much lower total price; sharing of system resources (e.g., peripheral devices, storage facilities, and communication channels) will eliminate the need for replicating expensive components; the system's computing power can be tailored to match specific user needs, thus optimizing utilization of each component; the computing environment can be gracefully re-configured, upgraded, and expanded, thus reducing maintenance costs and increasing system's life span.

## **2.2 General Principles**

In this thesis, we take a "broad" view of distributed, multiple-computer systems. In the most general sense, our definition includes any system which is composed of physically separate processors (computers) which cooperate with each other (via some communications and/or storage mechanism) in executing some number

of programs or processes. In the context of this paper, a program (process) is defined as a collection of *tasks*, each varying in complexity and functionality, which are ordered through precedence relationships in order to achieve some end result (e.g., solving a differential equation or replying to a database query). This broad definition includes not only the latest architectures, but also many systems which have already been in use for some time.

Distributed processing systems can vary in the type of communications mechanism used, the degree of coupling between different processors, the policy for scheduling execution of tasks, how synchronization of tasks is accomplished, and the degree to which functionality and control of the system is distributed. Some of the commonly used communications facilities are data buses, token rings, cross-bar switches, shared memories, and coaxial or twisted-pair cables. The systems with a high degree of connectivity or which utilize shared memories for intertask communication are normally termed "tightly coupled." Architectures with larger spatial distribution (i.e., longer propagation delays) and less frequent interaction between processors, such as message passing systems, are called "loosely coupled." Distribution of control can range from the logical star configuration (i.e., a single, master processor) to being equally shared by all computing resources. The functionality of the system can be either replicated at each processing site or uniformly distributed between all elements. The variations discussed above give rise to many different classes of parallel processing architectures. Some of the more important classes are systolic arrays, networks of von Neumann computers (either tightly or loosely coupled), and data flow architectures.

Unfortunately, the class of distributed, multiple-computer systems described above is not exempt from the idiom expressed by the common cliché: "There is no such thing as a free lunch!" Even though these systems offer an elegant and structured approach to attaining high computing performance, one must pay certain



“penalties” when using them. One such penalty is the communications overhead due to information interchange between processors. Another is the additional processing time and storage access needed to resolve precedence relationships between tasks. More will be said about these two penalties when we discuss design issues in section 2.5. Also, the task of programming such computers is made more complex by requiring that programs be partitioned into concurrently executable tasks, which are then “properly” allocated to different processing elements, and that asynchronous control be provided for synchronization of tasks. In order to have high performance and yet maintain low software development and maintenance costs, multiprocessor systems must be able to perform these tasks automatically and be high-level language programmable.

### **2.3 Architectural Variations**

In order to make our modeling methodology as computationally attractive as possible without, however, sacrificing its generality, we will explore the potential advantages of considering variations in certain key properties of distributed architectures. In particular, we will classify the general class of distributed, multiple-computer systems according to two pertinent criteria. The type of task allocation policy employed by the system constitutes the first criterion. The second criterion is the way the synchronization of tasks is performed in the system.

In this section, we will discuss each criterion individually, present our classification of distributed systems, and then offer examples of different system categories.

### 2.3.1 Task Allocation Policies

For the purposes of our methodology, we will separate all possible task allocation policies into two general categories: Dynamic and Static. With a *dynamic* task allocation policy, each currently enabled task (i.e., task which is ready to be executed) competes, on an equal basis, with other enabled tasks for the processing and communication resources of a distributed system. That is, the required system resources are dynamically assigned to a task (by the task scheduling and resource management components of the architecture) at the time when it becomes ready for execution. The assignment of resources to tasks can be made either purely *probabilistically* -- without considering the distribution of workload currently present in the system -- or purely *adaptively* -- as a *deterministic* function of the current system state -- or by using a combination of these two approaches -- as a *probabilistic* function of the current system state.

With a *static* task allocation policy, each task of a given program is a priori allocated a pre-specified subset of the system's processing and communication resources. Such subsets do not necessarily have to be mutually exclusive nor represent an exhaustive partitioning of the total available system capacity. This allocation of resources is performed before commencing the execution of a program. The main objective of most static task allocation policies is to balance, as much as possible, the expected workload, represented by a program, among different system components, while exploiting all of the potential parallelism available in that program. Upon becoming enabled, each task of a program utilizes only those resources that were statically assigned to it according to the particular policy adopted by the system in question. It competes for each system element against other enabled tasks which also have that element as part of their resource allocations. As will be discussed in the next chapter, when modeling the execution of a program in an environment where a static task allocation policy is employed, we will group the tasks of

that program according to what resources were allocated to them.

### 2.3.2 Synchronization Schemes

In the context of our modeling strategy, we will classify the various schemes for synchronizing execution of tasks as being either centralized or distributed. The reasons for making this distinction will become evident when the model itself is described in the next chapter. With a *centralized* synchronization of tasks, all information necessary to determine when each task can become enabled (i.e., status of each disabled task) is kept in a single (central) storage facility. Whenever a task completes its execution, it generates a completion acknowledgment -- or a result packet -- to indicate the occurrence of that event. Those result packets are then used to create operand packets which contain the data (operands) needed for updating the status of some disabled tasks. Since the information about each task is stored in the same location, only a *single* operand packet has to be generated from a given result packet. That is, no decision is required to determine how many operand packets to generate and where they should be sent. As will be shown in the following chapters, systems which utilize a centralized synchronization mechanism can be represented by much simpler models, which, in turn, significantly reduce the computational complexity of the corresponding solution process.

With a *distributed* synchronization of tasks, storage of information about disabled tasks is distributed among a number of memory modules. Each result packet will generate as many operand packets as the number of different storage modules which need to be updated with the information contained in that result packet. Each generated operand packet will contain only the data required to update the status of disabled tasks stored in a particular location and it will be sent to that module only. The transmission and processing of operand packets generated from the same result packet can be performed asynchronously. Usually, a distributed synchronization

mechanism is employed in conjunction with a static task allocation policy, although, there is no physical constraint requiring that. That is, potentially, the system resources utilized to execute a task can be independent of where the status of that task is stored.

From the perspective of our modeling methodology, there is a significant difference, when evaluating the execution of programs, between architectures employing distributed synchronization schemes and those with centralized synchronization schemes. This difference manifests itself in both the size and complexity of models and the computational cost of the associated solution process.

### **2.3.3 Classification of Distributed Systems**

As discussed in the preceding sections, we are primarily concerned with two architectural criteria for classifying the broad class of distributed, multiple-computer systems:

- (a) task allocation policy: dynamic or static; and
- (b) synchronization scheme: centralized or distributed.

Since the aforementioned criteria are independent of each other and each criterion offers two possible alternatives, there is a total of four feasible combinations. Thus, our classification of distributed, multiple-computer systems consists of the four categories listed below:

- (1) Dynamic Allocation with Centralized Synchronization
- (2) Dynamic Allocation with Distributed Synchronization
- (3) Static Allocation with Centralized Synchronization
- (4) Static Allocation with Distributed Synchronization

The major reason for distinguishing between these four categories is to optimize our methodology in terms of efficiency and ease of application. In other words, our objective is to avoid redundant complexity in our models as much as possible. In fact, as will be shown in the next chapter, the structure of models for representing systems of the fourth group is the most general one and can be used to model systems belonging to other categories as well. However, this would make the analysis of systems of other classes less efficient by introducing unnecessary complexity in constructing and solving their models. Thus, although closely related to each other, the modeling framework for each category is specifically adapted to the corresponding system properties, containing just enough detail to allow for all distinguishing features of that category to be represented.

In the next section, we will present examples of systems of each of the four categories.

### **2.3.4 Examples of Different Categories**

The original data-flow computer architecture proposed by Dennis, et al. at MIT [DEN83] exemplifies the “Dynamic Allocation with Centralized Synchronization” category. An example of the “Dynamic Allocation with Distributed Synchronization” class is the Cosmic Cube, which is a tightly coupled network of general purpose microprocessors, each with its own local memory [SEI85]. The “Static Allocation with Centralized Synchronization” category is well represented by particular configurations of the Rumbaugh data-flow machine [RUM77], the Texas Instruments system [GAU82], the Manchester University machine [GUR83], and Sandia Labs’ SANDAC computer. The UC Irvine data-flow simulator [THO78, GAU82], the BBN Butterfly Machine and Hughes Dataflow Multiprocessor [FIN84] belong to the “Static Allocation with Distributed Synchronization” category. The Cm\* system, built at the Carnegie-Mellon University, can be configured to

represent any one of the four categories listed above [SWA77].

## 2.4 Program Variations

Programs, as defined in the context of this dissertation, can vary on both the *local* (individual task) level and on the *global* (structural) level. Differences in individual task properties account for variations on the local level. Differences in the number of tasks of each type present in a program and in the precedence relationships among those tasks account for variations on the global level.

In this section, we will discuss program variations on each level and how these variations affect our modeling methodology, and then present examples of parallel algorithms.

### 2.4.1 Task Attributes

Individual tasks can consist of one or more “simple” instructions. By “simple,” we mean an instruction which can be translated into a single *machine-level* operation (in the context of the system being modeled). We use the term “granularity of a task” to mean the number of simple instructions comprising a task. Furthermore, depending on the types of processors used in the system, different machine-level operations may require different number of cycles, i.e., different instructions may have different execution times. Thus, the *processing* demands of tasks can vary considerably. Note that, by our definition, a task is always executed as a single unit, no matter what its granularity is. That is, all of its constituent simple instructions are executed sequentially by the same processor and a single result packet is generated after the last instruction has been executed. This definition of a task does not significantly limit the range of application of our methodology. For instance, a task which generates several results at *different* times during its execution can be represented as a sequential combination (described in the next

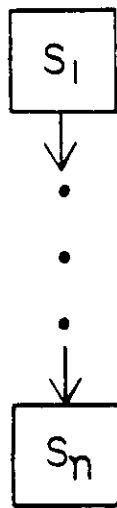
section) of smaller tasks. Upon completion, each “subtask” generates one of the results of the original task, the particular result being determined by the order in which that subtask is executed (i.e., the  $i$ -th subtask generates the  $i$ -th result).

Depending on its granularity and precedence relationships with other tasks, each task may generate a different number of results. Thus, the lengths of result and operand packets may differ, depending on the originating and destination tasks. Since the time to transmit a data packet, over most communications facilities, is a function of its length, the *communication* demands of tasks can also vary significantly.

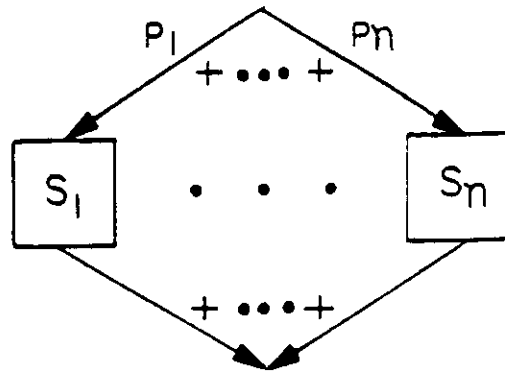
As will be shown in Chapters 3 and 4, both the complexity of our models and the computational cost of solving them are directly related to the degree of variation among tasks in the programs being considered.

#### 2.4.2 Structural Variations

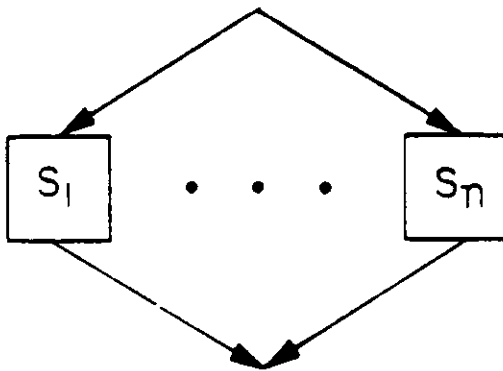
Figure 2.1 illustrates some of the common *programming constructs*, each represented as a combination of program modules, found in typical programs. The *sequential* construct is formed by combining modules in series. It is representative of programs consisting of a sequence of procedures, and of transactions involving a serial application of several operations to the same data item. In the *EXCLUSIVE-OR* construct, as its name implies, exactly one out of several possible modules is selected for execution. The  $i$ -th module is selected if and only if predicate  $p_i$  is satisfied. At each instance, exactly one predicate holds true, while the others are false. This construct is representative of the “Case” statement in programs and of the conditional transaction execution. The *parallel* construct is formed by combining modules in parallel. It is representative of concurrent transaction processing and of executing independent computations in an algorithm. The *loop* construct is formed by having module A enable itself after enabling module B (thus starting the



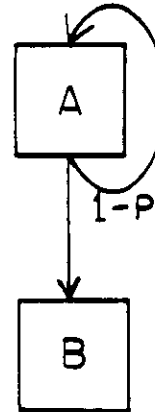
(a) SEQUENTIAL CONSTRUCT



(b) EXCLUSIVE-OR CONSTRUCT



(c) PARALLEL CONSTRUCT



(d) LOOP CONSTRUCT

Figure 2.1 Common Programming Constructs



loop over again) if predicate  $p$  is satisfied. This construct is representative of a “DO loop” which repeats itself as long as condition  $p$  holds true, where each iteration is a serial combination of modules  $A$  and  $B$ ; however, a new iteration can start as soon as module  $A$  of the current iteration is completed. It can also be used to model a recursive procedure, having modules  $A$  and  $B$  as its body, which calls itself after completing module  $A$  (as long as predicate  $p$  is satisfied), with module  $B$  being independent of the new procedure invocation.

Most well structured programs can be decomposed into a hierarchy of simple constructs, where constituents (program modules) of constructs on a given level are themselves constructs on the lower level of the hierarchy. Conversely, common programming constructs can be applied hierarchically in combining modules to produce programs having very complex structures. We will elaborate on the latter issues in Chapter 3 when describing program behavior models. In Chapter 5, we will show that this kind of structural decomposition of programs can be effectively used to increase the efficiency of our solution process.

### 2.4.3 Examples of Parallel Algorithms

The first example of a parallel algorithm comes from image processing applications. This algorithm is called the SOBEL operator and it is shown in Figure 2.2(a). The SOBEL operator is an edge enhancement algorithm which functions as follows. For each pixel ( $E$ ), a window of the eight nearest neighbors ( $A, B, C, D, F, G, H$  and  $I$ ) is defined. The gradient operator is then applied to process each inside pixel of the image matrix. This is a highly repetitive application, where operations on different pixels are all treated independently of each other [GAU83].

$$E = |X| + |Y|$$

$$X = (C + 2F + I) - (A + 2D + G)$$

$$Y = (A + 2B + C) - (G + 2H + I)$$

(a) SOBEL operator

```

DO 20 I = 1, N
C
C      DO ALL (COLUMN SWEEP)
C
C      DO 10 J = I + 1, N
C          X(J) = X(J) - L(J, I) * X(I)
10      CONTINUE
C
C      END DO
C
20 CONTINUE

```

(b) Column Sweep algorithm

```

procedure quicksort (A, n)
  i + n = 0 then return ({});
  i + n = 1 then return (A);
  m := n/2;
  i := 0;
  k := 0;
  for i from 1 to n (i /= m) do;
    if A[i] < A[m]
      then j := j+1;
      B[j] := A[i];
    else k := k+1;
      C[k] := A[i];
  end;
  return ( quicksort (B, j) || A[m] || quicksort (C, k) )
end quicksort

```

(c) recursive Quicksort procedure

**Figure 2.2 Examples of Parallel Algorithms**

The second illustration of parallel computation is taken from the domain of linear systems. We have selected one of the algorithms for solving linear recurrence equations. This algorithm, known as a “column sweep,” is a forward substitution of the current value of the unknown,  $x$ , into all of the expressions that require this value [MON81]. Columns from the  $N \times N$  coefficient matrix  $L$  are fetched and multiplied by the current  $x$  value, thus “sweeping” each column of coefficients in a single step. An implementation of the column sweep algorithm in FORTRAN is given by Figure 2.2(b).

The last example is given by a recursive implementation of the well known Quicksort algorithm [KNU73]. A design definition of such implementation is depicted in Figure 2.2(c). Procedure Quicksort sorts an  $n$ -element vector  $A$  into a non-decreasing order, utilizing two auxiliary vectors:  $B$  and  $C$ . At the beginning of each iteration, vectors  $B$  and  $C$  are empty and the middle element of  $A$  is chosen as the sort key for that iteration. At the end of the iteration,  $B$  contains all of the elements of  $A$  having values less than or equal to the value of the key (not including the key itself) and  $C$  contains those elements with greater values. Finally, vectors  $B$  and  $C$  are each sorted individually using Quicksort.

In Chapter 3, we will illustrate how to model the example programs described in this section.

## 2.5 Design Issues

From the material presented in this chapter, it is clear that there is virtually an infinite variety of possible distributed system designs. In order to make proper design choices, a systems architect must identify those issues which are important to achieving the specified objectives (e.g., having the system meet certain requirements). In the following, we will briefly discuss some of the design issues critical to achieving high system performance.

Perhaps the most important performance-oriented design issue is the communications mechanism used for interchanging data and control information (e.g., definitions of tasks, result packets, and operand packets) among processing elements and storage units of a system. Each individual property of such communications mechanism must be carefully selected in order to maximize the information throughput and system component utilizations and to minimize the communication delays. These properties include the physical transport medium, the topology of processing elements and data storage modules, data interchange protocols, addressing and routing mechanisms, data link reliability, message sizes, data encoding, etc. The selection of the aforementioned properties is usually constrained by the desired speed and the size of the system, the geographical distribution of its components, fault-tolerance and system availability requirements, and bounds on the development and maintenance costs. In most cases, it is not possible for a given choice of the communications mechanism to be optimal in both meeting high and balanced system utilization and maintaining fast response time. Thus, a designer has to make a compromise between effectively using available system resources and providing prompt service to the user community. Various alternatives for the communications mechanism employed by distributed architectures have been proposed and evaluated in a number of research papers [GUR83, KUC77, THO81, TRE82] and are beyond the scope of this dissertation.

The proper selection of the task allocation policy and the task synchronization scheme, such as those described in Section 2.3, is also very important to achieving high system performance. In architectures which are geared toward specialized applications, where properties of programs to be executed are known in advance, it is usually advantageous to employ a static task allocation policy with a distributed synchronization mechanism. Since, with special-purpose systems, only a specific set of applications has to be considered, a designer has an opportunity to experiment with various task allocations a priori and to adopt the optimal one. A general-

purpose system in an environment of dynamically changing and unpredictable user demands would be better off using a dynamic allocation policy and a centralized synchronization scheme.

Some of the other pertinent design issues are the granularity of individual tasks, how the storage of data is distributed among different modules, how system resources are shared, and how the functionality of a system is distributed among various processing units. The granularity of tasks, in particular, has been shown to significantly affect the execution time of a program, especially in systems having high communication and synchronization overhead [GAU82, GAU85].

## CHAPTER 3

### THE MODEL

Our methodology embodies two modeling domains. The first (physical) domain consists of the physical resources comprising the distributed, multiple-computer system being modeled, such as processors, communication buses, peripheral controllers, I/O device drivers, storage facilities, etc. The second (program) domain comprises programs or processes, each consisting of a set of cooperating tasks, which are being executed using the constituents of the first domain.

The physical domain model is used to solve for the throughput rates of the system for different types of tasks under various task loadings and population mixes. These rates are then used to compute the parameters for driving the program domain model. The solution of the program domain model, in turn, yields the average execution time of the program being analyzed.

#### 3.1 Physical Domain Model

Our model of the physical system resources, their interconnections, and interactions with each other consists of the conventional queueing network components, with an addition of a new modeling construct. This construct is termed a **Black Box** and is graphically represented as [?]. Its operation will be described in detail later in this chapter. For now, it suffices to say that those functions of a distributed system modeled by a Black Box are synchronization of execution of tasks and creation of executable operand sets (each set containing all of the operands required for the execution of some task) from results received from already completed tasks.

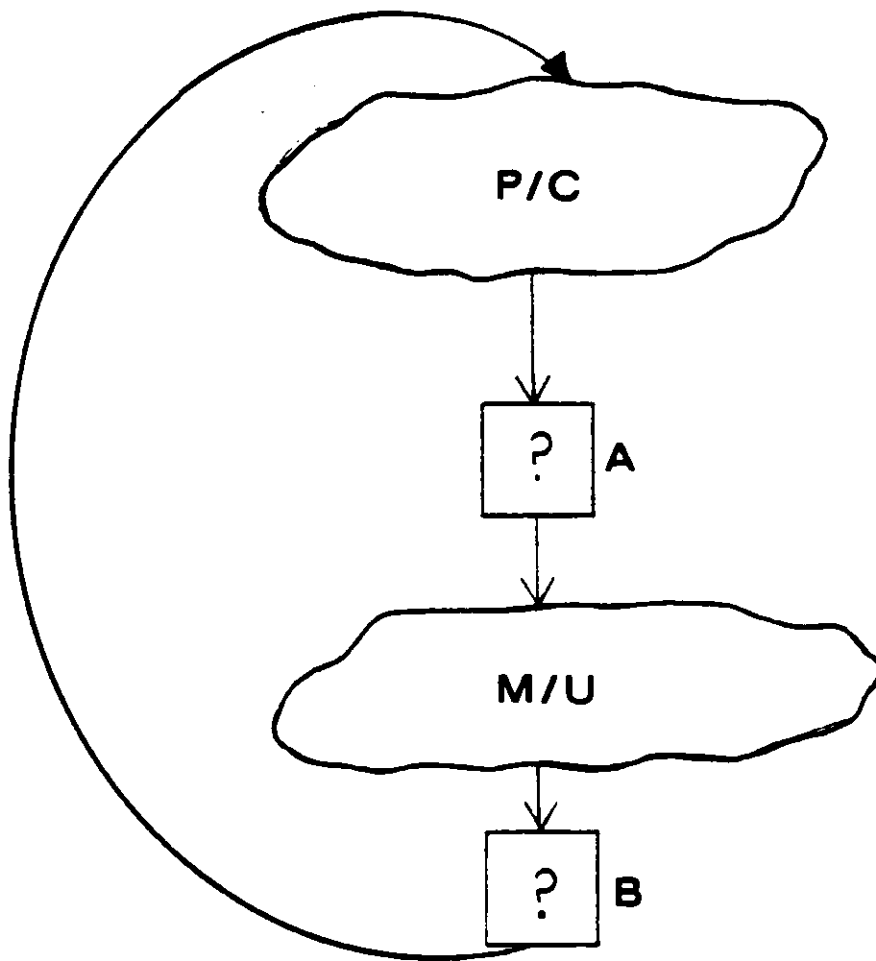
In essence, a Black Box provides a virtual link between the two modeling domains.

The most general type of a model in this modeling domain consists of a “*P/C*” (*P*rocessing and *C*ommunication) subnetwork, an “*M/U*” (*M*atching and *U*psdating) subnetwork and two Black Boxes. Figure 3.1 is a high-level representation of such a model. As will be shown later, however, it is sufficient (and desirable) to use just a *P/C* subnetwork and one Black Box for most modeling applications. We shall now describe each of these components in detail.

### 3.1.1 The *P/C* Subnetwork

The *P/C* subnetwork consists of standard queueing network elements (which are sufficient for modeling the organization and components of most computer architectures and communication networks). Its service centers and the associated queues represent the *processing* and *communication* resources of the distributed system being modeled and the contention for those resources. The interconnection of the elements, together with the associated routing probabilities, represent the architectural profile of the system. The attributes of each element (e.g., service time distribution of each customer class, number of servers, type of servers, server capacities, queueing discipline used) are determined by the characteristics of the underlying physical resource and the properties of tasks utilizing that resource.

The major distributed system functions modeled by the *P/C* subnetwork are: (1) receiving executable operand sets and corresponding task definitions from the synchronization subsystem (defined in the following section); (2) transmitting them to the proper processing elements for execution; (3) executing task code; and (4) transmitting the results back to the synchronization subsystem. An operand set, the corresponding task, and the generated result are all modeled by a single queueing network customer. In order to be able to model situations where the aforementioned entities utilize the same system resources, queueing network customers are allowed



**Figure 3.1 Distributed System Model**



to dynamically change classes. (In the context of this thesis, the terms “class” and “chain” are given the respective definitions used in the standard queueing network terminology.) Class changes enable a given customer to behave differently at the same service center during different visits. Thus, the particular entity that a customer is representing at a given time point depends on the service center it is then visiting and on the class to which it currently belongs.

Figure 3.2 shows an example of a section of a distributed system and the corresponding *P/C* subnetwork. Service center  $IB_i$  represents the delay (including any queueing time) incurred in transmitting an instruction packet (i.e., a task definition and its operand set), destined for processor  $i$ , on *input* bus  $i$ . Service center  $P_i$  represents the delay incurred in executing an instruction (task) by processor  $i$ . Service center  $OB_i$  represents the delay incurred in transmitting a result packet, constructed by processor  $i$  upon completion of a task execution, on *output* bus  $i$ .

As already mentioned above, the queueing network customers which visit the *P/C* subnetwork represent *tasks* (together with the corresponding operand sets) which are ready to be executed and *result packets* generated by those tasks. These customers may belong to different chains if the underlying tasks can be separated into groups, where the properties of tasks in each group (e.g., degree of granularity, average execution time, system resources required, etc.) are significantly different from those of other groups. In such a case, the *P/C* subnetwork model is said to be of a “multiple class” variety. We will elaborate on this issue when we discuss models in the program domain.

### 3.1.2 The M/U Subnetwork

Just like the *P/C* subnetwork, this subnetwork also consists of standard queueing network elements. Its service centers represent delays incurred in contending for, accessing, using, and updating the synchronization subsystem -- the part of

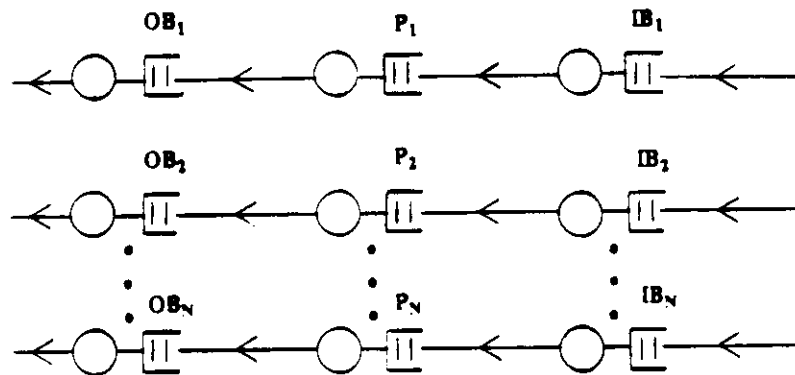
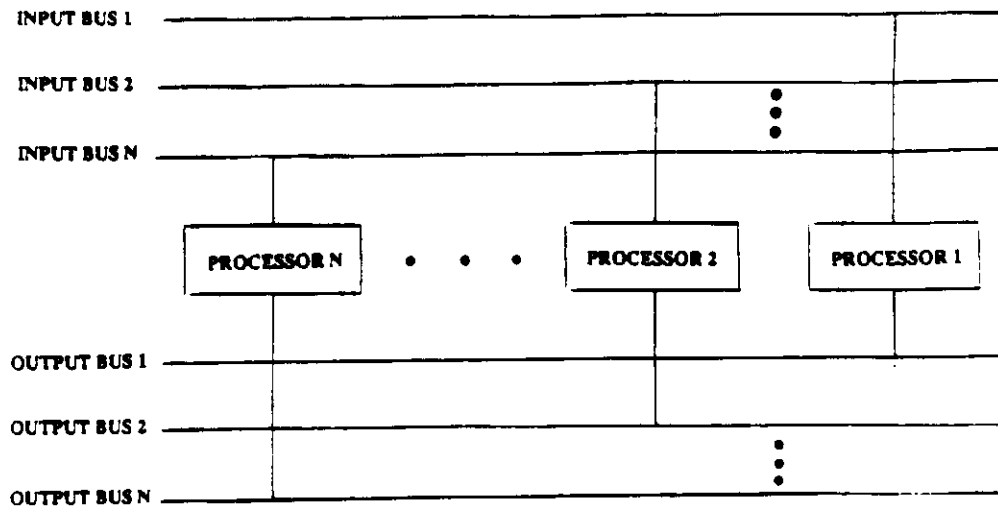


Figure 3.2 Physical System Components and the Corresponding P/C Subnetwork

the distributed system which is responsible for maintaining the information necessary for task synchronization. An example of such subsystem is the associative storage, where incomplete operand sets are temporarily stored. The service centers of the  $M/U$  subnetwork are used to model such physical system resources as associative memory modules, DMA controllers, channel controllers, memory buses, etc. If there is more than one memory module in the synchronization subsystem, then the system belongs to the "Distributed Synchronization" category, as discussed in the previous chapter. In our model, the physical resources represented by the  $P/C$  subnetwork and the ones represented by the  $M/U$  subnetwork constitute mutually exclusive and exhaustive partitions of the distributed system being modeled. The reason for requiring these subnetworks to be mutually exclusive is to allow each subnetwork to be solved independently of the other one, thus, reducing the computational complexity of our solution process. This issue will be elaborated on in Chapter 4. The aforementioned restriction should not significantly limit the range of application of our methodology.

The queuing network customers which visit the  $M/U$  subnetwork represent *operand packets* (or already-executed tasks' "completion acknowledgments") which contain information for updating and/or creating task operand sets in certain memory modules. Operand packets are created directly from the result packets sent by executed tasks. From each result packet, as many operand packets are generated as the number of affected memory modules. Each operand packet is received and processed by only a single memory module, i.e., operand packets are not shared by multiple memory modules. We will also assume that all operand packets destined for a given memory module exhibit the same behavior, i.e., they all place the same demands on the processing and communication resources of the system. Thus, each customer visits only those service centers which model the resources associated with a particular memory module. Furthermore, there are as many customer chains in this subnetwork as there are distinct memory modules. The particular chain to

which a given customer belongs is determined by the memory module for which the operand packet (represented by that customer) is destined.

As will be shown in the following section, it is not necessary to have an *M/U* subnetwork as a separate entity when dealing with systems of the “Centralized Synchronization” category (i.e., systems having a single, shared operand storage facility). In such cases, it is possible to integrate the service centers contained within the *M/U* subnetwork into the *P/C* subnetwork.

### 3.1.3 The Black Box Construct

This construct is introduced, as an addition to the standard set of tools used in describing queueing networks, in order to facilitate convenient modeling of the execution of parallel programs in the distributed, multiple computer environment. We will use the [?] symbol to graphically represent this new modeling component.

The Black Box is a conceptual artifact designed to model *interdependencies* between tasks in a program and *synchronization* of their execution, according to the precedence relationships among them. (Precedence relationships between tasks in programs can be easily determined from the underlying computation control graphs, which are discussed in the next section.) When incorporated into a queueing network model, a Black Box can be viewed as a service center having the following special properties. Let  $S$  be the current state of the physical system with respect to the execution of the program being modeled. (The current state of the Black Box is uniquely determined by  $S$ .) A queueing network customer which visits the Black Box service center is immediately “destroyed.” In turn, for each customer chain  $i$ ,  $f_i(d|S)$  customers are created, where  $d$  is either the task or the operand packet represented by the “destroyed” customer. The range of each  $f_i$  function is the set of non-negative integers and its definition is determined by the corresponding program domain model. Thus, a Black Box performs the roles of both a source and a sink in a

should be represented by such "reduced" models.

The selection of criteria for determining when such an approximation is justifiable and the selection of procedures for performing this approximate model reduction are application-dependent issues and cannot be universally specified for all cases. We will, however, attempt to give some "rules of thumb" to assist a modeler in making proper decisions. The number and diversity of storage sites used for maintaining incomplete operand sets are certainly important factors. In general, the fewer memory modules are used and the more similar those modules are (in terms of the associated queueing and access delays), the better the approximation should be. The quality of approximation also strongly depends on the relative processing and communication requirements of operand packets, as compared to those of tasks and result packets. The latter measure indicates what percentage of the total program execution time is spent on processing operand packets. Finally, the more homogeneous the behavior of operand packets destined for different storage locations is (in terms of utilization of system resources), the more accurate the approximate model should be. Note that the behavior of operand packets is always "homogeneous" if there is only a single storage location (i.e. if the system belongs to the "Centralized Synchronization" category).

Since, in systems employing centralized synchronization schemes, only one operand packet is generated from each result packet, the following heuristic approach may be used for approximating a distributed synchronization by a centralized one. All operand packets generated from the same result packet should be represented by a single queueing network customer. To obtain an upper bound on the program execution time, this customer should visit all service centers which represent system resources utilized by the corresponding operand packets. In order to obtain a lower bound, it should visit only those service centers which represent a single storage site -- normally the one with the smallest associated queueing and access de-

lays. One of the drawbacks of this approximation is that, depending on the application, the bounds it generates can potentially be very loose. An example of the application of the heuristic procedure described above will be given in Chapter 4.

### **3.2 Program Domain Model**

A program, as defined in the context of this dissertation, is a set of tasks, which are executed according to the precedence ordering given by the associated computation control graph. Computation control graphs are two-dimensional, directed structures, which pictorially represent interdependencies of tasks in a program. They will be defined and explained in detail in the following section. Tasks may differ in terms of the number of operands required, the number of results generated, the number of primitive (machine-level) instructions within a task, the execution time of each primitive instruction, and the way system's resources are utilized.

Of course, the representation of a program defined above is not the way an application programmer would normally code an algorithm. Most end-users write their programs in conventional, high-level languages, without regard for the architecture of the execution environment or any parallelism inherent in algorithms. Thus, we require an intermediate processing step, which compiles or translates a program from some high-level language into a computation control graph. This compilation process is itself a very extensive and complex subject, which has been addressed in detail by several research works [ACK79, ACK82], and is beyond the scope of this paper.

The reason for choosing this type of program representation is that computation control graphs lend themselves very well to behavioral analysis, from the perspective of tracing the states of a program's execution. This point will be elaborated on in the following chapter. Also, the compilation of a program into a computation control graph does not depend on the specific physical environment where the pro-

gram is to be executed. We will now give a detailed definition of computation control graphs and discuss their applicability to modeling program behavior.

### 3.2.1 Computation Control Graphs

*Computation control graphs* are a mechanism (created for the purposes of our methodology) for pictorially representing program behavior. The creation of this mechanism has drawn on many concepts inherent to the already existing graph models of program behavior [DEN72, EST78, FER72, KAR67, MOL81, PET81]. However, as will become evident later, it offers a number of salient features not found in other graph models, which make it particularly suitable for our methodology. These novel features include:

- convenient representation of recursive relationships;
  - flexibility in modeling looping constructs and multiple instantiations of tasks;
  - hierarchical grouping of precedence relationships;
  - allowing for automated generation of Markov processes
- (to be addressed in Section 4.3.2).

Each computation control graph is a collection of *nodes* and *directed arcs* connecting those nodes. The nodes represent the tasks comprising a program and the arcs model the precedence relationships among those tasks. An arc is said to be enabled when the task corresponding to its source node has completed its execution. Several arcs can emanate from a single node and a single node can be a destination for multiple arcs. Thus, with each node  $i$ , we will associate two sets of arcs:  $E(i)$  and  $D(i)$ .

$$E(i) = \{s \mid \text{arc } s \text{ emanates from node } i\}.$$

$$D(i) = \{s \mid \text{arc } s \text{ terminates at node } i\}.$$

Within each set, arcs are grouped using the AND, OR and UNION operators, which

will be discussed below.

The precedence relationships between tasks in a program are directly obtainable from the interconnection of nodes in the corresponding computation control graph (i.e., the way arcs are grouped in the  $E(i)$  and  $D(i)$  sets, for each node  $i$ ). The following rules apply when interpreting the "meaning" of arcs *terminating* at the same node. If there are several arcs terminating at a particular node without any symbols between their heads, then the task represented by that node cannot be initiated until all of the tasks represented by the source nodes for the arcs have completed their execution. This is an example of the (implicit) AND relationship among arcs. For node  $i$ , this relationship is formally expressed as:

$$D(i) = \{j_1 * j_2 * \dots * j_n\},$$

where  $j_1, j_2, \dots, j_n$  are the source nodes for the arcs terminating at node  $i$ . The AND operator is represented by the "\*" symbol. The plus sign "+" is used to represent the OR relationship among arcs. That is, if the arcs in the preceding example had "+" symbols between their heads, then the task modeled by the destination node could be executed as soon as any task modeled by one of the source nodes had completed its execution. For node  $i$ , this relationship is formally expressed as:

$$D(i) = \{j_1 + j_2 + \dots + j_n\},$$

Note that the OR relationship allows activation of multiple instances of the same task during a particular invocation of a program.

For a given node  $i$ , both AND and OR relationships can be intermixed together, i.e., both AND and OR operators can be used together in an expression for  $D(i)$ . When interpreting such heterogeneous combinations of several arcs, the AND relationship takes precedence over the OR relationship. For example, if  $D(i) = \{j_1 * j_2 + j_3\}$ , then node  $i$  can be activated when either both nodes  $j_1$  and  $j_2$  have been executed or when node  $j_3$  has been executed. In order to represent



more complex precedence relationships among tasks in a program, arcs terminating at the same node can be combined hierarchically by drawing ellipses around them. This is called the UNION relationship -- it is analogous to the parentheses operator in arithmetic. In an expression representing  $D(i)$ , for some node  $i$ , the UNION operator is represented by segregating the affected arcs using parentheses. The UNION relationship takes precedence over all other relationships and can be hierarchically applied. For instance, if we modify the preceding example to have  $D(i) = \{j_1 * (j_2 + j_3)\}$ , then node  $i$  can be activated when both node  $j_1$  and one of the other nodes have been executed. The rules for applying AND, OR and UNION relationships to unions of arcs are the same as those for individual arcs. A union is said to be enabled when enough arcs in that union have been enabled to permit the destination node to be activated if we ignore the condition of arcs external to the union.

*Arcs emanating* from the same node are subject to the following interpretation. If there is a weight  $w$  ( $w$  is a real number between 0 and 1) assigned to an arc's tail, then, upon completion of the task represented by that arc's source node, an operand is sent to the task represented by the destination node for the arc with probability  $w$  (i.e., the arc is *enabled* with probability  $w$ ). If  $w=1$ , then the arc is always enabled and the effect is equivalent to no weight being assigned at all, i.e.,  $w=1$  is a superfluous designation. If  $w=0$ , then the arc is never enabled and its occurrence in the graph is redundant, i.e., arcs with weights equal to zero should not be present in the graph. If there are no symbols between tails of several arcs emanating from the same node, then those arcs are joined by the (implicit) AND relationship. For node  $i$ , this relationship is formally expressed as:

$$E(i) = \{j_1 | w_1 * j_2 | w_2 * \dots * j_n | w_n\},$$

where  $j_1, j_2, \dots, j_n$  are the destination nodes for the arcs emanating from node  $i$  and  $w_1, w_2, \dots, w_n$  are their respective weights. In the case of arcs joined by the AND relationship, the decision as to whether or not to enable an individual arc is based on

that arc's weight alone, i.e., any combination of such arcs can be enabled. If there is a "+" symbol between the tail of each one of several arcs, then, upon completion of the task corresponding to the source node, exactly one of those arcs is enabled. This is called the EXCLUSIVE-OR relationship. The probability that a particular arc is enabled is equal to that arc's weight. Thus, the sum of weights of all arcs joined by the same EXCLUSIVE-OR relationship must equal to one. For node  $i$ , this relationship is formally expressed as:

$$E(i) = \{j_1|w_1 + j_2|w_2 + \dots j_n|w_n\} \text{ with } w_1 + w_2 + \dots + w_n = 1 \ .$$

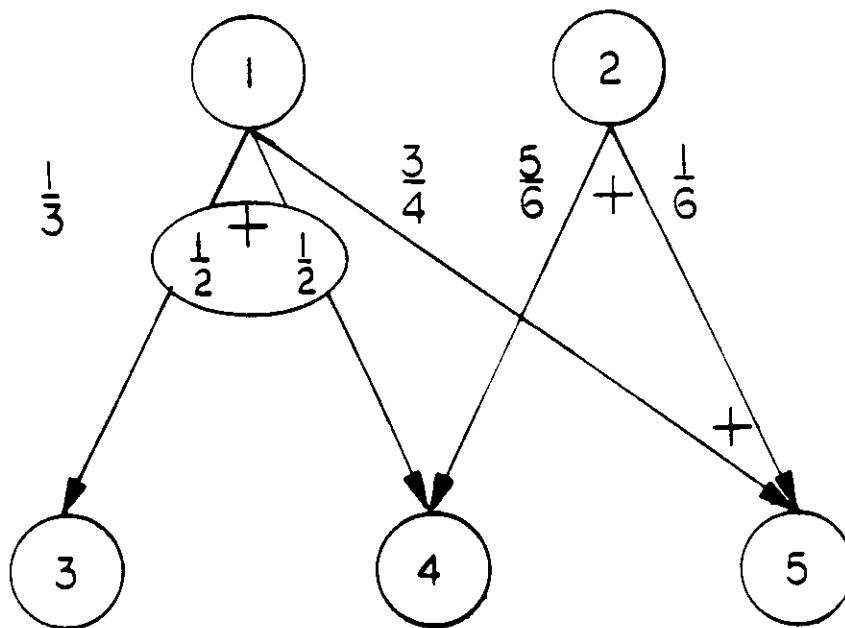
Both AND and EXCLUSIVE-OR relationships can be applied together, with the EXCLUSIVE-OR relationship taking precedence over the AND relationship. (This contrasts with the corresponding rule in the case of arcs terminating at the same node.) For example, if, for some node  $i$ ,  $E(i) = \{j_1|w_1 * j_2|w_2 + j_3|w_3\}$ , then, after node  $i$  is executed, the arc terminating at node  $j_1$  is enabled with probability  $w_1$  and either the arc terminating at node  $j_2$  or the arc terminating at node  $j_3$  is enabled, respective probabilities being  $w_2$  and  $w_3$ . Arcs emanating from the same node can also be combined hierarchically, by applying the UNION relationship (denoted by drawing ellipses around them in a graph), to represent even more complex precedence relationships. In an expression representing  $E(i)$ , for some node  $i$ , the UNION operator is represented by segregating the affected arcs using parentheses. The UNION relationship takes precedence over the AND and EXCLUSIVE-OR relationships and can be hierarchically applied. For instance, if we modify the preceding example to have  $E(i) = \{(j_1 * j_2)|w_2 + j_3|w_3\}$ , then, after node  $i$  is executed, either the arcs terminating at nodes  $j_1$  and  $j_2$  are both enabled or the arc terminating at node  $j_3$  is enabled, respective probabilities being  $w_2$  and  $w_3$ . The rules for assigning weights and for applying AND, EXCLUSIVE-OR, and UNION relationships to unions of arcs are the same as those for individual arcs.

We allow both cycles and loops in computation control graphs. A cycle occurs when, starting with a given node, a directed path can be traced back to that same node. A loop occurs when a particular arc both originates and terminates at the same node. However, we do not allow duplications, i.e., two distinct arcs cannot have both a common originating node and a common terminating node, since duplications increase the complexity of a graph without increasing its modeling power.

As an illustration of the rules defined above, let us consider the computation control graph pictured in Figure 3.3. Upon completion of node 1, with probability 3/4, the arc terminating at node 5 is enabled; with probability 1/3, either the arc terminating at node 3 or the arc terminating at node 4 is enabled, the particular arc being chosen at random. The arc between nodes 1 and 3 and the arc between nodes 1 and 4 are joined by the UNION relationship at the source node. Upon completion of node 2, an arc terminating at either node 4 or node 5 is enabled, the respective probabilities being 5/6 and 1/6. Node 3 can be activated only after node 1 is completed. Both nodes 1 and 2 must be completed before node 4 is activated. Node 5 can be activated after either node 1 or node 2 has been executed. The formal expressions describing the relationships between nodes in this graph are given below:

$$\begin{array}{ll}
 E(1) = \{3|1/2 + 4|1/2\}|1/3 * 5|3/4\} ; & D(1) = \{\emptyset\} \\
 E(2) = \{4|5/6 + 5|1/6\} ; & D(2) = \{\emptyset\} \\
 E(3) = \{\emptyset\} ; & D(3) = \{1\} \\
 E(4) = \{\emptyset\} ; & D(4) = \{1 * 2\} \\
 E(5) = \{\emptyset\} ; & D(4) = \{1 + 2\}
 \end{array}$$

We conclude this section by saying that computation control graphs constitute a very powerful and compact technique for pictorially representing program behavior, which is applicable to modeling a large variety of complex programming constructs and process scheduling algorithms. For example, the different types of



**Figure 3.3 Example of a Computation Control Graph**

programming constructs discussed in Chapter 2 can all be easily modeled with these graphs. One of the few types of program behavior which cannot be modeled, in some fashion, by computation control graphs is the conditional branching in the flow of a program which is dependent on the current state of its execution environment.

### 3.2.2 Classes of Tasks

In order to account for varying demands on the system's physical resources and differences in routing behavior, we can separate tasks into groups, where members of each group have similar processing and communication requirements. The number of groups to use is a subjective decision and depends on the degree of variability in the behavior of tasks, the level of solution accuracy desired, and the bounds on the computational cost of the solution process. The accuracy vs. computational efficiency tradeoff can be summarized by saying that, the more groups are used, the "closer" is the model to the actual system, and the larger is the time required to compute a solution. Each distinct group of tasks is modeled by a different chain of queueing network customers, in both *P/C* and *M/U* subnetworks. For each customer chain, the set of attributes of its customers, such as the service time distribution function at each service center and routing probabilities, should be chosen with the objective of "most closely" approximating the behavior of members of the corresponding task group in the actual system.

In the case of *static* task allocation policy, each task group should be further divided into smaller groups (subgroups), according to how system resources were pre-allocated to each task in the original group. This means that we must split each customer chain in our model into "subchains," with each subchain of customers corresponding to a specific subgroup of the group represented by the original chain. Each subchain of a particular customer chain would have the same service time distributions as the other subchains of that chain, but different routing probabilities in

the physical domain model. Again, the more subclasses are used, the greater will be the computational complexity of the solution of that model.

### 3.2.3 Segmentation: Hierarchical Perspective

We define a *segment* to be a set of nodes in a computation control graph which has the following properties. Those nodes in a segment which are destinations for arcs originating from some nodes external to the segment (i.e., the *internal* nodes enabled by some *external* nodes) are always all enabled simultaneously. In other words, after some set of nodes in a segment is enabled by external stimuli, the execution of the nodes inside that segment proceeds without any interaction with the nodes *outside* of the segment. However, all of the precedence relationships among the nodes inside the segment are obeyed. The arcs, which originate from the nodes internal to the segment and terminate at the nodes external to the segment, can be enabled only after *all* of the nodes in the segment have completed execution (i.e., the enabling of external nodes by internal nodes is always done simultaneously). The latter criterion requires that the completion of execution of nodes in a segment be somehow synchronized. In general, this requirement can only be satisfied by having exactly one node in a segment (perhaps representing a “dummy” task with zero processing and communication demands) which is the source for arcs terminating outside of the segment. Two distinct segments of the same computation control graph cannot have any nodes in common; the segments are connected only by arcs in the graph. A whole computation control graph itself constitutes a single segment. Segments can be viewed as “supernodes” in a “higher level” computation control graph. The rules for interpreting the interconnections of such “supernodes” are the same as those for “ordinary” nodes.

Using the definition of a segment given above, a program, as represented by a computation control graph, can be divided into segments usually in more than one

way. For a given program, the sizes of segments in a particular division are *inversely* related to the number of segments in that division. Segments usually represent some high level functionality of a program, such as procedures, functions, DO loops, etc. Alternatively, we can think of a program as being constructed out of segments linked by precedence relationships. For instance, each of the common programming constructs discussed in Chapter 2, can be represented by a proper combination of segments. It is important to note that each programming construct is itself a segment, which can be used as a constituent of any of the constructs discussed. Thus, this process of combining segments can be hierarchically applied to form very complicated program structures, while using only a few simple types of constructs. As already mentioned, a program itself constitutes one large segment and can be combined with other segments (including other programs) to form more complex programs. As will be shown in Chapter 5, the decomposition of programs into segments, as discussed here, presents a framework for applying structured analysis techniques and allows for a number of optimizations and heuristic approaches to be utilized.

### 3.2.4 Examples of Computation Control Graphs

We will use the algorithms described in section 2.4.3 to illustrate how programs can be represented by computation control graphs. The SOBEL operator is modeled by the graph depicted in Figure 3.4(a). Each node in this graph is labeled with the symbol of the arithmetic operation the corresponding task performs on the operand(s) supplied by the predecessor task(s). The structure of this graph is well suited for segmentation.

The computation control graph shown in Figure 3.4(b) represents the Column Sweep algorithm. The node labeled  $L(J, I)$  represents the task which computes:

$$L(J, I) * X(I) ; I=1, \dots, N-1 ; J=I+1, \dots, N$$

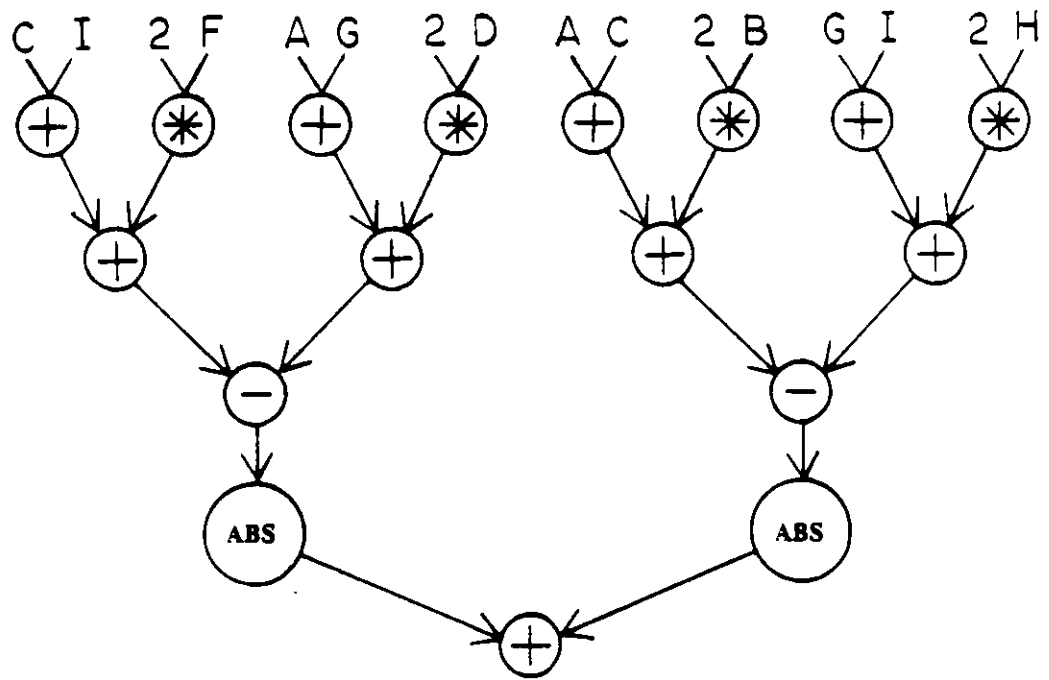
The task corresponding to the node labeled  $X(J, I)$  performs the computation of:

$$X(J) - L(J, J) * X(I) ; I=1, 2, \dots, N-1 ; J=I+1, \dots, N$$

This graph cannot be represented as a hierarchical combination of segments.

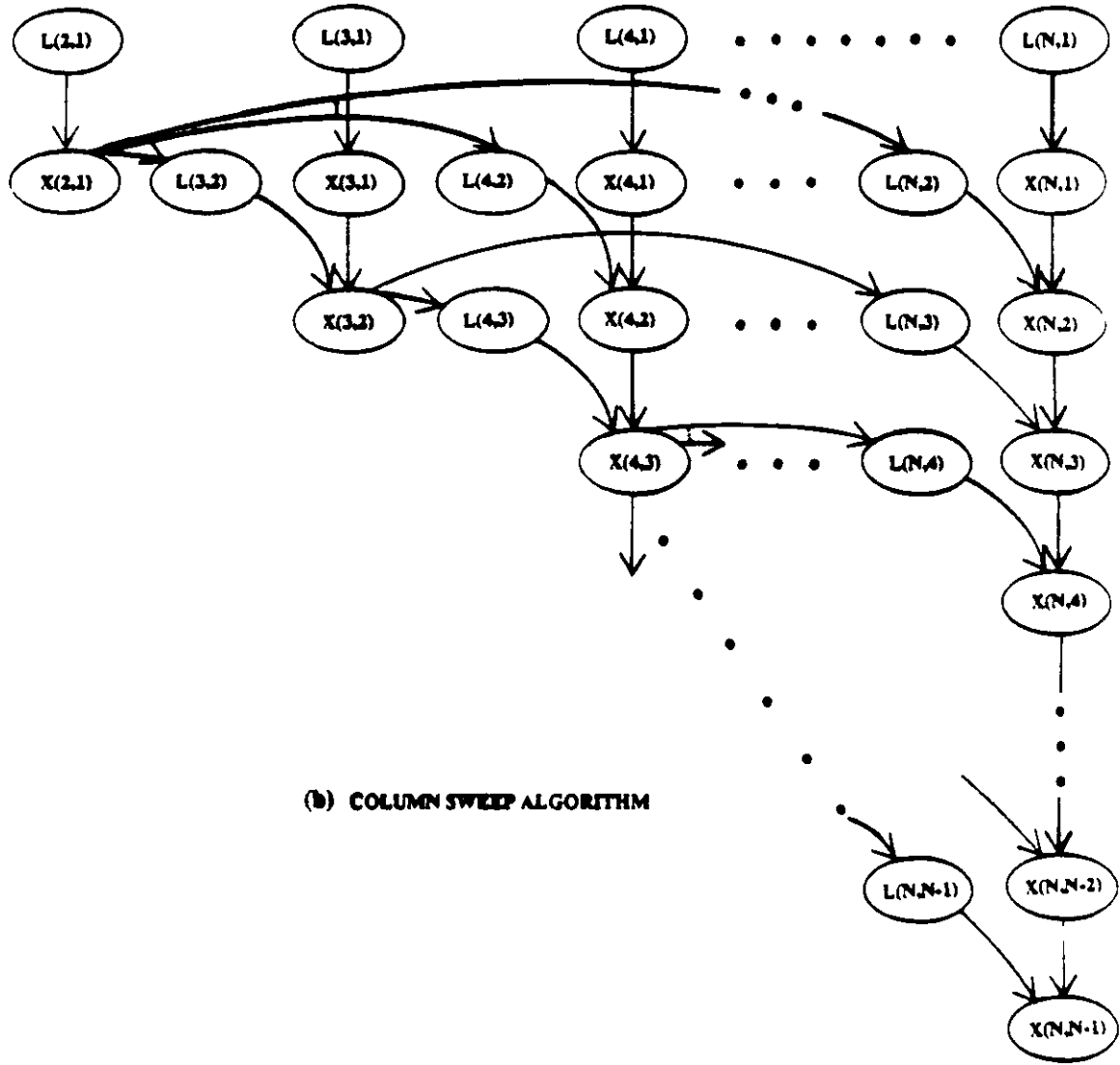
The last graph, illustrated in Figure 3.4(c), models the recursive implementation of the *Quicksort* procedure. Each node is labeled with the description of the operation performed by the corresponding task. The upper portion of the graph represents the initialization of the procedure. The nodes between those labeled “*check i*” and “ $i \leftarrow i+1$ ” inclusive, correspond to the DO loop section. The recursive procedure calls and return of results to the caller are represented by the lower portion of the graph. This computation control graph exemplifies all of the precedence relationships discussed in Section 3.2.1.



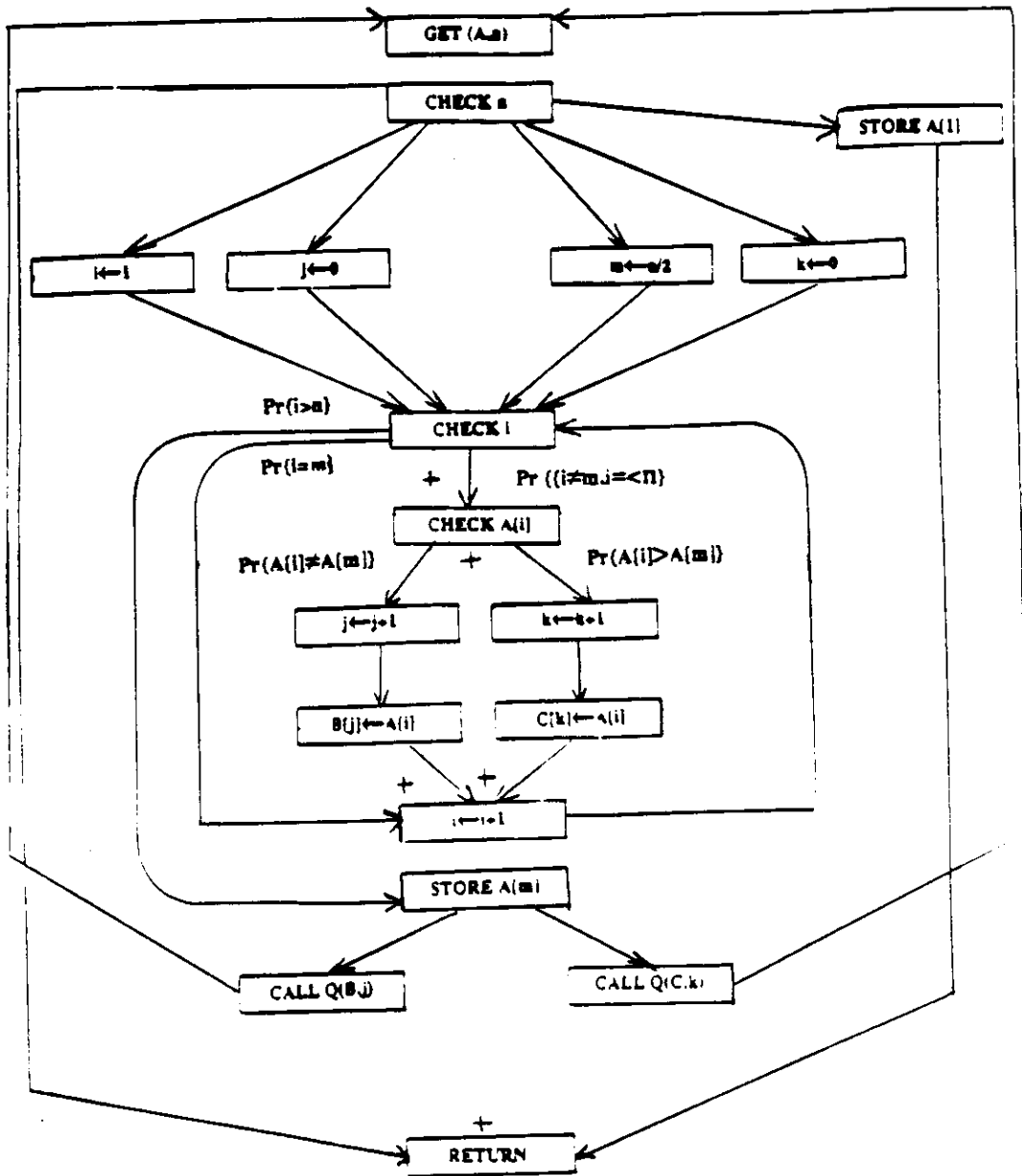


(a) SOBEL OPERATOR

Figure 3.4 Examples of Computation Control Graphs



(b) COLUMN SWEEP ALGORITHM



(e) QUICKSORT PROCEDURE

## CHAPTER 4

### APPROXIMATE ANALYTIC SOLUTION

In this chapter, we will present an analytic technique for approximately solving our model (described in the previous chapter) of program execution in a distributed, multiple-computer environment. Our starting point (i.e., input to the solution process) consists of the description of the program domain model (i.e., a computation control graph, the behavior and attributes of each task, allocation of tasks to system resources, and where synchronization of tasks is performed) and the description of the physical domain model (i.e., queueing network representations of the *P/C* and *M/U* subnetworks) where the program in question is to be executed. Having obtained this information, our methodology proceeds to:

1. solve the physical domain model to find system throughputs of various customer chains for different states of the system;
2. construct a *Markov process* whose state space consists of relevant states of program execution;
3. approximate state transition rates from the system throughputs; and, finally,
4. solve the Markov process to obtain an estimate of the average program execution time in the environment being considered.

In order to make it easier to explain our solution procedure and to illustrate its application to solving an actual model, we will first describe an example of a realistic, distributed system and present the associated physical domain and program domain models. We will then discuss, in detail, each individual step of our solution

process and illustrate it on the example model described below.

## 4.1 Example of a Model

In this section we present a model of a reduced version of the “Dataflow Multiprocessor for Continuous System Simulation” [CHA84, ERC84]. After discussing the architecture and operation of this system, we will describe its physical domain model, and then give a computation control graph of an example program to be executed in this environment.

This example will be referenced throughout the remainder of this chapter.

### 4.1.1 System Description

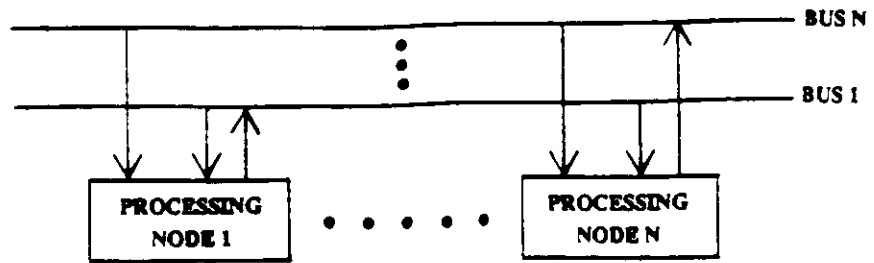
This section provides a brief overview of the system’s functional characteristics. For a more detailed description, consult the references given above.

The system consists of  $n$  identically organized processing nodes and  $n$  communication buses. Each node transmits data and control packets to the other nodes using its own, dedicated communication bus. Each node is able to receive packets from any of the  $n$  buses. The structure of this communication subnetwork allows for each result packet to be transmitted only once, no matter how many nodes it is destined to. The global view of the system’s architecture is shown in Figure 4.1(a).

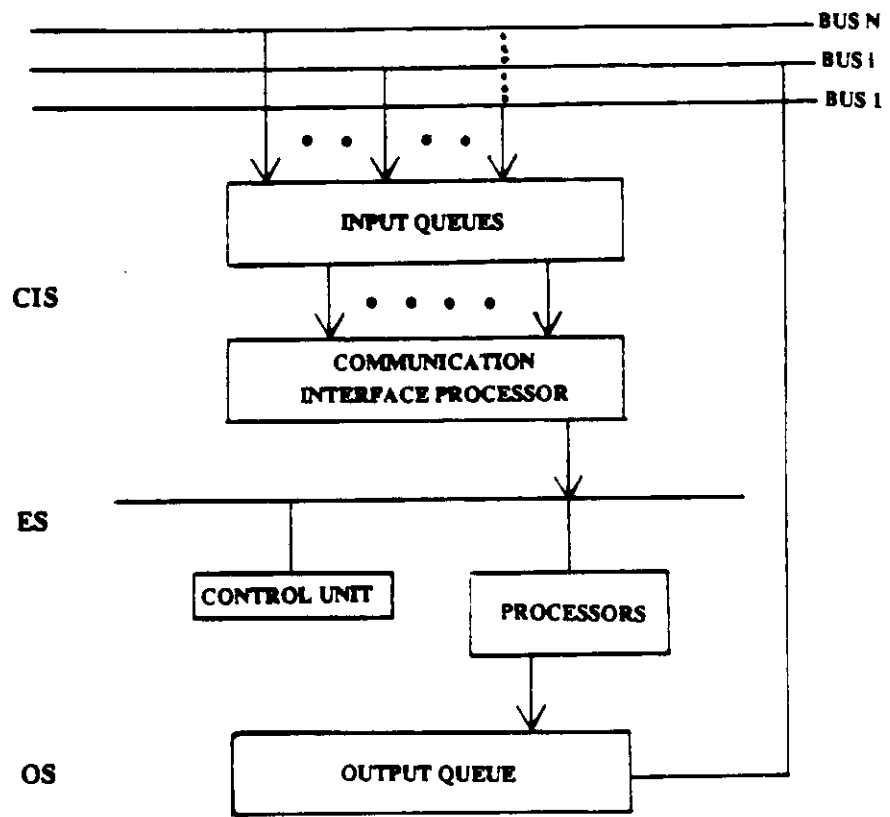
Each processing node  $i$  comprises the following functional components:

1. Communication Interface Section (CIS)
2. Execution Section (ES)
3. Output Section (OS)

Figure 4.1(b) shows how these components are interconnected with each other.



(a) GLOBAL SYSTEM ARCHITECTURE



(b) PROCESSING NODE I

Figure 4.1 Dataflow Multiprocessor

The Communication Interface Section of node  $i$  consists of  $n$  input queues, one for each of the  $n$  buses, and the Communication Interface Processor (CIP). Operand packets received from bus  $k$  are temporarily stored in Input Queue  $k$ . The CIP is responsible for polling input queues, examining each received operand packet, and, based on its contents, updating the incomplete operand sets stored in its local memory. If an operand set becomes complete, the Execution Section is notified.

The Execution Section of node  $i$  consists of the Control Unit (CU) and  $p_i$  processors. The CU receives completed operand sets from the CIP and forwards the corresponding tasks to one of the available processors for execution. Each processor executes tasks submitted to it by the CU, generates result packets for completed tasks, and forwards them to the Output Section.

The Output Section of node  $i$  consists of a single output queue, where result packets that are waiting to be *broadcast* over bus  $i$  are temporarily stored. Each result packet is received by each one of the  $n$  processing nodes.

The following rules govern the execution of programs in the environment described above. Each task of a program is pre-allocated to some processing node before commencing program execution. The information necessary to synchronize a task's execution with the rest of a program is stored at the processing node to which that task was allocated. Thus, each processing node has, in its local memory, the machine language code for the tasks assigned to it and sufficient information to determine when each of its tasks can be executed. Furthermore, a given node has no knowledge of where the other tasks are located. This requires each result packet to be broadcast to every node in the system.

#### 4.1.2 Physical Domain Model

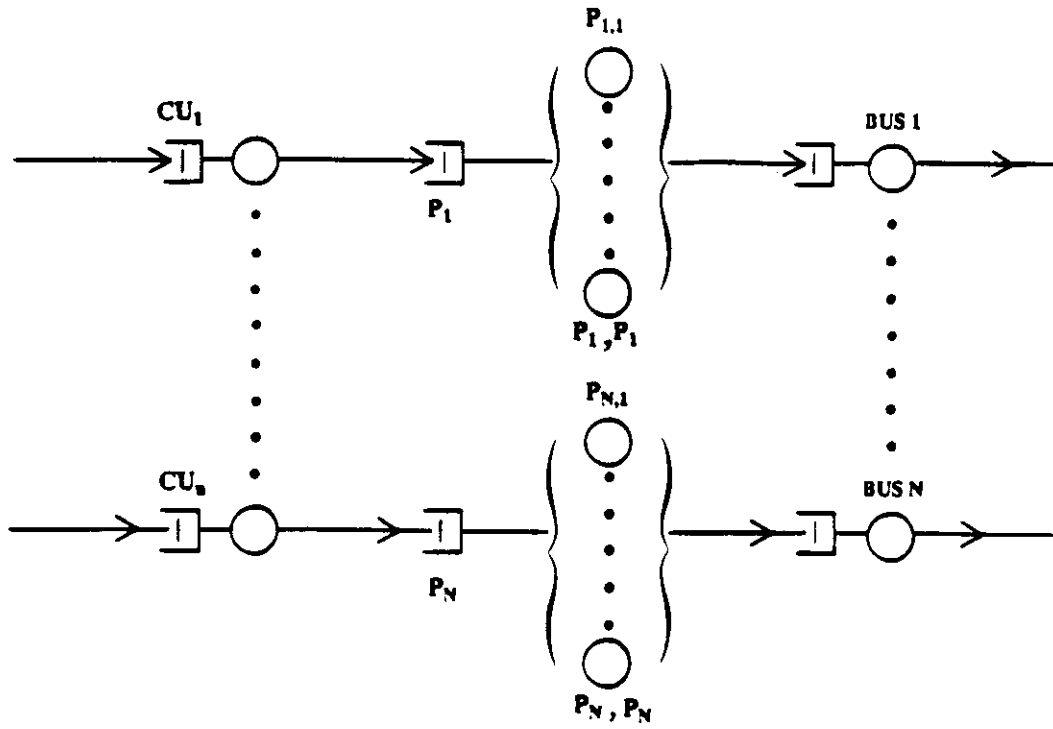
The distributed architecture presented in the preceding section falls into the “Static Allocation with Distributed Synchronization” category. Thus, we need to use both  $P/C$  and  $M/U$  subnetworks in its physical domain model.

The  $P/C$  subnetwork is depicted in Figure 4.2(a). This subnetwork consists of  $n$  disjoint and identically structured “chains” of service centers. The queueing network customers visiting the  $i$ -th chain belong to class  $T_i$ , since they represent tasks assigned to processing node  $i$  and result packets broadcast over bus  $i$ . The  $CU_i$  and  $P_i$  service centers together model the queueing and processing delays at the Execution Section of node  $i$ .  $P_i$  is a multiple-server center with  $p_i$  servers. The  $BUS_i$  service center represents both the queueing delay at the Output Section of processing node  $i$  and the transmission time on communication bus  $i$ . All service centers utilize the First-Come-First-Serve (FCFS) scheduling policy.

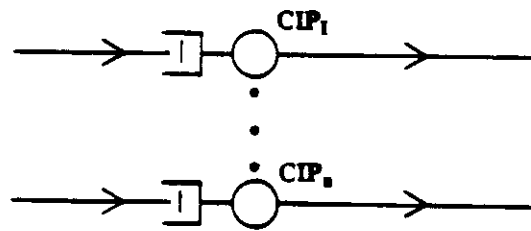
Figure 4.2(b) illustrates the  $M/U$  subnetwork. It also consists of  $n$  disjoint chains, each containing one service center. The queueing network customers visiting the  $i$ th chain belong to class  $O_i$ , since they represent operand packets destined for processing node  $i$ . The  $CIP_i$  service center represents the Communication Interface Section of node  $i$  -- it models the queueing delay at the input queues and the processing time of the Communication Interface Processor. This service center also adheres to the FCFS queueing discipline.

If we are willing to sacrifice some accuracy in order to reduce the complexity of the solution process, we can approximate this physical domain model by one representing centralized synchronization, as described in Section 3.1.3. In this approximation, there would be no class  $O_i$  customers,  $i=1,\dots,n$ . Instead, after visiting service center  $BUS_i$ , each customer of class  $T_i$  would also visit some of the  $CIP_j$  service centers. In order to obtain an upper bound on the program’s execution time,





(a) P/C SUBNETWORK



(b) M/U SUBNETWORK

**Figure 4.2 Physical Domain Model**

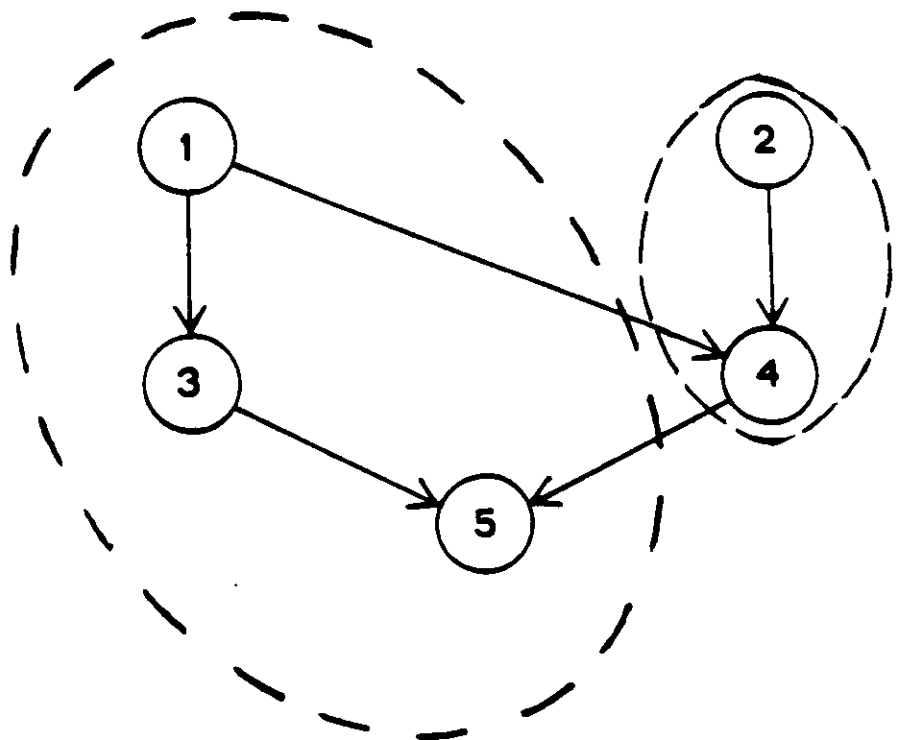
each  $CIP_k$ , where node  $k$  is a recipient of the corresponding result packet, would be visited. To obtain a lower bound, it would visit only one of those service centers -- the one with the smallest utilization. As stated in Section 3.1.3, depending on the particular values of model parameters, these bounds may potentially be very loose.

#### 4.1.3 Program Domain Model

The computation control graph of the program chosen for this example is shown in Figure 4.3. Even though this graph has a rather simple structure, it can, nevertheless, be used to demonstrate all of the pertinent features of our solution process, without obscuring the presentation by unnecessary complexity.

The configuration of the system for executing this program will consist of  $n=2$  processing nodes. Tasks 1, 3, and 5 will be allocated to node 1. Tasks 2 and 4 will be allocated to node 2. The intent of this allocation is to fully exploit the potential parallelism inherent to the program. The processing and communication requirements of tasks are exponentially distributed, with each task having the same distribution means. Thus, the queueing network customers in the  $P/C$  subnetwork representing tasks 1, 3 and 5, along with the associated result packets, belong to class  $T_1$ ; those representing tasks 2 and 4 belong to class  $T_2$ . The customers in the  $M/U$  subnetwork representing operand packets for tasks 3 and 5 belong to class  $O_1$ ; those representing operand packets for task 4 belong to class  $O_2$ .

The solution to this example will be presented as illustrations of the different steps of our solution procedure, which will be described in the remainder of this chapter.



**Figure 4.3 Program Domain Model**

## 4.2 Decomposition Approximation

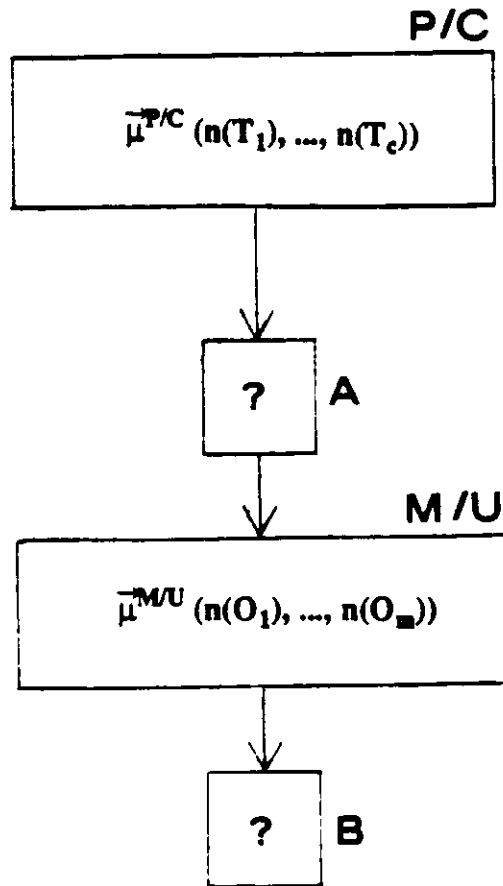
Since our objective is to estimate the average program execution time, we are only interested in the global, steady state throughput rates of the  $P/C$  and the  $M/U$  subnetworks for each customer chain, not in the individual throughput rates of each service center. Also, if we were to keep state information for each individual service center, the size and complexity of the Markov process representing a program's behavior (as defined in the next section) would become unmanageable for large physical domain models. It would also require more time and effort to be spent on evaluating the execution of different programs running in the same physical environment, thus increasing the overall computational complexity of our solution process. The reasons given above motivate our decision to use the approximation method described below. The objective of this method is to make the solution procedure "faster" by representing the behavior of the physical domain model in a more compact form.

The approximation procedure presented here is based on the notions inherent to the *Norton's Theorem* for decomposing closed queueing network models [LAV82]. The major premise of this theorem states that, if a part of a closed queueing network is replaced by a state-dependent, exponential server with properly chosen service rates and routing transition probabilities into the remainder of the network, then the newly formed network is equivalent, in terms of global performance measures (e.g., the average total throughput and the average system response time), to the original network [COU77, LAV82, VAN78]. The service rates are found by constructing a new, smaller closed queueing network from the part of the original network that is being aggregated into one server. This new network is formed by changing all routing transitions to service centers external to the subnetwork into transitions to internal service centers, the particular service centers being determined from the relative frequencies of routing transitions in the original network. We then

compute system throughputs of this network under all possible population mixes and use these values for service rates of the aggregate server. The steady state probabilities of transitions between the aggregate server and the remainder of the network are also determined from the relative transition frequencies in the original network.

However, Norton's Theorem is applicable *exactly* to *product-form* queueing networks only. Even if both *P/C* and *M/U* subnetworks are each product-form, when considered as closed networks individually, the complete queueing network representing a physical domain model is, in general (except for some very trivial cases), not product-form. Thus, in applying Norton's Theorem to our "extended" (i.e., augmented with the Black Box construct) queueing network, we are approximating the behavior of the physical domain model.

Figure 4.4 shows the new physical domain model, which was obtained from the original model (shown in Figure 3.1) by applying the approximation technique presented above. In this new, "reduced" model, *P/C* and *M/U* are state-dependent, exponential service centers.  $n(T_i)$  is the number of chain  $T_i$  customers present at the *P/C* service center -- those are the customers which represent *tasks*, along with the associated result packets, belonging to *group i* (how tasks of a given program are grouped is determined from the task allocation policy and behavioral properties of tasks).  $n(O_i)$  is the number of chain  $O_i$  customers present at the *M/U* service center -- those are the customers which represent *operand packets* destined for *memory module i* of the synchronization subsystem. Furthermore, we require that each service center employ the Processor-Sharing (PS) scheduling policy within each customer class. This is equivalent to assuming that, when a customer of some class departs from the *P/C* service center, it is equally likely to be any one of that class. The reason for making this constraint is that, with the PS queueing discipline (segregated between different classes) and an exponential service time distribution for each customer class, we eliminate the correlation between the order in which



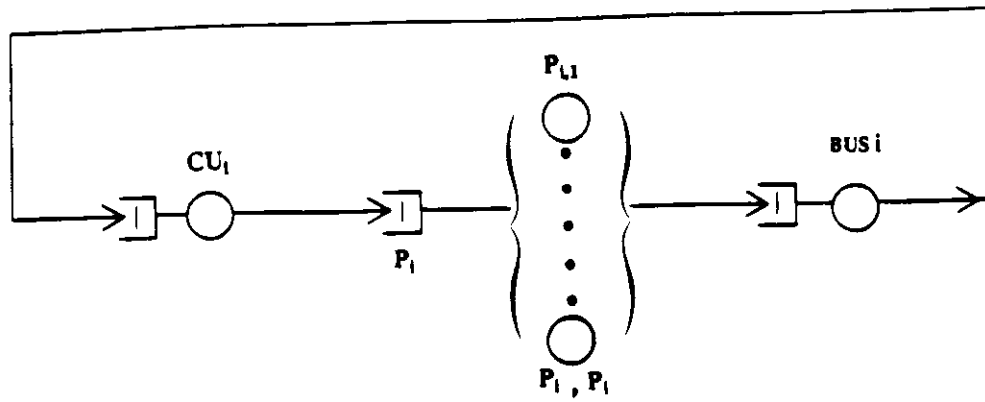
**Figure 4.4 Physical Domain Model after Applying Decomposition Approximation**

customers of the same chain enter the  $P/C$  service center and the order in which they depart. Thus, we need to keep track of only the number of customers of each chain at a service center, not the order in which those customers are queued.

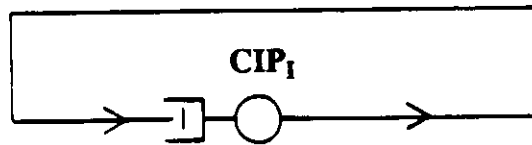
At a given time instant, the rate of departure from the  $P/C$  service center of a chain  $T_i$  customer is a function of the current values of  $n(T_j)$ , for all allowable  $j$ . For a given set of values, the departure rate is found by considering the  $P/C$  subnetwork, in isolation, as a closed queueing network, having  $n(T_j)$  customers of chain  $T_j$ , for all  $j$ , and solving for the throughput of chain  $T_i$ . In order to find the complete range of the service rate function for the  $P/C$  service center, we need to perform this operation with each possible customer population vector. The service rate function for the  $M/U$  service center (i.e., the set of departure rates for chain  $O_i$  customers, for all allowable  $i$ ) is computed analogously.

As an illustration of the procedure described above, let us consider the example presented in Section 4.1. The departure rate of a chain  $T_i$  customer, with  $n$  customers of chain  $T_i$  present at the  $P/C$  service center, is equal to the throughput of the closed queueing network shown in Figure 4.5(a) with population set to  $n$ . The departure rate of a chain  $O_i$  customer, with  $n$  customers of chain  $O_i$  present at the  $M/U$  service center, is equal to the throughput of the closed queueing network depicted in Figure 4.5(b) with population set to  $n$ . Note that, in this example, the departure rate of customers of a given chain is not dependent on the number of customers of other chains present (in general, not the case).

We can use a variety of existing, *exact* and *approximate* techniques for solving multiple-class, closed queueing network models, in order to compute the throughput rates of the  $P/C$  and  $M/U$  subnetworks. These techniques range, in terms of solution accuracy and computational complexity, from the exact and very “expensive” *Mean Value Analysis* [REI80] to the very fast but approximate (only bounds are obtainable) *Asymptotic Analysis* [MUN74, ZAH82]. The choice of what



(a) CLASS  $T_i$  DEPARTURE RATES



(b) CLASS  $O_i$  DEPARTURE RATES

Figure 4.5 Solving for Departure Rates of the Example Model



particular method to use depends on the complexity (i.e., size and connectivity) of each subnetwork, the number of customer classes used, relative utilizations of servers by different classes, the level of solution accuracy required, and constraints on the computational cost of the solution process.

### 4.3 Markov Process Construction

In order to find the expected time of execution of a program in the distributed environment being modeled, we need to “track” its behavior during the time period when it is being executed. Of particular interest is the *degree of parallelism* attained by a program (i.e., the number of enabled tasks -- those which are either executing or ready to be executed) during each instant of its execution time. In order to obtain any time-dependent performance measure, we have to find a transient solution to the system being modeled. Obtaining a transient solution analytically is a formidable task and one usually has to resort to detailed simulation of the system. Thus, in order to make our solution process analytically tractable, we will settle for the *distribution of parallelism* instead, which is a steady state performance measure. In other words, we will solve for the probability that (or fraction of time when) there are exactly  $k$  enabled tasks,  $p(k)$ , for  $k=1,2,3,\dots$ , during the program’s execution time period, given that the program is repeatedly executed an infinite number of times. The distribution of parallelism for program A is formally represented by the following expression:

$$p_A(k) , \quad k = 1, 2, 3, \dots \quad (4.1)$$

If there are different chains of queueing network customers in the corresponding physical domain model, then  $k$  becomes a vector, instead of a scalar. For instance, if  $c$  customer chains are used, then expression (4.1) becomes:

$$p_A(k_1, \dots, k_c) , \quad k_1, \dots, k_c = 1, 2, 3, \dots , \quad (4.2)$$

where  $k_i$  is the number of enabled tasks represented by customers of chain

$i, i=1, 2, \dots, c$ . After obtaining the distribution of parallelism, the average task throughput of the system, while executing the program being modeled,  $\mu$ , is found by summing over system throughputs under all possible mixes of customer chains, multiplied by the respective probabilities. For program A, the average task throughput,  $\mu_A$ , is given by:

$$\mu_A = \sum_{k_1=0}^{\infty} \cdots \sum_{k_c=0}^{\infty} \mu(k_1, \dots, k_c) \cdot p_A(k_1, \dots, k_c) \quad , \quad (4.3)$$

where  $\mu(k_1, \dots, k_c)$  is the system throughput (i.e., the total departure rate from the P/C service center) with  $k_i$  customers of chain  $i$  present,  $i=1, 2, \dots, c$ . Finally, an estimate for the average program execution time,  $T$ , is computed by dividing the average number of tasks executed during each invocation of the program,  $N$ , by the average task throughput, e.g., for program A:

$$T_A = N_A / \mu_A \quad (4.4)$$

Recently, a solution approach similar to the one presented in this section was independently developed; it is described in [THO85]. That procedure is based on the concepts of Courtois' Decomposition theory, which allows the level of detail of a Markov process to be hierarchically reduced by aggregating a number of "lower-level" states into a single, "higher-level" state [COU77]. Furthermore, only *acyclic* graph models of programs are considered in [THO85].

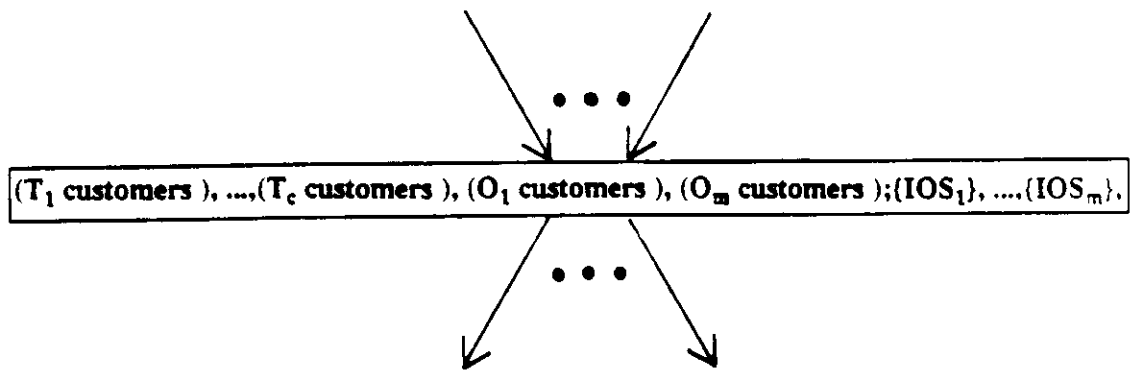
#### 4.3.1 State Space Description

To begin with, we must construct a Markov process consisting of all of the "important" states of the program's execution history. The granularity of the Markov process has to be of a level which is sufficient to allow for unambiguous, memoryless transitions between states. Each state must capture all essential information for determining all of the possible transition paths to other states and their corresponding rates. One of the key advantages of this methodology is that, for a

given program, the structure of the associated Markov process does not depend on a particular execution environment. It only depends on the *category* (as classified in Chapter 2) to which the selected physical system belongs. However, the numerical parameters of that Markov process, such as state transition rates, are functions of the specific system architecture.

In order to avoid having extremely large Markov processes with overly complex state descriptors, we will use the approximation to the physical domain model described in Section 4.1 to “compactly” represent the program’s execution environment. Figure 4.6 depicts a state descriptor for the most general category (“Static Allocation with Distributed Synchronization”) of the distributed, multiple-computer systems. The part of the descriptor on the *left* side of the “;” separator specifies all currently “active” queueing network customers, i.e., those representing enabled tasks, result packets, and operand packets. The “ $T_i$  Customers” section of this part lists the identities of those enabled tasks which are represented by customers of chain  $T_i$  (i.e., the tasks belonging to group  $i$ ), for all  $i$  -- these customers are currently being served at the  $P/C$  service center. The “ $O_i$  Customers” section of the left part specifies those operand packets that have not yet been processed and which are represented by customers of chain  $O_i$  (i.e., the operand packets destined for memory module  $i$ ), for all  $i$  -- these customers are currently being served at the  $M/U$  service center. Each operand packet is described by the operands it contains, with each operand identified by the associated task and its “operand position” number for that task. The “ $O_i$  Customers” sections are not needed when modeling systems with centralized synchronization since, in models of such systems, operand packets are integrated with result packets (as explained in Section 3.1.3).

The part of the descriptor on the *right* side of the “;” separator provides additional information (for use by the Black Boxes in the physical domain model) by describing all incomplete operand sets present in the synchronization subsystem.



**Figure 4.6** Descriptor of a State in a Markov Process

The “*Incomplete\_Operand\_Sets<sub>i</sub>*” section,  $\{IOS_i\}$ , of the right part lists all incomplete operand sets stored in memory module  $i$ , for all  $i$ . Each set is described by the identity of some currently disabled task and a list of operands received for that task so far.

A state transition occurs whenever an *active* customer completes service at either *P/C* or *M/U* service center. For a state with a non-empty “*T<sub>i</sub> Customers*” section, whenever a chain  $T_i$  customer departs from the *P/C* service center, a transition takes place into one of several possible states, as determined by the computation control graph of the program being modeled. Each possible transition corresponds to an activation of some “feasible” subset of those arcs in the graph which emanate from the node corresponding to the departing customer. We define a feasible subset of arcs, for a given node in the graph, as a set where all member arcs may be enabled simultaneously upon completion of execution of the task represented by that node. The number and constituency of such subsets is determined by the relationships operating on the aforementioned arcs (as discussed in section 3.2.1). If a given feasible subset of arcs includes a union of arcs, that union may be represented by any feasible subset of arcs within the union -- each different subset internal to the union would generate a different “global” feasible subset, while keeping arcs external to the union the same. Note that all arcs (or unions of arcs) having weights equal to one must be included in every feasible subset, since they are always enabled upon completion of execution of the source node. Also, exactly one arc or union from each set of arcs and unions joined by the same EXCLUSIVE-OR relationship must be included in every feasible subset. The probability of enabling a particular feasible subset (i.e., making the corresponding state transition) is a function of how the arcs in that subset are grouped and of their respective weights. The procedure for computing these probabilities will be presented in the following section. For a particular state transition, the descriptor of the destination state is found by: (1) deleting the identity of the departing customer from the “*T<sub>i</sub> Customers*” section of the

descriptor of the current state; and (2) for each memory module  $j$  in the synchronization subsystem, adding one customer to the " $O_j$  Customers" section if an operand packet destined for that memory module was generated by enabling the corresponding subset of arcs.

For a state with a non-empty " $O_i$  Customers" section, whenever a chain  $O_i$  customer departs from the  $M/U$  service center, a transition takes place from the current state into a state identified by the following descriptor. Starting with the descriptor of the current state, we perform the following steps: (1) delete the identity of the departing customer from the " $O_i$  Customers" section of the left part; and (2) update the incomplete operand sets listed in the  $\{IOS_i\}$  section of the right part, using the information contained in the operand packet corresponding to the customer in question. Any operand set which becomes completed (after  $\{IOS_i\}$  is updated) is immediately deleted from the  $\{IOS_i\}$  section and a customer, corresponding to the task that became enabled, is added to the " $T_j$  Customers" section of the state descriptor, where  $j$  is the group to which the enabled task belongs.

The *initial* state of the Markov process is described by listing those queueing network customers which represent the tasks enabled at the start of the program. We define a *final* state of the Markov process as a state which contains no active customers (i.e., the part of its descriptor to the left of ";" is empty), since no transition is possible out of such a state. Note that, depending on the underlying computation control graph, there may be several final states. However, if a program is well specified (i.e., the corresponding graph is well behaved), then there is only one final state -- the "empty" state (i.e., no extraneous operands are left after the program completes execution). If there is no final state, then the process either has an infinite state space or constitutes one single cycle, i.e., the program never terminates and its execution time is infinite. In order to be able to compute the steady state probabilities, when constructing the Markov process, we convert all transitions to a final state

into transitions back to the initial state. In fact, the corresponding Markov process represents a continuously repeated execution of the same program. This interpretation correlates our methodology with both measurement and simulation approaches, where a program (or a simulation of it) is run a number of times and its expected execution time is estimated by averaging over execution times of individual runs.

In order to illustrate the application of the concepts described above, we will again recall our example model. The Markov process for the computation control graph given in that example is depicted in Figure 4.7(a). We have shown all possible state transition paths but have not included the corresponding rates -- computation of transition rates will be discussed in Section 4.3.3. Each active customer corresponding to an operand packet is represented in a state descriptor as:

$$[t_1 | o_1, \dots, t_k | o_k]$$

where  $k$  is the number of operands contained in that packet and  $t_j | o_j$  is operand number  $o_j$  for task  $t_j$ . The incomplete operand set for task  $t$  is represented in a state descriptor as:

$$\{t | o_1, \dots, o_k\}$$

where  $k$  is the number of operands for task  $t$  received so far and each  $o_j$  is the position number of one of the received operands. A detailed procedure for constructing Markov processes from computation control graphs is given in the following section.

If the original physical domain model were to be approximated by one with centralized synchronization, as discussed in Section 4.1.2, the Markov process for the aforementioned example could be considerably simplified, as illustrated in Figure 4.7(b).

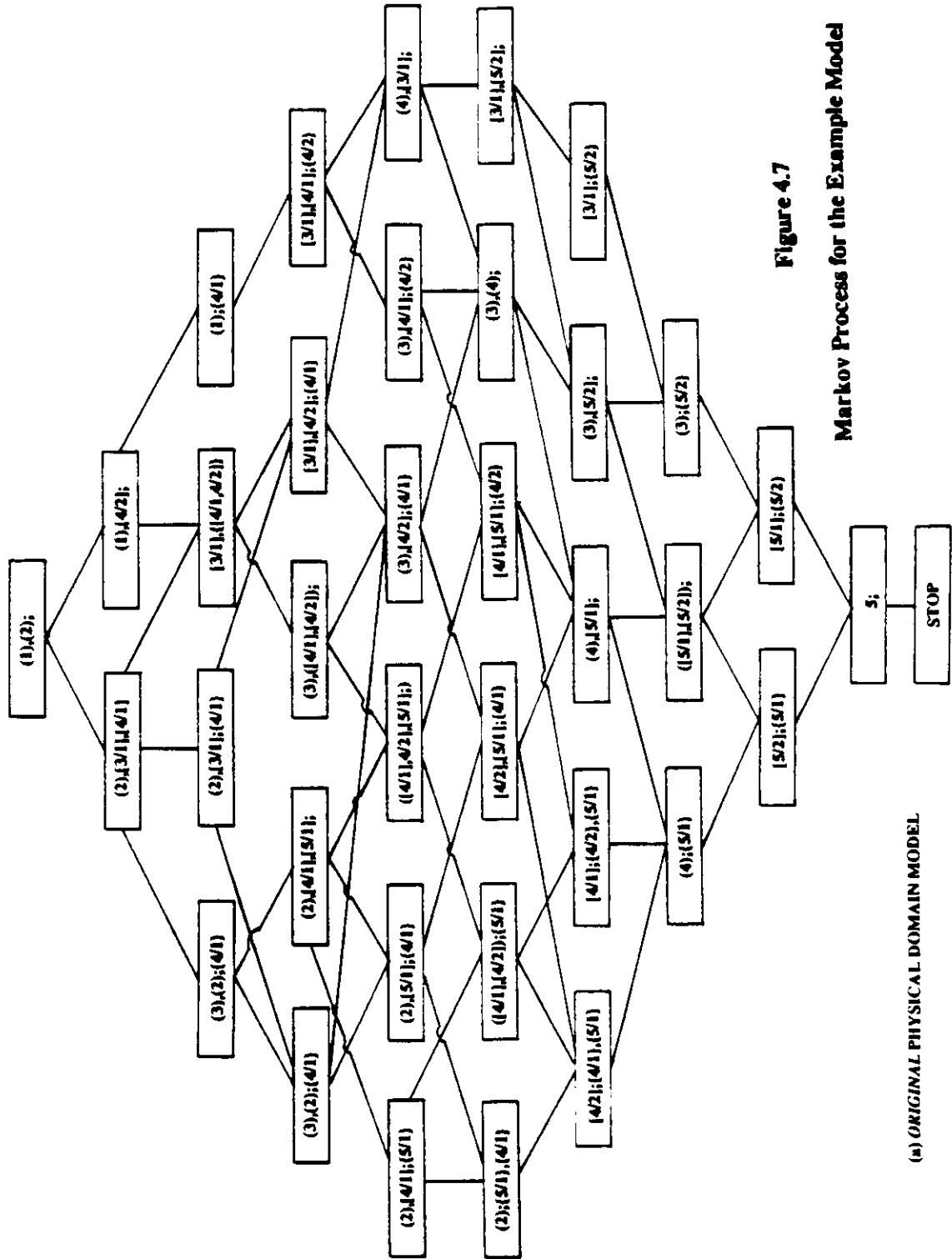
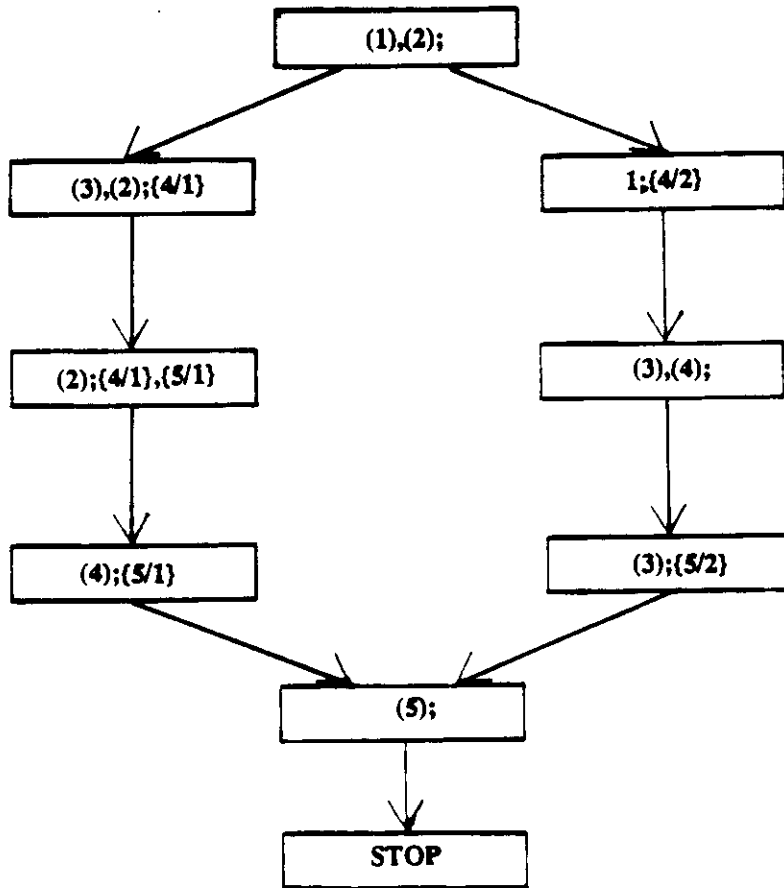


Figure 4.7  
Markov Process for the Example Model

(a) ORIGINAL PHYSICAL DOMAIN MODEL





**(b) APPROXIMATED PHYSICAL DOMAIN MODEL**

### **4.3.2 Generation of a Markov Process from a Computation Control Graph**

As can be seen from the previous section, the construction of Markov processes is procedurally well defined and can be easily automated and implemented as a program. One of the main reasons for choosing computation control graphs (to represent programs or transactions in our methodology) is that the generation of Markov processes from such program representation lends itself well to automation. In this section, we will present a detailed algorithm for generating a Markov process from a computation control graph, when modeling systems of the “Centralized Synchronization” categories. This algorithm can be easily extended to be used in solving systems of other categories. We will also discuss ways to reduce the size of the state space of generated processes.

#### **4.3.2.1 Markov Process Generation Algorithm**

This section describes an algorithm for generating a Markov process from a computation control graph. The Markov process is used to model the execution of the program represented by the given graph. This algorithm assumes that, in the system being considered, there is a single, central facility for task synchronization. That is, each queueing network customer is assumed to model a task, its result packet, and the associated operand packet. In this case, departure of a customer represents completion of processing of the corresponding operand packet. Thus, operand packets are not explicitly represented in a state descriptor (i.e., “ $O_i$  Customers” sections are not included). Therefore, the algorithm given here is primarily applicable to modeling systems belonging to the “Centralized Synchronization” categories. Although, it can also be used to model approximations to systems of “Distributed Synchronization” categories (see Section 3.1.3).

Before presenting our algorithm, we need to make some definitions. First, recall that we define a “feasible” arc activation, for a particular node in a computation control graph, to be a subset of arcs and unions of arcs, *emanating* from that node, which may be *concurrently* enabled (a detailed definition was given in Section 4.3.1). Note that arcs (unions) joined through the EXCLUSIVE-OR relationship (a “+” symbol in the graph), either directly or indirectly (i.e., through the UNION relationship), cannot belong to the same feasible subset of arcs. Let  $S$  be a subset of arcs and unions representing a feasible arc activation. Each member of  $S$  is either an arc, which is external to any union, or a union of arcs, which is external to any other union. If union  $i$  is a member of  $S$ , then  $i(S)$  is defined as the particular feasible subset of arcs and unions internal to union  $i$ , which is represented in  $S$ . We also define another set of arcs and unions,  $S'$ , to be a maximal superset of  $S$ , which also represents a feasible arc activation for the source node. Let  $T(S)$  be a set of arcs and unions given by  $S'-S$ . Informally,  $T(S)$  can be expressed as:

$$T(S) = \{j \mid j \text{ is an “outermost” arc or union, emanating from the same node as members of } S; j \text{ is not a member of } S; j \text{ is not joined by the EXCLUSIVE-OR relationship with any member of } S\}. \quad (4.5)$$

The probability of the feasible arc activation represented by  $S$  occurring,  $f(S)$ , is equal to the probability that each member of  $S$  will be enabled and none of the members of  $T(S)$  will be enabled. This probability is given by the following recursive expression:

$$f(S) = \prod_{i \in S} x_i \prod_{j \in T(S)} (1-w_j) \quad (4.6)$$

where  $w_i$  is the weight on arc (union)  $i$  and

$$x_i = \begin{cases} w_i, & \text{if } i \text{ is an arc} \\ w_i \times f(i(S)), & \text{if } i \text{ is a union of arcs} \end{cases} \quad (4.7)$$

The probability of the feasible arc activation represented by  $i(S)$  occurring,  $f(i(S))$ ,

can be computed in the same fashion as  $f(S)$  in Equation (4.6).

The notation used in describing the mechanics of our algorithm will now be explained. The labels used during the execution of the algorithm have the following meaning. The state which is currently being “worked on” is marked CURRENT. A state for which all possible transitions have already been found is marked PROCESSED. In the descriptor of the state marked CURRENT, each task label, such that all possible transitions due to a departure of the corresponding queueing network customer have been found, is marked EXPANDED. The task label which is currently being “worked on” is marked ACTIVE. For the ACTIVE task label, each feasible arc activation, for which a state transition has already been created, is marked DONE. The feasible arc activation which is currently being processed is marked NEW.

There is a variable associated with each task label  $t$ , denoted  $INST(t)$ .  $INST(t)$  is the currently existing number of instances of the task labeled  $t$ , i.e., there are tasks labeled  $t_1, t_2, \dots, t_{INST(t)}$ . Different instances of a task are represented by hierarchically appending instance numbers to the original (root) label of that task. For example, the label  $t_{i,j}$  represents the  $j$ -th instance of the  $i$ -th instance of the task having label  $t$ . The  $INST$  variable is also associated with each instance of a task, e.g.,  $INST(t_{i,j})$ .

The complete algorithm is presented below.

0. Create the descriptor for the initial state and initialize  $INST$  variables:
  - a. For each task  $t$  enabled at the start of the program, add label  $t_1$  to the proper “ $T_i$  Customers” section of the state descriptor and set

$$INST(t) = 1.$$

b. For each task  $t'$  disabled at the start of the program, set  $INST(t') = 0$ .

1. Find a state which is not marked PROCESSED.

If none found, go to (4). Otherwise, mark the chosen state CURRENT.

2. Find a task label in the descriptor of the state marked CURRENT, but not marked EXPANDED.

If none found, change the marking of the CURRENT state to PROCESSED, and go to (1).

Otherwise, mark the chosen task label ACTIVE.

Make a list of all feasible arc activations for the ACTIVE task.

3. Find a feasible arc activation for the ACTIVE task which is not marked DONE.

If none found, change the marking of the ACTIVE task to EXPANDED, and go to (2).

Otherwise, mark the chosen feasible arc activation NEW, and perform the following steps:

a. Make a copy of the descriptor of the CURRENT state.

b. Delete the ACTIVE task label from the descriptor copy.

c. Add operands generated by the NEW feasible arc activation to the proper incomplete operand sets in the *right* part of the copy of the CURRENT state descriptor. (Each operand is suffixed by the instance number(s) of the ACTIVE task label.)

- d. For each task  $t$  which became enabled (i.e., its operand set was completed) after step (c), erase the corresponding incomplete operand set from the descriptor copy, increment  $INST(t)$ , and add label  $t_{INST(t)}$  to the proper " $T_i$  Customers" section of the descriptor copy.
  - e. If the left part of the created descriptor is empty, then the proper state transition for the NEW feasible arc activation is to the *initial* state; go to (g).
  - f. Compare the created descriptor with all of the *existing* state descriptors. If a match is found, erase the created descriptor -- the desired state already exists. Otherwise, create a new state which is labeled by the created descriptor.
  - g. Create a transition to the *proper* (either newly created or already existing) state from the CURRENT state and label it with the probability of the NEW feasible arc activation occurring.
  - h. Change the marking of the NEW feasible arc activation to DONE and go to start of (3).
4. STOP; the Markov process has been generated.

#### 4.3.2.2 State Space Reduction

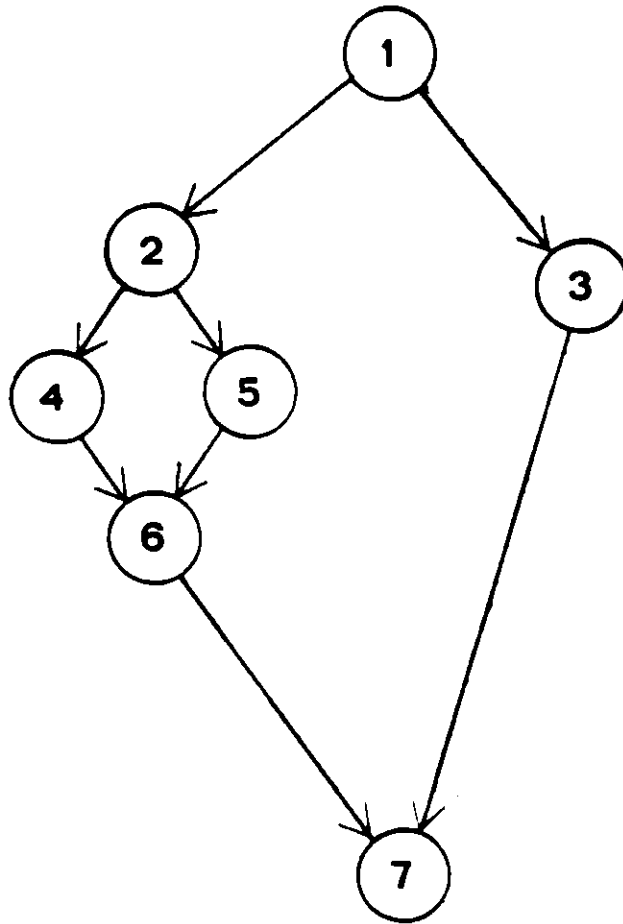
In order to reduce the total number of states in a Markov process, we can combine two or more "equivalent" states into a single, *aggregate* state. We define two different states to be equivalent if they have identical transitions with identical rates to other states and if their respective descriptors have the same number of queuing network customers in each " $T_i$  Customers" section and in each " $O_i$  Cus-

tomers'' section. The descriptor of the aggregate state is formed by joining the individual descriptors of its constituents through the ''/'' symbol(s). That is, if ''D1'' and ''D2'' are the respective descriptors of two ''equivalent'' states, say  $S_1$  and  $S_2$ , then the descriptor of the aggregate state is given by ''D1/D2.'' This state space reduction process can significantly reduce the computational cost of solving the Markov process to obtain state probabilities. Figures 4.8 and 4.9 show a computation control graph and its ''optimized'' Markov process (in the case of a system belonging to the ''Dynamic Allocation with Centralized Synchronization'' category), respectively. Note that all tasks in this example belong to the same group.

If we are willing to sacrifice some accuracy, we can further improve the efficiency of our solution process by aggregating states that are not equivalent but yet ''similar.'' The criteria for determining whether or not ''similarity'' exists between two states are subjective ones and depend, among other things, on the properties of the particular environment and the particular program being modeled. We will, however, attempt to give some heuristic guidelines to assist a modeler in making such a determination. Note that the following are suggestions only, as there is no sufficient empirical evidence to support them. Thus, no claim as to the extent of their applicability can be made.

For any two states  $S_1$  and  $S_2$ , let  $e(S_1, S_2)$  be the maximum (in absolute value) of the relative differences, expressed as percentiles, between the transition rates out of  $S_1$  and the corresponding transition rates out of  $S_2$ . Let  $M$ , expressed as a percentile, be the minimum of the following quantities:

- a. the accuracy of the parameters for the physical domain model (e.g., server capacities and routing probabilities);
- b. the accuracy of the parameters for the program domain model (e.g., mean task service times and weights on arcs in the computation control graph);



**Figure 4.8** Computation Control Graph of an Example Program



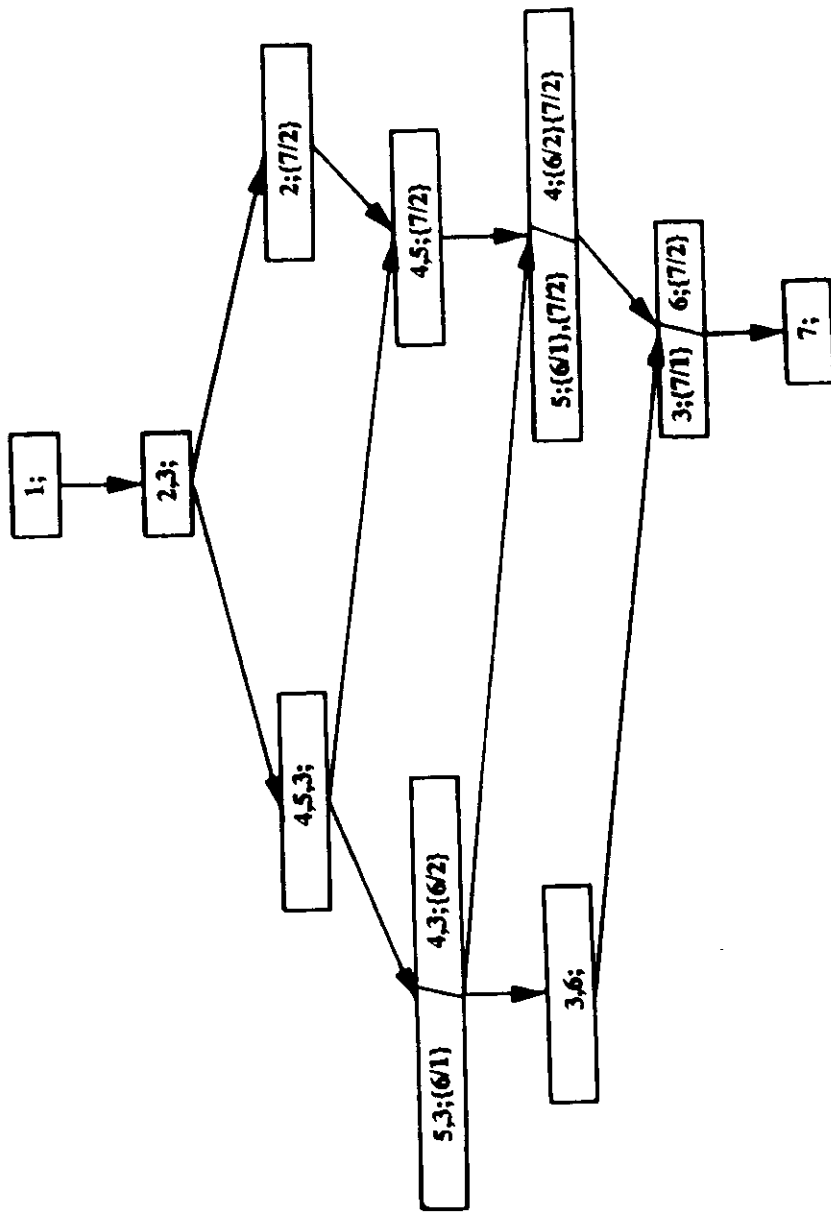


Figure 4.9 Markov Process for the Example Program

- c. the accuracy level of the method used to solve the Markov process; and
- d. the desired accuracy of the final result (i.e., the average execution time of the program being modeled).

If  $e(S_1, S_2) < 100 - M$ , then, in most cases, states  $S_1$  and  $S_2$  may be aggregated without significantly decreasing the accuracy of the final solution. Even if  $e(S_1, S_2) > 100 - M$ , these states may still be combined if their steady state probabilities are “much smaller” than those of other states. Note, however, that there are pathological cases where small perturbations of state transition rates can significantly alter the final solution [COU77].

Let state  $S$  be the aggregate of states  $S_1$  and  $S_2$ . For each state  $S_i$  in the Markov process, the rate of transition from  $S$  to  $S_i$ ,  $r(S, S_i)$ , should be an interpolation of the respective rates for  $S_1$  and  $S_2$ ,  $r(S_1, S_i)$  and  $r(S_2, S_i)$ . Formally, this can be expressed as:

$$r(S, S_i) = f \cdot r(S_1, S_i) + (1 - f) \cdot r(S_2, S_i) \quad (4.8)$$

where  $0 < f < 1$ . The value of  $f$  in Equation (4.8) should be chosen to reflect the “anticipated” relative difference between the values of the respective steady state probabilities for states  $S_1$  and  $S_2$ .

### 4.3.3 State Transition Rates

In the preceding sections, we described how to create a Markov process for modeling program execution, starting with a computation control graph and a description of each task, including obtaining the topology of the chain representing that process and probabilities of choosing particular transition paths. However, we did not discuss how to determine the rate of making a particular state transition.

In order to compute state transition rates, let us again refer to the “reduced” physical domain model presented in Section 4.2. In that model, both  $P/C$  and  $M/U$  service centers have exponential, state-dependent service rates and use the Processor-Sharing queueing discipline. As stated in that section, the departure rate of a chain  $T_i$  customer from the  $P/C$  service center, with  $n(T_j)$  customers of chain  $T_j$  present, for all possible  $j$ , is set equal to the throughput of customer chain  $T_i$  from the  $P/C$  subnetwork, having  $n(T_j)$  customers of chain  $T_j$ , for all  $j$ . We will express this throughput as:

$$\mu_{T_i} (n(T_1), \dots, n(T_c)) \quad (4.9)$$

where  $c$  is the number of different customer chains in the  $P/C$  subnetwork. Due to the fact that all customers of the same chain have identical service demands and routing probabilities, the rate of a *particular* chain  $T_i$  customer departing is equal to the expression (4.9) divided by  $n(T_i)$ :

$$\mu_{T_i} (n(T_1), \dots, n(T_c)) / n(T_i) \quad (4.10)$$

The departure rate of a particular chain  $O_i$  customer from the  $M/U$  service center is computed analogously and is expressed as:

$$\mu_{O_i} (n(O_1), \dots, n(O_m)) / n(O_i) \quad (4.11)$$

where  $m$  is the number of different customer chains in the  $M/U$  subnetwork (i.e., the number of distinct memory modules in the synchronization subsystem).

For a given state of a Markov process, the rate of leaving that state due to a departure of a particular chain  $T_i$  customer is computed from expression (4.10), with  $n(T_j)$  set to the number of customers listed in the “ $T_j$  Customers” section of that state’s descriptor, for all  $j$ . The rate of making a particular state transition, associated with the departure of that customer, is equal to the probability of making that transition (i.e., the probability of enabling the corresponding feasible subset of arcs, as discussed in Section 4.3.2.1) multiplied by the rate given above. There is only

one possible state transition associated with a departure of a particular chain  $O_i$  customer. Its rate is given by expression (4.11), with  $n(O_j)$  set to the number of customers listed in the “ $O_j$  Customers” section of that state’s descriptor, for all  $j$ .

Figure 4.10 shows one of the states of the Markov process depicted in Figure 4.7. It gives state transition rates for all possible transitions out of that state.

#### 4.4 Solving Markov Processes

After constructing the Markov process and obtaining, for each state, the transition paths and rates to other states, we now have to solve for the steady-state probability of each state, in order to find the distribution of parallelism of the program and compute an estimate of the expected program execution time. Various solution approaches and possible optimizations are discussed below.

##### 4.4.1 Exact vs. Approximate Techniques

There are many different *exact* and *approximate* methods available for solving Markov processes. The choice of what particular method to use depends on the structure of the process, the state transition rates, and the accuracy level desired. The solution of a Markov process is equivalent to solving a linear system of  $N$  equations,  $N$  being the number of states in a process, where variables are states and coefficients are transition rates. An exact, but computationally expensive, approach is to compute an inverse of the  $N \times N$  matrix of transition rates. However, this process is impractical for large processes as its computational complexity is  $O(N^3)$ . An important observation is that, for most programs, the Markov process will have a very *sparse* state transition rate matrix. That is, on the average, the number of transitions with non-zero rates from a particular state will be a small percentage of the total number of states. There are many techniques which take advantage of this pro-

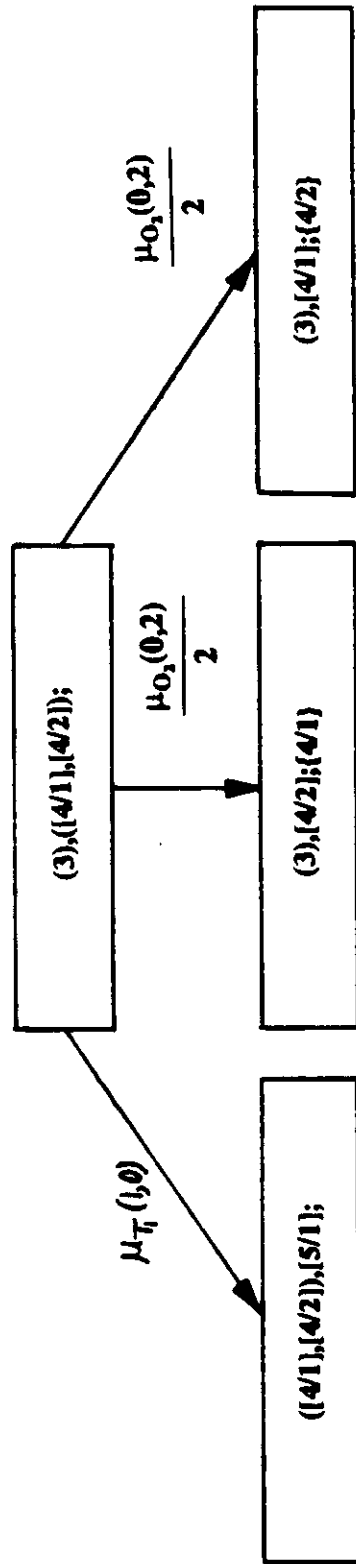


Figure 4.10 Example of State Transition Rates in a Markov Process

perty to significantly reduce the computational and storage requirements of the solution process.

For programs which exhibit the locality of reference property, computation control graphs can be viewed as consisting of several regions, where program execution “loops” for a while in a given region before moving on to other regions. The state space of Markov processes generated from such graphs can usually be divided into sets, where transition rates between states within each set are much greater than rates of transitions to “external” states. The decomposition method proposed by Courtois [COU77] would give a good tradeoff between speed and accuracy requirements, when applied to such Markov processes. In this method, the total state space is divided into a number of groups, where states in each group “interact” more frequently with each other than with states in other groups. Each group is then solved individually as a closed system and replaced by a single aggregate state. The global interactions among groups are then found by solving the Markov process consisting of the aggregate states. Finally, the solution of each group is combined with the global solution to produce estimates of state probabilities in the original process.

#### 4.4.2 Acyclic Processes and Symbolic Solution

If, after the transitions between final states and the initial state are eliminated, a Markov process is *acyclic*, that is, for each state, there is no directed path back to that state, then its transition rate matrix has a *triangular* structure and the solution time is proportional to the number of arcs in the process. Thus, acyclic processes (i.e., those having topologies of trees) require considerably less computation to solve than cyclic processes. Furthermore, an acyclic process can be conveniently solved symbolically -- that is the solution for state probabilities can be represented symbolically in terms of transition rates, by using distinct symbols for distinct rates. We will call such solutions *algebraic*. If a Markov process is solved algebraically, then

we can evaluate it with different sets of transition rates by substituting corresponding numerical values for symbols. Depending on the number of states and *distinct* transition rates in the associated Markov process, the latter approach may significantly reduce the computational cost of evaluating the execution of a particular program on many different systems.

As an illustration of the potential benefit of algebraic solutions, let us again recall the example model given in Section 4.1. Suppose, for a given application, we want to minimize its execution time by varying the number of processors, the capacity of each processor, and the bandwidth of the communication bus used by node  $i$ , for all  $i$ . However, we are constrained by the requirement that the total system cost must remain under a certain upper limit. Let  $C_1, \dots, C_k$  be a set of all feasible system configurations meeting this cost constraint. For each  $C_i$ ,  $i=1, \dots, k$ , the solution of the corresponding physical domain model produces a different set of customer throughput rates, which in turn yields a different set of state transition rates, without, however, changing the topology of the Markov process for the application in question. If the solution for the mean execution time of that application were given in terms of an algebraic equation, we would have been able to quickly evaluate  $C_i$ ,  $i=1, \dots, k$ , by substituting symbols with numerical values from the corresponding set of transition rates.

It is important to note that acyclic computation control graphs always generate acyclic Markov processes and that acyclic Markov processes can only be generated from acyclic graphs. In the next chapter, we will show that some cyclic graphs can be made acyclic by extracting cycles, solving each cycle separately, and then integrating individual solutions back into the original graph.

## CHAPTER 5

### OPTIMIZATIONS AND HEURISTICS

In this chapter, we will describe heuristic procedures for obtaining approximate solutions to some of the common programming constructs in a computationally efficient way. We assume that each constituent (program segment) of a given programming construct has been completely solved (i.e., we have its mean execution time, average number of tasks executed of each class, and *distribution of parallelism*) and can be replaced by a compact segment descriptor (defined in Section 5.1). We will show how the key properties of a construct (e.g., average execution time and distribution of parallelism) can be estimated from the compact segment descriptors of its constituents.

For a particular physical system, supporting a total of  $c$  classes of tasks, the distribution of parallelism for a program segment  $S$  is given by the following discrete function of  $c$ -dimensional vectors of non-negative integers:

$$p_S(k_1, \dots, k_c) \quad , \quad k_i \text{ is a non-negative integer, } i=1, \dots, c \quad (5.1)$$

The range of this function is the set of real numbers between 0 and 1 (inclusive) and the value of  $p_S(k_1, \dots, k_c)$  is the probability (or fraction of time) that, while segment  $S$  is being executed alone, there are exactly  $k_i$  enabled tasks of class  $i$ ,  $i=1, \dots, c$ . Since, during every instant of a segment's execution time period, there is at least one enabled task,  $p_S(0, \dots, 0) = 0$  for any  $S$ . If the computation control graph representing segment  $S$  is acyclic, then the distribution of parallelism for  $S$  has a finite number of nonzero members. Note that the same program segment will, in general, have different distributions of parallelism with different system architectures.



After defining a compact description of program segments, we will present techniques for approximately solving several programming constructs, discuss hierarchical application of those techniques and compare tradeoffs between gain in computational efficiency and loss in accuracy. In the following development, we will restrict our attention primarily to systems of the ‘‘Centralized Synchronization’’ categories.

### 5.1 Segments: Compact Description

A program segment, as defined in Section 3.2.3, can be solved as a stand-alone program, using the procedure developed in the previous chapter, to obtain its mean execution time, average number of tasks executed, and distribution of parallelism. Thus, for a given segment  $S$ , we can ‘‘ignore’’ its underlying computation control graph and describe it by the following triple:

$$\{N_S, T_S, p_S(k_1, \dots, k_c)\} \quad (5.2)$$

where  $T_S$  is its mean execution time,  $p_S(k_1, \dots, k_c)$  is its distribution of parallelism, and  $c$  is the number of *all* task classes supported by the system. Note that the tasks of an individual segment may only represent a subset of those classes. If  $c=1$ , then  $N_S$  is the average number of tasks executed during an invocation of segment  $S$ . If  $c > 1$ , then  $N_S$  is the vector  $(N_{S,1}, \dots, N_{S,c})$ , where  $N_{S,i}$  is the average number of tasks of class  $i$  executed during an invocation of  $S$ . This compact description captures only the ‘‘steady-state’’ properties of a segment and ignores its time-dependent behavior. This is analogous to making ‘‘fluid flow’’ approximations when modeling queueing systems [KLE76].

How accurate a representation of a segment’s behavior this description is depends primarily on how much the segment’s degree of parallelism varies during its execution time. (The reader is reminded that the degree of parallelism is a time-

dependent performance measure.) In other words, the larger the differences between degrees of parallelism during different time intervals, the less accurate is our representation. We can make our description more accurate (but less compact) by representing segment  $S$  as a sequential combination of  $n$  segments,  $S_1, S_2, \dots, S_n$ , where the degree of parallelism of  $S_j$ , for all  $j$ , is more uniform than the degree of parallelism of  $S$ . This representation is shown in Figure 5.1. The new, more detailed description of segment  $S$  is given by the  $n$ -tuple

$$[d(S_1), d(S_2), \dots, d(S_n)] , \quad (5.3)$$

where, for all  $j$ ,

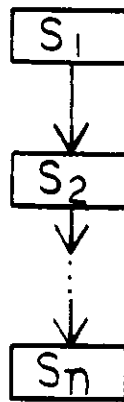
$$d(S_j) = \{ N_{S_j}, T_{S_j}, p_{S_j}(k_1, \dots, k_c) \} . \quad (5.4)$$

## 5.2 Sequential Combination

First we will consider a *sequential* combination,  $S$ , of two segments, segment  $A$  and segment  $B$ . The mean execution time of this combination,  $T_S$ , is equal to  $T_A + T_B$ , where  $T_A$  and  $T_B$  are the mean execution times of segments  $A$  and  $B$ , respectively. Let the distributions of parallelism for segments  $A$  and  $B$  be given by  $p_A(k_1, \dots, k_c)$ ,  $k_i \geq 0$ ,  $i=1, \dots, c$ , and  $p_B(k_1, \dots, k_c)$ ,  $k_i \geq 0$ ,  $i=1, \dots, c$ , respectively. Then the distribution of parallelism for the combination is  $p_S(k_1, \dots, k_c)$ ,  $k_i \geq 0$ ,  $i=1, \dots, c$ , where:

$$p_S(k_1, \dots, k_c) = \frac{[p_A(k_1, \dots, k_c) \cdot T_A + p_B(k_1, \dots, k_c) \cdot T_B]}{T_S} \quad (5.5)$$

The average number of tasks executed during an invocation of  $S$  is  $N_S = N_A + N_B$ . ( $N_S$ ,  $N_A$  and  $N_B$  are either scalars or vectors, depending on  $c$ .) Thus, we can replace  $S$  by a single segment, having description  $\{ N_S, T_S, p_S(k_1, \dots, k_c) \}$ . Note that, as discussed in the previous section, the greater the difference in distribution of parallelism between segments  $A$  and  $B$ , the less accurate is this representation of  $S$ .



**Figure 5.1 More Accurate Representation of Segment S**

A sequential combination of several segments can be solved in an analogous fashion. Note that, in this algorithm, the sequential combination of segments  $A$  and  $B$  is equivalent to the combination of  $B$  and  $A$ . Also, the combination of the combination of  $A$  and  $B$  with segment  $C$  is equivalent to the combination of  $A$  with the combination of  $B$  and  $C$ . Thus, our algorithm has both commutative and associative properties, which allows the combination of several segments to be solved in any order.

### 5.3 EXCLUSIVE-OR Combination

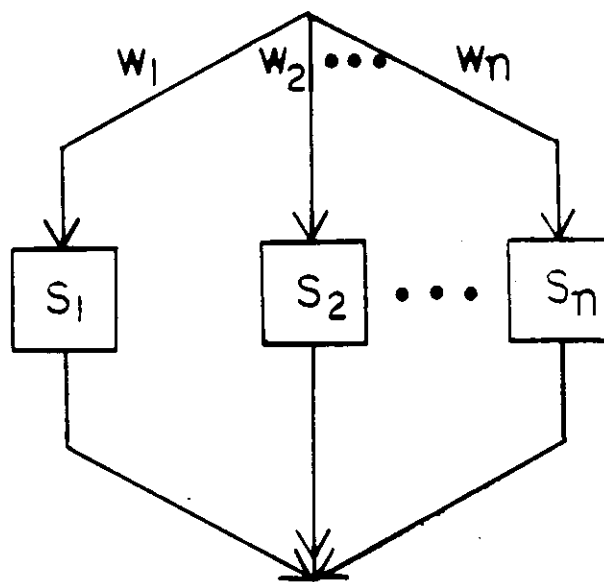
Next we will study an EXCLUSIVE-OR combination,  $E$ , of segments  $S_1, S_2, \dots, S_n$ , as shown in Figure 5.2. In this combination, exactly one out of  $n$  possible segments is executed during each invocation of  $E$ , with segment  $S_j$  being executed with probability  $w_j$ ,  $j=1, \dots, n$  (the sum of  $w_j$ 's must equal to one). In other words,  $100 \cdot w_j$  percent of the invocations of the combination  $E$  will "behave" like segment  $S_j$ ,  $j=1, \dots, n$ .

The mean execution time of  $E$  is given by:

$$T_E = \sum_{j=1}^n w_j \cdot T_{S_j} \quad (5.6)$$

where  $T_{S_j}$  is the mean execution time of segment  $S_j$ ,  $j=1, \dots, n$ . Thus, the fraction of time when combination  $E$  will "behave" like segment  $S_j$  is  $w_j \cdot T_{S_j}/T_E$ ,  $j=1, \dots, n$ . Let the distribution of parallelism for segment  $S_j$  be given by  $p_{S_j}(k_1, \dots, k_c)$ ,  $k_i \geq 0$ ,  $i=1, \dots, c$ , for all  $j$ . Then the distribution of parallelism for the combination is given by  $p_E(k_1, \dots, k_c)$ ,  $k_i \geq 0$ ,  $i=1, \dots, c$ , where:

$$p_E(k_1, \dots, k_c) = \frac{\sum_{j=1}^n w_j \cdot T_{S_j} \cdot p_{S_j}(k_1, \dots, k_c)}{T_E} \quad (5.7)$$



**Figure 5.2 EXCLUSIVE-OR Combination**

The average number of tasks executed during an invocation of  $E$  is

$$N_E = \sum_{j=1}^n w_j \cdot N_{S_j} \quad (5.8)$$

( $N_{S_1}, \dots, N_{S_n}$ , and  $N_E$  are either scalars or vectors, depending on  $c$ .) Thus, we can replace  $E$  by a single segment having description  $\{ N_E, T_E, p_E(k_1, \dots, k_c) \}$ .

## 5.4 Parallel Combination

Now we will analyze a *parallel* combination,  $P$ , of segments  $A$  and  $B$ . We will show how to estimate its mean execution time,  $T_P$  and its distribution of parallelism,  $p_P(k_1, \dots, k_c)$   $k_i \geq 0$ ,  $i=1, \dots, c$ . Note that the average number of tasks executed during an invocation of this combination,  $N_P$ , is simply equal to  $N_A + N_B$ . After obtaining all of these values, combination  $P$  can be replaced by a single segment, having description  $\{ N_P, T_P, p_P(k_1, \dots, k_c) \}$ .

### 5.4.1 Computing Mean Execution Time

Before proceeding with the derivation, we need some definitions. Let  $T_{A|B}$  be the mean time to execute segment  $A$  if segment  $B$  started to execute at exactly the same time that segment  $A$  did (on the same system of course).  $T_{B|A}$  is defined analogously. Thus,  $T_P$  is equal to either  $T_{A|B}$  or  $T_{B|A}$ , depending on which segment completes last. Without loss of generality, let us suppose that segment  $A$  is first to finish and let  $\hat{B}$  be the portion (i.e., the fraction of total number of tasks) of segment  $B$  which was completed during the time that  $A$  had been executing (i.e., during the first  $T_{A|B}$  time units of  $T_{B|A}$ ).  $\hat{B}$  may contain a nonintegral number of tasks if some tasks were only partially completed. The time to finish executing segment  $B$  is given by  $T_B - T_{\hat{B}}$ . ( $T_{\hat{B}}$  is the average time it would have taken to execute the  $\hat{B}$  portion of segment  $B$  if  $B$  were running alone.) Thus,

$$T_P = T_{B|A} = T_{A|B} + (T_B - T_{\hat{B}}) \quad (5.9)$$

In the case that segment  $B$  completes first,

$$T_P = T_{A|B} = T_{B|A} + (T_A - T_{\hat{A}}) \quad (5.10)$$

with segment  $\hat{A}$  analogously defined.

We will now show how to determine which segment completes first (on the average) and how to estimate  $T_{A|B}$ ,  $T_{B|A}$ ,  $T_{\hat{A}}$  and  $T_{\hat{B}}$  as defined above, which are necessary for obtaining the mean execution time of the parallel combination of segments  $A$  and  $B$ . The following development is applicable to solving models with only one class of tasks (i.e.,  $c=1$ ). (With  $c > 1$ , some of the “independence” and “fluid flow” assumptions necessary for this development are no longer valid.)

#### 5.4.1.1 Definition of Notation

First, we will define the notation being used in this development.  $N_A$  and  $N_B$  are the average numbers of tasks executed during invocations of segments  $A$  and  $B$ , respectively.  $\mu(k)$  is the throughput (or mean departure rate) of tasks of the distributed system being modeled, when  $k$  tasks are being executed concurrently,  $k=1,2,\dots$ . Let  $\mu_A$  and  $\mu_B$  be the *average* throughputs of tasks of segments  $A$  and  $B$ , respectively, when each is being executed *alone*:

$$\mu_A = N_A/T_A \quad (5.11)$$

$$\mu_B = N_B/T_B \quad (5.12)$$

Thus,  $\mu_A$  and  $\mu_B$  depend on both  $\mu(k)$ , for all  $k$ , and the distributions of parallelism of segments  $A$  and  $B$ . Let  $\mu'_A$  and  $\mu'_B$  be the average task throughputs of segments  $A$  and  $B$ , respectively, during the time period when both segments are executing *concurrently*.

### 5.4.1.2 Estimating Average Throughputs of Tasks

We will first present the formula for computing  $\mu'_A$  and discuss how it was derived. This formula is given by the following equation:

$$\mu'_A = \mu(i+j) \cdot \frac{i}{(i+j)} Pr(i \text{ of } A, j \text{ of } B | A\&B) \quad (5.13)$$

Each component in the summation is a product of two terms. The first term,  $\mu(i+j) \cdot i/(i+j)$ , is the throughput of *segment A* tasks, when there are  $i$  segment *A* tasks and  $j$  segment *B* tasks “competing” for system resources. This term is derived from the fact that the *P/C* subnetwork in the original physical domain model is approximated by a single state-dependent service center which adheres to the *Processor-Sharing* scheduling discipline. (Note that this expression holds true exactly if and only if the *P/C* subnetwork is equivalent to a single processor-sharing resource.) The second term represents the *joint probability* of exactly  $i$  tasks of segment *A* and exactly  $j$  tasks of segment *B* being enabled, while both segments are executing in parallel. In order to obtain the *exact* value of this term, one must construct and solve the complete Markov process for combination  $P$ , which defeats the purpose of our approximation procedure. Thus, we will attempt to estimate this joint probability using only already known values.

First, observe that:

$$Pr(i \text{ of } A, j \text{ of } B | A\&B) = Pr(i \text{ of } A | A\&B) \cdot Pr(j \text{ of } B | i \text{ of } A | A\&B) \quad (5.14)$$

Now, we will approximate  $Pr(j \text{ of } B | i \text{ of } A | A\&B)$  by  $Pr(j \text{ of } B | A\&B)$ . This approximation is motivated by the observation that the precedence relationships of tasks in one segment are *independent* of those in the other segment. However, note that  $Pr(i \text{ of } A | A\&B)$  and  $Pr(j \text{ of } B | A\&B)$  are not truly independent of each other. The current number of enabled tasks of segment *A* is correlated with the number of segment *A* tasks that have already completed execution. The same is true of segment *B*. Also, the number of segment *A* tasks that have already departed is correlate



with the number of segment  $B$  tasks that have already departed (since they all had utilized the same resources). Thus, the number of segment  $A$  tasks which are currently enabled is correlated with that of segment  $B$ .

Next, we will replace  $Pr(i \text{ of } A|A\&B)$  by  $p_A(i)$  and  $Pr(j \text{ of } B|A\&B)$  by  $p_B(j)$ . This approximation is based on the assumption that the *relative* throughputs of segment  $A$  tasks are not dependent on the number of segment  $B$  tasks present, and vice-versa. The more linear the  $\mu(k)$  function is, the more accurate the latter estimates are.

Finally, combining all of the equations presented above, we get:

$$\mu'_A = \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \mu(i+j) \cdot \frac{i}{(i+j)} p_A(i) \cdot p_B(j) \quad (5.15)$$

The formula for estimating  $\mu'_B$  can be derived in an analogous fashion and is presented below.

$$\mu'_B = \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \mu(i+j) \cdot \frac{j}{(i+j)} p_A(i) \cdot p_B(j) \quad (5.16)$$

#### 5.4.1.3 Estimations of $T_{\hat{A}}$ & $T_{\hat{B}}$

First, let us consider the case where segment  $A$  completes execution before segment  $B$  does. Since, by definition,  $\hat{A}$  is the portion of segment  $A$  completed by the time segment  $B$  completes execution,  $T_{\hat{A}}$  is equal to  $T_A$ . The average number of segment  $B$  tasks completed during the execution of segment  $A$ ,  $N_{\hat{B}}$ , is given by  $\mu'_B \cdot T_{A|B}$ . If segment  $B$  were executing alone, its average task throughput rate would be  $\mu_B$ . Thus,  $T_{\hat{B}}$  can be approximated by:

$$T_{\hat{B}} = \frac{N_{\hat{B}}}{\mu_B} = \frac{(\mu'_B \cdot T_{A|B})}{\mu_B} \quad (5.17)$$

Now, let us consider the case where segment  $B$  is first to finish. Using arguments analogous to the ones presented above, we will estimate  $T_{\hat{A}}$  by:

$$T_{\hat{A}} = \frac{N_{\hat{A}}}{\mu_A} = \frac{(\mu'_A \cdot T_{B|A})}{\mu_A} \quad (5.18)$$

#### 5.4.1.4 Computing (Average) First Complete Execution

Our first step is to determine which one of the two segments completes execution first (on the average). Toward this goal, we prove the following lemma.

**Lemma 5.1:** Under the assumptions and definitions stated above,

$$N_A/\mu'_A < N_B/\mu'_B \text{ if and only if } T_{A|B} < T_{B|A}$$

**Proof:** Suppose that  $N_A/\mu'_A < N_B/\mu'_B$  but  $T_{A|B} > T_{B|A}$ . Then the execution of segments  $A$  and  $B$  is represented by the timing diagram shown in Figure 5.3. From this diagram, we obtain the following relationship:

$$\mu'_A \cdot T_{B|A} < N_A \rightarrow T_{B|A} < N_A/\mu'_A \rightarrow N_B/\mu'_B < N_A/\mu'_A$$

Thus, we obtain a contradiction.

Conversely, assuming that  $T_{A|B} < T_{B|A}$  while  $N_A/\mu'_A > N_B/\mu'_B$ , we can analogously show that this leads to a contradiction. Q.E.D.

Thus, all we need to do in order to compute  $T_{A|B}$  and  $T_{B|A}$  is to obtain  $\mu'_A$  and  $\mu'_B$  and apply Lemma 5.1. If  $N_A/\mu'_A < N_B/\mu'_B$ , then segment  $A$  completes execution first,  $T_{A|B} = N_A/\mu'_A$ , and  $T_{B|A} = T_{A|B} + (T_B - T_{\hat{B}})$ . If  $N_A/\mu'_A > N_B/\mu'_B$ , then segment  $B$  completes execution first,  $T_{B|A} = N_B/\mu'_B$ , and  $T_{A|B} = T_{B|A} + (T_A - T_{\hat{A}})$ .

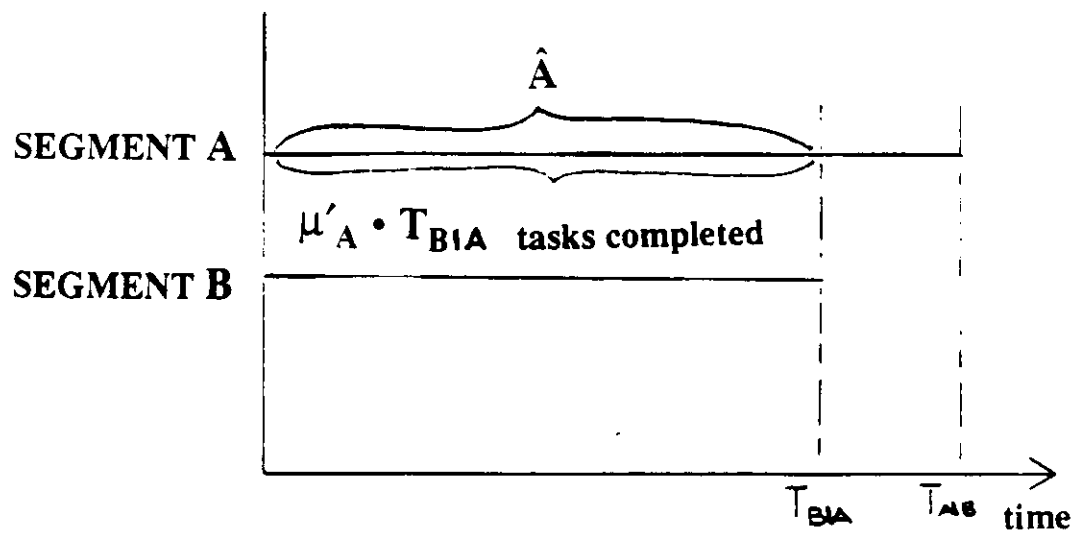


Figure 5.3 Parallel Execution of Segments A & B

It is important to note that, since Lemma 5.1 uses average values, we can only determine which one of the two segments is most probable to complete execution first. More precisely, Lemma 5.1 states that, if  $N_A/\mu'_A < N_B/\mu'_B$ , then, when combination  $P$  is executed an infinite number of times, segment  $A$  will be the first to finish in a majority of instances. Observe that the greater the difference  $[N_B/\mu'_B - N_A/\mu'_A]$  is, the greater are the number of instances where  $B$  finishes first (i.e., the greater is the probability that  $B$  completes execution before  $A$  does during a particular invocation of  $P$ ). If segments  $A$  and  $B$  have identical segment descriptors, then this difference is exactly zero -- both segments are equally likely to complete execution first. Therefore, in general, the more similar the segment descriptors of the constituents of  $P$  are, the less accurate is our solution.

#### 5.4.2 Computing the Distribution of Parallelism

In this section, we will show how to estimate the distribution of parallelism,  $p_P(k)$ ,  $k \geq 0$ , for the parallel segment combination  $P$ . Without loss of generality, we will assume that segment  $A$  is first (on the average) to complete execution. Let  $s(k)$ ,  $k \geq 0$ , be the distribution of parallelism of combination  $P$  during the first  $T_{A|B}$  time units of the execution time period of  $P$ , i.e.,  $s(k)$  is the "joint" distribution of parallelism of segments  $A$  and  $B$ . Let  $r(k)$ ,  $k \geq 0$ , be the distribution of parallelism of  $P$  during the remainder of the combination's execution time. Applying arguments similar to the ones used in section 5.4.1.2 to derive equations (5.15) and (5.16),  $s(k)$  for all values of  $k$  can be estimated by:

$$s(k) = \sum_{i=0}^k p_A(i) \cdot p_B(k-i) \quad (5.19)$$

Using the "fluid flow" assumption that the distribution of parallelism of segment  $B$  remains uniform over all subintervals of  $T_B$ ,  $r(k)$  is approximately equal to  $p_B(k)$ . Finally, it follows from Bayesian principles that, in the case that segment  $A$  finishes

first,  $p_P(k)$  can be estimated by:

$$p_P(k) = \frac{[s(k) \cdot T_{A|B} + r(k) \cdot (T_P - T_{A|B})]}{T_P} \quad \text{for all } k \quad (5.20)$$

From the symmetry of the preceding derivation, in the case that segment  $B$  finishes first,  $p_P(k)$  can be estimated by:

$$p_P(k) = \frac{[s(k) \cdot T_{B|A} + r(k) \cdot (T_P - T_{B|A})]}{T_P} \quad \text{for all } k \quad (5.21)$$

with  $r(k)$  now being approximated by  $p_A(k)$ .

### 5.4.3 Parallel Combination of Several Segments

A parallel combination of several segments can be solved by an iterative application of the procedures presented in sections 5.4.1 and 5.4.2. Suppose that there are  $n$  segments to begin with. We can take two of those segments, solve them as a parallel combination, replace them with a single, aggregate segment and end up with a total of  $n-1$  segments. We proceed in this fashion until only one segment is left, which is then the desired solution. As can be seen from the formulas given above, the algorithm for combining segments in parallel possesses both commutative and associative properties, which allows us to combine segments in any order. However, as discussed in section 5.4.1.4, the more similar the two segments are in terms of size and structure, the greater a potential error in our solution. Thus, at each step, it is best to try to choose two segments that are as ‘‘dissimilar’’ as possible.

### 5.4.4 Parallel Combination of Sequential Combinations

In this section, we will present a procedure for solving a parallel combination of two sequential combinations, where we do not want to replace each sequential combination by an aggregate segment due to accuracy considerations. The result of this procedure is itself a sequential combination of segments. This procedure is also

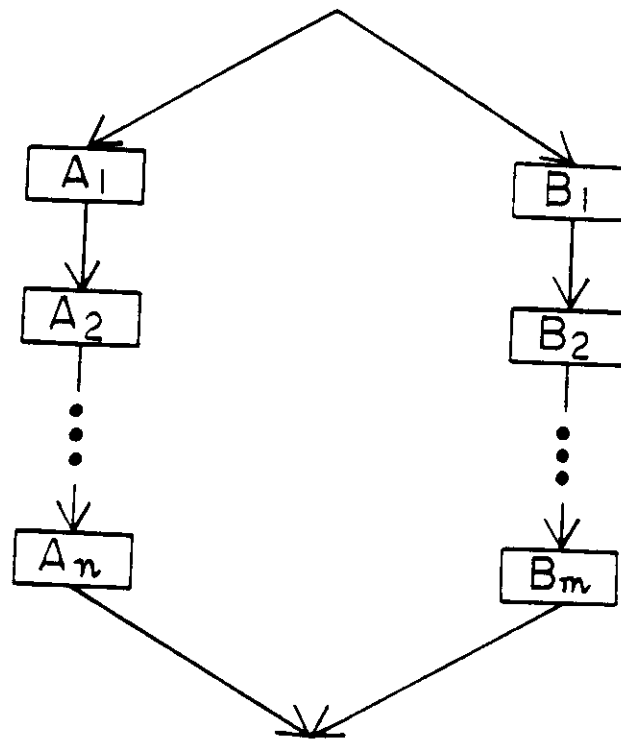
applicable to solving a parallel combination of two segments, where each constituent must be represented as a sequence of smaller segments (i.e., we cannot aggregate it into a single segment) as discussed in section 5.1.

The type of segment combination we are considering is illustrated in Figure 5.4.  $A$  is a sequential combination of segments  $A_1, A_2, A_3, \dots, A_n$ .  $B$  is a sequential combination of segments  $B_1, B_2, \dots, B_m$ . The result of our solution process will be a sequential combination  $P$ , consisting of segments  $P_1, P_2, \dots, P_q$ , where  $\max(n, m) < q < n+m$ . The exact value of  $q$  can only be determined at the end of the solution process.

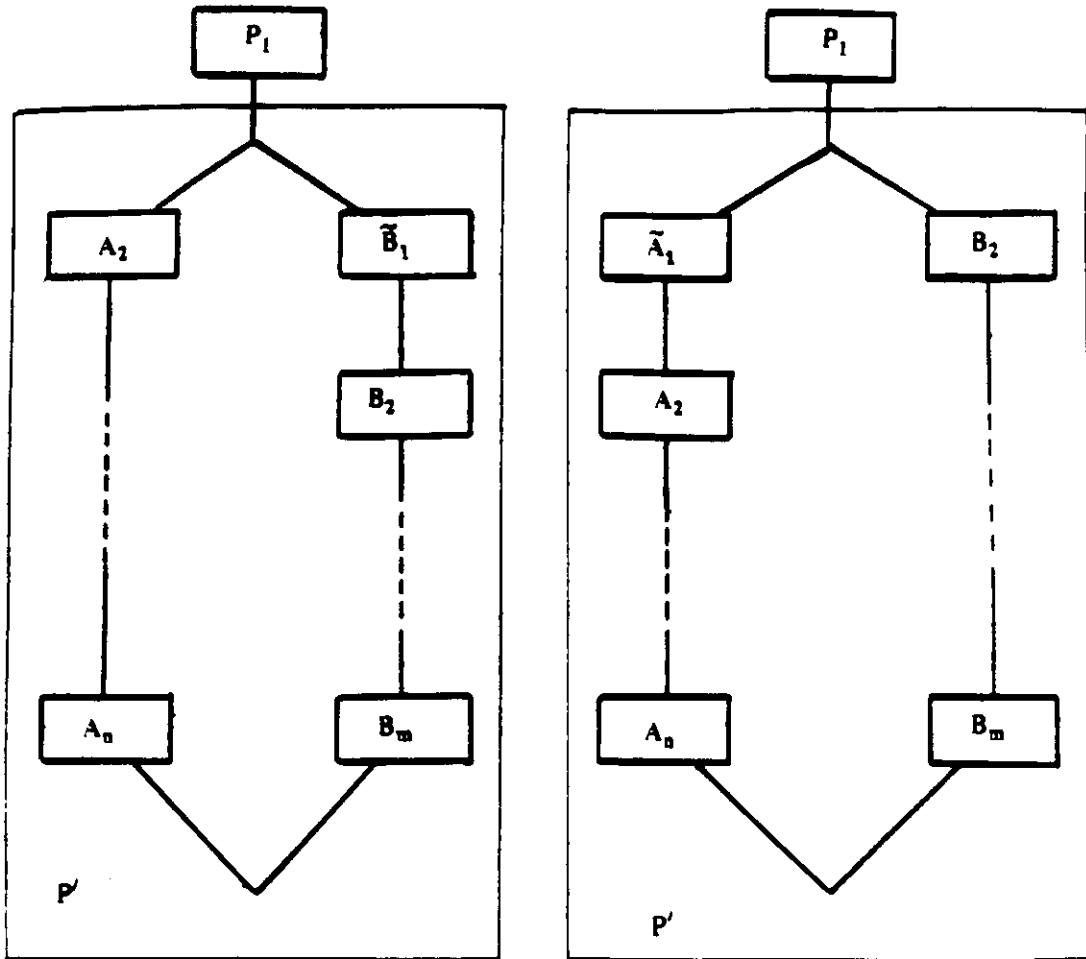
Our first step is to solve a parallel combination of segments  $A_1$  and  $B_1$ , in order to obtain segment  $P_1$ . Without loss of generality, let us suppose that segment  $A_1$  completes execution before segment  $B_1$  does. Then  $P_1$  will be defined by the following segment descriptor:  $\{N_{P_1}, T_{P_1}, p_{P_1}(k)\}$ , where  $N_{P_1} = N_{A_1} + N_{\hat{B}_1}$  and  $T_{P_1} = T_{A_1|B_1}$ .  $p_{P_1}(k)$  is the joint distribution of parallelism of segments  $A_1$  and  $B_1$  and can be estimated in the same fashion as  $s(k)$  in Equation (5.19).  $P$  can now be re-defined as the sequential combination of segment  $P_1$  and of the result of solving the combination  $P'$ , as depicted in Figure 5.5(a). In this combination, segment  $\tilde{B}_1$  is the complement of segment  $\hat{B}_1$ , with respect to segment  $B_1$ .

In the case that segment  $B_1$  completes execution first,  $P_1$  will be analogously defined, with  $N_{P_1} = N_{B_1} + N_{\hat{A}_1}$  and  $T_{P_1} = T_{B_1|A_1}$ . The portion of the original combination remaining after this step,  $P'$ , is shown in Figure 5.5(b), with segment  $\tilde{A}_1$  being the complement of segment  $\hat{A}_1$  with respect to segment  $A_1$ .

The next step is to solve a parallel combination of segments  $A_2$  and  $\tilde{B}_1$  (or, depending on the result of the first step,  $\tilde{A}_1$  and  $B_2$ ) to obtain segment  $P_2$ . Thus, by iteratively "reducing" the original combination, we can proceed to solve for seg-



**Figure 5.4** A Parallel Combination of Sequential Combinations



(a)  $A_1$  IS FIRST TO FINISH

(b)  $B_1$  IS FIRST TO FINISH

Figure 5.5 Result of Completing the First Step in Solving for P



ments  $P_2, P_3, P_4, \dots$ , until we “run out” of either  $A$ -segments or  $B$ -segments. The “leftover” segments can now be simply “appended,” in sequential order, to the portion of  $P$  that has been obtained so far, in order to produce the final solution. At each step, we “eliminate” either one  $A$ -segment or one  $B$ -segment or, in the case that the segments (being then considered) complete execution simultaneously, both an  $A$ -segment and a  $B$ -segment. Therefore, the maximum number of steps involved in solving this combination is  $n+m$ .

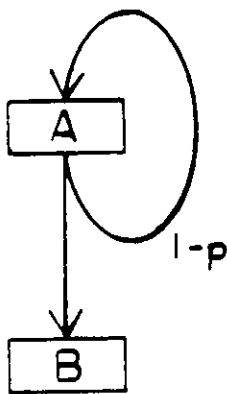
A parallel combination of several sequential combinations can be solved in a way analogous to solving a parallel combination of several *single* segments. That is, we can proceed by successively solving two sequential combinations at a time, and replacing them by an aggregate sequential combination, until only one combination is left, which is then the desired solution.

## 5.5 Looping Constructs

In this section, we examine the programming constructs which contain loops or cycles. These types of constructs are representative of DO loops and recursive procedures commonly used in programs. First, we will analyze and obtain a solution for a simple looping construct. We will then use the results obtained in that analysis as “building blocks” in solving more complex combinations.

### 5.5.1 Simple Loops

A simple “loop” combination,  $L$ , of segments  $A$  and  $B$  is depicted in Figure 5.6. In this combination, after segment  $A$  completes execution, it enables segment  $B$  and, with probability  $1-p$ , enables another invocation of itself, thus, starting the loop over again. After the iteration in which segment  $A$  does not re-enable itself, no new segments are enabled -- the execution of the loop terminates after all of the then existing segment invocations are completed. The number of iterations in loop  $L$  is a



**Figure 5.6 A Simple Loop Combination, L**

random variable which is geometrically distributed with mean  $1/p$ . In this section, we will present procedures for approximately computing the mean execution time of loop  $L$ ,  $T_L$  and its distribution of parallelism,  $p_L(k)$ ,  $k \geq 0$ . Note that  $N_L$  is given by:

$$N_L = \frac{N_A + N_B}{p} \quad (5.22)$$

After obtaining all of these values, loop  $L$  can be replaced by a single segment having description  $\{N_L, T_L, p_L(k)\}$ .

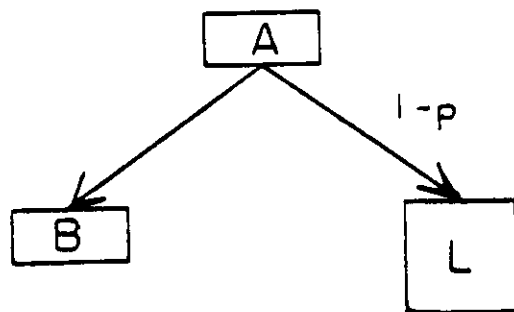
### 5.5.1.1 Computing Loop Mean Time Execution

In order to facilitate the following derivation, we will first expand the underlying computation control graph in a recursive fashion. The new, expanded graph is shown in Figure 5.7. Note that this type of recursive expansion is possible due to the fact that the termination probability  $p$  is time invariant. Let  $T_{L_1}$  be the mean time to execute  $L$ , given that the loop has only one iteration. Let  $T_{L_2}$  be the mean time to execute  $L$ , given that the loop has more than one iteration. Using Bayesian principles, the mean time to execute this loop,  $T_L$ , is given by:

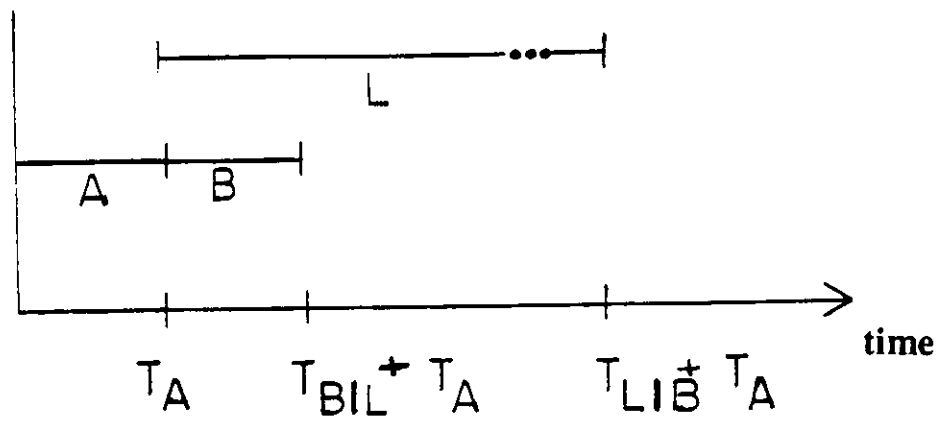
$$T_L = p \cdot T_{L_1} + (1-p) \cdot T_{L_2} \quad (5.23)$$

Now,  $T_{L_1}$  is simply  $T_A + T_B$ . In order to find  $T_{L_2}$ , let us refer to the timing diagram shown in Figure 5.8. Since combination  $L$  includes at least one instance of segment  $A$  and at least one instance of segment  $B$ , it is clear that  $T_{L|B}$  (as defined in section 5.4.1) is greater than  $T_{B|L}$  (i.e.,  $B$  is more likely to complete execution before  $L$ ). Thus,  $T_{L_2} = T_A + T_{L|B}$ . By applying our algorithm for solving parallel segment combinations, we get  $T_{L|B} = T_{B|L} + (T_L - T_C)$ . The result for  $T_L$  is obtained by combining all of the equations derived above.

$$\begin{aligned} T_L &= p \cdot [T_A + T_B] + (1-p) \cdot [T_A + T_{B|L} + (T_L - T_C)] \\ &= T_A + p \cdot T_B + (1-p) \cdot (T_{B|L} - T_C) + (1-p) \cdot T_L \end{aligned}$$



**Figure 5.7 Expanded Representation of the Simple Loop**



**Figure 5.8 Timing Diagram for Loop Mean Time Execution**

$$= \frac{[T_A + p \cdot T_B + (1-p) \cdot (T_{B|L} - T_L)]}{p} \quad (5.24)$$

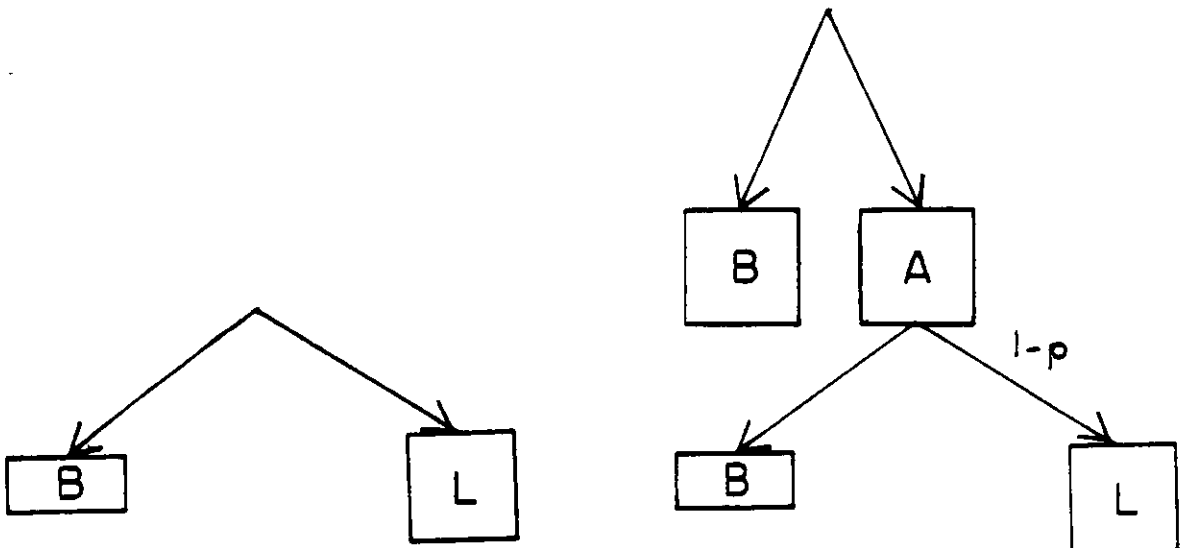
We will now show how to estimate the values of  $T_{B|L}$  and  $T_L$  in Equation (5.24). The following development is applicable to solving models with only one class of tasks (i.e.,  $c=1$ ). (With  $c>1$ , some of the ‘‘independence’’ and ‘‘fluid flow’’ assumptions necessary for this development are no longer valid.)

Obtaining  $T_{B|L}$  and  $T_L$  entails solving a parallel combination of segment  $B$  and loop  $L$ , which is represented by a computation control graph shown in Figure 5.9(a). In order to facilitate further development, it is advantageous to recursively expand this graph, as depicted in Figure 5.9(b). Such recursive expansion is possible due to the fact that the termination probability  $p$  is time-invariant. As can be seen from this expanded graph, our task is to solve a parallel combination of sequential combinations. The procedure for solving such a combination has been presented in Section 5.4.4. Thus, our first step is to solve a parallel combination of segments  $A$  and  $B$ . First let us consider the case where  $T_{B|A} < T_{A|B}$ , i.e., segment  $B$  completes execution before the first instance of segment  $A$  in loop  $L$  does. In this case, the solution is quite simple:

$$T_{B|L} = T_{B|A} \quad (5.25)$$

$$T_L = T_A \quad (5.26)$$

Now let us look at the (more complicated) case where  $T_{A|B} < T_{B|A}$ . Once again, we will apply the Bayesian approach to obtaining the desired solution. We will condition on the event of loop  $L$  consisting of only one iteration, i.e., the rightmost arc emanating from segment  $A$  (in Figure 5.9(b)) not being activated. The probability of this event occurring is  $p$ . First we will find  $T_{B|L}$  conditioned on the occurrence of the event. By applying the technique of section 5.4.4, we get:



(a) ORIGINAL GRAPH

(b) EXPANDED GRAPH

Figure 5.9 Parallel Combination of Segment B and Loop L

$$T_{B|L} = T_{A|B} + T_{\bar{B}|B} \quad (5.27)$$

where  $\bar{B}$  is the portion of segment  $B$  *not* completed during the “ $T_{A|B}$ ” time period. Using our “fluid flow” assumption of  $p_B(k)$  for all  $k$  being uniform over all subintervals of  $T_B$ , we can approximate  $T_{\bar{B}|B}$  by:

$$T_{\bar{B}|B} = \frac{N_{\bar{B}}}{N_B} \cdot T_{B|B} \quad (5.28)$$

$N_{\bar{B}}$  is the number (possibly non-integral) of tasks constituting  $\bar{B}$  and is given by  $N_B - N_{\hat{B}}$  (the formula for estimating  $N_{\hat{B}}$  was given in section 5.4.1.3).

We will now solve for the conditional value of  $T_{B|L}$  given that loop  $L$  consists of more than one iteration, i.e., the *non-occurrence* of the chosen event. The probability of this situation is  $1-p$ . By considering the timing diagram in Figure 5.10, and again applying the procedure given in section 5.4.4, we get:

$$T_{B|L} = T_{A|B} + T_{\bar{B}|B|L} \quad (5.29)$$

In equation (5.29),  $T_{\bar{B}|B|L}$  is the mean time to execute  $\bar{B}$  when both another instance of segment  $B$  and loop  $L$  are running concurrently with  $\bar{B}$ . Applying the argument used to derive equation (5.28),  $T_{\bar{B}|B|L}$  can be estimated by:

$$T_{\bar{B}|B|L} = \frac{N_{\bar{B}}}{N_B} \cdot T_{B|B|L} \quad (5.30)$$

The only remaining unknown value that is needed to complete the solution is  $T_{B|B|L}$ . In order to estimate this value, we will use the following “heuristic” argument. Let us consider the difference  $T_{B|B|L} - T_{B|B}$ . This difference represents the increase in the execution time of segment  $B$  (while executing concurrently with loop  $L$ ) caused by concurrent execution of a second instance of segment  $B$ . Our main assumption is that the “interference” of the second instance of  $B$  with the execution of the first instance of  $B$  is not significantly affected by the presence (or absence) of



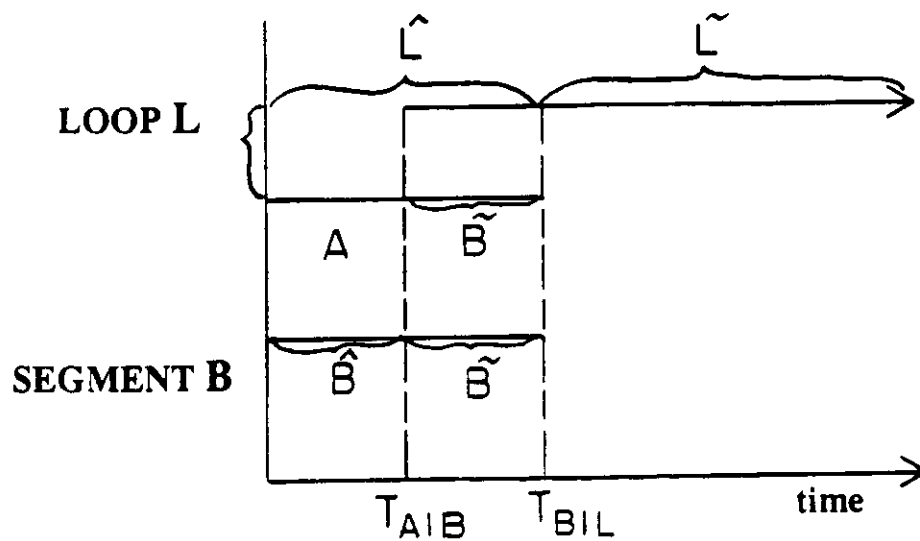


Figure 5.10 Timing Diagram of the Concurrent Execution of B & L

loop  $L$ . The latter assumption leads to the following approximation:

$$\begin{aligned} (T_{B|B|L} - T_{B|L}) &= (T_{B|B} - T_B) \\ \rightarrow T_{B|B|L} &= T_{B|L} + (T_{B|B} - T_B) \end{aligned} \quad (5.31)$$

In general, the greater the utilization of system's resources is, the greater is the impact of any additional workload (e.g., the execution of a second instance of segment  $B$ ) on the completion of the current workload (e.g., the execution of the first instance of  $B$ ). Since the concurrent execution of  $L$  represents an increase in the utilization of system's resources, the (linear) approximation given by equation (5.31) is, in general, a lower bound on the value of  $T_{B|B|L}$ . Note, however, that, if both  $B$  and  $L$  have a constant degree of parallelism and the  $P/C$  subnetwork consists of a single processor-sharing resource, then equation (5.31) becomes an *exact* relationship. Thus, since the global throughput of a saturated queueing network is linear, the "sooner" the  $P/C$  subnetwork saturates, the tighter is this lower bound on  $T_{B|B|L}$ .

Now that we have all of the necessary components, we are ready to "put together" the final (approximate) solution for  $T_{B|L}$ . By unconditioning on the event of loop  $L$  having exactly one iteration, we get:

$$\begin{aligned} T_{B|L} &= p \cdot [T_{A|B} + T_{B|B}] + (1-p) \cdot [T_{A|B} + T_{B|B|L}] \\ &= T_{A|B} + [p \cdot T_{B|B} + (1-p) \cdot T_{B|B|L}] \cdot \frac{N_B}{N_B} \\ &= T_{A|B} + [T_{B|B} + (1-p) \cdot T_{B|L} - T_B] \cdot \frac{N_B}{N_B} \\ &= \frac{T_{A|B} + [T_{B|B} - (1-p) T_B] \frac{N_B}{N_B}}{[1 - (1-p) \frac{N_B}{N_B}]} \end{aligned} \quad (5.32)$$

We still have to solve for the value of  $T_{\hat{L}}$ , where  $\hat{L}$  is the portion of loop  $L$  completed by the time the execution of segment  $B$  has been completed, i.e., during

the first  $T_{B|L}$  time units. Let us take another look at equation (5.32):

$$T_{B|L} = T_{A|B} + p \cdot T_{B|B} + (1-p) \cdot T_{\bar{B}|B|L} \quad (5.33)$$

Thus,  $T_{B|L}$  can be interpreted as consisting of the “ $T_{A|B}$ ” time period and of either the “ $T_{\bar{B}|B}$ ” time period or the “ $T_{\bar{B}|B|L}$ ” time period. The respective probabilities of each case being  $p$  and  $1-p$ . During the “ $T_{A|B}$ ” time period, the instance of segment  $A$  in the first iteration of loop  $L$  is completed. The mean time to execute this part of the loop, while  $L$  is running alone on the system, is, of course,  $T_A$ . Since, in our physical domain model, the whole system is represented by a single state-dependent, processor-sharing service center, identical segments are assumed to utilize system’s resources on an “equal” basis. Thus, the average number of tasks of the *second* instance of segment  $B$  completed during the “ $T_{\bar{B}|B}$ ” time period is  $N_{\bar{B}}$ . From our “fluid flow” assumption that  $p_B(k)$ , for all  $k$ , is uniform over all subintervals of  $T_B$ , the mean time to execute this portion of  $L$  is  $T_{\bar{B}}$ . Using the same analysis, the average number of tasks of the *second* instance of segment  $B$  completed during the “ $T_{\bar{B}|B|L}$ ” time period is  $N_{\bar{B}}$ . Due to the “processor-sharing” nature of our physical domain model and the assumption that there is only one chain of queueing network customers, the *relative* rates at which two segments are being completed are independent of any additional workload present in the system. The preceding observations indicate that the portion of the second instance of loop  $L$  completed during the “ $T_{\bar{B}|B|L}$ ” time period is the same as that completed during the “ $T_{\bar{B}|L}$ ” time period. Thus, the average time needed to execute the part of the loop completed during the “ $T_{\bar{B}|B|L}$ ” time period is  $T_{\bar{B}|L}$ . The estimate for  $T_{\bar{L}}$  can now be constructed by combining the individual results, derived above, for each of the three time periods.

$$T_{\bar{L}} = T_A + p \cdot T_{\bar{B}} + (1-p) \cdot T_{\bar{B}|L}$$

$$= T_A + [p \cdot T_B + (1-p) \cdot T_{B|L}] \cdot \frac{N_B}{N_B} \quad (5.34)$$

### 5.5.1.2 Loop Combination Distribution of Parallelism

In this section, we will show how to find the distribution of parallelism,  $p_L(k)$ ,  $k \geq 0$ , for the loop combination of segments  $A$  and  $B$ ,  $L$ . The following development is also applicable to solving models with only one class of tasks (i.e.,  $c=1$ ).

#### 5.5.1.2.1 Definition of Notation

Consider the timing diagram shown in Figure 5.11. In this diagram, the time period during which loop  $L$  is being executed,  $T_L$ , has been separated into two phases: Phase I and Phase II. During the first phase, the  $\hat{L}$  portion of the loop is being executed. Recall that  $\hat{L}$  is the portion of loop  $L$  completed during the time required to execute segment  $B$ , when the parallel combination of  $B$  and  $L$  is being executed. During the second (final) phase, the remaining portion of loop  $L$ ,  $\tilde{L}$ , is being executed to completion. We now introduce the following definitions:

- $t_{\hat{L}}(k)$ : the average amount of time in Phase I of  $T_L$  during which there are exactly  $k$  enabled tasks;
- $t_{\tilde{L}}(k)$ : the average amount of time in Phase II of  $T_L$ , during which there are exactly  $k$  enabled tasks;
- $t_L(k)$ : the average amount of time, in both phases of  $T_L$  combined, during which there are exactly  $k$  enabled tasks.

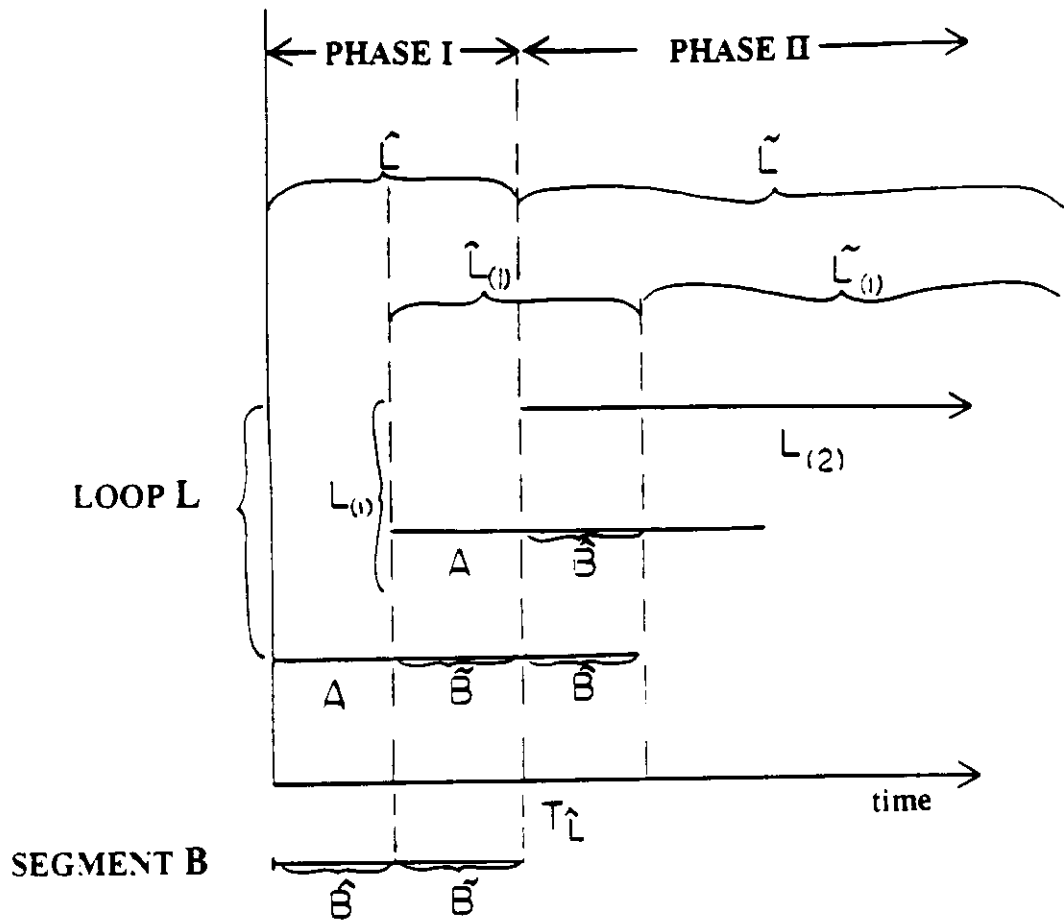


Figure 5.11 Breakdown of Execution of L

### 5.5.1.2.2 Derivation of Average Amount of Time in Phase I

In the case where  $T_{B|A} < T_{A|B}$ ,  $\hat{L}$  is simply  $\hat{A}$ . Using the “fluid flow” assumption that  $p_{\hat{A}}(k) = p_A(k)$ , for all  $k$ , we get:

$$t_{\hat{L}}(k) = T_{\hat{A}} \cdot p_A(k) \quad (5.35)$$

We will now consider the case where  $T_{A|B} < T_{B|A}$ . From the derivation of  $T_{\hat{L}}$  given in section 5.5.1.1, we have that  $\hat{L}$  is the sequential combination of segment  $A$  and the EXCLUSIVE-OR combination of  $\bar{B}$  and a portion of a parallel combination of  $\bar{B}$  and  $L$  (the portion completed during the “ $T_{\bar{B}|L}$ ” time period), respective probabilities being  $p$  and  $1-p$ . Before proceeding, we will introduce two assumptions: (1)  $p_B(k)$ , for all  $k$ , is uniform over all subintervals of  $T_B$ ; and (2)  $p_{\hat{L}}(k)$ , for all  $k$ , is uniform over all subintervals of  $T_{\hat{L}}$ . Note that, since  $\bar{B}$  is a part of  $B$ , the portion of  $L$  completed during the “ $T_{\bar{B}|L}$ ” time period is actually a part of  $\hat{L}$ . From our assumptions, it follows that the distribution of parallelism during the latter time period is the “joint” distribution of parallelism of segments  $B$  and  $\hat{L}$ . Using equation (5.19), the “joint” distribution of parallelism of  $B$  and  $\hat{L}$ ,  $p_{B|\hat{L}}(k)$ , can be estimated by:

$$p_{B|\hat{L}}(k) = \sum_{i=0}^k p_B(i) \cdot p_{\hat{L}}(k-i) \quad \text{for } k=1,2,\dots \quad (5.36)$$

By applying the techniques developed in sections 5.2 and 5.3 to the results and assumptions presented above, we get:

$$\begin{aligned} t_{\hat{L}}(k) &= T_A \cdot p_A(k) + p \cdot T_B \cdot p_B(k) + (1-p) \cdot T_{\bar{B}|L} \cdot p_{B|\hat{L}}(k) \\ &= T_A \cdot p_A(k) + pT_B \cdot p_B(k) + (1-p)T_{B|L} \cdot p_{B|\hat{L}}(k) (N_{\bar{B}}/N_B) \end{aligned} \quad (5.37)$$

### 5.5.1.2.3 Derivation of Average Amount of Time in Phase II

In the case where  $T_{B|A} < T_{A|B}$ ,  $\tilde{L}$  is the sequential combination of segment  $\tilde{A}$  and the EXCLUSIVE-OR combination of segment  $B$  and a parallel combination of  $B$  and  $L$ , respective probabilities being  $p$  and  $1-p$ . Note that  $T_{L|B} = T_{B|\hat{L}} + T_{\hat{L}}$ . Also note that, during the " $T_{\hat{L}}$ " time period, the average amount of time when there are exactly  $k$  enabled tasks is, by definition,  $t_{\hat{L}}(k)$ . Using these two observations and the assumption that  $p_{\tilde{A}}(k) = p_A(k)$ , for all  $k$ , we get:

$$\begin{aligned} t_{\hat{L}}(k) &= T_{\tilde{A}} \cdot p_A(k) + pT_B \cdot p_B(k) + (1-p) T_{B|\hat{L}} \cdot p_{B|\hat{L}}(k) + t_{\hat{L}}(k) \\ &= \frac{T_{\tilde{A}} \cdot p_A(k) + p \cdot T_B \cdot p_B(k) + (1-p)T_{B|\hat{L}} \cdot p_{B|\hat{L}}(k)}{p} \end{aligned} \quad (5.38)$$

The "joint" distribution of parallelism of  $B$  and  $\hat{L}$ ,  $p_{B|\hat{L}}(k)$ , can be estimated by equation (5.36).

We will now consider the case where  $T_{A|B} < T_{B|A}$ . First we note that, due to our assumption of  $p_B(k)$ , for all  $k$ , being uniform over all subintervals of  $T_B$ , any part of segment  $B$ , with  $N_{\tilde{B}}$  being the average number of tasks executed, has the same segment descriptor as segment  $\tilde{B}$ . From the latter observation and the timing diagram shown in Figure 5.11, we can see that  $\tilde{L}$  is the EXCLUSIVE-OR combination of segment  $\tilde{B}$  and the parallel combination of  $\tilde{B}$  with the sequential combination of a portion of  $\tilde{L}$  (the portion completed during the  $T_{\tilde{B}|\tilde{L}}$  time period) and  $\tilde{L}$ , respective probabilities being  $p$  and  $1-p$ . Thus,

$$T_{\tilde{L}} = p \cdot T_{\tilde{B}} + (1-p) \cdot (T_{\tilde{B}|\tilde{L}} + T_{\tilde{L}}) \quad (5.39)$$

From the assumptions stated in section 5.5.1.2.2, it follows that the distribution of parallelism during the  $T_{\tilde{B}|\tilde{L}}$  time period is the joint distribution of parallelism of segments  $B$  and  $\hat{L}$ ,  $p_{B|\hat{L}}(k)$ . By applying the arguments used in the preceding paragraph, we get:

$$\begin{aligned}
t_L(k) &= p \cdot T_B \cdot p_B(k) + (1-p) \cdot T_{B|L} \cdot p_{B|L}(k) + t_L(k) \\
&= \frac{p \cdot T_B \cdot p_B(k) + (1-p) \cdot T_{B|L} \cdot p_{B|L}(k)}{p} \\
&= \frac{p \cdot T_B \cdot p_B(k) + (1-p) \cdot T_{B|L} \cdot p_{B|L}(k) \cdot (N_B/N_B)}{p} \tag{5.40}
\end{aligned}$$

#### 5.5.1.2.4 Average Fractions of Time during Loop L

Since  $T_L = T_L + T_L$ ,

$$t_L(k) = t_L(k) + t_L(k) \tag{5.41}$$

By definition,  $p_L(k)$  is the average fraction of time, during the execution of loop L (i.e., during the " $T_L$ " time period), when there are exactly  $k$  enabled tasks. Thus,

$$p_L(k) = \frac{t_L(k)}{T_L} \tag{5.42}$$

Analogously,

$$p_L(k) = \frac{t_L(k)}{T_L} \tag{5.43}$$

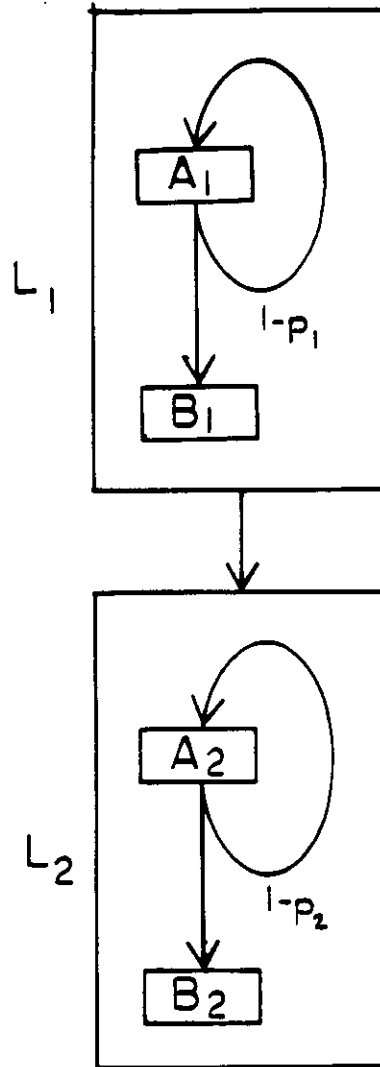
$$p_L(k) = \frac{t_L(k)}{T_L} \tag{5.44}$$

It is important to note that the equations for estimating  $p_L(k)$  form a recurrence relationship. That is, in order to solve for  $p_L(k)$ , one must first obtain  $p_L(k-1)$ ,  $k=2,3,\dots$ . The same observation applies to  $t_L(k)$ ,  $p_L(k)$ ,  $t_L(k)$ ,  $p_L(k)$  and  $t_L(k)$ .

## 5.5.2 Complex Loops

In this section, we will consider some of the more complex looping structures. First, we will take a look at nested loops. Figure 5.12 is an example of such structure. In this structure, segments  $L_1$  and  $L_2$ , which constitute loop  $L$ , are them





**Figure 5.12 Nested Loop**

loops. We will call  $L1$  and  $L2$  *inner loops*. We can solve this segment combination in a hierarchical fashion. First, we solve each inner loop individually, using the procedures presented in the previous section, and replace each by a single, composite segment. We then have a loop consisting of two “simple” segments, which is identical in structure to the simple loop that had been analyzed and solved in Section 5.5.1. We can then apply our procedures for solving simple loops one more time to obtain the final result.

Another possible looping construct is one where the probability of termination is time-varying (i.e. it varies with each new iteration of the loop). This type of a loop is depicted in Figure 5.13. In this combination,  $LV$ ,  $p_i$  is the probability that, on the  $i$ -th iteration (if it occurs), the  $i$ -th instance of segment  $A$  will *not* enable the  $(i+1)$ -th instance of segment  $A$ . The probability that this loop repeats itself exactly  $n$  times,  $q(n)$ , is given by:

$$q(n) = p_n \cdot \sum_{i=1}^{n-1} (1-p_i) \quad (5.45)$$

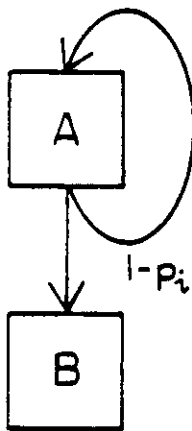
The average number of iterations,  $q$ , is then equal to:

$$q = \sum_{n=1}^{\infty} n \cdot q(n) \quad (5.46)$$

We will define  $L(n)$ ,  $n=1,2,\dots$  to be the variant of a simple loop which has *exactly*  $n$  iterations. Loop  $LV$  can now be viewed as the EXCLUSIVE-OR combination of loops  $L(1)$ ,  $L(2)$ ,  $L(3)$ , ..., with the weight on loop  $L(n)$  being equal to  $q(n)$ . Thus, the mean time to execute  $LV$ ,  $T_{LV}$ , is given by:

$$T_{LV} = \sum_{n=1}^{\infty} q(n) \cdot T_{L(n)} \quad (5.47)$$

Its distribution of parallelism is  $p_{LV}(k)$ ,  $k \geq 0$ , where:



**Figure 5.13 Time-Varying Loop (LV)**

$$p_{LV}(k) = \frac{\sum_{n=1}^{\infty} q(n) \cdot T_{L(n)} \cdot p_{L(n)}(k)}{T_{LV}} \quad \text{for all } k \quad (5.48)$$

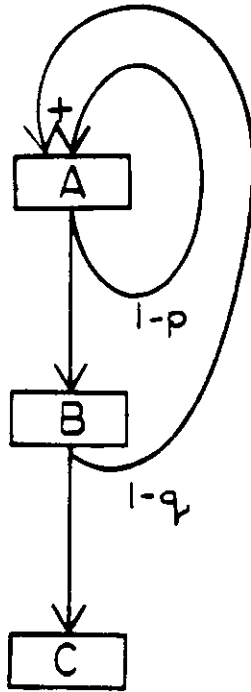
The average number of tasks executed during an invocation of loop  $LV$  is  $N_{LV} = q \cdot (N_A + N_B)$ . An approximate solution to  $L(n)$ , for each  $n$ , can be found by representing  $L(n)$  as a simple loop, with the time-invariant probability of termination,  $p$ , set to  $1/n$ . Note that, if  $L$  is a simple loop with  $p = 1/n$ , then its average number of iterations equals  $n$ , and its mean execution time and distribution of parallelism can be expressed as follows:

$$T_L = \sum_{j=1}^{\infty} q(j) \cdot T_{L(j)} \quad (5.49)$$

$$p_L(k) = \frac{\sum_{j=1}^{\infty} q(j) \cdot T_{L(j)} \cdot p_{L(j)}(k)}{T_L} \quad \text{for all } k, \quad (5.50)$$

where  $q(j)$  is computed from equation (5.45), with  $p_i = p$ , for all  $i$ . Therefore, this representation of  $L(n)$  yields the exact solution if and only if  $T_{L(n)}$  and  $p_{L(n)}(k)$ , for all  $k$ , are linear functions of  $n$ .

Some types of looping constructs do not lend themselves easily to the kind of combinatorial analyses presented above. Interleaved loops are one of those types. An example of an interleaved loop is shown in Figure 5.14. We have attempted to use a heuristic approach, analogous to the ones described earlier, to optimize the solution process of such loops. However, we have not been able to develop a procedure of less than an "overwhelming" complexity. Thus, for these types of constructs, we have to resort to applying the solution procedure developed in Chapter 4 to the computation control graph for the entire construct.



**Figure 5.14 Interleaved Loop**

## **5.6 Other Heuristics**

We will now discuss some additional methods for optimizing our solution procedures for certain types of program structures and distributed system architectures.

### **5.6.1 Hierarchical Combinations**

As stated in Section 2.4.2, complicated program structures can usually be represented as nested combinations of segments, with simple programming constructs being used at each nesting level. Thus, we can view a program as being represented in progressively more detail as we traverse its structural hierarchy from top to bottom. Each level of the hierarchy is a combination of segments (of varying degrees of complexity), with the description of each segment being hierarchically defined by the lower levels. A combination at a given level can be solved once we know the segment descriptors (as defined in section 5.1) of all of its constituents.

Thus, starting with the lowest level, we can progressively apply the algorithms presented earlier in this chapter to solve each higher level and, finally, the whole program. This approach has already been utilized in Section 5.5.2 in solving nested loops. For most combinations of segments, there is much less computation involved in solving each constituent individually and then combining the solutions in the aforementioned fashion, than in constructing and solving the Markov process for the computation control graph of the whole combination. Therefore, by hierarchically applying the techniques presented above, we can make the analysis of very complex programs computationally feasible.

### 5.6.2 Multi-Chain Models

The various solution optimization procedures that have been developed in this chapter do not readily apply to physical domain models with different chains of queueing network customers. In this section, we will discuss some heuristic approaches one can take in applying the “single-chain” algorithms presented above to solving multi-chain systems.

One approach is to convert a multi-chain system into a single-chain system by creating a new, composite chain of queueing network customers from the different chains present in the original system. All of the tasks in the program being modeled will then be represented by customers of this new chain. Attributes of the composite chain, such as the service time distribution at each service center and routing probabilities, will be determined from the corresponding attributes of the original chains. We can estimate the value of a particular attribute in the composite chain by a linear interpolation of the values of that attribute in the original chains, with the “weight” on each value being based on the average number of tasks (represented by the corresponding chain) executed during the program runtime.

If, for a given segment combination, the system resources used by the tasks of different segments are completely disjoint, then we can take the following approach. Since the execution of tasks of one segment does not “interfere” with the execution of tasks of another segment, the task throughput of a given segment is not affected by other segments executing at the same time. Based on the latter observation, we can apply all of the algorithms developed in this chapter to such combinations, without explicitly considering the fact that tasks from different segments are represented by different customer chains. For example, using this approach, the mean time to execute a parallel segment combination would be estimated by the maximum of the mean execution times of individual segments.

The approach of the last paragraph can be extended to cases where the resources used by different segments are not completely disjoint, i.e., some resources are shared by several segments. In such cases, the task throughput of a given segment, with other segments being concurrently executed, can be estimated by reducing the capacities of the shared resources. The degree of the capacity reduction for each resource would have to be estimated from the expected utilization of that resource by the “interfering” segments. We can improve upon the latter approximation by iteratively solving the same model, using, at each step, the best available (most recent) set of estimates. This procedure is analogous to the method for solving queueing networks proposed by de Souza e Silva [SOU83].

### 5.6.3 Process Arrivals

So far in this chapter, we have considered one program (or a set of programs) running alone in a distributed environment and have shown how to estimate its mean response time and throughput. In this section, we will analyze the behavior of a distributed, multiple-computer system with stochastic arrivals of programs or processes. We will assume that all processes are identically structured and that their inter-arrival times are exponentially distributed (i.e., the arrival process is Poisson). For process  $A$ , let  $T_{A(n)}$  be the average time needed to execute the parallel combination of  $n$  instances of  $A$  in the environment being considered. The procedure for estimating  $T_{A(n)}$  was presented in section 5.4.3. (Note that, in computing  $T_{A(n)}$ , the estimates for  $T_{A(2)}$ , ...,  $T_{A(n-1)}$  are obtained as a by-product.) The relative increase in the throughput of processes of the system, when  $n$  processes are available for execution, is given by  $\frac{n \cdot T_{A(1)}}{T_{A(n)}}$ . Our objective is to find the average system response time for process  $A$  with different values for the process arrival rate.



Due to the processor-sharing nature of our physical domain model, we can approximate the solution to this system by representing it as an M/G/1 service center with state-dependent service rates. The work demands of the jobs (each representing an instance of process  $A$ ) arriving to this service center have a general distribution with the mean equal to  $T_A(1)$ . These jobs are scheduled according to the PS queueing discipline. (Since the PS scheduling policy is used, higher moments of the work demand distribution are not necessary to solve this queueing model.) The capacity function for this service center is given by:

$$n \cdot \frac{T_A(1)}{T_A(n)}, \quad n=1,2,3,\dots \quad (5.51)$$

The solution to the type of the queueing system described above is well known [LAV82]. Thus, in order to estimate the mean response time for process  $A$ , all one has to do is to obtain  $T_A(n)$ ,  $n=1,2,\dots$ , and then apply appropriate formulas. Note that, in practice, one would only compute the first  $I$  elements of the capacity function. The value of  $I$  is chosen such that:

$$\left( \frac{i}{T_A(i)} - \frac{I}{T_A(I)} \right) < e, \quad \text{for all } i > I, \quad (5.52)$$

where  $0 < e \ll I / T_A(I)$ . That is, the composite departure rate from the M/G/1 service center with  $i$  jobs present,  $i > I$ , is “nearly the same” as that with  $I$  jobs present. Thus,  $I$  approximates the number of instances of process  $A$  at which the physical system “saturates.”

## 5.7 Degree of Aggregation vs. Accuracy

In this chapter, we have presented several heuristic techniques and strategies for reducing the computational cost of solving for the mean execution time of programs (or processes) having well-defined structures and being composed of commonly used programming constructs. Most of the aforementioned methods rely on

abstracting the dynamic behavior of a program segment into a set of steady-state performance measures, such as the ones described in section 5.1. They also assume that, when executing concurrently, segments are stochastically independent of each other in queueing for and utilizing system resources. Thus, in our optimization procedures, we ignore the transient changes in a segment's behavior and how a segment's distribution of parallelism is affected by concurrent execution of other segments. These simplifications will often result in reducing the accuracy of the final solution for the program being modeled.

The degree to which the accuracy is reduced varies from model to model and depends on such factors as how much the segments' behavior varies with time, relative sizes of segments, properties of individual tasks, and how many times the optimizations are applied hierarchically. For a particular model, the actual impact on accuracy of a certain optimization procedure can only be determined by solving the same model without applying that optimization. In general, the smaller the segments are (relative to the total program size) which are being aggregated and the lower the level of the program's hierarchy is where the optimizations are performed, the greater is the potential error in the final results. Thus, a modeler can control the tradeoff between computational cost and accuracy by "properly" choosing what segments to aggregate and at what level of the program's hierarchy to apply our heuristic methods.

CHAPTER 6  
CASE STUDY A:  
CONCURRENCY CONTROL IN DISTRIBUTED COMPUTATIONS

After presenting the development of the theoretical framework of our methodology, we are now ready to demonstrate its effectiveness when applied to solving practical system design and analysis problems. In this and the following chapter, we will present two case studies. Each is drawn from a different application area of distributed computing. These studies serve to illustrate the power and scope of our modeling approach and provide an indication of the expected accuracy level. In each case study, we first describe the application and the environment(s) being modeled, including the objectives of our analysis (e.g. obtaining some performance measures of interest). We then present our model and solution procedure and, finally, tabulate the numerical results given by our methodology and compare them with those obtained from simulation.

In this chapter, we will analyze the tradeoffs between different parallel implementations of the same algorithm. Each implementation will be evaluated on several distributed architectures. The particular algorithm selected for this experiment is that used for computing the steady state distribution of the number of customers at a service center of a queueing network. The presentation of this case study will be deferred, however, until after we describe the simulation environment that had been used for validating the accuracy and robustness of the results of this and other modeling applications.

## **6.1 Simulation Environment**

We used the RESQ (RESearch Queueing) package (developed at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York), running under the VM/CMS operating system, to conduct simulations of our case studies. In this section, we will briefly review RESQ and its usage, discuss the features pertinent to our simulations and describe major components of our simulation models.

### **6.1.1 Overview of RESQ**

RESQ is a system for constructing and solving (via numerical or simulation methods) extended queueing network models [SAU81b]. The class of RESQ networks is called "extended" because it allows model characteristics not permitted in product form queueing network models. RESQ incorporates a high level language to concisely describe the structure of the model and to specify constraints on the solution process. These constraints include the required accuracy level (in terms of relative widths of confidence intervals) and bounds on the length of a simulation run. A main feature of the language is the capability to describe models in a hierarchical fashion, allowing an analyst to define parametric submodels, which are analogous to macros or procedures in programming languages.

Major components of RESQ models are queues and nodes. Traditional service centers are represented by "active" queues, which can utilize a variety of job scheduling disciplines. "Passive" queues are pools of tokens, which allow convenient representation of simultaneous resource possession. The types of nodes used in RESQ models include class nodes (associated with active queues); allocate, release, create and destroy nodes (associated with passive queues); sources and sinks; split nodes; fission and fusion nodes; and dummy nodes. Each RESQ job travels from one node to another, according to the routing definition given for the customer chain to which it belongs. Models can also be supplemented with different types

of variables; each variable can be associated with either a particular job, a single chain or the whole model.

RESQ provides a simulation capability with special features not found in general purpose simulation languages [SAU81b]. The most important of these are statistical output analysis techniques, which provide confidence intervals for simulation results and stopping rules for determining when the simulation should end. These techniques include Independent Replications, Regenerative and Spectral methods.

The RESQ user interfaces are based on interactive dialogues, which can both accommodate sophisticated users and large models as well as educate new users. A transcript of a model definition dialogue is kept for the user. The user may edit this transcript and have it translated again, with or without additional interactive dialogue. In addition to the model definition dialogue and translator, there is a model evaluation dialogue associated with the solution components. This dialogue allows the user to selectively obtain performance measures. Models may be defined with parameters, so that solutions of several related models may be obtained in a single evaluation, without retranslation of the model.

### **6.1.2 Simulation Models**

Each simulation model consists of two parts. The first part, consisting primarily of *active* queues, is used to represent the physical resources of the distributed system being modeled. The second part, consisting primarily of *passive* queues, is used to represent the intertask dependencies of the selected program as mandated by the corresponding computation control graph. This kind of "separation of roles" facilitates a convenient mechanism for simulating different programs running on the same system or simulating the same program running in different environments.

Each system resource is modeled by an active queue with attributes such as the number of servers, the service time distribution of each customer class, the queueing discipline, etc., chosen to match the properties of that resource and the tasks it services. The interconnection of elements in the physical system is represented by properly selected routing functions in the model. There are as many customer chains in the model as there are tasks in the corresponding program, each being associated with a specific task. Each chain is defined as closed and contains as many customers as the maximum number of possible instances of the associated task in the program. A job variable is used to identify the particular instance of the task represented by a given queueing network customer.

The precedence relationships among tasks are modeled using arrays of global variables and passive queues. A set of global variable arrays and a passive queue are associated with each customer chain in the model. Each array in a given set represents one of the operands needed by the corresponding task. The  $i$ -th element of each array contains information about the  $i$ -th instance of that task. Passive queues act as “semaphores” by controlling the “dispatching” of waiting customers into the “active” part of the model. Upon leaving the active part of the model, each customer updates the proper global variable arrays, as determined by the underlying computation control graph. It then waits to obtain a token from the passive queue associated with its customer chain.

The *initial* state of the RESQ model has “starting” customers, those which represent tasks enabled at the start of the program at the entrance to the “active” part of the model; the other customers wait for tokens at their respective passive queues; all passive queues have no tokens; global variable arrays are initialized to indicate that no operands have been received. After the customer representing the *last* task of the program departs from the “active” part, it “creates operands” for the “starting” customers. Thus, each simulation run is equivalent to repeatedly exe-

cuting the same program.

The statistical output analysis method of Independent Replications was used to place confidence intervals on the performance measures. Each replication consisted of a number of program executions, that number being chosen according to the confidence level desired.

## 6.2 Application Description

The goal of our first study is to assess the performance of three different parallel implementations of a particular algorithm. Each implementation will be evaluated with a number of distributed systems, each varying in both configuration size and architectural profile. Subsequently, we should be able to determine which implementation is best suited for a particular environment and, conversely, which architecture is most appropriate for a given implementation.

### 6.2.1 The Algorithm

The algorithm selected for this case study comes from the domain of computational methods for queueing network analysis. Its formula is shown in Figure 6.1. The function of this algorithm is to obtain the steady state distribution of the number of customers present at service center  $j$ ,  $n_j$ , of a closed, product-form queueing network having a total population of  $N$  customers [LAV82]. This algorithm is a by-product of the Convolution Algorithm for computing the normalization constant,  $G(N)$ , for closed, product-form queueing networks [LAV82]. It is applicable to fixed-capacity service centers only. Constant  $G(k)$ ,  $k=1,2,3,\dots,N$  is the normalization constant for the corresponding queueing network having a total population of

$$Pr\{n_j = i\} = (\rho_j^i / G(N)) \cdot [G(n-i) - \rho_j \cdot G(N-i-1)] , \quad i=0, 1, \dots, N$$

**Figure 6.1**  
**Computation of the Distribution of the Number of Customers at Service Center j**



only  $k$  customers and is assumed to have been previously computed. By definition,  $G(0)=1$  and  $G(k)=0, k < 0$ .

In order to obtain the full range of values for the probability density function  $Pr\{n_j=i\}, i=0,1,\dots,N$ , one must apply the formula given in Figure 6.1  $N+1$  times. This involves the computation of factor  $(\rho_j)^i$  for  $i=1,\dots,N$ . An efficient way to accomplish the latter computation is to preserve the value of  $(\rho_j)^k$  for calculating the value of  $(\rho_j)^{(k+1)}$ . That is:

$$(\rho_j)^{(k+1)} = \rho_j \cdot [(\rho_j)^k].$$

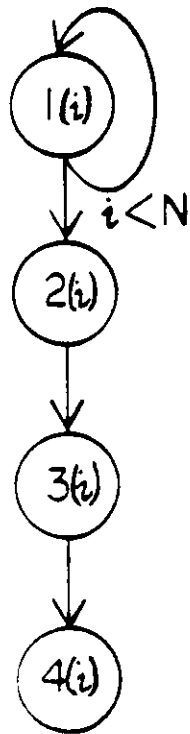
This “work conserving” approach will be utilized in each one of the three parallel implementations of this algorithm presented below.

### 6.2.2 Implementation A

Before describing this first implementation, we will designate a task for each of the arithmetic operations in the algorithm. In the definitions given below,  $R[t]$  represents the result generated by task  $t$ , for all  $t$ .

task 1( $i$ ):	$1/G(N)$ $\rho_j \cdot R[1(i-1)]$	$i=0$ $i=1,2,\dots,N$
task 2( $i$ ):	$\rho_j \cdot G(N-i-1)$	$i=0,1,\dots,N$
task 3( $i$ ):	$G(N-1)-R[2(i)]$	$i=0,1,\dots,N$
task 4( $i$ ):	$R[1(i)] \cdot R[3(i)]$	$i=0,1,\dots,N$

The average execution times of these tasks are assumed to be equal.



**Figure 6.2** Computation Control Graph for Implementation A

The computation control graph for implementation *A* is depicted in Figure 6.2. Node labeled “ $t(i)$ ” in this graph represents task  $t(i)$ ;  $t=1,2,3,4$ ;  $i=0,1,\dots,N$ . In this implementation, the algorithm is essentially the body of a DO loop, with  $Pr\{n_j=i\}$  being computed on the  $i$ -th iteration. However, the iterations are not performed sequentially but are “pipelined” -- the  $(i+1)$ -th iteration can start as soon as task  $1(i)$  of the  $i$ -th iteration has completed its execution.

Note that the arc going from node  $1(i)$  to node  $2(i)$  is not necessary for maintaining proper flow of the computation as there is no computational dependency between task  $1(i)$  and task  $2(i)$ . The reason for introducing this “artificial” dependency is to “throttle” the execution of tasks  $2(0)$ ,  $2(1)$ , ... and  $2(N)$ . As will be empirically illustrated later in this chapter, such “flow control” can improve the overall response time of the algorithm when the number of available processors is very small compared to  $N$  and *dynamic* task allocation policy is employed. The reason for the latter behavior can be inferred from the observation that the result of task  $3(i)$  is not “needed” until the result of task  $1(i)$  has been obtained (i.e., until tasks  $1(0)$ ,  $1(1)$ , ... and  $1(i)$  have been executed). Thus, “early” activation of tasks  $2(i)$  and  $3(i)$  may “impede” the execution of tasks  $1(j)$ ,  $j=0, 1,\dots,i$ , and actually delay the activation of task  $4(i)$ . The latter statements will be substantiated in the next section, when we analyze and compare the respective degrees of parallelism of implementations *A* and *B*.

### 6.2.3 Implementation B

The second implementation of the algorithm in question is described by the computation control graph illustrated in Figure 6.3. In this implementation, each task is enabled as soon as all of the operands it needs are available. The major drawback of implementation *B* is that, as will be shown below, the workload (i.e., the number of enabled tasks) offered to the system is not uniformly distributed over the

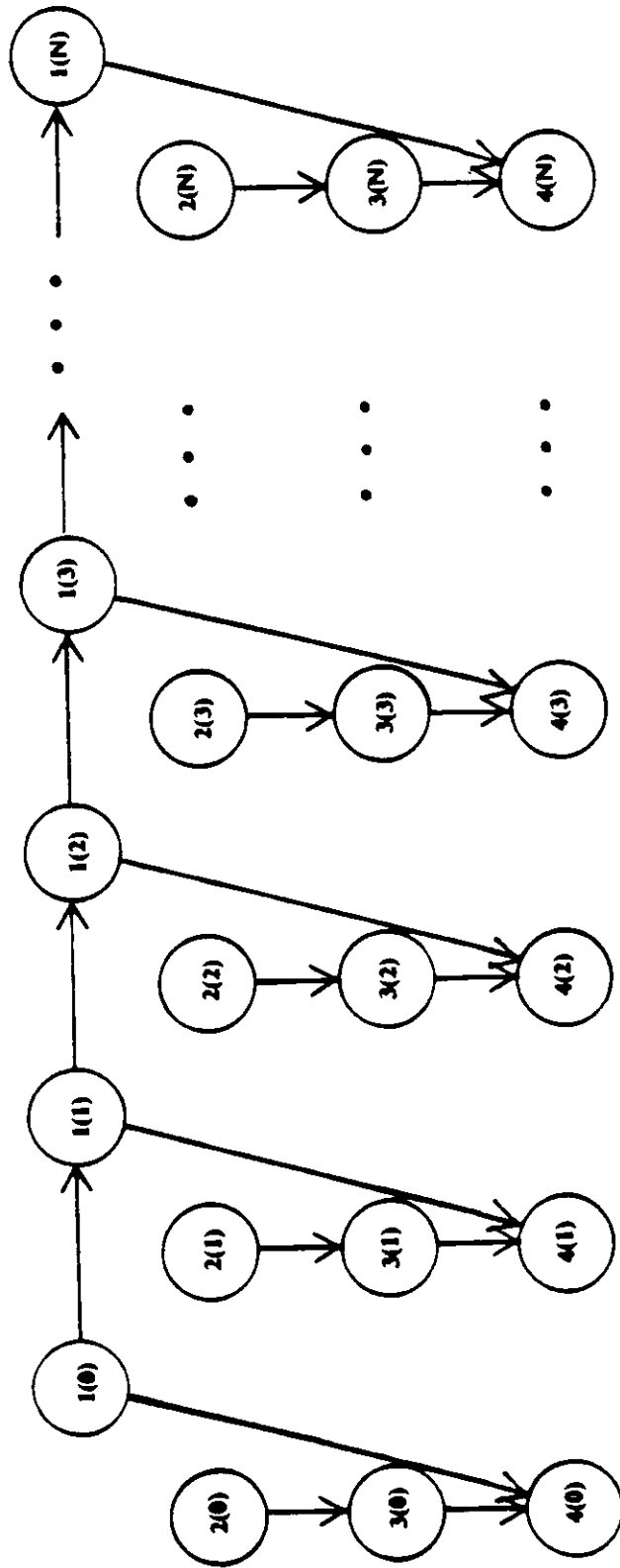
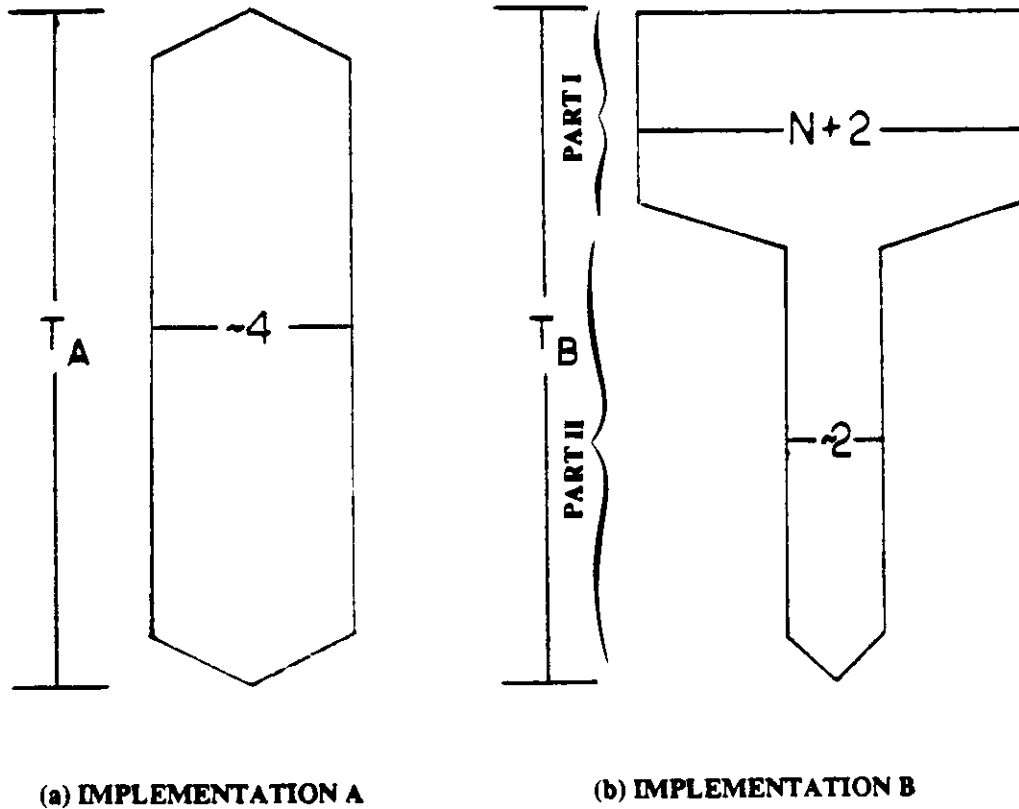


Figure 6.3 Computation Control Graph for Implementation B

algorithm's execution time period. As already discussed in the preceding section, such workload imbalance may make this implementation less attractive in systems with limited processing resources. Alternatively, if the average, combined throughput of the execution environment were to increase linearly with the number of enabled tasks (e.g., dynamic assignment of tasks to an "infinite" pool of processors), then implementation *B* would give the best average response time of all possible implementations of this algorithm.

We will now clarify and elaborate on the issues discussed above. If a dynamic task allocation policy is employed, then, since the average execution times of different tasks are the same, the characteristics of the degree of parallelism of implementation *A* (during its execution time period) are approximately represented by Figure 6.4(a) when  $N \gg 1$ . The estimate for the average width of this curve is given by the observation that tasks  $1(i+3)$ ,  $2(i+2)$ ,  $3(i+1)$  and  $4(i)$  can all, potentially, be executed concurrently, for  $i=0, \dots, N-3$ . The characteristics of the degree of parallelism of implementation *B*, for  $N \gg 1$ , are approximately represented by Figure 6.4(b). The shape of this curve is based on the following observations. Tasks  $1(0)$ ,  $2(0)$ ,  $2(1)$ , ..., and  $2(N)$  are all enabled at the start of implementation *B*. Furthermore, upon completion of task  $1(i)$ , task  $1(i+1)$  is immediately enabled and, upon completion of task  $2(i)$ , task  $3(i)$  is immediately enabled. Thus, during phase I of this implementation's execution time period, tasks  $1(i)$ ,  $X(0)$ ,  $X(1)$ , ... and  $X(N)$  can all, potentially, be executed concurrently, where each "X" is either "2" or "3." During phase II, exactly one of the "1" tasks is always enabled (except for the very end of the execution time period). Also, upon completion of task  $1(i)$ , task  $4(i)$  can be immediately enabled (since task  $3(i)$  was already completed in phase I). Thus, on the average, one of the "1" tasks and one of the "4" tasks can, potentially, be executed concurrently during phase II.



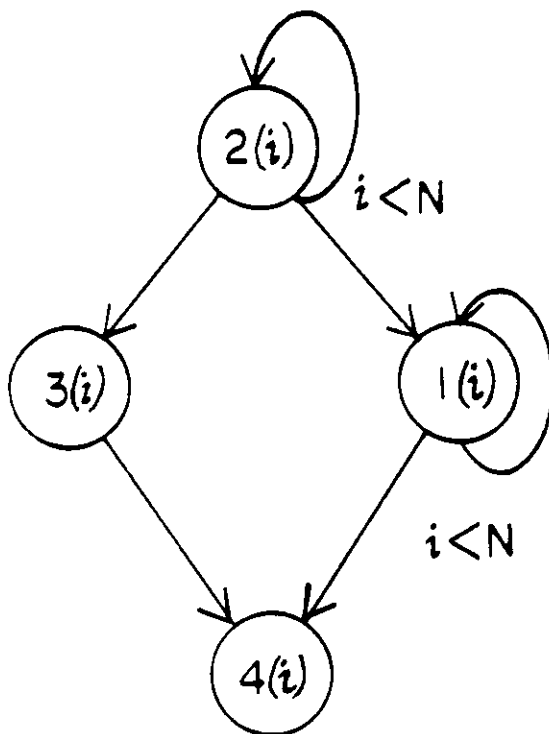
**Figure 6.4 Characteristics of the Degree of Parallelism**

Let  $C$  be the maximum capacity (in terms of the number of tasks that can be executed concurrently) of the execution environment being considered. For  $C < 2$ , implementations  $A$  and  $B$  should have almost identical average execution times, since each utilizes the full capacity of the system. When  $2 < C < 4$ , implementation  $B$  utilizes the system less during phase II of its execution than implementation  $A$  does; thus, it should give a worse performance than  $A$ . If  $C > N+1$ , then, since the system can always execute concurrently all of the enabled tasks, the “flow control” of implementation  $A$  would actually impede the execution of the algorithm and result in a mean response time larger than that of implementation  $B$ . The interesting case is when  $4 < C < N+1$ . In this case, implementation  $B$  utilizes the system more than implementation  $A$  does during phase I and less than  $A$  during phase II. Thus, there must be a constant  $C'$ ,  $4 < C' < N+1$ , such that, if  $4 < C < C'$ , then  $A$  should perform better, and, if  $C' < C < N+1$ , then  $B$  should perform better.

The analysis presented above is in concurrence with the empirical results of this case study, which are presented later in this chapter.

#### 6.2.4 Implementation C

The last implementation, which is represented by the computation control graph depicted in Figure 6.5, is analogous to implementation  $A$ , since it also includes “artificial” dependencies for “throttling” the execution of otherwise enabled tasks. However, in implementation  $C$ , it is the completion rate of tasks  $2(i)$ ,  $i=0,1,\dots,N$ , that provides flow control for the execution of other tasks. This is accomplished by having two extra arcs in the graph in addition to those which are necessary for maintaining computational integrity, namely the arcs going from node  $2(i)$  to nodes  $2(i+1)$  and  $1(i)$ .



**Figure 6.5** Computation Control Graph for Implementation C



Implementation *C* is based on the observation that, in order to enable task  $4(i)$ , for each  $i$ , tasks  $2(i)$  and  $3(i)$  must be executed in sequence, as well as task  $1(i)$ . Thus, the intent of this implementation is to give “higher priority” to task  $2(i)$ ,  $i=0,1,\dots,N$ , by delaying the execution of task  $1(i)$  until the time when task  $3(i)$  can be enabled.

### 6.3 Experimentation with Different Architectures

In this section, we will evaluate the computational efficiency of each one of the three algorithm implementations described above, using different types of distributed architectures for the execution environment. The types of architectures we have experimented with include:

- tightly-coupled processors
- loosely-coupled processors/centralized synchronization
- loosely-coupled processors/distributed synchronization

Several different system configurations have been considered within each distinct type. Descriptions of specific execution environments analyzed in this case study, along with the corresponding numerical results which have been obtained from our experiments, will be presented below. In section 6.4, the data from these experiments will be used to assess the performance of each implementation on different system types relative to that of the other implementations. All results produced analytically will be compared with those yielded by performing detailed simulations, in order to determine the accuracy level attained by our methodology.

#### 6.3.1 Tightly-Coupled System

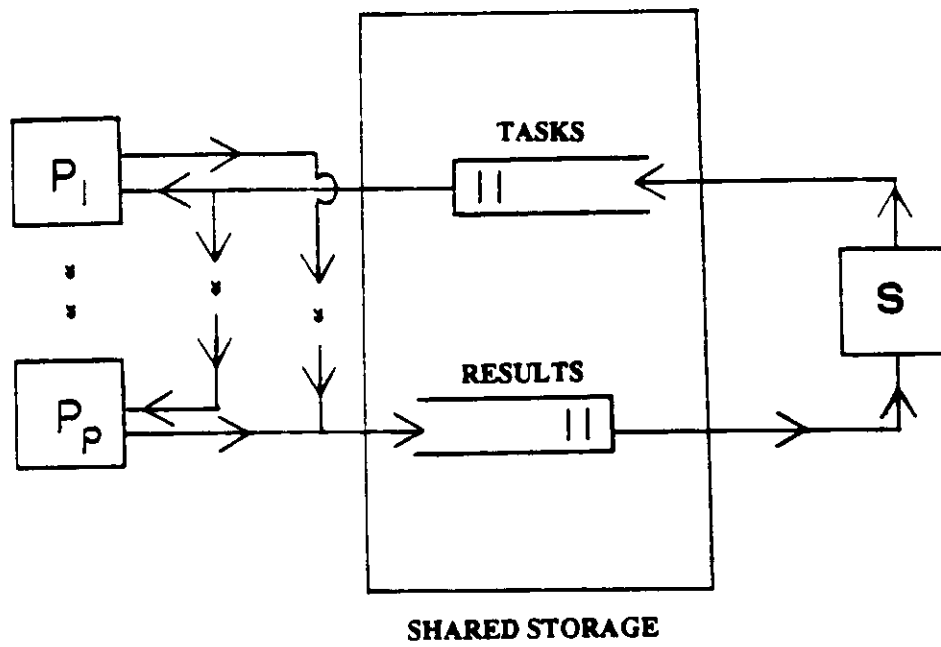
First, we will consider a very tightly-coupled architecture, where all globally accessible information is almost “instantaneously” available to each processing element in the system. After giving a description of such a system, the corresponding

physical domain model will be described, and then the numerical results will be presented.

### 6.3.1.1 System Description

An example of a tightly-coupled system is illustrated in Figure 6.6. The system shown consists of  $p+1$  processors ( $P_1, P_2, \dots, P_p$  and  $S$ ), each having its own local storage, and a high-speed, multi-ported memory module which is accessible to all processors in the system. Processors  $P_1, \dots, P_p$  are dedicated to executing tasks (i.e., performing the computations of the algorithm that is running), whereas processor  $S$  is reserved for maintaining synchronization between different tasks (i.e., matching received results into executable operand sets and enabling appropriate tasks). The global memory module is used for exchanging data between  $S$  and the "computational" processors via two externally accessible queues. All incomplete operand sets are maintained in the local storage of processor  $S$ .

Whenever a task is enabled (i.e., a complete operand set is formed), it is placed by  $S$ , along with the required operands, in the TASKS queue residing in the shared memory. Each idle processor continuously scans the TASKS queue for tasks available for execution. After an idle processor finds an enabled task, it removes that task from the TASKS queue, executes it and places the generated result(s) in the RESULTS queue, which also resides in the global storage. Processor  $S$  continuously polls the RESULTS queue for newly generated results. When a new result is available, it is dequeued, interpreted and used to update the incomplete operand sets. The delays encountered in accessing the TASKS and RESULTS queues are assumed to be negligible in comparison with task execution times and the synchronization overhead.



**Figure 6.6 Tightly-Coupled System**

### 6.3.1.2 Physical Domain Model

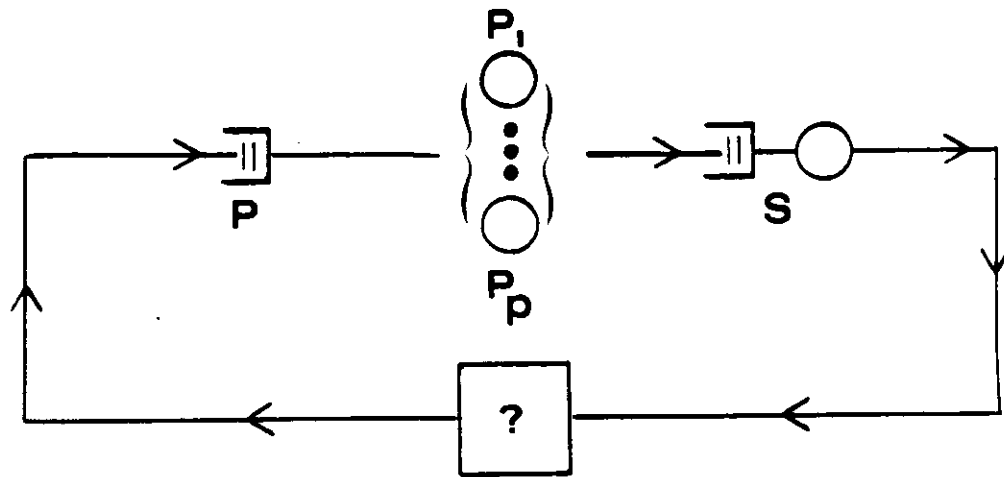
Since the system described above employs a centralized synchronization mechanism, the  $M/U$  subnetwork need not be included in its physical domain model. The  $P/C$  subnetwork for this architecture is shown in Figure 6.7. It consists of two service centers:  $P$  and  $S$ .  $P$  is a multiple-server center containing  $p$  fixed-capacity servers and is used to model the  $p$  computational processors in the system.  $S$  contains a single fixed-capacity server which represents the synchronization processor. Both  $P$  and  $S$  adhere to the FCFS queueing discipline. A customer in this queueing network is first routed to service center  $P$ , representing an enabled task, and thereafter to  $S$ , representing the generated result.

### 6.3.1.3 Numerical Results

The results of experimentation with this tightly-coupled system are presented in a tabular form in Figure 6.8. The size of each configuration considered is determined by parameter  $p$  -- the number of computational processors. Parameter  $n$  represents the number of distinct values of variable  $i$  to which the algorithm shown in Figure 6.1 was applied, i.e.,  $n=N+1$ , where  $N$  is as defined in section 6.2.1. The other system parameters were assigned the following values:

mean time to execute a task:	5 time units
mean time to process a result packet:	2 time units

As can be seen from this table, each solution derived analytically was compared with the corresponding result obtained from simulation. Relative errors were computed by assuming the simulation results to be the "actual" values. For implementation  $A$ , these relative errors seem to be insensitive to the value of  $n$ . However, they appear to first increase and then decrease as the value of  $p$  increases. The same is generally true for the implementation  $C$  errors. However, the results for implementation  $B$  indicate that the errors seem to be greater for larger values of  $n$  and



**Figure 6.7 Physical Domain Model for the Tightly-Coupled System**

		n=10			n=25		
P		ANALYTIC	SIMULATED	% REL. ERROR	ANALYTIC	SIMULATED	%REL. ERROR
A	1	210.4	216.7	-2.91	516.5	531.3	-2.79
	2	130.3	138.9	-6.19	306.1	329.7	-7.16
	3	115.2	122.5	-5.96	263.7	280.4	-5.96
	4	111.7	116.3	-3.96	252.8	270.3	-6.47
	8	111.7	116.0	-3.96	252.8	265.6	-4.82
B	1	210.7	224.0	-5.94	528.2	571.8	-7.62
	2	124.6	135.8	-8.25	313.5	341.0	-8.06
	3	112.5	117.2	-4.01	286.3	297.3	-3.70
	4	112.5	113.3	-0.71	286.3	291.3	-1.72
	8	112.5	110.5	-1.81	286.3	291.2	-1.68
C	1	208.5	214.2	-2.66	514.6	528.9	-2.70
	2	126.8	135.8	-6.63	302.7	325.5	-7.00
	3	109.9	118.1	-6.94	258.4	274.8	-5.97
	4	106.0	110.9	-4.42	247.0	264.1	-6.47
	8	106.0	110.4	-3.99	247.0	259.7	-4.89

**Figure 6.8 Mean Response Times for the Tightly-Coupled System**

that they tend to decrease as the configuration size (value of  $p$ ) increases, contrary to the other implementations. Also, overall, the relative errors for implementation  $B$  appear to be larger than those for implementations  $A$  and  $C$ . The implications of the observations made above will be discussed in Section 6.4.

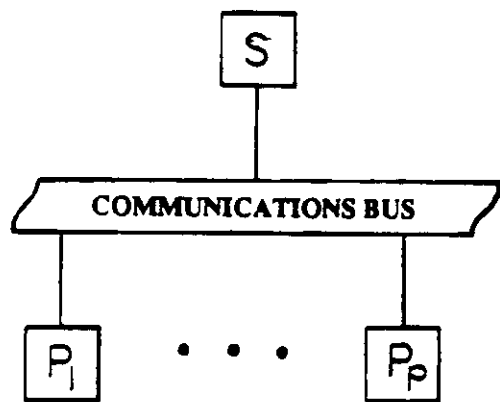
### 6.3.2 Loosely-Coupled System -- Centralized Synchronization

We will now study a distributed execution environment where processors are loosely-coupled with each other. A notable aspect of such an architecture (when compared to the tightly-coupled system considered in the preceding section) is that the communication delays incurred in interchanging data between different computing elements can significantly impact the overall system performance. After providing a system description, the corresponding physical domain model will be described, and then the numerical results will be presented.

#### 6.3.2.1 System Description

The loosely-coupled architecture considered in this section employs a centralized synchronization mechanism, i.e., all incomplete operand sets are stored at a single location. The organization of this system is depicted in Figure 6.9. There are again  $p+1$  processors, each having its own local storage:  $P_1, P_2, \dots, P_p$  are used for performing the computations;  $S$  is responsible for matching the operands (received from the other processors) into executable sets and dispatching enabled tasks.

The processors exchange information among themselves by sending data packets over the communications bus. The bus can transmit only a single packet at a time and operates in a collision avoidance mode. Whenever a task becomes enabled, processor  $S$  sends its description, along with the required operands, to one of the computational processors, the particular processor being chosen at random. After completing the execution of a task, a processor packetizes the result(s) and transmits



**Figure 6.9 Loosely-Coupled System (Centralized Synchronization)**



the packet to  $S$ , which uses it for updating the incomplete operand sets stored in its local memory.

### 6.3.2.2 Physical Domain Model

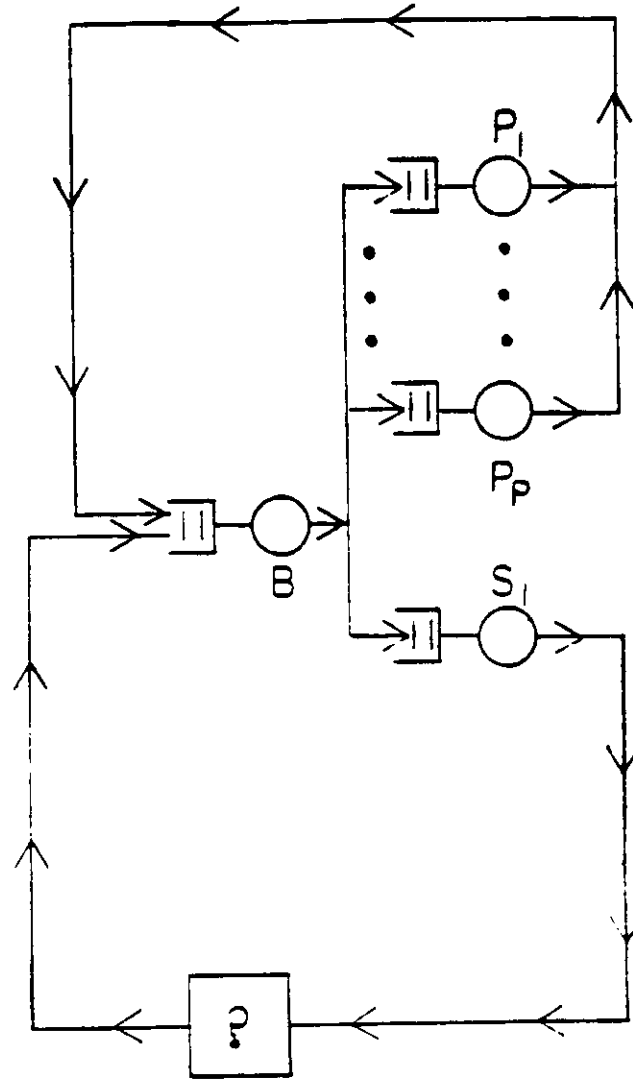
The physical domain model for this architecture is given by the queueing network shown in Figure 6.10. Note that all system elements can be included in the  $P/C$  subnetwork, as a centralized synchronization scheme is utilized. The model consists of  $p+2$  service centers. All centers are of a fixed-capacity variety and each employs the FCFS scheduling policy. Service centers  $P_1$  through  $P_p$  correspond to the computational processors;  $S$  represents the synchronization processor; and  $B$  models the communications bus. Representing an enabled task, a customer in this queueing network is first routed to service center  $B$ , followed by  $P_i$ , the value of  $i$  being chosen at random. Thereafter, modeling the generated result packet, it returns back to server  $B$  and, finally, as an operand packet, it visits  $S$ .

### 6.3.2.3 Numerical Results

The results of experimentation with the loosely-coupled system described above are given by the table shown in Figure 6.11. The size of each configuration considered is determined by parameter  $p$  -- the number of computational processors. Parameter  $n$  represents the number of different points at which the algorithm shown in Figure 6.1 was evaluated, i.e.,  $n=N+1$ , where  $N$  is as defined in section 6.2.1. The other system parameters were assigned the following values:

mean time to execute a task:	5 time units
mean time to process a result packet:	2 time units
mean time to transmit a task on the bus:	1 time unit
mean time to transmit a result packet:	1 time unit

As shown in Figure 6.11, each solution derived analytically had been compared with



**Figure 6.10 Physical Domain Model for the Loosely-Coupled System (Centralized Synchronization)**

P	n=10			n=25			
	ANALYTIC	SIMULATED	% REL. ERROR	ANALYTIC	SIMULATED	%REL. ERROR	
A	1	223.4	236.0	-5.34	535.5	563.4	-4.95
	2	179.9	191.9	-6.25	419.2	445.7	-5.95
	3	167.4	179.0	-6.48	385.9	413.4	-6.65
	4	161.0	171.5	-6.12	368.9	395.1	-6.63
	8	153.2	162.5	-5.72	348.5	373.1	-6.59
B	1	225.5	241.7	-6.70	567.3	619.5	-8.43
	2	167.2	180.9	-7.57	409.8	450.9	-9.12
	3	153.4	166.1	-7.65	373.9	417.8	-10.51
	4	148.6	160.7	-7.53	367.0	410.7	-10.64
	8	143.5	154.3	-6.99	358.4	399.2	-10.22
C	1	218.6	230.5	-5.16	530.8	555.8	-4.50
	2	173.1	184.7	-6.28	412.4	435.9	-5.39
	3	160.1	174.1	-8.04	378.7	406.1	-6.75
	4	153.4	163.2	-6.00	361.4	386.7	-6.54
	8	144.4	156.1	-6.85	340.7	368.0	-7.42

**Figure 6.11**  
**Mean Response Times for the Loosely-Coupled System**  
**(Centralized Synchronization)**

the corresponding result obtained from simulation. Relative errors had been computed by assuming the simulation results to be the "actual" values. For implementation *A*, these relative errors seem to be insensitive to the value of  $n$ ; however, they appear to increase as the value of  $p$  increases. Overall, the same is true for the implementation *C* errors. In contrast, the results for implementation *B* indicate that the errors seem to be greater for larger values of  $n$  and that they do not change much with different configuration sizes. Also, in general, the relative errors for implementation *B* appear to be larger than those for implementations *A* and *C*. The implications of the observations made above will be analyzed in section 6.4.

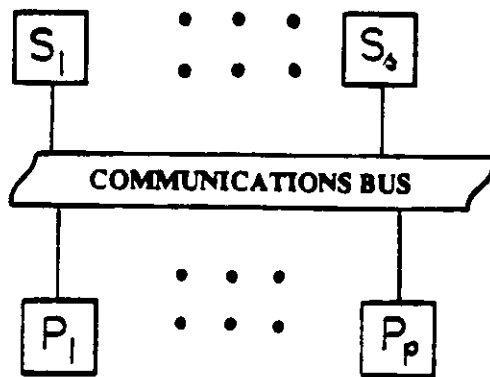
### 6.3.3 Loosely-Coupled System -- Distributed Synchronization

In this section, we will also evaluate a loosely-coupled architecture, but one which employs a distributed synchronization mechanism. After providing a system description, the corresponding physical domain model will be described, and then the numerical results will be presented.

#### 6.3.2.1 System Description

The system being analyzed here is illustrated diagrammatically in Figure 6.12. Its architecture is similar to the one considered in the preceding section -- the difference being that, in this system, several processors are used for maintaining intertask synchronization (as opposed to just one). There are  $s$  synchronization processors:  $S_1, \dots, S_s$ . Each one is responsible for maintaining operand sets for a particular group of tasks.

Whenever the execution of a task is completed, the generated result packet is broadcast over the communications bus to all of the synchronization processors. However, only those processors which can use the corresponding operand(s) actual-



**Figure 6.12 Loosely-Coupled System (Distributed Synchronization)**

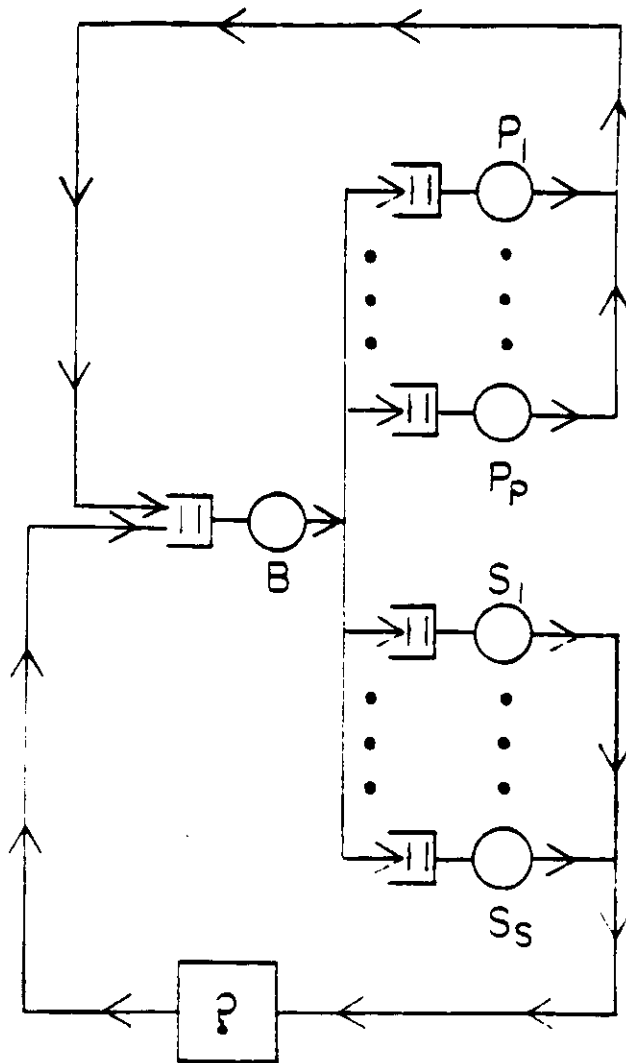
ly accept this result packet -- the others simply discard it. We will assume that the time needed for determining whether or not to accept a result packet is negligible compared to the operand matching overhead. For the purposes of this case study, it will also be assumed that each result packet is accepted by only one synchronization processor and that the maintenance of operand sets is evenly distributed among all processors (i.e., on the average, each processor accepts the same number of result packets).

### 6.3.3.2 Physical Domain Model

The physical domain model for this architecture is given by the queueing network shown in Figure 6.13. The model consists of  $p+s+1$  service centers. All centers are of a fixed-capacity variety and each employs the FCFS scheduling policy. Service centers  $P_1$  through  $P_p$  correspond to the computational processors;  $S_1$  through  $S_s$  represent the synchronization processors; and  $B$  models the communications bus. Representing an enabled task, a customer in this queueing network is first routed to service center  $B$ , followed by  $P_i$ , the value of  $i$  being chosen at random. Thereafter, modeling the generated result packet, it returns back to server  $B$  and, finally, as an operand packet, it visits  $S_j$ ,  $j$  being randomly chosen (since the task synchronization work load is evenly balanced among all synchronization processors).

### 6.3.3.3 Numerical Results

The table in Figure 6.14 presents the numerical results for the distributed synchronization system discussed above. The size of each configuration considered is determined by parameter  $p$  -- the number of computational processors. Parameter  $n$  represents the number of iterations of the algorithm being studied. The other system parameters were assigned the following values:



**Figure 6.13**  
**Physical Domain Model for Loosely-Coupled System (Distributed Synchronization)**

		n=10			n=25		
P		ANALYTIC	SIMULATED	% REL. ERROR	ANALYTIC	SIMULATED	% REL. ERROR
A	3	160.6	175.0	-8.23	368.3	399.2	-7.74
	4	154.3	166.1	-7.10	350.7	379.5	-7.59
	6	146.2	160.0	-8.63	329.7	364.3	-9.50
	8	143.0	153.8	-7.02	321.2	347.2	-7.49
B	3	147.0	159.2	-7.66	362.5	403.7	-10.21
	4	141.7	151.5	-6.47	354.8	393.4	-9.81
	6	136.7	145.2	-5.85	348.6	384.9	-9.43
	8	135.0	142.8	-5.46	344.1	381.1	-9.71
C	3	153.2	166.2	-7.82	360.8	389.9	-7.46
	4	146.5	159.4	-8.09	342.9	373.1	-8.09
	6	138.1	150.8	-8.42	321.5	360.0	-10.69
	8	134.9	145.3	-7.16	313.1	337.7	-7.28

**Figure 6.14**  
**Mean Response Times for the Loosely-Coupled System**  
**(Distributed Synchronization)**



mean time to execute a task:	5 time units
mean time to process a result packet:	2 time units
mean time to transmit a task on the bus:	1 time unit
mean time to transmit a result packet:	1 time unit
number of synchronization processors, $s$ :	$\text{CEIL}(p/2)$

As the table indicates, each solution derived analytically was compared with the corresponding result obtained from simulation. Relative errors were computed by assuming the simulation results to be the “true” values. For implementation *A*, these relative errors seem to be insensitive to the value of  $n$ ; also, they do not appear to be correlated with the value of  $p$ . In general, the latter statement also holds for the implementation *C* errors. The results for implementation *B*, however, indicate that the errors seem to be greater for larger values of  $n$  and that they do not vary significantly with different configuration sizes. Furthermore, at larger values of  $n$ , the relative errors for implementation *B* appear to be greater than those for implementations *A* and *C*. The implications of the observations made above will be discussed in the following section.

#### 6.4 Discussion of Results

We will now analyze the combined results for all three distributed architectures studied in this chapter and discuss what conclusions may be drawn from the collected data.

Irrespective of the architecture being considered, the accuracy level of the analytic results produced by our methodology appears to be better for implementations *A* and *C* than for implementation *B*, especially for larger values of  $n$ . This observation can be attributable to the following effect. The respective degrees of parallelism for implementations *A* and *C* remain almost uniform throughout their respective execution time periods. The larger the value  $n$  is (i.e., the longer the algorithm’s

execution time), the more uniform the aforementioned degrees of parallelism are. However, since implementation  $B$  does not have any “flow control” to throttle the activation of tasks, its degree of parallelism during the earlier stages of its execution is greater than that during the later stages of its execution. As the number of tasks, which are enabled at the beginning of implementation  $B$ ’s execution, is directly proportional to  $n$ , the greater  $n$  is, the larger is the degree of parallelism during the first portion of the execution time period. In other words, the larger the value of  $n$ , the greater is the disparity between degrees of parallelism during different execution stages and, thus, the greater is the variance of the distribution of parallelism of implementation  $B$ . Since most of our approximation techniques are based on “fluid flow” assumptions, in general, the larger the variance of a program’s distribution of parallelism, the less accurate is our analytic solution for its execution time.

The data gathered in this study indicates that, for all three implementations, the relative errors in our analytic results in the case of the tightly-coupled architecture are, overall, fewer than those for either of the two loosely-coupled systems. A probable explanation for the latter trend is that the global behavior of the tightly-coupled system is closer to the characteristics of a single state-dependent, exponential service center than the respective behavior of the loosely-coupled architectures. The particular characteristics that are important to our solution process are: (1) the lack of correlation between the order in which customers (of the same class) arrive and the order in which they depart; and (2) the lack of correlation between the time that a customer has already spent in the system and its remaining time in the system. Recall that, as a part of our solution procedure, the physical domain model is aggregated into a single state-dependent, exponential service center, which utilizes the PS queueing discipline within each customer class. Thus, the greater the correlation, in the “original” physical domain model, between the *arrival* times of customers (belonging to the same chain) and *departure* times of those customers, the greater is the potential error in our analytic solution.

Another important empirical observation is that virtually all analytic solutions are underestimates of the corresponding simulation results. This observation can be theoretically substantiated by the fact that the segment aggregation techniques used in our solution process rely on “fluid flow” approximations to the dynamics of segments’ behavior. Since (as the name implies) such “fluid flow” assumptions do not explicitly consider the time-dependent variations in a segment’s behavior, they will generally result in underestimating the actual response times.

As far as the relative performance of the three implementations is concerned, the following observations can be made. The response times of implementations *A* and *C* are very close to each other for all of the different types of architectures and configuration sizes used in the experiments. However, implementation *C* consistently produced slightly better response times than implementation *A*. Thus, in terms of performance, implementation *C* should generally be preferable to *A*, regardless of the execution environment. The performance of implementation *B*, relative to the performance of the other implementations, appears to depend on both the configuration size and the number of iterations of the algorithm. The larger the value of  $p$ , the better is its relative performance. The larger the value of  $n$ , the worse is its relative performance. Also, in general, the relative performance of implementation *B* is better on both loosely-coupled systems than on the tightly-coupled one. A conclusion which may be drawn from the latter observations is: the larger the ratio of  $n$  to the number of enabled tasks at which the system saturates, the worse is the relative performance of implementation *B*.

## CHAPTER 7

### CASE STUDY B: SIGNAL PROCESSING APPLICATION

In this chapter, we will present an assessment of the merits of our methodology when used in modeling the execution of a real-time, signal processing application. The results obtained from a previous analysis of the same application will be used to help evaluate the viability of our approach. We will consider two types of distributed execution environment for this application: (1) a hypothetical environment with a simple structure and (2) a realistic system having a sophisticated architecture. The performance measures for the execution of the application in the simple environment have already been obtained in a previous research study [CHU84] and will be used to establish confidence in the results produced by employing our methodology. In the second experiment, the analytic results for a more complex architecture will be compared with those obtained from running a detailed simulation of the program execution in that environment. The purpose of the second experiment is to illustrate the power of our methodology in dealing with complex system architectures.

#### 7.1 Application Description and Analysis

The application proposed for this case study is a real-time, signal processing program (process). This process is being periodically executed, after random time intervals, as invoked by “arrivals” of new sets of input signals. Such type of behavior is typical of on-line radar monitor and control systems. The particular process,  $P$ , we have chosen is the one considered in the performance modeling study by Chu and Leung [CHU84]. The computation control graph for this process is shown

in Figure 7.1. Note that this graph features all of the different segment combinations analyzed in Chapter 5: sequential, EXCLUSIVE-OR, parallel and looping. These structural properties will be fully exploited to minimize the computational cost of our solution process.

The analysis of the behavior of this application, in each of the two environments being considered, will be accomplished in two phases. In the first phase, we will obtain the performance measures for the execution of a single instance of process  $P$ , when it is the only program running on the system being modeled. In the second phase, the behavior of the process in the case of random arrivals of  $P$  will be analyzed using the results of the first phase.

### 7.1.1 Single Instance of Process $P$

As can be seen from its computation control graph, the process being considered has a well structured form and can be conveniently decomposed into a hierarchical combination of segments. This hierarchical decomposition is illustrated in Figure 7.2, where process  $P$  is represented at seven different levels of abstraction. We will now discuss each level of the process' hierarchy in detail. On level 0, segment  $P$  represents the whole process. On level 1, segment  $P$  is revealed to be the sequential combination of task 1, segment  $A$ , and task 15. On the second level, segment  $A$  is detailed as a simple loop, having segment  $B$  as its body. The third level allows us to view segment  $B$  as the sequential combination of task 2, segment  $C$ , and task 14. Level 4 reveals that segment  $C$  is the EXCLUSIVE-OR combination of segments  $D$ ,  $E$  and  $F$ . On the fifth level, we find that: segment  $D$  is the sequential combination of tasks 3 and 12; segment  $E$  is the sequential combination of tasks 4, 6, and 9; and segment  $F$  is the sequential combination of task 5, segment  $G$ , and task 13. Finally, on level 6, segment  $G$  is represented by the computation control graph shown in Figure 7.2(g).

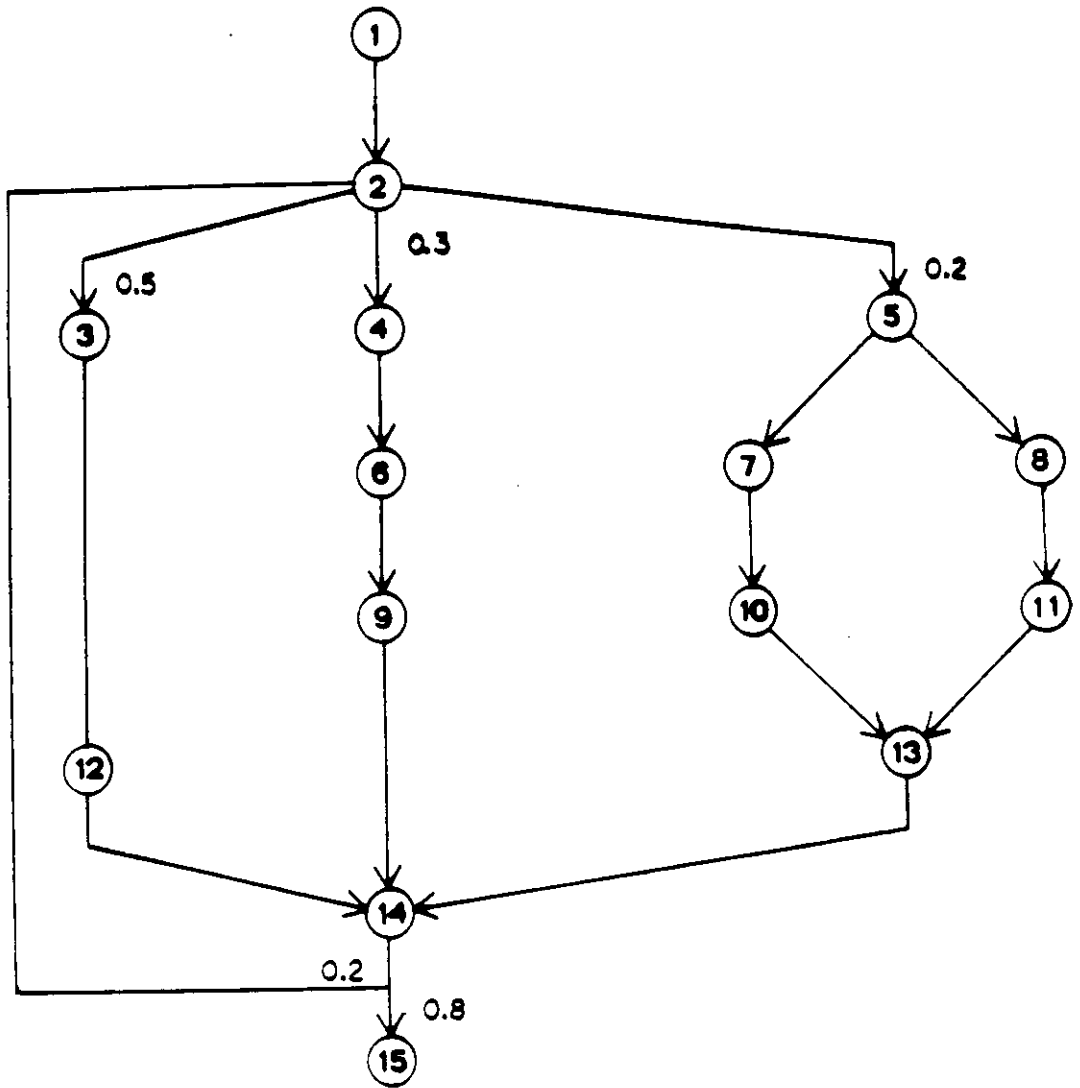


Figure 7.1 Computation Control Graph for Process P



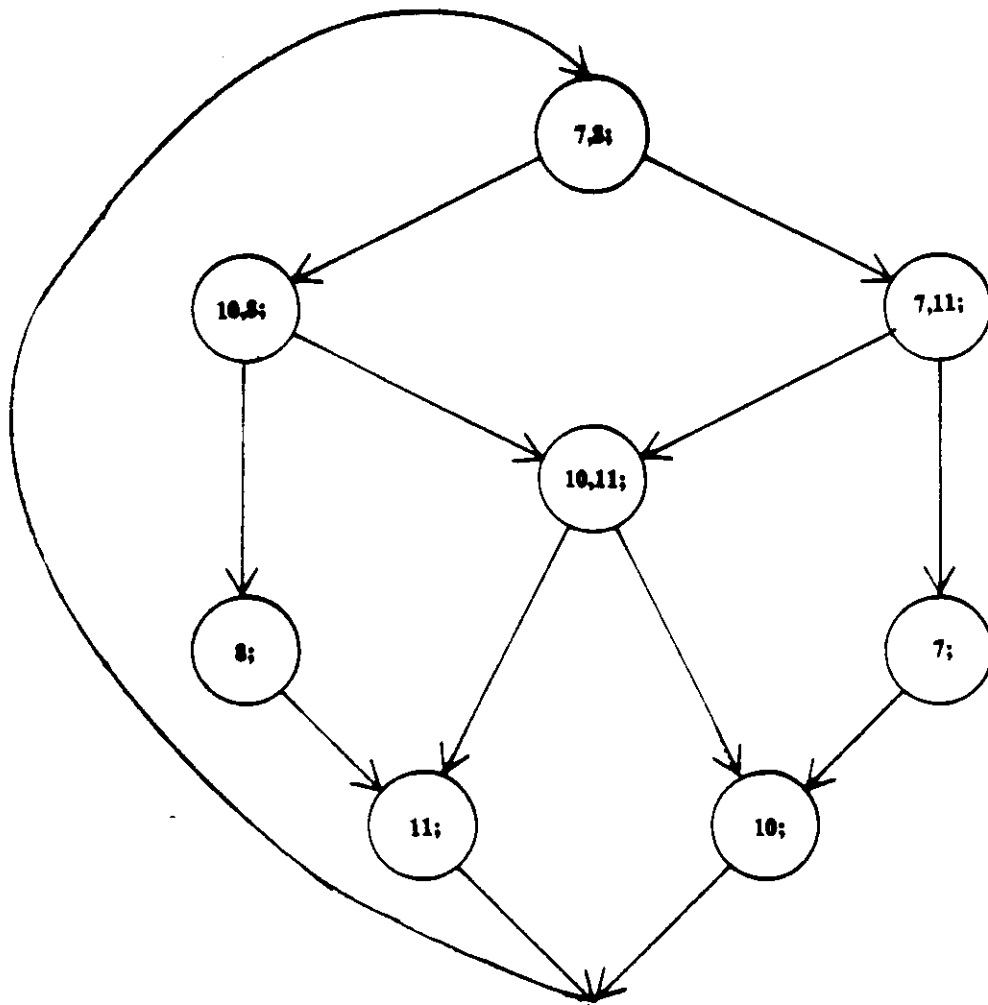
The hierarchical decomposition of the process' computation control graph described above permits us to employ the segment combination techniques developed in Chapter 5 to efficiently obtain the desired performance measures (i.e. the segment descriptor of the whole process  $P$ ). The computation of segment  $G$ 's descriptor involves constructing and solving the Markov process depicted in Figure 7.3, according to the procedure presented in Chapter 4. Descriptors of segments  $D$ ,  $E$  and  $F$  are found by applying the sequential combination algorithms. Segment  $C$  is solved using the EXCLUSIVE-OR combination techniques. Proceeding in the same fashion, we can compute the descriptors of segments  $B$ ,  $A$  and, finally,  $P$  itself. Since only sequential and EXCLUSIVE-OR segment aggregation techniques are employed, the solution approach described above is applicable even if tasks of process  $P$  belong to different classes.

Thus, at the end of the first phase, we have the values for the mean number of tasks (of each class, if applicable) executed during an invocation of process  $P$ , the average execution time of a single instance of  $P$ , and its distribution of parallelism. Having obtained these results, we are now ready to commence the second stage of our analysis.

### 7.1.2 Random Process Arrivals

In this phase of our analysis, we will utilize the procedure presented in section 5.6.3. Let  $P(n)$  denote a *parallel* combination of  $n$  instances of process  $P$ . According to the aforementioned procedure, the whole execution environment is represented by a single M/G/1 service center with state-dependent service rates, where each arriving job represents an instance of process  $P$ . In order to obtain the service capacity function (or set of departure rates) for this M/G/1 queue, we have to compute  $T_{P(n)}$  for  $n = 1, 2, \dots, I$ . If there is only one class of tasks, then the parallel combination procedure presented in section 5.4 can be directly applied to estimate





**Figure 7.3 Markov Process for Segment G**

the latter quantities. In the case that tasks of process  $P$  belong to different classes, there are two possible solution approaches. One way is to construct and solve a Markov process for each of the  $I-1$  parallel combinations,  $P(2), \dots, P(I)$ . This can be very expensive computationally, especially for large  $I$ . The other way is to first use the heuristics described in section 5.6.2 to combine the different task classes into a single, aggregate class and then apply the approximation procedure of section 5.4. The latter approach will be utilized in this case study.

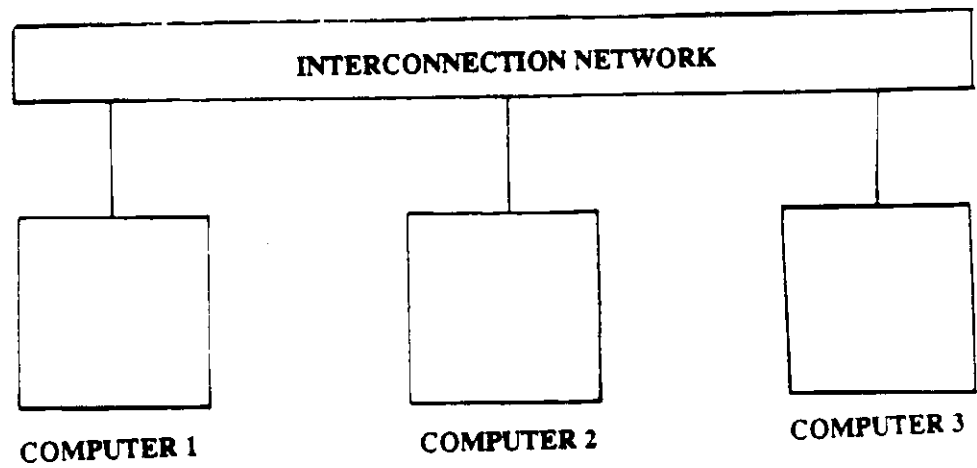
After obtaining the service capacity function, we can simply apply the already existing formulas for solving an M/G/1 queue with state-dependent service rates [LAV82] to obtain the desired performance measures and complete our evaluation of this application. The numerical values of the performance measures will, of course, depend on the particular values of the parameters  $T_{P(n)}$ ,  $n = 1, \dots, I$ , which, in turn, depend on the specific execution environment being considered.

## **7.2 Evaluation of a Simple System**

In this section, we will consider the simple, hypothetical environment proposed by Chu and Leung in the aforementioned study [CHU84]. After describing their system, we will present the physical domain model for it, and then compare our numerical results with those obtained in that study. This experiment will serve as a benchmark test of the accuracy of our methodology.

### **7.2.1 System Description**

The execution environment in question is depicted in Figure 7.4. It consists of three identical computers, which communicate with each other through some kind of interconnection network. Each task of process  $P$  is a priori assigned to one of the computers to be executed. Whenever a task becomes enabled, its execution begins on the computer allocated to it. Upon completion, it generates and sends a "mes-



**Figure 7.4 Simple Execution Environment**

sage” to the task(s) which must be informed accordingly. Message passing between tasks residing on the same computer occurs instantaneously. However, if a message has to be transmitted over the interconnection network, it incurs a constant communications delay (of 0.2 seconds in the model) -- queueing delays are ignored. It is also assumed that the task synchronization overhead is nonexistent, i.e. it takes zero time to update incomplete operand sets and enable tasks. The table shown in Figure 7.5 lists, for each task, its average execution time and the computer it is allocated to. All task service times are assumed to be exponentially distributed.

### 7.2.2 Physical Domain Model

The system described above belongs to the “Static Allocation with Distributed Synchronization” category. However, since the task synchronization delays are ignored, there is no need to include the *M/U* subnetwork and Black Box *A* in our physical domain model. The *P/C* subnetwork for this system is illustrated in Figure 7.6. It consists of four service centers: *C*1, *C*2, *C*3 and *IN*. Service center *C*<sub>*i*</sub>, *i*=1,2,3 represents the time spent by a task in queueing for and executing on Computer *i*. Each employs the FCFS scheduling discipline. *IN* represents the transmission delay experienced by messages on the interconnection network. Since this delay is assumed to be constant, *IN* is an Infinite Servers (IS) service center.

Each queueing network customer visits exactly one of *C*1, *C*2 and *C*3 service centers, the choice being dependent on which task that customer is representing. If, upon completing its execution, the corresponding task has to send its result to some task(s) residing on (an)other computer(s), i.e., a message has to be transmitted over the interconnection network (a single message may be destined to multiple locations), the same customer also visits the *IN* service center. Thus, a given queueing network customer represents both a task and a message it generates (if any).

TASK #	AVERAGE EXECUTION TIME (SEC.)	COMPUTER ALLOCATED TO
1	1	1
2	1	2
3	1	3
4	1	2
5	1	1
6	2	3
7	2	1
8	2	2
9	2	1
10	2	3
11	3	2
12	3	1
13	3	1
14	3	3
15	3	2

**Figure 7.5 Task Characteristics for the Simple System**

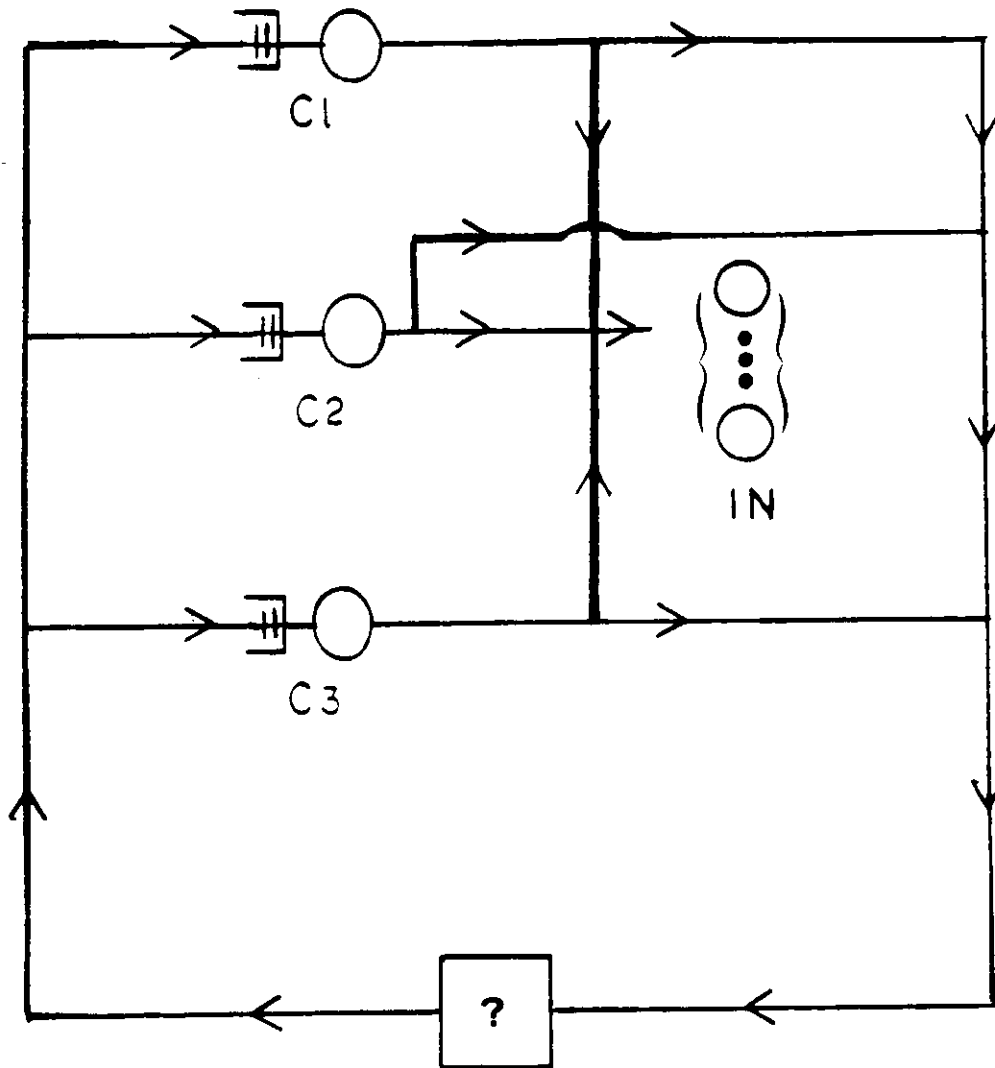


Figure 7.6 Physical Domain Model for the Simple System

### 7.2.3 Numerical Comparisons

The average individual processing loads on the three computers of this system are 5.125 sec., 5.875 sec., and 5.625 sec., respectively, per process invocation [CHU84]. Thus, the bottleneck system resource is Computer 2 and the maximum attainable "capacity" of the system is  $1/5.875 = 0.1702$  processes/sec. The values for  $I$ ,  $T_{P(n)}$ ,  $n = 1, \dots, I$ , and the capacity function for the M/G/1 queue are listed in the table shown in Figure 7.7.

The solutions for mean process response times under various process arrival rates are given by the table of Figure 7.8. This table also compares our analytic results with those obtained by Chu and Leung in their study. Relative errors have been computed by assuming the previous results to be the "actual" values. As can be seen from this comparison, the results produced by our methodology closely correspond to the previously obtained analytic solution.

We will now assess the computational complexity of obtaining analytic solution for this system architecture using our methodology. Since the queueing network representing the physical domain model is very simple and saturates quickly, the amount of time required for the computation of the task throughput rates is negligible. Approximately 600 arithmetic operations (i.e., +, -, ×, or /) are needed in order to obtain the capacity function for the aforementioned M/G/1 queue. Also, about 60 arithmetic operations must be performed for each arrival rate considered in order to compute the process response times. Even if all of the computations were performed on a microcomputer, the total time (including memory access overhead) required to produce the analytic results shown in Figure 7.8 would still be well below 1 sec. Thus, in evaluating this simple architecture, our methodology should be at least as "fast" as the analytic techniques employed in [CHU84], based on the information provided in that paper.

$i$	$T_{P(i)}$	$Y_{(i)}$
1	17.61	1
2	22.18	1.588
3	27.22	1.941
4	32.64	2.158
5	38.40	2.293
6	44.40	2.380
7	50.60	2.436
8	56.94	2.474
9	63.37	2.501
10	69.87	2.520

**Figure 7.7 Capacity Function for the Simple System**



ARRIVAL RATE (processes/sec.)	MEAN RESPONSE TIME (SEC.)		RELATIVE ERROR (percent)
	OUR ANALYTIC	PREVIOUS RESULTS	
0.00	17.610	17.50	+0.63
0.04	21.737	23.0	-5.49
0.08	29.988	32.5	-7.73
0.10	38.175	41.0	-6.89

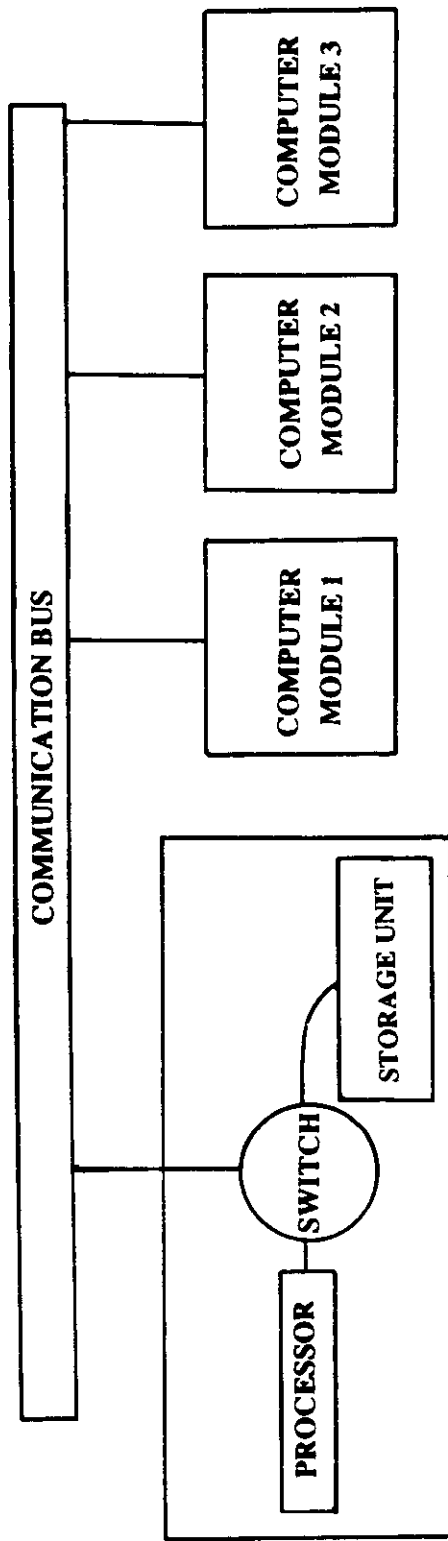
**Figure 7.8 Comparison of Results for the Simple System**

## **7.3 Evaluation of a Sophisticated Architecture**

In this section, we will analyze the performance of the application in question when it is executed in a realistic, sophisticated environment. The architecture of the proposed system is a loosely-coupled, multiple computer configuration, based on the Cm\* multiprocessor [DEM82]. The model of this system and its solution will illustrate the advantages of our methodology when considering the effects of contention for communication resources, memory access delays, and task synchronization overhead. We will first give a detailed description of this architecture, then describe its physical domain model and, finally, compare our analytic results with those obtained from simulation.

### **7.3.1 System Description**

The architecture of the distributed system we are evaluating is shown in Figure 7.9. The basic component of this architecture is a Computer Module. Four such components are present in our system, being interconnected through a communication bus. Each Computer Module consists of a processor, an intelligent switch, and a storage unit with the associated controller. All memory references made by a processor are sent to and interpreted by the switch of that Computer Module. All local (with respect to the Computer Module) requests are forwarded directly to the attached storage unit controller. Whenever a remote memory request is issued, it is intercepted by the local switch, converted into a request to the switch of the proper Computer Module, and transmitted as a data packet over the communication bus. Upon receiving this request, the remote switch converts it into the proper memory reference to its local storage. If data is to be returned to the requesting switch, it is packetized and sent back via the bus. Remote memory requests are also used by processors to exchange messages among themselves.



COMPUTER MODULE 0  
 (COMPUTER MODULES 1, 2 & 3 HAVE THE SAME ARCHITECTURE AS COMPUTER MODULE 0.)

Figure 7.9 Complex Execution Environment

In this configuration, Computer Module 0 is designated to perform the synchronization of tasks, while modules 1, 2 and 3 are dedicated to executing enabled tasks. All of the incomplete operand sets are maintained in the local memory of Computer Module 0. When an operand packet is received, it is first stored in memory by the switch and then the processor is notified, which updates the affected operand sets. The tasks corresponding to the operand sets which were completed are enabled and each is sent to either module 1, 2 or 3 for execution, the particular module being chosen at random. When Computer Module  $i$ ,  $i=1,2,3$  receives a task from Computer Module 0, it first acquires the necessary data from its local storage unit, then executes the task code, and, finally, generates a result packet (which also serves as an operand packet) and returns it to Computer Module 0.

The timings of various operations are given by the table shown in Figure 7.10. All service times are given as means of the exponential distribution and do not include any associated queuing delays. All execution of tasks, storage access requests, and transmission of data packets are performed in the order received. Since each resource has a finite, fixed capacity, there may be contention for any one of the system's resources.

### 7.3.2 Physical Domain Model

The system described above belongs to the "Dynamic Allocation with Centralized Synchronization" category. The  $P/C$  subnetwork of the physical domain model is depicted in Figure 7.11. It consists of four "clusters" of service centers,  $CM_0$ ,  $CM_1$ ,  $CM_2$ ,  $CM_3$ , and of service center  $BUS$ . Cluster  $CM_i$ ,  $i=0,1,2,3$  which represent Cluster  $CM_i$ ,  $i=0,1,2,3$  which represents the resources of Computer Module  $i$ , consists of service centers  $P_i$ ,  $M_i$  and  $S_i$ .  $P_0$  models the time needed to process a received operand packet.  $M_0$  represents storage access delays incurred in updating incomplete operand sets. Service center  $S_0$  models processing of local

OPERATION	AVERAGE TIME (SEC.)
EXECUTION OF A TASK	2
STORAGE ACCESS	0.5
MATCHING OF OPERANDS	0.2
TASK TRANSMISSION	0.2
RESULT PACKET TRANSMISSION	0.2
SWITCH PROCESSING	0.1

**Figure 7.10** Timings for the Complex System

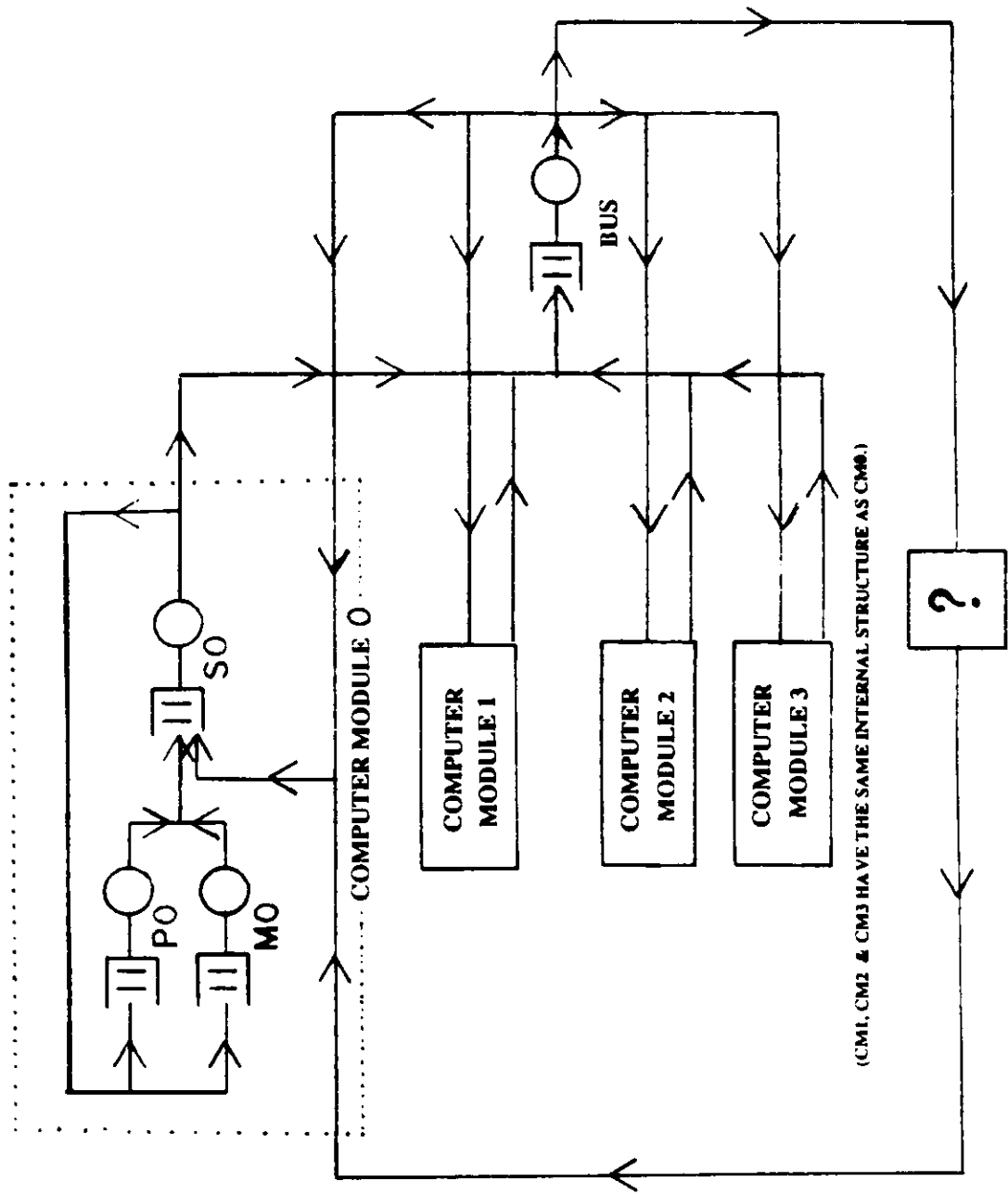
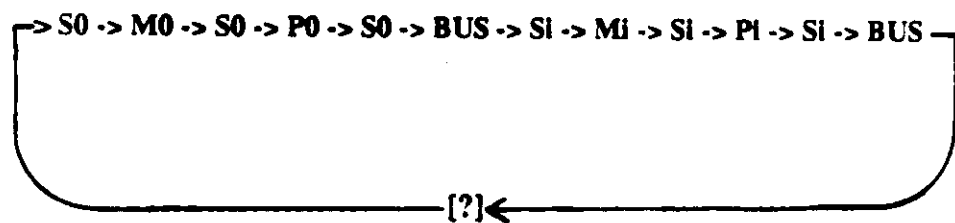


Figure 7.11 Physical Domain Model for the Complex System

packets from the communications bus, and forwarding enabled tasks to the bus for transmission to other computer modules. Together, service centers  $P_0$ ,  $M_0$  and  $S_0$  represent the task synchronization overhead of our system.  $P_i$ ,  $i=1,2,3$  models the execution and waiting times of a task at Computer Module  $i$ . Delays at server  $M_i$ ,  $i=1,2,3$  correspond to fetching the data, which is needed to execute a received task, from the local storage unit of module  $i$ .  $S_i$  represents receiving of tasks (sent to module  $i$  by module 0) from the communications bus, processing of storage access requests, and transferring of result packets (generated by completed tasks) to the bus for transmission to Computer Module 0. Service center  $BUS$  represents the contention for the communications bus by and the transmission time of tasks and result packets.

Each queueing network customer in this  $P/C$  subnetwork represents either a task, a result packet, an operand packet, or a storage access request, depending on the stage of its "lifetime." The route of a customer through the  $P/C$  subnetwork is schematically presented below.



We shall now explain the routing behavior given above in detail. A customer first visits service centers  $S_0$  and  $M_0$ , representing a storage access request. It then proceeds to  $S_0$  and  $P_0$ , while modeling an operand packet. Afterwards, representing the transmission of an enabled task to Computer Module  $i$  ( $i$  is chosen randomly

as either 1, 2 or 3) it again visits server  $S_0$ , followed by  $BUS$  and  $S_i$ . As a local memory reference, it proceeds to  $M_i$  and then back to  $S_i$ . During the subsequent visit to service center  $P_i$ , it models the task submitted for execution. Finally, representing a result packet, it returns to  $S_i$  and then completes its route at  $BUS$  service center. Different customer classes are used in this queueing network in order for customers to be able to make proper routing decisions.

### 7.3.3 Numerical Comparisons

The bottleneck operation in this system is the actual execution of tasks by processors. Since the mean task execution time is 2 sec. and there are three processors available for executing tasks, the maximum achievable system throughput is 1.5 tasks/sec. The average number of tasks executed during a single instance of the process being modeled is 8.375. Thus, the maximum "capacity" of our system is  $1.5/8.375 = 0.1791$  processes/sec. The values for  $I$ ,  $T_{P(n)}$ ,  $n=1, \dots, I$ , and the capacity function for the M/G/1 queue are listed in the table shown in Figure 7.12.

The solutions for mean process response times under various process arrival rates are given in the table of Figure 7.13. This table also compares our analytic results with those obtained by performing a detailed simulation of this execution environment using the RESQ package. Relative errors have been computed by assuming the simulation results to be the "actual" values. As can be seen from this comparison, the results produced by our methodology match very closely the ones yielded by simulation.

We will now assess the computational complexity of obtaining analytic solution for this system architecture. If the Mean Value Analysis method is used to solve the queueing network representing the physical domain model, then the computation of the task throughput rates requires at most 1000 arithmetic (i.e., +, -,  $\times$  or /) operations. Approximately 500 arithmetic operations are needed in order to



i	$T_{P(i)}$	Y(i)
1	33.639	1
2	37.930	1.774
3	42.500	2.375
4	46.752	2.878
5	51.627	3.258
6	55.554	3.633
7	62.113	3.791
8	65.923	4.082
9	71.991	4.205
10	75.964	4.428
11	79.168	4.674
12	83.109	4.857
13	88.883	4.920
14	93.628	5.030
15	98.145	5.141

**Figure 7.12 Capacity Function for the Complex System**

ARRIVAL RATE (processes/sec.)	MEAN RESPONSE TIME (SEC.)		RELATIVE ERROR (per cent)
	ANALYTIC	SIMULATED	
0.00	33.639	34.180	-1.58
0.04	40.840	38.854	+5.11
0.08	52.870	49.910	+5.93
0.10	62.000	58.610	+5.78

**Figure 7.13 Comparison of Results for the Complex System**

obtain the capacity function for the aforementioned M/G/1 queue. Finally, about 85 operations must be performed for each process arrival rate considered. Even if all of the computations were performed on a microcomputer, the total time (including memory usage overhead) required to produce the analytic results shown in Figure 7.13 would still be well below 1 sec. On the other hand, for each process arrival rate, the length of the simulation run needed to obtain the process response time was on the order of an hour -- several orders of magnitude greater than the time needed by our methodology!

#### **7.4 Interpretation of Results**

The experiments described in this chapter illustrate that our methodology is well suited for modeling realistic features of distributed systems, e.g., task synchronization overhead, contention for communications facilities, and storage access delays. As can be seen from the comparison of results of the two experiments, it is very important to be able to represent such "overhead" operations in a model, in order to accurately assess the actual performance of a system. Even if the execution of tasks is the bottleneck operation in a system, ignoring effects of overhead delays will lead to significant underestimation of mean process response time, especially under light load. These errors may not be as apparent in the case of heavy load, when a system is saturated and the bottleneck resource(s) determine(s) system throughput.

To further clarify the issues discussed above, let us consider the difference in performance between the two architectures analyzed in this chapter. The task execution capacity of each system was identical: 3 sec. of work/sec. The total average execution load offered to each system by an instance of the process was nearly the same: 16.625 sec. of work in the first case and 16.750 sec. of work in the second case. However, in the first system, the overhead due to task synchronization,

memory references, and contention for the bus was assumed to be nonexistent. Thus, under a low process arrival rate, the mean process response time computed from the first model was substantially less than that given by the second model. Under heavy traffic, this difference was less significant. Note that, when each system is completely saturated, the response time of the second architecture will be better than that of the first architecture, even though the execution load per process invocation in the second system is slightly greater than that in the first system. This behavior is due to the fact that, in the second experiment, the execution load is evenly balanced among the three processors, whereas, in the first case, it is not.

Another observation obtained from this case study is that the larger  $I$  (the number of values *explicitly* computed for the capacity function of the  $M/G/1$  queue) is, the greater is the impact of the size of the physical domain model on the overall computational complexity of our solution process. In other words, the “faster” the saturation of the system occurs (as the number of processes present in the system increases), the less prominent is the dependency of the total solution time on the complexity of the architecture being considered.

Finally, it is important to note that, once the aforementioned capacity function has been obtained, the complexity of computing process response times for specific arrival rates is independent of the size of the corresponding physical domain model. This contrasts with the simulation approach, where a complete new simulation run is required for each different process arrival rate. In other words, the computational complexity of the simulation solution is proportional to the *product* of the number of distinct arrival rates and the physical system size. Thus, the benefits of our methodology are especially evident when many different arrival rates are being considered.

## CHAPTER 8

### SUMMARY AND CONCLUSIONS

In this chapter, we shall summarize the general principles and major features of our methodology, discuss its applicability to performance modeling and design evaluation of distributed systems, assess its merits with respect to computational efficiency and generality, compare it with other currently available methods and, finally, provide suggestions for further research in this area.

#### 8.1 Summary of the Methodology

We have drawn upon the concepts of both *queueing network*-based techniques and *graph*-based techniques and have combined them in an original way to produce a foundation for an effective modeling methodology. Execution environment and program behavior are separately represented by physical domain and program domain models, respectively. Queueing network elements of a physical domain model are used to model the operation of *physical components* of a distributed system. A particular architecture is represented by the proper interconnection of such elements and a corresponding routing algorithm for each customer class. Depending on the category of the system being evaluated, its model includes either one or two queueing subnetworks (called *P/C* and *M/U*). We have augmented the standard queueing network definition with a new modeling primitive, the Black Box. This element is used to explicitly represent the activation of tasks according to the precedence relationships among them in a program. Queueing network customers, possibly of different classes, are used to represent the behavior of tasks, result packets and operand packets. Finally, we have utilized the principles of Norton's

Decomposition Theorem to simplify the physical domain model. This is done by approximation of each subnetwork by a single service center having an exponential service time distribution with properly chosen, state-dependent service rates. In order to reduce the complexity of system status description, it is assumed that each "flow equivalent" server uses the processor-sharing (PS) scheduling discipline within each customer class.

Models in the program domain are used to represent structural properties of programs or processes, i.e., interdependencies among different tasks. These models are expressed in the form of computation control graphs, which constitute a powerful program representation mechanism. Each computation control graph is a collection of nodes and directed arcs connecting those nodes. Complex precedence relationships among tasks can be modeled by combining arcs using AND, OR, EXCLUSIVE-OR and UNION operators. These graphs were defined to satisfy the special requirements of our methodology. One such requirement is to be able to automatically (and in a computationally efficient way) generate a Markov process from a graph representation of a program's behavior. A Markov process is used to model the progression of events during the execution of a program. States of the corresponding Markov process represent system status after occurrences of certain "important" events; transition rates among those states represent the rates at which the associated events occur, given the current state. Each state is described by listing all currently enabled tasks, operand packets (or task completion acknowledgments) being processed and the contents of all incomplete operand sets at each storage facility. A state transition occurs whenever a queueing network customer departs from a subnetwork of the physical domain model. Thus, a state transition corresponds to the event of completing either execution of a task or the processing of some operand packet. State transition rates are computed from the throughputs of *P/C* and *M/U* subnetworks with different customer populations. The average number of tasks executed during an invocation of a program, its mean execution

time and its distribution of parallelism are obtained by constructing (from the corresponding computation control graph) and solving the Markov process.

We have also presented cost-effective methods for dealing with program variations on both local (individual task) and global (structural) levels. We use different chains of queueing network customers to model variations in individual task properties such as granularity, number of results generated, execution times for individual instructions, etc. We developed segment aggregation techniques, which approximate the behavior of a program segment by a "fluid flow" representation, to optimize the analysis of different programming constructs. The techniques presented in this thesis cover sequential, EXCLUSIVE-OR, parallel and looping segment combinations. These techniques can be applied on a hierarchical basis when modeling well-structured programs. Furthermore, in addition to modeling the execution of stand-alone programs, our methodology can also be used to analyze random arrivals of processes, the limitations and potential disadvantages of which will be addressed in the following section.

## **8.2 Application Considerations**

Our methodology can be applied to evaluating the performance of distributed architectures at virtually any level of the computer systems hierarchy. Its applicability is not limited to specific structures of programs nor to certain types of physical systems. The same general modeling principles can be utilized for each individual case. When representing the system architecture of a particular execution environment (by a physical domain model), a modeler does not have to be concerned about what specific programs are actually going to be evaluated with that system. All that need to be known are the number of different types of tasks and the attributes of each type. Conversely, when modeling the execution of a specific program, only the category of the system where the program is to be run must be known in order to be

able to generate the structure (not the transition rates) of the corresponding Markov process. Such "loose coupling" between models of program behavior and models of system architectures is especially beneficial during system design and early development stages, when the intended applications are usually not yet sufficiently detailed. It also allows for decomposition of the solution process since the solution of a queueing network and the construction of a Markov process can, potentially, be performed independently.

By properly using the set of tools provided by our methodology, designers and system planners can evaluate the performance of a proposed system with several benchmark applications in a cost-effective way, enabling them to "pilot" their design to meet specific objectives. Analysts can use the same methods for experimenting with various parameters within a system (without disturbing the standard configuration and/or operation of the system itself) to optimally adjust resource utilizations and response times or to determine system bottlenecks. With dynamically reconfigurable architectures, optimal (or nearly optimal) configurations for specific applications can be quickly determined. In the case of a static task allocation policy being employed by a system, one can "test" a number of potential assignment algorithms and adopt the one yielding the best performance.

It is important to emphasize that the solutions obtained by applying the analytic techniques presented in this thesis are estimates of the actual values and should be treated as such. The quality of approximation will vary with each particular case and, in general, cannot be determined a priori. However, we do believe that, in most cases, the results yielded by our methodology will provide a reasonable indication of the relative performance of the system being considered with respect to competing architectures or different parameter selections. Thus, we feel that this methodology is best suited for the kind of applications described in the preceding paragraph, where the relative merits of alternate design proposals and different system



configurations are being compared.

Even though no general statement on the accuracy of our methods can be made, we would like to discuss some factors which may adversely affect the quality of the produced estimates. The following guidelines were obtained from both experimentation and theoretical consideration of the approximations and assumptions used in the development of our methodology. If tasks utilize many resources in a sequential and deterministic order and the FCFS queueing discipline is used, then the validity of the "processor-sharing" approximation to a physical domain model may be questionable. Analogously, if tasks have deterministic execution times and a static allocation policy is used (with a mutually exclusive partitioning of system resources), then the "behavior" of the aggregate server representing the physical domain model would certainly not be close to exponential. Architectural features which make *P/C* and/or *M/U* subnetworks non-product-form, may also reduce the accuracy of the "Norton's" approximation. If a program includes many *types* of tasks, each type exhibiting a "much different" behavior than the others, then that program may not be accurately modeled unless each task type is represented by a distinct customer chain. The behavior of a program segment, whose degree of parallelism exhibits strong variance over the course of its execution, cannot, in general, be adequately abstracted by the segment descriptor given in Section 5.1. Since such segments violate some of the assumptions used by the segment aggregation techniques developed in Chapter 5, those techniques may not yield good estimates when utilized in such cases. Note that, when those techniques are applied hierarchically, the variance of the degree of parallelism of the aggregate segment will, in general, increase with each higher level of application.

It is also important to indicate what factors hinder the computational efficiency of our methodology. Intricate looping constructs may result in state space "explosion" of the corresponding Markov processes. Systems employing distribut-

ed synchronization and using many storage units are also subject to the latter syndrome. The computational complexity of obtaining state transition rates, i.e., solving for throughputs of a physical domain model, is combinatorially related to the number of customer chains in that model. Also, many of the optimization techniques presented in Chapter 5 are not directly applicable to multi-chain models.

### 8.3 Comparisons with Other Methods

First, we will compare our methodology with the *simulation* category of performance prediction tools. With respect to general-purpose simulation packages, our methodology requires substantially less processing time in both the model development and solution phases. In virtually all cases, the design and implementation of a model using a general-purpose simulation language would be significantly more time-consuming than the construction of the corresponding Markov process (even if no segment aggregation techniques are used). More importantly, the computation of state transition rates and the solution of the Markov process would, in general, take several orders of magnitude less time than the required simulation runs. It should be noted, however, that our analytic results are only estimates of the “actual” values and that no guarantee of achieving a specific accuracy level can be made. Thus, our methodology is no substitute for detailed simulation in cases where high accuracy is of utmost importance. However, the available empirical data indicates that its overall level of accuracy is sufficient for “first level” comparisons of performance of different system architectures. Finally, as far as performance evaluation of distributed systems is concerned, the potential range of applications of our analytic techniques is comparable with that of most general-purpose simulation tools. (Note, though, that the computational savings offered by this methodology may not be as significant when modeling applications with different classes of tasks.)

The considerations of computational efficiency and accuracy discussed above are also applicable to special-purpose simulators. The high development cost of such simulators is also an important factor. Furthermore, the generality of application of our analytic methods far exceeds that of an individual special-purpose simulator. These simulators, though able to produce highly accurate performance statistics, are particularly effective only when the same system is frequently evaluated with different programs or when intricate architectural or operating system details are to be included in the model. Thus, our methodology and special-purpose simulators complement each other by satisfying different performance prediction needs.

We will now consider other currently available analytic methods. Since techniques based only on queueing networks do not support explicit representation of interdependencies of tasks, they are harder to use and, in general, will yield less accurate estimates than our techniques. Furthermore, such methods are usually applicable to modeling only very simple precedence relationships, e.g., fork-join constructs, and cannot capture the behavior of complex program structures. As far as graph-based methods go, there are two basic categories. Those in the first category assume either an infinite capacity of each system resource or some other overly-simplified architecture. Such methods, although usually reasonably accurate, are very limited in their scope of practical application since they are not capable of modeling (much more complex) features of realistic systems -- which is easily handled by our methodology. In the second category, all architectural details included in a model are represented by nodes and arcs in a graph, intermixed together with the representation of a program's precedence relationships. Such techniques, e.g., Stochastic Petri Nets [MOL81], are prone to rapid state space explosion as the number of system components increases. (In our methodology, the complexity of a Markov process is not dependent on the system size; the cost of computing state transition rates increases with the system size, but *not nearly as fast!*) Also, a minor architectural change may require that a completely new graph be constructed and solved.

Thus, compared to our methodology, the latter graph models are much more “expensive” to both generate and solve, and they are not work conserving.

#### 8.4 Suggestions for Further Research

From the material presented in the previous chapters, we can see that research already accomplished has resulted in the development of an original and viable methodology for modeling the performance of distributed, multiple-computer systems. However, there are still a number of interesting and important issues left to be resolved, worthy of further investigation. In this section, we will identify the more important of these issues and discuss possible research approaches.

If the physical domain model consists of many service centers in tandem or if it has many servers in parallel with deterministic service times (with static allocation employed), then the “Norton’s” approximation made in Section 4.2 may not always yield sufficiently accurate results. In light of this observation, it is desirable to investigate other ways of aggregating the  $P/C$  and  $M/U$  subnetworks (such as using a general service time distribution and/or a different queueing discipline for the aggregate server or employing two or more servers) without significantly increasing the size and complexity of the corresponding Markov processes.

In section 4.2.4.2, we have already discussed the advantages of solving Markov processes “algebraically.” If the Markov process is *acyclic*, then it is relatively straightforward to obtain the algebraic solution. It may be worthwhile to attempt to develop computationally feasible algorithms for algebraically solving Markov processes containing loops and to find subsets of cyclic Markov processes where those algorithms would be *cost-effective*. It would also be useful to classify computation control graphs according to whether or not algebraic solutions of the corresponding Markov processes are cost-effective.

In Chapter 5, we have presented heuristic techniques for optimizing the solution procedure of several common programming constructs. It would be beneficial to identify other constructs pertinent to characterizing program behavior and research the feasibility of developing analogous methods for evaluating those programming constructs. It may also be possible to extend those optimization procedures to explicitly handle physical domain models containing multiple chains of customers. Furthermore, one may want to consider how to improve the "fluid flow" representation of a program segment (e.g., by including some kind of time-dependent variance), thereby increasing the accuracy of the estimates produced by the aforementioned techniques.

We have not yet been able to develop a general procedure for theoretically determining the accuracy level of the results yielded by our methodology. It is important to investigate whether or not it is possible to place theoretical *error bounds* on our analytic estimates, as functions of the program properties and of the architecture of the system being modeled. Using such bounds, a modeler can determine the range of application spectrum where the accuracy of our methodology is sufficient, the range where it is marginal and the range (if any) where it is unacceptable.

We believe that computation control graphs, as defined in Section 3.2.1, constitute a powerful, compact and versatile tool for representing the structure of programs, processes and transactions. As stated previously, one of the main reasons for developing this program representation tool is that the procedure for generating a Markov process from a computation control graph can be conveniently automated. However, it is conceivable that some program structures cannot easily be represented by computation control graphs. Thus, it would be useful to attempt to identify such types of programs (if any) and develop special procedures for modeling their behavior.

## REFERENCES

- [ACK79] Ackerman, W. B. & J. B. Dennis, "VAL -- A Value Oriented Algorithmic Language: Preliminary Reference Manual," MIT/LCS/TR- 218, Cambridge, MA, June 1979.
- [ACK82] Ackerman, W. B., "Data Flow Languages," *Computer*, 15(2), February 1982, pp. 15-25.
- [ADA85] Adams, G. B., R. L. Brown & P. J. Denning, "Report on an Evaluation Study of Data Flow Computation," RIACS TR 85.2, NASA Ames Research Center, April 1985.
- [ARV78] Arvind, K. P. Gostelow & W. Plouffe, "An Asynchronous Programming Language and Computing Machine," Dept. of Information and Computer Science, Technical Report TR-113, UCI, February 1978.
- [ARV80] Arvind, V. Kathail & K. Pingali, "A Data-Flow Architecture with Tagged Tokens," TM, Laboratory for Computer Science, MIT, August 1980.
- [BAC78] Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM*, vol. 21, August 1978, pp. 613-641.
- [BAS75] Baskett, F., K. M. Chandy, R. R. Muntz & F. G. Palacios, "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," *JACM*, vol. 22, April 1975, pp. 248-260.
- [BAU76] Baudet, G.M., "Asynchronous Iterative Methods for Multi-Processors," Research Report, Carnegie-Mellon University, November 1976.
- [CAR82] Carson, G. S., "Message-Based Distributed Computing," *Proceedings, IEEE 1982 Real-time Systems Symposium*.
- [CHA84] Chan, P. K., "A Dataflow Multiprocessor: Programming, Simulation and Performance Prediction," M.S. Thesis, UCLA, 1984.
- [CHA77] Chandy, K. M., J. H. Howard, Jr. & D. F. Towsley, "Product Form and Local Balance in Queueing Networks," *JACM*, Vol. 24, pp. 250-263, 1977.
- [CHA80] Chandy, K. M. & C. H. Sauer, "Computational Algorithms for Product Form Queueing Networks," *CACM*, Vol. 23, pp. 573-583, 1980.
- [CHA82] Chandy, K. M. & D. Neuse, "Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems," *CACM*, Vol. 25, February 1982, pp. 126-133.
- [CHU84] Chu, W. W. & K. K. Leung, "Task Response Time Model and Its Applications for Real-Time Distributed Processing Systems," *Proceedings, Real-Time Systems Symposium*, December 1984.

- [COF79] Coffman, E. G., Jr. & Kimming So, "On the Comparison between Single and Multiple Processor Systems," Research Report, UCSB, Santa Barbara, CA, August 1979.
- [COU77] Courtois, P.J., *Decomposability, Queueing and Computer System Application*, Academic Press, New York, 1977.
- [DEM82] Deminet, J., "Experience with Multiprocessor Algorithms," *IEEE Trans. on Computers*, Vol. C-31, No. 4, April 1982, pp. 278-288.
- [DEN78] Denning, P. J. & J. P. Buzen, "The Operational Analysis of Queueing Network Models of Computer Systems," *Comp. Surveys*, 10,3, September 1978, pp. 225-262.
- [DEN72] Dennis, J. B., J. B. Fosseen & J. P. Linderman, "Data Flow Schemas," Research Report, MIT, July 1972.
- [DEN83] Dennis, J. B., W. Y. P. Lim & W. B. Ackerman, "The MIT Data Flow Engineering Model," *Proceedings, IFIP*, 1983, pp. 553-563.
- [ENS77] Enslow, P., Jr., "Multiprocessor Organization -- A Survey," *Comp. Surveys*, Vol. 9, No. 1, March 1977, pp. 122-126.
- [ERC83] Ercegovac, M. D. & S. L. Lu, "A Functional Language Approach in High-Speed Digital Simulation," *Proceedings, Summer Computer Simulation Conference*, 1983, pp. 383-387.
- [ERC84] Ercegovac, M. D. & W. J. Karplus, "A Dataflow Approach in High-Speed Simulation of Continuous Systems," *Proceedings, Intl. Workshop on High-level Computer Architecture 84*, May 1984, pp. 2.1-2.8.
- [EST78] Estrin G., "A Methodology for Design of Digital Systems -- Supported by SARA at the Age of One," *AFIPS Proceedings, National Computer Conference*, 1978.
- [FER72] Fernandez, E., "Activity Transformations on Graph Models of Parallel Computations," Ph.D. Thesis, Computer Science Dept., UCLA, Los Angeles, CA, October 1972.
- [FIN85] Finkel, R. & U. Manber, "DIB -- A Distributed Implementation of Backtracking," Research Report, University of Wisconsin, Madison, WI, 1985.
- [FIN84] Finn, D. J., "Simulation of a Dataflow Architecture," Master's Thesis, UCLA, 1984.
- [FIS78] Fishman, G. S., *Principles of Discrete Event Simulation*, John Wiley and Sons, New York, 1978.
- [FRA85] Frank, G. A., D. L. Franke & W. F. Ingogly, "An Architecture Design and Assessment System," *VLSI Design*, August 1985, pp. 30-38.

- [GAU82] Gaudiot, J.-L., "On Program Decomposition and Partitioning in Data-flow Systems," Ph.D. Dissertation, UCLA, December 1982.
- [GAU83] Gaudiot, J.-L. & M. D. Ercegovac, "Simulation of a Data-flow Machine Using the SARA System," Research Report, USC and UCLA, 1983.
- [GAU85] Gaudiot, J.-L. & M. D. Ercegovac, "Performance Evaluation of a Simulated Data-Flow Computer with Low-Resolution Actors," *Journal of Parallel and Dist. Computing*, Vol. 2, 1985, pp. 321-351.
- [GIT77] Gittins, J. C. & K. D. Glazebrook, "On Bayesian Models in Stochastic Scheduling," *J. Appl. Prob.*, Vol. 14, 1977, pp.556-565.
- [GOR75] Gordon, G., *The Application of GPSS V to Discrete System Simulation*, Prentice-Hall, Englewood Cliffs, NJ, 1975.
- [GUR83] Gurd, J. & I. Watson, "Preliminary Evaluation of a Prototype Dataflow Computer," *Proceedings*, IFIP, 1983, pp. 545-551.
- [HAN77] Hansen, P. B., *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1977.
- [HAY82] Haynes, L. S., "Highly Parallel Computing," *Computer*, Vol. 15, No. 1, January 1982, pp. 7-9.
- [HEI83] Heidelberg, P. & K. S. Trivedi, "Analytic Queueing Models for Programs with Internal Concurrency," *IEEE Trans. on Computers*, Vol. C-32, No. 1, January 1983, pp. 73-82.
- [INF81] Information Research Associates, "PAWS -- Performance Analyst's Workbench System: Introduction and Technical Summary," I.R.A., Austin, TX, 1981.
- [JAC81] Jacobson, P. A. & E. D. Lazowska, "Analyzing Queueing Networks with Simultaneous Resource Possession," *CACM*, Vol. 25, 1981, pp. 142-151.
- [JEN77] Jenny, C. J., "Process Partitioning in Distributed Systems," *Proceedings*, National Telecomm. Conf., 1977.
- [KAR67] Karp, R. M., R. E. Miller & S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *JACM*, Vol. 14, 1967, pp. 563-590.
- [KEL73] Keller, R. M., "Parallel Program Schemata and Maximal Parallelism I: Fundamental Results," *JACM*, Vol. 20, No. 3, July 1973, pp. 514-537.
- [KLE75] Kleinrock, L., *Queueing Systems, Volume I: Theory*, John Wiley and Sons, New York, 1975.



- [KLE76] Kleinrock, L., *Queueing Systems, Volume II: Computer Applications*, John Wiley and Sons, New York, 1976.
- [KNU73] Knuth, D., *The Art of Computer Programming -- Sorting and Searching*, Addison-Wesley, Reading, MS, 1973.
- [KUC77] Kuck, D. J., "A Survey of Parallel Machine Organization and Programming," *ACM Comp. Surveys*, Vol. 9, No. 1, March 1977, pp. 29-59.
- [KUN76] Kung, H. T., "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors," Research Report, Carnegie-Mellon University, June 1976.
- [LAM77] Lam, S. S., "Queueing Networks with Population Size Constraints," *IBM J. Res. Dev.*, Vol. 21, No. 4, July 1977, pp. 370-378.
- [LAV82] Lavenberg, S. S., E. A. MacNair, H. M. Markowitz, C. H. Sauer, G. S. Shedler & P. D. Welch, *Computer Performance Modeling Handbook*, Academic Press, 1982.
- [MAR] Marie, R. A., "An Approximate Analytical Method for General Queueing Networks," *IEEE Trans. on Software Engineering*, SE-5, 5.
- [MOL81] Molloy, M., "On the Integration of Delay and Throughput Measures in Distributed Processing Models," Ph.D. Dissertation, UCLA, September 1981.
- [MON81] Montoye, R. K., "Simulation of the Solution of Linear Recurrences on a Parallel Processing System," Master's Thesis, University of Illinois, Urbana, IL, June 1981.
- [MUN74] Muntz, R. R. & J. Wong, "Asymptotic Properties of Closed Queueing Network Models," *Proceedings*, 8th Annual Princeton Conf. on Info. Sci. and Systems, Princeton University, March, 1974
- [PET81] Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice Hall, New Jersey, 1981.
- [RAM80] Ramamoorthy, C. V. & G. S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 5, September 1980, pp. 440-449.
- [RAZ79] Razouk R. R., M. Vernon & G. Estrin, "Evaluation Methods in SARA -- the Graph Model Simulator," *Proceedings*, 1979 Conference on Simulation, Measures and Modeling of Computer Systems, pp.189-206
- [REI75] Reiser, M. & H. Kobayashi, "Queueing Networks with Multiple Closed Chains: Theory and Computational Algorithms," *IBM J. Res. & Dev.*, 19, May 1975, pp. 283-294.

- [REI80] Reiser, M. & S. S. Lavenberg, "Mean-Value Analysis of Closed Multichain Queueing Networks," *JACM*, Vol. 27, 1980, pp. 313-322.
- [RUM77] Rumbaugh, J. E., "A Dataflow Multiprocessor," *IEEE Transactions on Computers*, C-26, No. 2, February 1977, pp.138-146
- [SAU81a] Sauer, C. H. & K. M. Chandy, *Computer Systems Performance Modeling*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [SAU81b] Sauer, C. H. & E. A. MacNair, *Computer/Communication System Modeling with the Research Queueing Package, Version 2*, IBM T. J. Watson Research Center, Yorktown Heights, New York, May, 1981.
- [SCH79] Schweitzer, P. J., "Approximate Analysis of Multiclass Closed Networks of Queues," *Proceedings*, Int'l. Conf. on Stochastic Control and Optimization, 1979.
- [SEI85] Seitz, C. L., "The Cosmic Cube," *CACM*, Vol. 28, No. 1, January 1985, pp. 22-33.
- [SIN85] Singh, V. & M. R. Genesereth, "A Variable Supply Model for Distributing Deductions," Research Report, Stanford University, Stanford, CA, 1985.
- [SOU83] de Souza e Silva, E., S. S. Lavenberg & R. R. Muntz, "A Perspective on Iterative Methods for the Approximate Analysis of Closed Queueing Networks," Research Report, UCLA, Los Angeles, CA, 1983.
- [SWA77] Swan R. J., et al., "The Implementation of the Cm\* Multiprocessor," *Proceedings*, National Computer Conference, 1977.
- [TER83a] Teradata Corp., *DBC/1012 Data Base Computer Concepts and Facilities*, Inglewood, CA, 1983.
- [TER83b] Teradata Corp., *DBC/1012 Data Base Computer System Manual*, Inglewood, CA, 1983.
- [THO78] Thomas, R. E., "Performance Analysis of Two Classes of Data-Flow Computing Systems," Dept. of Information and Computer Science, UCI, TR-120, 1978.
- [THO81] Thomas, R. E., "A Dataflow Architecture with Improved Asymptotic Performance," Ph.D. Dissertation, UCI, 1981.
- [THO85] Thomasian, A. & P. Bay, "Performance Analysis of Task Systems Using a Queueing Network Model," *Proceedings*, Int'l. Workshop on Timed Petri Nets, Torino, Italy, July, 1985
- [TOW80] Towsley, D. F., "Queueing Network Models with State-Dependent Routing," *JACM*, Vol. 27, 1980, pp. 323-337.

- [TRE82] Treleaven, P. C., D. R. Brownbridge & R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architectures," *ACM Computing Surveys*, Vol. 14, No. 1, March 1982.
- [VAN78] Vantilborgh, H., "Exact Aggregation in Exponential Queueing Networks," *JACM*, Vol. 25, 1978, pp. 620-629.
- [VER82] Vernon, M. K., "Performance-Oriented Design of Distributed Systems," Ph.D. Dissertation, UCLA, December 1982.
- [ZAH80] Zahorjan, J., "The Approximate Solution of Large Queueing Network Models," Ph.D. Thesis, Univ. Toronto, Toronto, Ont., Canada, 1980.
- [ZAH82] Zahorjan, J., K. C. Sevcik, D. L. Eager & B. Galler, "Balanced Job Bound Analysis of Queueing Networks," *CACM*, Vol. 25, February 1982, pp. 134-141.