

**Computer Science Department Technical Report  
University of California  
Los Angeles, CA 90024-1596**

**ON THE BEHAVIOR OF ALGORITHMS IN A  
MULTIPROCESSING ENVIRONMENT**

**Jau-Hsiung Huang**

**October 1988  
CSD-880085**



**ON THE BEHAVIOR OF ALGORITHMS  
IN A MULTIPROCESSING ENVIRONMENT**

by  
**Jau-Hsiung Huang**

This research, conducted under the chairmanship of Professor Leonard Kleinrock, was sponsored by the Defense Advanced Research Projects Agency, Department of Defense..

**Computer Science Department  
School of Engineering and Applied Science  
University of California  
Los Angeles**



UNIVERSITY OF CALIFORNIA

Los Angeles

ON THE BEHAVIOR OF ALGORITHMS  
IN A MULTIPROCESSING ENVIRONMENT

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

by

Jau-Hsiung Huang

1988

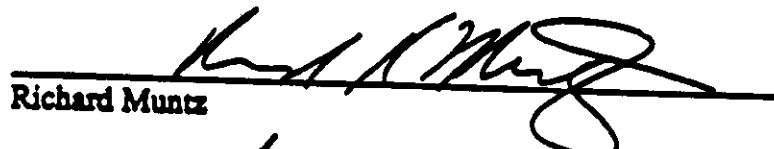


The dissertation of Jau-Hsiung Huang is approved.

  
Bruce Rothschild

  
Steven Lippman

  
Mario Gerla

  
Richard Muntz

  
Leonard Kleinrock, Committee Chair

University of California, Los Angeles

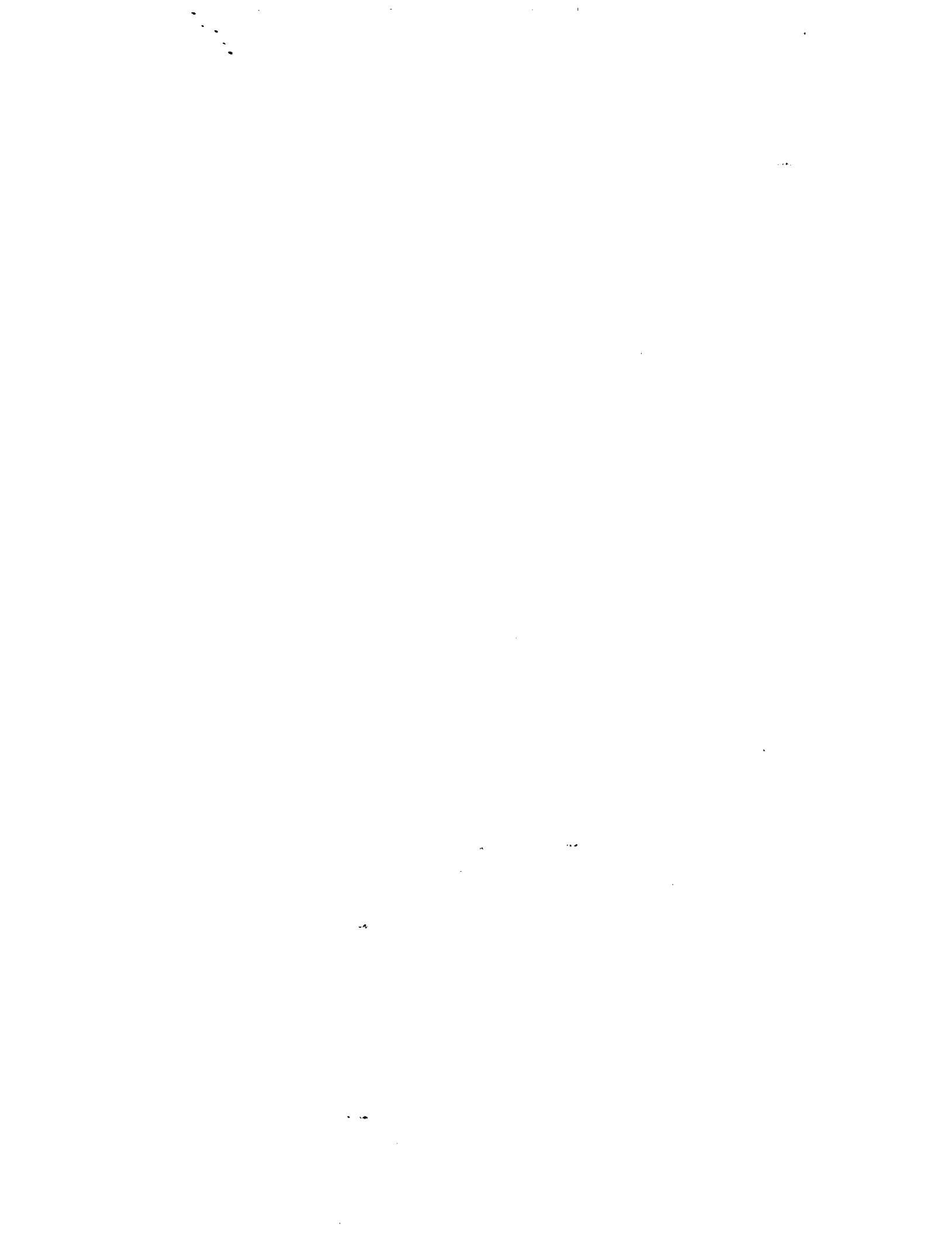
1988





## TABLE OF CONTENTS

	page
<b>1 INTRODUCTION</b> .....	1
1.1 Architectures: Distributed and Parallel Processing Systems .....	1
1.2 Algorithms: Distributed and Parallel Algorithms .....	2
1.3 Performance Measures .....	3
1.3.1 Speedup .....	3
1.3.2 Inter-processor Communication .....	4
1.3.3 Power .....	4
1.4 Outline of this Dissertation .....	6
 <b>2 PARALLEL PROCESSING SYSTEMS WITH VARYING REQUIRED PROCESSORS WITH A SINGLE JOB IN SERVICE</b> .....	 8
2.1 Previous Work .....	8
2.2 Model Descriptions of Jobs .....	10
2.2.1 Model I .....	11
2.2.2 Model II .....	11
2.2.3 Model III .....	14
2.3 Systems Without Queueing .....	15
2.3.1 Model I .....	15
2.3.2 Model II .....	22
2.3.2.1 A Special Case: Jobs with Two Stages .....	22
2.3.2.2 A General Case .....	23
2.3.3 Model III .....	26
2.3.4 An Iterative Procedure .....	29
2.4 Systems with Queueing .....	35
2.4.1 Model I .....	35
2.4.2 Model II and III .....	41
2.4.2.1 Finding the Response Time Speedup .....	41
2.4.2.1.1 The Discrete Case: Jobs with Two Stages .....	41
2.4.2.1.2 A General Case of Model II .....	42
2.4.2.2 Finding the Optimal Arrival Rate .....	43
2.4.2.3 Finding the optimal number of Processors .....	45
2.5 Conclusions .....	46
 <b>3 PARALLEL PROCESSING SYSTEMS WITH VARYING REQUIRED PROCESSORS WITH MULTIPLE JOBS IN SERVICE</b> .....	 47
3.1 Previous Work .....	48
3.2 Model Description and Assumptions .....	48
3.3 A Serial - Parallel Model .....	50
3.3.1 The Serial Stage Precedes the Parallel Stage .....	50
3.3.1.1 Mean Response Time Analysis .....	52
3.3.1.2 Optimization Issue with Power .....	54
3.3.2 The Parallel Stage Precedes the Serial Stage .....	56
3.3.2.1 Mean Response Time Analysis .....	56
3.3.2.2 Optimization Issue with Power .....	57
3.3.3 A Combined Model .....	58
3.4 A Two-stage model .....	60
3.4.1 The Low-Concurrency Stage Precedes the High-concurrency Stage .....	60



3.4.2	The High-concurrency Stage Precedes the Low-concurrency Stage .....	66
3.4.3	A Comparison .....	71
3.5	Scale-up Rule .....	71
3.5.1	Previous Work: Classical Queueing Model .....	71
3.5.2	Queueing Model with Varying Required Processors .....	73
3.5.3	Available Processors Exceed Maximum Number of Processors Required in the Two-Stage Model .....	77
3.6	An Approximation for the General Cases .....	77
3.7	Conclusion .....	87
<b>4 AN OPTIMAL PARALLEL MERGING AND SORTING ALGORITHM USING <math>\sqrt{N}</math> PROCESSORS WITHOUT MEMORY CONTENTION .....</b>		
4.1	Previous Work .....	91
4.2	Multi-way Parallel Merge Algorithm .....	92
4.3	Multi-way Parallel Sorting Algorithm .....	92
4.4	Conclusion .....	96
<b>5 A DISTRIBUTED SORTING ALGORITHMS USING BROADCAST COMMUNICATION NETWORKS .....</b>		
5.1	Previous Work .....	97
5.2	A Distributed Sorting Algorithm .....	97
5.3	Improvement of the Algorithm .....	98
5.4	Modification of the Algorithm: A Parameterized Algorithm .....	105
<b>6 WHY DISTRIBUTED OR PARALLEL SYSTEMS? .....</b>		
6.1	Previous Work .....	110
6.2	Distributed Systems .....	111
6.2.1	The Effect of Cost-Capacity Function .....	113
6.2.1.1	Polynomial Cost Function .....	113
6.2.1.2	Exponential Cost Function .....	115
6.2.2	When to Use Distributed Systems? .....	117
6.2.3	Optimization Issues Using Power .....	117
6.2.3.1	Optimize the Budget and the Capacity with a Given Arrival Rate .....	122
6.2.3.2	Optimize the Arrival Rate and the Capacity with a Given Budget .....	124
6.2.4	The Generally Distributed Service Time .....	128
6.2.4.1	Optimize the Budget and the Capacity with a Given Arrival Rate .....	132
6.2.4.2	Optimize the Arrival Rate and the Capacity with a Given Budget .....	132
6.3	Parallel Systems .....	136
6.3.1	Polynomial Cost Function - Polynomial Speedup Function .....	139
6.3.2	Polynomial Cost Function - Logarithm Speedup .....	141
6.3.3	Exponential Cost Function - Polynomial Speedup Function .....	142
6.3.4	Exponential Cost Function - Logarithm Speedup Function .....	143
6.4	Conclusions .....	146
<b>7 CONCLUSION AND SUGGESTIONS FOR FUTURE STUDY .....</b>		
		148



## LIST OF FIGURES

	page
Figure 1.1: A Parallel Processing System .....	2
Figure 1.2: A Distributed Computing System .....	2
Figure 2.1(a) A Task Graph When the Number of Processors in the System $\geq$ Maximum Required by the Job .....	10
Figure 2.1(b) A Task Graph When the Number of Processors in the System $<$ Maximum Required by the Job .....	10
Figure 2.2(a). An Example for $\vec{P} = [5, 3, 8, 4]$ and $\vec{T} = [1, 3, 2, 1]$ and $P \geq 8$ .....	12
Figure 2.2(b). An Example for $\vec{P} = [5, 3, 8, 4]$ and $\vec{T} = [1, 3, 2, 1]$ when $P = 6$ .....	12
Figure 2.3. Model III: A Model with Continuously Changing Number of Processors .....	14
Figure 2.4. $\frac{P}{n^k}$ for $k$ from 1 to 40 .....	21
Figure 3.1(a) Serial Stage Precedes Parallel Stage .....	50
Figure 3.1(b) Parallel Stage Precedes Serial Stage .....	51
Figure 3.1(c) Serial Stage Precedes and Follows Parallel Stage .....	51
Figure 3.2: The Markov Chain for Serial Stage - Parallel Stage .....	52
Figure 3.3: Average System Time Versus Load ( $\mu_1 = \mu_2 = 1$ ) .....	59
Figure 3.4: Average System Time Versus Load for Various $g$ and $f$ .....	61
Figure 3.5: A Sample Task Graph .....	62
Figure 3.6: The Markov Chain for Figure 3.5 .....	62
Figure 3.7: A Sample Task Graph .....	66
Figure 3.8: The Markov Chain for Figure 3.7 .....	67
Figure 3.9: Comparison Between $T_1$ and $T_2$ .....	72
Figure 3.10: An Example Using Scale-up Rule: M/D/3 with $\bar{x} = 2$ .....	74
Figure 3.11: An Example Using Scale-up Rule: M/E4/2 with $\bar{x} = 4$ .....	75
Figure 3.12: An Example Using Scale-up Rule: M/H3/3 with $\bar{x} = [1, 10, 100]$ .....	76



Figure 3.13: An Approximation Result for $\vec{P} = [1,2]$ and $P = 4$ .....	78
Figure 3.14: An Approximation Result for $\vec{P} = [2,1]$ and $P = 4$ .....	79
Figure 3.15: An Approximation Result for $\vec{P} = [1,2]$ and $P = 6$ .....	80
Figure 3.16: An Approximation Result for $\vec{P} = [2,1]$ and $P = 6$ .....	81
Figure 3.17: An Approximation Result for $\vec{P} = [1,2]$ and $P = 10$ .....	82
Figure 3.18: An Approximation Result for $\vec{P} = [2,1]$ and $P = 10$ .....	83
Figure 3.19(a): An Example with $\vec{P} = [3,1,3,9]$ and $\vec{T} = [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$ .....	84
Figure 3.19(b): An Approximation Model for Figure 3.19(a) .....	84
Figure 3.20: An Approximation Result for Figure 3.19(a) and $P=12$ .....	85
Figure 3.21: An Approximation Result for Figure 3.19(a) and $P=45$ .....	86
Figure 3.22(a): An Example with $\vec{P} = [3,9,3,1]$ and $\vec{T} = [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$ .....	87
Figure 3.22(b): The First Step Toward Approximation Model for Figure 3.22(a) .....	88
Figure 3.22(c): The Second Step Toward Approximation Model for Figure 3.22(a) .....	88
Figure 3.23: An Approximation Result for Figure 3.19(a) and $P=12$ .....	89
Figure 3.24: An Approximation Result for Figure 3.19(a) and $P=45$ .....	90
Figure 4.1: An Example .....	95
Figure 5.1(a): An Example .....	100
Figure 5.1(b): After Step 1 of Figure 5.1(a) .....	101
Figure 5.1(c): After Step 3 of Figure 5.1(a) .....	102
Figure 5.1(d): After Step 4 of Figure 5.1(a) .....	103
Figure 5.1(e): Step 2 of Figure 5.1(a) .....	104
Figure 5.2: Step 2 of Figure 5.1(a) .....	107
Figure 5.3: Step 2 of Figure 5.1(a) .....	109
Figure 6.1: A Symmetrical Distributed-Processing Network .....	112
Figure 6.2: Economics of Computer Power .....	112





Figure 6.3: Time Versus $n$ ( $W = 10, \lambda = 2, D = 40, a = 1.4$ ) .....	116
Figure 6.4: Time Versus $n$ ( $W=10, \lambda = 1, D=10000, a=1$ ) .....	118
Figure 6.5: A Distributed System .....	119
Figure 6.6: $m$ versus $\rho_1$ ( $n=5$ ) .....	121
Figure 6.7: $m$ versus $c^2$ ( $\rho_1 = 0.9$ ) .....	123
Figure 6.8: Power Versus $n$ for three Different $D$ ( $\lambda = 1, W=10$ ) .....	129
Figure 6.9: $D^*$ versus $c^2$ .....	134
Figure 6.10: $\lambda^*$ versus $c^2$ .....	137
Figure 6.11: $\bar{x}(n)$ versus $n$ ( $W=100, a=3, D=10000$ ) .....	144
Figure 6.12: $\bar{x}(n)$ versus $n$ ( $W=100, D=10000, a=3$ ) .....	145
Figure 6.13: $\bar{x}(n)$ versus $n$ ( $W=100, D=100, a=1$ ) .....	147



## ACKNOWLEDGEMENTS

I would like to express my appreciation to my doctoral committee consisting of Professors Leonard Kleinrock, Richard Muntz, Mario Gerla, Steven Lippman, and Bruce Rothschild. I am particularly grateful to the committee chairman and my advisor, Dr. Leonard Kleinrock for his encouragement and enlightening discussions throughout the course of this work.

The support by the Advanced Research Projects Agency of the Department of Defense (Contract number MDA903-87-C-0663) for this work is greatly appreciated and I owe a debt of gratitude to the following staff of that contract: Lily Chien, Lillian Larijani, Jodi Feiner, Saba Hunt, and Cheryle Childress. I would especially like to thank Lily Chien for her friendship and administrative assistance.

To the former and current members of our research group, Dr. Richard Gail, Dr. Hanoch Levy, Dr. Yehuda Afek, Dr. Abdelfetah Belghith, Joseph Green, Willard Korfhage, Chris Ferguson, Bob Felderman, Simon Horng, and Farid Mehovic, I offer my sincere thanks for helpful discussions. I also like to thank Steven Berson, Tom Page, Brian Livezey, Scott Spetka, Alan Downing, Robert Lindell, and all the Chinese students from Taiwan for making the environment friendly and exciting.

I dedicate this dissertation to Alice, my wife, for her constant understanding and encouragement and making life joyful when my research was going nowhere. My last and greatest gratitude goes to my parents, Kuan-Dong and Bin-Yu Huang, for their everlasting love and support.



ABSTRACT OF THE DISSERTATION  
ON THE BEHAVIOR OF ALGORITHMS  
IN A MULTIPROCESSING ENVIRONMENT

by

Jau-Hsiung Huang  
Doctor of Philosophy in Computer Science  
University of California, Los Angeles, 1988  
Professor Leonard Kleinrock, Chair

As multiprocessing systems attract more and more attention, we would like to understand more about how the system performs and hence learn how to design the system accordingly. For a parallel processing system in which a job requires varying numbers of processors as it progresses through its computation, three models for describing a job are presented. These models describe the precedence relationship between stages in a job and also specify how many processors can be concurrently used for each stage. With these models, we are able to derive the mean response time and the utilization of the processors under two different service disciplines. One of the service disciplines is to allow only one job to be processed at a time and the other is to allow more than one job to be processed concurrently as long as there are available processors for each job. With these derived performance measures and the various definitions of *power* (the simplest definition is  $\frac{\text{throughput}}{\text{response time}}$ ), we are able to derive the optimal number of processors to be used and the optimal system operating point (i.e., optimal arrival rate) such that the power is maximized. Furthermore, we also obtain the *processing time speedup* and the *response time speedup* for the service discipline which admits only one job into service at a time.

Also in this dissertation, as an application of multiprocessing systems, we present one parallel merging algorithm and one parallel sorting algorithm for use in a parallel processing system. We also present one distributed sorting algorithm to be used with a distributed computing system with broadcast communication networks. Our last topic is to study the economic issue of the scale of computing. One result from queueing theory says that the centralized system has a smaller response time than the distributed computing system as long as both systems have the same aggregate capacity. However, we still see a rapid proliferation of multiprocessing systems. If we believe there is a diseconomy of scale in computing, then we begin to understand why multiprocessing systems have evolved. Furthermore, by giving the cost-capacity function and the speedup function, we are able to find the optimal way of designing the system (i.e., the optimal capacity of each processor, the optimal budget to spend, the optimal number of processors to use in the system, and the optimal arrival rate of jobs to the system) such that either the mean response time will be minimized or the *power* (defined in chapter six) will be maximized. The impact of the distribution of the service time on the system design is also studied.

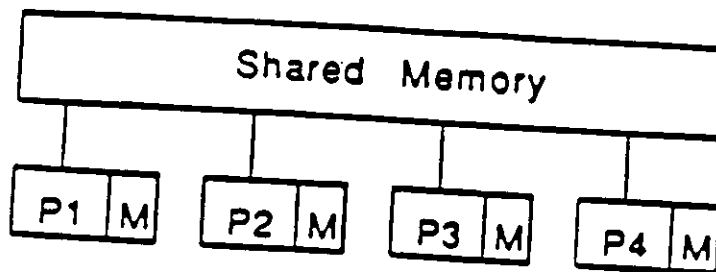


## CHAPTER 1 INTRODUCTION

As distributed and parallel processing systems get more and more popular, we are confronted with many problems in research areas and in practical situations. These problems include how to design the system, how to coordinate and schedule each component of the system, how to evaluate the performance of the system, how to design efficient parallel and distributed algorithms, how to determine the parallelism in an algorithm, and how to evaluate the performance of these algorithms. In this dissertation we focus our attention on evaluating some key performance measures of a multiprocessor system. For a multiprocessing system, we will be looking at the mean response time of the system, the utilization of the processors, the *power* (defined in the next section) of the system, and the speedup of the system. By the use of *power*, we find the optimal values of many key design parameters such that *power* is maximized. As an application, we present one parallel merging algorithm and one parallel sorting algorithm to be used with a parallel processing system. The speedup of these parallel algorithms is used as the performance measure. We also present one distributed sorting algorithm to be used with a distributed computing system which uses a broadcast communication network among processors. Inter-processor communication is used as the performance measure for this distributed algorithm. In this chapter, we first give the difference between a parallel processing system and a distributed computing system. Also given is the difference between a parallel algorithm and a distributed algorithm. Then we give the definitions of some of the performance measures which will be used in this dissertation.

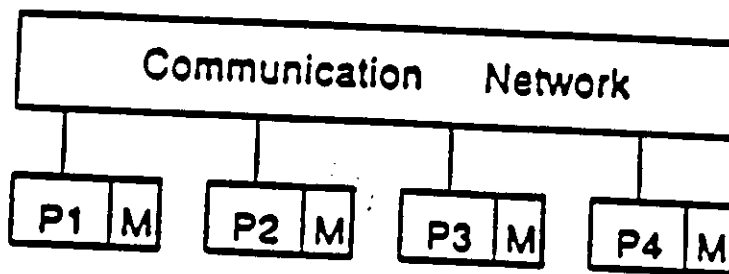
### 1.1 Architectures: Distributed and Parallel Processing Systems

We first make a distinction between a parallel processing system and a distributed computing system. Generally speaking, both systems contain many processors, ranging from several to tens of thousands or even more. By careful hardware and software design, we can coordinate these processors to work on a problem (or, problems) concurrently to reduce the time to completion. The major difference between them is how the processors communicate and coordinate with each other. For a parallel processing system, the processors usually communicate with each other by writing into and reading from the shared memory as shown in Figure 1.1.  $P_i$  refers to processor  $i$  and  $M$  refers to a local memory. For a distributed computing system, the processors usually communicate with each other through a communication network as shown in Figure 1.2. This architectural difference makes the inter-processor communication a much more severe problem for distributed computing systems than for parallel processing systems.



**Figure 1.1 A Parallel Processing System**

Parallel and distributed processing systems can be further classified into many classes. Parallel processing systems can be classified into SIMD (Single Instruction, Multiple Data stream) and MIMD (Multiple Instruction, Multiple Data stream) computing systems. Both SIMD and MIMD can further be classified as EREW (Exclusive Read Exclusive Write), CREW (Concurrent Read Exclusive Write), or CRCW (Concurrent Read Concurrent Write). Distributed computing systems can be classified by the different communication networks used to connect all the processors. For example, the communication network can be a point-to-point fully connected network, a ring-network, a mesh-connected network, a tree-network, an n-cube network, a broadcast communication network, etc.



**Figure 1.2 A Distributed Computing System**

## 1.2 Algorithms: Distributed and Parallel Algorithms

Algorithms for parallel processing systems are known as parallel algorithms. The number of processors used is sometimes large and they communicate with each other through the shared memory. The performance of a parallel algorithm is measured primarily as a function of the processing time speedup of the algorithm in terms of the number of processors used.



Algorithms for distributed computing systems are known as distributed algorithms. One characteristic of distributed computing systems is that the number of processors used is usually small relative to the input data size of the problem being solved. The processors exchange messages while cooperating on solving a problem. Usually, the time required to perform computations on data within a processor between two message exchanges is negligible when compared to the time it takes a message to be sent (including switching time and transmission time) from one processor to another. It is for this reason that we often count the number of messages exchanged when analyzing a distributed algorithm: the fewer the messages, the better the algorithm.

### 1.3 Performance Measures

There are many performance measures of interest for a computing system. Some measure directly how well the system performs, for instance, mean response time, mean queuing time, utilization of resources, and computation complexity. Some measure the overhead generated by the system, for instance, communication overhead and inter-processor communication. Some others measure the relative performance between single processor and multiple processors, for instance, speedup and concurrency. In this dissertation, we focus mainly on the speedup, communication overhead, utilization, and mean response time. Among these, we combine the utilization and the mean response time together to define another performance measure, power.

#### 1.3.1 Speedup

*Speedup* is a performance measure which is used to describe how much faster a job can be processed using multiple processors, as opposed to using a single processor. Let us define

- $\rho \triangleq$  the system utilization, i.e., the fraction of time when there is at least one job in the system.
- $\bar{x}(P) \triangleq$  the mean service time of a job given  $P$  processors in the system
- $T_1(1, \rho) \triangleq$  the mean response time of a queuing system with a single processor at system utilization  $\rho$ .
- $T_1(P, \rho) \triangleq$  the mean response time of a queuing system with  $P$  processors at system utilization  $\rho$ .

With these definitions, we define the *processing time speedup* with  $P$  processors, denoted as  $S_p(P)$ , to be

$$S_p(P) = \frac{\bar{x}(1)}{\bar{x}(P)} \quad (1.1)$$

and we define the *response time speedup* with  $P$  processors at system utilization  $\rho$ , denoted as  $S_r(P, \rho)$ , to be

$$S_r(P, \rho) = \frac{T_1(1, \rho)}{T_1(P, \rho)} \quad (1.2)$$

Clearly, the processing time speedup is applied when queuing is not allowed in the system and the response time speedup is applied when queuing is allowed in the system.

For evaluating a parallel algorithm, processing time speedup is more oftenly used as the performance measure than response time speedup. It is not hard to show that  $S_p(P)$  can never exceed  $P$ , or, the speedup for any problem using  $P$  processors can never be greater than  $P$ . For many applications,  $S_p(P)$  grows slower than a linear function of  $P$  (so called linear speedup). Therefore, the goal of designing a parallel algorithm is to achieve a high processing time speedup, hopefully a linear speedup.

### 1.3.2 Inter-processor Communication

As the technology of computer and communication advances, distributed computing systems become more and more popular because they have the potential to speed up the computation time at a lower cost. However, one major concern of distributed systems is the communication between processors which is required to control and synchronize the algorithm and the system. If the communication time (which includes the packetizing time for messages, the hardware switching time, and the transmission time of messages) is much greater than the computation time, then inter-processor communication becomes a bottleneck and therefore serves as a major measure of the performance. Taking the inter-processor communication as the performance measure, we will be measuring how many bits have to be sent across the communication network.

Furthermore, if the communication network between processors is a multi-access communication channel, we have to consider not only how many bits have to be transmitted through the communication network, but we also have to consider "how" these bits will be sent through the network. For example, it can be one byte (generally eight bits) per packet, hence we have to access the channel more frequently. Or, it can be several bytes per packet, hence we do not have to access the channel too frequently. More details of this issue will be addressed in chapter five.

### 1.3.3 Power

For a parallel processing system, there are two performance measures which compete with each other: *utilization* and *response time*. That is, by raising the utilization of the system, which is desirable, the mean response time will also be raised, which is not desirable. Similarly, by reducing the mean response time, the utilization of the system will also be lowered. In this paper, these two performance measures are combined into a single measure, known as *power*, which increases either by lowering the mean response time or raising the utilization of the system.

In [KLEI78] and [KLEI79], *power* was defined as

$$power = \frac{\rho}{T/\bar{x}} \quad (1.3)$$

With this measure we see that an increase in utilization ( $\rho$ ) or a decrease in mean response time ( $T$ ) increases the power. Here  $\bar{x}$  is the average total service time required by a job. Note that this normalized definition is such that  $0 \leq \rho < 1$  and  $1 \leq T/\bar{x}$  and so  $0 \leq power < 1$ . The symbol \* will be used to denote variables which are optimized respect to power. In [KLEI79], it was found that for any M/G/1 queueing system, power, as defined in (1.3), is maximized when

$$\bar{N}^* = 1 \quad (1.4)$$

where  $\bar{N}$  = the average number of jobs in the system. This result says that an M/G/1 system has a maximum power when on the average there is only one job in the system. This result is intuitively pleasing since it corresponds to our deterministic reasoning that the proper operating point for a single server system is exactly when only one job is being served in the system and no others are waiting for service at the same time.

In this dissertation, we define the following notation:

- $u_1(P) \triangleq$  the average processor efficiency given there are  $P$  processors in the system
- $u_2(\lambda, P) \triangleq$  the average processor efficiency given the job arrival rate  $\lambda$  and  $P$  processors in the system.
- $T_2(\lambda, P) \triangleq$  the mean response time (i.e., service time + queueing time) given the job arrival rate  $\lambda$  and  $P$  processors in the system
- $\Pi^{(r)}(P) \triangleq$  power with parameter  $r$  given  $P$  processors and no arrivals come to the system.
- $\Pi_2^{(r)}(\lambda, P) \triangleq$  power with parameter  $r$  given the job arrival rate  $\lambda$  and  $P$  processors in the system.

Note the difference between  $u$ , which is the average processor utilization, and  $\rho$ , which is the average system utilization. Whenever there is a job in the system, the system utilization is "1" but the processor utilization does not have to be "1" since there may be some idle processors in the system because the job in service does not need all of them. Hence, the system utilization is always greater than or equal to the processor utilization.

If queueing effects are not considered in the system, we define power as:

$$\Pi^{(1)}(P) = \frac{u_1(P)}{\bar{x}(P)} \quad (1.5)$$

With this definition, a more general definition of power (as originally studied in [KLEI79]) is given as:

$$\Pi^{(r)}(P) = \frac{u_1(P)^r}{\bar{x}(P)} \quad (1.6)$$

where  $r$  is a non-negative number. With this generalization, we have the freedom to favor the processor utilization more heavily over the service time by simply increasing the parameter  $r$ .

If queueing effects are considered in the system, we define power as

$$\Pi_2^{(1)}(\lambda, P) = \frac{u_2(\lambda, P)}{T_2(\lambda, P)} \quad (1.7)$$

and the generalization of (1.7) is given as

$$\Pi_2^{(r)}(\lambda, P) = \frac{u_2(\lambda, P)^r}{T_2(\lambda, P)} \quad (1.8)$$

## 1.4 Outline of this Dissertation

In chapter two, we extend the notion of "power" as applied to queueing systems to parallel processing systems. We consider a stream of jobs arriving to a parallel processing system; the system admits one job into service at a time and we model a job as a concatenation of stages where the number of processors required by each stage can be different. With this model and the definition of power, we are able to find the optimal system operating point (i.e., input rate of jobs) and the optimal number of processors to use in the parallel processing system such that power is maximized. Expressions for the processing time speedup and the response time speedup are also obtained. We obtain these results for two cases: queueing and no queueing. These results can be very helpful in designing a parallel processing system.

For the systems described in chapter two, some computing power will be wasted when there is one job in service which does not require all the processors and there are jobs in the queue waiting to be served. It is natural to redesign the system such that whenever there is an available processor in the system and there are jobs in the queue waiting for service, we allow jobs in the queue to use the available processor in the system following a FCFS order. However, a higher priority is given to jobs which are closer to completion and we allow a higher priority job to preempt a lower priority job if the higher priority job needs more processors than it currently possesses. Because of this priority assignment and the preemption characteristics, the number of processors possessed by a job during its execution time varies. In chapter three we investigate such kinds of systems to find the mean response time. The analysis of these systems turn out to be very difficult as pointed out in [KLE186]. However, we are able to find the exact solution for some special cases. With the exact solution for some special cases and the scale-up rule, which will be discussed in detail in chapter three, we are able to provide a very good approximation for general cases.

As an application of parallel processing systems, we present a parallel merging algorithm and a parallel sorting algorithm in chapter four. Both of these algorithms use  $\sqrt{N}$  (where  $N$  is the number of data to be merged or sorted) processors to achieve a linear speedup. Furthermore, these algorithms require neither concurrent read nor concurrent write. These characteristics make both of these algorithms optimal.

A distributed sorting algorithm is presented in chapter five for a distributed system which uses a broadcast communication network to communicate between processors. As described before, the performance measure used in this chapter is the communication requirement across the network. Several versions of this algorithm will be discussed in this chapter. In one of the versions, we can adjust a parameter to trade between memory requirement and communication requirement.

In chapter six we explore why more and more distributed and parallel processing systems have been built although queueing theory tells us that a centralized system performs better than a distributed/parallel system given that the aggregate capacity for the centralized and the distributed/parallel system are the same. An issue of diseconomy of scale in computing and an issue of processing time speedup are discussed in this chapter. Assuming the diseconomy of scale in computing exists, we revisit the definition of power and give two other definitions to power. With these new definitions, we are able to find the optimal capacity of the processors to be used and the optimal number of processors to be used in the system in order to maximize power. In this chapter, the impact of the variance of the job's service time distribution is also studied. The results suggest that for jobs with higher service time variance, the

multiprocessor system becomes more attractive. Chapter seven gives the conclusion and suggestions for future study.

## CHAPTER 2

### PARALLEL PROCESSING SYSTEMS WITH VARYING REQUIRED PROCESSORS WITH A SINGLE JOB IN SERVICE

We extend the notion of *power* as applied to queuing systems to parallel processing systems. We consider a stream of jobs coming to a parallel processing system; the system admits one job into service at a time and we model a job as a concatenation of stages where the number of processors required by each stage can be different. With this model and the definition of power given in chapter one, we are able to find the optimal system operating point (i.e., input rate of jobs) and the optimal number of processors to use in the parallel processing system such that power is maximized. Expressions for the processing time speedup and the response time speedup are also obtained. We obtain these results for two cases: queueing and no queueing. These results can be very helpful in designing a parallel processing system.

We model a parallel processing system as a system with a single queue. Only one job can be admitted into service at a time following a FCFS discipline while the others have to wait in the queue. A job in such a system is modeled as a concatenation of stages and the number of processors required in each stage can be different. If, for some stage, the job in service requires fewer processors than the system provides, then the job will simply use all that it needs and the other processors will be idle for that stage. If, for some stage, the job in service requires more processors than the system provides, then it will use all the processors in the system for an extended time such that the work done in that stage is preserved.

#### 2.1 Previous Work

In [KLEI79], power was defined as

$$\text{power} = \frac{\gamma}{T} \quad (2.1)$$

where  $\gamma$  is defined as the throughput of the system and  $T$  is defined as the mean response time of the system. With this definition, it was proved that for any M/G/1 queuing system, power is maximized when

$$\bar{N}^* = 1 \quad (2.2)$$

In that same paper, the definition of power was generalized to be

$$\text{power} = \frac{\rho^r}{r \bar{X}} \quad (2.3)$$

where  $\rho$  is the system utilization and  $r$  is a nonnegative real variable. With this new definition, The following results were proved:

For an M/M/1 system, power is maximized when

$$\bar{N}^* = r \quad (2.4)$$

For an M/G/1 system, power is maximized when

$$\bar{N}^* = \frac{2(r-1)^2 x^2 + 4x(r^2 - r + 2) + 8(r+1) + 2A[x(r-1) - 2]}{4r[(x+2)(r+1) - A]} \quad (2.5)$$

where

$$A = \sqrt{(r-1)^2 x^2 + 4x(r^2 + 1) + 4(r+1)^2}$$

and

$$x = c_b^2 - 1$$

where  $c_b$  is the coefficient of variation of the service time distribution.

An extensive study of power applied in a queueing network was given in [GAIL83]. In [KUNG84] a system was studied where a job has a deterministic service requirement and has a processor requirement which changes during its execution time. In that model, a job requires more and more processors from the system (linearly in time) until half of the work of the job is finished, then the job needs fewer and fewer processors also linearly (as shown in Figure 2.1(a) where B is the maximum number of processors required by the job). Only one job is in the system and, hence, queueing effects are not considered. It is assumed in [KUNG84] that whenever the number of required processors is greater than the number of available processors in the system, the job will use all the available processors by elongating the service time under the constraint that the overall work is not changed (as shown in Figure 2.1(b)). In [KUNG84], power was defined in the unnormalized form (from [KLE179]) as

$$\text{power} = \frac{u}{\bar{x}}$$

where  $u$  is defined as the processor utilization during the service time of the job. From this definition of power, it is clear that queueing is not considered in this model. It was shown that in order to maximize power, the optimal number of processors to use in the system is about 60 % of the maximum number of processors required by the job. This result was achieved by numerically solving a 5<sup>th</sup> order polynomial equation.

In this chapter, we define power as in (1.5) to (1.8), depending on the situation. With these definitions of power, we will find analytical expressions for the optimal system operating point (i.e., input rate of jobs) and the optimal number of processors to use in the parallel processing system such that power is maximized.

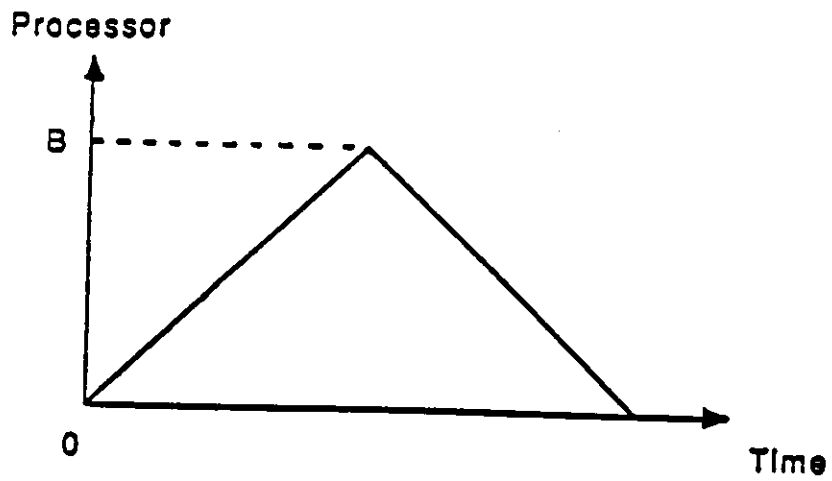


Figure 2.1(a)

A Task Graph where the Number of Processors in the System  $\geq$  Maximum Required by the Job

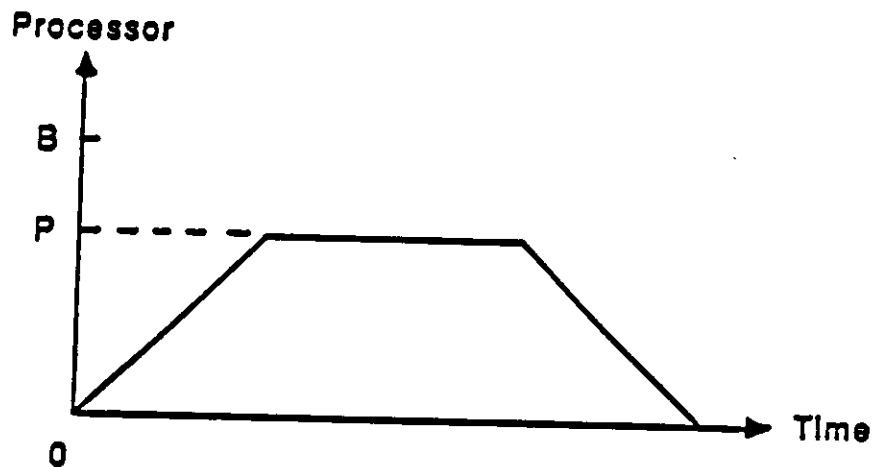


Figure 2.1(b)

A Task Graph where the Number of Processors in the System  $<$  Maximum Required by the Job

## 2.2 Model Descriptions of Jobs

We use three different models to describe a job in a parallel processing system. We name these models Model I, Model II, and Model III. The difference between Model I and Model II is that in Model I, we allow *each stage* to have its own service time distribution while in Model II we vary the overall service time distribution. In Model I and Model II the number of processors required between stages is a discrete variable, whereas in Model III we assume that the number of processors required by a job changes continuously over its execution time.



### 2.2.1 Model I

In this model, a job is composed of several stages: in each stage a given number of processors will be used for a random period of time. The numbers of processors needed for different stages need not be the same. Hence, a task can be described using three vectors. The first vector is the *processor* vector which specifies the number of processors required by each stage. The second vector is the *time* vector which specifies the *average* service time required for each stage. The third vector is the *standard deviation* vector which gives the standard deviation of the distribution of the service time for each stage corresponding to the time vector. We denote these three vectors as

$$\vec{P} = [P_1, P_2, P_3, \dots, P_n]$$

$$\vec{T} = [t_1, t_2, t_3, \dots, t_n]$$

$$\vec{\sigma} = [\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_n]$$

Since there is only one job in the system, we can arbitrarily interchange the stages without affecting the service time and the processor utilization. For the convenience of computation, we rearrange the processor vector such that the number of processors required in each stage is non-decreasing, i.e.,  $P_i \leq P_{i+1}$  for  $1 \leq i \leq n-1$ . The time vector and the standard deviation vector will also be adjusted accordingly. By doing this, we have

$$\vec{P} = [P_1, P_2, P_3, \dots, P_n]$$

where  $P_i \leq P_{i+1}$  for  $1 \leq i \leq n-1$  and the corresponding time vector and the corresponding standard deviation vector are given by

$$\vec{T} = [t_1, t_2, t_3, \dots, t_n]$$

and

$$\vec{\sigma} = [\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_n]$$

where  $\vec{P}$ ,  $\vec{T}$ , and  $\vec{\sigma}$  stand for the adjusted processor vector, the adjusted time vector, and the adjusted standard deviation vector respectively and  $n$  is defined as the number of stages in a job. An example is given in Figure 2.2(a) for  $\vec{P} = [5, 3, 8, 4]$  and  $\vec{T} = [1, 3, 2, 1]$ . We have not represented the  $\vec{\sigma}$  vector graphically. An example corresponding to Figure 2.2(a) is given in Figure 2.2(b) when the maximum number of processors required by the job is greater than the number of processors in the system. Note in Figure 2.2(b) that the service time for stage 3 is elongated such that the area (work) in that stage is preserved.

### 2.2.2 Model II

In this model, we assume a job is composed of many tasks and the number of tasks in a job is a random variable  $\hat{M}$  with mean  $M$  and coefficient of variation  $c_M$ . We assume the service time for all tasks is deterministic with unit service time. A job is described by using  $M$  and  $c_M$  along with two other vectors. The first vector is called the *fraction* vector,  $\vec{f}$ , and the second vector is called the *processor* vector,  $\vec{P}$ . Without loss of generality, we can rearrange the processor vector and the fraction vector is such a way that

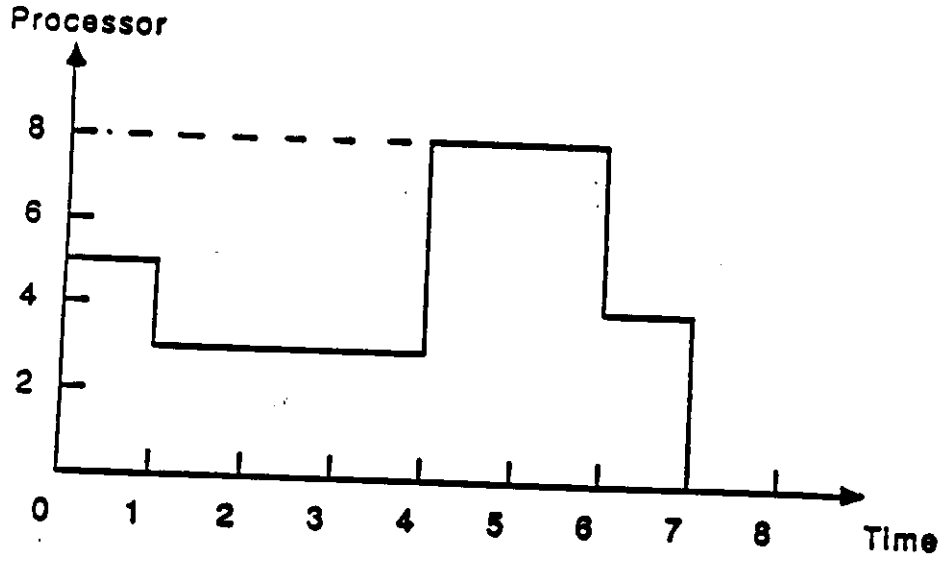


Figure 2.2(a). An Example for  $\vec{P} = \{5, 3, 8, 4\}$  and  $\vec{T} = \{1, 3, 2, 1\}$  and  $P \geq 8$

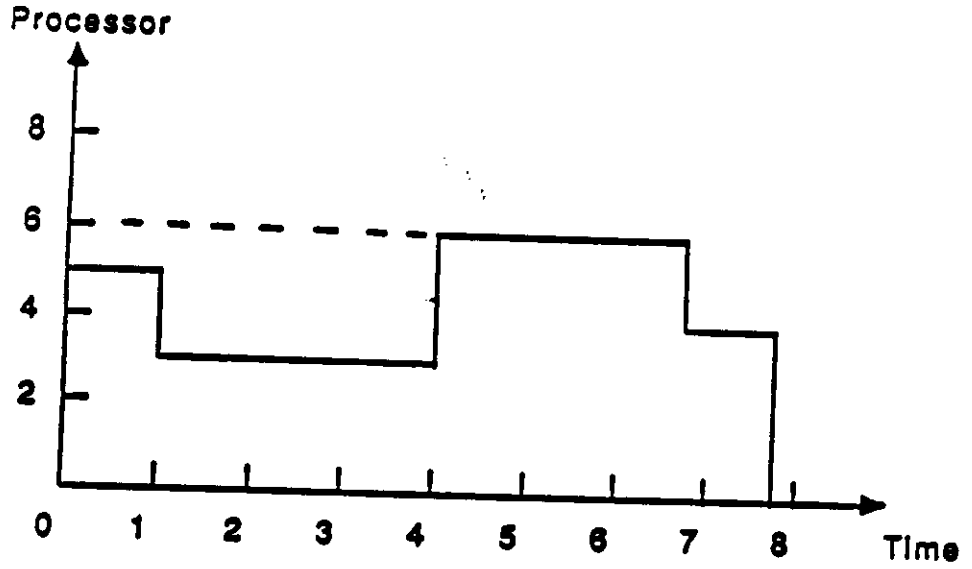


Figure 2.2(b). An Example for  $\vec{P} = \{5, 3, 8, 4\}$  and  $\vec{T} = \{1, 3, 2, 1\}$  when  $P = 6$

$P_i \leq P_{i+1}$  for  $1 \leq i \leq n-1$  in the processor vector as explained in section 2.2.1. We denote the adjusted fraction vector ( $\vec{f}$ ) and the adjusted processor vector ( $\vec{P}$ ) as:

$$\vec{f} = [f_1, f_2, f_3, \dots, f_n]$$

$$\vec{P} = [P_1, P_2, P_3, \dots, P_n]$$

The meaning of  $\vec{f}$  and  $\vec{P}$  can be explained as follows: Over the total tasks of the job, a fraction  $f_i$  of the total tasks ( $M$ ) in a job can use  $P_i$  processors to concurrently process these tasks. By this definition, it is clear that

$$\sum_{i=1}^n f_i = 1$$

**Example 2.1:**

In this example, for the simplicity of illustration, we assume the number of tasks in a job to be deterministic, i.e.,  $c_M = 0$ , and the service time for each task is one second.

$$M = 1000$$

$$\vec{f} = [0.2, 0.5, 0.3]$$

$$\vec{P} = [4, 7, P]$$

where  $P \triangleq$  the number of processors in the system.

This example means 20 % of the 1000 tasks (= 200 tasks) can use 4 processors to concurrently process this workload. Hence, this portion of tasks will take 50 seconds to complete. If there are more than 4 processors in the system, the extra processors will not be used at all. However, if there are only 2 processors in the system, then this portion of tasks can use only 2 processors to process this amount of tasks in 100 seconds. Similarly, 50 % of the tasks (= 500 tasks) can use 7 processors to concurrently process this tasks. Lastly, 30 % of the tasks (= 300 tasks) can be concurrently processed using all the processors,  $P$ , in the system.

Note that this example does not mean that there are only three stages in a job. The fraction vector is the result after grouping all the stages which use the same number of processors to process the work. A job described in example 2.2 has the same performance as the one in example 2.1.

**Example 2.2:**

As in Example 2.1, we assume the number of tasks in a job to be deterministic.

$$M = 1000$$

$$\vec{f} = [0.05, 0.15, 0.12, 0.18, 0.2, 0.1, 0.2]$$

$$\vec{P} = [4, 4, 7, 7, 7, P, P]$$

### 2.2.3 Model III

We now describe a continuous version of the above model. In this model we assume the number of processors required by jobs is a continuous variable which changes continuously over time. A special model with deterministic workload per job will be first described. After that, a general model with random workload per job will be described.

For the special case with deterministic workload, we define  $R(t) = f(t)$  to be the function which gives the number of processors required by a job at time  $t$  and  $t$  is in the range of  $[0, b]$  such that  $R(b) = B$ . For such a model, the workload for each job is deterministic with value  $\int_0^b f(t) dt$ .

For the general case, we define  $\hat{R}(t) = f(\frac{t}{\hat{K}})$  to be the function which gives the number of processors required by the job at time  $t$ .  $\hat{K}$  is a random variable with mean  $K$  and coefficient of variation  $c_K$  and  $f$  is a function such that it has a fixed maximum value  $B$  and  $t$  is in the range  $[0, \hat{K}b]$  as shown in Figure 2.3. As explained in the discrete model (section 2.2.1), we assume  $\hat{R}(t) = f(\frac{t}{\hat{K}})$  to be a monotonically increasing continuous function in  $t$  without loss of generality.

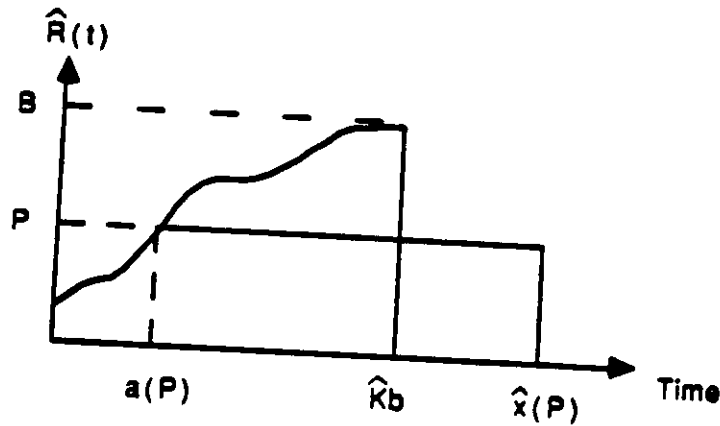


Figure 2.3: Model III: A Model with Continuously Changing Number of Processors

### 2.3 Systems Without Queuing

In this section, queuing is not allowed. That is, we are given a job to be processed and no other jobs will enter the system. Therefore, we use (1.5) or (1.6) as the definition of power and we want to find the optimal number of processors to be used in the system in order to maximize power. Furthermore, we also obtain an expression for the processing time speedup. Below, Models I and II are discussed and an iterative procedure is introduced in order to find the optimal number of processors for the system.

#### 2.3.1 Model I

We assume the adjusted processor vector and the corresponding time vector of a job are given by

$$\vec{P} = [P_1, P_2, P_3, \dots, P_n]$$

and

$$\vec{T} = [t_1, t_2, t_3, \dots, t_n]$$

where  $P_i \leq P_{i+1}$  for  $1 \leq i \leq n-1$ .

We assume there are  $P$  processors available in the system. We first find the index "m" such that  $P_{m-1} \leq P < P_m$  if  $P_1 \leq P < P_n$ ; or,  $m = n + 1$  if  $P \geq P_n$ ; or,  $m = 1$  if  $P < P_1$ . The mean service time of a job can then be found as:

$$\bar{x}(P) = \sum_{i=1}^{m-1} t_i + \frac{1}{P} \sum_{i=m}^n t_i P_i$$

We define

$$\alpha = \sum_{i=1}^{m-1} t_i \tag{2.6}$$

$$\beta = \sum_{i=m}^n t_i P_i \tag{2.7}$$

$$\gamma = \sum_{i=1}^n t_i P_i$$

Hence,

$$\bar{x}(P) = \alpha + \frac{1}{P} \beta$$

The mean service time for a job using one processor is clearly  $\gamma$ . Hence, the processing time speedup is

$$S_p(P) = \frac{\gamma}{\alpha + \frac{\beta}{P}} \tag{2.8}$$

From equation (2.8) we can show that the highest possible speedup, denoted as  $S_{p,max}$ , is achieved when  $P \geq P_n$ , which leads to  $\alpha = \sum_{i=1}^n t_i$  and  $\beta = 0$ ; therefore, the highest possible processing time speedup, no

matter how many processors are available, is

$$S_{p, \max} = \frac{\gamma}{\alpha} = \frac{\sum_{i=1}^n P_i t_i}{\sum_{i=1}^n t_i}$$

This result is intuitive since the numerator is the time for one processor to finish a job while the denominator is the time to finish a job when there is always a sufficient number of processors for each stage and hence no prolonged stage service time.

Since the computing capability of  $P$  processors during the job's service time  $\bar{x}(P)$  is  $P\bar{x}(P)$  and the total work of a job is simply  $\gamma$ , the processor utilization of the system during the job's service time equals

$$u_1(P) = \frac{\gamma}{P\bar{x}(P)}$$

Defining power as in (1.5), we have

$$\Pi^{(1)}(P) = \frac{u_1(P)}{\bar{x}(P)} = \gamma \frac{1}{P\bar{x}(P)^2} = \gamma \frac{1}{\alpha^2 P + 2\alpha\beta + \frac{\beta^2}{P}}$$

Hence,

$$\frac{d\Pi^{(1)}(P)}{dP} = -\gamma \frac{(\alpha + \frac{\beta}{P})(\alpha - \frac{\beta}{P})}{(\alpha^2 P + 2\alpha\beta + \frac{\beta^2}{P})^2}$$

By carefully examining  $\Pi^{(1)}(P)$ , we find the following characteristics of  $\Pi^{(1)}(P)$ . (1)  $\Pi^{(1)}(P)$  is a continuous function since  $\bar{x}(P)$  is a continuous function. (2)  $\Pi^{(1)}(P)$  is everywhere differentiable with respect to  $P$  except when  $P = P_i$  where  $1 < i < n$ . (3)  $\Pi^{(1)}(P)$  has a global maximum value and no local maximum value. To see this, more explanation will be provided. First of all, note that  $\alpha$  increases as  $P$  increases; meanwhile,  $\beta$  decreases as  $P$  increases. Also note that when  $P$  is small,  $\alpha < \frac{\beta}{P}$ ; meanwhile, when  $P$  is large,  $\alpha > \frac{\beta}{P}$ . Hence, from  $\frac{d\Pi^{(1)}(P)}{dP}$  we can see that  $\frac{d\Pi^{(1)}(P)}{dP}$  is positive when  $P$  is small and  $\frac{d\Pi^{(1)}(P)}{dP}$  keeps positive until at one point  $\alpha > \frac{\beta}{P}$ , then  $\frac{d\Pi^{(1)}(P)}{dP}$  becomes negative and will remain negative as  $P$  increases. Therefore, (3) is true. The explanation given here also explains (4) which follows. (4) There is at most one value for  $P$  such that  $\frac{d\Pi^{(1)}(P)}{dP} = 0$  (as long as it is differentiable) and the value of  $\Pi^{(1)}(P)$  at that particular point has the maximum value. (5) The maximum value of  $\Pi^{(1)}(P)$  may happen when  $P = P_m$  where  $1 < m < n$ . If this happens,  $\frac{d\Pi^{(1)}(P)}{dP}$  will change sign from positive to negative as  $P$  changes from  $P_m - \epsilon$  to  $P_m + \epsilon$  where  $\epsilon \rightarrow 0$ .

From the above characteristics of  $\Pi^{(1)}(P)$ , we can find  $P^*$  as follows. If  $P_{m-1} < P^* < P_m$ , then we can find  $P^*$  by setting  $\frac{d\Pi^{(1)}(P)}{dP} = 0$ ; hence, we have

$$P^* = \frac{\beta}{\alpha} \quad (2.9)$$

It can be shown that  $\frac{d^2}{dP^2} \Pi^{(1)}(P^*) < 0$ ; hence, this  $P^*$  maximizes power. If power is maximized when  $P^* = P_m$  where  $1 < m < n$ , we know  $\frac{d\Pi^{(1)}(P)}{dP}$  will change sign at  $P_m$ ; hence, we have

$$\alpha - \frac{\beta}{P_m} < 0 \quad \text{and} \quad \alpha' - \frac{\beta'}{P_m} > 0$$

or

$$\left[ \alpha - \frac{\beta}{P_m} \right] \left[ \alpha' - \frac{\beta'}{P_m} \right] < 0$$

where  $\alpha' \triangleq \sum_{i=1}^m t_i$  and  $\beta' \triangleq \sum_{i=m+1}^n t_i P_i$ .

Using the more general definition of power as given in (1.6) we have

$$\Pi^{(r)}(P) = \frac{u_1(P)^r}{\bar{x}(P)} = \frac{\gamma^r}{P^r \bar{x}(P)^{r+1}} = \frac{\gamma^r}{(\alpha P + \beta)^r \left( \alpha + \frac{\beta}{P} \right)}$$

Hence,

$$\frac{d\Pi^{(r)}(P)}{dP} = -\gamma^r \cdot \frac{(\alpha P + \beta)(r\alpha P - \beta)}{P^2(\alpha P + \beta)^{r+1} \left( \alpha + \frac{\beta}{P} \right)^2}$$

Optimizing  $\Pi^{(r)}(P)$  with respect to  $P$ , we have (if  $P_{m-1} < P^* < P_m$ )

$$P^* = \frac{\beta}{r\alpha} \quad (2.10)$$

Similarly, if  $P^* = P_m$  ( $1 < m < n$ ), we have

$$(r\alpha P_m - \beta)(r\alpha' P_m - \beta') < 0$$

From these results, we arrive at the following theorem.

**THEOREM 2.1:**

Given  $\vec{P}$  and  $\vec{T}$  (and for  $\alpha$  and  $\beta$  as in (2.6) and (2.7) respectively), the processing time speedup for any  $P$  is given by

$$S_p(P) = \frac{\gamma}{\alpha + \frac{\beta}{P}}$$

If  $P_{m-1} < P^* < P_m$  and power is defined as in (1.5), power is maximized when

$$P^* = \frac{\beta}{\alpha}$$

If  $P^* = P_m$  ( $1 < m < n$ ), power is maximized when the following condition is met:

$$\left( \alpha - \frac{\beta}{P_m} \right) \left( \alpha' - \frac{\beta'}{P_m} \right) < 0$$

If power is defined as in (1.6), it is maximized (if  $P_{m-1} < P^* < P_m$ ) when

$$P^* = \frac{\beta}{r\alpha}$$

If  $P^* = P_m$  ( $1 < m < n$ ), power is maximized when the following condition is met:

$$(r\alpha P_m - \beta)(r\alpha' P_m - \beta') < 0$$

**Corollary 2.1:**

If the elements of the processor vector form a linear function and the elements of the time vector equal unity, i.e.,  $P_i = i$  and  $t_i = 1$  for all  $i$ , power as defined as in (1.5), is maximized when

$$\frac{P^*}{n} = \frac{\left[ 12 + \frac{1}{n^2} + \frac{12}{n} \right]^{1/2} - \frac{1}{n}}{6}$$

if  $n \gg 1$ ,  $\frac{P^*}{n} \approx \frac{1}{\sqrt{3}}$ .

If power is defined as in (1.6), it is maximized when

$$\frac{P^*}{n} = \frac{\left[ 4(2r+1) + \frac{1}{n^2} + \frac{4(2r+1)}{n} \right]^{1/2} + \frac{1}{n}}{2(2r+1)}$$

if  $n \gg 1$ ,  $\frac{P^*}{n} \approx \frac{1}{\sqrt{2r+1}}$ .

[Proof]

For such a  $\vec{P}$  and  $\vec{T}$ , we can easily find the value of  $m$  to be  $\lfloor P^* \rfloor + 1$  ( $\lfloor x \rfloor$  is defined as the largest integer which is smaller than  $x$ , which is a real number). Hence,

$$\alpha = \sum_{i=1}^{\lfloor P^* \rfloor} 1 = \lfloor P^* \rfloor$$

$$\beta = \sum_{i=\lfloor P^* \rfloor+1}^n i = \frac{n^2 + n - \lfloor P^* \rfloor^2 - \lfloor P^* \rfloor}{2}$$

Let us approximate  $\lfloor P^* \rfloor$  with  $P^*$  and define power as in (1.5), we have from (2.9)



$$P^* = \frac{\beta}{\alpha} = \frac{n^2 + n - P^{*2} - P^*}{2P^*}$$

Solving for  $P^*$  we have

$$P^* = \frac{\sqrt{12(n^2 + n) + 1} - 1}{6}$$

Hence

$$\frac{P^*}{n} = \frac{\left[12 + \frac{1}{n^2} + \frac{12}{n}\right]^{1/2} - \frac{1}{n}}{6}$$

If  $n \gg 1$ , we have

$$\frac{P^*}{n} = \frac{\sqrt{12}}{6} = \frac{1}{\sqrt{3}} = 57.7\%$$

By using the more general definition of power as in (1.6), we have from (2.10) that

$$P^* = \frac{\beta}{r\alpha} = \frac{n^2 + n - P^{*2} - P^*}{2rP^*}$$

Solving for  $P^*$  we have

$$P^* = \frac{\sqrt{1 + 4(2r+1)(n^2 + n)} - 1}{2(2r+1)}$$

Hence

$$\frac{P^*}{n} = \frac{\left[4(2r+1) + \frac{1}{n^2} + \frac{4(2r+1)}{n}\right]^{1/2} + \frac{1}{n}}{2(2r+1)}$$

If  $n \gg 1$ , we have

$$\frac{P^*}{n} = \frac{1}{\sqrt{2r+1}}$$

Q.E.D.

Note that the actual processor vector of a job does not have to be in the form of  $P_i = i$ . Only after rearrangement (such that  $P_{i-1} \leq P_i$ ) must the processor vector of the job be of this form.

**Corollary 2.2:**

If the elements of the processor vector form a power function of order  $k$  and the elements of the time vector all equal unity (i.e.,  $P_i = i^k$  and  $t_i = 1$  for all  $i$ ) and  $n \gg 1$ , power, as defined as in (1.5), is maximized when

$$\frac{P^*}{n^k} = \left(\frac{1}{k+2}\right)^{\frac{k}{k+1}}$$

If power is defined as in (1.6), it is maximized when

$$\frac{P^*}{n^k} = \left(\frac{1}{rk+r+1}\right)^{\frac{k}{k+1}}$$

[Proof]

Using this  $P$ , and  $t$ , and defining power as in (1.5), we have

$$\alpha = \sum_{i=1}^n 1 = \lfloor P^{1/k} \rfloor$$

$$\beta = \sum_{i=\lfloor P^{1/k} \rfloor + 1}^n i^k = \frac{n^{k+1}}{k+1} + O(n^k) - \frac{\lfloor P^{1/k} \rfloor^{k+1}}{k+1} - O(\lfloor P^{1/k} \rfloor)$$

Let us define  $y \triangleq P^{1/k}$ . If  $n \gg 1$ , then we have

$$\beta = \frac{n^{k+1}}{k+1} - \frac{y^{k+1}}{k+1}$$

From (2.9) we have

$$P^* = y^k = \frac{\beta}{\alpha} = \frac{\frac{n^{k+1}}{k+1} - \frac{y^{k+1}}{k+1}}{y}$$

hence,

$$y = \left(\frac{1}{k+2}\right)^{\frac{1}{k+1}} n$$

Therefore

$$\frac{P}{n^k} = \left(\frac{1}{k+2}\right)^{\frac{k}{k+1}}$$

When power is defined as in (1.6), we have from (2.10) that

$$y^k = \frac{\frac{n^{k+1}}{k+1} - \frac{y^{k+1}}{k+1}}{ry}$$

It can easily be shown that

$$\frac{P^*}{n^k} = \left(\frac{1}{rk+r+1}\right)^{\frac{k}{k+1}}$$

Q.E.D.

Figure 2.4 shows  $\frac{P}{n^k}$  for  $k$  from 1 to 40.

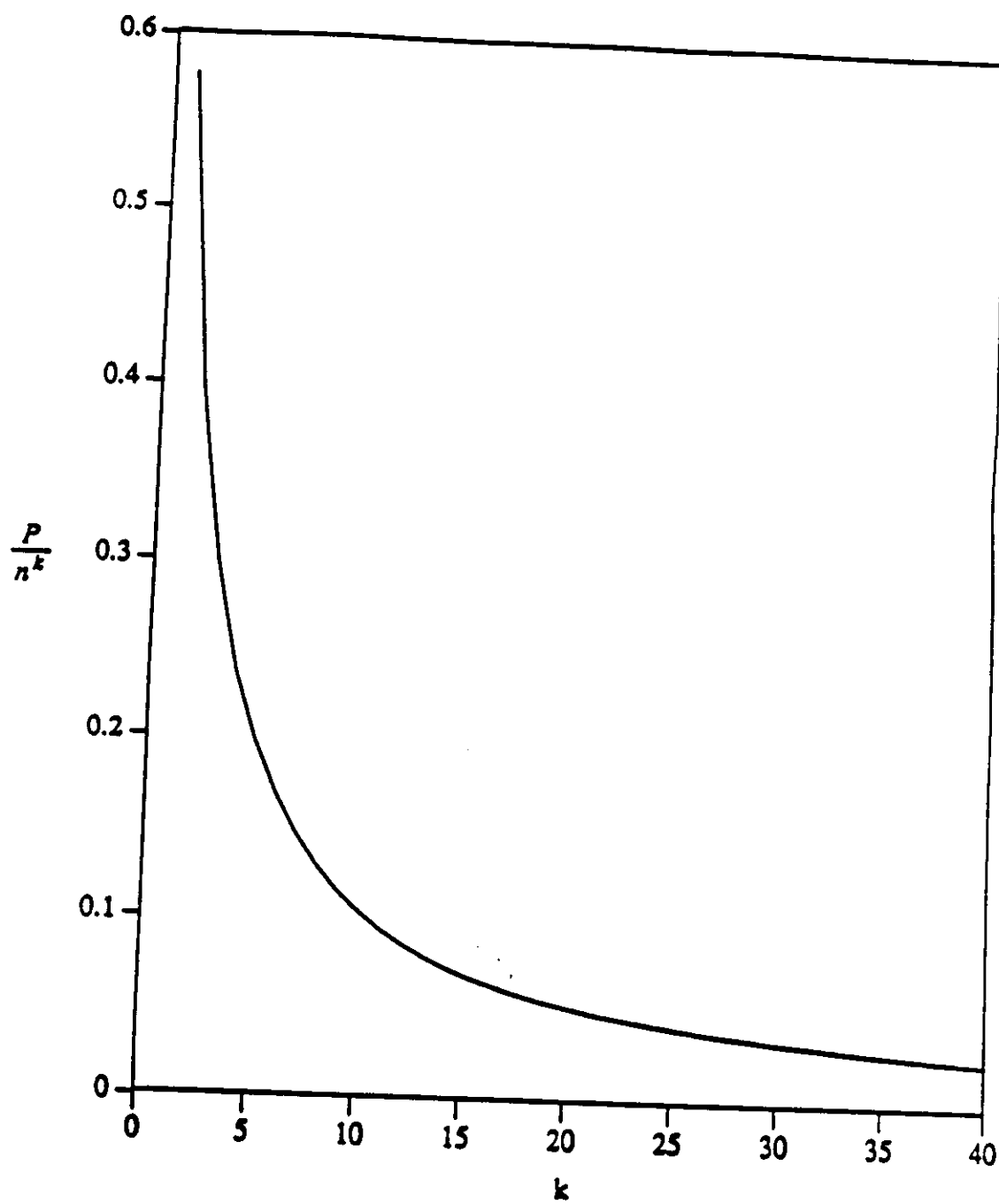


Figure 2.4  
 $\frac{P}{n^k}$  for  $k$  from 1 to 40 and  $r = 1$

### 2.3.2 Model II

In Model II we first study a special case which has only two stages in a job. The result of this special case study allows more physical interpretation than the general case study. The general case study will be given afterwards.

#### 2.3.2.1 A Special Case: Jobs with Two Stages

Assuming a job consists of  $M$  (deterministic) tasks in which a portion " $f$ " of all tasks ( $0 \leq f \leq 1$ ) has to be done serially and the remaining portion  $(1-f)$  can be done concurrently. This is equivalent to  $\vec{f} = [f, 1-f]$  and  $\vec{P} = [1, P]$ . The processing time speedup of this model given  $P$  processors was found by Amdahl [AMDA67] as

$$S_p(P) = \frac{P}{fP + 1 - f}$$

This expression says that the processing time speedup of the system depends very much on the characteristics of the job ( $f$ ); the processing time speedup can be much smaller than the number of processors used if " $f$ " is large.

It can easily be shown that the mean service time equals:

$$\bar{x}(P) = Mf + \frac{(1-f)M}{P} \quad (2.11)$$

During the entire service time ( $\bar{x}(P)$ ), the processing capability is  $P\bar{x}(P)$  while the work completed is simply  $M$  (since the service time for each task is 1). Hence, the processor utilization equals

$$u_1(P) = \frac{M}{P\bar{x}(P)} = \frac{1}{Pf + 1 - f} \quad (2.12)$$

Defining power as in (1.5), we have

$$\Pi^{(1)}(P) = \frac{u_1(P)}{\bar{x}(P)} = \frac{1}{M} \cdot \frac{P}{(Pf + 1 - f)^2}$$

Setting  $\frac{d\Pi^{(1)}(P)}{dP} = 0$  we have

$$P^* = \frac{1-f}{f} \quad (2.13)$$

It can be shown that  $\frac{d^2}{dP^2} \Pi^{(1)}(P^*) < 0$ ; hence, this  $P^*$  maximizes power. Note that  $P^*$  cannot be smaller than 1; therefore,  $P^* = 1$  if  $\frac{1-f}{f}$  is smaller than 1.

This result matches our intuition. It says that when most of the tasks have to be done serially (a larger  $f$ ),  $P^*$  will tend to be smaller since even with more processors, they will still be wasted for most of the time. Similarly, if most of the tasks can be processed concurrently (a smaller  $f$ ),  $P^*$  will tend to be larger since we can then achieve a higher concurrency.

If we define power as in (1.6), we have

$$\Pi^{(r)}(P) = \frac{[u_1(P)]^r}{\bar{x}(P)} = \frac{1}{M} \cdot \frac{P}{(Pf + 1 - f)^{r+1}}$$

Optimizing  $\Pi^{(r)}(P)$  with respect to  $P$ , we have

$$P^* = \frac{1-f}{f} \tag{2.14}$$

Note that  $P^*$  cannot be smaller than 1; therefore,  $P^* = 1$  if  $\frac{1-f}{f}$  is smaller than 1. From these results, we arrive at the following theorem.

**THEOREM 2.2:**

If a job has  $M$  tasks in two stages with the fraction vector  $\vec{f} = [f, 1-f]$  and the processor vector  $\vec{P} = [1, P]$ , the processing time speedup for any  $P$  is given as:

$$S_p(P) = \frac{P}{fP + 1 - f}$$

Power, as defined in (1.5), is maximized when

$$P^* = \begin{cases} 1 & \text{if } \frac{1-f}{f} \leq 1 \\ \frac{1-f}{f} & \text{if } \frac{1-f}{f} > 1 \end{cases}$$

If power is defined as in (1.6), it is maximized when

$$P^* = \begin{cases} 1 & \text{if } \frac{1-f}{f} \leq 1 \\ \frac{1-f}{f} & \text{if } \frac{1-f}{f} > 1 \end{cases}$$

**2.3.2.2 A General Case**

For the general case, we assume a job has  $M$  tasks (deterministic) and the adjusted fraction vector and the adjusted processor vector are

$$\vec{f} = [f_1, f_2, f_3, \dots, f_n]$$

$$\vec{P} = [P_1, P_2, P_3, \dots, P_n]$$

where  $P_i < P_{i+1}$  for  $1 < i < n-1$  and  $\sum_{i=1}^n f_i = 1$ .

We assume there are  $P$  processors available in the system. We find the index "m" such that  $P_{m-1} \leq P < P_m$  if  $P_1 \leq P < P_n$ ; or,  $m = n + 1$  if  $P \geq P_n$ ; or,  $m = 1$  if  $P < P_1$ . The mean service time can then be found as

$$\bar{x}(P) = M \left( \sum_{i=1}^{m-1} \frac{f_i}{P_i} + \frac{1}{P} \sum_{i=m}^n f_i \right)$$

Let us define the following:

$$\alpha = \sum_{i=1}^{m-1} \frac{f_i}{P_i} \quad (2.15)$$

$$\beta = \sum_{i=m}^n f_i \quad (2.16)$$

hence, we have

$$\bar{x}(P) = M \left( \alpha + \frac{1}{P} \beta \right)$$

The processing time speedup with  $P$  processors can easily be derived as:

$$S_p(P) = \frac{M}{\bar{x}(P)} = \frac{1}{\alpha + \frac{\beta}{P}} \quad (2.17)$$

The highest possible processing time speedup, denoted as  $S_{p,max}$ , of this model can be achieved when  $P \geq P_m$ , which leads to  $\alpha = \sum_{i=1}^m \frac{f_i}{P_i}$  and  $\beta = 0$ . Hence,

$$S_{p,max} = \frac{M}{\sum_{i=1}^m \frac{f_i M}{P_i}} = \frac{1}{\sum_{i=1}^m \frac{f_i}{P_i}} \quad (2.18)$$

This is a generalization of Amdahl's result [AMDA67]! If power is defined as in (1.5), we have power as

$$\Pi^{(1)}(P) = \frac{u_1(P)}{\bar{x}(P)} = \frac{M}{P \bar{x}(P)^2} = \frac{M}{\alpha^2 P + 2\alpha\beta + \frac{\beta^2}{P}}$$

Hence,

$$\frac{d\Pi^{(1)}(P)}{dP} = -M \cdot \frac{(\alpha + \frac{\beta}{P})(\alpha - \frac{\beta}{P})}{(\alpha^2 P + 2\alpha\beta + \frac{\beta^2}{P})^2}$$

Note that  $\Pi^{(1)}(P)$  in this section has the same characteristics as described in section 2.3.1. Therefore, using the same analysis in section 2.3.1 we are able to find the optimal number of processors in order to maximize the power (if  $P_{m-1} < P^* < P_m$ ) as

$$P^* = \frac{\beta}{\alpha}$$

If  $P^* = P_m$  ( $1 < m < n$ ), power is maximized when the following condition is met:

$$\left(\alpha - \frac{\beta}{P_m}\right) \left(\alpha' - \frac{\beta'}{P_m}\right) < 0$$

where  $\alpha' \triangleq \sum_{i=1}^m \frac{f_i}{P_i}$  and  $\beta' \triangleq \sum_{i=m+1}^n f_i$ .

Similarly, if power is defined as in (1.6), we find

$$\Pi^{(r)}(P) = \frac{u_1(P)^r}{\bar{x}(P)} = \frac{M^r}{P^r \bar{x}(P)^{r+1}}$$

We find the optimal number of processors to be used such that power is maximized as (if  $P_{m-1} < P^* < P_m$ ):

$$P^* = \frac{\beta}{r\alpha}$$

If  $P^* = P_m$  ( $1 < m < n$ ), power is maximized when the following condition is met:

$$(r\alpha P_m - \beta)(r\alpha' P_m - \beta') < 0$$

Note that  $P^*$  and  $S_p(P)$  do not depend on  $M$ . From these results, we arrive at the following theorem.

**THEOREM 2.3:**

Given  $\vec{f}$  and  $\vec{P}$  (and for  $\alpha$  and  $\beta$  as defined in (2.15) and (2.16) respectively), the processing time speedup for any  $P$  is given by

$$S_p(P) = \frac{1}{\alpha + \frac{\beta}{P}}$$

Power, as defined in (1.5), is maximized when (if  $P_{m-1} < P^* < P_m$ )

$$P^* = \frac{\beta}{\alpha}$$

If  $P^* = P_m$  ( $1 < m < n$ ), power is maximized when the following condition is met:

$$\left(\alpha - \frac{\beta}{P_m}\right) \left(\alpha' - \frac{\beta'}{P_m}\right) < 0$$

If power is defined as in (1.6), it is maximized when (if  $P_{m-1} < P^* < P_m$ )

$$P^* = \frac{\beta}{r\alpha}$$

If  $P^* = P_m$  ( $1 < m < n$ ), power is maximized when the following condition is met:

$$(r\alpha P_m - \beta)(r\alpha' P_m - \beta') < 0$$

### 2.3.3 Model III

We assume  $R(t)$  to have the following three properties: (1)  $R(t)$  is continuous and everywhere differentiable, (2)  $R(t)$  is monotonically increasing, and (3)  $R(t)$  is a one-to-one mapping. We define:

$a(P) \triangleq$  length of the interval from when a job first begins service until it first requires more processors than the system supplies, i.e.,  $a(P) = \min(t: R(t) > P)$  (see Figure 2.3).

$b \triangleq$  the service time of the job if the number of processors in the system is always greater than the number of processors required by the job (see Figure 2.3).

$$I(P) \triangleq \int_{a(P)}^b R(t) dt$$

**THEOREM 2.4:** Power, defined as  $\frac{u_1(P)}{\bar{x}(P)}$ , is maximized when

$$P^* = \frac{I(P^*)}{a(P^*)}$$

[Proof] From the definitions, we have

$$\bar{x}(P) = a(P) + \int_{a(P)}^b \frac{R(t)}{P} dt = a(P) + \frac{I(P)}{P} \quad (2.19)$$

Define  $u_1(P)$  to be the efficiency of the processors. We have

$$u_1(P) = \frac{\int_0^b R(t) dt}{P\bar{x}(P)} = \frac{W}{P\bar{x}(P)} \quad (2.20)$$

where  $W = \bar{x}(1)$  = total number of seconds of work required by a job. Thus, power becomes

$$\Pi^{(1)}(P) = \frac{u_1(P)}{\bar{x}(P)} = \frac{W}{P[\bar{x}(P)]^2} \quad (2.21)$$

Maximizing power with respect to  $P$ , we require

$$\frac{d}{dP} [P\bar{x}(P)^2] = 0$$

which leads to

$$\bar{x}(P) = -2P \frac{d\bar{x}(P)}{dP} \quad (2.22)$$

But from Eq. (2.19) we have

$$\frac{d\bar{x}(P)}{dP} = \frac{da(P)}{dP} + \frac{P \frac{dI(P)}{dP} - I(P)}{P^2} \quad (2.23)$$

hence, substituting equations (2.19) and (2.23) into equation (2.22), we have



$$a(P) = -2P \left[ \frac{da(P)}{dP} + \frac{1}{P} \frac{dI(P)}{dP} \right] + \frac{2I(P)}{P} - \frac{I(P)}{P}$$

Solving for  $P$ , the optimal value of  $P$  must be such that

$$P = \frac{I(P)}{a(P)} - \frac{2P^2}{a(P)} \left[ \frac{da(P)}{dP} + \frac{1}{P} \frac{dI(P)}{dP} \right] \quad (2.24)$$

Note that

$$\frac{dI(P)}{dP} = \frac{dI(P)}{da(P)} \frac{da(P)}{dP}$$

Now,

$$I(P) = \int_{a(P)}^b R(t) dt$$

therefore

$$\frac{dI(P)}{da(P)} = -\Pi^{(1)}(a(P))$$

But  $a(P)$  is such that  $\Pi^{(1)}(a(P)) = P$ , thus  $\frac{dI(P)}{da(P)} = -P$  and so  $\frac{dI(P)}{dP} = -P \frac{da(P)}{dP}$ ; therefore

$$\frac{da(P)}{dP} + \frac{1}{P} \frac{dI(P)}{dP} = 0 \quad (2.25)$$

From equations (2.24) and (2.25) we see that the optimal value,  $P^*$ , is

$$P^* = \frac{I(P^*)}{a(P^*)}$$

It can easily be shown that  $\frac{d^2}{dP^2} \Pi^{(1)}(P^*) < 0$ ; therefore,  $P^* = \frac{I(P^*)}{a(P^*)}$  indeed maximizes power.

Q.E.D.

**Corollary 2.3:** Power, defined as  $\frac{u_1(P)}{\bar{x}(P)}$ , is maximized if and only if  $\bar{x}(P^*) = 2a(P^*)$ .

[Proof] Since  $\frac{I(P^*)}{P^*} = a(P^*)$ , hence  $\bar{x}(P^*) = a(P^*) + a(P^*) = 2a(P^*)$

Q.E.D.

Let us look at what  $\bar{x}(P^*) = 2a(P^*)$  means physically. From Figure 2.3 we know that  $a(P)$  is the portion of the service time when the job has enough processors than it needs. Therefore,  $\bar{x} = 2a(P)$  means that the portion of the service time when there are enough processors equals the portion of the service time when there are not enough processors for it needs. Let us define  $a(P)$  to be the "unextended service time" and  $\frac{I(P)}{P}$  to be the "extended service time". This corollary states that the "unextended service time"

equals the "extended service time" when power is maximized. Also note that during the unextended service time period the processors are not fully utilized (*processor utilization* < 1) while during the extended service time period the processors are fully utilized (*processor utilization* = 1). Therefore, the service time period for *processor utilization* < 1 equals the service time period for *processor utilization* = 1.

**THEOREM 2.5:** Power, defined as  $\frac{u_1(P)^r}{\bar{x}(P)}$ , is maximized when

$$P^* = \frac{I(P^*)}{ra(P^*)}$$

[Proof] The proof can easily be derived following the procedure given in the proof for Theorem 1.

Q.E.D.

**Corollary 2.4:** Power, defined as  $\frac{u_1(P)^r}{\bar{x}(P)}$ , is maximized when  $\bar{x}(P^*) = (r+1)a(P^*)$ .

In THEOREM 1 we require  $R(t)$  to satisfy three constraints. Unfortunately, all three constraints will be violated if we want to apply this result in discrete cases, which are more practical ones. In the following Theorem, we will show that Corollary 2.3 still holds even when some of the constraints for  $R(t)$  are violated.

**Example 2.3:** If  $R(t)$  is a linear function, i.e.,  $R(t) = \frac{B}{b}t$  for  $0 \leq t \leq b$ , and power is defined as in (1.6), then we have

$$P^* = \frac{B}{\sqrt{2r+1}}$$

or

$$\frac{P^*}{\text{Maximum number of processors required}} = \frac{1}{\sqrt{2r+1}}$$

[Proof] From  $R(t)$  we have

$$a(P) = \frac{b}{B}P$$

$$I(P) = \int_0^b \frac{B}{bP} t dt = \frac{Bb}{2} - \frac{b}{2B}P^2$$

Hence,

$$P^* = \frac{I(P^*)}{ra(P^*)} = \frac{B^2 - P^{*2}}{2rP^*}$$

Solving for  $P^*$ , we have

$$P^* = \frac{B}{\sqrt{2r+1}}$$

Note that  $B$  is the maximum number of processors required by the job, therefore,

$$\frac{P^*}{\text{Maximum number of processors required}} = \frac{1}{\sqrt{2r+1}}$$

Q.E.D.

In this example, by setting  $r = 1$ , we have  $\frac{P^*}{B} = \frac{1}{\sqrt{3}} \approx 58\%$  (where  $B$  is the maximum number of processors required). This is the case approximated in [KUNG84]; here we have the exact value of  $P^*$ .

**Example 2.4:** If  $R(t) = \frac{B}{b^n} t^n$  for  $0 \leq t \leq b$  and power is defined as in (1.6), we have

$$P^* = \frac{B}{[(n+1)r+1]^{\frac{n}{n+1}}}$$

or

$$\frac{P^*}{\text{Maximum number of processors required}} = \frac{1}{[(n+1)r+1]^{\frac{n}{n+1}}}$$

**[Proof]** This proof is similar to the proof in Example 2.3.

### 2.3.4 An Iterative Procedure

In Corollary 2.1 and Corollary 2.2, we studied the cases when  $P_i$  and  $t_i$  can be formulated in a nice form such that  $P^*$  can be found mathematically by simply solving an equation. However, there are cases when  $P_i$  and  $t_i$  cannot be expressed nicely as a function. Hence, we are not able to find  $P^*$  by mathematically solving an equation for these cases. In this section, we introduce an iterative procedure which solves the problem just described. The procedure will first select an arbitrary index  $m$  and find  $P^*$  for this value of  $m$  assuming  $P_{m-1} \leq P^* < P_m$ . If  $P^*$  has a value between  $P_{m-1}$  and  $P_m$  as it should be, then this  $P^*$  is the solution. On the other hand, if  $P^* > P_m$ , which violates the assumption, then we choose a higher value of  $m$  and repeat the same procedure. Similarly, if  $P^* < P_{m-1}$ , then we choose a lower value of  $m$  and repeat the same procedure. However, note that  $\frac{d\bar{x}(P)}{dP} = \frac{d}{dP}(\alpha + \frac{\beta}{P}) = -\frac{\beta}{P^2}$  is not a continuous function at the intersections between stages since the value of  $\beta$  would have a jump between stages; therefore, if the optimal  $P^*$  occurs exactly at the intersection between stages, then  $P^*$  will always be out of the range. In this case, the procedure described above will arrive at a situation such that when  $m = k$ ,  $P^*$  is greater than  $P_k$  and when  $m = k + 1$ ,  $P^*$  is smaller than  $P_k$ . When this case happens,  $P_k$  is the value for  $P^*$ . The algorithm can be written as follows:

(1)

$lid = 1;$             / lid stands for lower index. /  
 $hid = n;$             / hid stands for higher index. /

$$(2) \quad m = \left\lfloor \frac{lid + hid}{2} \right\rfloor$$

where  $\lfloor x \rfloor$  is defined as the largest integer which is smaller than  $x$ .

(3) According to  $m$ , find  $\alpha$  and  $\beta$  according to the model (I or II) used.  
 $P^* = \frac{\beta}{\alpha}$

(4) if  $P^* > P_n$   
     then  $lid = m$   
     else if  $P^* < P_{n-1}$   
         then  $hid = m$ , goto (2)  
         else STOP.        /  $P^*$  is found. /  
     endif  
 endif  
 if  $hid = lid + 1$   
     then STOP        /  $P^*$  is found. /  
     else goto (2)  
 endif

This procedure is obviously an application of binary search; hence, the number of steps for this procedure is upper bounded by  $\log_2 n$ , where  $n$  is the number of stages in the job.

Example 2.5:

Let us use an example to show how this procedure works. This example is a special case of Corollary 2.2. Assuming  $n = 100$ , we have

$$P_i = i^2; \quad 1 \leq i \leq 100$$

and

$$t_i = 1; \quad 1 \leq i \leq 100$$

Step 1:

$m = \left\lfloor \frac{100}{2} \right\rfloor = 50$ . With  $m = 50$  we have  $49^2 \leq P < 50^2$ , or,  $2401 \leq P < 2500$ .

$$\alpha = \sum_{i=1}^{49} 1 = 49$$

$$\beta = \sum_{i=50}^{100} i^2 = 297,925$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{297925}{49} = 6080$$

Since  $P^*$  is too high, we have to adjust  $m$  to be higher.

Step 2:

$$m = \left\lfloor \frac{50 + 100}{2} \right\rfloor = 75. \text{ With } m = 75 \text{ we have } 74^2 \leq P < 75^2, \text{ or, } 5476 \leq P < 5625.$$

$$\alpha = \sum_{i=1}^{74} 1 = 74$$

$$\beta = \sum_{i=75}^{100} i^2 = 200,525$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{200525}{74} = 2709$$

Since  $P^*$  is too low, we have to adjust  $m$  to be lower.

Step 3:

$$m = \left\lfloor \frac{50 + 75}{2} \right\rfloor = 62. \text{ With } m = 62 \text{ we have } 61^2 \leq P < 62^2, \text{ or, } 3721 \leq P < 3844.$$

$$\alpha = \sum_{i=1}^{61} 1 = 61$$

$$\beta = \sum_{i=62}^{100} i^2 = 260,819$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{260819}{61} = 4275$$

Since  $P^*$  is too high, we have to adjust  $m$  to be higher.

Step 4:

$$m = \left\lfloor \frac{62 + 75}{2} \right\rfloor = 68. \text{ With } m = 68 \text{ we have } 67^2 \leq P < 68^2, \text{ or, } 4489 \leq P < 4624.$$

$$\alpha = \sum_{i=1}^{67} 1 = 67$$

$$\beta = \sum_{i=68}^{100} i^2 = 235,840$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{235840}{67} = 3520$$

Since  $P^*$  is too low, we have to adjust  $m$  to be lower.

Step 5:

$$m = \left\lfloor \frac{62 + 68}{2} \right\rfloor = 65. \text{ With } m = 65 \text{ we have } 64^2 \leq P < 65^2, \text{ or, } 4096 \leq P < 4225.$$

$$\alpha = \sum_{i=1}^{64} 1 = 64$$

$$\beta = \sum_{i=65}^{100} i^2 = 248,910$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{248910}{64} = 3889$$

Since  $P^*$  is too low, we have to adjust  $m$  to be lower.

Step 6:

$$m = \left\lfloor \frac{62 + 65}{2} \right\rfloor = 63. \text{ With } m = 63 \text{ we have } 62^2 \leq P < 63^2, \text{ or, } 3844 \leq P < 3969.$$

$$\alpha = \sum_{i=1}^{62} 1 = 62$$

$$\beta = \sum_{i=63}^{100} i^2 = 256,975$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{256975}{62} = 4145$$

Since  $P^*$  is too high, we have to adjust  $m$  to be higher.

Step 7:

$$m = \left\lceil \frac{63 + 65}{2} \right\rceil = 64. \text{ With } m = 64 \text{ we have } 63^2 \leq P < 64^2, \text{ or, } 3969 \leq P < 4096.$$

$$\alpha = \sum_{i=1}^{63} 1 = 63$$

$$\beta = \sum_{i=64}^{100} i^2 = 253,006$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{253006}{63} = 4016$$

Since  $P^* = 4016$  is between 3969 and 4096,  $P^* = 4016$  is the number of processors to use in order to maximize power. Note that the maximum processors required is  $100^2 = 10,000$ , hence  $\frac{P^*}{n^2} = \frac{4016}{10000} = 40.2\%$ , which agrees with the result given in Corollary 2.2 (i.e.,  $\frac{P^*}{n^2} = \left(\frac{1}{2+2}\right)^{\frac{2}{2+1}} = 39.7\%$ ).

Example 2.6:

We use this example to show the procedure when  $P^*$  is one of the  $P_k$  in the processor vector. We use the same example as in Example 2.5 except that we have  $n = 50$  in this example.

Step 1:

$$m = \left\lfloor \frac{50}{2} \right\rfloor = 25. \text{ With } m = 25 \text{ we have } 24^2 \leq P < 25^2, \text{ or, } 576 \leq P < 625.$$

$$\alpha = \sum_{i=1}^{24} 1 = 24$$

$$\beta = \sum_{i=25}^{50} i^2 = 38025$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{38025}{24} = 1584$$

Since  $P^*$  is too high, we have to adjust  $m$  to be higher.

Step 2:

$$m = \left\lfloor \frac{25 + 50}{2} \right\rfloor = 37. \text{ With } m = 37 \text{ we have } 36^2 \leq P < 37^2, \text{ or, } 1296 \leq P < 1369.$$

$$\alpha = \sum_{i=1}^{36} 1 = 36$$

$$\beta = \sum_{i=37}^{50} i^2 = 26719$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{26719}{36} = 742$$

Since  $P^*$  is too low, we have to adjust  $m$  to be lower.

Step 3:

$$m = \left\lfloor \frac{25 + 37}{2} \right\rfloor = 31. \text{ With } m = 31 \text{ we have } 30^2 \leq P < 31^2, \text{ or, } 900 \leq P < 961.$$

$$\alpha = \sum_{i=1}^{30} 1 = 30$$

$$\beta = \sum_{i=31}^{50} i^2 = 33470$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{33470}{30} = 1116$$

Since  $P^*$  is too high, we have to adjust  $m$  to be higher.

Step 4:

$$m = \left\lceil \frac{31 + 37}{2} \right\rceil = 34. \text{ With } m = 34 \text{ we have } 33^2 \leq P < 34^2, \text{ or, } 1089 \leq P < 1156.$$

$$\alpha = \sum_{i=1}^{33} 1 = 33$$

$$\beta = \sum_{i=34}^{50} i^2 = 30396$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{30396}{33} = 921$$

Since  $P^*$  is too low, we have to adjust  $m$  to be lower.

Step 5:

$$m = \left\lfloor \frac{31 + 34}{2} \right\rfloor = 32. \text{ With } m = 32 \text{ we have } 31^2 \leq P < 32^2, \text{ or, } 961 \leq P < 1024.$$

$$\alpha = \sum_{i=1}^{31} 1 = 31$$

$$\beta = \sum_{i=32}^{50} i^2 = 32509$$

From the  $\alpha$  and the  $\beta$  derived above, we have



$$P^* = \frac{32509}{31} = 1049$$

Since  $P^*$  is too high, we have to adjust  $m$  to be higher.

Step 6:

$$m = \left\lceil \frac{32 + 34}{2} \right\rceil = 33. \text{ With } m = 33 \text{ we have } 32^2 \leq P < 33^2, \text{ or, } 1024 \leq P < 1089.$$

$$\alpha = \sum_{i=1}^{32} 1 = 32$$

$$\beta = \sum_{i=33}^{50} i^2 = 31485$$

From the  $\alpha$  and the  $\beta$  derived above, we have

$$P^* = \frac{31485}{32} = 984$$

Since  $P^*$  is too low, we have to adjust  $m$  to be lower.

From Step 5 we have  $P \geq 32$  while from Step 6 we have  $P \leq 32$ , therefore, we have  $P^* = 32^2 = 1024$ . We have  $\frac{P^*}{n^2} = \frac{1024}{2500} = 41\%$ , which again agrees with the result given in Corollary 2.2.

## 2.4 Systems with Queueing

In this section, queueing is allowed; hence, power is defined as either in (1.7) or in (1.8). We assume that the jobs come to the system according to a Poisson process with a rate  $\lambda$ . Besides finding  $P^*$ , we also find the optimal operating point  $\lambda^*$  which, in combination with  $P^*$ , will achieve the maximum power. An expression for the response time speedup is also derived.

### 2.4.1 Model I

For a system which admits only one job into service at a time, we can arbitrarily interchange the stages without affecting the service time (hence the system utilization) and the processor utilization. For the convenience of computation, once again we rearrange the processor vector such that the number of processors required in each stage is non-decreasing. The time vector will also be adjusted accordingly. We assume the processor vector of a job after rearrangement is given by

$$\vec{P} = [P_1, P_2, P_3, \dots, P_n]$$

where  $P_i \leq P_{i+1}$  for  $1 \leq i \leq n-1$ . The corresponding time vector and standard deviation are given by

$$\vec{T} = [t_1, t_2, t_3, \dots, t_n]$$

$$\vec{\sigma} = [\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_n]$$

We assume there are  $P$  processors available in the system. We find " $m$ " such that  $P_{m-1} \leq P < P_m$  if  $P_1 \leq P < P_n$ ; or,  $m = n + 1$  if  $P \geq P_n$ ; or,  $m = 1$  if  $P < P_1$ . Denote the variance of the service time of the job, given  $P$  processors in the system, as  $\sigma_{z_p}^2$ , which can be found as:

$$\sigma_{z_p}^2 = \sum_{i=1}^{m-1} \sigma_i^2 + \frac{1}{P^2} \sum_{i=m}^n \sigma_i^2 P_i^2$$

We define  $\alpha$ ,  $\beta$ , and  $\gamma$  as before and we further define

$$a \triangleq \sum_{i=1}^{m-1} \sigma_i^2$$

$$b \triangleq \sum_{i=m}^n \sigma_i^2 P_i^2$$

Hence,

$$\bar{x}(P) = \alpha + \frac{1}{P} \beta$$

$$\sigma_{z_p}^2 = a + \frac{1}{P^2} b$$

The coefficient of variation of the service time can be found as:

$$c_{z_p}^2 = \frac{\sigma_{z_p}^2}{\bar{x}(P)^2} = \frac{aP^2 + b}{\alpha^2 P^2 + 2\alpha\beta P + \beta^2}$$

Denote  $\rho$  as the system utilization (i.e., the fraction of time there is at least one job in the system), we have

$$\rho = \lambda \bar{x}(P)$$

and the processor utilization can be found as

$$u_2(\lambda, P) = \frac{\lambda \gamma}{P}$$

Using results from M/G/1 [KLEI75], we have

$$T_1(P, \rho) = \bar{x}(P) \left[ 1 + \rho \frac{1 + c_{z_p}^2}{2(1 - \rho)} \right]$$

With the above expression for  $T_1(P, \rho)$ , we can find the mean system time when the number of processors in the system is 1 or  $P$ . Hence, the response time speedup is

$$S_r(P, \rho) = \frac{\gamma}{\alpha + \frac{\beta}{P}} \cdot \frac{1 + \rho \frac{1 + c_{x_1}^2}{2(1-\rho)}}{1 + \rho \frac{1 + c_{x_p}^2}{2(1-\rho)}} = \frac{\gamma}{\alpha + \frac{\beta}{P}} \cdot \frac{2 - \rho + \rho c_{x_1}^2}{2 - \rho + \rho c_{x_p}^2}$$

where

$$c_{x_1}^2 = \frac{\sum_{i=1}^n \sigma_i^2 P_i^2}{\gamma^2}$$

Note that when the service time is deterministic for all stages, i.e.,  $\sigma_i = 0$  for all  $i$ , then the response time speedup equals the processing time speedup as given in (2.8), i.e.,  $S_r(P, \rho) = \frac{\gamma}{\alpha + \frac{\beta}{P}}$

Defining power as in (1.7), we have

$$\Pi_2^{(1)}(\lambda, P) = \frac{u_2(\lambda, P)}{T_2(\lambda, P)} = \frac{2\gamma}{P\bar{x}(P)} \frac{\lambda - \bar{x}(P)\lambda^2}{2(1 - \lambda\bar{x}(P)) + \lambda\bar{x}(P)(1 + c_{x_p}^2)}$$

Optimizing  $\Pi_2^{(1)}(\lambda, P)$  with respect to  $\lambda$ , we have

$$\lambda^* = \frac{2}{2 + \sqrt{2 + 2c_{x_p}^2}} \cdot \frac{1}{\bar{x}(P)} \quad (2.26)$$

and

$$\rho^* = \frac{2}{2 + \sqrt{2 + 2c_{x_p}^2}}$$

Using Little's result [LITT61], we have

$$\bar{N}^* = 1$$

Note that this is the same result as derived in [KLEI79]. Therefore, the intuitive explanation given in [KLEI79] also applies here. That is, the proper operating point ( $\lambda^*$ ) for the system is when there is exactly one job in service and no others are waiting for service since only one job is admitted into service at a time. Substituting (2.26) into  $\Pi_2^{(1)}(\lambda, P)$  we have

$$\Pi_2^{(1)}(\lambda^*, P) = \gamma \frac{\lambda^{*2}}{P}$$

Hence,

$$\frac{d}{dP} \Pi_2^{(1)}(\lambda^*, P) = \frac{\gamma\lambda^*}{P^2} \left[ 2P \frac{d\lambda^*}{dP} - \lambda^* \right] \quad (2.27)$$

Optimizing  $\Pi_2^{(1)}(\lambda^*, P)$  with respect to  $P$  and assuming  $P_{n-1} < P^* < P_n$ , we have the following condition:

$$\frac{d\lambda^*}{dP} = \frac{1}{2} \frac{\lambda^*}{P^*} \quad (2.28)$$

Defining

$$f_1(P) = 2 + \left[ 2 + 2 \frac{aP^2 + b}{(\alpha P + \beta)^2} \right]^{1/2} \quad (2.29)$$

and

$$f_2(P) = \alpha P + \beta \quad (2.30)$$

we have

$$\lambda^* = \frac{2}{f_1(P)} \frac{P}{f_2(P)}$$

Hence,

$$\frac{d}{dP} \lambda^* = \frac{2f_1(P)f_2(P) - 2P \left[ f_2(P) \frac{df_1(P)}{dP} + f_1(P) \frac{df_2(P)}{dP} \right]}{f_1(P)^2 f_2(P)^2} \quad (2.31)$$

From (2.28), we have to solve the following equation numerically to get  $P^*$ :

$$f_1(P^*)f_2(P^*) - 2P^*f_2(P^*) \frac{d}{dP} f_1(P^*) - 2\alpha P^*f_1(P^*) = 0 \quad (2.32)$$

where

$$\frac{d}{dP} f_1(P^*) = \frac{-\sqrt{2}(ab - a\beta P^*)}{(\alpha P^* + \beta)^2 \left[ 1 + \frac{aP^{*2} + b}{(\alpha P^* + \beta)^2} \right]^{1/2}}$$

However, if  $P^* = P_m$ ,  $\frac{d}{dP} \Pi_2^{(1)}(\lambda^*, P)$  will change sign at  $P_m$ . Hence, from (2.27), the following condition for  $P^*$  has to be met:

$$\left[ 2P \frac{d\lambda^*}{dP} - \lambda^* \right]_{\alpha, \beta, a, b} \cdot \left[ 2P \frac{d\lambda^*}{dP} - \lambda^* \right]_{\alpha', \beta', a', b'} < 0$$

where  $\left[ 2P \frac{d\lambda^*}{dP} - \lambda^* \right]_{\alpha', \beta', a', b'}$  stands for replacing all  $\alpha$ ,  $\beta$ ,  $a$ , and  $b$  by  $\alpha'$ ,  $\beta'$ ,  $a'$ , and  $b'$  in the brackets.

$\alpha$ ,  $\beta$ ,  $a$ , and  $b$  are defined as above and  $\alpha'$ ,  $\beta'$ ,  $a'$ , and  $b'$  are defined as

$$\alpha' = \sum_{i=1}^m t_i$$

$$\beta' = \sum_{i=m+1}^n t_i P_i$$

$$a' = \Delta \sum_{i=1}^m \sigma_i^2$$

$$b' \triangleq \sum_{i=m+1}^n \sigma_i^2 P_i^2$$

**THEOREM 2.6:**

Given  $\vec{P}$ ,  $\vec{\tau}$  and  $\vec{\sigma}$  and  $P$  (and for  $\alpha$ ,  $\beta$ ,  $\gamma$ ,  $a$ ,  $b$ , and  $c_{x_p}$  as defined earlier), the response time speedup for any  $P$  is

$$S_r(P, \rho) = \frac{\gamma}{\alpha + \frac{\beta}{P}} \cdot \frac{2 - \rho + \rho c_{x_1}^2}{2 - \rho + \rho c_{x_p}^2} \quad \text{where } c_{x_1}^2 = \frac{\sum_{i=1}^n \sigma_i^2 P_i^2}{\gamma^2}$$

If power is defined as in (1.7), the optimal system operating point is

$$\lambda^* = \frac{2}{2 + \sqrt{2 + 2c_{x_p}^2}} \cdot \frac{1}{\bar{x}(P)}$$

such that

$$\bar{N}^* = 1$$

The optimal number of processors ( $P_{m-1} < P^* < P_m$ ) to be used should satisfy the following equation:

$$f_1(P^*)f_2(P^*) - 2P^*f_2(P^*) \frac{d}{dP} f_1(P^*) - 2\alpha P^* f_1(P^*) = 0$$

where  $f_1(P^*)$  and  $f_2(P^*)$  are defined as in (2.29) and (2.30) respectively.

If  $P^* = P_m$ , the following condition should be met:

$$\left[ 2P \frac{d\lambda^*}{dP} - \lambda^* \right]_{\alpha, \beta, a, b} \cdot \left[ 2P \frac{d\lambda^*}{dP} - \lambda^* \right]_{\alpha', \beta', a', b'} < 0$$

**Corollary 2.5:**

If  $\sigma_i = 0$  for all  $i$ ,

$$S_r(P, \rho) = \frac{\gamma}{\alpha + \frac{\beta}{P}}$$

If power is defined as in (1.7), then

$$\lambda^* = \frac{2 - \sqrt{2}}{\bar{x}(P)}$$

$$\bar{N}^* = 1$$

If  $P_{m-1} < P^* < P_m$  ( $1 \leq m \leq n$ ) and we define  $\alpha$  and  $\beta$  as shown in equations (2.6) and (2.7), we have

$$P^* = \frac{\beta}{\alpha} \quad (2.33)$$

If power is defined as in (1.8), then

$$\lambda^* = \frac{1 + 3r - \sqrt{r^2 - 6r + 1}}{2r\bar{x}(P)}$$

$$\bar{N}^* = \lambda^* T_2(\lambda^*, P) = \frac{r^2 - 4r - 1 + (r+1)\sqrt{r^2 + 6r + 1}}{2r(-r - 1 + \sqrt{r^2 + 6r + 1})}$$

Similarly, if  $P_{m-1} < P^* < P_m$  ( $1 \leq m \leq n$ ) and we define  $\alpha$  and  $\beta$  as shown in equations (2.6) and (2.7), we have

$$P^* = \frac{\beta}{r\alpha} \quad (2.34)$$

If  $r \gg 1$  we have

$$\lim_{r \rightarrow \infty} \bar{N}^* = \lambda^* T_2(\lambda^*, P) = \frac{r^2 - 4r - 1 + (r+1)\sqrt{r^2 + 6r + 1}}{2r(-r - 1 + \sqrt{r^2 + 6r + 1})} = \frac{r}{2}$$

[Proof]

Given  $\sigma_i = 0$  for all  $i$ , we have  $c_{s_1} = c_{s_2} = 0$ . Hence, we can easily prove the response time speedup from THEOREM 2.4. If power is defined as in (1.7), we have from (2.26) that

$$\lambda^* = \frac{2 - \sqrt{2}}{\bar{x}(P)}$$

Hence, from (2.27) we have

$$\Pi_2^{(1)}(\lambda^*, P) = \frac{(6 - 4\sqrt{2})\gamma}{P\bar{x}(P)^2}$$

From here we can easily prove (2.33). The rest of the lemma can be proved similarly.

Q.E.D.

Note from Corollary 2.5 that by defining power as in (1.7),  $P^* = \frac{\beta}{\alpha}$  is exactly the same as derived in (2.9) when queuing is not allowed. An intuitive explanation follows. In Corollary 2.5,  $P^*$  is achieved when  $\bar{N}^* = 1$ , i.e., the system is operating at a point such that, on the average, only one job is being served in the system and no others are waiting for service at the same time. Operating under such a condition, this system would behave as if queuing is not allowed. Therefore, the result achieved here should be the same as the result achieved when queuing is not allowed.

### 2.4.2 Model II and III

Models II and III will be studied in this section. The common characteristic of models II and III is that the coefficient of variation of the service time distribution for both models is not a function of  $P$ . This characteristic will be made clear later in this section. We assume the work brought by jobs is a random variable with a general distribution. Defining  $\hat{X}$  to be the random variable representing the service time distribution, we can show that in the continuous job model:

$$\hat{X} = \hat{K} \left[ a(P) + \int_{a(P)}^b \frac{f(t)}{P} dt \right] \quad (2.35)$$

Defining  $c_{x_p}$  to be the coefficient of variation of the service time given  $P$  processors, (2.35) shows that  $\hat{X}$  equals  $\hat{K}$  multiplied by a constant; hence,

$$c_{x_p} = c_K \quad (2.36)$$

Note from (2.36) that  $c_{x_p}$  is not a function of  $P$ ; this property will be used later. Defining  $\hat{W}$  to be the random variable representing the work brought by a job, we can show that

$$\hat{W} = \int_0^{\hat{K}b} f\left(\frac{t}{\hat{K}}\right) dt = \hat{K} \int_0^b f(t) dt \quad (2.37)$$

Since  $\int_0^b f(t) dt$  is a constant, (2.37) shows that the work brought by a job is a random variable which has the same coefficient of variation as  $\hat{K}$ .

#### 2.4.2.1 Finding the Response Time Speedup

In this section we will find the response time speedup for the discrete cases.

##### 2.4.2.1.1 The Discrete Case: Jobs with Two Stages

**THEOREM 2.7:** Given  $\vec{f} = [f, 1-f]$  and  $\vec{P} = [1, P]$ , the response time speedup is:

$$S_r(P, \rho) = \frac{P}{fP + 1 - f}$$

[Proof] Denote  $\hat{X}(P)$  as the random variable for the service time, we have

$$\hat{X}(P) = \hat{W}f + \frac{(1-f)\hat{W}}{P} = \hat{W} \left[ f + \frac{(1-f)}{P} \right] \quad (2.38)$$

Hence,

$$\bar{x}(P) = Wf + \frac{(1-f)W}{P} \quad \text{and} \quad c_{x_p} = c_W \quad (2.39)$$

We define  $\rho$  to be the system utilization; hence,

$$\rho = \lambda \bar{x}(P)$$

On the average,  $\lambda W$  seconds of work comes to the system per second and the capacity of the system is  $P$  seconds per second, therefore the average processor efficiency of the system can be found as:

$$u_2(\lambda, P) = \frac{\lambda W}{P}$$

Using the M/G/1 result [KLEI75] and (2.39), we have

$$T_1(P, \rho) = \bar{x}(P) \left[ 1 + \rho \frac{1 + c_w^2}{2(1-\rho)} \right] = \bar{x}(P) \left[ \frac{2 + (c_w^2 - 1)\rho}{2(1-\rho)} \right] \quad (2.40)$$

From (2.40), we find the response time speedup as

$$S_r(P, \rho) = \frac{T_1(1, \rho)}{T_1(P, \rho)} = \frac{\bar{x}(1)}{\bar{x}(P)} = S_p(P) = \frac{P}{fP + 1 - f}$$

Q.E.D.

Interestingly enough, this response time speedup when queueing is allowed is the same as the processing time speedup when queueing is not allowed. Therefore, the response time speedup and the processing time speedup are solely determined by the job specifications and  $P$  in our model.

#### 2.4.2.1.2 A General Case of Model II

We assume the number of tasks in a job is a random variable  $\hat{W}$  with mean  $W$  and coefficient of variation  $c_w$ . The fraction vector and the processor vector are given as:

$$\vec{f} = [f_1, f_2, f_3, \dots, f_n]$$

$$\vec{P} = [P_1, P_2, P_3, \dots, P_n]$$

where  $P_i < P_{i+1}$  for  $1 < i < n - 1$ .

**THEOREM 2.8:** Given  $\vec{f}$ ,  $\vec{P}$  and defining  $\alpha$  and  $\beta$  as in (14) and (15) respectively, the response time speedup for any  $P$  and  $\rho$  is given as

$$S_r(P, \rho) = \frac{1}{\alpha + \frac{\beta}{P}}$$

[Proof] Defining  $\rho$  to be the system utilization, we have

$$\rho = \lambda \bar{x}(P)$$

The response time speedup can be found using M/G/1 theory as:



$$S_r(P, \rho) = \frac{\bar{x}(1) \left[ 1 + \rho \frac{1 + c_{x_1}}{2(1 - \rho)} \right]}{\bar{x}(P) \left[ 1 + \rho \frac{1 + c_{x_p}}{2(1 - \rho)} \right]}$$

It can be shown that  $c_{x_1} = c_{x_p} = c_w$ , hence

$$S_r(P, \rho) = \frac{\bar{x}(1)}{\bar{x}(P)} = S_p(P) = \frac{1}{\alpha + \frac{\beta}{P}}$$

Q.E.D.

Again, this response time speedup (considering queueing) is not a function of  $\rho$  and is the same as the processing time speedup (considering no queueing).

#### 2.4.2.2 Finding the Optimal Arrival Rate

Since only one job can be admitted into service at a time, this system can be analyzed as a single server system. Hence, we can apply results from M/G/1 model [KLEI75] to find the average response time for this system. In this section, we find the optimal operating point ( $\lambda^*$ ) for both the discrete case and the continuous case. Amazingly, even though the definitions of power in this paper and in [KLEI79] are different (since  $\rho \neq u$ ), the results obtained in both papers are the same. Therefore, all the deterministic reasoning given in [KLEI79] also applies in this paper.

**THEOREM 2.9:** Power, defined as  $\frac{u_2(\lambda, P)}{T_2(\lambda, P)}$ , is maximized (given  $P$ ) when

$$\lambda^* = \frac{2}{2 + \sqrt{2 + 2c_{x_p}^2}} \cdot \frac{1}{\bar{x}(P)}$$

[Proof]:

$$u_2(\lambda, P) = \frac{\lambda W}{P}$$

From the M/G/1 results we have

$$T_2(\lambda, P) = \bar{x}(P) \left[ 1 + \rho \frac{1 + c_{x_p}^2}{2(1 - \rho)} \right] = \bar{x}(P) \left[ \frac{2 + (c_{x_p}^2 - 1)\lambda\bar{x}(P)}{2(1 - \lambda\bar{x}(P))} \right]$$

Defining power as in (1.7), we have

$$\Pi_2^{(1)}(\lambda, P) = \frac{u_2(\lambda, P)}{T_2(\lambda, P)} = \frac{\lambda W}{P} \cdot \frac{2(1 - \lambda\bar{x}(P))}{2 + (c_{x_p}^2 - 1)\lambda\bar{x}(P)}$$

By maximizing power with respect to  $\lambda$ , we have

$$\lambda^* = \frac{2}{2 + \sqrt{2 + 2c_{z_p}^2}} \cdot \frac{1}{\bar{x}(P)}$$

Q.E.D.

Corollary 2.6: When power is maximized with respect to  $\lambda$ ,

$$\rho^* = \frac{2}{2 + \sqrt{2 + 2c_{z_p}^2}}$$

and

$$\bar{N}^* = 1$$

[Proof] From Theorem 2.9 we can show that

$$\rho^* = \lambda^* \bar{x}(P) = \frac{2}{2 + \sqrt{2 + 2c_{z_p}^2}}$$

Using Little's result [LITT61], it is easy to show that

$$\bar{N}^* = \lambda^* T_2(\lambda^*, P) = 1$$

Q.E.D.

$\bar{N}^*$  is the deterministic reasoning given in [KLE179] which is described in the introduction in section I.

**THEOREM 2.10:** Power, defined as  $\frac{u_2(\lambda, P)^r}{T_2(\lambda, P)}$ , is maximized (given  $P$ ) when

$$\lambda^* = \frac{4r}{(-c_{z_p}^2 + 3)r + (c_{z_p}^2 + 1) + \sqrt{(c_{z_p}^2 + 2c_{z_p}^2 + 1)r^2 + 2(-c_{z_p}^2 + 2c_{z_p}^2 + 3)r + (c_{z_p}^2 + 1)^2}} \cdot \frac{1}{\bar{x}(P)}$$

Corollary 2.7: When power is maximized with respect to  $\lambda$ ,

$$\rho^* = \frac{4r}{(-c_{z_p}^2 + 3)r + (c_{z_p}^2 + 1) + \sqrt{(c_{z_p}^2 + 2c_{z_p}^2 + 1)r^2 + 2(-c_{z_p}^2 + 2c_{z_p}^2 + 3)r + (c_{z_p}^2 + 1)^2}}$$

and

$$\bar{N}^* = \frac{2r \left[ (1 + c_{z_p}^2)r + b(r) + (1 + c_{z_p}^2) \right]}{(c_{z_p}^4 - 1)r^2 + 2(2 - c_{z_p}^2)(1 + c_{z_p}^2)r + \left[ (1 - c_{z_p}^2)r + (1 + c_{z_p}^2) \right] b(r) + (c_{z_p}^2 + 1)^2}$$

where

$$b(r) = \sqrt{(c_{z_p}^2 + 2c_{z_p}^2 + 1)r^2 + 2(-c_{z_p}^2 + 2c_{z_p}^2 + 3)r + (1 + c_{z_p}^2)^2}$$

If  $r \gg 1$ , we have

$$\lim_{r \rightarrow \infty} \rho^* = \frac{r}{r+1}$$

and

$$\lim_{r \rightarrow \infty} \bar{N}^* = \frac{(1 + c_{sp}^2)r}{2}$$

### 2.4.2.3 Finding the optimal number of Processors

In this section, we will first study the relationship between  $\Pi^{(1)}(P)$  and  $\Pi_2^{(1)}(\lambda, P)$ . From the result shown below, we show that there are many cases that the  $P^*$  for the system with no arrivals and the  $P^*$  for systems with arrivals are the same.

$$\begin{aligned} u_2(\lambda, P) &= (\text{processor utilization}) \\ &= (\text{processor utilization} \mid \text{system is busy}) \cdot P [\text{system is busy}] \\ &\quad + (\text{processor utilization} \mid \text{system is idle}) \cdot P [\text{system is idle}] \\ &= (\text{processor utilization} \mid \text{system is busy}) \cdot P [\text{system is busy}] \\ &= u_1(P) \cdot \rho \end{aligned}$$

Substituting  $u_2(\lambda, P) = \rho u_1(P)$  into the definition of power, we can show that

$$\Pi_2^{(1)}(\lambda, P) = \frac{u_2(\lambda, P)}{T(\lambda, P)} = \frac{\rho u_1(P)}{T(\lambda, P)} = \frac{\rho}{T(\lambda, P) / \bar{x}(P)} \cdot \frac{u_1(P)}{\bar{x}(P)}$$

Since  $\frac{u_1(P)}{\bar{x}(P)} = \Pi^{(1)}(P)$  and  $\frac{\rho}{T(\lambda, P) / \bar{x}(P)} = \frac{2\rho(1-\rho)}{2-\rho+\rho c_{sp}^2}$ , we finally have

$$\Pi_2^{(1)}(\lambda, P) = \frac{2\rho(1-\rho)}{2-\rho+\rho c_{sp}^2} \cdot \Pi^{(1)}(P)$$

In order to find  $P^*$  when the system is operating under the optimal operating point ( $\lambda^*$ ), we have

$$\Pi_2^{(1)}(\lambda^*, P) = \frac{2\rho^*(1-\rho^*)}{2-\rho^*+\rho^* c_{sp}^2} \cdot \Pi^{(1)}(P)$$

Note that  $\frac{2\rho^*(1-\rho^*)}{2-\rho^*+\rho^* c_{sp}^2}$  is only a function of  $c_{sp}$  (since  $\rho^*$  is a function of  $c_{sp}$  only as shown in Corollary 2.7) and is not a function of  $P$ ; therefore, for cases where  $c_{sp}$  is not a function of  $P$ , the  $P^*$  for  $\Pi_2^{(1)}(\lambda^*, P)$  is the same as the  $P^*$  for  $\Pi^{(1)}(P)$ . Therefore, all the results obtained in sections 2.3.2 and 2.3.3 for Models II and III in finding  $P^*$  can be used here. However, not every model has this characteristic. For example, Model I does not have this characteristic.

**Corollary 2.8:** For the continuous model, power, as defined in (1.7), is maximized when

$$P^* = \frac{1(P^*)}{a(P^*)}$$

and

$$\lambda^* = \frac{1}{a(P^*)(2 + \sqrt{2 + c_k})}$$

[Proof] This proof can easily be derived from Theorems 2.4 and 2.9.  
Q.E.D.

**Corollary 2.9:** For the two-stage discrete model, power, as defined in (1.7), is maximized when

$$\lambda^* = \frac{1}{fW(2 + \sqrt{2 + 2c_k})}$$

and

$$P^* = \begin{cases} 1 & \text{if } \frac{1-f}{f} \leq 1 \\ \frac{1-f}{f} & \text{if } 1 < \frac{1-f}{f} \end{cases}$$

[Proof] This proof can easily be derived from Theorems 2.2 and 2.9.  
Q.E.D.

## 2.5 Conclusions

By maximizing power as defined in this chapter, we found the optimal system utilization ( $\rho^*$ ) and the optimal mean number of jobs in the system ( $\bar{N}^*$ ) to be the same as derived in [KLEI79]; therefore, all the pleasing rules of thumb discovered in [KLEI79] also apply here. Moreover, we found the optimal number of processors ( $P^* = \frac{\beta}{\alpha}$ ) to be used in the system is the same for cases when queueing is allowed and when queueing is not allowed. This result serves as an important design rule when implementing a parallel processing system which admits only one job into service at a time. The expressions of the processing time speedup and the response time speedup are also given so that we learn how much gain (speedup) can be achieved for a specified job by using  $P$  processors.

## CHAPTER 3

### PARALLEL PROCESSING SYSTEMS WITH VARYING REQUIRED PROCESSORS WITH MULTIPLE JOBS IN SERVICE

In chapter two there are times when the job in service does not require all the processors while there are jobs waiting in the queue. Therefore, this system does not work to its full power since the computing capability of those unused processors are simply wasted and cannot be used by those jobs waiting in the queue (only one job in service at a time). In this chapter, we avoid such an inefficiency by allowing jobs in the queue to use those unused processors. However, a higher priority is given to jobs which are closer to completion and we allow a higher priority job to preempt a lower priority job if the higher priority job needs more processors than it currently possesses. That is, we preempt on demand in a FCFS fashion. Because of this priority assignment and because of preemption, the number of processors possessed by a job during its execution time varies. This characteristic makes the exact analysis for the general case very difficult because we have to keep track of how many jobs there are in every stage. The main goal of this chapter is to find the mean response time of the system.

In this chapter, we are able to achieve the exact solution for jobs with two stages when the number of processors in the system equals the maximum number of processors required by the job under the assumption that the service time distribution of each stage is exponential. We then propose a *Scale-up Rule* with which we obtain a very good approximation result when the number of processors in the system is larger than the maximum number of processors required. Combining our two-stage results and the *Scale-up Rule*, we give an approximation for the general case analysis. From these studies, we propose a rule of thumb for designing parallel algorithms.

The rest of this chapter is organized as follows. Section one describes some previous work which looked at a problem with a similar environment but with a different service discipline. Section two gives the model description and assumptions for this chapter. In section three we study the serial-parallel behavior of a job which has a serial stage, which requires no processing capacity at all, and a parallel stage, which requires all the processing capacity, during its execution. In section four we analyze, exactly, the mean response time of a two-stage task graph (will be defined later) given that the number of processors in the system equals the maximum number of processors required by the jobs. In section five we introduce the *Scale-up Rule* with some examples. In section six we use the *Scale-up Rule* and the exact result obtained in section four to approximate the mean response time for any general processor-time task graph given any number of processors. A final conclusion is given in section seven.

### 3.1 Previous Work

In [BELG86], some results for the mean response time in a parallel processing system were derived. In [KUNG84], a task graph model was used to describe a job and was later converted into an execution graph which identified the execution stages assumed by any job throughout the job's execution time in the system. The scheduling policy (i.e., a service discipline) was based on a non-egalitarian sharing of the processors' capacity among the jobs present in the system. The scheduling policy can be described as follows:

Let  $\hat{n}$  denote the random variable representing the total number of ready tasks, which can be processed immediately, from all the jobs present in the system in the steady state. The *Discriminatory Processor Sharing Discipline* for the multiprocessing system is defined as follows:

1. If the total number of ready tasks,  $\hat{n}$ , in the system is less than or equal to the number of processors  $P$ , then each ready task is allocated one processor, that is each ready task is processed at a rate of 1 second per second.
2. If the total number of ready tasks,  $\hat{n}$ , in the system is greater than or equal to the number of processors  $P$ , then each ready task is served at a rate of  $\frac{P}{\hat{n}}$  seconds per second. The ready tasks equally share the  $P$  processors.

An Execution Graph for a given process graph is an acyclic directed graph with nodes representing the state of execution of a job (i.e., the identity of the ready tasks of the job), and edges representing the precedence relationships among the nodes. A simple approximation for the mean response time was given by assuming that all stages have the same concurrency degree i.e., all stages require the same number of processors. However, by this assumption we lose the characteristics of a job which requires varying number of processors.

### 3.2 Model Description and Assumptions

We use Model I as described in chapter two as our model in this chapter. However, instead of allowing the time required by each stage of a job to have a general distribution, we assume the time required by each stage to be exponentially distributed. Therefore, a job can be fully described by two vectors, the processor vector and the time vector. We assume the arrivals are generated from a Poisson process with rate  $\lambda$ . Note that we cannot rearrange the processor vector without affecting the mean response time as we did in chapter two. Later in this chapter we discover how the sequence of the processor vector can affect the mean response time.

As described in chapter two, whenever a job in a stage requires more processors than it is allowed, this job simply uses all the processors available to it with a prolonged stage service time. For example, if a job in a stage needs 10 processors for 1 second while there are only 5 processors available to it, it uses these 5 processors for 2 seconds.

The service discipline applied in this chapter is a version of priority queueing with preemption. However, the priority is not based upon the different classes of jobs entering the system; instead, the priority is based upon the stage that the job is currently working on. The purpose of this priority assignment is to give a higher priority to a job which is closer to completion. Hence, a higher priority is given to a job which is working on a later stage because this job is closer to completion. Hence, when a job finishes one stage and advances to the next stage, the priority ranking of this job will advance by one. Therefore, if by advancing to next stage this job requires more processors than it currently possesses, this job will try to gain more processors by preempting other jobs in the system beginning from the lowest priority group until either it has gained enough processors or there are no more lower priority jobs in service. Those jobs that have been preempted will be pushed back to the head of the queue and follow the FCFS (First Come First Serve) discipline waiting to get into service again. Example 3.1 gives an illustration of how the system works.

**Example 3.1:**

In this example we assume each job has two stages, the first stage requires 2 processors and the second stage requires 5 processors. The system has a total of 5 processors. Initially the system is empty.

- Event 1:** Job A arrives.  
Job A uses two processors to work on the first stage at a rate of 2 seconds per second.  
The other three processors remain unused.
- Event 2:** Job B arrives while Job A is still in stage 1.  
Job B uses two processors from the three unused processors to work on the first stage at a rate of 2 seconds per second. Only one processor remains unused.
- Event 3:** Job C arrives while Jobs A and B are still in stage 1.  
Job C uses the only unused processor to work on the first stage at a rate of 1 second per second. There are no more processors unused.
- Event 4:** Job D arrives while Jobs A, B, and C are still in stage 1.  
Job D joins the queue waiting for any processor to become available.
- Event 5:** Job A finishes stage one and begins stage two.  
Job A needs three more processors to work on the second stage; therefore, jobs B and C are both preempted and are pushed back to the head of the queue such that job A can use all the processors. After gaining all the processors, job A works at a rate of 5 seconds per second.
- Event 6:** Job A finishes stage two and leaves the system.  
All the five processors are released. Jobs B and C reenter service and both use two processors to work on the first stage at a rate of 2 seconds per second. Job D also enters service and uses the remaining processor to work on the first stage at a rate of 1 second per second.

The reason we claim the exact analysis is very difficult can be illustrated by using one special case. In this special case, the number of processors required by all stages equals "1" and the mean service time for all stages are the same. Obviously this very simple special case corresponds to an  $M/E_1/m$  queue, where  $m$  is the number of processors in the system; this is a queuing system which has no known closed form solution.

### 3.3 A Serial - Parallel Model

In this section, we assume the execution of a job can be classified into two stages, namely a *serial* stage and a *parallel* stage. In the serial stage, the job requires a negligible amount of total processors while in the parallel stage the job requires all the processors from the system. One example of this model is for systems with a large number of processors from which the serial stage requires only one processor while the parallel stage requires all the processors. Therefore, if the parallel stage has a higher priority, then there can be at most one job working in the parallel stage. Once there is a job in the parallel stage, all other jobs will be stopped because there are no available processors left. On the other hand, if the serial stage has a higher priority, then all the jobs in the serial stage will have no effect on all jobs because the capacity taken by the jobs in the serial stage is negligible. We first study the case when the serial stage precedes the parallel stage (Figure 3.1(a)), then we study the case when the parallel stage precedes the serial stage (Figure 3.1(b)). Finally we study the case when the parallel stage is preceded and followed by a serial stage (Figure 3.1(c)). We derive both the mean response time and the z-transform of the distribution of the number of jobs in the system.

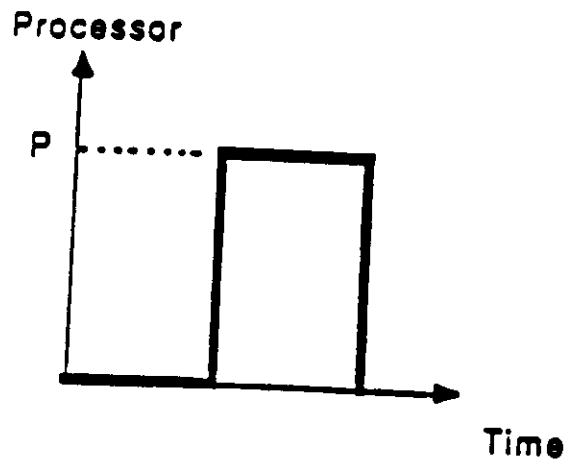


Figure 3.1(a) Serial Phase Precedes Parallel Phase

#### 3.3.1 The Serial Stage Precedes the Parallel Stage

In this section, stage one is the serial stage and stage two is the parallel stage and the higher priority is given to stage two as mentioned earlier. If there are no jobs in stage two, then all the jobs in stage one are allowed to work concurrently since there are always enough processors for every job. However, if one of the jobs finishes stage one and advances to stage two, this job will occupy all the processors and preempt all other jobs still in stage one.



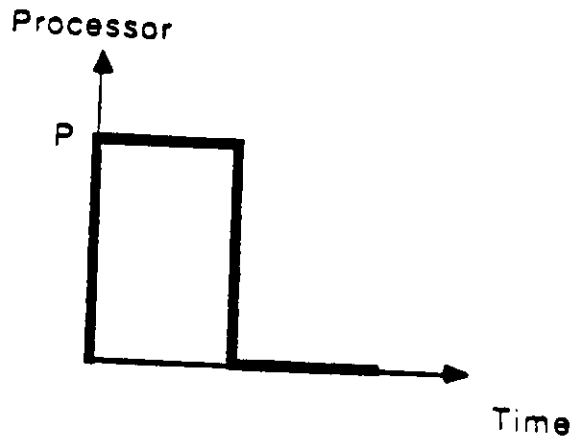


Figure 3.1(b) Parallel Phase Precedes Serial Phase

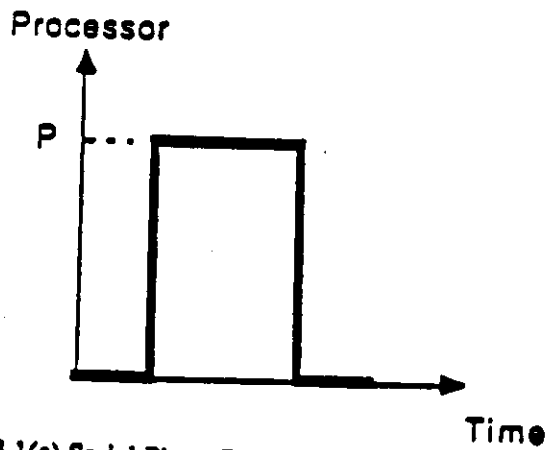


Figure 3.1(c) Serial Phase Precedes and Follows Parallel Phase

One application of this model is for a distributed database which uses a locking mechanism to preserve data integrity whenever more than one transaction tries to write into the database. We assume there are a large number of workstations sharing this database. We model a transaction as one query (read) followed by one update (write). If concurrent read is allowed in the database, then we allow all the queries to be processed in the system concurrently. Furthermore, when a transaction is in the query phase, it occupies only the processor which issues the transaction. This read phase corresponds to the serial stage in our model since they bear the same characteristics. However, once a transaction finishes the read phase and issues a write update, the entire database will be locked such that all other transactions will be stopped. This update phase corresponds to our parallel stage since they possess the same characteristics. If we further assume the time required for the read phase and the write phase are both exponentially distributed, then our model matches this distributed database model perfectly.

### 3.3.1.1 Mean Response Time Analysis

Let us define the service rate of the serial stage to be  $\mu_1$  and the service rate of the parallel stage to be  $\mu_2$ . We draw a Markov chain for such a system where the state  $(n_1, n_2)$  represents the situation where there are  $n_1$  jobs in stage one (the serial stage) and  $n_2$  jobs in stage two (the parallel stage).

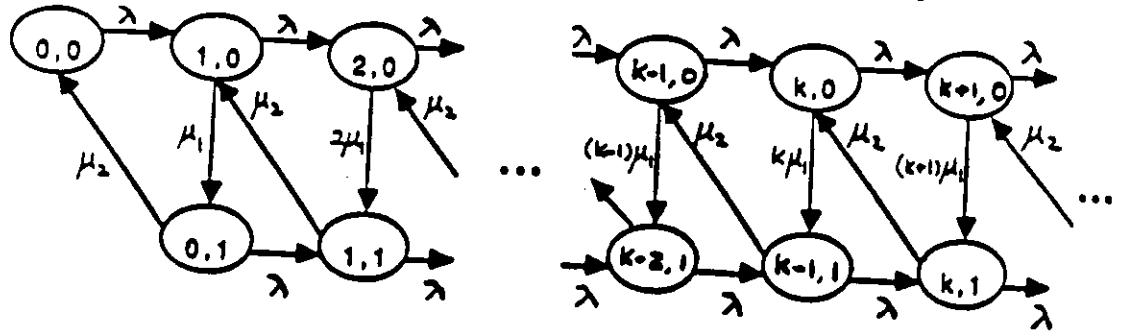


Figure 3.2  
The Markov Chain for Serial Stage - Parallel Stage

From this Markov chain, we have the following equilibrium balance equations:

$$(\lambda + k\mu_1)p(k, 0) = \lambda p(k-1, 0) + \mu_2 p(k, 1) \quad k \geq 1 \quad (3.1)$$

$$(\lambda + \mu_2)p(k-1, 1) = k\mu_1 p(k, 0) + \lambda p(k-2, 1) \quad k \geq 2 \quad (3.2)$$

The boundary conditions are

$$\lambda p(0, 0) = \mu_2 p(0, 1) \quad (3.3)$$

$$(\lambda + \mu_2)p(0, 1) = \mu_1 p(1, 0) \quad (3.4)$$

Define the following two z-transforms:

$$P_0(z) \triangleq \sum_{k=0}^{\infty} p(k, 0)z^k$$

$$P_1(z) \triangleq \sum_{k=0}^{\infty} p(k, 1)z^k$$

Define  $p(n)$  to be the probability that there are a total of  $n$  jobs in the system, each job is either in stage one or in stage two. Define the z-transform of  $p(n)$  to be

$$P(z) \triangleq \sum_{k=0}^{\infty} p(k)z^k$$

From the definition we have

$$p(k) = p(k, 0) + p(k-1, 1) \quad k \geq 1 \quad (3.5)$$

$$p(0) = p(0, 0) \quad (3.6)$$

From (3.5) and (3.6) we have

$$P(z) = P_0(z) + zP_1(z) \quad (3.7)$$

From (3.1) and (3.3) we have

$$\lambda P_0(z) + \mu_1 z \frac{d}{dz} P_0(z) = \lambda z P_0(z) + \mu_2 P_1(z) \quad (3.8)$$

From (3.2) and (3.4) we have

$$(\lambda + \mu_2)P_1(z) = \mu_1 \frac{d}{dz} P_0(z) + \lambda z P_1(z) \quad (3.9)$$

From (3.8) and (3.9) we have the following differential equation for  $P_0(z)$ .

$$(\lambda z - \mu_2)\mu_1 \frac{d}{dz} P_0(z) + \lambda(\lambda + \mu_2 - \lambda z)P_0(z) = 0$$

Solving this linear differential equation we obtain the following explicit expression of  $P_0(z)$ :

$$P_0(z) = c \cdot e^{\frac{\lambda}{\mu_1}(z - \ln(1 - \lambda z - \mu_2))} \quad (3.10)$$

where  $c$  is a constant yet to be decided. From (3.8) and (3.9) we have

$$P_1(z) = \frac{\lambda}{\mu_2 - \lambda z} P_0(z) \quad (3.11)$$

Combining (3.7), (3.10), (3.11), and  $P(1) = 1$  we find  $c$  to be

$$c = \frac{\mu_2 - \lambda}{\mu_2} e^{-\frac{\lambda}{\mu_1} [1 - \ln(\mu_2 - \lambda)]}$$

From the above results we have the z-transform of the number of jobs in the system as:

$$P(z) = \frac{\mu_2 - \lambda}{\mu_2 - \lambda z} e^{-\frac{\lambda}{\mu_1} [1 - \ln(\mu_2 - \lambda)]} e^{\frac{\lambda}{\mu_1}(z - \ln(1 - \lambda z - \mu_2))}$$

We define  $\bar{N}$  to be the average number of jobs in the system; hence,

$$\bar{N} = \left. \frac{dP(z)}{dz} \right|_{z=1} = \frac{\lambda(\mu_1 + \mu_2)}{\mu_1(\mu_2 - \lambda)}$$

Let us define the mean response time of this model to be  $T_{sp}$  where "sp" stands for serial-parallel. Using Little's result[LITT61], we finally arrive at

$$T_{sp} = \frac{\bar{N}}{\lambda} = \frac{1/\mu_1 + 1/\mu_2}{1 - \lambda/\mu_2} \quad (3.12)$$

Interestingly, note that  $\frac{\lambda}{\mu_2}$  in (3.12) is the load of the system since the serial stage contributes no load at all. Also note that  $1/\mu_1 + 1/\mu_2$  is the service time of a job. Hence, if we define

$$\bar{x} = \frac{1}{\mu_1} + \frac{1}{\mu_2} \quad (3.13)$$

and

$$\rho = \frac{\lambda}{\mu_2}$$

then (3.12) becomes

$$T_{sp} = \frac{\bar{x}}{1-\rho} \quad (3.14)$$

which is the expression of the mean response time of an M/M/1 queue. From these results, we have the following theorem.

### THEOREM 3.1:

Given a serial - parallel model as shown in Figure 3.1(a), where  $\mu_1$  is the service rate of the serial stage and  $\mu_2$  is the service rate of the parallel stage, the z-transform of the number of jobs in the system is:

$$P(z) = \frac{\mu_2 - \lambda}{\mu_2 - \lambda z} e^{-\frac{\lambda}{\mu_1} [1 - \ln(\mu_2 - \lambda)]} e^{\frac{\lambda}{\mu_1} (z - \ln(\lambda z - \mu_2))}$$

and the mean response time of the system is:

$$T_{sp} = \frac{\bar{N}}{\lambda} = \frac{1/\mu_1 + 1/\mu_2}{1 - \lambda/\mu_2}$$

#### 3.3.1.2 Optimization Issue with Power

We define *power* as defined in chapter one. We will find the optimal operating point ( $\lambda^*$ ) given  $P$  processors in the system, as well as the optimal number of processors to use in the system given the arrival rate, in order to maximize power. We define the workload of the parallel stage of the jobs to be  $\hat{W}$ , which is an exponentially distributed random variable with a mean  $W$ . Therefore

$$\mu_2 = \frac{P}{W}$$

which leads to

$$T_{sp} = \frac{1/\mu_1 + 1/\mu_2}{1 - \lambda/\mu_2} = \frac{P + W\mu_1}{\mu_1 P - \lambda W\mu_1} \quad (3.15)$$

The utilization of the processors, denoted as  $u_2(\lambda, P)$ , can be found by dividing the average workload arriving to the system per second by  $P$ ; hence

$$u_2(\lambda, P) = \frac{\lambda W}{P} \quad (3.16)$$

Defining power as in (1.7), we have

$$\Pi_2^{(1)}(\lambda, P) = \frac{u_2(\lambda, P)}{T_{sp}} = \frac{\lambda W \mu_1 P - \lambda^2 W^2 \mu_1}{P^2 + PW \mu_1} \quad (3.17)$$

Optimizing  $\Pi_2^{(1)}(\lambda, P)$  with respect to  $\lambda$ , we have

$$\lambda^* = \frac{P}{2W} \quad (3.18)$$

From (3.18) we have

$$u_2(\lambda^*, P) = \frac{1}{2}$$

and

$$\bar{N}^* = \lambda^* T^* = 1 + \frac{P}{W \mu_1} > 1 \quad (3.19)$$

It is not surprising to observe that  $\bar{N}^* > 1$  since this  $\bar{N}^*$  includes jobs both in the serial stage and the parallel stage. Because jobs in the serial stage occupy no capacity and we know from (1.4) that  $\bar{N}^* = 1$  for the classical M/G/1 queueing system where each job contributes a load to the system, therefore, the  $\bar{N}^*$  should be greater than unity in our model to generate enough load to maximize power.

Optimizing  $\Pi_2^{(1)}(\lambda, P)$  with respect to  $P$ , we have

$$\lambda W \mu_1 (P^2 + PW \mu_1) - (\lambda W \mu_1 P - \lambda^2 W^2 \mu_1)(2P + W \mu_1) = 0$$

Solving this equation we have

$$P^* = \left[ 1 + \sqrt{1 + \frac{\mu_1}{\lambda}} \right] \lambda W \quad (3.20)$$

If we use the more general definition of power as given in (1.8), we have

$$\Pi_2^{(1)}(\lambda, P) = \frac{u_2(\lambda, P)^r}{T_{sp}} = \frac{\lambda^r W^r \mu_1 P - \lambda^{r+1} W^{r+1} \mu_1}{P^{r+1} + W \mu_1 P^r} \quad (3.21)$$

Optimizing  $\Pi_2^{(1)}(\lambda, P)$  with respect to  $\lambda$ , we have

$$\lambda^* = \frac{rP}{(r+1)W}$$

Substituting  $\lambda^*$  into (3.21) we have

$$u_2(\lambda, P)^* = \frac{r}{r+1}$$

hence

$$\bar{N}^* = r + \frac{r\rho}{\mu_1 W} > r$$

Note that  $\bar{N}^* > r$ .

Optimizing  $\Pi_2^{(1)}(\lambda, P)$  with respect to  $P$ , we have

$$P^* = \frac{\lambda(r+1) - \mu_1(r-1) + \sqrt{\lambda^2(r+1)^2 + \mu_1^2(r-1)^2 + 2\lambda\mu_1 r^2 + 2\lambda\mu_1}}{2r} \cdot W$$

### 3.3.2 The Parallel Stage Precedes the Serial Stage

Here we change the sequence of the stages such that stage one is the parallel stage and stage two is the serial stage. We assume the service rate for the serial stage is  $\mu_1$  and the service rate for the parallel stage is  $\mu_2$ , as defined in the previous section.

#### 3.3.2.1 Mean Response Time Analysis

The analysis of this model is simple since the serial stage (with the higher priority) will never affect the parallel stage because the serial stage occupies no capacity at all. This model can be treated as a classic M/M/1 queueing system with service rate  $\mu_2$  with the modification that each job will suffer an extra delay of mean  $\frac{1}{\mu_1}$  before leaving the system. Let us define the mean response time of this model to be  $T_{ps}$  where "ps" stands for "parallel-serial". From the result for M/M/1 we obtain

$$T_{ps} = \frac{1/\mu_2}{1 - \lambda/\mu_2} + \frac{1}{\mu_1} \quad (3.22)$$

Let us compare  $T_{sp}$  and  $T_{ps}$  from (3.14) and (3.22); it can be shown that  $T_{sp}$  is always greater than  $T_{ps}$  if  $\frac{\lambda}{\mu_2} < 1$ .

$$T_{sp} - T_{ps} = \frac{1}{\mu_1} \cdot \frac{\lambda/\mu_2}{1 - \lambda/\mu_2} = \frac{1}{\mu_1} \cdot \frac{\rho}{1 - \rho} > 0$$

$T_{sp}$  can be much greater than  $T_{ps}$  when  $\rho$  approaches one. If we regard the parallel stage as the blocking stage since it blocks all other jobs in the system, then this result suggests that a job with an early blocking stage performs better (i.e., has a lower mean response time) than a job with a late blocking stage given other conditions remain the same.

### 3.3.2.2 Optimization Issue with Power

We use the same definitions and notations as in section 3.3.1.2. We have

$$\mu_2 = \frac{P}{W}$$

$$T_{ps} = \frac{1/\mu_2}{1 - \frac{\lambda}{\mu_2}} + \frac{1}{\mu_1} = \frac{W\mu_1 + P - \lambda W}{\mu_1(P - \lambda W)}$$

The utilization stays the same and is

$$u_2(\lambda, P) = \frac{\lambda W}{P}$$

Defining power as in (1.7), we have

$$\Pi_2^{(1)}(\lambda, P) = \frac{u_2(\lambda, P)}{T_{ps}} = \frac{W\mu_1}{P} \cdot \frac{\lambda(P - \lambda W)}{W\mu_1 + P - \lambda W}$$

Optimizing  $\Pi_2^{(1)}(\lambda, P)$  with respect to  $\lambda$ , we have

$$\lambda^* = \frac{W\mu_1 + P - \sqrt{W\mu_1(W\mu_1 + P)}}{W} \quad (3.23)$$

$$u_2(\lambda, P)^* = 1 - \frac{\sqrt{W\mu_1(W\mu_1 + P)} - W\mu_1}{P}$$

hence

$$\bar{N}^* = \lambda^* T^* = 1 + P \cdot \frac{\left[1 + \frac{P}{W\mu_1}\right]^{1/2} - 1}{\sqrt{W\mu_1(W\mu_1 + P)} - W\mu_1} = 1 + \frac{P}{W\mu_1} > 1 \quad (3.24)$$

Note that the result in (3.24) is the same as the result in (3.19). Also note that the result in (3.23) is greater than the result in (3.18). This can be explained as following. Since  $T_{sp}$  is greater than  $T_{ps}$  and  $\bar{N}^*$  for both systems are the same; therefore, using Little's result [LITT61] we can show that  $\lambda^*$  in (3.23) is greater than  $\lambda^*$  in (3.18). This result shows that the optimal throughput of the parallel-serial model is greater than that of the serial-parallel model when power is maximized for both cases.

Optimizing  $\Pi_2^{(1)}(\lambda, P)$  with respect to  $P$ , we have

$$P^* = (1 + \sqrt{\frac{\mu_1}{\lambda}})\lambda W$$

If we use the more general definition of power as given in (1.8), we have

$$\Pi_2^{(1)}(\lambda, P) = \frac{u_2(\lambda, P)^r}{T_{ps}} = \frac{W^r \mu_1}{P^r} \cdot \frac{\lambda^r (P - \lambda W)}{W\mu_1 + P - \lambda W}$$

Optimizing  $\Pi_2^{(1)}(\lambda, P)$  with respect to  $\lambda$ , we have

$$\lambda^* = \frac{2rP + (r+1)W\mu_1 - \sqrt{W\mu_1[(r+1)^2W\mu_1 + 4rP]}}{2rW}$$

Optimizing  $\Pi_2^{(1)}(\lambda, P)$  with respect to  $P$ , we have

$$P^* = \frac{2r\lambda W + W\mu_1 - rW\mu_1 + \sqrt{(rW\mu_1 - 2r\lambda W - W\mu_1)^2 - 4r\lambda^2 W^2 + 4r\lambda W^2 \mu_1}}{2}$$

Let us make a comparison between the two cases discussed above. Figure 3.3 shows the mean response time versus load assuming  $\mu_1 = \mu_2 = 1$  for both cases. If we regard the behavior of the parallel stage as a blocking stage, then from the comparison we see that early blocking performs better (i.e., with smaller mean response time) than late blocking. Therefore, if we have the choice to decide when to place the blocking stage in designing a parallel algorithm, we would like to place it as early as possible (rule of thumb).

### 3.3.3 A Combined Model

We define " $f$ " to be the fraction of the service time the job spends in the serial stage over the overall service time, i.e.

$$f \triangleq \frac{1/\mu_1}{1/\mu_1 + 1/\mu_2}$$

Without loss of generality, we normalize the service time to be unity, i.e.

$$\frac{1}{\mu_1} + \frac{1}{\mu_2} = 1$$

Thus

$$\frac{1}{\mu_1} = f \quad \text{and} \quad \frac{1}{\mu_2} = 1 - f$$

We consider cases when part of the serial stage precedes the parallel stage and the remaining part of the serial stage follows the parallel stage as shown in Figure 3.1(c). We define " $g$ " to be the fraction of the serial stage which precedes the parallel stage (i.e., a percent  $g$  of the serial stage precedes the parallel stage and the remaining fraction  $(1-g)$  of the serial stage follows the parallel stage). We denote the mean response time of this system to be  $T_{fs}$ . By using the results of (3.12) and (3.22) we obtain

$$\begin{aligned} T_{fs} &= \frac{\frac{1}{\mu_2} + g \cdot \frac{1}{\mu_1}}{1 - \frac{\lambda}{\mu_2}} + (1-g) \cdot \frac{1}{\mu_1} \\ &= \frac{(1-f) + gf}{1 - \lambda(1-f)} + (1-g)f \end{aligned}$$

The utilization of the resource is



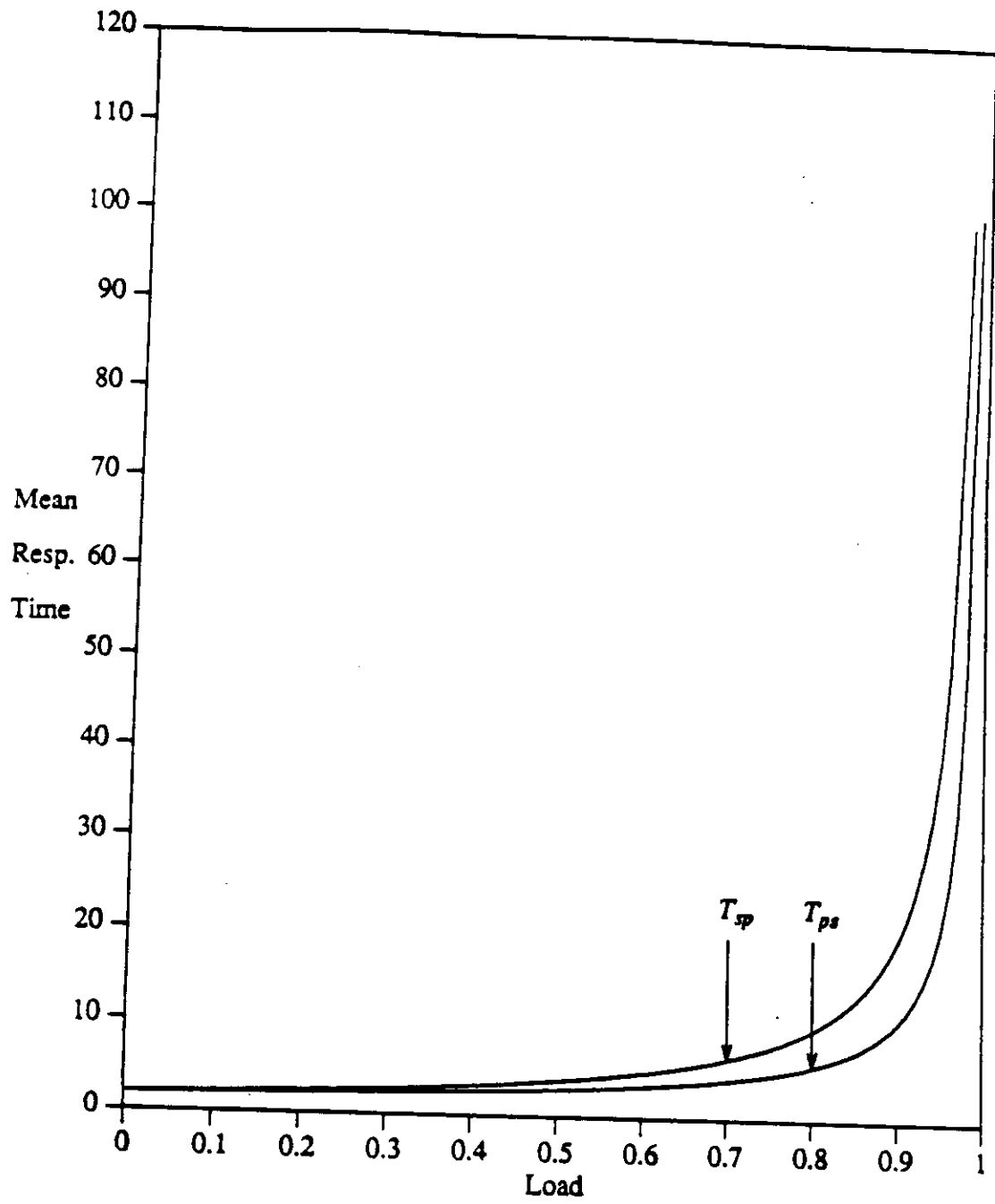


Figure 3.3  
Average System Time versus Load ( $\mu_1 = \mu_2 = 1$ )

$$\rho = \frac{\lambda}{\mu_2} = \lambda(1-f)$$

Hence, the condition for the system not to saturate is

$$\rho < 1$$

or

$$\lambda(1-f) < 1$$

Figure 3.4 shows the mean response time versus load over different values of  $g$  and  $f$ . From these results, we have the following theorem.

**THEOREM 3.2:**

For a serial - parallel model as shown in Figure 3.1(c) and with " $f$ " and " $g$ " as defined above, the mean response time is

$$T_{fs} = \frac{(1-f) + gf}{1 - \lambda(1-f)} + (1-g)f$$

under the condition that

$$\lambda(1-f) < 1$$

### 3.4 A Two-stage model

In this section, we consider a job with only two stages where in one stage the job uses  $P_1$  processors to process its work while in the other stage the job uses  $mP_1$  processors, where  $m$  is a real number greater than 1, for processing concurrently. We define the stage which uses  $P_1$  processors as the *low-concurrency* stage and the other stage which uses  $mP_1$  processors concurrently as the *high-concurrency* stage. We assume there are  $mP_1$  processors in the system, i.e., the number of processors in the system equals the number of processors required by the high-concurrency stage. In this section, two cases are analyzed: one is when the low-concurrency stage precedes the high-concurrency stage and the other is when the high-concurrency stage precedes the low-concurrency stage. For cases when the number of processors in the system is greater than  $mP_1$ , a good approximation will be given after the introduction of the *scale-up rule*, which will be described in section 3.5.

#### 3.4.1 The Low-Concurrency Stage Precedes the High-concurrency Stage

In this section, stage one is the low-concurrency stage while stage two is the high-concurrency stage. This job can be modeled as shown in Figure 3.5. This model is similar to the model described in section two except that in this model there can be at most  $m$  jobs working concurrently in the first stage. The Markov chain of this model is shown in Figure 3.6. From the Markov chain we have

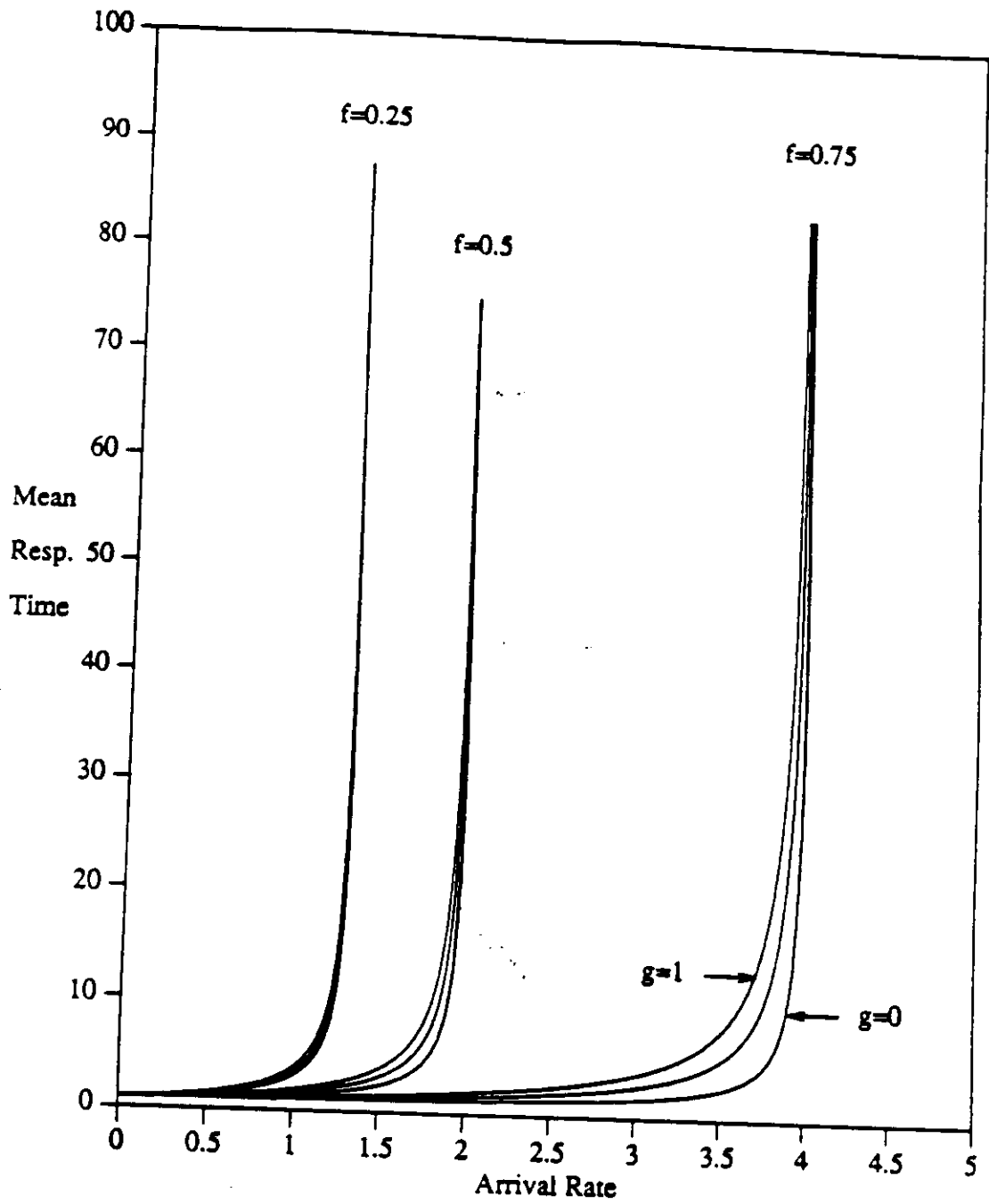


Figure 3.4  
 Average System Time versus Arrival Rate for Various  $g$  and  $f$   
 (For each set of curves,  $g=1, 0.5,$  and  $0$  respectively from left to right)

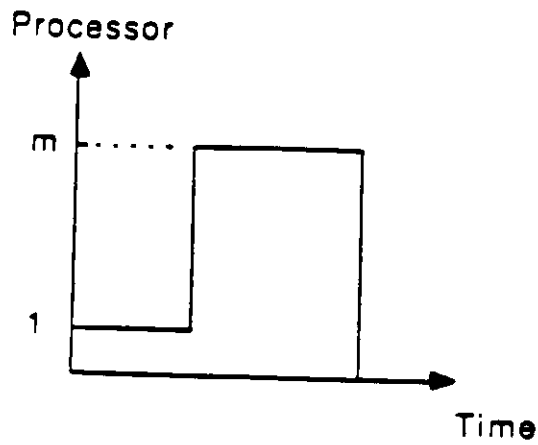


Figure 3.5  
A Sample Task Graph

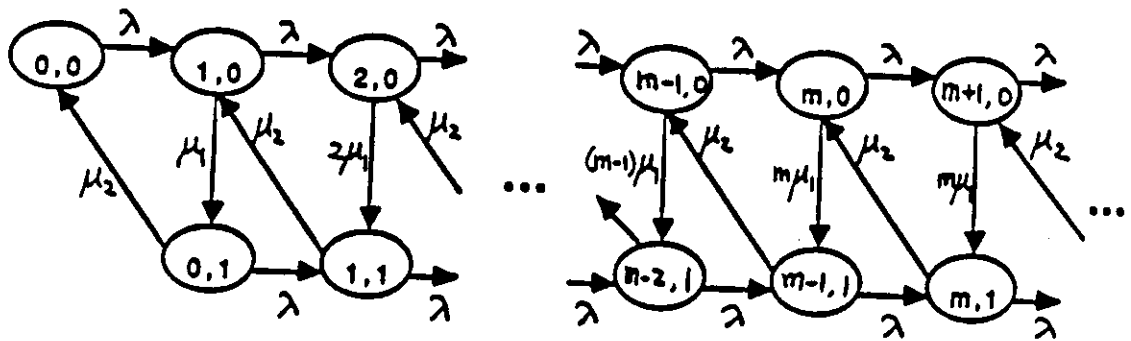


Figure 3.6  
The Markov Chain for Figure 3.5

$$(\lambda + k\mu_1)p(k, 0) = \lambda p(k-1, 0) + \mu_2 p(k, 1); \quad 1 \leq k \leq m \quad (3.25)$$

$$(\lambda + m\mu_1)p(k, 0) = \lambda p(k-1, 0) + \mu_2 p(k, 1); \quad m < k \quad (3.26)$$

$$(\lambda + \mu_2)p(k-1, 1) = \lambda p(k-2, 1) + k\mu_1 p(k, 0); \quad 2 \leq k \leq m \quad (3.27)$$

$$(\lambda + \mu_2)p(k-1, 1) = \lambda p(k-2, 1) + m\mu_1 p(k, 0); \quad m < k \quad (3.28)$$

The boundary conditions are

$$\lambda p(0, 0) = \mu_2 p(0, 1) \quad (3.29)$$

$$(\lambda + \mu_2)p(0,1) = \mu_1 p(1,0) \quad (3.30)$$

We define the following two z-transforms:

$$P_0(z) \triangleq \sum_{k=0}^{\infty} p(k,0)z^k$$

$$P_1(z) \triangleq \sum_{k=0}^{\infty} p(k,1)z^k$$

We also define

$$p(k) \triangleq \text{Prob} [ k \text{ jobs in the system} ]$$

$$P(z) \triangleq \sum_{k=0}^{\infty} p(k)z^k$$

From the above definitions we have

$$P(z) = P_0(z) + zP_1(z)$$

From (3.25) and (3.26) we have

$$\sum_{k=1}^{\infty} (\lambda + k\mu_1)p(k,0)z^k + \sum_{k=m+1}^{\infty} (\lambda + m\mu_1)p(k,0)z^k = \sum_{k=1}^{\infty} \lambda p(k-1,0)z^k + \sum_{k=1}^{\infty} \mu_2 p(k,1)z^k$$

After some algebra we have

$$(\lambda + m\mu_1 - \lambda z)P_0(z) = \mu_2 P_1(z) + \mu_1 \sum_{k=0}^{m-1} (m-k)p(k,0)z^k \quad (3.31)$$

Similarly, from (3.27) and (3.28) we have

$$\sum_{k=2}^{\infty} (\lambda + \mu_2)p(k-1,1)z^k = \sum_{k=2}^{\infty} \lambda p(k-2,1)z^k + \sum_{k=2}^{\infty} k\mu_1 p(k,0)z^k + \sum_{k=m+1}^{\infty} \mu_1 p(k,0)z^k$$

After some algebra we have

$$[(\lambda + \mu_2)z - \lambda z^2]P_1(z) = m\mu_1 P_0(z) - m\mu_1 - \mu_1 P_0(z) - \mu_1 \sum_{k=0}^{m-1} (m-k)p(k,0)z^k \quad (3.32)$$

From (3.31) and (3.32) we have

$$P_1(z) = \frac{\lambda}{\mu_2 - \lambda z} P_0(z) \quad (3.33)$$

Using (3.31), (3.32), and (3.33) we have

$$P_0(z) = \frac{(\mu_2 - \lambda z)\mu_1}{-\lambda^2 z + m\mu_1 \mu_2 - m\lambda \mu_1 z - \lambda \mu_2 z + \lambda^2 z^2} \sum_{k=0}^{m-1} (m-k)p(k,0)z^k$$

$$P_1(z) = \frac{\lambda \mu_1}{-\lambda^2 z + m\mu_1 \mu_2 - m\lambda \mu_1 z - \lambda \mu_2 z + \lambda^2 z^2} \sum_{k=0}^{m-1} (m-k)p(k,0)z^k$$

Thus

$$P(z) = \frac{\mu_1 \mu_2}{-\lambda^2 z + m\mu_1 \mu_2 - m\lambda \mu_1 z - \lambda \mu_2 z + \lambda^2 z^2} \sum_{k=0}^{m-1} (m-k)p(k,0)z^k \quad (3.34)$$

The remaining problem is to find all  $p(k,0)$  for  $0 \leq k \leq m-1$ . Using (3.34) and  $P(1) = P_0(1) + P_1(1) = 1$ , we have

$$\sum_{k=0}^m (m-k)p(k,0) = \frac{m\mu_1 \mu_2 - m\mu_1 \lambda - \mu_2 \lambda}{\mu_1 \mu_2} \quad (3.35)$$

Rearranging (3.25) and (3.27) we have

$$p(k,1) = \frac{\lambda + k\mu_1}{\mu_2} p(k,0) - \frac{\lambda}{\mu_2} p(k-1,0); \quad 1 \leq k \leq m \quad (3.36)$$

$$p(k,0) = \frac{\lambda + \mu_2}{k\mu_1} p(k-1,1) - \frac{\lambda}{k\mu_1} p(k-2,1); \quad 2 \leq k \leq m \quad (3.37)$$

From (3.29) and (3.30) we have

$$p(0,1) = \frac{\lambda}{\mu_2} p(0,0) \quad (3.38)$$

$$p(1,0) = \frac{\lambda(\lambda + \mu_2)}{\mu_1 \mu_2} p(0,0) \quad (3.39)$$

From (3.37), (3.38), and (3.39) we have

$$p(2,0) = \frac{\lambda^2[(\lambda + \mu_2)^2 + \lambda\mu_1]}{2\mu_1^2 \mu_2^2} p(0,0) \quad (3.40)$$

Apply (3.36) to (3.37) we have for  $3 \leq k \leq m$ ,

$$p(k,0) = \frac{(\lambda + \mu_2)[\lambda + (k-1)\mu_1]}{k\mu_1 \mu_2} p(k-1,0) - \frac{\lambda[\lambda + 2\mu_1(k-1)]}{k\mu_1 \mu_2} p(k-2,0) - \frac{\lambda^2}{k\mu_1 \mu_2} p(k-3,0) \quad (3.41)$$

From (3.39), (3.40), (3.41), and (3.35) we are able to find all  $p(k,0)$  for  $0 \leq k \leq m-1$ . Applying these results we obtain  $P(z)$  and consequently  $\bar{N}$  and  $T$ .

Differentiating (3.34) we have

$$\begin{aligned} \bar{N} &= \frac{d}{dz} P(z) \Big|_{z=1} \\ &= \frac{\mu_1 \mu_2}{m\mu_1 \mu_2 - m\lambda \mu_1 - \lambda \mu_2} \sum_{k=1}^{m-1} k(m-k)p(k,0) - \frac{\lambda^2 - m\lambda \mu_1 - \lambda \mu_2}{m\mu_1 \mu_2 - m\lambda \mu_1 - \lambda \mu_2} \end{aligned} \quad (3.42)$$

Hence,

$$T = \frac{\bar{N}}{\lambda} = \frac{\mu_1 \mu_2}{\lambda(m\mu_1 \mu_2 - m\lambda\mu_1 - \lambda\mu_2)} \sum_{k=1}^{m-1} k(m-k)\rho(k,0) - \frac{\lambda - m\mu_1 - \mu_2}{m\mu_1 \mu_2 - m\lambda\mu_1 - \lambda\mu_2} \quad (3.43)$$

From (3.43) we observe that the constraint for  $\lambda$  not to saturate the system is

$$\lambda < \frac{m\mu_1 \mu_2}{m\mu_1 + \mu_2}$$

The processor utilization ( $u$ ) can be found as

$$u = \frac{\lambda(\mu_2 + m\mu_1)}{m\mu_1 \mu_2}$$

Hence, the constraint of  $\lambda$  is equivalent to

$$u < 1$$

### THEOREM 3.3:

For a two-stage model as shown in Figure 3.6, the z-transform of the number of jobs in the system is

$$P(z) = \frac{\mu_1 \mu_2}{-\lambda^2 z + m\mu_1 \mu_2 - m\lambda\mu_1 z - \lambda\mu_2 z + \lambda^2 z^2} \sum_{k=0}^{m-1} (m-k)\rho(k,0)z^k$$

and the mean response time is

$$T = \frac{\mu_1 \mu_2}{\lambda(m\mu_1 \mu_2 - m\lambda\mu_1 - \lambda\mu_2)} \sum_{k=1}^{m-1} k(m-k)\rho(k,0) - \frac{\lambda - m\mu_1 - \mu_2}{m\mu_1 \mu_2 - m\lambda\mu_1 - \lambda\mu_2}$$

where  $\rho(k,0)$  for  $0 \leq k \leq m-1$  can be obtained from (3.35), (3.39), (3.40), and (3.41). The condition for the system not to saturate is

$$\lambda < \frac{m\mu_1 \mu_2}{m\mu_1 + \mu_2}$$

An example is now given for  $m=2$  which will later be used for comparison. From (3.35) we have

$$2\rho(0,0) + \rho(1,0) = \frac{2\mu_1 \mu_2 - 2\mu_1 \lambda - \mu_2 \lambda}{\mu_1 \mu_2} \quad (3.44)$$

From (3.30) we have

$$(\lambda + \mu_2)\rho(0,1) = \mu_1 \rho(1,0) \quad (3.45)$$

From (3.29) we have

$$\lambda \rho(0,0) = \mu_2 \rho(0,1) \quad (3.46)$$

Solving (3.44), (3.45), and (3.46) we have

$$p(0,0) = \frac{2\mu_1\mu_2 - 2\mu_1\lambda - \mu_2\lambda}{\lambda^2 + \mu_2\lambda + 2\mu_1\mu_2} \quad (3.47)$$

$$p(1,0) = \frac{(\lambda^2 + \mu_2\lambda)(2\mu_1\mu_2 - 2\mu_1\lambda - \mu_2\lambda)}{\mu_1\mu_2(\lambda^2 + \mu_2\lambda + 2\mu_1\mu_2)} \quad (3.48)$$

Substituting (3.47) and (3.48) into (3.42) we have

$$\bar{N} = \frac{\lambda(4\mu_1\mu_2^2 + 4\mu_1^2\mu_2 - \lambda^3)}{(\lambda^2 + \mu_2\lambda + 2\mu_1\mu_2)(2\mu_1\mu_2 - 2\mu_1\lambda - \mu_2\lambda)}$$

Using Little's result [LIT76] we have

$$T = \frac{\bar{N}}{\lambda} = \frac{4\mu_1\mu_2^2 + 4\mu_1^2\mu_2 - \lambda^3}{(\lambda^2 + \mu_2\lambda + 2\mu_1\mu_2)(2\mu_1\mu_2 - 2\mu_1\lambda - \mu_2\lambda)} \quad (3.49)$$

### 3.4.2 The High-concurrency Stage Precedes the Low-concurrency Stage

We study the case when the high-concurrency stage precedes the low-concurrency stage. The processor-time task graph can be modeled as shown in Figure 3.7. We can draw the Markov chain as shown in Figure 3.8.

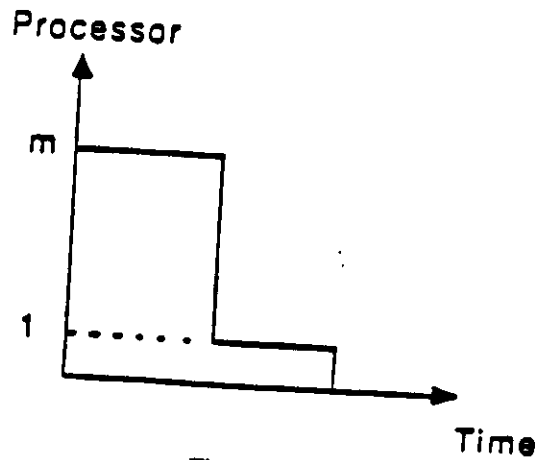


Figure 3.7  
Task Graph

From this Markov chain, we have

$$(\lambda + \mu_1)p(k, 0) = \lambda p(k-1, 0) + \mu_2 p(k, 1) \quad k \geq 1 \quad (3.50)$$

$$\left[ \lambda + j\mu_2 + \frac{m-j}{m}\mu_1 \right] p(k, j) = \lambda p(k-1, j) + \frac{m-j+1}{m}\mu_1 p(k+1, j-1) +$$



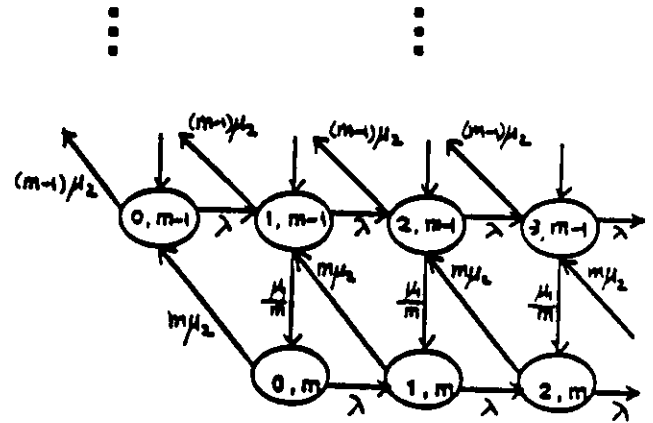
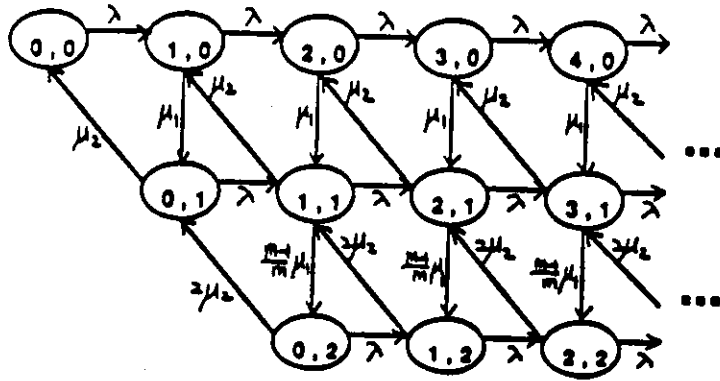


Figure 3.8  
Markov Chain for Figure 3.7

$$(j+1)\mu_2 p(k, j+1) \quad k \geq 1 \text{ \& } m > j > 0 \quad (3.51)$$

$$(\lambda + m\mu_2)p(k, m) = \lambda p(k-1, m) + \frac{1}{m}\mu_1 p(k+1, m-1) \quad k \geq 1 \quad (3.52)$$

The boundary conditions are

$$\lambda p(0, 0) = \mu_2 p(0, 1) \quad (3.53)$$

$$(\lambda + j\mu_2)p(0, j) = \frac{m-j+1}{m}\mu_1 p(1, j-1) + (j+1)\mu_2 p(0, j+1) \quad m-1 \geq j \geq 1 \quad (3.54)$$

$$(\lambda + m\mu_2)p(0, m) = \frac{1}{m}\mu_1 p(1, m-1) \quad (3.55)$$

Finding the z-transform of (3.50), (3.51), and (3.52) we have

$$P_1(z) = \frac{\lambda + \mu_1 - \lambda z}{\mu_2} P_0(z) - \frac{\mu_1}{\mu_2} p(0, 0) \quad (3.56)$$

$$P_j(z) = a(j, z) P_{j-1}(z) - b(j, z) P_{j-2}(z) - c(j, z) \quad m \geq j \geq 2 \quad (3.57)$$

$$(\lambda + m\mu_2 - \lambda z) P_m(z) = \frac{\mu_1}{mz} P_{m-1}(z) - \frac{\mu_1}{mz} p(0, m-1) \quad (3.58)$$

where

$$a(j, z) = \frac{1}{j\mu_2} \left[ \lambda + (j-1)\mu_2 + \frac{m-j+1}{m}\mu_1 - \lambda z \right]$$

$$b(j, z) = \frac{\mu_1(m-j+2)}{jm\mu_2 z}$$

$$c(j, z) = \frac{m-j+1}{jm\mu_2} \mu_1 p(0, j-1) - \frac{\mu_1(m-j+2)}{jm\mu_2 z} p(0, j-2)$$

Define  $p(k)$  and  $P(z)$  as in section 3.4.1. We have

$$p(n) = \sum_{j=0}^{n \wedge (m, n)} p(n-j, j)$$

$$P(z) = \sum_{n=0}^{\infty} \sum_{j=0}^n p(n-j, j) z^n + \sum_{n=m+1}^{\infty} \sum_{j=0}^m p(n-j, j) z^n = \sum_{j=0}^m z^j P_j(z)$$

From  $P(z)$  we can derive the mean number of jobs in the system by

$$\bar{N} = \frac{d}{dz} P(z) \Big|_{z=1} = \sum_{j=0}^m \left[ jz^{j-1} P_j(z) + z^j \frac{d}{dz} P_j(z) \right] \Big|_{z=1}$$

$$= \sum_{j=0}^m \left[ j P_j(z) + \frac{d}{dz} P_j(z) \right]_{z=1} \quad (3.59)$$

We are not able to solve this problem for an arbitrary value of  $m$  since there are too many unknowns. But, the result for  $m = 2$  can be found.

We now derive the result for the case when  $m = 2$ . From (3.53), (3.54), and (3.55) we have the following boundary conditions:

$$\lambda p(0,0) = \mu_2 p(0,1) \quad (3.60)$$

$$(\lambda + \mu_2) p(0,1) = \mu_1 p(1,0) + 2\mu_2 p(0,2)$$

$$(\lambda + 2\mu_2) p(0,2) = \frac{1}{2} \mu_1 p(1,1)$$

From (3.56), (3.57), and (3.58) we have the following z-transforms:

$$P_1(z) = \frac{\lambda(1-z) + \mu_1}{\mu_2} P_0(z) - \frac{\mu_1}{\mu_2} p(0,0) \quad (3.61)$$

$$P_2(z) = \frac{\lambda(1-z) + \mu_2 + \frac{\mu_2}{2}}{2\mu_2} P_1(z) - \frac{\mu_1}{2\mu_2 z} P_0(z) - \frac{\mu_1}{4\mu_2} p(0,1) + \frac{\mu_1}{2\mu_2 z} p(0,0) \quad (3.62)$$

$$P_2(z) = \frac{\mu_1}{2z(\lambda + 2\mu_2 - \lambda z)} P_1(z) - \frac{\mu_1}{2z(\lambda + 2\mu_2 - \lambda z)} p(0,1) \quad (3.63)$$

Solving (3.61), (3.62), and (3.63) with (3.60) we have

$$P_0(z) = \frac{-\mu_1 \lambda^2 z^2 + \lambda \mu_1 \left( \frac{3}{2} \lambda + 3\mu_2 + \frac{\mu_1}{2} \right) z - \mu_1 \mu_2 (\mu_1 + 2\lambda + 2\mu_2)}{\lambda^3 z^3 - \lambda^2 (2\lambda + 3\mu_2 + \frac{3}{2} \mu_1) z^2 + Az - \mu_1 \mu_2 (2\lambda + \mu_1 + 2\mu_2)} p(0,0) \quad (3.64)$$

where

$$A = \lambda(\lambda^2 + 3\mu_2 \lambda + \frac{3}{2} \mu_1 \lambda + 4\mu_1 \mu_2 + 2\mu_2^2 + \frac{\mu_1^2}{2})$$

From (3.64) we have

$$P_0(1) = \frac{\mu_1 \lambda^2 + 2\mu_1 \mu_2 \lambda + \mu_1^2 \lambda - 2\mu_1^2 \mu_2 - 4\mu_1 \mu_2^2}{4\mu_2^2 \lambda + \mu_1^2 \lambda + 4\mu_1 \mu_2 \lambda - 2\mu_1^2 \mu_2 - 4\mu_1 \mu_2^2} p(0,0) \quad (3.65)$$

From (3.61) we have

$$P_1(1) = \frac{\mu_1}{\mu_2} P_0(1) - \frac{\mu_1}{\mu_2} \rho(0,0) \quad (3.66)$$

From (3.63) we have

$$P_2(1) = \frac{\mu_1^2}{4\mu_2^2} P_0(1) - \frac{\mu_1^2 + \mu_1 \lambda}{4\mu_2^2} \rho(0,0) \quad (3.67)$$

From (3.65), (3.66), (3.67) and  $P_0(1) + P_1(1) + P_2(1) = 1$  we have

$$\rho(0,0) = \frac{2\mu_1\mu_2 - \mu_1\lambda - 2\mu_2\lambda}{\mu_1\lambda + 2\mu_1\mu_2}$$

From (3.64) we have

$$P'_0(1) = \frac{\beta'(1)\alpha(1) - \beta(1)\alpha'(1)}{[\alpha(1)]^2} \rho(0,0)$$

where

$$\alpha(1) = 4\mu_2^2\lambda + \mu_1^2\lambda + 4\mu_1\mu_2\lambda - 2\mu_1^2 - 4\mu_1\mu_2^2$$

$$\beta(1) = \mu_1\lambda^2 + 2\mu_1\mu_2\lambda + \mu_1^2\lambda - 2\mu_1^2\mu_2 - 4\mu_1\mu_2^2$$

$$\alpha'(1) = -6\mu_2\lambda^2 - 3\mu_1\lambda^2 + 8\mu_1\mu_2\lambda + 4\mu_2^2\lambda + \mu_1^2\lambda$$

$$\beta'(1) = -\mu_1\lambda^2 + 6\mu_1\mu_2\lambda + \mu_1^2\lambda$$

From (3.61) we have

$$P'_1(1) = -\frac{\lambda}{\mu_2} P_0(1) + \frac{\mu_1}{\mu_2} P'_0(1)$$

From (3.63) we have

$$P'_2(1) = \frac{2\mu_1\mu_2\lambda - 2\mu_1^2\mu_2 + \mu_1^2\lambda}{8\mu_2^2} P_0(1) + \frac{\mu_1^2}{4\mu_2^2} P'_0(1) + \frac{\mu_1(\mu_1 + \lambda)(2\mu_2 - \lambda)}{8\mu_2^2} \rho(0,0)$$

From (3.59) we have

$$\begin{aligned} \bar{N} &= P_1(1) + 2P_2(1) + P'_0(1) + P'_1(1) + P'_2(1) \\ &= \frac{\lambda}{\lambda + 2\mu_2} \frac{(3\mu_2^2 + \mu_1\mu_2 + 2\mu_1^2)\lambda^2 - \mu_2(\mu_1^2 - \mu_1\mu_2 - 4\mu_2^2)\lambda - 4\mu_2^2(\mu_1^2 + 3\mu_1\mu_2 + 2\mu_2^2)}{\mu_2(\mu_1 + 2\mu_2)(\mu_1\lambda + 2\mu_2\lambda - 2\mu_1\mu_2)} \end{aligned}$$

Using Little's result [LITT61] we finally have

$$\begin{aligned} T &= \frac{\bar{N}}{\lambda} \\ &= \frac{(3\mu_2^2 + \mu_1\mu_2 + 2\mu_1^2)\lambda^2 - \mu_2(\mu_1^2 - \mu_1\mu_2 - 4\mu_2^2)\lambda - 4\mu_2^2(\mu_1^2 + 3\mu_1\mu_2 + 2\mu_2^2)}{\mu_2(\lambda + 2\mu_2)(\mu_1 + 2\mu_2)(\mu_1\lambda + 2\mu_2\lambda - 2\mu_1\mu_2)} \quad (3.68) \end{aligned}$$

### 3.4.3 A Comparison

We make a comparison between the models of sections 3.4.1 and 3.4.2 for  $\mu_1 = \mu_2 = 1$ . These two examples have every condition the same except that the sequence of the stages is different. The result from (3.49) shows that

$$T_1 = \frac{1 + \lambda}{\lambda^2 + \lambda + 2} + \frac{3 - \lambda}{2 - 3\lambda}$$

where  $T_1$  is defined as the mean response time when the low-concurrency stage precedes the high-concurrency stage. The result from (3.68) shows that

$$T_2 = \frac{6\lambda^2 + 4\lambda - 24}{9\lambda^2 + 12\lambda - 12}$$

where  $T_2$  is defined as the mean response time when the high-concurrency stage precedes the low-concurrency stage. By finding  $T_1 - T_2$ , we have

$$T_1 - T_2 = \frac{\lambda(4 - \lambda - \lambda^2)}{3(\lambda + 2)(\lambda^2 + \lambda + 2)}$$

It can be shown easily that for  $\lambda < \frac{2}{3}$  (or,  $u < 1$ ),  $T_1 - T_2$  is always positive. If we regard the high-concurrency stage as the blocking stage (since it blocks other jobs in the system), then  $T_1 - T_2 > 0$  is consistent with the result we obtained in section 3.3.2. Hence, the rule of thumb in designing parallel algorithms is to put the blocking stage as early as possible (if allowed). The system time versus load of  $T_1 - T_2$  is shown in Figure 3.9. Notice that  $T_1$  and  $T_2$  are very close to each other, this is because there is no pole in the denominator in the expression of  $T_1 - T_2$ .

### 3.5 Scale-up Rule

The results derived in sections 3.3 and 3.4 are for cases when the number of processors in the system equals the maximum number of processors required by the job. This assumption simplified the Markov chain dramatically, hence making the analysis feasible. The analysis applied above will be infeasible when the number of processors is greater than the maximum number of processors required by the job. In order to solve this problem, we propose the *Scale-up Rule* which gives a very good approximation result without adding any analytical complexity. We will first show some examples for the classical queueing system. Then, an application of the scale-up rule for the two-stage model when the number of processors in the system is greater than the maximum number of processors required by the job is given.

#### 3.5.1 Previous Work: Classical Queueing Model

Although there has been considerable effort paid to the analysis of the M/G/m queue, almost no exact result has been achieved for the mean waiting time. Two exceptions are provided by M/M/m and M/D/m systems although in both cases the solutions are not in a closed form ([CROM34], [SYSK84]). Besides these two exceptions, only bounds, approximations, and numerical results have been found for other cases. [HILL71] gave some numerical results for the M/E<sub>k</sub>/m queue, [KING10], [BRUM71].

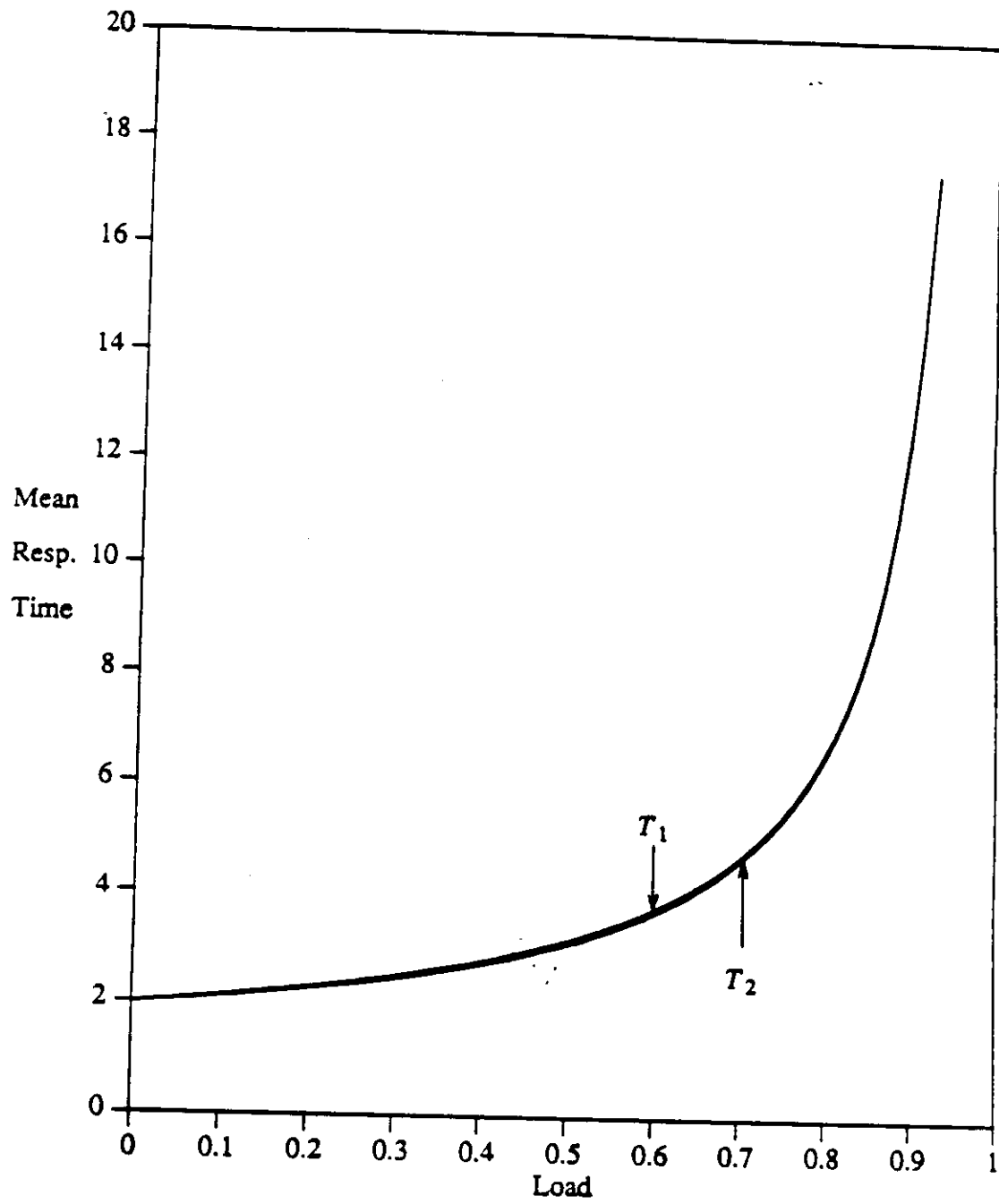


Figure 3.9  
Comparison Between  $T_1$  and  $T_2$

[STOY74], and [MORI75] provided some bounds, and [BOXM79], [YAKA77], and [HOOR82] derived some approximation results for such systems.

In this section, we focus our attention on the results provided by [MAAL73] and [NOZA75] who provided an approximation for the mean waiting time in an M/G/m system. [COSM76] and [BOXM79] later extended this model to achieve a better result. Let us define

$W_{M/G/m} \triangleq$  mean waiting time for an M/G/m system

[MAAL73] and [NOZA75] suggested the following approximation for  $W_{M/G/m}$ :

$$W_{M/G/m} \approx \frac{W_{M/G/1}}{W_{M/M/1}} \cdot W_{M/M/m}$$

Some simulation results are shown in Figures 3.10, 3.11, and 3.12 for three different queueing models to show the preciseness of this approximation. Figure 3.10 shows the simulation result along with the approximation result using the above equation for an M/D/3 queueing system with the mean service time  $\bar{x} = 2$ . Figure 3.11 shows the result for an M/E<sub>4</sub>/2 queueing system where the mean service time for each stage equals 1. Figure 3.12 shows the result for an M/H<sub>3</sub>/3 queueing system where the mean service time is 1, or 10, or 100 with equal probability,  $\frac{1}{3}$ .

### 3.5.2 Queueing Model with Varying Required Processors

Although the number of processors required by a job varies during its execution time, we were surprised to discover that the rule which applies to the classical queueing system as stated in the previous section also applies in our model. In other words, if we find the mean response time for a system with  $P$  processors, we can find a good approximation result when the number of processors is greater than  $P$  by using the scale-up rule.

**Scale-up Rule:** Given a processor-time task graph of a job and the number of processors in the system, say  $P$ , equals the maximum number of processors required by the job, we define the average waiting time of the system to be  $W_{M/TG/1}(\rho)$ , where  $\rho$  is the system load and TG stands for "Task Graph". Similarly, define  $W_{M/M/1}(\rho)$  to be the average waiting time of an M/M/1 queueing system with the same mean service time as the job. Define  $W_{M/TG/m}(\rho)$  to be the average waiting time if the number of processors is  $mP$ . The Scale-up Rule says that

$$W_{M/TG/m}(\rho) = \frac{W_{M/TG/1}(\rho)}{W_{M/M/1}(\rho)} \cdot W_{M/M/m}(\rho)$$

The scale-up rule can be useful in two situations. First, if we can analytically obtain  $W_{M/TG/1}$ , as we did in sections 3.3 and 3.4, then we can use the scale-up rule to approximate  $W_{M/TG/m}$ . Second, if we cannot obtain the analytical result, then we have to run the simulation. Since the time required to run the simulation for the M/TG/m system is longer than the time required to run the M/TG/1 system, we can run the simulation to find the mean waiting time for the M/TG/1 system and then apply the scale-up rule to find

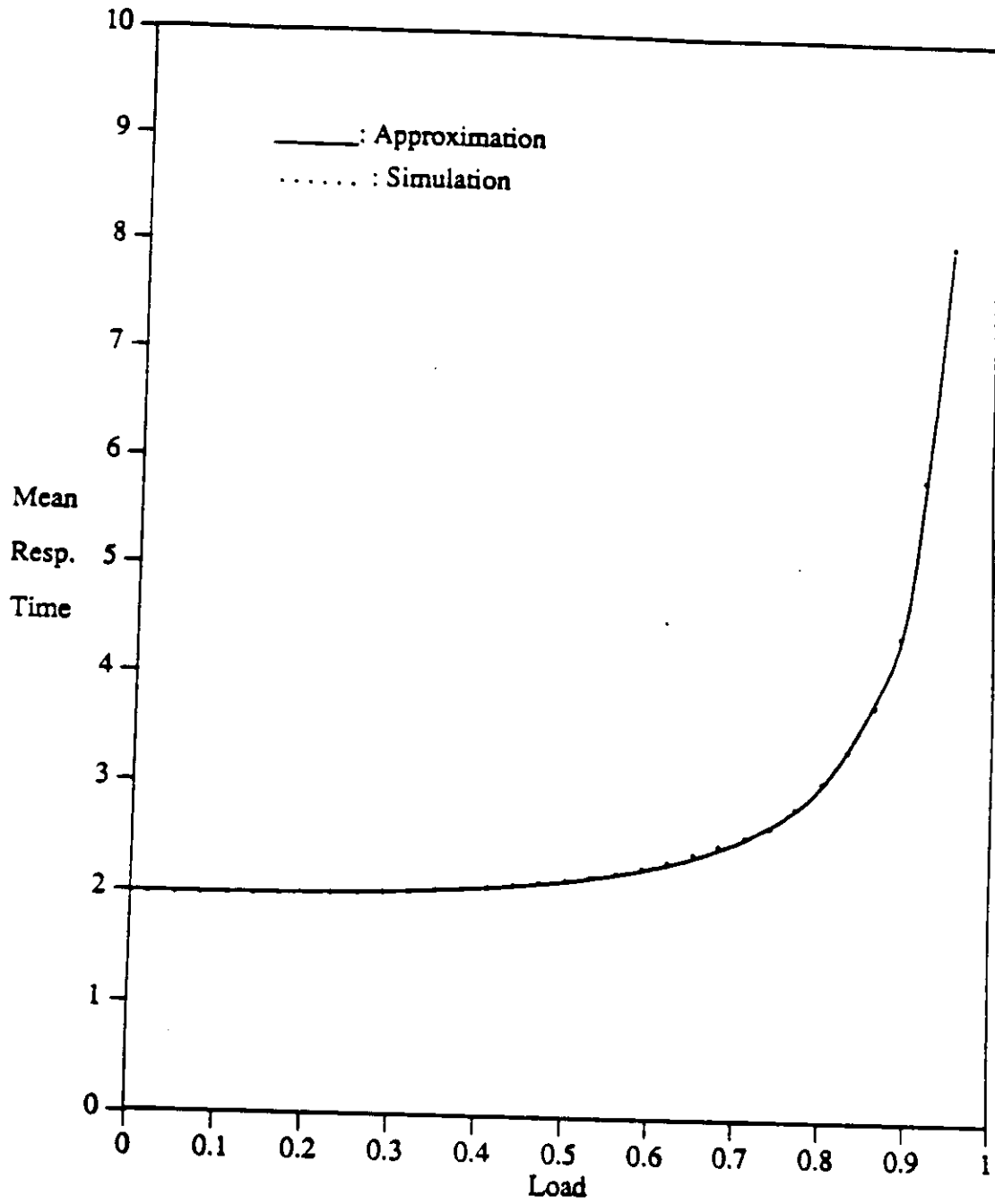


Figure 3.10  
 An Example Using Scale-up Rule: M/D/3 with  $\bar{x} = 2$



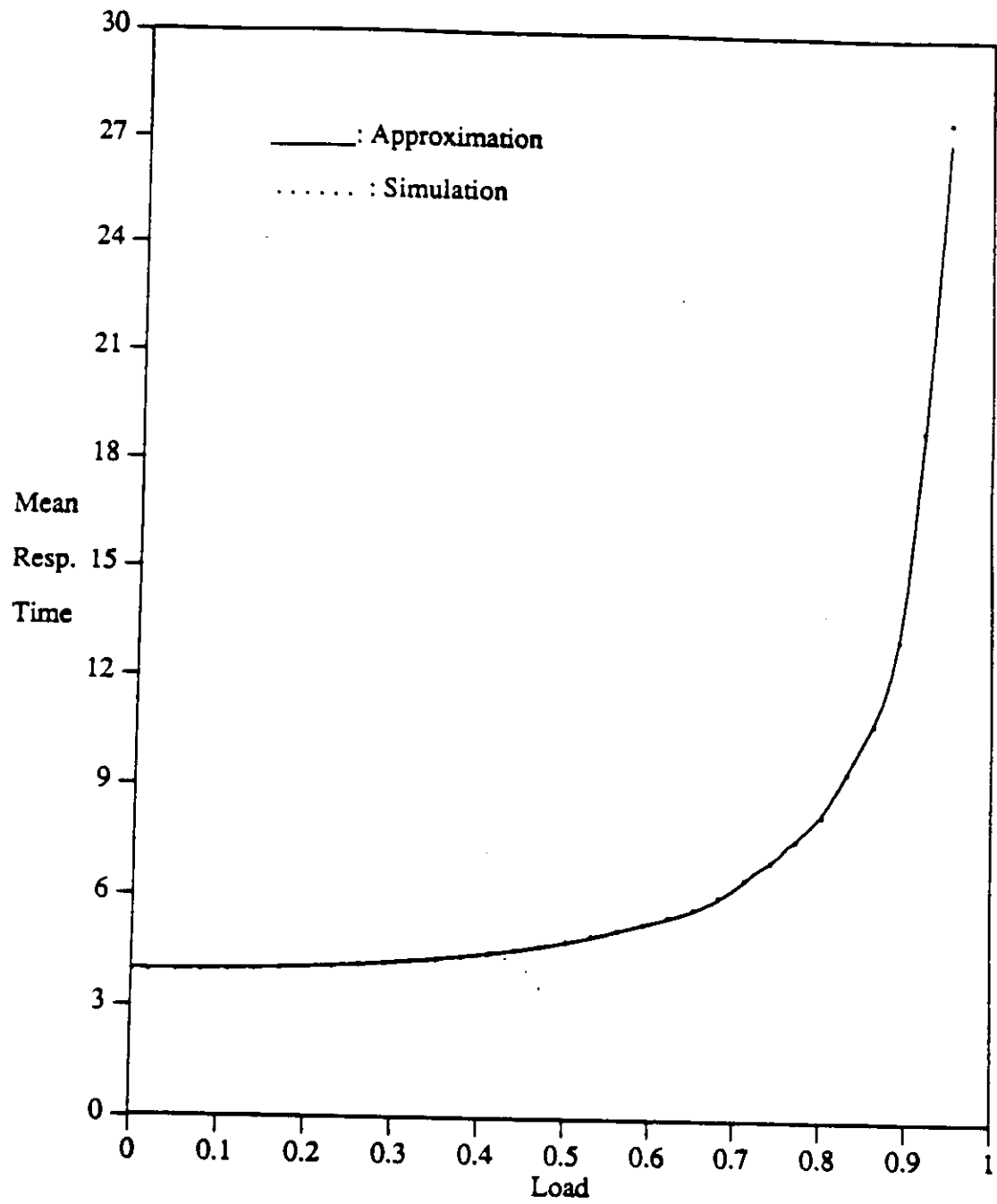


Figure 3.11  
 An Example Using Scale-up Rule:  $M/E4/2$  with  $\bar{x} = 4$

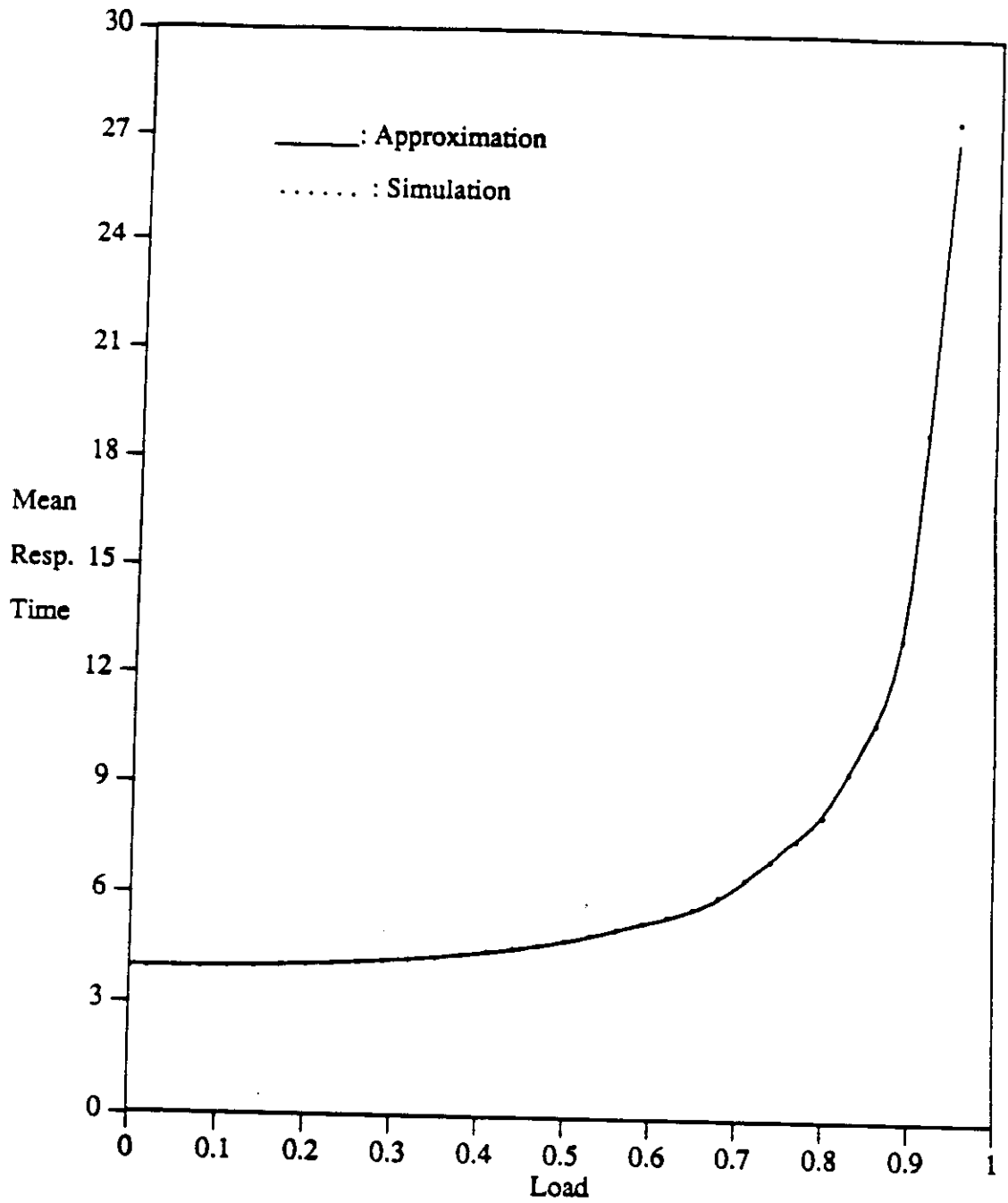


Figure 3.12  
 An Example Using Scale-up Rule:  $M/H3/2$  with  $\bar{x} = [1, 10, 100]$

the result for the  $MITG/m$  system. Thus large amount of time can be saved using this rule.

### 3.5.3 Available Processors Exceed Maximum Number of Processors Required in the Two-Stage Model

In this section we study the case where the available processors exceed the maximum number of processors required in the two-stage model. The exact solution is not feasible since the analysis gets too complicated; hence, an approximation model will be given. The approximation result is a combination of the use of the exact result from section 3.4 and the use of the Scale-up Rule.

Suppose the number of available processors is  $m$  times of the number of processors required by the high-concurrency stage. We first find the results of the system with the same processor-time task graph assuming the number of processors in the system equals the number of processors required by the high-concurrency stage. The result for this model is provided in section 3.4. With these results, we apply the scale-up rule to get the approximate mean response time. Some examples are given to show how good these approximation results are. Figures 3.13 to 3.18 give a comparison between the simulation result and the approximation result using the Scale-up Rule. Three cases are given for two different jobs: one is with  $\vec{P} = [1, 2]$  and  $\vec{T} = [1, 1]$ , and the other is with  $\vec{P} = [2, 1]$  and  $\vec{T} = [1, 1]$ . Figures 3.13 and 3.14 shows the result when the available number of processors is twice as many as the maximum number of processors required. Figures 3.15 and 3.16 shows the result when the available number of processors is three times as many as the maximum number of processors required. Figures 3.17 and 3.18 shows the result when the available number of processors is five times as many as the maximum number of processors required.

### 3.6 An Approximation for the General Cases

As we mentioned earlier, to exactly evaluate the performance of a general case is extremely difficult. In this section, using the exact solution of the two-stage model and the scale-up rule, we give an approximation method for any general processor-time task graph and any number of processors in the system. The simulation of this approximation method shows reasonably good results.

Given a processor-time task graph, we divide the processor-time task graph into two pieces such that the mean service time for each piece is the same. In each piece, we average the work load over its service time to find the average number of processors needed. By doing this, we achieve a two-stage model as analyzed in section 3.4. We use this two-stage model as the basis for the approximate model.

If the first stage requires fewer processors than the second stage, we use the result obtained in section 3.4.1. An example is shown in Figure 3.19. In Figure 3.19(a), the processor vector for this 4-stage job is  $[3, 1, 3, 9]$  and the corresponding time vector is  $\left[ \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2} \right]$ . By dividing the time vector into two equal pieces, the first two stages of Figure 3.19(a) will be merged into the first stage of the approximation model as shown in Figure 3.19(b). Similarly, the last two stages of Figure 3.19(a) will be merged into the second stage of the approximation model as shown in Figure 3.19(b). In order to make the work in Figure 3.19(b) to be the same as in Figure 3.19(a), the processor vector for Figure 3.19(b) is  $[2, 6]$  and the time vector is  $[1, 1]$ . Figures 3.20 and 3.21 shows the approximation result for this example given 12 and 45

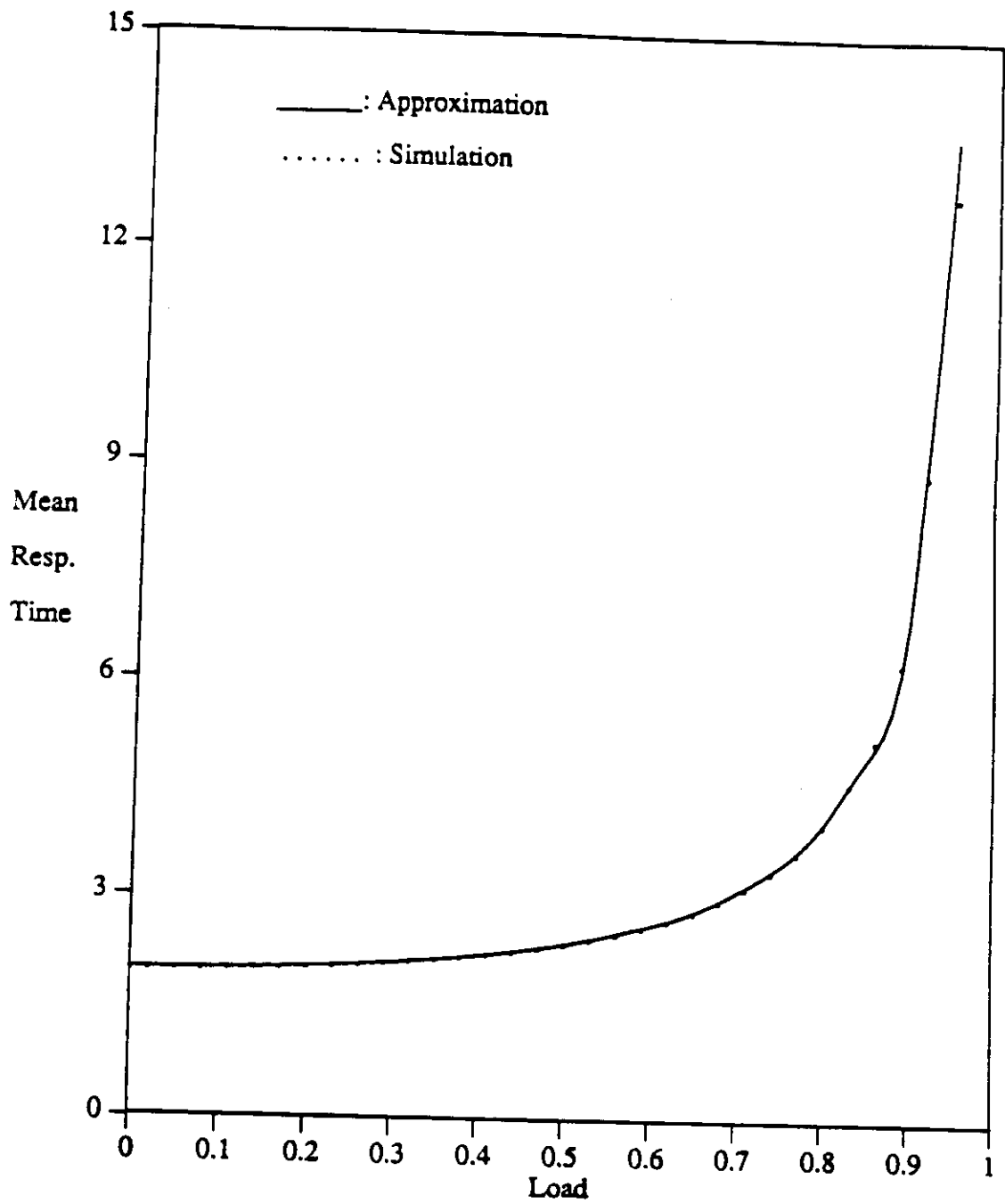


Figure 3.13  
 An Approximation Result for  $\vec{P} = [1,2]$  and  $P=4$

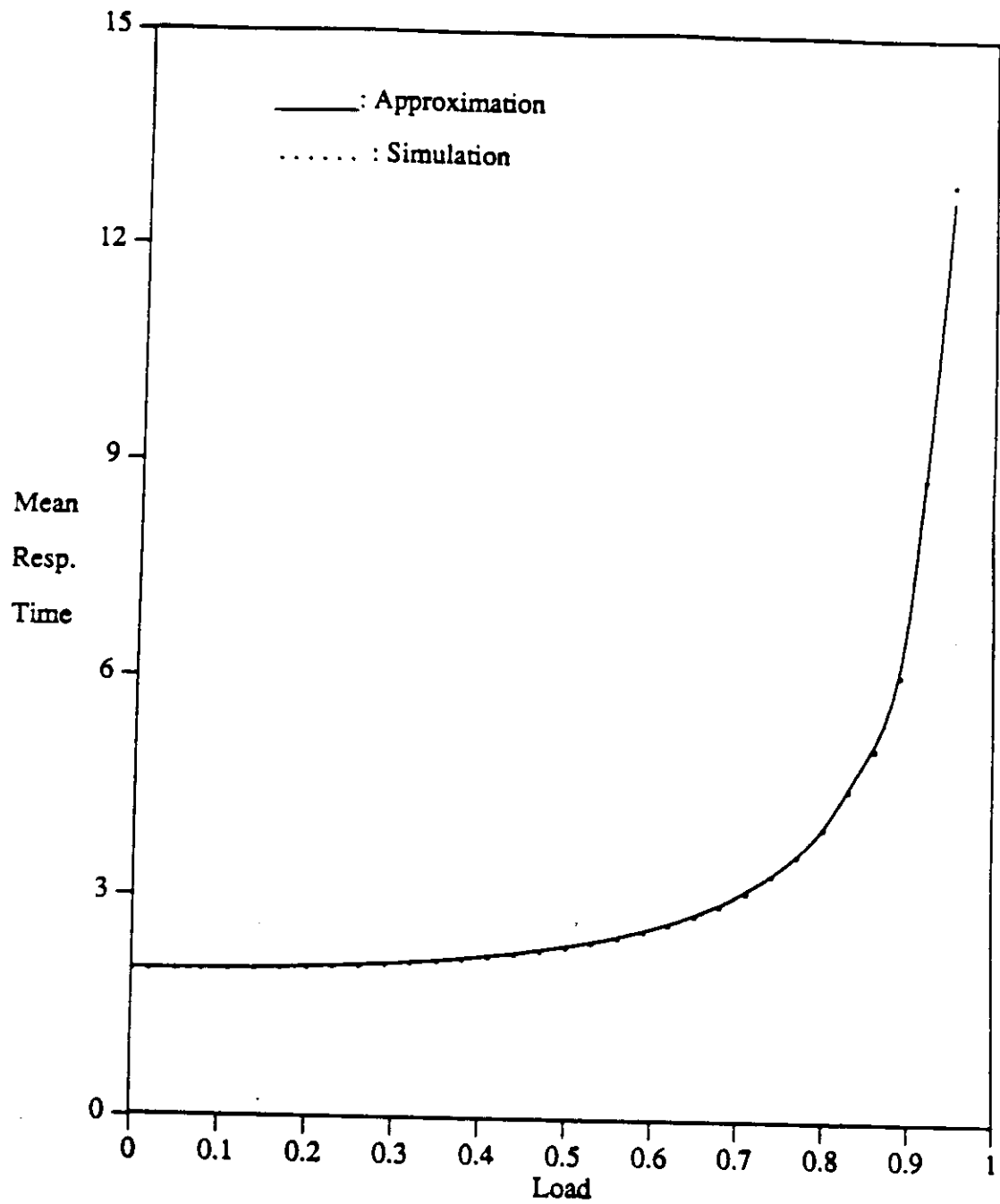


Figure 3.14  
 An Approximation Result for  $\vec{P} = [2.1]$  and  $P=4$

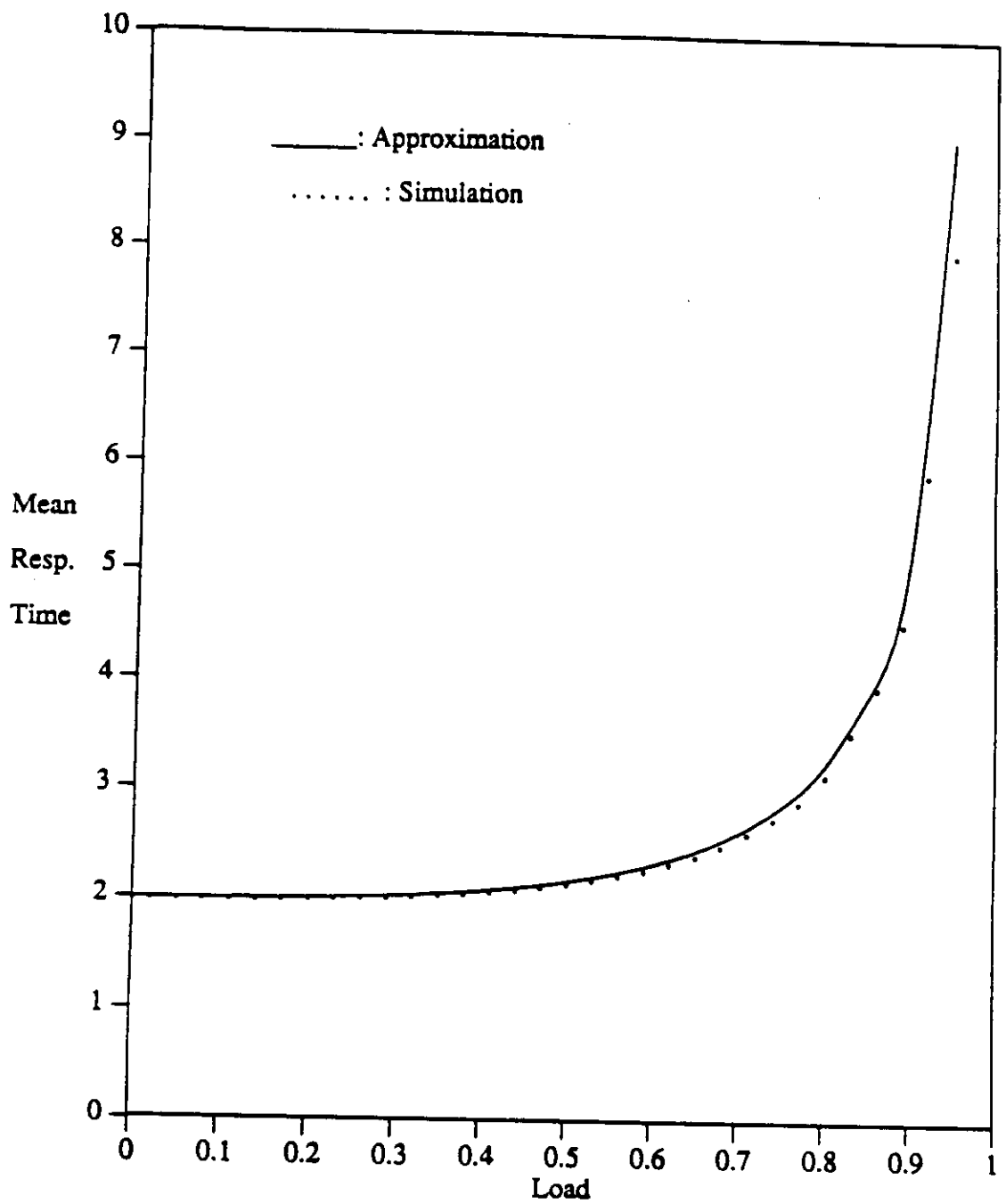


Figure 3.15  
 An Approximation Result for  $\vec{P} = [1,2]$  and  $P=6$

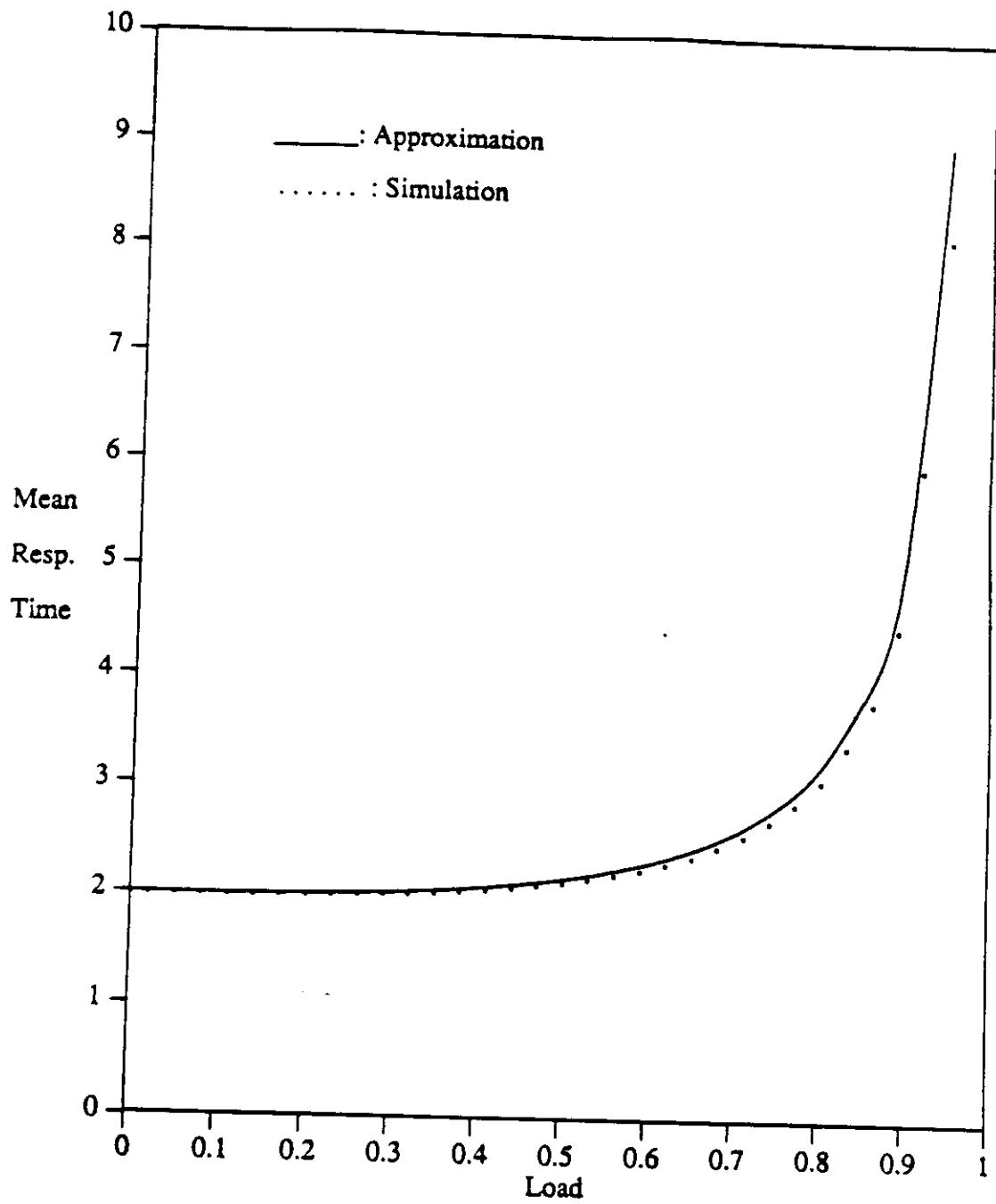


Figure 3.16  
 An Approximation Result for  $\vec{P} = [2,1]$  and  $P=6$

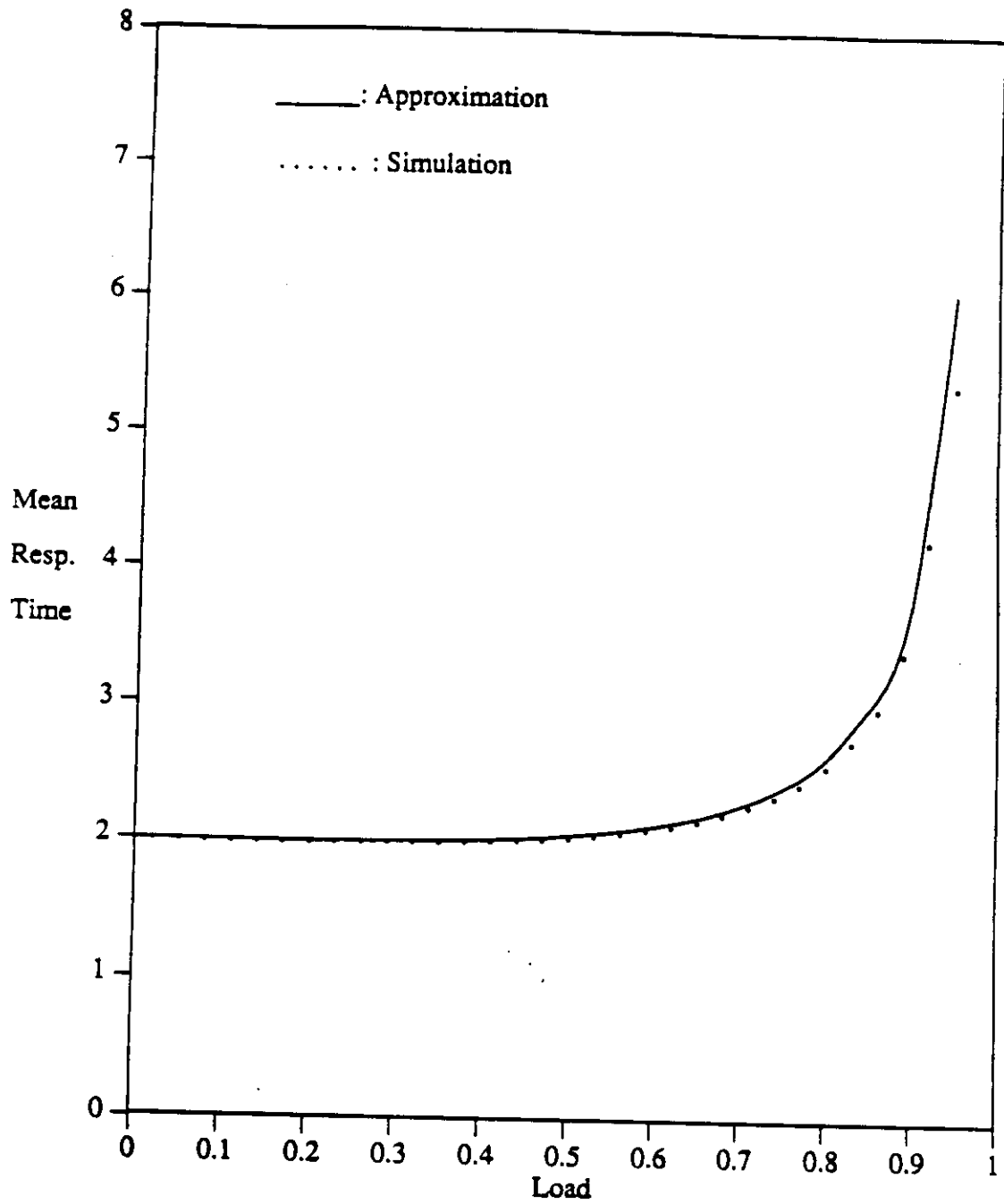


Figure 3.17  
 An Approximation Result for  $\vec{P} = [1,2]$  and  $P=10$



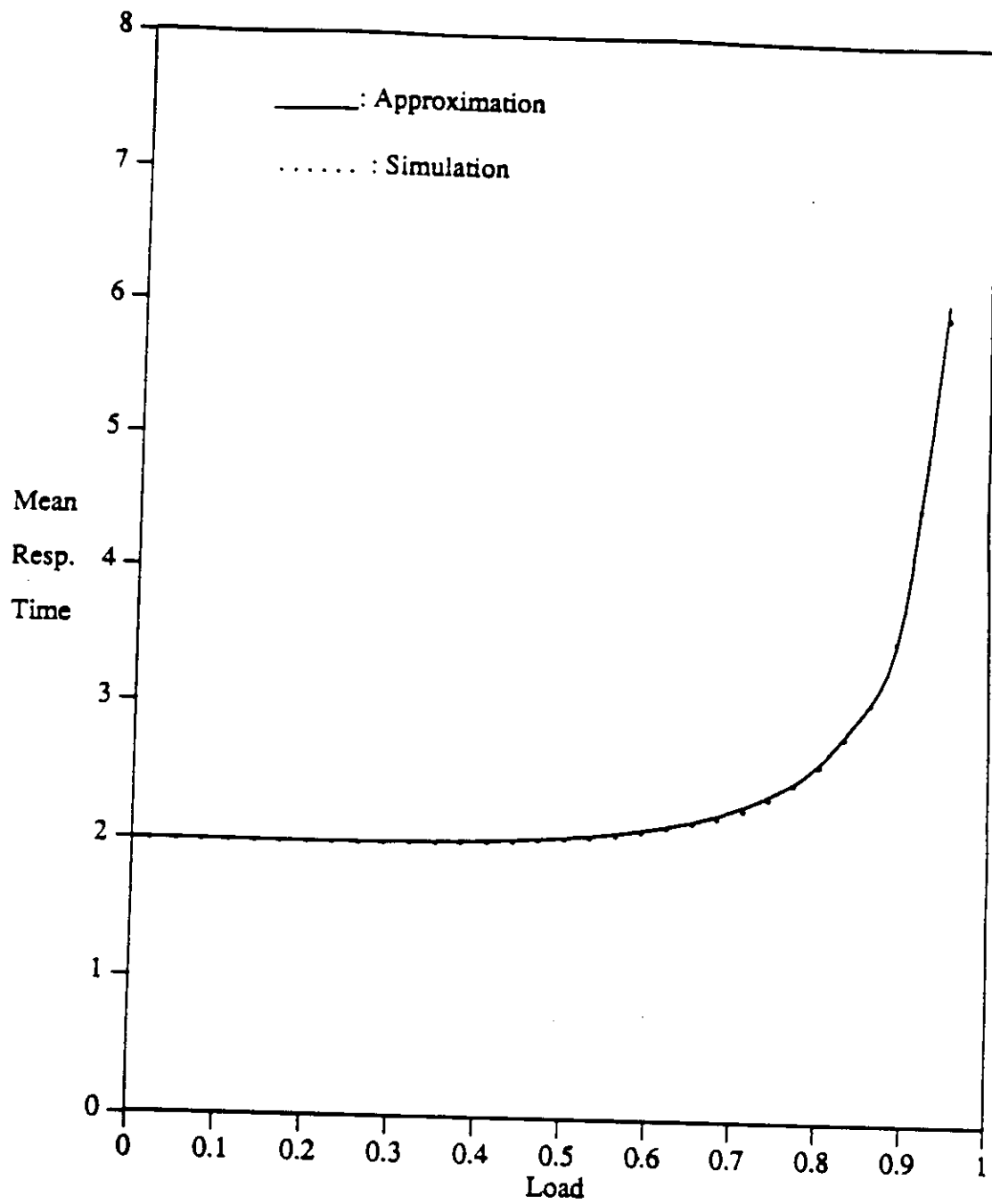


Figure 3.18  
 An Approximation Result for  $P = [2,1]$  and  $P=10$

processors in the system respectively. From these figures, we see that the approximation results are very close to the simulation results.

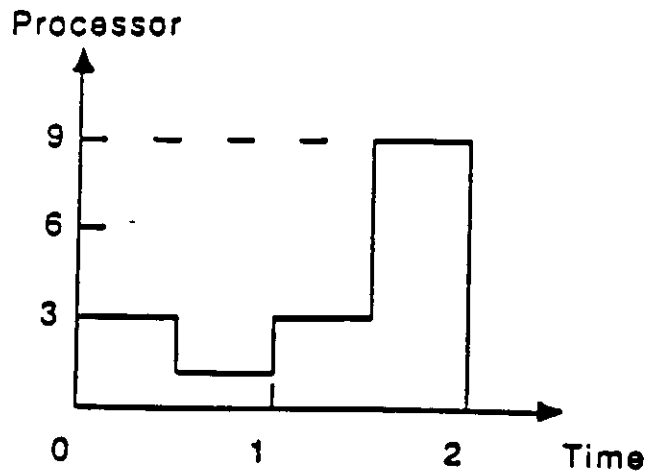


Figure 3.19(a)

An Example with  $\vec{P} = [3, 1, 3, 9]$  and  $\vec{T} = [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$

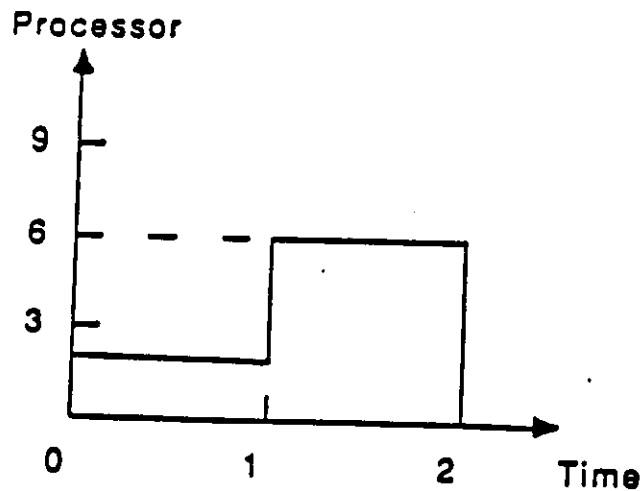


Figure 3.19(b)

An Approximation Model for Figure 3.19(a)

If the first stage requires more processors than the second stage, there is one more step to be done in the approximation method. An example is given in Figure 3.22. In Figure 3.22(a), the processor vector is  $[3, 9, 3, 1]$  and the time vector is  $[\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$ . Applying the same rule as we did in Figure 3.19, we convert Figure 3.22(a) into 3.22(b) such that the processor vector and the time vector of Figure 3.22(b) are

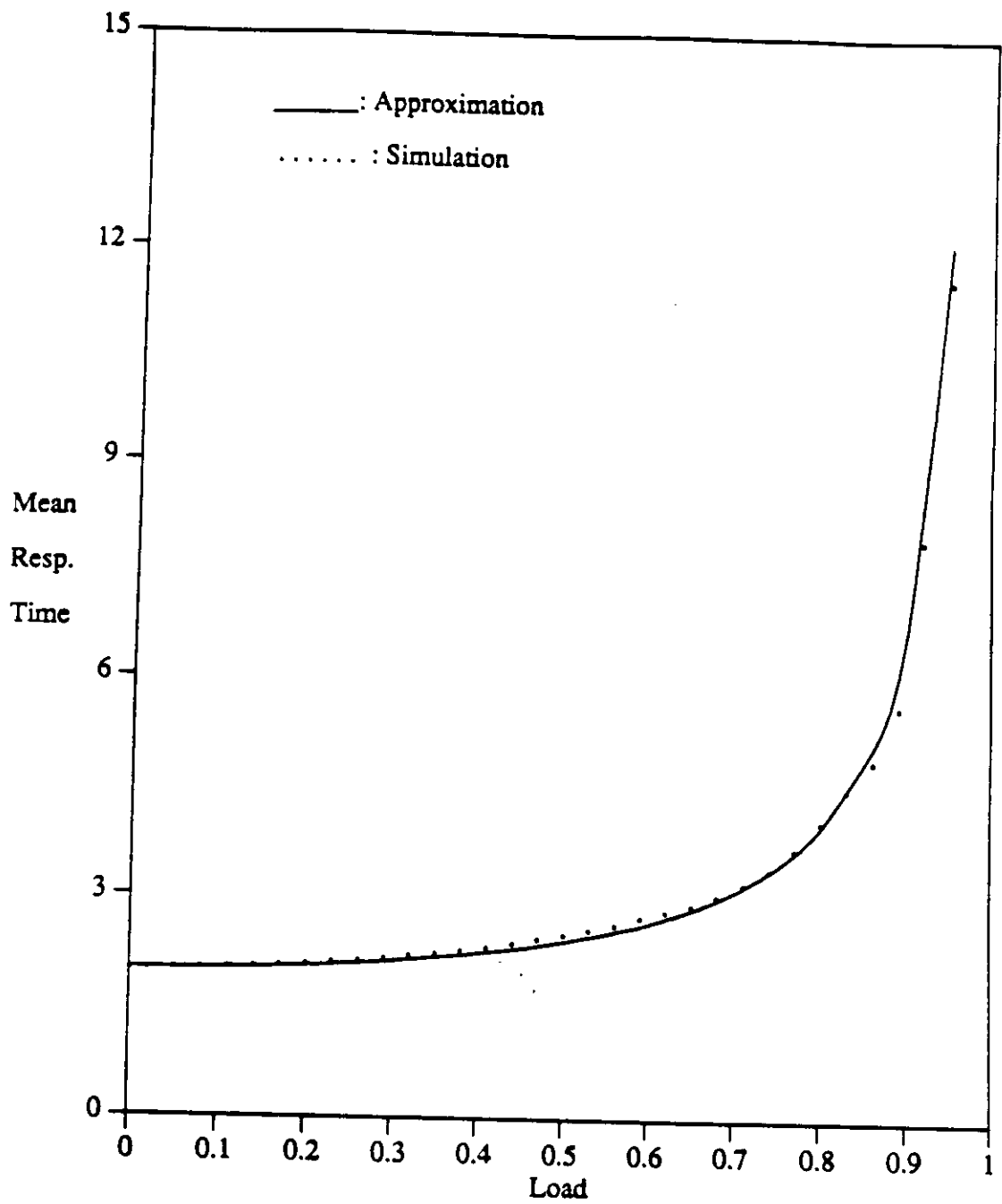


Figure 3.20  
 An Approximation Result for Figure 3.19(a) and P=12

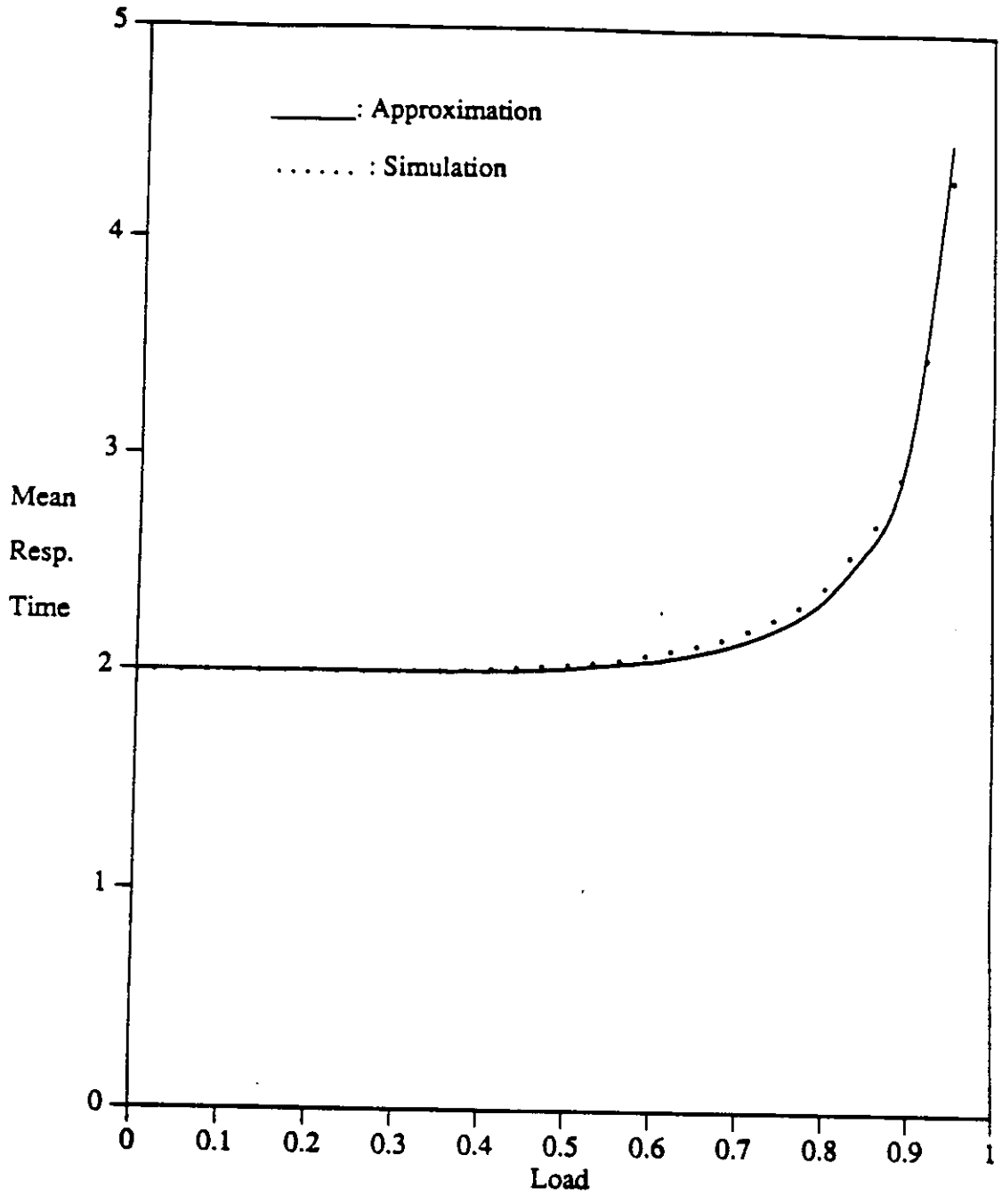


Figure 3.21  
 An Approximation Result for Figure 3.19(a) and P=45

[6,2] and [1,1] respectively. Since we can analyze the system only when the number of processors required in the first stage is exactly twice that in the second stage, we have to modify the two-stage approximation model by using a three-stage approximation model. The first stage of the three-stage model (as shown in Figure 3.22(c)) is the same as the first stage of the two-stage model (as shown in Figure 3.22(b)). The second stage of the three-stage model is modified such that it requires exactly half of the processors required in the first stage and the total work required in the stage is the same as that of the second stage from the two-stage model. The third stage in the 3-stage model is used to adjust the no-queueing service time such that it is the same for the 2-stage approximation model and the 3-stage approximation model. The processor vector and the time vector for Figure 3.22(c) are  $[6,3,0]$  and  $[1, \frac{2}{3}, \frac{1}{3}]$  respectively. Although the model in Figure 3.22(c) is different from the model described in section 3.4.2, it can be solved by using the result from section 3.4.2. Notice that the third stage has the highest priority and requires no processors from the system at all, the existence of stage three has no impact on stages one and two. Hence, we can solve this problem by first neglecting the third stage in Figure 3.22(c) and apply the results obtained from section 3.4.2; from this result, we add the mean service time of stage three to it to get the overall mean response time. Figures 3.23 and 3.24 shows the approximation results for this example given there are 12 and 45 processors in the system respectively.

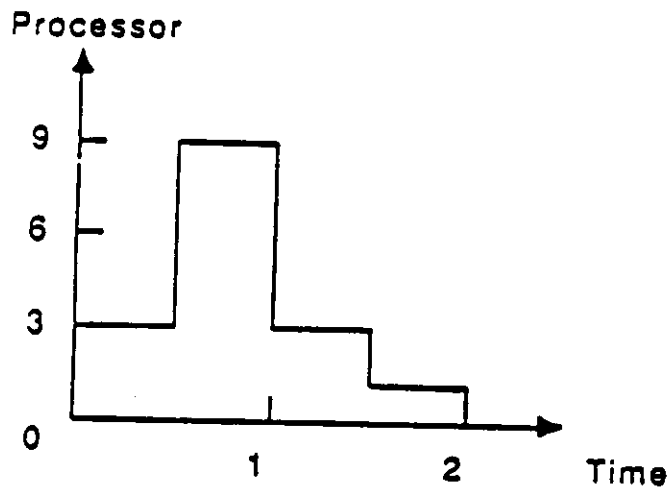


Figure 3.22(a)

An Example with  $\vec{P} = [3, 9, 3, 1]$  and  $\vec{T} = [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$

### 3.7 Conclusion

In this chapter, we were able to analyze some interesting models exactly to study the behavior of jobs in a multi-processor system. By introducing the scale-up rule, which is one of the major contributions of this chapter, we are able to give an approximation method for finding the mean response time for any general application.

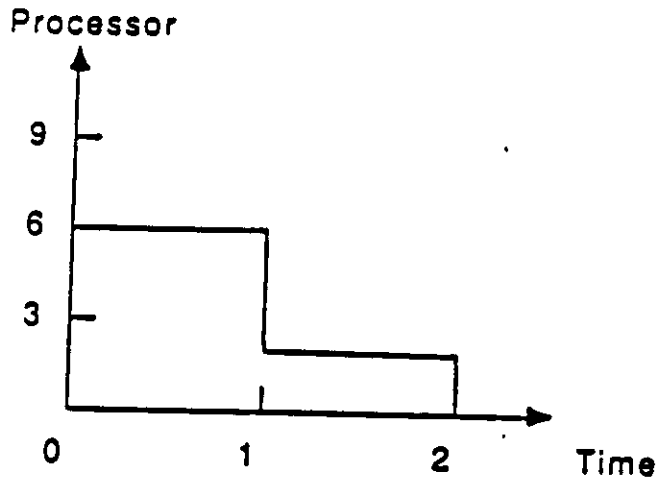


Figure 3.22(b)  
The First Step Toward Approximation Model for Figure 3.22(a)

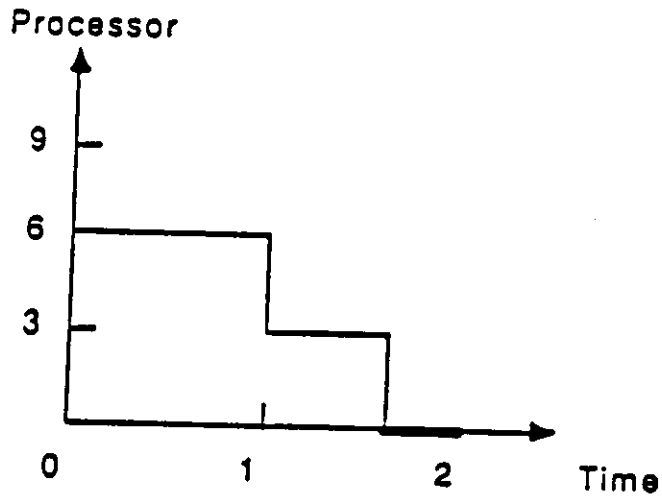


Figure 3.22(c)  
The Second Step Toward Approximation Model for Figure 3.22(a)

By looking at the comparisons in sections 3.3.2 and 3.4.3, we have the following rule of thumb in designing parallel algorithms: If there is blocking in the algorithm, late blocking will generate a longer mean response time than early blocking if the service discipline is the same as given in this chapter. Hence, if we have the choice, we would like to have early blocking rather than late blocking in the algorithm.

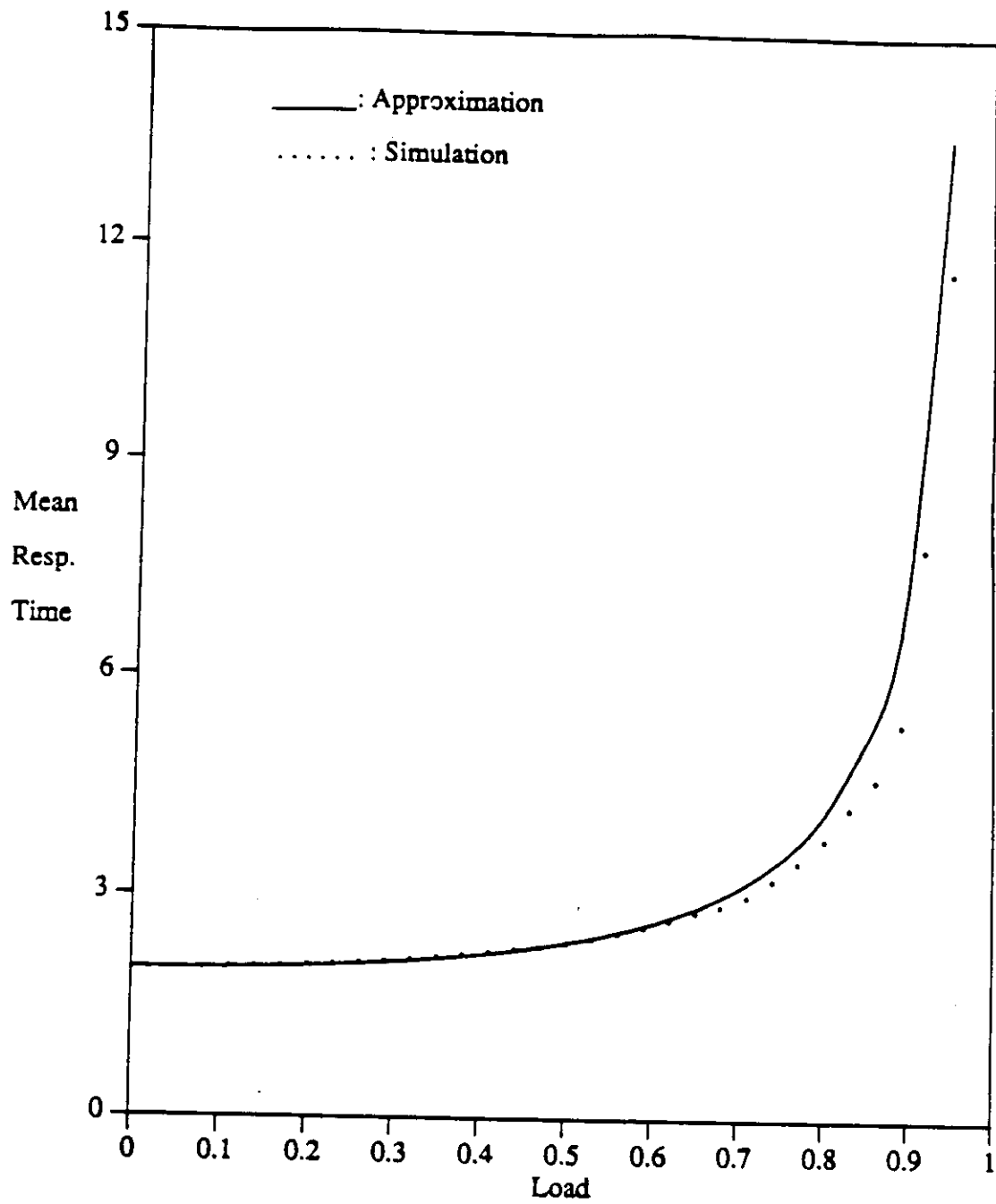


Figure 3.23  
An Approximation Result for Figure 3.22(a) and  $P=12$

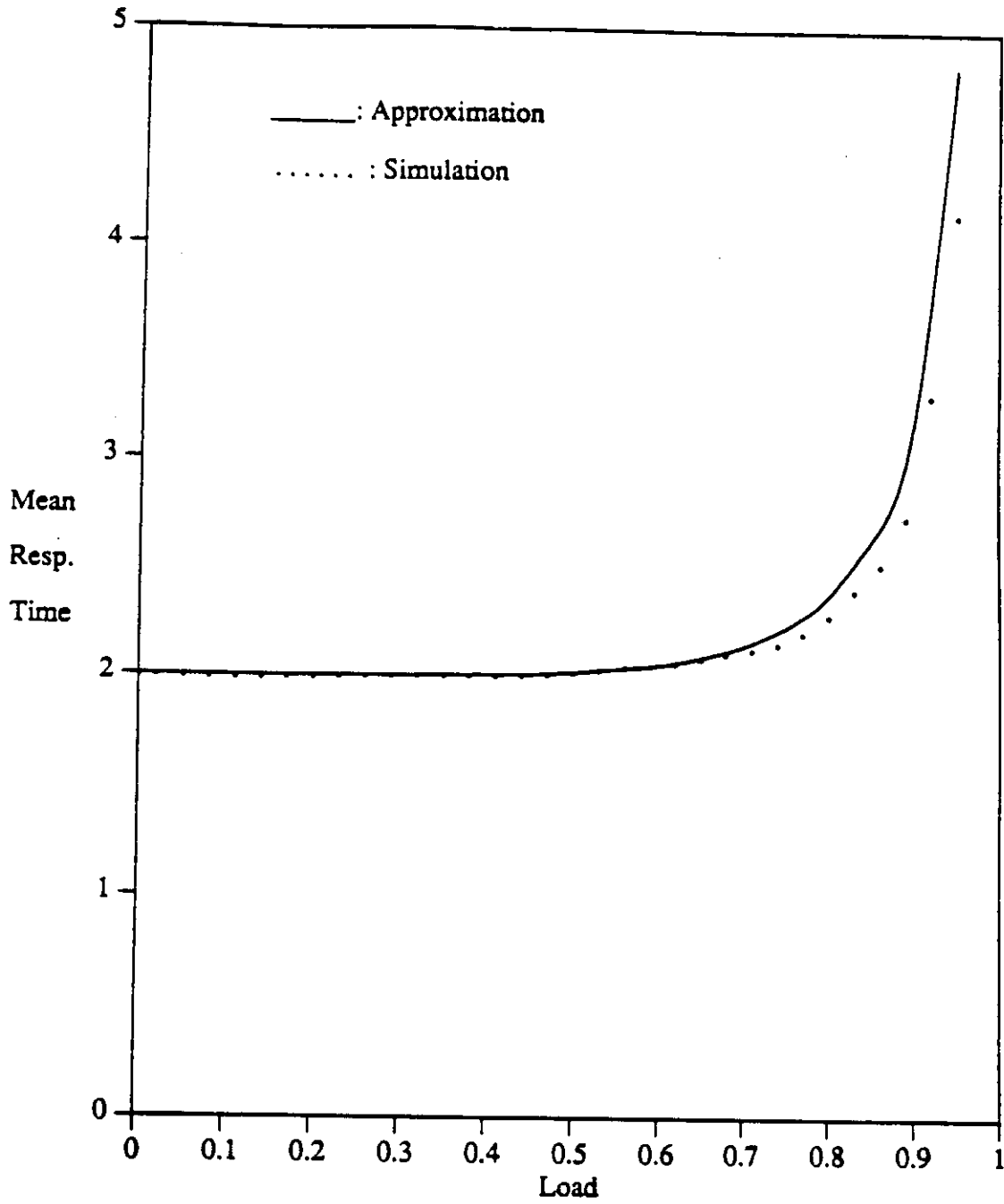


Figure 3.24  
An Approximation Result for Figure 3.22(a) and  $P=45$



## CHAPTER 4

### AN OPTIMAL PARALLEL MERGING AND SORTING ALGORITHM USING $\sqrt{N}$ PROCESSORS WITHOUT MEMORY CONTENTION

Having studied the theoretical performance of parallel processing systems in the previous two chapters, we would like to look at some applications (parallel algorithms) which can be run on a parallel processing system. In this chapter, we first present an optimal parallel merging algorithm which uses  $\sqrt{N}$  processors to achieve a linear processing time speedup without memory contention. By using this optimal parallel merging algorithm, we present a parallel sorting algorithm also using  $\sqrt{N}$  processors to achieve a linear processing time speedup without memory contention.

Merging and sorting algorithms probably are two of the most commonly used algorithms in the business world. Knuth pointed out in [KNUT73] that computer manufacturers estimated that over 25 percent of the running time on their computers is being spent on sorting and there are many installations in which sorting uses more than half of the computing time. Furthermore, the sorting algorithm is not only the most commonly used algorithm, it is also a time consuming algorithm. An efficient algorithm for both merging and sorting is so important in the world of algorithm design that numerous papers have been published in this field. When the distributed and parallel processing systems were brought into light, people tried to have more processors working concurrently to speedup the computation. This is also the motivation of this chapter.

A parallel algorithm for any given computational problem is optimal if the processing time speedup by using  $P$  processors compared to the known best serial algorithm (i.e., the best algorithm when only one processor is used) is a linear function of  $P$ . In other words, if we denote  $T(P)$  to be the time complexity of the algorithm using  $P$  processors, the algorithm is optimal if  $P \cdot T(P)$  has the same time complexity of any known best serial algorithm.

In this chapter two algorithms will be presented which run on a shared-memory parallel processing system. Furthermore, no simultaneous read from or write into the same memory location is required for these algorithms. We first present a parallel merging algorithm which uses  $P = \sqrt{N}$  processors to merge two sorted lists, each of length  $N$ , in time  $O\left(\frac{N}{P}\right) = O(\sqrt{N})$ , where  $P$  is the number of processors used. We then apply this merging algorithm to construct a sorting algorithm which uses  $\sqrt{N}$  processors to sort  $2N$  elements in time  $O\left(\frac{N \cdot \log N}{P}\right) = O(\sqrt{N} \log N)$ . Both algorithms are optimal because they both achieve linear processing time speedup (i.e., linear with  $P$ ) and optimal time complexity.

#### 4.1 Previous Work

There have been many parallel sorting algorithms published in the literature, [AKL85] and [LAKS84] provide good references to many of these algorithms. However, there are not many parallel algorithms for merging.

Batcher has an odd-even merge algorithm [BATC68] which, in order to merge two lists each of length  $N$ , will first merge the odd sequences from both lists, then merge the even sequences from both lists. Finally, it applies the comparison-interchange operation to yield a sorted list. This algorithm uses  $1 + N \log N$  processors to achieve a time complexity of  $1 + \log N$ . This algorithm does not achieve linear processing time speedup.

Valiant [VALI75] has a parallel merging algorithm which uses  $N$  processors and has a time complexity of  $O(\log \log N)$ . This algorithm does not quite achieve linear processing time speedup.

Thompson and Kung [THOM77] have a merging algorithm which uses  $2N$  processors connected in a *linear array* containing two sorted lists each of length  $N$ , where the first list is in the first  $N$  processors and the second list is in the second  $N$  processors. Their algorithm will rearrange these elements in such a way that the  $i^{\text{th}}$  smallest element is in the  $i^{\text{th}}$  processor. By using  $2N$  processors their algorithm achieves a time complexity of  $\log N$  and  $8N$  routings, where one routing is defined as sending one element from one processor to another processor. A similar result, which also uses a linear array, is reported by Kumar and Hirschberg [KUMA83] which merges two sorted lists using  $2N$  processors in  $\log N$  time and  $\frac{3}{2}N$  routings. Thompson and Kung [THOM77] also have another parallel merging algorithm which uses a two-dimensional array of  $2N$  mesh-connected processors to merge two sorted lists each of length  $N$  in  $O(N)$  time. These algorithms do not achieve linear processing time speedup also.

In this chapter, we first describe a Multi-way Parallel Merging algorithm which uses  $\sqrt{N}$  processors to achieve a processing time speedup which is linearly proportional to the number of processors used. By recursively applying this algorithm, we can show that for  $P = N^{\frac{k-1}{2}}$ , the processing time speedup of the algorithm is  $\frac{P}{3^k}$ . For example, if  $P = N^{3/4}$ , then the processing time speedup of this algorithm is  $\frac{P}{9}$ , which is a very high speedup.

#### 4.2 Multi-way Parallel Merge Algorithm

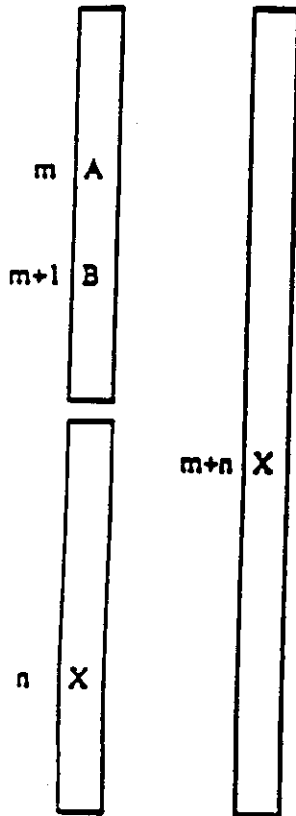
The input of this merging algorithm is two sorted lists, namely  $L_1$  and  $L_2$ , each of length  $N$ . We would like to merge these two lists into one sorted list using  $P$  processors. We will describe the algorithm for the case when  $N = P^2$ .

The idea of this merging algorithm is first to divide each of the two sorted lists into  $P$  sorted sub-lists in such a way that each sub-list in  $L_1$  or  $L_2$  contains elements which are  $P$  positions apart in  $L_1$  or  $L_2$ . We then assign processor  $P_i$  to merge the  $i^{\text{th}}$  sub-list from  $L_1$  and the  $i^{\text{th}}$  sub-list from  $L_2$  ( $i = 1, 2, \dots, P$ ). We call this procedure phase 1. Obviously, all  $P$  processors will be working concurrently and will finish approximately at the same time (i.e., in  $2 \cdot \frac{N}{P}$  steps) since the two sub-lists for each processor to merge are

of the same length ( $\frac{N}{P}$ ). As we prove in Theorem 4.1, after phase 1, every element is at most a distance  $P$  from its final position! Next we take the result from phase 1 and group every  $P$  consecutive elements in the full list as one group. After the grouping we have  $2P$  groups. Note that the  $P$  elements in each group are sorted. In phase 2 we merge the  $i^{\text{th}}$  group with the  $(i+1)^{\text{st}}$  group using  $P_{(i+1)/2}$  (for all odd  $i$ ). Since there are  $2P$  groups and each processor merges 2 groups, we have all the processors working concurrently and they will finish at the same time once again. In phase 3, we merge the  $j^{\text{th}}$  group with the  $(j+1)^{\text{st}}$  group using  $P_{j/2}$  (for all even  $j$ ). Note that in phase 3 only  $2P - 2$  groups need to be merged; hence we use  $P - 1$  processors concurrently and they will finish at the same time. In phase 2 and 3, every element is free to move up or down by more than  $P$  positions; therefore, after phase 3 this list is sorted.

**THEOREM 4.1:** After phase 1, every element is within  $\pm P$  positions of its final position.

[Proof]



Suppose elements A, B and X are assigned to processor  $P_i$ , where A is the  $m^{\text{th}}$  and B is the  $(m+1)^{\text{st}}$  element in the sublist from  $L_1$  for  $P_i$ , and X is the  $n^{\text{th}}$  element in the sublist from  $L_2$  for  $P_i$ . We also assume  $A < X < B$ .

(1) Number of elements smaller than X is at least

$$(m-1)P + i + (n-1)P + i = [(m+n-1)P + i] + (i-P)$$

(2) Number of elements greater than X is at most

$$\begin{aligned} & [(m+1-1)P + i - 1] + [(n-1)P + i] \\ &= [(m+n-1)P + i] + (i-1) \end{aligned}$$

From (1) and (2), the ranking of X is between

$$[(m+n-1)P + i] + (i-P) \text{ and } [(m+n-1)P + i] + (i-1).$$

The position of X after phase 1 is:

$$(m+n-1)P + i$$

Therefore, element X is within a distance  $P$  from its final position.

**ALGORITHM:**

Step 1:

Divide  $L_1$  into  $P$  sub-lists where the  $i^{\text{th}}$  sub-list contains the  $i, P+i, 2P+i, \dots, N-P+i$ . Also divide  $L_2$  into  $P$  sub-lists in the same way as  $L_1$ .

Step 2:

For all  $i$  from 1 to  $P$ , assign the  $i^{\text{th}}$  sub-list from  $L_1$  and the  $i^{\text{th}}$  sub-list from  $L_2$  to  $P_i$ . Have each processor merge its two sub-lists using the same memory spaces (i.e., every  $P$  slots).

Step 3:

Group the list resulting from step 2 into  $2P$  groups such that each group contains  $P$  elements. For all  $i$  from 1 to  $P$ , assign the  $(2i-1)^{\text{th}}$  group and the  $(2i)^{\text{th}}$  group to  $P_i$ . Have each processor merge its two groups.

Step 4:

For all  $i$  from 1 to  $(P-1)$ , assign the  $2i^{\text{th}}$  group and the  $(2i+1)^{\text{th}}$  group to  $P_i$ . Have each processor merge its two groups.

Example 4.1:

Figure 4.1(a) shows two sorted lists of equal length. After step 2 in the algorithm we have the list as shown in Figure 4.1(b). After step 3 we have the list as shown in Figure 4.1(c). After step 4 we have the final sorted list as shown in Figure 4.1(d) and the algorithm is completed.

Complexity Analysis:

In phase 1, we have all  $P$  processors working concurrently with each processor merging two sub-lists each of length  $\frac{N}{P}$ ; hence, the total computational complexity for phase 1 is  $P \cdot \left[2 \cdot \frac{N}{P}\right] = 2N$ . This is also true for phase 2. For phase 3, the total computational complexity is  $(P-1) \cdot \left[2 \cdot \frac{N}{P}\right] = 2N - \frac{2N}{P}$ . Therefore the total computational complexity is at most  $6N$ . It is easy to see from the algorithm that all  $P$  processors are working all the time (except that one processor is idle in phase 3) and that there is no overlapped work between processors; therefore the time complexity is at most  $\frac{6N}{P}$ . We know that the computational complexity for merging two lists, each of length  $N$ , using one processor, is at least  $2N$ . Therefore, we have a processing time speedup of  $\frac{P}{3}$ .

Now, we show how this algorithm can be modified to be able to use more than  $N^{1/2}$  processors. In the original algorithm ( $P = N^{1/2}$ ), we use one processor to merge two sub-lists each with  $N^{1/2}$  elements. We now focus our attention on merging these two sub-lists. In the modified algorithm, we will use more than one processor to do this merging. Since there are  $N^{1/2}$  elements in each sub-list, we can use  $(N^{1/2})^{1/2} = N^{1/4}$  processors to merge these two sub-lists using the original algorithm by tripling the time required. Since  $N^{1/2}$  of these happen concurrently, the total number of processors required is  $N^{1/2} \cdot N^{1/4} = N^{3/4}$ . However, since the required time is tripled, the overall speedup becomes  $\frac{P}{3} \cdot \frac{1}{3} = \frac{P}{3^2}$ . By repeatedly using this idea on merging two sub-lists with only one processor, it can easily be shown that we can use  $P = N^{\frac{2^k - 1}{2^k}}$  processors for this merging algorithm to achieve a speedup of  $\frac{P}{3^k}$ .

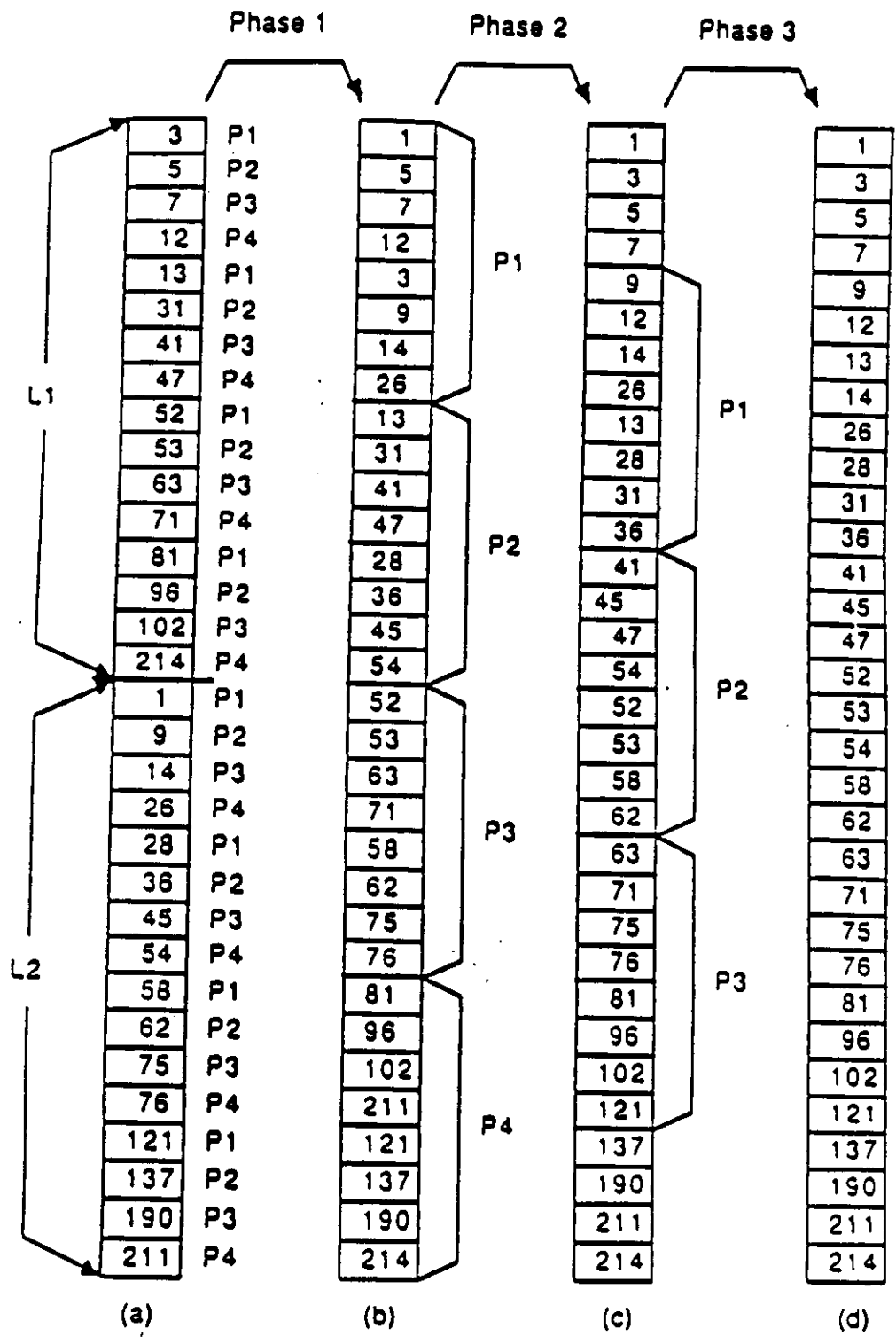


Figure 4.1  
An Example

### 4.3 Multi-way Parallel Sorting Algorithm

We construct a multi-way parallel sorting algorithm using the merging algorithm described above. We use  $\sqrt{N}$  processors to sort  $2N$  elements. For ease of explanation, we assume  $P = \sqrt{N} = 2^k$ . In the first step of the algorithm, we assign  $2\sqrt{N}$  elements to each processor and have each processor sort its data using any known optimal sequential sorting algorithm. We divide the second step into  $k$  phases. At the beginning of the  $i^{\text{th}}$  phase ( $i=1,2,\dots,k$ ), there are  $2^{k+i-1}$  sorted lists with  $2^{i-1}$  processors associated with each list. In each phase, we concurrently merge pairs of sorted lists using the processors associated with those two sorted lists. Therefore, in the  $i^{\text{th}}$  phase, we merge two sorted lists, each of length  $\frac{2N}{2^{k+i-1}} = 2^{k+i}$ , using  $2 \cdot 2^{i-1} = 2^i$  processors. Let us define  $N^{(i)}$  to be the length of each sublist to be merged in phase  $i$  and  $P^{(i)}$  to be the number of processors used to merge two sublists in phase  $i$ . Since  $\sqrt{N^{(i)}} \geq P^{(i)}$  ( $\sqrt{N^{(i)}} = \sqrt{2^{k+i}} \geq \sqrt{2^{i+i}} = 2^i = P^{(i)}$ ), we can apply the multi-way merging algorithm to achieve linear processing time speedup. After  $k$  phases, there is only one sorted list left; hence the algorithm terminates.

Complexity Analysis:

The time complexity of step 1 is  $O(2\sqrt{N} \log(2\sqrt{N})) = O\left(\frac{N \log \sqrt{N}}{\sqrt{N}}\right) = O\left(\frac{N \log N}{P}\right)$ . In step 2, the time complexity for the  $i^{\text{th}}$  phase is  $O\left(\frac{2^{k+i}}{2^i}\right) = O(2^k) = O\left(\frac{N}{P}\right)$ . Since there are  $\log P$  phases in step 2, the total time complexity of step 2 is  $O\left(\frac{N}{P} \cdot \log P\right) < O\left(\frac{N \log N}{P}\right) =$  time complexity of step 1, then the total time complexity of this sorting algorithm is  $O\left(\frac{N \log N}{P}\right)$ .

### 4.4 Conclusion

Multi-way parallel merging and sorting algorithms provide a speedup of  $\frac{P}{3}$  when  $P = \sqrt{N}$ , which is a very high speedup. Furthermore, for  $P = N^{\frac{k-1}{2^k}}$ , the processing time speedup of the algorithm is  $\frac{P}{3^k}$ . This is a very high speedup as long as  $k$  is small, which is true for many applications. Another advantage of this algorithm is that we do not require concurrent read or concurrent write for this algorithm, hence this algorithm can be implemented on any version of parallel processing systems (EREW, CREW, or CRCW).

## CHAPTER 5

### A DISTRIBUTED SORTING ALGORITHMS USING BROADCAST COMMUNICATION NETWORKS

In this chapter we present a sorting algorithm for distributed computing systems using a broadcast channel to communicate between processors. An important issue in such an environment, which is usually neglected, is the queueing behavior of accessing the communication channel. In computational complexity theory, if algorithm A has a computational complexity  $N$  while algorithm B has a computational complexity  $2N$ , then  $T_B = 2T_A$ , where  $T_A$  ( $T_B$ ) is the time for algorithm A (B) to complete. This is not true for our system because of the queueing behavior of accessing the broadcast communication channel. From queueing theory [KLEI76] we know that doubling the load of the system does not imply doubling the system time. Especially when the load (utilization) of the system is high, a slight increase in load might incur a much longer system time. Therefore, for such a system, a slight decrease in the communication requirement might improve the performance dramatically.

Another issue which is also important is that for a random-access broadcast channel, even if the total amount of information (bytes) which has to be transmitted over the network is the same, the performance of a system with a smaller message size, which contains fewer bytes per message, might be far worse than a system with a larger message size, which contains more bytes per message. This is because the system with a larger message length requires fewer messages to convey the same amount of information (bytes). This reduces the amount of message overhead (e.g. header and check bits) and the time wasted on propagation delay; it also reduces the chance of collisions and retransmissions.

The beauty of this algorithm we present is that it requires much less inter-processor communication than some previously published algorithms. We also have an improved version which provides a parameter " $\alpha$ " which gives the user the freedom to decide whether to spend more time on communication and less time on computation or more time on computation and less time on communication.

#### 5.1 Previous Work

There have been some papers dealing with distributed sorting algorithms [WEGN82] [ROTE85] [ZAKS85] [DECH81] [MARB86]. Since we are interested in distributed computing systems with a broadcast communication channel, we will refer only to [DECH81] and [MARB86] for our comparisons.

Dechter and Kleinrock [DECH81] have an algorithm which makes use of a broadcast communication network to implement a distributed sorting algorithm. The advantage of their algorithm is that, regardless of the number of processors used, their algorithm has a communication complexity between  $N$  and  $2N$

(the average is  $\frac{3}{2}N$ ), where  $N$  is the number of elements which need to be sorted. The disadvantage of the algorithm is that each broadcast message contains only one element. It can be shown that their algorithm will incur, on the average,  $\frac{3}{2}N$  messages and  $\frac{1}{2}PN$  propagation delays, where  $P$  is the number of processors in the system. As we mentioned earlier, since there is a communication overhead for each message sent, the total communication overhead of their algorithm is fairly high, which is completely neglected in their model.

Marberg [MARB86] has another distributed sorting algorithm which also makes use of a broadcast communication network using more than one channel. This algorithm can achieve even higher concurrency by using all the communication channels. However, its disadvantage is that it has a communication complexity of  $4N$ , where  $N$  is defined as above.

## 5.2 A Distributed Sorting Algorithm

In this section we first present a sorting algorithm and then an improved version of it. We also present a modification of this algorithm in section 5.4 such that the communication complexity is  $N$  plus a term which is not a function of  $N$ . Note that sorting in such an environment requires a communication complexity of at least  $N$  since every element must be broadcasted at least once to the processor it should reside at the end of the algorithm. Therefore, our algorithm only adds an extra term which is not a function of  $N$ . This implies when  $N$  is large, our algorithm will be better than that in either [DECH81] or [MARB86].

As we mentioned earlier, inter-processor communication is usually the bottleneck of a distributed system. We propose an algorithm which can reduce the communication complexity. Furthermore, by having reduced the communication complexity, we can pack more than one element into a message which can further reduce the communication overhead.

The system has  $P$  processors and there are  $N$  elements to be sorted assuming  $N \gg P$ . We assume these  $N$  elements are in  $P$  unsorted lists (each of length  $\frac{N}{P}$ ), one for each processor. The purpose of this algorithm is to sort these lists such that all elements in  $P_1$  are smaller than all elements in  $P_2$  and all elements in  $P_2$  are smaller than all elements in  $P_3$  and so on. Furthermore, for each processor, its list will also be sorted and so the full list of  $N$  elements will be sorted. This algorithm will be partitioned into four steps. The first step is to have all the processors sort their own lists. The second step will be to find the  $(\frac{N}{P})^{\text{th}}$ ,  $(\frac{2N}{P})^{\text{th}}$ , ....., and the  $(\frac{(P-1)N}{P})^{\text{th}}$  elements over all the processors (using the communication network). We call these elements *delimiters*. The third step is to transfer all the elements smaller than the  $(\frac{N}{P})^{\text{th}}$  element to processor 1 and all the elements which are between the  $(\frac{N}{P})^{\text{th}}$  element and the  $(\frac{2N}{P})^{\text{th}}$  element to processor 2 and so on. The last step is to have all processors sort their own lists.

Example 5.1:



Let us use an example to illustrate the operation of this algorithm. Suppose we have four processors with input data as shown in Figure 5.1(a). After step 1, we have the system as shown in Figure 5.1(b). In step 2, since we have 80 elements with 4 processors, we must find the 20<sup>th</sup>, the 40<sup>th</sup>, and the 60<sup>th</sup> elements. We explain the details of step 2 after this example and we assume we have found these three elements (which are 109, 205, and 282). In step 3, we require that all elements less than or equal to 109 be sent to processor 1. Similarly, all elements between 109 and 205 (including 205) must be sent to processor 2, all elements between 205 and 282 (including 282) must be sent to processor 3, and all elements greater than 282 must be sent to processor 4. If processor 1 starts sending its data to the other processors, followed by 2, 3 and 4, then after step 3 the system will be as shown in Figure 5.1(c). After step 4, we have the final result as shown in Figure 5.1(d) and this algorithm is completed.

Now we explain in detail how to implement step 2 as shown in Figure 5.1(e), which contains the major idea of this algorithm. The idea behind step 2 is a combination of binary searching and counting. We first explain how counting is used in this algorithm. If a processor, say  $P_1$ , wants to find the ranking of one of its elements with value  $M$ , it will broadcast  $M$  to all the other processors. Every processor (except  $P_1$ ) will then calculate how many elements in its list are smaller than  $M$  and will then broadcast this number to  $P_1$ .  $P_1$  can add up all these numbers from these messages to determine the overall ranking of  $M$ . Now we explain how binary searching is used in step 2. Suppose  $P_1$  wants to find out whether its list contains the  $m^{\text{th}}$  element and the value of that element if it is in  $P_1$ 's list. In our algorithm,  $P_1$  will first find out the ranking of its median element using the method just described. If the ranking of that element is higher than  $m$ , then we know the  $m^{\text{th}}$  element will either be in the first half of  $P_1$ 's list or not in  $P_1$ 's list. Therefore, we then take the first half of  $P_1$ 's list as the working list and find the overall ranking of the median element in this working list. The same process will be repeated until we either find the  $m^{\text{th}}$  element or we are sure that the  $m^{\text{th}}$  element is not in  $P_1$ 's list.

We now show how to find the 20<sup>th</sup> element in Example 5.1. Note that the 20<sup>th</sup> element may be in any of the four processors, hence every processor must try to find it in its list. In step 2,  $P_1$  first broadcasts its median element's value, say  $v_1$  (253), then  $P_2$  broadcasts its median element's value, say  $v_2$  (259) and the number of elements in its list which are smaller than  $v_1$  (9), then  $P_3$  broadcasts its median element's value, say  $v_3$  (187), and the numbers of elements in its list which are smaller than  $v_1$  (14) and  $v_2$  (14), then  $P_4$  broadcasts its median element, say  $v_4$  (124), and the number of elements in its list which are smaller than  $v_1$  (20),  $v_2$  (20), and  $v_3$  (15). After  $P_4$  finishes its broadcast,  $P_1$  has determined the ranking of  $v_1$  (53). If  $v_1$  is the 20<sup>th</sup> element,  $P_1$  will broadcast into the network a special message containing "&", which tells all the other processors that the 20<sup>th</sup> element is found and its value is  $v_1$ . If  $v_1$  is not the 20<sup>th</sup> element,  $P_1$  repeats the searching process applying binary search. After the 20<sup>th</sup> element is found, then all the processors will advance to find the next delimiter, which is the 40<sup>th</sup> element in example 5.1. In this example, the procedure goes on until the 60<sup>th</sup> element is found.

#### ALGORITHM:

Step 1: Each processor sorts its own list using the best sequential sorting algorithm.

Step 2: For  $i = 1$  to  $(P-1)$  do

find the  $\left[ i \cdot \frac{N}{P} \right]^{\text{th}}$  element using the broadcast network

226	341	13	101
19	69	199	31
61	274	47	43
286	122	115	196
109	181	130	64
329	304	292	67
208	232	151	79
16	247	153	22
347	250	175	107
349	259	187	124
269	91	28	133
282	289	205	228
85	211	235	160
311	313	301	169
316	326	262	184
322	331	265	191
202	337	271	58
343	39	147	217
244	352	297	223
253	361	241	145

P1                      P2                      P3                      P4

Figure 5.1(a)  
An Example

16	39	13	22
19	69	28	31
61	91	47	43
85	122	115	58
109	181	130	64
202	211	147	67
208	232	151	79
226	247	153	101
244	250	175	107
253	259	187	124
269	274	199	133
282	289	205	145
286	304	235	160
311	313	241	169
316	326	262	184
322	331	265	191
329	337	271	196
343	341	292	217
347	352	297	223
349	361	301	228

P1
P2
P3
P4

**Figure 5.1(b)**  
**After Step 1 from Figure 5.1(a)**

16	122	235	286
19	181	241	311
61	202	262	316
85	115	265	322
109	130	271	329
39	147	208	343
69	151	226	347
91	153	244	349
13	175	253	289
28	187	269	304
47	199	282	313
22	205	211	326
31	124	232	331
43	133	247	337
58	145	250	341
64	160	259	352
67	169	274	361
79	184	217	292
101	191	223	297
107	196	228	301

P1
P2
P3
P4

**Figure 5.1(c)**  
**After Step 3 from Figure 5.1(a)**

13	115	208	286
16	122	211	289
19	124	217	292
22	130	223	297
28	133	226	301
31	145	228	304
39	147	232	311
43	151	235	313
47	153	241	316
58	160	244	322
61	169	247	326
64	175	250	329
67	181	253	331
69	184	259	337
79	187	262	341
85	191	265	343
91	196	269	347
101	199	271	349
107	202	274	352
109	205	282	361
P1	P2	P3	P4

**Figure 5.1(d)**  
**After Step 4 from Figure 5.1(a)**

1	253	...	...	...
2	9	259	...	...
3	14	14	187	...
4	20	20	15	124
1	109	10	5	5
2	3	181	5	6
3	3	9	130	4
4	9	14	10	64
1	&	5	5	3
1	253	...	...	...
2	9	259	...	...
3	14	14	187	...
4	20	20	15	124
1	109	10	5	5
2	3	181	5	6
3	3	9	262	4
4	9	9	20	184
1	208	5	10	5
2	5	232	10	5
3	12	12	205	9
4	17	20	17	217
1	202	8	6	7
2	5	211	5	6

3	11	12	&	12
1	253	...	...	...
2	9	259	...	...
3	14	14	187	...
4	20	20	15	124
1	316	10	5	5
2	14	326	5	4
3	20	20	262	4
4	20	20	20	184
1	282	16	10	5
2	11	289	10	5
3	17	17	292	9
4	20	20	20	217
1	&	13	13	7

Figure 5.1(e)  
Step 2 of Figure 5.1(a)

Step 3: For  $i = 1$  to  $P$  do  
           for  $j = 1$  to  $P$  do

$P_i$  sends the data between  $\left[ (j-1) \cdot \frac{N}{P} \right]^{th}$  and  $\left[ j \cdot \frac{N}{P} \right]^{th}$  to  $P_j$ .

Step 4: Each processor sorts its list using merge sort.

Complexity Analysis:

We denote  $\log_2$  to be  $lg$ , then the optimum computational complexity for step 1 is  $O\left(\frac{N}{P} lg\left(\frac{N}{P}\right)\right)$  for each processor (using, for example, quick sort). The communication complexity for step 2 is  $O\left(P^3 lg\left(\frac{N}{P}\right)\right)$ . This complexity is calculated as follows. We define a "run" to be that every processor broadcasts once; hence, there are  $P$  messages per run. In order to find a delimiter, the number of runs is upper bounded by  $lg\left(\frac{N}{P}\right)$  which is the complexity for binary searching. Therefore, the number of messages is upper bounded by  $P lg\left(\frac{N}{P}\right)$ ; hence, the number of "elements" is upper bounded by  $P^2 lg\left(\frac{N}{P}\right)$  since there are  $P$  elements per message. Since we have to find  $(P-1)$  delimiters, the overall communication complexity is upper bounded by  $P^3 lg\left(\frac{N}{P}\right)$ . The communication complexity for step 3 is simply  $O(N)$ . The computational complexity for step 4 is  $O\left(\frac{N}{P} lg P\right)$  for each processor since each processor will simply merge its remaining list with  $P - 1$  sub-lists received from other processors and the complexity can be shown to be  $\frac{N}{P} lg P$ .

One advantage of this algorithm, as we mentioned earlier, is that in step 2 we can send  $P$  elements per message and we can send as many elements in step 3 per message as there are elements in the sublist. Also note that all communication is scheduled and so no collisions will occur on the multiaccess broadcast channel. However, there are some inefficiencies in step 2, which will be improved in the next section.

### 5.3 Improvement of the Algorithm

In the previous algorithm, we repeated the same procedure whenever we tried to find the element with a specified ranking. We did not make use of the information we had acquired from previous searches. Here, we improve the algorithm so that we can make use of that information to cut down the effort of searching in step 2.

Let us use the previous example to convey this idea. After we have found the 20<sup>th</sup> element, every processor knows that the next element to search for is the 40<sup>th</sup> element. However, at that point,  $P_3$  already knows that 187 is the 35<sup>th</sup> element and  $P_1$  already knows that 253 is the 53<sup>rd</sup> element. If  $P_1$  and  $P_3$  broadcast this information to all the processors, then every processor would search for the 40<sup>th</sup> element between 187 and 253 and they do not have to search the entire list to find the 40<sup>th</sup> element. Although we will incur some memory overhead by this change to memorize this extra information, it is easy to show that the memory overhead is  $2P$  for each processor, which is small compared to  $N$ . By this change, we save

communication.

### Example 5.2

We will redo example 5.1 using this modification. Since only step 2 is modified, we will only re-do step 2 in this example as shown in Figure 5.2. Note that after the first delimiter 109 is found,  $P_1$  has learned that 253 is the 53<sup>rd</sup> element,  $P_2$  has learned that 259 is the 54<sup>th</sup> element,  $P_3$  has learned that 187 is the 35<sup>th</sup> element, and  $P_4$  has learned that 124 is the 25<sup>th</sup> element. Therefore, after the first delimiter is found,  $P_1$  will broadcast a message telling all the processors that the 40<sup>th</sup> element should have the value between 109 and 253. Similarly,  $P_2$  will broadcast 109 and 259,  $P_3$  will broadcast 187 and "---" (which means  $\infty$ ), and  $P_4$  will broadcast 124 and "---" ( $\infty$ ). After these four messages are received by all processors, every processor will take the maximum value of the first element from each message and the minimum value of the second element from each message as the range for finding the 40<sup>th</sup> element. The binary searching will be applied only in this range. Figure 5.2 explains the rest of this example. (In Figure 5.2 and 5.4, a processor broadcasts a message containing "&&" means that this processor has nothing left in its list. Note that this processor will not broadcast any more message after it broadcasts a "&&" message.)

### 5.4 Modification of the Algorithm: A Parameterized Algorithm

In this section, we make a slight change to the previous algorithm to reduce the communication complexity dramatically with a slight sacrifice in computation. Since communication is presently much more expensive than computation, this modification of the algorithm will give us a great benefit. The most significant effect of this modification is that the communication complexity for step 2 is no longer a function of  $N$ . This makes our algorithm a very good algorithm for large  $N$ .

The idea of this modification is not to require to find the  $(\frac{N}{P})^{\text{th}}$  exactly as the first delimiter but rather only to within  $\alpha\%$  ( $0 \leq \alpha \leq 100$ ) of  $\frac{N}{P}$ . That is, the first delimiter can be any element whose ranking is between  $\left[ \frac{N}{P} - \frac{N}{P} \cdot \frac{\alpha}{100} \right]$  and  $\left[ \frac{N}{P} + \frac{N}{P} \cdot \frac{\alpha}{100} \right]$ . Similarly, the second delimiter can be any element whose ranking is between  $\left[ \frac{2N}{P} - \frac{N}{P} \cdot \frac{\alpha}{100} \right]$  and  $\left[ \frac{2N}{P} + \frac{N}{P} \cdot \frac{\alpha}{100} \right]$ . This modification applies to all delimiters. By doing this, it can be shown that the number of runs is upper bounded by  $\lg \frac{100}{\alpha}$ ; hence the communication complexity for step 2 after this modification is  $O(P^3 \lg \frac{100}{\alpha})$ . Note that this complexity is independent of  $N$ . Also note that this communication complexity can be much better than  $P^3 \lg \frac{N}{P}$  if  $N \gg P$ , which is obtained in section 5.2. However, by this saving in communication, we also incur more computation in step 4. Fortunately, the computation complexity for step 4 will be at most  $2\alpha\%$  more than the original algorithm. Note that this increase in computation is incurred in step 4 only, whose computational complexity is dominated by the computational complexity in step 1; hence this increase in computation has only a minor effect.



1	253	...	...	...
2	9	259	...	...
3	14	14	187	...
4	20	20	15	124
1	109	10	5	5
2	3	181	5	6
3	3	9	130	4
4	9	14	10	64
1	&	5	5	3
1	109	253	/	/
2	109	259	/	/
3	187	...	/	/
4	124	...	/	/
2	...	232	4	1
3	...	12	205	3
4	...	20	17	217
1	208	8	6	7
2	5	211	5	6
3	12	12	&	12
1	253	...	/	/
2	259	...	/	/
3	205	...	/	/
4	205	...	/	/
4	17	17	...	&&
1	316	7	...	/
2	14	326	...	/
3	20	20	271	/
1	282	16	11	/
2	11	289	10	/
3	17	17	297	/
1	&	...	...	/
	P1	P2	P3	P4

Figure 5.2  
Step 2 of Figure 5.1(a)

**Example 5.3:**

Let us redo example 5.1 by choosing  $\alpha = 10$ . Since steps 1, 3, and 4 are not affected by this modification, we will only show the procedure for step 2. For  $\alpha$  equals 10, we know that the first delimiter can be any one of the 18<sup>th</sup>, 19<sup>th</sup>, 20<sup>th</sup>, 21<sup>st</sup>, and 22<sup>nd</sup> elements (i.e.,  $20 \pm 2$ ). Similarly, the second delimiter can be any one of the 38<sup>th</sup>, 39<sup>th</sup>, 40<sup>th</sup>, 41<sup>st</sup>, and 42<sup>nd</sup> elements, and the third delimiter can be any one of the 58<sup>th</sup>, 59<sup>th</sup>, 60<sup>th</sup>, 61<sup>st</sup>, and 62<sup>nd</sup> elements. Figure 5.3 shows the broadcast messages for step 2.

Another advantage of this modification is that by adjusting the value of  $\alpha$ , we are able to trade between communication complexity and computational complexity. A larger  $\alpha$  will incur less communication and more computation than a smaller  $\alpha$ .

1	253	...	...	...
2	9	259	...	...
3	14	14	187	...
4	20	20	15	124
1	109	10	5	5
2	3	181	5	6
3	3	9	130	4
4	9	14	10	64
1	&	5	5	3
1	109	253	/	/
2	109	259	/	/
3	187	...	/	/
4	124	...	/	/
2	...	232	4	1
3	...	12	205	3
4	...	20	17	217
1	208	8	6	7
2	5	211	5	6
3	12	12	&	12
1	253	...	/	/
2	259	...	/	/
3	205	...	/	/
4	205	...	/	/
4	17	17	...	&&
1	316	7	...	/
2	14	326	...	/
3	20	20	271	/
1	282	16	11	/
2	11	289	10	/
3	17	17	&	/
	P1	P2	P3	P4

Figure 5.3  
Step 2 of Figure 5.1(a)

## CHAPTER 6 WHY DISTRIBUTED OR PARALLEL SYSTEMS?

In this chapter we explore why more and more distributed and parallel processing systems are being built despite the well known queueing result that a centralized system has better performance than a distributed system. In queueing theory we have the following result (see [KLEI76]):

$$T(1, \lambda, C) < T(m, \lambda, C) \quad (6.1)$$

This says that given the same arrival rate ( $\lambda$ ) and the same aggregate capacity ( $C$ ) for two systems, one with only one *big* processor and the other one with  $m$  *small* processors, the mean response time of the system with only one *big* processor (denoted as  $T(1, \lambda, C)$ ) is smaller than the mean response time of the system with  $m$  *small* processors (denoted as  $T(m, \lambda, C)$ ). This result suggests that we should use centralized computing systems rather than distributed computing systems.

However, as the technology improves, more and more distributed and parallel processing systems are being introduced in the real world. The Connection machine [HILL85], which uses 65,536 *weak* processors in one system, is one of the most famous distributed system. Here, *weak* processors means processors which are not powerful. A distributed computing system described in [MUTK87] and the hypercube machines are all examples of this trend. Since there are so many distributed and parallel processing systems built, there must be a reason to support it.

With these two facts stated above, we inquire us to why people build distributed or parallel processing systems. In this chapter, we focus on one issue, which is the cost issue. In other words, if there is a diseconomy of scale in computing, i.e., the average cost per computing capacity (MIPS, million instructions per second) for a less powerful processor is lower (per MIPS) than that for a more powerful processor, then we can buy more capacity for a distributed (or, parallel) system than a centralized system using the same budget. Equation (6.1) does not fit in this situation since the aggregate capacities for the centralized system and the distributed system may well be different. In this case, a natural question arises which is to find the optimal capacity (i.e., computing power) of each processor and the optimal number of such processors to be used in a system given the cost-capacity function such that the mean response time is minimized. In this chapter, we distinguish between distributed computing systems and parallel processing systems by the following. Let us assume there are  $P$  processors in the system. For a distributed system, there are  $P$  queues in the system, one for each processor. When a job comes to the system, it can join any of the queues with equal probability  $\frac{1}{P}$ . Therefore, a job can have only one processor to process its work. For a parallel processing system, there is only one queue for the system and every job that comes to the system will have to join this queue. However, when a job is admitted into service, it will occupy all  $P$  processors to process its work concurrently. By this definition, it is clear that the service time (not the mean

response time) of the parallel processing system will be smaller than the distributed system but the waiting time of the parallel processing system may be greater than the distributed system.

For a distributed system, by using more processors with less capacity per processor, we decrease the queuing time of a job by suffering an increased service time. On the contrary, by using fewer processors with higher capacity per processor, we decrease the service time by suffering a longer queuing time. Given the cost-capacity function, we would like to find the optimal capacity of a processor and the optimal number of processors to use such that the mean response time is minimized. We also study the impact of the distribution of the service time on this issue.

For a parallel processing system, although a job can use all the processors, it is not guaranteed that the service time of the job will be  $P$  ( $P$  is the number of processors in the system) times smaller than the service time when the job uses only one processor. This can be achieved only when the speedup of the job in this system is exactly  $P$ , which is usually not the case. Therefore, if we assume a diseconomy of scale in computing and we do not take the speedup effect into consideration, we would like to use as many small processors as possible since by doing so we can get the most capacity for the system and hence the smallest service time. However, if the speedup of the system is considered, choosing many small processors may hurt the performance if the speedup is not linear. Hence, by considering both the cost-capacity effect and the speedup effect, we are able to find the optimal capacity of a processor and the optimal number of processors to use in order to minimize the mean response time.

## 6.1 Previous Work

In [KLEI84] a case is considered by adding more processors to the system, but in a fashion which maintains a constant total system capacity. The particular structure considered is the regular series-parallel structure as shown in Figure 6.1 where a total processing capacity of  $C$  MIPS are divided equally into  $mn$  processors, each behaving as an  $M/M/1$  queue, and each of  $\frac{C}{mn}$  MIPS. On entering the system, a job selects (equally likely) any one of the  $m$  series branches down which it will travel. It will receive  $\frac{1}{n}$  of its total processing needs at each of the  $n$  series-connected processors.

The key result obtained in [KLEI84] for this system is that the mean response time for a job in this series-parallel pipeline system is  $mn$  times as large as it would have been had the jobs been processed by a single processor of  $C$  MIPS! This result clearly suggests that multiprocessing processing of this kind is terrible. Then, the question is why multiprocessing systems attract so much attention and interest. In [KLEI84], it was pointed out that the answer may be that a large number of small processors (e.g., microprocessors) with an aggregate capacity of  $C$  MIPS is less expensive than a large uniprocessor of the same total capacity. [EIND85] studies this issue by actually looking at existing systems to draw the conclusion that there is a diseconomy of scale in computing over all systems (from micro to super computers). Figure 6.2 shows the result of his study. Figure 6.2 says that within each family of computers (e.g., micro computers or minicomputers, etc.), there is an economy of scale. However, comparing all the computers over all families, there is a diseconomy of scale in computing.

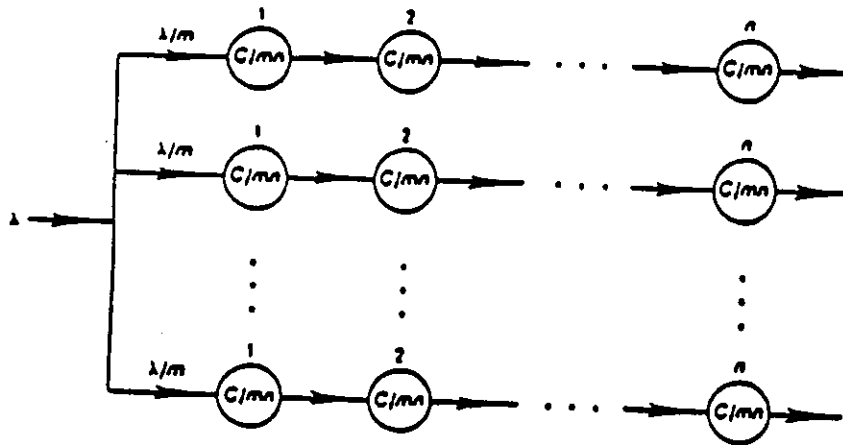


Figure 6.1  
A Symmetrical Distributed-Processing Network

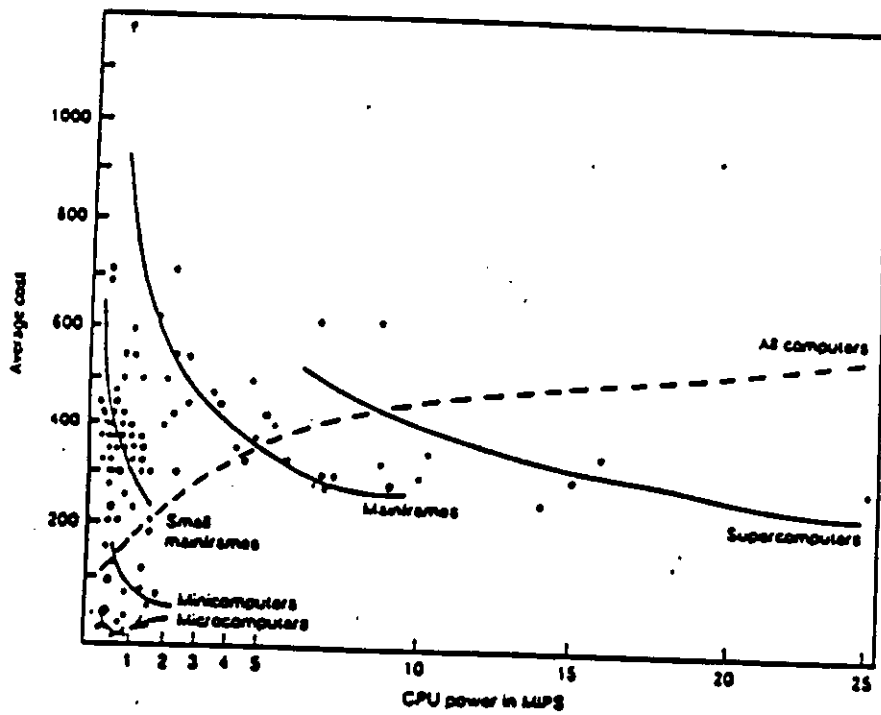


Figure 6.2  
Economics of Computer Power

## 6.2 Distributed Systems

In this section we first examine the effect of the cost-capacity function on the mean response time in the distributed system. Then, assuming there is diseconomy of scale in computing, we revisit the definition of power as used in the previous chapters. Two new definitions of power will be given and by using these new definitions, we find the optimal values of some design parameters such that power is maximized. These parameters are the optimal budget to spend, the optimal job arrival rate, the optimal capacity of a processor, and the optimal number of processors to use in the system. We also study the effect of the variance of the service time on this issue.

### 6.2.1 The Effect of Cost-Capacity Function

We assume the cost of a processor is a monotonically increasing convex function of the capacity of the processor (i.e., the average cost per CPU power in MIPS is monotonically increasing). Given a fixed budget to build a distributed system, intuitively we would choose to buy as many *small* processors as we can since we can get the most capacity out of it. With higher capacity, we can accept heavier traffic into the system without running into saturation. However, the disadvantage of using a smaller processor (with less capacity) is that it incurs a longer service time. Therefore, even if we have many small processors such that the aggregate capacity is so high that there is almost no queuing time, the longer service time will still result in a larger mean response time. Also, the utilization of the overall capacity would be lower since the arrival rate is fixed while the total capacity grows.

We define the following notation:

- $W \triangleq$  the average work load for each job
- $C \triangleq$  the capacity for each processor
- $d \triangleq$  the cost (in dollars) for each processor
- $f(C) \triangleq$  the cost for one processor with capacity  $C$  ( $d = f(C)$ )
- $n \triangleq$  the number of processors used
- $D \triangleq$  the budget to build the system
- $\bar{x} \triangleq$  the average service time for jobs
- $T(C) \triangleq$  the mean response time given each processor has a capacity  $C$
- $\lambda \triangleq$  the Poisson arrival rate
- $\rho \triangleq$  the utilization of the system
- $\bar{N} \triangleq$  the average number of jobs per processor

We assume the arrival rate ( $\lambda$ ) and the average workload for the jobs are given. Also given is the total budget and the cost-capacity function. The goal in this section is to find the optimal capacity of a processor and the optimal number of processors ( $n^*$ ) to use in a system under the given conditions in order to minimize the mean response time. In summary, the objective of this section is:

Given:  $\lambda$  and  $f(C)$  and assuming all processors have the same capacity

Find:  $n^*$  and  $C^*$  to minimize  $T$

$$\text{Constraint: Fixed budget } D \leq \sum_{i=1}^n d(C_i) = n \cdot d$$

We assume the cost function  $f(C)$  of a processor with a capacity  $C$  is a monotonically increasing convex function.

$$d = f(C) \quad (6.2)$$

From (6.2) we have

$$n = \frac{D}{d} = \frac{D}{f(C)} \quad (6.3)$$

$$\bar{x} = \frac{W}{C} \quad (6.4)$$

$$\rho = \frac{\lambda \bar{x}}{n} = \frac{\lambda W}{D} \cdot \frac{f(C)}{C} \quad (6.5)$$

Hence,

$$T(C) = \frac{\bar{x}}{1 - \rho} = \frac{W}{C - \frac{\lambda W}{D} f(C)} \quad (6.6)$$

or

$$T(n) = \frac{W}{f^{-1}\left(\frac{D}{n}\right) - \frac{\lambda W}{n}}$$

Minimizing  $T(C)$  with respect to  $C$ , we have

$$\frac{d}{dC} f(C^*) = \frac{D}{\lambda W} \quad (6.7)$$

Equation (6.7) is the equation that  $C^*$  must satisfy in order to minimize the mean response time. Moreover, notice that for a given budget  $D$ , the most capacity we can get for a single processor is  $f^{-1}(D)$ , which can easily be derived from (6.2). Therefore, if the  $C^*$  derived from (6.7) is higher than  $f^{-1}(D)$ , then  $C^*$  should be equal to  $f^{-1}(D)$ .

Also notice that since  $f(C)$  is an increasing convex function, then  $\frac{d}{dC} f(C)$  is a monotonically increasing function. Therefore, if there exists any solution for (6.7), the solution must be *unique*. If there is no solution for (6.7), then the optimal  $C^*$  must be at the boundary, i.e., the optimal  $C^*$  would be either  $f^{-1}(D)$  or as small as possible.

By finding  $C^*$ , we find  $n^*$  by first finding  $d^*$ .



$$d^* = f(C^*)$$

Hence,

$$n^* = \frac{D}{d^*} = \frac{D}{f(C^*)}$$

Notice that if  $f(C)$  is a linear function, Eq. (6.7) does not apply: in that case we have from (6.6) that

$$T(C) = \frac{W}{C(1 - \frac{k\lambda W}{d})} \quad (6.8)$$

where  $f(C) = kC$ . From Eq. (6.8) we observe that in order to minimize  $T$ , we would like to choose  $C$  to be as high as possible, i.e., we would spend all the budget to buy one big processor. This is equivalent to saying that the centralized system performs better than the distributed system, which is the same result as described in (6.1).

### 6.2.1.1 Polynomial Cost Function

In this section we assume the cost function to be a polynomial function of the form  $C^a$ : (6.9)

$$d = C^a \quad \text{where } a > 1$$

Substituting (6.9) into (6.7) we have

$$C^* = \min \left[ \left( \frac{D}{\lambda a W} \right)^{\frac{1}{a-1}}, D^{1/a} \right]$$

$$n^* = \max \left[ \frac{(\lambda a W)^{\frac{a}{a-1}}}{D^{\frac{1}{a-1}}}, 1 \right]$$

$$\rho^* = \min \left[ \frac{1}{a}, \frac{\lambda W}{D^{1/a}} \right]$$

Let us define  $L = \frac{a}{\lambda W}$ , we have

$$C^* = \min \left[ \left( \frac{D}{aL} \right)^{\frac{1}{a-1}}, D^{1/a} \right]$$

$$n^* = \max \left[ \left[ \frac{(aL)^a}{D} \right]^{\frac{1}{a-1}}, 1 \right]$$

$$\text{Total capacity} = n^* C^* = aL \quad \text{if } n^* > 1$$

Figure 6.3 shows the time versus  $n$  curve for an example with  $W = 10$ ,  $\lambda = 2$ ,  $D = 40$ , and  $a = 1.4$ .

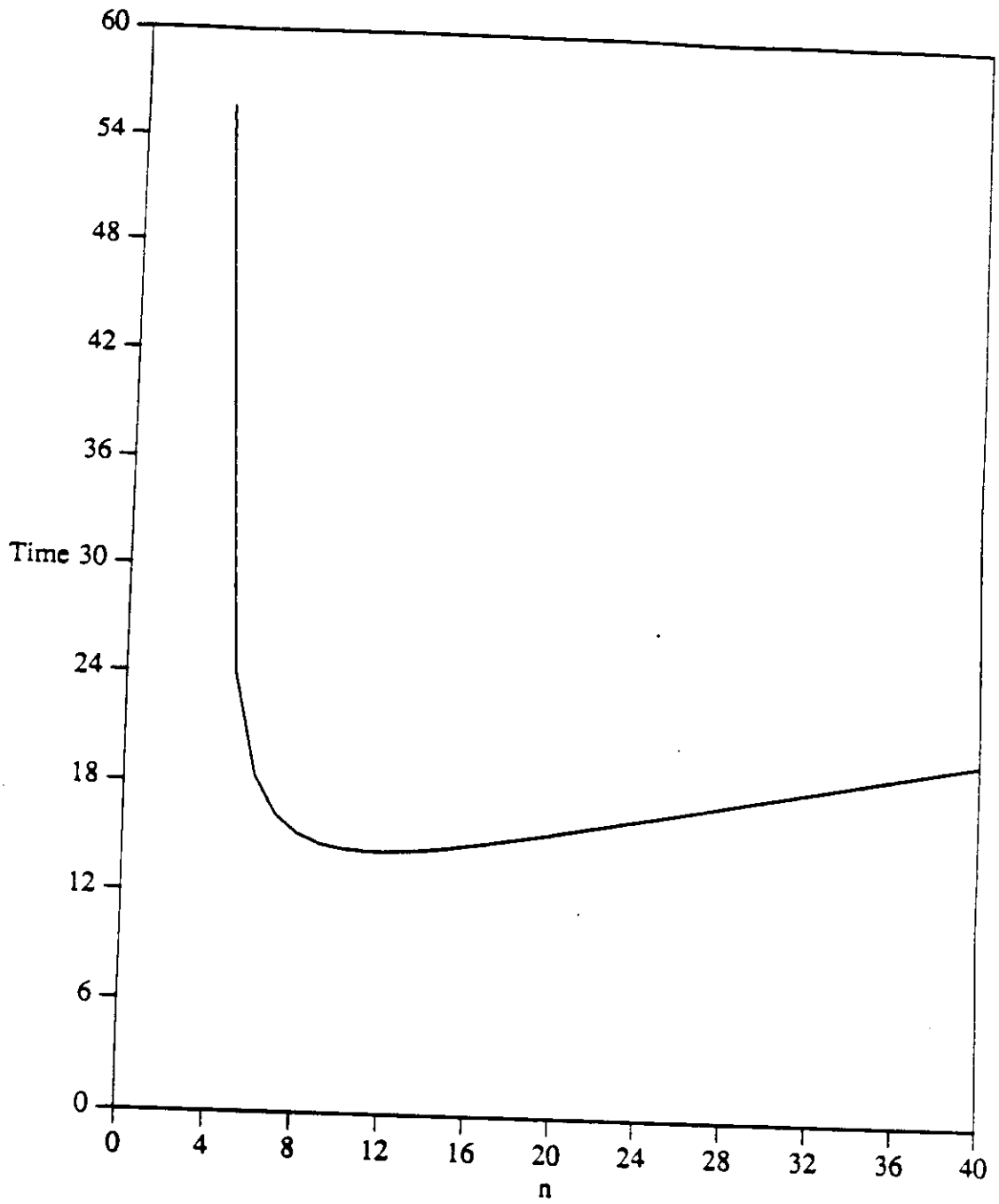


Figure 6.3  
 Time versus n ( $\lambda = 2, W=10, D=40, a=1.4$ )

### 6.2.1.2 Exponential Cost Function

In this section we assume the cost function to be an exponential function,

$$d = f(C) = e^{aC} \quad (6.10)$$

Substituting (6.10) into (6.7) we have

$$C^* = \min \left[ \frac{1}{a} \ln \left( \frac{D}{a\lambda W} \right), \frac{1}{a} \ln D \right] = \min \left[ \frac{1}{a} \ln \left( \frac{D}{aL} \right), \frac{1}{a} \ln D \right]$$

$$n^* = \max [a\lambda W, 1] = \max [aL, 1]$$

Figure 6.4 shows the time versus  $n$  curve for an example with  $W = 10$ ,  $\lambda = 1$ ,  $D = 10,000$ , and  $a = 1$ .

### 6.2.2 When to Use Distributed Systems?

In this section we discuss the conditions when a distributed system performs better than a centralized system. We will be comparing a centralized system with one big processor, with a distributed system, with  $m$  processors each with  $\frac{1}{n}$  of the capacity of the big processor, as shown in Figure 6.5. We wish to find out how many small processors (i.e., the value of  $m$ ) we need to achieve the same mean response time. We assume the workload for each job is an exponentially distributed random variable  $\hat{W}$  with mean  $W$ . We define the following notation:

- $W \triangleq$  the average work load for each job
- $m \triangleq$  total number of small processors in the distributed system
- $C \triangleq$  the capacity for that BIG processor in the centralized system
- $C_m \triangleq$  the capacity for each of the small processors in the distributed system
- $n \triangleq$  the ratio between  $C$  and  $C_m$
- $\bar{x}_1 \triangleq$  the mean service time for the centralized system
- $\bar{x}_m \triangleq$  the mean service time for the distributed system
- $\rho_1 \triangleq$  the utilization of the centralized system
- $\rho_m \triangleq$  the utilization of the distributed system
- $T_1 \triangleq$  the mean response time for the centralized system
- $T_m \triangleq$  the mean response time for the distributed system

From the definitions, we have

$$C_m = \frac{C}{n}$$

and

$$\bar{x}_1 = \frac{W}{C}$$

and

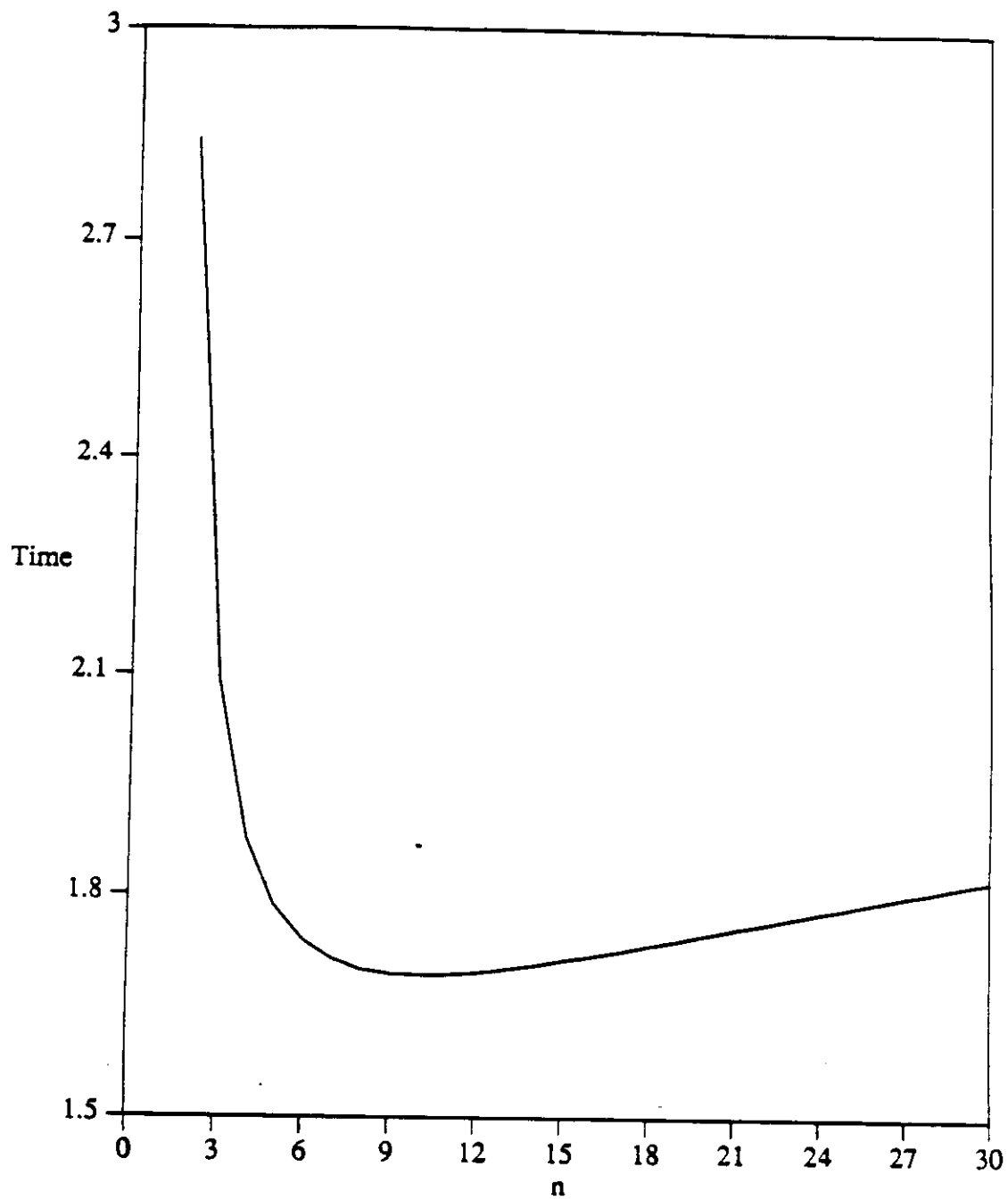


Figure 6.4  
Time versus  $n$  ( $\lambda = 1$ ,  $W=10$ ,  $D=10000$ ,  $a=1$ )

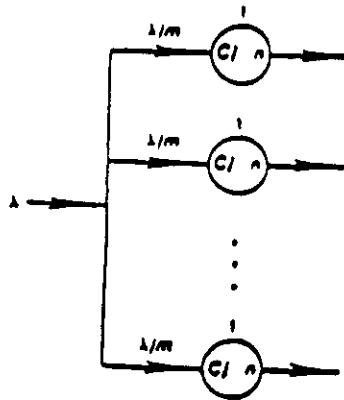


Figure 6.5  
A Distributed System

$$\rho_1 = \lambda \bar{x}_1 = \frac{\lambda W}{C}$$

Using results from M/M/1 [KLEI75] we have

$$T_1 = \frac{\bar{x}_1}{1 - \rho_1} = \frac{W}{C - \lambda W}$$

For system two, we have

$$\bar{x}_m = \frac{W}{C_m} = \frac{nW}{C}$$

and

$$\rho_m = \frac{\lambda \bar{x}_m}{m} = \frac{\lambda n W}{m C}$$

Using results from M/M/1 [KLEI75] we have

$$T_m = \frac{\bar{x}_m}{1 - \rho_m} = \frac{m n W}{m C - \lambda n W}$$

Setting  $T_1 = T_m$  we have

$$m = \frac{n \rho_1}{n \rho_1 + 1 - n} \tag{6.11}$$

The condition for  $m$  to exist is

$$1 > \rho_1 > \frac{n-1}{n} \tag{6.12}$$

Notice that when  $\rho_1 \leq \frac{n-1}{n}$ , the centralized system will always outperform the distributed system. Therefore, load is an important factor in deciding whether to use a distributed system or not. Figure 6.6 shows  $m$  versus  $\rho_1$  with  $n = 5$ .

**Corollary 6.1:**  $m > n$  if (6.12) is met.

[Proof]

From (6.11) we have

$$m = n \cdot \frac{\rho_1}{n\rho_1 + 1 - n}$$

All we have to do is to prove

$$\frac{\rho_1}{n\rho_1 + 1 - n} > 1$$

or,

$$\rho_1 - n\rho_1 - 1 + n > 0$$

the left hand side equals  $(n-1)(1-\rho_1)$ , which is always positive since  $1 > \rho_1$ . It is thus proved.

Q.E.D.

If  $\hat{W}$  has a general distribution with coefficient of variation  $c$ , we have the following theorem.

**THEOREM 6.1:**

For an M/G/1 system,

$$m = \frac{(2-n+n\rho_1-\rho_1) + (n-n\rho_1+\rho_1)c^2}{(2-n+n\rho_1-\rho_1) - n(1-\rho_1) + \rho_1 c^2} \cdot n\rho_1 \quad (6.13)$$

and the condition for  $m$  to exist is

$$1 > \rho_1 > \frac{2(n-1)}{2n-1+c^2} \quad (6.14)$$

From equation (6.14) we conclude that if the service time distribution has a higher variance, then the distributed system will become attractive at a lower load.

**Corollary 6.2:** If a job has a higher variance for its service time distribution,  $m$  (as calculated in equation (6.13)) will be lower.

Proof:

From equation (6.13) we have

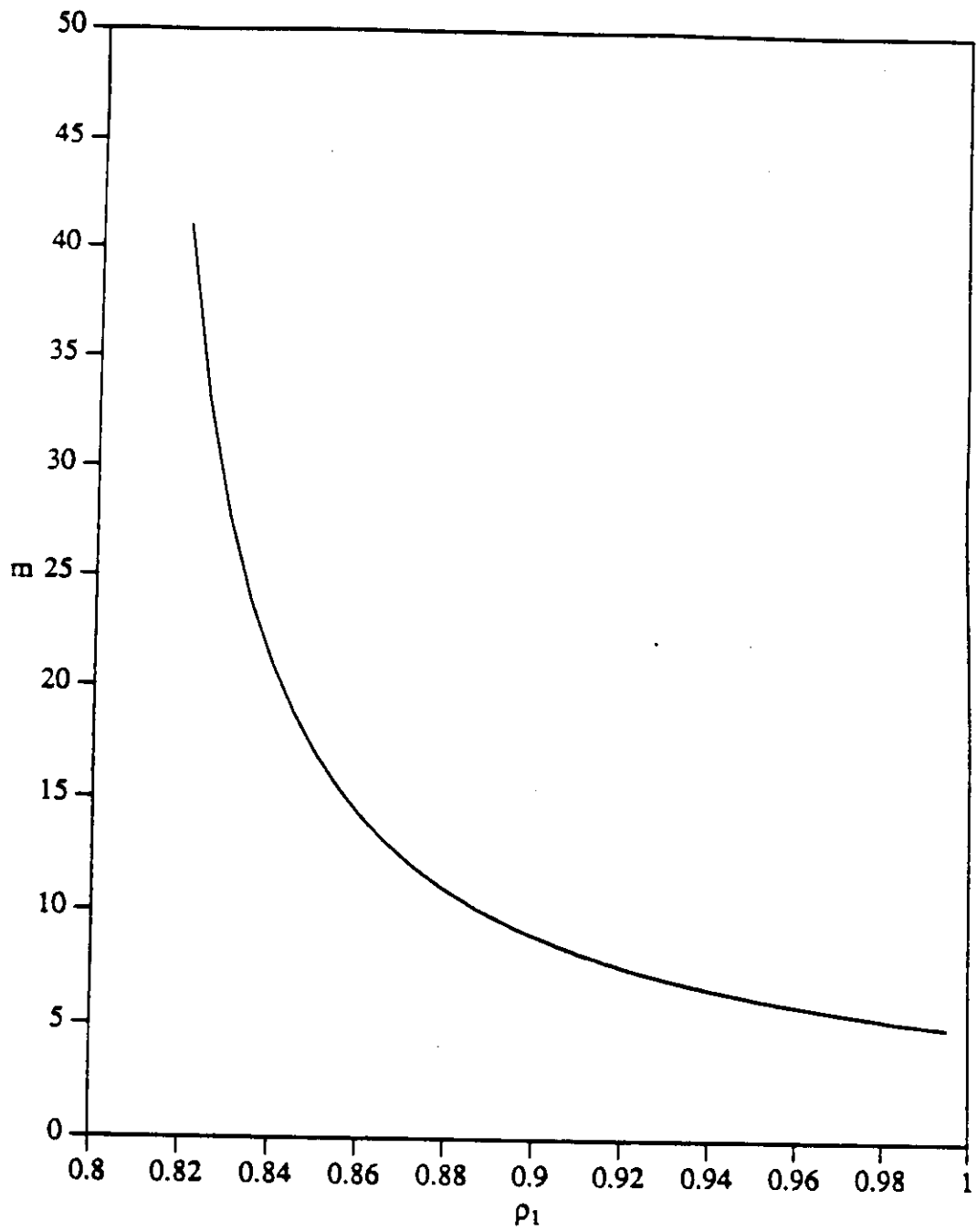


Figure 6.6  
 $m$  versus  $\rho_1$  ( $n=5$ )

$$m = \frac{\frac{2 - n + n\rho_1 - \rho_1}{n - n\rho_1 + \rho_1} + c^2}{\frac{2 - 2n + 2n\rho_1 - \rho_1}{\rho_1} + c^2} \cdot n(n - n\rho_1 + \rho_1)$$

We can prove this lemma by proving

$$\frac{2 - 2n + 2n\rho_1 - \rho_1}{\rho_1} < \frac{2 - n + n\rho_1 - \rho_1}{n - n\rho_1 + \rho_1}$$

or

$$\frac{2 - n + n\rho_1 - \rho_1}{n - n\rho_1 + \rho_1} - \frac{2 - 2n + 2n\rho_1 - \rho_1}{\rho_1} > 0 \quad (6.15)$$

The left hand side of equation (6.15) equals

$$\frac{(n - 1)(1 - \rho_1)^2}{\rho_1(n - n\rho_1 + \rho_1)}$$

which is always positive, i.e., equation (6.15) is always true.

Q.E.D.

Figure 6.7 shows  $m$  versus  $c^2$  for  $\rho_1 = 0.9$ .

Let us explain the physical intuition of Corollary 6.2. With a higher variance of service time, the probability of having a job which requires a relatively long service time in the system is higher. For a centralized system, whenever there is a big job in service, it blocks jobs which come after it from service. Hence these jobs incur a long queueing time even if they are very small. Therefore, for situations like this, a distributed system will become more attractive since the long jobs will not block the entire system. This lemma says that for jobs with higher variance in service time distribution, the number of processors required for the distributed system to have the same mean response time of the centralized system is smaller. This implies that for a service time with higher variance, the cost required to set up the distributed system will be lower; hence, a higher the motivation for using distributed systems.

With the value of  $m$  found from THEOREM 6.1, we can calculate the overall cost for the distributed system as long as we have the cost of the processors. If the total cost for the distributed system is lower than the cost for the centralized system, then we should choose to use the distributed system. Otherwise, the centralized system prevails.

### 6.2.3 Optimization Issues Using Power

In section 6.2.1 we found  $n^*$  in order to minimize the mean response time given the values of all other parameters. In this section, we leave one more parameter open to be decided in order to optimize the performance measure we choose to use. Two cases will be discussed in this section. In case one, we have to decide the optimal budget ( $D^*$ ) to spend, the optimal capacity ( $C^*$ ) of each processor, and the optimal number of processors to be used ( $n^*$ ) in the system. In case two, we have to decide the optimal operating



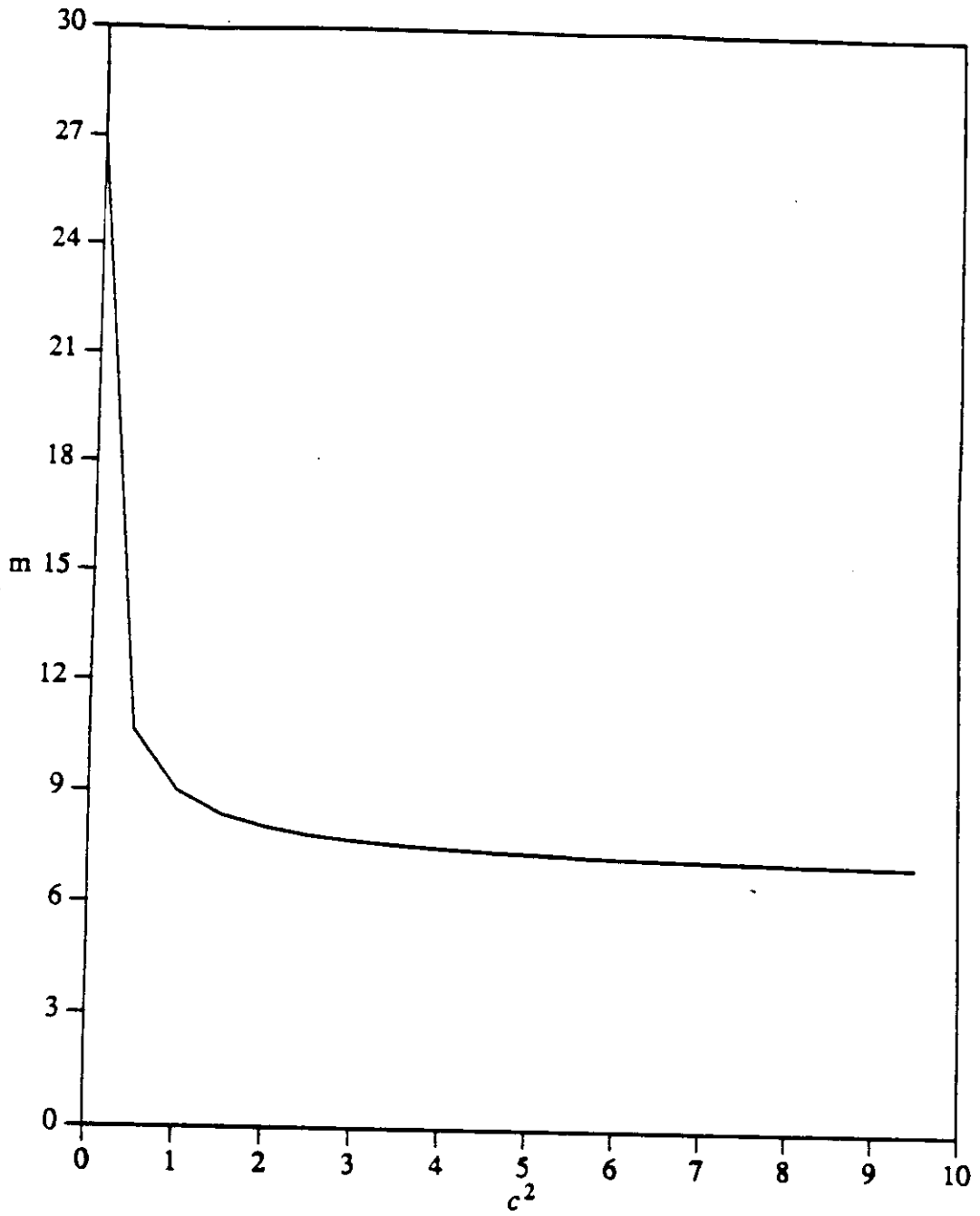


Figure 6.7  
 $m$  versus  $c^2$

point ( $\lambda^*$ ), the optimal capacity ( $C^*$ ) of each processor, and the optimal number of processors to be used ( $n^*$ ) in the system. For both cases, it would be meaningless to choose the objective function to be minimizing the mean response time. Since for case one, we can always spend more money to get lower mean response time. Similarly, for case two we can always lower the arrival rate to lower the mean response time. Clearly, these solutions are not very useful. In this section, we introduce the definition of power again and by maximizing power for both cases, we derive the optimal values we need. We assume the distribution of the service requirement to be exponential.

### 6.2.3.1 Optimize the Budget and the Capacity with a Given Arrival Rate

In this section, we first use power as previously defined. However, the results derived from this definition of power do not seem reasonably intuitively; hence, we give another definition of power and use that to derive the results required to maximize power. Let us first use power as previously defined:

$$R = \frac{\mu}{T} = \frac{\rho}{T} \quad (6.16)$$

From (6.5) and (6.6) we have

$$R = \frac{\lambda f(C)[CD - \lambda W f(C)]}{CD^2} \quad (6.17)$$

Setting  $\frac{\partial R}{\partial D} = 0$  we have

$$D^* = \frac{2\lambda W f(C)}{C} \quad (6.18)$$

Substituting (6.18) into (6.17) we have

$$R_{D^*} = \frac{C}{4W} \quad (6.19)$$

From (6.19) we know in order to maximize  $R_{D^*}$  we would like to choose  $C$  to be as large as possible. That means:

$$n^* = 1$$

hence,

$$C^* = f^{-1}(D^*) \quad (6.20)$$

Substituting (6.20) into (6.18) we have

$$D^* = \frac{2\lambda W D^*}{C^*}$$

which leads to

$$C^* = 2\lambda W$$

Hence,

$$D^* = f(2\lambda W)$$

From  $n = 1$  and  $C^*$  we have

$$\rho^* = \frac{1}{2} \quad (6.21)$$

and

$$\bar{N}^* = 1$$

This result tells us that no matter what the cost-capacity function is, the maximum power can be achieved by spending all the budget to get one big processor. An intuitive explanation follows. Instead of using one processor, we choose to use  $n$  processors (each one is less powerful than the centralized processor) with the total capacity of all these  $n$  processors to be  $m$  times of the centralized (big) processor ( $n > m$ ). Since the workload for both system are the same and the capacity is  $m$  times more for the distributed system, the utilization of the system should become  $\frac{1}{m}$  of the original utilization. Now the question is whether the mean response time of the distributed system operating at a point where power is maximized can be less than  $\frac{1}{m}$  of the mean response time of the centralized system operating at the point where power is maximized. For the system with only one processor, power is maximized when  $T = 2\bar{x}$  which can be easily derived from (6.21). Hence, if  $m \geq 2$ , then the mean response time of the  $n$ -processor system will never be lower than  $\frac{2\bar{x}}{m}$  because the mean response time of the  $n$ -processor system will never be smaller than  $\bar{x}$ ; therefore, power of the  $n$ -processor system will be always smaller than the one-processor system. If  $m < 2$ , the same result can be proved mathematically, but not intuitively.

Intuitively, if the cost per MIPS rises significantly with respect to  $C$ , it is natural to buy more smaller processors to get a much higher total capacity than buying only one bigger processor under the same budget. However, the result of  $n^* = 1$  is contrary to this intuition. This contradiction suggests that defining power as  $\frac{P}{T}$  may not be practically meaningful because  $\rho$  may not be of interest since more total capacity can be bought using the same budget. Therefore, we would like to change the definition of power. In designing a system, it is natural to ask for lower budget and lower mean response time; hence, we define power to be

$$R = \frac{1}{DT} \quad (6.22)$$

Hence we have

$$R = \frac{DC - \lambda W f(C)}{D^2 W} \quad (6.23)$$

Setting  $\frac{\partial R}{\partial D} = 0$  we have

$$D^* = \frac{2\lambda W f(C)}{C} \quad (6.24)$$

Substituting (6.24) into (6.5) we have

$$\rho^* = \frac{1}{2}$$

Therefore, the mean number of jobs in each queue is

$$\bar{N}^* = 1 \quad (6.25)$$

Equation (6.25) shows a very interesting result. It says that no matter how much capacity we assign to each processor, the optimal budget to maximize power as defined in (6.22) is such that the mean number of jobs per processor in the system is one. This is the same result as obtained in [KLEI79] which bears the intuition that the proper operating point for the system is exactly when there is only one job being served by each processor and no others are waiting for service at the same time for every queue.

Substituting (6.24) into (6.23) we have

$$R_{D^*} = \frac{C^2}{4\lambda W^2 f(C)} \quad (6.26)$$

Optimizing  $R_{D^*}$  with respect to  $C$ , we have

$$2f(C^*) = C^* f'(C^*) \quad (6.27)$$

This is the condition that  $C^*$  has to meet. This can be solved either mathematically or numerically once  $f(C)$  is given. From Eq. (6.27) and (6.24) we have

$$D^* = \lambda W f'(C^*)$$

If we define power as

$$R \triangleq \frac{1}{D^r T} \quad (6.28)$$

then

$$R = \frac{DC - \lambda W f(C)}{D^{r+1} W}$$

Setting  $\frac{\partial R}{\partial D} = 0$  we have

$$D^* = \frac{(r+1)\lambda W f(C)}{rC} \quad (6.29)$$

Substituting  $D^*$  from (6.29) into (6.5) we have

$$\rho^* = \frac{r}{r+1}$$

Hence

$$\bar{N}^* = r$$

Again, this result is the same as found in [KLEI79]. From (6.29) we have

$$R_{D^*} = \frac{r'}{(r+1)^{r'+1} \lambda^r W^{r'+1}} \cdot \frac{C^{r'+1}}{f'(C)} \quad (6.30)$$

Setting  $\frac{\partial R_{D^*}}{\partial C} = 0$  we have

$$f(C^*) = \frac{(r+1)f(C^*)}{rC^*} \quad (6.31)$$

From Eq. (6.31) and (6.29) we have

$$D^* = \lambda W f'(C^*)$$

Notice that  $D^*$  does not change with  $r$ . From these results, we have the following theorem.

**THEOREM 6.2:**

For a distributed system, if power is defined as  $R = \frac{1}{DT}$ , power is maximized when  $C^*$  meets the following condition:

$$2f(C^*) = C^* f'(C^*)$$

and the optimal budget is

$$D^* = \lambda W f'(C^*)$$

Furthermore

$$\bar{N}^* = 1$$

If power is defined as  $R = \frac{1}{D^r T}$ , it is maximized when

$$f(C^*) = \frac{(r+1)f(C^*)}{rC^*}$$

and the optimal budget is

$$D^* = \lambda W f'(C^*)$$

Furthermore

$$\bar{N}^* = r$$

**Example 6.1:**

In this example we assume the cost-capacity function to be a power function. That is,

$$f(C) = C^a ; \quad a > 1$$

We define power as defined in (6.22). From (6.26) we have

$$R_{D^*} = \frac{C^2}{4\lambda W^2 C^a}$$

Therefore, we have the following conclusion:

- \* If  $a < 2$ , we should choose only one big processor. Hence  $C^* = 2\lambda W$ .
- \* If  $a > 2$ , we should choose as many small processors as possible.

If we define power as in (6.28), from (6.30) we have

$$R_{D^*} = \frac{r^r}{(r+1)^{r+1} \lambda^r W^{r+1}} \cdot C^{r+1-a}$$

Therefore, the conclusion for this definition of power is:

- \* If  $a < \frac{r+1}{r}$ , we should choose only one big processor. Hence  $C^* = 2\lambda W$ .
- \* If  $a > \frac{r+1}{r}$ , we should choose as many small processors as possible.

Example 6.2:

In this example we assume the cost-capacity function to be exponential. Therefore,

$$f(C) = e^C$$

From (6.27) we have

$$C^* = 2$$

and

$$D^* = \lambda W e^2$$

Therefore

$$n^* = \frac{D}{f(2)} = \frac{\lambda W e^2}{e^2} = \lambda W = L$$

Figure 6.8 shows the power versus  $n$  curve for three different values of  $D$ .

### 6.2.3.2 Optimize the Arrival Rate and the Capacity with a Given Budget

In this section, we are given the budget and we would like to find the optimal arrival rate and the optimal capacity for each processor such that power is maximized. Since the budget is fixed, we can no longer use the same definition of power as in the previous section ( $R = \frac{1}{DT}$ ); therefore we have to give another definition of power.

For a queuing system which does not saturate, the throughput of the system equals the arrival rate into the system. For a system with fixed budget, it is reasonable to ask for higher throughput and lower mean response time. Hence, we define power to be:

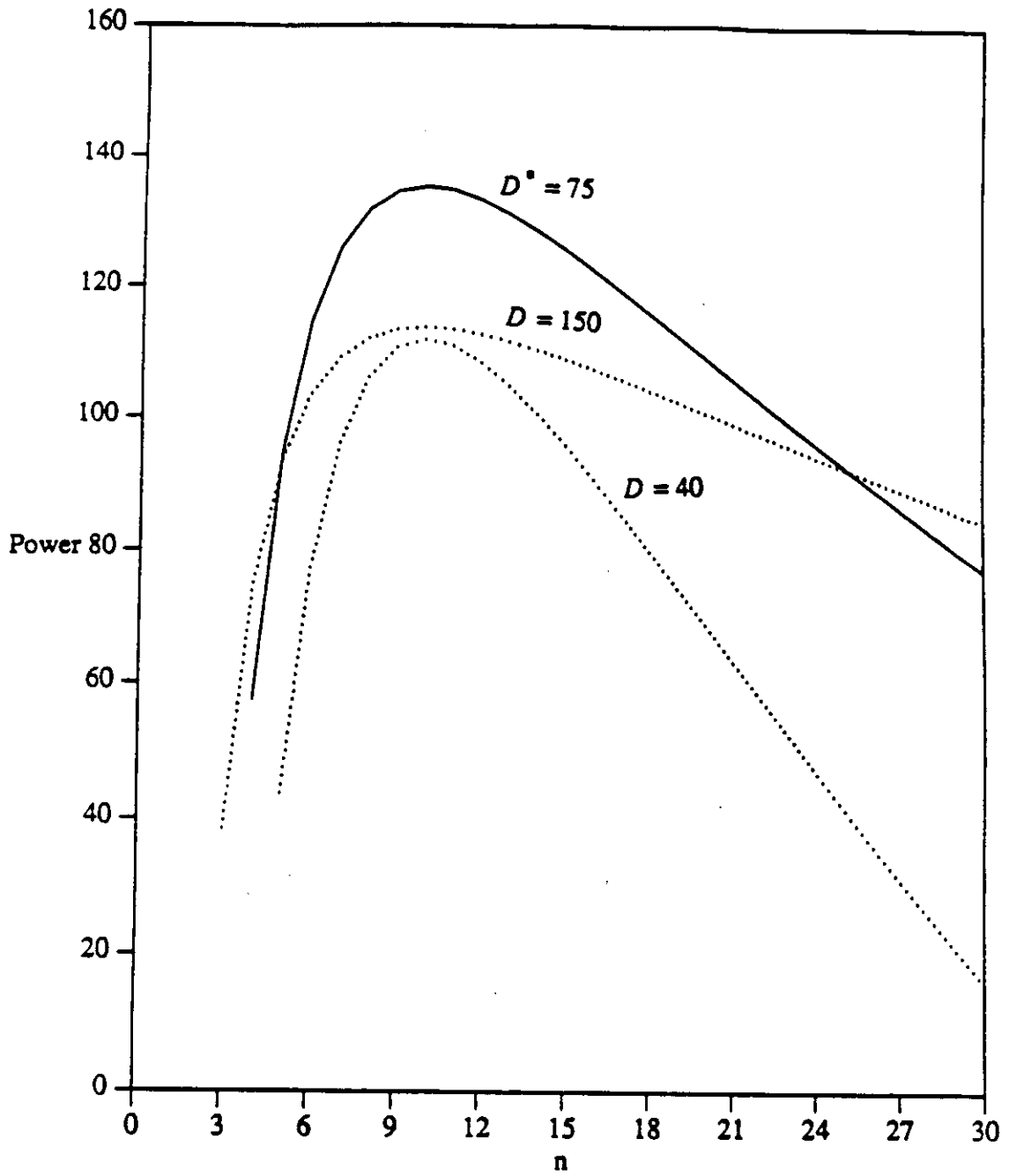


Figure 6.8  
 Power versus n for Three Different D ( $\lambda = 1, W=10$ )

$$R = \frac{\lambda}{T} = \frac{\lambda DC - \lambda^2 W f(C)}{DW} \quad (6.32)$$

Setting  $\frac{\partial R}{\partial \lambda} = 0$  we have

$$\lambda^* = \frac{DC}{2Wf(C)} \quad (6.33)$$

From (6.33) and (6.5) we have

$$\rho^* = \frac{1}{2}$$

and

$$\bar{N}^* = 1$$

Again, this is the result with the same intuitive explanation as described before. Substituting (6.33) into (6.32) we have

$$R_{\lambda^*} = \frac{D}{4W^2} \cdot \frac{C^2}{f(C)}$$

Setting  $\frac{dR_{\lambda^*}}{dC} = 0$  we have the following condition to meet for  $C^*$

$$2f(C^*) = C^* f'(C^*) \quad (6.34)$$

Interestingly, this is the same condition as given in (6.27) for  $C^*$  to meet. Hence, by defining power as we did, the problem in this section is a dual problem to the one described in section 6.2.3.1. From Eq. (6.34) and (6.33) we have

$$\lambda^* = \frac{D}{Wf'(C^*)}$$

If we define power to be

$$R = \frac{\lambda^r}{T}$$

then

$$R = \frac{\lambda^r}{T} = \frac{\lambda^r DC - \lambda^{r+1} W f(C)}{DW} \quad (6.35)$$

Optimizing  $R$  with respect to  $\lambda$ , we have

$$\lambda^* = \frac{r}{r+1} \cdot \frac{DC}{Wf(C)} \quad (6.36)$$

From (6.36) and (6.5) we have

$$\rho^* = \frac{r}{r+1}$$

and



$$\bar{N}^* = r$$

Substituting (6.36) into (6.35) we have

$$R_{\lambda^*} = \frac{1}{(r+1)W} \left[ \frac{rD}{(r+1)W} \right]^r \frac{C^{r+1}}{f^r(C)}$$

Setting  $\frac{dR_{\lambda^*}}{dC} = 0$  we have the following condition to meet for  $C^*$

$$f'(C^*) = \frac{(r+1)f(C^*)}{rC^*} \quad (6.37)$$

From Eq. (6.37) and (6.36) we have

$$\lambda^* = \frac{D}{Wf'(C^*)}$$

Notice that  $\lambda^*$  is not affected by  $r$ . From these results, we have the following theorem.

**THEOREM 6.3:**

For a distributed system, if power is defined as  $R = \frac{\lambda}{T}$ , power is maximized when  $C^*$  meets the following condition:

$$2f(C^*) = C^* f'(C^*)$$

and the optimal arrival rate is

$$\lambda^* = \frac{D}{Wf'(C^*)}$$

Furthermore

$$\bar{N}^* = 1$$

If power is defined as  $R = \frac{\lambda'}{T}$ , it is maximized when

$$f'(C^*) = \frac{(r+1)f(C^*)}{rC^*}$$

and the optimal arrival rate is

$$\lambda^* = \frac{D}{Wf'(C^*)}$$

Furthermore

$$\bar{N}^* = r$$

## 6.2.4 The Generally Distributed Service Time

In the previous section, we assumed the distribution of the service requirement to be exponential. In this section, we relax this assumption and assume a general distribution for the service time. By this assumption, we study the impact of the distribution on the issues we discussed in the previous sections in order to maximize power as defined in section 6.2.3.

### 6.2.4.1 Optimize the Budget and the Capacity with a Given Arrival Rate

In this section, we look at the same issues discussed in section 6.2.3.1 except that we assume the service time has a general distribution. Using Eq. (6.4), (6.5), and the formula from M/G/1 we have

$$T = \bar{x} \left[ 1 + \rho \frac{1+c^2}{2(1-\rho)} \right] = \frac{W}{C} \left[ \frac{2DC - 2\lambda W f(C) + \lambda W(1+c^2)f(C)}{2DC - 2\lambda W f(C)} \right]$$

Define power as in Eq. (6.22) we have

$$R = \frac{1}{DT} = \frac{C}{W} \cdot \frac{2DC - 2\lambda W f(C)}{2D^2C - 2\lambda W f(C)D + \lambda W(1+c^2)f(C)D} \quad (6.38)$$

Setting  $\frac{\partial R}{\partial D} = 0$  we have

$$D^* = \frac{2 + \sqrt{2 + 2c^2}}{2C} \lambda W f(C) \quad (6.39)$$

From Eq. (6.39) and (6.5) we have

$$\rho^* = \frac{2}{2 + \sqrt{2 + 2c^2}}$$

Using Little's result we have

$$\bar{N} = \lambda T = \rho \left[ 1 + \rho \frac{1+c^2}{2(1-\rho)} \right]$$

Hence

$$\bar{N}^* = \lambda^* T = 1$$

Again, we arrive at the same result here. From Eq. (6.3) we also have

$$n^* = \frac{2 + \sqrt{2 + 2c^2}}{2C} \lambda W \quad (6.40)$$

Note that if the service time is exponentially distributed, that is  $c^2 = 1$ , then we have the same result as before. Substituting  $D^*$  from Eq. (6.39) into Eq. (6.38), we have

$$R_{D^*} = \frac{2}{aW} \cdot \frac{aC^2 - \lambda WC^2}{2af(C) - 2\lambda W f(C) + \lambda W(1+c^2)f(C)}$$

Setting  $\frac{dR_{D^*}}{dC} = 0$  we have

$$2f(C^*) = C^* f'(C^*) \quad (6.41)$$

From Eq. (6.41) and (6.39) we have

$$D^* = \frac{2 + \sqrt{2 + 2c^2}}{4} \cdot \lambda W f'(C^*) \quad (6.42)$$

Notice that  $C^*$  does not change with  $c^2$ . Observing from (6.40), we notice that if the jobs have a higher  $c^2$ , we should buy more processors compared to jobs with a smaller  $c^2$ . The intuitive explanation is that with higher  $c^2$ , the probability of having some big jobs would be higher. If the number of processors is small, all the processors would more likely be occupied by these big jobs such that even a small job will have to suffer a long queuing time. Hence, the mean response time will get higher and therefore lower the power. Therefore, from (6.42) we observe that we should have a higher budget for jobs with a larger variance while maintaining the capacity for each processor invariant to the variance. Figure 6.9 shows the curve of  $D^*$  versus  $c^2$ . If we define power as in Eq. (6.28), then

$$R = \frac{1}{D^* T} = \frac{2C}{W} \cdot \frac{DC - \lambda W f(C)}{2D^{r+1}C - 2\lambda W f(C)D^r + \lambda W(1 + c^2)f(C)D^r}$$

Setting  $\frac{\partial R}{\partial D} = 0$  we have

$$D^* = \frac{(-c^2 + 3)r + (c^2 + 1) + b(r)}{4rC} \cdot \lambda W f(C) \quad (6.43)$$

where

$$b(r) = \sqrt{(c^4 + 2c^2 + 1)r^2 + 2(-c^4 + 2c^2 + 3)r + (c^2 + 1)^2}$$

From equation (6.43) and (6.5) we have

$$\rho^* = \frac{4r}{(-c^2 + 3)r + (c^2 + 1) + \sqrt{(c^4 + 2c^2 + 1)r^2 + 2(-c^4 + 2c^2 + 3)r + (c^2 + 1)^2}}$$

and

$$\bar{N}^* = \frac{2r \left[ (1 + c^2)r + b(r) + (1 + c^2) \right]}{(c^4 - 1)r^2 + 2(2 - c^2)(1 + c^2)r + \left[ (1 - c^2)r + (1 + c^2) \right] b(r) + (c^2 + 1)^2}$$

where

$$b(r) = \sqrt{(c^4 + 2c^2 + 1)r^2 + 2(-c^4 + 2c^2 + 3)r + (1 + c^2)^2}$$

If  $r \gg 1$ ,

$$\begin{aligned} \rho^* &= \frac{4r}{(-c^2 + 3)r + (c^2 + 1) + \sqrt{(c^4 + 2c^2 + 1)r^2 + 2(-c^4 + 2c^2 + 3)r + c^4 - 6c^2 + 9}} \\ &= \frac{r}{r + 1} \end{aligned}$$

and

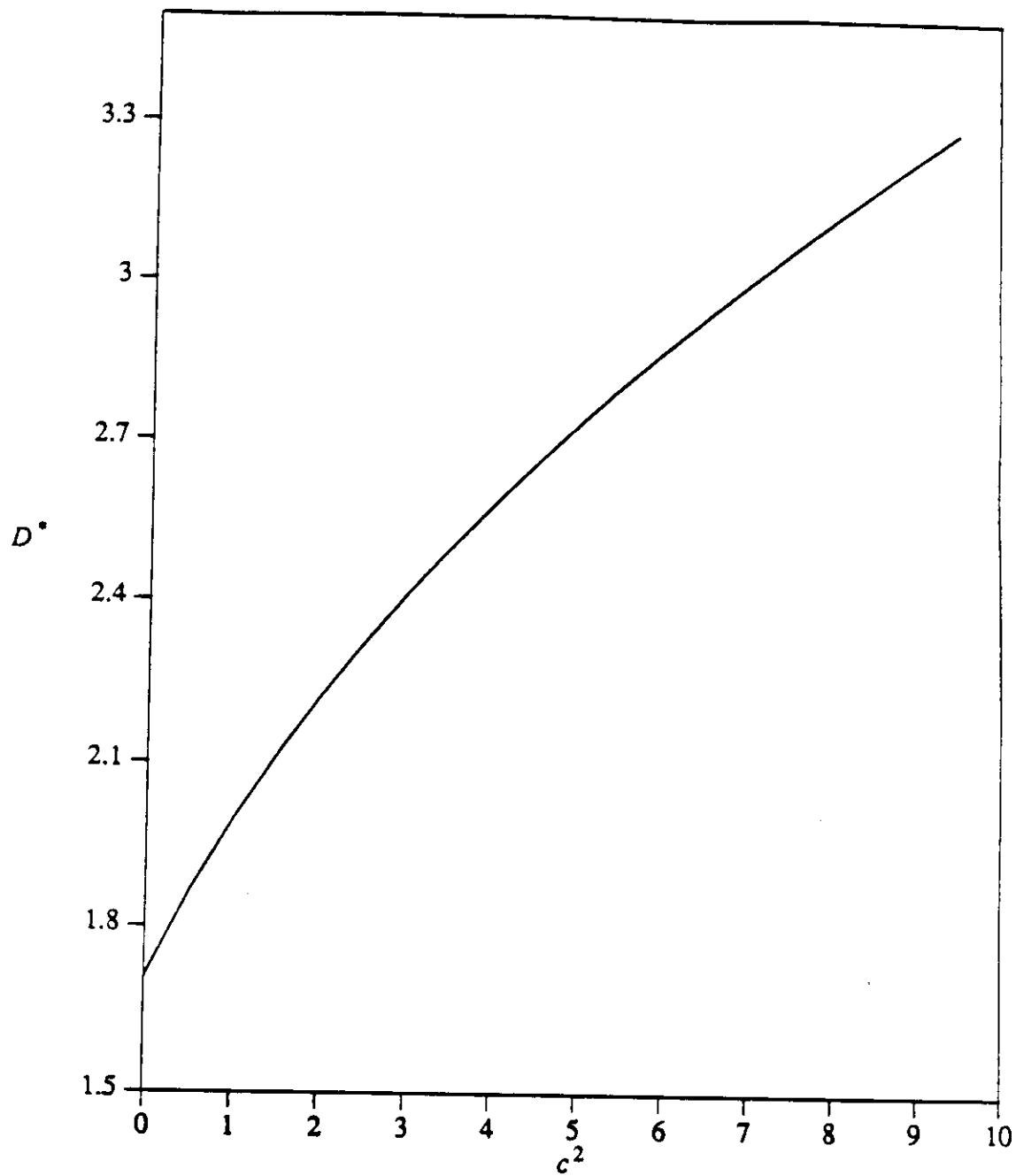


Figure 6.9  
 $D^*$  versus  $c^2$

$$\bar{N}^* \equiv \frac{(1+c^2)r}{2}$$

From equation (6.43) we have

$$R_{D^*} = \frac{2[a(r) - \lambda W]}{W a(r)^c [2a(r) - 2\lambda W + \lambda W(1+c^2)]} \cdot \frac{C^{r+1}}{f'(C)}$$

where  $a$  is defined as

$$a(r) = \frac{(-c^2+3)r + (c^2+1) + b(r)}{4r} \cdot \lambda W$$

where

$$b(r) = \sqrt{(c^4+2c^2+1)r^2 + 2(-c^4+2c^2+3)r + (c^2+1)^2}$$

Setting  $\frac{d}{dC} R_{D^*} = 0$  we have

$$(r+1)f(C^*) = rC^*f'(C^*)$$

**THEOREM 6.4:**

If the service time of a job has a general distribution with coefficient of variation  $c$  and power is defined as  $R = \frac{1}{DT}$ , power is maximized when  $C^*$  meets the following condition:

$$2f(C^*) = C^*f'(C^*)$$

and the optimal budget is

$$D^* = \frac{2 + \sqrt{2+2c^2}}{4} \cdot \lambda W f'(C^*)$$

Furthermore

$$\bar{N}^* = 1$$

If power is defined as  $R = \frac{1}{D'T}$ , it is maximized when

$$(r+1)f(C^*) = rC^*f'(C^*)$$

and the optimal budget is

$$D^* = \frac{(-c^2+3)r + (c^2+1) + b(r)}{4r} \cdot \frac{r\lambda W f'(C^*)}{r+1}$$

Furthermore

$$\bar{N}^* = \frac{2r \left[ (1+c^2)r + b(r) + (1+c^2) \right]}{(c^4-1)r^2 + 2(2-c^2)(1+c^2)r + \left[ (1-c^2)r + (1+c^2) \right] b(r) + (c^2+1)^2}$$

where

$$b(r) = \sqrt{(c^4 + 2c^2 + 1)r^2 + 2(-c^4 + 2c^2 + 3)r + (1 + c^2)^2}$$

#### 6.2.4.2 Optimize the Arrival Rate and the Capacity with a Given Budget

In this section, we discuss the same issues as in section 6.2.3.2 except that the distribution of the service time is a general distribution here. Let us define power as

$$R = \frac{\lambda}{T}$$

then

$$R = \frac{\lambda}{T} = \frac{2C}{W} \cdot \frac{DC\lambda - Wf(C)\lambda^2}{2DC - 2Wf(C)\lambda + W(1 + c^2)f(C)\lambda} \quad (6.44)$$

Setting  $\frac{\partial R}{\partial \lambda} = 0$  we have

$$\lambda^* = \frac{2}{2 + \sqrt{2 + 2c^2}} \cdot \frac{DC}{Wf(C)} \quad (6.45)$$

From equation (6.45) and (6.5) we have

$$\rho^* = \frac{2}{2 + \sqrt{2 + 2c^2}}$$

Using Little's result, we have

$$\bar{N}^* = 1$$

Substituting  $\lambda^*$  into (6.44) we have

$$R_{\lambda^*} = \frac{2a(1-a)D}{W^2(2-a+ac^2)} \cdot \frac{C^2}{f(C)}$$

where

$$a = \frac{2}{2 + \sqrt{2 + 2c^2}}$$

Setting  $\frac{dR_{\lambda^*}}{dC} = 0$  we have

$$2f(C^*) = C^* f'(C^*)$$

Notice that  $C^*$  is not affected by  $c^2$  and  $\rho^*$  is a decreasing function of  $c^2$ . Hence, as  $c^2$  increases, the optimal capacity for each processor remains unchanged; but the optimal system utilization and the optimal arrival rate would be lower. Figure 6.10 shows the curve of  $\lambda^*$  versus  $c^2$ .

If we define power as

$$R = \frac{\lambda}{T}$$

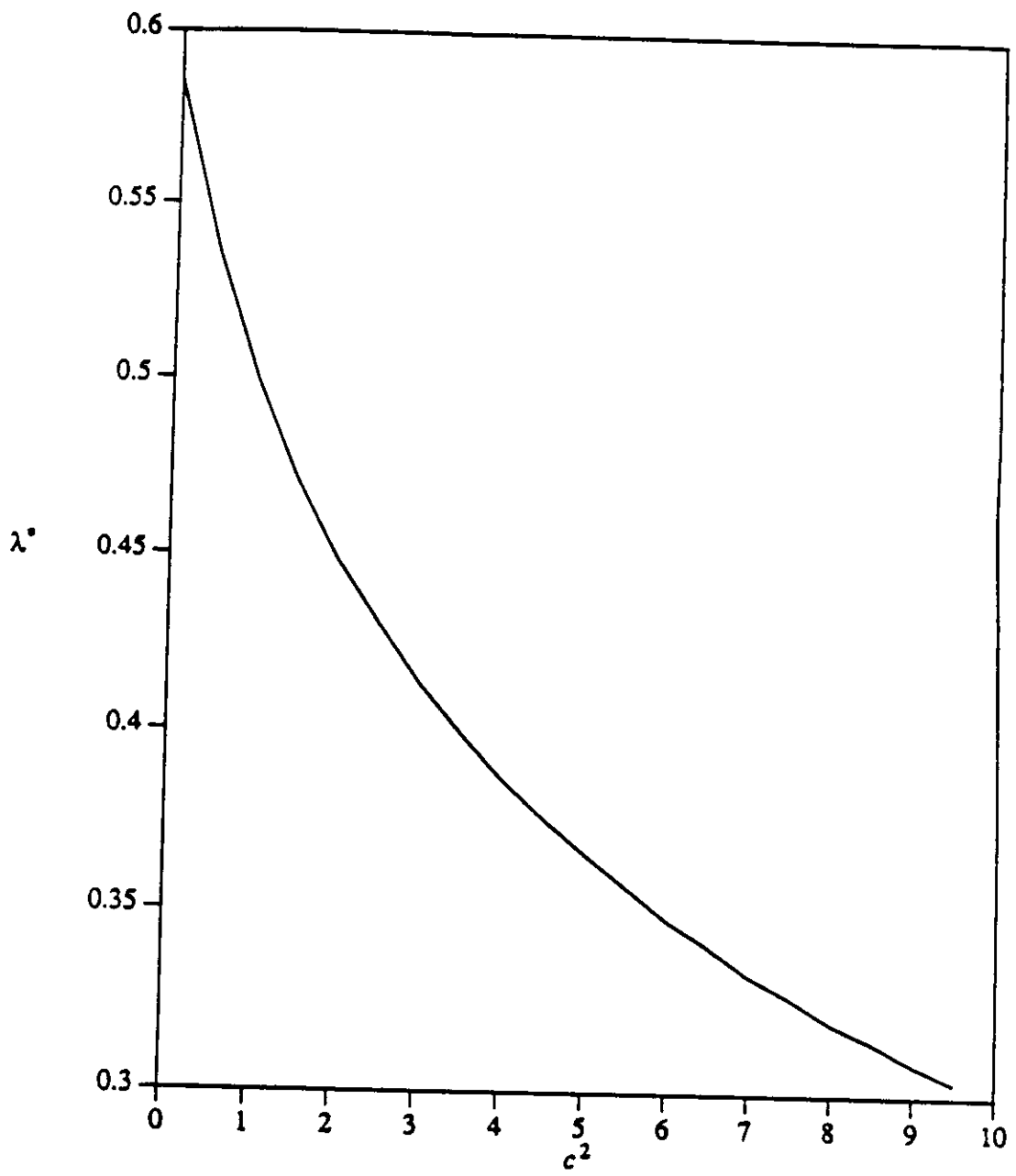


Figure 6.10  
 $\lambda^*$  versus  $c^2$

then

$$R = \frac{2C}{W} \cdot \frac{DC\lambda^* - Wf(C)\lambda^{*+1}}{2DC - 2Wf(C)\lambda + W(1+c^2)f(C)\lambda}$$

Setting  $\frac{\partial R}{\partial \lambda} = 0$  we have

$$\lambda^* = \frac{4r}{(-c^2+3)r + (c^2+1) + b(r)} \cdot \frac{DC}{Wf(C)}$$

where

$$b(r) = \sqrt{(c^4 + 2c^2 + 1)r^2 + 2(-c^4 + 2c^2 + 3)r + (c^2 + 1)^2}$$

Substituting  $\lambda^*$  into (6.5) we have

$$\rho^* = \frac{4r}{(-c^2+3)r + (c^2+1) + \sqrt{(c^4 + 2c^2 + 1)r^2 + 2(-c^4 + 2c^2 + 3)r + (c^2 + 1)^2}}$$

Using Little's result we are able to derive

$$\bar{N}^* = \frac{2r \left[ (1+c^2)r + b(r) + (1+c^2) \right]}{(c^4-1)r^2 + 2(2-c^2)(1+c^2)r + \left[ (1-c^2)r + (1+c^2) \right] b(r) + (c^2+1)^2}$$

where

$$b(r) = \sqrt{(c^4 + 2c^2 + 1)r^2 + 2(-c^4 + 2c^2 + 3)r + (1+c^2)^2}$$

If  $r \gg 1$ ,

$$\begin{aligned} \rho^* &\cong \frac{4r}{(-c^2+3)r + (c^2+1) + \sqrt{(c^4 + 2c^2 + 1)r^2 + 2(-c^4 + 2c^2 + 3)r + c^4 - 6c^2 + 9}} \\ &= \frac{r}{r+1} \end{aligned}$$

and

$$\bar{N}^* \cong \frac{(1+c^2)r}{2}$$

Substituting  $\lambda^*$  into  $R$  we have

$$R_{\lambda^*} = \frac{2}{W} \cdot \frac{b(r) - 4r}{b(r) - 4r + 4rc^2} \cdot \left[ \frac{4rD}{Wb(r)} \right]^r \cdot \frac{C^{r+1}}{f(C)}$$

Setting  $\frac{dR_{\lambda^*}}{dC} = 0$  we have

$$(r+1)f(C^*) = rC^*f(C^*)$$

#### THEOREM 6.5:

If the service time of a job has a general distribution with coefficient of variation  $c$  and power is defined as  $R = \frac{\lambda}{T}$ , power is maximized when  $C^*$  meets the following condition:



$$2f(C^*) = C^* f'(C^*)$$

and the optimal arrival rate is

$$\lambda^* = \frac{2}{2 + \sqrt{2 + 2c^2}} \cdot \frac{2D}{Wf'(C^*)}$$

Furthermore

$$\bar{N}^* = 1$$

If power is defined as  $R = \frac{\lambda^r}{T}$ , it is maximized when

$$f'(C^*) = \frac{(r+1)f(C^*)}{rC^*}$$

and the optimal arrival rate is

$$\lambda^* = \frac{4r}{(-c^2 + 3)r + (c^2 + 1) + b(r)} \cdot \frac{(r+1)D}{rWf'(C^*)}$$

where

$$b(r) = \sqrt{(c^4 + 2c^2 + 1)r^2 + 2(-c^4 + 2c^2 + 3)r + (c^2 + 1)^2}$$

Furthermore

$$\bar{N}^* = \frac{2r \left[ (1 + c^2)r + b(r) + (1 + c^2) \right]}{(c^4 - 1)r^2 + 2(2 - c^2)(1 + c^2)r + \left[ (1 - c^2)r + (1 + c^2) \right] b(r) + (c^2 + 1)^2}$$

where

$$b(r) = \sqrt{(c^4 + 2c^2 + 1)r^2 + 2(-c^4 + 2c^2 + 3)r + (1 + c^2)^2}$$

### 6.3 Parallel Systems

In this section, we consider parallel processing systems. We define a parallel processing system to be a system which can use all the processors in the system to process a job jointly. At any instant of time, there will be at most one job in service with all the processors working on this job concurrently.

The advantage of the parallel processing system over the distributed system, as stated earlier, is that the service time of a job will be much smaller. Another advantage is that for a parallel processing system, all the processors will be working as long as there is one job in the system. The major disadvantage of the parallel processing system is that although a job is allowed access to all the processors, the synchronization restrictions (precedence relationships between tasks) of the job (algorithm) may interfere with linear speedup. Minsky [MINS71] conjectured a pessimistic form for the typical speedup, namely,

$$S = \log P$$

and this kind of performance is not unlikely to be observed. Further evidence was observed in chapter two

which says that no matter how many processors are available in the system, the maximum speedup of the system cannot be greater than

$$S_{p,max} = \frac{1}{\sum_{i=1}^n \frac{f_i}{P_i}}$$

given  $f_i$  and  $P_i$  for all  $i$ .

If we believe there is a diseconomy of scale in computing, then the cost effect and the speedup effect are playing opposite roles. Considering the cost effect, because of the diseconomy of scale, we would tend to buy more processors with smaller capacity under the same budget such that the overall capacity is higher. On the other hand, if we cannot get a linear speedup, we would tend to use fewer processors, each with a higher capacity but with a lower overall capacity. Therefore, for a given budget, a given cost function, and a given speedup function, there is an optimal number of processors to use, each with a capacity such that the total budget is fixed. In this section, we assume the cost function to be a convex function and the speedup function to be a concave function.

For the rest of this section, we consider the cost function to be a polynomial function of the form  $x^a$  or an exponential function and we consider the speedup function to be a polynomial function of the form  $x^a$  or a logarithm function. For each combination, we find the optimal number of processors to use under a fixed budget.

We will first look at the general study to find the optimal number of processors to use in order to minimize the system time given the cost function and the speedup function. Let us define the following notation.

- $C \triangleq$  the capacity for each processor
- $S(n) \triangleq$  the speedup of the system given  $n$  processors in the system
- $\bar{x}_n \triangleq$  the average service time for each job given  $n$  processors in the system

With these definitions, we assume the workload for the jobs is sampled from a random variable  $\hat{W}$ , which is generally distributed with mean  $W$ . We denote the cost-capacity function to be

$$d = f_1(C) \tag{6.46}$$

Define the inverse function of  $f_1(\cdot)$  to be  $g(\cdot)$ , i.e.,

$$g(x) = f_1^{-1}(x)$$

Define the speedup function to be

$$S = f_2(n) \tag{6.47}$$

It can be shown that

$$\bar{x}_n = \frac{W}{C} \frac{1}{S(n)} \quad (6.48)$$

From (6.46) we have

$$n = \frac{D}{f_1(C)}$$

Therefore

$$C = f_1^{-1}\left(\frac{D}{n}\right) = g\left(\frac{D}{n}\right) \quad (6.49)$$

Substituting (6.49) into (6.48) we have

$$\bar{x}(n) = \frac{W}{f_1^{-1}\left(\frac{D}{n}\right)f_2(n)} = \frac{W}{g\left(\frac{D}{n}\right)f_2(n)} \quad (6.50)$$

Since this system accepts one job at a time, it behaves like an M/G/1 queueing system. Hence, in order to minimize the mean response time we need only to minimize the average service time (since the coefficient of variation does not change) given in (6.50). Therefore, in order to find the optimal  $n^*$ , we set

$$\frac{d}{dn} \bar{x}(n) = 0$$

which leads to

$$\frac{d}{dn} g\left(\frac{D}{n^*}\right) f_2(n^*) + g\left(\frac{D}{n^*}\right) \frac{d}{dn} f_2(n^*) = 0 \quad (6.51)$$

(6.51) is the equation that  $n^*$  would have to satisfy in order to minimize the mean response time.

### 6.3.1 Polynomial Cost Function - Polynomial Speedup Function

In this section we assume the cost function and speedup function are both polynomial functions of the form  $x^a$ . Since we assume the cost function to be convex and the speedup function to be concave, we assume

$$f_1(x) = x^a \quad ; \quad a > 1$$

$$f_2(x) = x^b \quad ; \quad b < 1$$

From (6.50) we have

$$\bar{x}(n) = \frac{W}{\left(\frac{D}{n}\right)^{1/a} n^b} = \frac{W}{D^{1/a} n^{a(ab-1)}} \quad (6.52)$$

From (6.52), in order to minimize  $\bar{x}(n)$ , there are three cases to be considered:

Case 1:  $ab = 1$ .

From (6.52), we have

$$\bar{x}(n) = \frac{W}{D^{1/a}}$$

Since  $\bar{x}(n)$  is not a function of  $n$ , therefore, the mean response time is indifferent to  $n$ .

Case 2:  $ab > 1$ .

For this case, we have

$$\bar{x}(n) = \frac{W}{D^{1/a} n^{a(ab-1)}}$$

Hence, we should choose as many small processors as possible.

Case 3:  $ab < 1$ .

For this case, we have

$$\bar{x}(n) = \frac{W n^{a(ab-1)}}{D^{1/a}}$$

Hence, we should choose only one BIG processor.

In summary, we have the following rules:

- \* If  $ab = 1$ , the mean response time is indifferent to  $n$ .
- \* If  $ab > 1$ , we should choose as many small processors as possible.
- \* If  $ab < 1$ , we should choose only one BIG processor

It is important to see the impact of the speedup function besides the cost function.

### 6.3.2 Polynomial Cost Function - Logarithm Speedup

In this section, we assume the cost-capacity function to be a polynomial function of the form  $C^a$  and the speedup function to be a logarithm function. Let us assume

$$d = f_1(C) = C^a \quad ; \quad a > 1 \quad (6.53)$$

$$S = f_2(n) = 1 + \log n \quad (6.54)$$

Substituting (6.53) and (6.54) into (6.50) we have

$$\bar{x}(n) = \frac{W}{D^{1/a}} \cdot \frac{n^{\frac{1}{a}}}{1 + \log n}$$

Setting  $\frac{d}{dn} \bar{x}(n) = 0$  we have

$$n^* = e^{a-1}$$

Hence,

$$C^* = \left(\frac{D}{n}\right)^{1/a} = \left(\frac{D}{e^{a-1}}\right)^{1/a}$$

Notice that  $n^*$  is not a function of  $D$  in this case. Hence, as the budget gets higher, only  $C^*$  will become larger and  $n^*$  will not be affected. Figure 6.11 shows the curve of  $\bar{x}(n)$  versus  $n$ .

### 6.3.3 Exponential Cost Function - Polynomial Speedup Function

In this section, we assume the cost-capacity function to be an exponential function and the speedup function to be a polynomial function of the form  $x^a$ .

Let us define

$$d = f_1(C) = e^{aC} \quad ; \quad a > 0$$

$$S = f_2(n) = n^b \quad ; \quad b < 1$$

It can be verified that

$$g(x) = f_1^{-1}(x) = \frac{1}{a} \ln x$$

Therefore

$$\bar{x}(n) = \frac{aW}{n^b(\ln D - \ln n)}$$

Setting  $\frac{d}{dn} \bar{x}(n) = 0$ , we have

$$n^* = \left(\frac{1}{e}\right)^{\frac{1}{b}} D \quad (6.55)$$

Hence

$$C^* = \frac{1}{a} \cdot \ln \left[ \frac{D}{e^{-1/b} D} \right] = \frac{1}{ab}$$

Notice that in this case,  $C^*$  is not affected by the budget. Hence, as the budget gets higher,  $n^*$  will get higher while  $C^*$  remains unchanged.

As shown in (6.55),  $n^*$  is linearly proportional to  $D$ . Figure 6.12 shows the curve of  $\bar{x}(n)$  versus  $n$ .

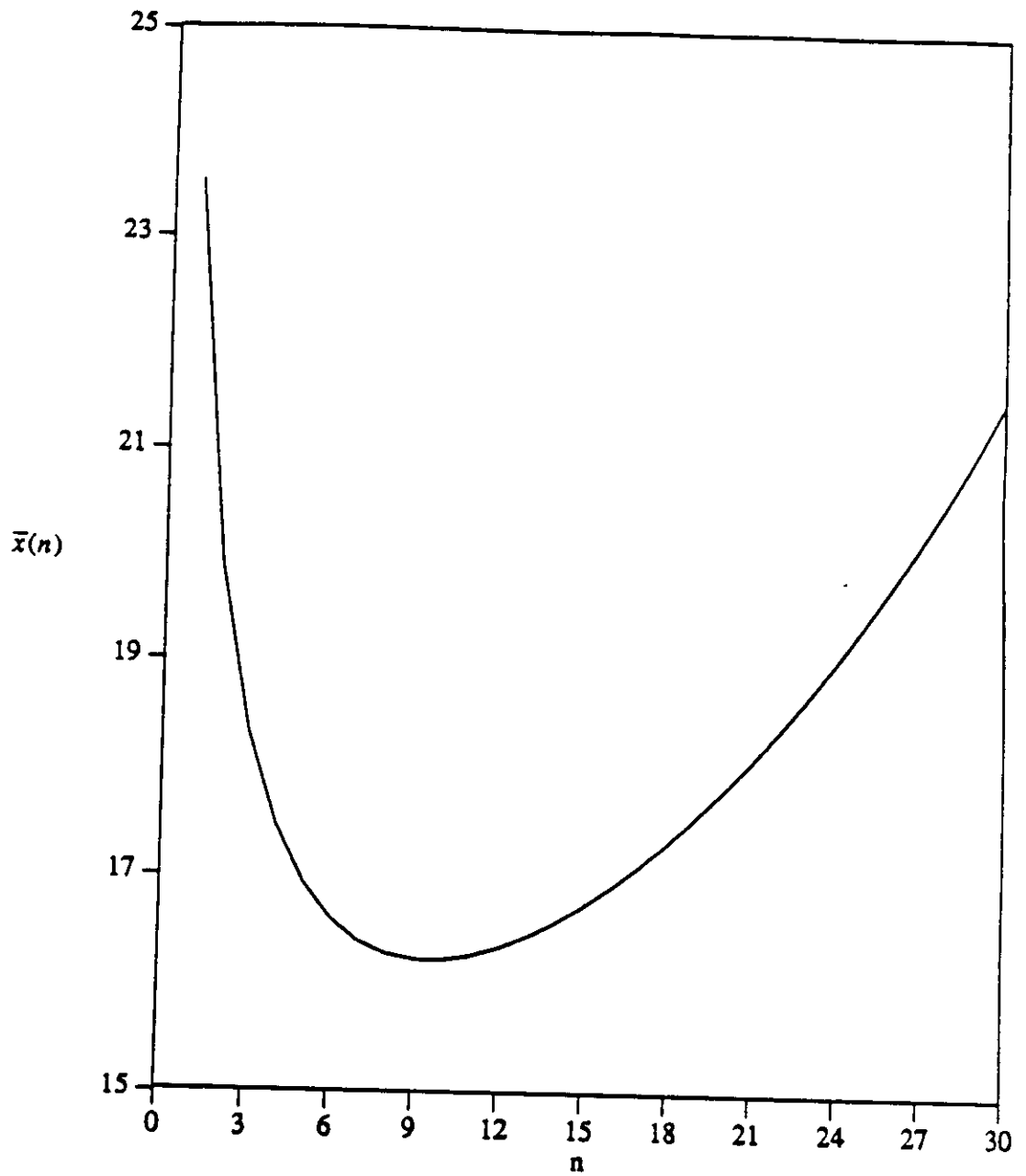


Figure 6.11  
 $\bar{x}(n)$  versus  $n$  ( $W=100$ ,  $a=3$ ,  $D=10000$ )

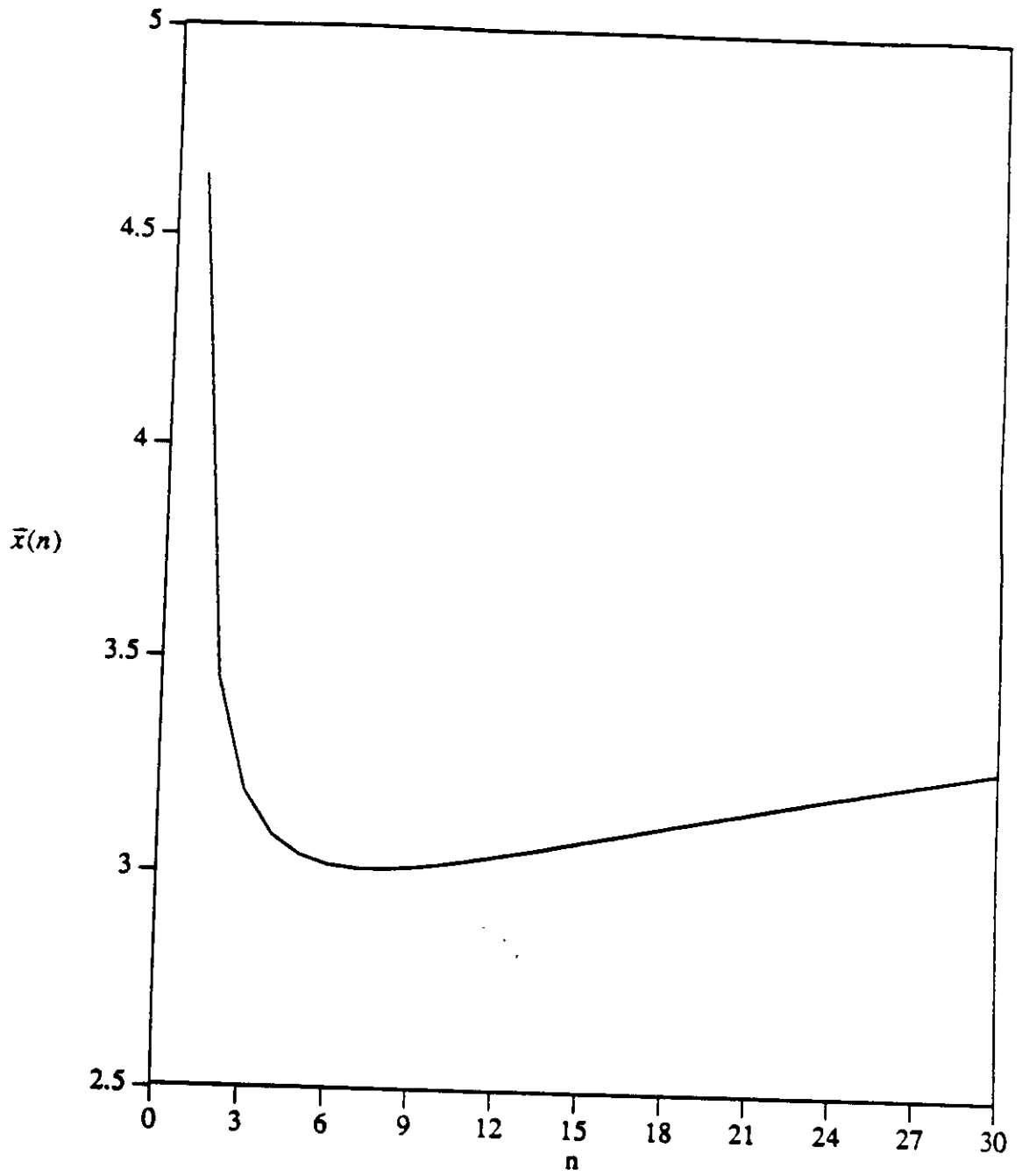


Figure 6.12  
 $\bar{x}(n)$  versus  $n$

### 6.3.4 Exponential Cost Function - Logarithm Speedup Function

In this section, we assume the cost-capacity function to be exponential and the speedup function to be logarithmic. Let us assume

$$d = f_1(C) = e^{aC} \quad ; \quad a > 0$$

$$S = f_2(n) = 1 + \ln n$$

Therefore

$$\bar{x}(n) = \frac{aW}{\ln\left(\frac{D}{n}\right)(1 + \ln n)}$$

Setting  $\frac{d}{dn}\bar{x}(n) = 0$ , we have

$$n^* = \left(\frac{D}{e}\right)^{1/2}$$

Hence,

$$C^* = \frac{1}{2a}(1 + \ln D)$$

In this case, as the budget gets higher, both  $C^*$  and  $n^*$  will both get larger. Figure 6.13 shows the curve of  $\bar{x}(n)$  versus  $n$ .

### 6.4 Conclusions

It is interesting to observe that how the cost-capacity function, the speedup function, the load of the system, and the variance of the job's service time affect the performance of a multiprocessing system. Our conclusion tells that neither the centralized system nor the distributed/parallel system prevails under all conditions. However, we do arrive at the conclusion that a distributed system would become more attractive as the load of the system or the variance of the job's service time distribution increases.



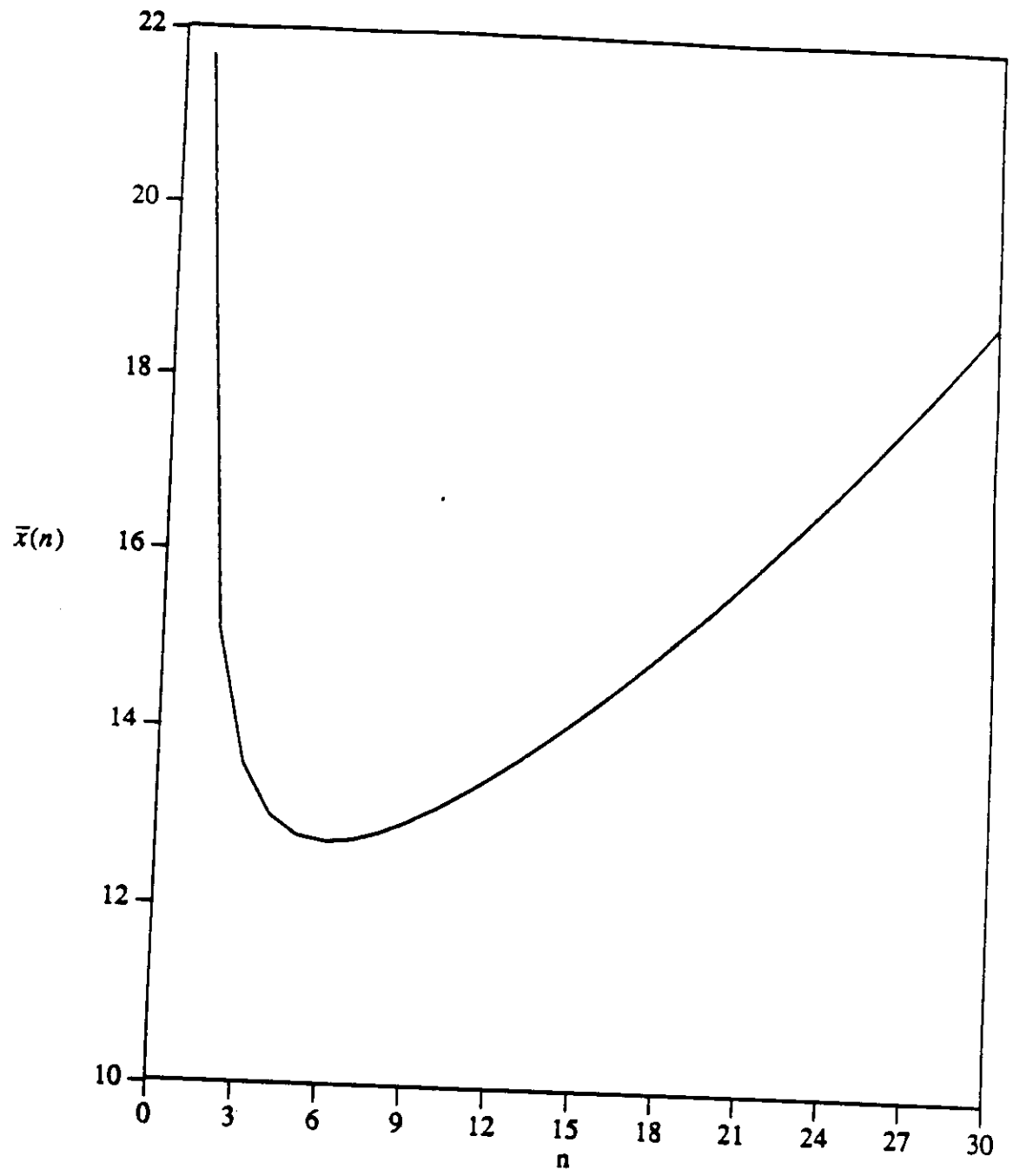


Figure 6.13  
 $\bar{x}(n)$  versus  $n$

## CHAPTER 7 CONCLUSION AND SUGGESTIONS FOR FUTURE STUDY

In this dissertation, we analyzed various performance issues of distributed and parallel processing systems. In chapter 2, for the parallel processing systems which admit only one job into service at a time, we were able to find the optimal operating point ( $\lambda^*$ ) and the optimal number of processors to be used in the system by the use of the definition of power. We also gave an expression for the processing time speedup or the response time speedup, depending on the situation. In chapter 3, for parallel processing systems which admit more than one job into service at a time, it is a very difficult mathematical problem just to find the mean response time for the general case. Hence, we provided the exact analysis for some special cases and the scale-up rule as a tool to be used as an approximation. In chapters 4 and 5, we presented two sorting algorithms as an application for distributed and parallel processing systems. Lastly, in chapter 6, we studied the effect of diseconomy of scale in computing and the effect of speedup. We also studied the effect of the variance of service time in such systems.

What remains for further study can be classified into three areas:

**Area 1:** Although we have presented three models for representing an algorithm executed in the parallel processing system, there are still examples which our models do not fit perfectly. For example, an algorithm may have a conditional branch such that it can either use  $P_1$  processors for time  $T_1$  with a probability  $p_1$ , or it can use  $P_2$  processors for time  $T_2$  with a probability  $p_2$ , ....., or, it can use  $P_n$  processors for time  $T_n$  with probability  $p_n$  under the constraint that  $\sum_{i=1}^n p_i = 1$ . In our models, we do not have this flexibility.

**Area 2:** Under the environment discussed in chapter three, we have only a two stage approximation for the general case. A minor ambition is to approximate the general cases with a three stage model. A major ambition is to get the exact solution for the general case, or the more general special cases. To achieve a better approximation than we have will also be a great contribution. Of course, in order to achieve these, a totally different analysis tool may be used. Tighter bounds, upper and lower, are also desired to understand the behavior of the system better.

**Area 3:** Although we assume there is a diseconomy of scale in computing, there is also an article [MEND87] which says that there is no economy or diseconomy of scale in computing. The economy of scale in computing is an interesting issue which we believe should deserve more attention because it may help us in designing a cost efficient multiprocessing system. Speedup, the counter part of the cost issue, is another interesting issue to be studied more carefully. Some experiments may be needed in order to get a general picture about speedup for practical applications.

## REFERENCES

- [AJTA83] Ajtai, J., Komlos, E., and Szemerédi, E., "An  $O(n \log n)$  Sorting Network," *Combinatorica*, 3(1983), pp. 1-19.
- [AKL85] Akl, S.G., *Parallel Sorting Algorithms*, Academic Press, 1985.
- [AKL87] Akl, S.G. and Santoro N., "Optimal Parallel Merging and Sorting Without Memory Conflicts," *IEEE Trans. Comput. C-36*, 1987, pp1367-1369.
- [AMDA67] Amdahl, G.M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings of AFIPS*, Vol. 30, 1967.
- [BAUD78] Baudet, G. and Stevenson, D., "Optimal Sorting Algorithms for Parallel Computers," *IEEE Trans. Comput. C-27*, 1978, pp. 84-87.
- [BELG86] Belghith, A., "Response Time and Parallelism in Parallel Processing Systems with Certain Synchronization Constraint," Ph.D. Dissertation, Computer Science Department, UCLA, 1986.
- [BORO82] Borodin, A. and Hopcroft, J.E., "Routing, Merging and Sorting on Parallel Models of Computation," *Proc. 14<sup>th</sup> Annu. Assoc. Comput. Mach. Symp. Theory Comput.*, 1982, pp. 338-344.
- [BOXM79] Boxma, O.J., Cohen, J.W., and Huijels, N., "Approximations of the mean waiting time in an M/G/c queueing system," *Oper. Res.* 27, 1979, pp. 1115-1127.
- [BRUM71] Brumelle, S.L., "Some Inequalities for Parallel Server Queues," *Operations Research*, 19, 1971, pp 402-413.
- [COLE86] Cole, R., "Parallel Merge Sort," 27<sup>th</sup> Annual IEEE Symposium on Foundation of Computer Science, 1986, pp. 511-516.
- [COOK80] Cook, C.R. and Kim, D.J., "Best Sorting Algorithm for Nearly Sorted Lists," *Commun. ACM*, Vol.23, 1980, pp. 620-624.
- [COSM76] Cosmetatos, G.P., "Some Approximate Equilibrium Results for the Multi-server Queue (M/G/r)," *Opnl. Res. Quart.* 27, 1976, pp.615-620.
- [CROM34] Crommelin, C.D., "Delay Probability Formulae," *P.O. Elec. Engrs. J.* 26, 1934, pp. 266-274
- [DECH81] Dechter, R. and Kleinrock, L., "Parallel Algorithms for Multiprocessors Using Broadcast Channel," Working Paper, No. 81002, Computer Science Department, UCLA. November 1981.

- [EIND85] Ein-Dor, P., "Grosch's Law Re-visited: CPU Power and the Cost of Computation," *CACM*, Vol. 28, No. 2, February 1985. pp. 142-151.
- [GAIL83] Gail, H.R., "On the Optimization of Computer Network Power," Ph.D. dissertation, Computer Science Department, UCLA. September 1983.
- [HILL71] Hillier F.S. and Lo F.D., "Tables for Multiple-Server Queueing Systems Involving Erlang Distribution," Technical Report No. 31, Dept. of Operations Research, Stanford University, 1971.
- [HILL85] Hillis, W.D., *The Connection Machine*, The MIT Press, 1985.
- [HOOR82] van Hoom, M.H. and Tijms, H.C., "Approximations for the Waiting Time Distribution of the M/G/c Queue," *Performance Evaluation* 2, 1982, pp. 22-28.
- [KING70] Kingman J.F.C., "Inequalities in the Theory of Queues," *Journal of the Royal statistical Society, Series B*, 32, 1970, pp 102-110.
- [KLEI75] Kleinrock, L., *Queueing Systems, Vol. 1: Theory*, Wiley- Interscience, New York, 1975.
- [KLEI76] Kleinrock, L., *Queueing Systems, Vol. 2: Computer Applications*, Wiley- Interscience, New York, 1976.
- [KLEI78] Kleinrock, L., "On Flow Control in Computer Networks," *Conference Record, International Conference on Communications* 2, pp. 27.2.1 - 27.2.5, June 1978.
- [KLEI79] Kleinrock, L., "Power and Deterministic Rules of Thumb for Probabilistic Problems in Computer Communications," *Conference Record, International Conference on Communications*, June 1979, pp. 43.1.1-43.1.10.
- [KLEI84] Kleinrock, L., "On the Theory of Distributed Processing," *Proceedings of the 22nd Annual Allerton Conference on Communication, Control and Computing*, Univ. of Illinois, Monticello, October 1984.
- [KLEI86] Kleinrock, L., "Performance Models for Distributed Systems," *Teletraffic Analysis and Computer Performance Evaluation, Proceedings of the International Seminar held at the centre for Mathematics and Computer Science (CWI)*, June 2-6, 1986. Amsterdam, the Netherlands. pp. 1-15.
- [KNUT73] Knuth, D.E., *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison Wesley, 1973.
- [KUCK84] Kuck, D.J., et al., "The Effects of Program Restructuring, Algorithm Change and Architecture Choice on Program," *Proceedings of the International Conference on Parallel Processing*, 1984.
- [KUMA83] Kumar, M. and Hirschberg, D.S., "An Efficient Implementation of Batcher's Odd-Even Merge Algorithms and its Application to Parallel Sorting Schemes," *IEEE Trans. Comput.* C-32, 1983, pp. 254-264.

- [KUNG84] Kung, K.C., "Concurrency in Parallel Processing Systems," Ph.D. Dissertation, Computer Science Department, UCLA, 1984.
- [LAKS84] Lakshminarayanan, S., Dhall, S.K., and Miller, L.L., "Parallel Sorting Algorithms," *Advances in Computers*, Vol.23 (1984), pp. 295-354.
- [LITT61] Little, J.D.C., "A Proof of the Queueing Formula  $L = \lambda W$ ," *Operations Research*, 9, 1961. pp. 383-387.
- [MAAL73] Maaloe, E., "Approximation Formulae for Estimation of Waiting-time in Multiple-Channel Queueing System," *mgmt. Sci.* 19, 1973. pp. 703-710.
- [MANN85] Mannila, H., "Measures of Presortedness and Optimal Sorting Algorithms," *IEEE Trans. on Computers*, Vol. C-34, No.4, April 1985, pp.318-325.
- [MARB86] Marberg, J., "Distributed Algorithms for Multi-Channel Broadcast Networks," Ph.D. Dissertation, Computer Science Department, UCLA, 1986.
- [MEND87] Mendelson, H., "Economies of Scale in Computing: Grosch's Law Revisited," *CACM*, Vol. 30, No. 12, December 1987. pp. 1066-1072.
- [MINS71] Minsky, M. and Papert, S., "On Some Associative, Parallel and Analog Computations," Jacks E.J. (ed.), *Associative Information Technologies*, (Elsevier North-Holland, New York, 1971)
- [MORI75] Mori, M., "Some Bounds for Queues," *Journal of Operations Research Soc. Japan* 18, 1975, pp. 152-181.
- [MUTK87] Mutka, M.W. and Livny, M., "Profiling Workstation's Available Capacity for Remote Execution," Computer Science Technical Report #697, University of Wisconsin-Madison
- [NOZA75] Nozaki, S.A. and Ross, S.M. "Approximations in Multi-server Poisson Queues," Report University of California, Berkeley, 1975.
- [ROTE85] Rotem, D., Santoro, N., and Sidney J.B., "Distributed Sorting," *IEEE Trans. on Computers*, Vol. C-34, No.4, April 1985. pp. 372-376.
- [STOY74] Stoyan, D., "Some Bounds for Many-server Systems  $G1/G/s$ ," *Math. Operationsforsch. Statist.* 5, 1974, pp. 117-129.
- [SYSK84] Syski, R., *Introduction Congestion Theory in Telephone Systems (second edition)*, North Holland, 1984.
- [THOM77] Thompson, C.D. and Kung, H.T., "Sorting on a Mesh Connected Parallel Computer," *Commun. ACM* 20, 1977, pp. 263-272.

- [TODD78] Todd, S., "Algorithm and Hardware for a Merge Sort Using Multiple Processors," IBM J. Res. Dev. 22, 1978, pp 509-517.
- [VALE75] Valiant, L.G., "Parallelism in Comparison Problems," SIAM J. Comput. 4, 1975, pp. 384-355
- [WIRT76] Wirth, N., *Algorithms + Data Structure = Programs*, Prentice-Hall, 1976.
- [WEGN82] Wegner, L.M., "Sorting a distributed file in a network," Proc. 1982 Conf. Information Sci. Systems, Princeton, New Jersey, March 1982, pp. 505-509.
- [YAKA77] Yakahashi, Y. "An Approximation Formula for the Mean Waiting Time of an M/G/m Queue," J. Oper. Res. Soc. Japan 20, 1977, pp. 150-163.
- [ZAKS85] Zaks S., "Optimal Distributed Algorithms for Sorting and Ranking," IEEE Trans. on Computers, Vol. C-34, No.4, April 1985. pp. 376-379.