CONCURRENCY IN PARALLEL PROCESSING SYSTEMS

Kenneth Ching-Yu Kung

1984

CSD-840039

UNIVERSITY OF CALIFORNIA

Los Angeles

Concurrency in Parallel Processing Systems

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

by

Kenneth Ching-Yu Kung

1984

The dissertation of Kenneth Ching-Yu Kung is approved.

[illegible]

Milos Ercegovac

Mario Gerla

Mario Gerla

[illegible]

Bennet Lientz

Steven Lippman

Steven Lippman

[illegible]

Leonard Kleinrock, Committee Chair

University of California, Los Angeles

1984

To my wife Amy and my parents

Table of Contents

List of Figures

page

List of Tables

Page

Acknowledgments

I would like to express my deepest gratitude to my advisor Professor Leonard Kleinrock for his guidance and friendship. I would like to thank Professors Milos Ercegovac, Mario Gerla, Bennet Lientz, and Steven Lippman, members of my doctoral committe, for their time and effort.

This research was supported by the Advanced Research Project Agency of the Department of Defense under contract MDA 903-82-C-0064. Transaction Technology Inc. offered tuition assistance while I was an employee there and supported this endeavor. To my fellow students at UCLA, including but not limited to Richard Gail, Mart Molle, Michael Molloy, Randy Nelson, and Hideaki Takagi, I offer my thanks for their stimulating discussions. Acknowledgement is also due the staff of our research group, Trudy Cook, Amanda Daniels, Frank Heath, George Ann Hornor, Linda Infeld, Lillian Larijani, Terry Peters, and Ruth Porty. I am grateful to members on both sides of my family for their moral support.

Most of all, I would like to thank my wife Amy, whose encouragement and understanding made this possible.

VITA

March 20, 1953	— Born, Taipei, Taiwan, ROC
1974-1976	— Teaching Assistant, University of California, Los Angeles
1976-1978	— Computer Programmer Analyst, System Development Corp., Santa Monica, California
1978	— B.S. in Engineering, University of California, Los Angeles
1978	— M.S. in Computer Science, University of California, Los Angeles
1978-1981	— Member of Technical Staff, Transaction Technology Inc., Santa Monica, California
1979-1984	— Post-Graduate Research Engineer, University of California, Los Angeles

PUBLICATIONS

Kung, Kenneth C., 'Interactive Graphics for Graph and Network Applications,' MS thesis, UCLA, 1978.

Kung, Kenneth C, and Leonard Kleinrock, 'Flow Deviation Algorithm in a Multi-hop Packet Radio Network,' Working Paper Report #81006, UCLA, 1981.

Kung, Kenneth C, and Leonard Kleinrock, 'On the Bounds of the Average System Time for Random Process Graphs,' Working Paper Report #82009, 1982.

ABSTRACT OF THE DISSERTATION

Concurrency in Parallel Processing Systems
by
Kenneth Ching-Yu Kung
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1984
Professor Leonard Kleinrock, Chair

The idea of multiprocessing has been with us for many years. We would like to know, however, how much gain (i.e., speedup) is really achieved when multi-processors are used. In this dissertation, we model a computer job as a Directed Acyclic Graph (DAG), each node in the DAG representing a separate task that can be processed by any processor. Four parameters are used to characterize the concurrency problem which results in 16 cases. The four parameters are:

1. How the jobs arrive: either a fixed number of jobs at time zero or jobs arriving from a Poisson source;

2. the DAG: either the same for each job or each job randomly selecting its DAG;

3. service time of each task: constant or exponentially distributed;

4. the number of processors: either a fixed number or an infinite number (infinite number of processors meaning that whenever a task requires a processor, one will be available).

For all cases studied, we define a common concurrency measure which gives a comparison of how much parallelism can be achieved. The concurrency measure is obtained exactly for several cases by first converting the DAG into a Markov chain where each state represents a possible set of tasks that can be executed in parallel. From this Markov chain, and by utilizing a special feature in the chain, we are able to find the equilibrium probabilities of each state and the average time required to process a single job.

We also find upper and lower bounds for the concurrency measure for certain cases studied. The upper bound is found by synchronizing of the execution at various places in the DAG.

We present two algorithms for assigning the tasks to processors. One algorithm minimizes the expected time to complete all jobs while the other algorithm maximizes the utilization of the processors.

The communication cost between any two tasks that reside on different processors is modeled as a task. We study the effect of the communication costs on the gains that are achieved from multi-processing.

CHAPTER 1
Introduction

1.1 Distributed Processing in a Network of Processors

Central processing units have been the backbone of the computing centers for many years. These machines are generally very powerful but also very expensive. Communication networks transfer data among these central processors so that the processing power of several processors may be combined and the processing resources may be shared with users of other sites. But researchers recognize the fact that even though the processing capabilities of each machine are shared by all users, the large communication time between hosts in comparison with memory access times often precludes the parallel execution of the same job on more than one machine if the networks are slow and/or costly. Thus the processors are often loosely coupled to each other with this type of communication network.

Many applications, however, require the high speed capabilities not achievable with a single serial processor. The quality of the answer a processor returns in the areas such as meteorology, cryptography, image processing and sonar and radar surveillance [HAYN82, POTT83, ROSE83] is proportional to the amount of computation performed. There are only two avenues to improve the performance. One is to speed up the processor by having faster circuits, reducing the logic levels, reducing the cycle time per operation, having high speed algorithms, and having better storage organization. The other method is to try to handle more than one task within a job simultaneously. The latter is the direction taken by the Japanese Fifth Generation Computer project.

But despite the impressive speed of many of the latest model computers, their basic architecture limits them to being serial machines and hinders their usefulness to computationally intensive problems. With the recent advances in the design and fabrication of VLSI circuits, a computing center consisting of up to tens of thousands of computing elements can be built. If we can decompose a large problem into many small concurrently executable tasks and allow several processors to work on them in parallel, we can improve the processing speed not attainable by serial machines.

Of course there is the complexity of multiprogramming and the low utilization associated with a processing center with so many processors.

Distributed processing can be defined as an architecture that has no master/slave relations among a set of processors. Instead, all processors are equal and each can access any network resources without the interference from centralized controllers [PARR83].

Each processor in a distributed processing network, therefore, needs the same basic software tools:

- the operating system software
- the application software
- the database access method and query language
- a dictionary defining the location and structure of the data in the network
- a directory defining the structure of the network
- a standard message protocol.

In order to distribute the processing of one function among various machines, these processors must be connected in some fashion. Even though there is still some communication delay, the delay between processors in a locally interconnected switch is much smaller than that in long haul communication network as described before.

Many issues are involved in distributed processing among a network of processors. In particular, the following set of problems must be addressed:

1. efficient multi-access communication protocols
2. management of the databases -- centralized or distributed
3. network management -- file directories [POPE81], network resource directories
4. security and privacy [SCHE83]
5. reliability [AVIZ81, MAKA81, NG80]
6. topology [UPFA82]

7. scheduling

8. language for parallel processing [HOLT78, SCHU81, HASE75, HASE77]

9. concurrency in processing jobs

In our research we concentrate on the last item in the above list. Concurrency of the jobs is not very well understood because the machines have often been used in a serial fashion and therefore the possibility of parallel execution in a single job has not been extensively explored.

If we have a large number of processors and these computing elements can be organized in such a way that they can cooperatively solve a single problem or attack many problems simultaneously, tremendous speed improvement can be realized. We recognize that in addition to the service time of jobs there are overhead associated with the organization of these processors. But this overhead is limited to the organization of the processors assigned to the jobs rather than the organization of all processors. Besides the advantage of the speed, this type of system offers a distributed processing environment with increased reliability, availability, expandability and better utilization of resources.

In order to take full advantage of these cooperating processors, we must understand the parallelism within computer jobs and systems. We wish to find out just how much speed up is achievable, how do we really coordinate these processors, and whether the communication between the processors is too costly for distributed processing. At the same time, the development of programming languages for parallel processing must proceed at a faster pace. Most of the existing languages do not allow parallel processing. To pick out the concurrency in these programs requires extensive preprocessing. Since most of the algorithms are not sequential, once the language for parallel processing is available, it will be easier to produce programs for the multiprocessor environment.

1.2 Existing Examples

The idea of performing more than one operation simultaneously is at least 140 years old [KUCK77]. In an October 1842 publication Menabrea describes Babbage's lecture [MORR61]:

" ... when a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes."

So, clearly, the idea of parallel processing has been around for quite a while.

Since the early 1960's, there have been many attempts to speed up execution by giving the hardware some multioperational capability. The IBM 360/91 is a pipeline machine which operates on arrays of data. ILLIAC IV [BARN68] was a parallel array machine with 64 processing elements. CRAY-1 was designed specifically for vector array processing [KOZD80]. The CRAY-1's Fortran compiler is designed to give the scientific user immediate access to the benefits of CRAY-1's vector processing architecture. This compiler vectorizes the innermost DO loops such that they can be executed in parallel. [ENSL77] contains a list of multiprocessors and parallel systems in chronological order for the years form 1958 to 1977.

A good example of using ILLIAC IV is to find the solution to partial differential equations. The difference method defines the problem on a coordinate system with given boundary values. Each grid point then uses the weighted average of the values from its neighboring grid points to find its own value. Since each grid point at each iteration can be processed concurrently, we can define each grid point at each iteration to be a separate task. Usually several of these tasks are assigned to one processor. Solutions are found when the difference of the value for each point between two consecutive iterations is smaller than a predetermined value or the solution is not obtainable due to the unstability.

1.3 Summary of Results

In Chapter 3, the concurrency problem is model by 4 parameters; they are:

1. How the jobs arrive: either a fixed number of jobs at time zero (k) or jobs arriving from a Poisson source (λ)
2. The DAG: either the same for each job (G) or each job randomly selecting its DAG (G')
3. Service time of each task: constant (x) or exponentially distributed (x')

4. The number of processors: either a fixed number (P) or an infinite number ($P = \infty$)

A process graph is defined (in Section 3.2) as a directed acyclic graph where the nodes represent the tasks within a job and the edges represent the precedence relationships among the tasks.

We use the shorthand notation "α, β, γ, δ" where α, β, γ, and δ represent how the jobs arrive, the type of DAG, the service time of each task, and the number of processors, respectively. For example, k, G, x, $P = \infty$ is shorthand notation for a system with a fixed number of jobs at time zero, a fixed process graph, a constant service time for each task and an infinite number of processors.

From these four parameters, we have sixteen separate cases as shown in Figure 3.6. Besides the two trivial cases (k, G, x, $P = \infty$, and λ, G, x, $P = \infty$) discussed in Section 3.5, the case of fixed process graphs is discussed in Chapter 4 and in Chapter 5 we discuss random process graphs. For all the cases studied, we look for a common parameter - the concurrency measure, σ, which is defined (in Section 3.3) as the average time a single job spends in the system using P processors divided by the average time a job spends in the system when only one processor is used.

1.3.1 Fixed Process Graphs

For the k, G, x^*, $P = \infty$ case (Section 4.2), we develop a method for finding the average system time. Because the number of processors is assumed to be infinite, the results obtained are independent of the number of jobs, k. The process graph G is first converted into a Markov Chain by Algorithm CPM (Section 4.2.1.1) where each state represents a possible set of tasks that can be processed in parallel. Since we know the rate out of and into each state, we have a set of balance equations. If we put these balance equations in a matrix format, we have a lower triangular matrix which can be inverted easily to obtain the equilibrium state probabilities. From these equilibrium probabilities, we can find the average system time and the concurrency measure σ.

Bounds on the concurrency measure are relatively easier to obtain than the exact value. An upper bound can be found by forcing the execution of a job to synchronize at each level of the process graph. No tasks in a level can start execution until all the tasks in its previous level have completed execution. We first study the average time required for a node to wait for all its predecessors in the previous level to complete. In each level there exists a node which has the maximum number of edges entering it. Therefore, no tasks in this level can start executing until all the predecessors of this task have been completed. By summing the time required at each level to process the task with the maximum in-degree, we obtain an upper bound on the average system time.

A lower bound is simply the average processing time of a task multiplied by the number of tasks in the longest path from the initial node to the terminating node.

For the λ, G, x^*, $P = \infty$ case (Section 4.3), the results obtained in Section 4.2 can be applied directly. With an infinite number of processors, once a job enters the system, it is immediately served and no waiting time is required.

We briefly discuss the case of k, G, x^*, $P < \infty$ using the Stochastic Petri Nets model in Section 4.4. Any process graph can be converted into a Petri Net. A 'place' representing the available processors is added to the Petri Net such that at each 'transition' if a processor is needed, a 'token' is obtained from this place, and whenever a transition with a processor token finishes, the token is also returned to this place. By using the analysis provided by Stochastic Petri Nets theory, we can find.the average utilization of the processors.

In Section 4.5, we study the assignment problem for the case of k, G, x, $P < \infty$. Two scheduling algorithms are analyzed for diamond-shaped process graphs - one algorithm gives the worst case assignment and the other algorithm gives the best case assignment. By studying the ratio of the average system time using the worst algorithm and the best algorithm, we find that the ratio between the two assignments is not large (less than two). Therefore, if we allow for random assignment (an available processor is given to any task that is ready to execute), the resulting average system time will fall in between the two boundary values.

1.3.2 Random Process Graphs

In Chapter 5 we look at some of the properties of random process graphs. We find (in Section 5.2.3) that the number of arrangements for N tasks with respect to the number of levels may be approximated by a Gaussian distribution (recognizing that this approximation permits a negative number of levels, which is clearly impossible). Since no arrangements can have less than one level, we assume the probability of any arrangement with less than one level equals to zero. In other words, most arrangements have $\frac{N}{2}$ levels as N becomes large. Using the Chernoff bound, the tail probability of this distribution is found in Section 5.2.4.

In the case $k, G^*, z, P = \infty$, as N approaches infinity, where N is the number of tasks within a job, we find the average system time approaches the value of $\frac{N}{2}$ multiplied by the average task service time, and the concurrency measure approaches $\frac{1}{2}$.

For the case $k, G^*, z^*, P = \infty$, we found and proved an arrangement that will provide the upper bound for system time over all arrangements with N nodes. An upper bound is presented in Section 5.4.1.1 while a lower bound is presented in Section 5.4.1.2. Both bounds are expressed probabilistically; they are approximately $\frac{1}{2}\frac{N}{\mu} \leq S \leq \frac{3}{4}\frac{N}{\mu}$ where S is the average system time of a job, N is the number of tasks within a job and $\frac{1}{\mu}$ is the average service time of a single task.

If the number of precedence relationships is also given, then we define the *minimally connected process graph* in Section 5.4.2.1. M_c is defined to be the minimum number of edges required to fix all the nodes of a particular process graph at their proper levels. We also give expressions for $Max\ M_c$ and $Min\ M_c$ for any process graph with N nodes and L levels. Using this concept, we give an upper bound in Section 5.4.2.2. Section 5.4.3 then compares the two upper bounds obtained in Section 5.4.1.1 and 5.4.2.2.

In Section 5.5, we try to find the optimal number of processors a diamond-shaped process graph in the case $k, G^*, z, P < \infty$ will require such that a function called power is maximized. Power is defined to be the average utilization of the processors divided by the normalized average system time [KLEI79]. An expression is provided for the optimal number of processors per job.

Section 5.6 briefly discusses two loose bounds for the average system time for the case $k, G^*, z, P < \infty$.

1.3.3 Communication Overhead

For the case $k, G, z^*, P = \infty$, we study the effects of the communication overhead between processors. We add a communication task between any two neighboring tasks in G that do not reside on the same processor. The average time for the communication tasks is expressed as a multiple 'a' of the average service time for a regular task. Using the same technique presented in Section 4.2, we find the resulting average system time as a function of 'a'.

In Section 6.2, we limit the number of processors to P, and study the effects of the communication overhead with various values of P on a process graph. By varying the parameter 'a', we obtain a family of curves for the average system time versus the number of processors.

In Section 6.3, we put a further condition on the communication overhead by allowing only one communication bus. Thus, only one communication task may be transmitting at any particular instant. We modify the technique of Section 4.2 for the analysis required in this section.

CHAPTER 2
Background and Related Work

2.1 Brief History

In the late 50's, relatively few people knew how to work with computers, and an entire computer was dedicated to one person at a time. If one needed to use the computer, he would reserve a time slot, and the machine during that time period would be dedicated to him. Each user waited for an empty time slot in order to use the machine (i.e., the sign up sheet was the scheduler). The advantage of this concept was that each user had the full processing power of the machine while he was using it. The drawback, on the other hand, was the low utilization of the processor since users spent great deals of time thinking.

As the machine became faster and more costly, it was not economically feasible to maintain the previous arrangement. To utilize the machine more efficiently, users were required to punch their computer jobs on cards and submit them to a computer operator. The computer operator would then schedule jobs by putting the cards of different jobs into the card reader in some predefined order. In this manner, the computer was kept busy most of the time, but the turnaround time for jobs could not be predicted and could vary from several hours to several days.

The next step was a compromise between the two extremes of either low utilization but dedicated machine or high utilization but long turnaround time. Operating systems were developed so several jobs could share the processing facilities of the system at the same time. Priorities were given to jobs so that interactive users who required small amounts of processing time had the highest priorities and long batch jobs had the lowest priorities. Because the memory was also shared among the users, paging and the technique of virtual memory were developed. Various methods of sharing the processor were studied and used to increase the throughput and to lower the average waiting time of jobs.

An expensive and powerful central processing unit has several drawbacks. The overhead of the operating system software in controlling a multi-programming environment is high. Jobs are constantly being swapped in and out of the high speed memory. Such operating systems are also very cumbersome, as witnessed by the fact that there are still errors in the IBM/MVS operating system,despite many releases.

The central processor also has the problem of reliability and availability. When the processor in a single processor machine goes down, the entire machine is not available to users. A few computer manufacturers have tried to solve this problem by introducing the fault-tolerant machine (e.g., NONSTOP MACHINE^{TM} by TANDEM). It is comprised of several processors and memories with the operation of one processor backed up by another processor; so, whenever one processor goes down, its twin processor will start up right away.

This leads to the multi-processor environment that we are addressing. As computer jobs enter the system, they are processed by one or more identical processors. Thus, the system will be running even though several processors might not be available, and it is more reliable since any processor can process all the jobs. As the price of the VLSI keeps on dropping, it will be possible to build a computer center consisting of large numbers of processors. They are many difficult issues related to multi-processors as mentioned in Chapter 1, but we would like to explore the concurrency within the jobs and to take advantage of multi-processing for improvement of system performance.

2.2 Graph Model of Behavior

Our representation of jobs, as described in Chapter 3 and used throughout the later chapters, is similar to the UCLA Graph Model of Behavior (GMB), which uses a control graph and an associated data graph. The control graph is, essentially, a variation of the Petri Net because the edges represent conditions and the circles the transitions. Logic expressions are assigned to the sets of both input and output edges. These expressions are made up of 'and' and 'or' logic. Computation is simulated by the movement of tokens from edges through nodes to edges. The logic expressions determine from which input edges the tokens are removed and to which edges the tokens are delivered. Each processor is associated with one or more operations in the control graph. When an operation is initiated by the control graph, the processor associated with that operation executes its procedure. For each operation, the data graph provides the locations for both the input data and the stored output data. After the control graph has been determined, analysis of the GMB is carried out by simulation. This requirement for simulation places a large overhead on the analysis of each problem. Part of our work is based on a whole class of jobs instead of an individual job; hence, it is easier to obtain system parameters and to generalize for a large number of jobs.

In a series of works on GMB, a sequence of researchers [ESTR63, MART66, MART67a, MART67b, MART67c, MART69, BAER68, BOVE68, RUSS69] associated various types of input and output logic with each node of a directed graph model identified the random variables which arise as a result of the application of programs to different sets of input data and applied the model to evaluation of the effectiveness of parallel processor systems. In the course of these studies, they evolved algorithms for the transformation of cyclic graphs to acyclic graphs, maintaining equivalence of the graphs in the sense that mean path length is preserved for any transformed cycle [MART67b]. They developed effective algorithms to calculate the probability of ever reaching a given node in the graph [BAER70] and formed upper and lower bounds on the number of processors required for maximum parallelism [MART69, BAER69]. Fernandez [FERN72] transformed the acyclic graphs by adding precedence relationships in such a way that the execution time of the resulting graph does not change but utilizes the minimum number of processors. Using GMB, Ramamoorthy [RAMA72] also scheduled the tasks such that the total execution time is minimized, and the minimum number of processors required to realize this schedule is obtained.

2.3 Related Work

2.3.1 Petri Nets

Petri Nets [PETE81] were designed to model systems with interacting concurrent components. They are widely used in the area of software verification. By modeling a program using a Petri Net and generating all possible 'markings,' we can detect the existence of deadlocks. By themselves, though, Petri Nets ignore the random time duration between the firing of two transitions, i.e., the time interval between two markings of a Petri Net. In [RAMA80], a constant time unit is associated with each transition. The performance is measured by finding the minimum cycle time, which is the time required to process a job. Molloy [MOLL81] introduced the Stochastic Petri Net (SPN), in which a random variable representing the firing delay is assigned to each 'transition.' Each marking in the Stochastic Petri Net, which represents a set of concurrently active tasks, is associated with a state in a Markov chain. By solving for the state probabilities in this Markov chain, we can obtain the density of 'tokens' in each place or each marking. In Chapter 4 we will show how this model can assist us in solving for some system parameters.

A disadvantage of the SPN is that each marking reachable from the initial marking is a state in the Markov chain. As the number of tokens increases, the number of states in the Markov chain grows at an even faster rate, making the SPN analysis very difficult. For this reason, the SPN cannot model a system with open arrivals, where the number of jobs is undetermined or the number of tokens and states is unlimited. Even when analysis is possible, a generalization of the result obtained for one specific job to other jobs is not possible.

2.3.2 Automatic Detection of Parallelism

Parallelism in programs may be either explicit or implicit. Explicit parallelism is specifically indicated by programming features such as COBEGIN/COEND.

Implicit parallelism is the parallelism that exists in the algorithm but is not explicitly stated. Some common techniques used by compilers for detecting implicit parallelism are:

1. Loop Distribution
   Sometimes the statements within the loop may be executed in parallel. This idea was introduced by Muraoka [MURA71] and later was implemented by Kuck [KUCK72,KUCK74] in their FORTRAN program analyzer to measure potential parallelism in ordinary programs.

2. Tree Height Reduction
   By making use of the associative, commutative and distributive properties, compilers may detect implicit parallelism in algebraic expressions and produce object code for multiprocessors. For example,

   $$((p+q)+r)-s)$$

   can be replaced by

   $$(p+q)+(r-s)$$

   and

   $$a*(b*c*d+e)$$

   can be replaced by

   $$a*b*c*d + a*e$$

   Assuming that only associativity and commutativity are used to transform expressions, Baer and Bovet [BAER68] gave a comprehensive tree-height reduction algorithm.

Later, Beatty [BEAT72] showed the optimality of this method.

In [RAMA69], a survey of techniques for recognizing parallel processable streams in FORTRAN programs was presented. These algorithms are primarily concerned with detection of parallelism within the arithmetic expressions. The problem of protecting common data was recognized. If two tasks are executed in parallel and they both access the same data cell, then different orders of execution will possibly result in different answers.

Russell [RUSS69] developed an interactive system in which a graphical display of potential parallelism in Fortran programs together with detected bottlenecks, is presented for further analysis by the use.

In [KUCK77], the possibility of speed up in FORTRAN programs is also studied. Three levels of parallelism were discussed. They are:

1. Parallelism within a line of code
   This referred to the reduction of the tree height in an arithmetic expression.
2. Parallelism within a program
   Concurrent execution of the loops in a program was explored.
3. Parallelism within the hardware
   The hardware organization of pipeline and array processors was discussed.

Several specific FORTRAN programs were analyzed [KUCK72,KUCK74]. They showed the speedup of the programs and the efficiency and utilization of processors for each of the programs. All of the programs resulted in some speed up, most of them by a large amount. The utilization is, as expected, quite low. But most interestingly, they conclude that, as the number of processors increased, the speed up is more than the logarithm of the number of processors predicted by Amdahl in [AMDA67].

2.3.3 Multiprocessor Hardware Organization

One of the problems in the design of a multiprocessor system is determining the means of connecting the multiple processors and the I/O processors to the storage units.

The four common multiprocessor system organizations are:

1. Bus

   The bus organization uses a single communication path (such as Ethernet) between all functional units -- processors, storage units and I/O processors. Multi-access protocols are required to share this common transmission medium.

2. Crossbar-switch

   In this organization, there is a separate path to every storage unit. The hardware must be capable of resolving conflicts within the same storage unit.

3. Shuffle/Exchange [THAN81]

   In the Shuffle/Exchange network, there exist $\log_2 N$ columns of routing switches connecting N processors to N memory modules. Each column consists of $N/2$ two-input, two-output switches. Figure 2 shows an example of this organization with $N = 8$.

4. Hierarchical [THAN81]

   A hierarchy is imposed on the set of processors and memory units. In such a structure, each processor has immediate access only to part of the system memory. Any reference to remaining memory must be handled by a higher level processor.

   Two examples of this organization are C-mmp and Cm^*. C-mmp [SIEW78a,SIEW78b,WULF80] is a 16-processor system consisting of PDP-11/40 minicomputers. The processors share 16 storage modules through a crossbar-switch matrix. Cm^* [SIEW78a,SIEW78b,HAYN82b] consists of 50 LSI-11 microprocessors. It is constructed from processor-storage pairs called computer modules. Each of these is referred to as a Cm. Cm's are grouped into clusters, and clusters are connected by intercluster busses.

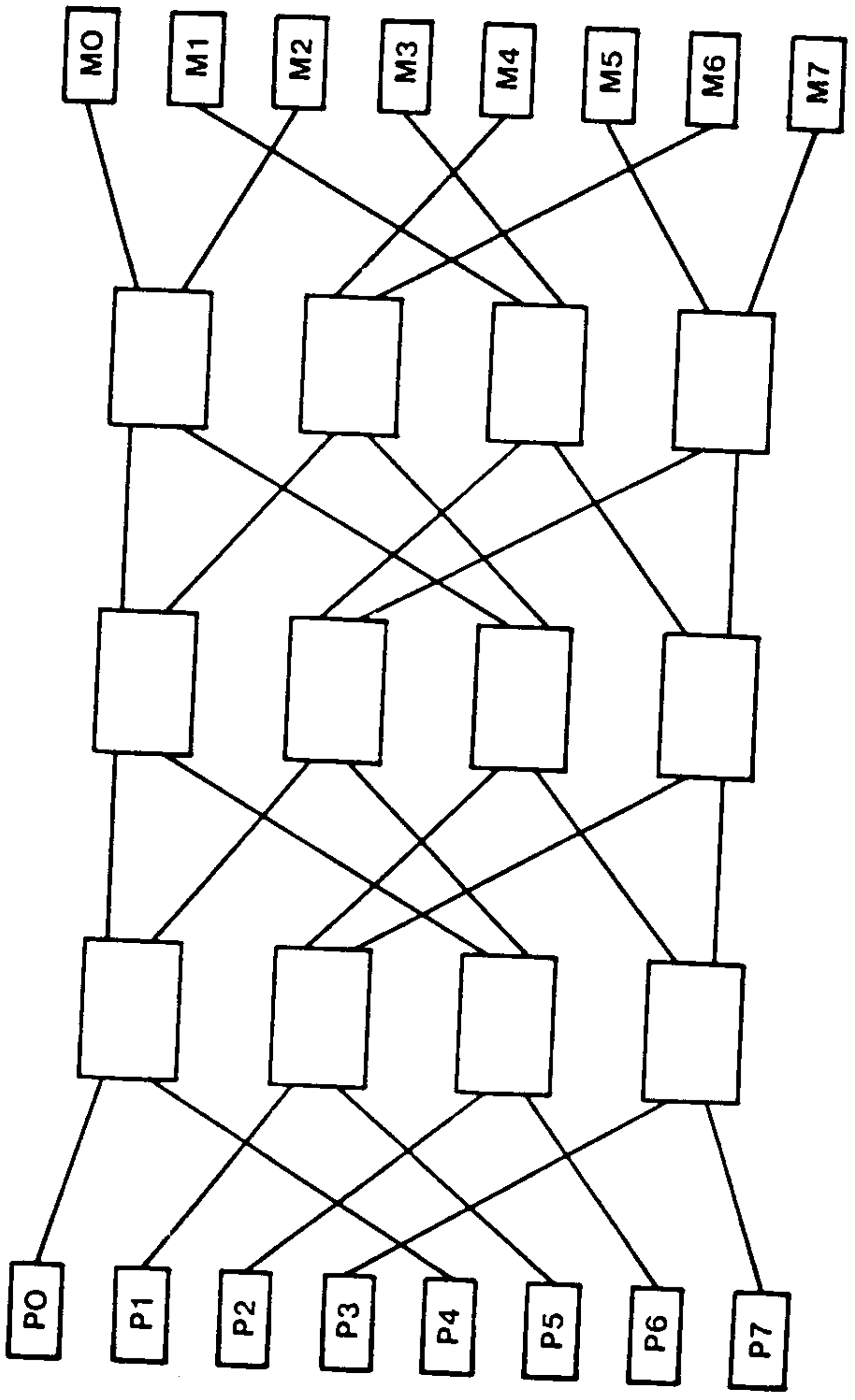


Figure 2 Shuffle/Exchange Network

2.3.4 Theory of Branching Processes

The theory of the branching process [HARR63] deals with the problem of having one node initially, with probability, p_k, that there will be k descendents at each iteration, i, for each of the nodes on the $(i-1)^{th}$ level, $k = 0, 1, 2, \cdots$. It deals with the expected number of descendents at the i^{th} iteration would be, and what would be the probability that, after i iterations (for a large i), there would be no descendents left. The tree of descendents obtained is similar to the structured process graph described in Chapter 4. Since the number of descendents at each level is random, the resulting tree can also be thought of as a random process graph (defined in Section 3.3).

The generating function of the number of descendents at the i^{th} iteration was found to be $[f(z)]^k$, where k is the number of descendents at the $(i-1)^{th}$ iteration,

$$f(z) = \sum_{k=0}^{\infty} p_k z^k$$

and z is the transformation variable. The expected value and the variance of the number of nodes at the i^{th} iteration have been found.

Two problems, however, prevent this model from representing the tasks in computer jobs. One is that there exists the possibility that the descendents will not die out. This corresponds to the fact that a computer job will not terminate. If the descendants do die out, another problem is how to merge the task having no descendants together. This corresponds to the question of how, after individual tasks are completed, the results are to be incorporated into each other to form the final solution.

2.3.5 Bounds on the Average System Time

In [ROBI79], bounds on the average system time of a tree-shaped process graph were obtained, using arguments similar to those that we have used in Section 4.3.

Since the process graph is in the form of a tree, there exist distinct paths from each task toward the root of the tree (the terminating task). Let $C_1, C_2, \cdots, C_m$ be all the paths from leaf tasks (i.e., tasks without any precedence relationships entering it) to the terminating task, and let H_i be the set of all tasks at level i, for $1 \leq i \leq L$, where L is the number of levels in the tree. Assume that the number of processors is infinite and that each task T_j has a random process time equals to t_j. Then, the expected time to process this process graph, S, is bounded by

$$\underset{1 \le i \le m}{Max} \left(\sum_{T_j \epsilon C_i} E(t_j) \right) \le S \le \sum_{1 \le i \le L} E\left(\underset{T_j \epsilon H_i}{Max}\ t_j \right)$$

In Section 4.2.2, we develop this bounding technique for general process graphs (using the concept of a "structured process graph") instead of restricting to the special case of a tree-shaped process graphs.

2.3.6 Task Scheduling

There are many scheduling algorithms in the literature. However, the majority of them deal with the single processor scheduling problem. In this section we look at some algorithms that discuss the scheduling problem in the multiprocessor situation.

Price [PRIC83] discussed a shortest path algorithm which is used to solve the scheduling problem of assigning tasks to processors. First, a distributed algorithm for finding the shortest path from one node to all other nodes in a directed acyclic graph (DAG), using as many processors as needed, is presented. At the i^{th} iteration of the algorithm, the shortest path from the root to node j, $d_j^{(i)}$, is computed as the minimum of the distance obtained at the $(i-1)^{st}$ iteration, $d_j^{(i-1)}$, or the distance from other nodes at the $(i-1)^{st}$ iteration plus the edge cost to node j, e_{kj}, where k is a neighbor of node j:

$$d_j^{(i)} = \min_k \left(d_j^{(i-1)},\ d_k^{(i-1)} + e_{kj} \right)$$

This computation can be performed in parallel at each node. For an N-node DAG, this algorithm will find the shortest path to all nodes at the end of the N^{th} iteration. To use this algorithm to solve the task assignment problem, we execute the following changes. Suppose e_{ij} is the cost of executing task i on processor j, and c_{ik} is the cost of communication incurred if task i and task k reside on different processors. The desired assignment [PRIC81, PRIC83] is one which minimizes

$$C = \sum_{i=1}^{N} \sum_{j=1}^{P} e_{ij}\, x_{ij} + \sum_{i=1}^{N} \sum_{k=1}^{N} c_{ik} - \sum_{i=1}^{N-1} \sum_{j=1}^{P} \sum_{k=i+1}^{N} c_{ik}\, x_{ij}\, x_{kj}$$

where $x_{ij} = 1$ if task i is assigned to processor j, $x_{ij} = 0$ otherwise, $\sum_{j=1}^{P} x_{ij} = 1$ for all i (each task i is assigned to exactly one processor) and P is the number of processors.

If the process graph is tree-shaped, then a DAG is generated from it by creating $N*P$ nodes

$$[1,1] \cdots [1,P][2,1] \cdots [2,P] \cdots [N,1] \cdots [N,P]$$

where node $[i,j]$ represents the assignment of task i to processor j. In addition there is an initial node $[0]$ and a terminating node $[t]$. Edges are generated by the following rule:

- edges from $[0]$ to $[i,j]$ are labeled with e_{ij} where i corresponds to the root task in the process graph.

- edges from $[i,j]$ to $[k,j]$ are labeled e_{kj}.

- edges from $[i,j]$ to $[r,q]$ are labeled $(e_{rq} + c_{ir})$.

- edges from $[i,j]$ to $[i,k]$ and from $[i,j]$ to $[i,j]$ do not exist.

- edges from $[i,j]$ to $[t]$ are labeled with the value 0 for every task i that is a terminal task in the process graph.

A specific assignment of N tasks to the P processors consists of a path from node $[0]$ to node $[t]$, and the optimal assignment is the path that minimizes the objective function C.

Stone [STON77] uses the Network Flow Algorithm to solve for the optimal solution of assigning tasks to two processors. The N tasks are so connected that the edge weights represent the cost of inter-task references (the communication costs) when the two tasks are assigned to different processors. Next, two nodes, S_1 and S_2, which represent processors P_1 and P_2, are connected to each task with the edge weight representing the cost of executing this task on that processor.

Assuming that all the edge weights are the capacities in a flow network, Stone finds the maximum flow from S_1 to S_2. A cut set is found which divides the tasks into two sets. The tasks in the same set as S_1 are assigned to processor P_1, and the other tasks are assigned to processor P_2.

Although this method does provide the optimal solution, it is not easy to generalize into cases with more than two processors. In the case where P is very large, this method is, indeed, very difficult to apply.

Van Tilborg [VANT81] discussed the Wave Scheduling technique. The processors are organized into a tree-shaped hierarchical structure with the 'worker' processor at the leaves and 'manager' processors at the higher levels in the control tree.

Assume that a job requiring S processors enters any processor (either worker or manager). If this is a worker processor, it will pass the request to its manager. The manager at this level will try to assign S tasks to the workers under its control that are not busy. If it cannot schedule a job of size S, the job is passed up the control tree one level at a time until a manager can find at least S workers under its control that are not busy and assigns the workers to the tasks. Because of the communication delay, the manager might not have updated information regarding the busy status of all the workers under his control; therefore, the manager will always try to assign the S tasks to P processors where P is slightly larger than S. The difficulty is then to estimate the value of P.

Lee [LEE77] studied the problem of optimally assigning tasks to processors by minimizing a cost function, which is the sum of two parts:

1. processing cost of a task on the processor assigned;

2. communication cost, which is the product of the volume of data to be transferred and the distance of the two processors measured by the number of hops or the physical distance.

He then discussed several assignment algorithms that minimize the above cost function for tree-shaped process graphs and more generalized process graphs.

Except for the algorithms discussed in [LEE77], none of this previous work included precedence relationships among the set of tasks. In Chapters 4 and 5, we do incorporate precedence relationships among the tasks into the scheduling algorithms.

2.4 Discussion

In this chapter, we introduced some background information on why multi-processor systems have become more important. We looked at some results obtained from the analysis of the Graph Model of Behavior and some previous attempts which take advantage of parallelism existing in programs. Some multi-processing hardware organization(s) and the scheduling problems on multi-processors were also studied. As summerized in Section 1.3 we extend these results to our model of computer jobs and find the speed up achievable in the multi-processor environment.

CHAPTER 3
A General Model

We define our system to be a set of processors plus a queue with unlimited waiting room. In the case where there are a fixed number of jobs in the system, all jobs are initially present; otherwise, jobs arrive at the system by some random arrival process. Each job brings to the system a set of tasks represented by a process graph (described in section 3.2), and each task requires processing by a resource (described in section 3.1). A job departs our system after its final task has been completed. The system time of a job is defined as the interval from the time of arrival until the completion time of the last of its tasks.

3.1 Resources

The resources studied here consist of a set of identical processors, connected via a local communication network, and each capable of independent operation on a single task. In this dissertation, we concentrate on the problem of task assignment; we defer questions regarding the types of connection networks most suitable for parallel processing communications, the amount of storage required for each processor in order for it to process the largest task assigned to it, the amount of communication bandwidth necessary to keep the communication time small in comparison to the processing time and the overhead of this communication to the references [METC76, BUX81, KIES81, LELA82]. Of course, we recognize that there will be higher communication delay if the tasks of the same job are assigned to processors far apart in the processor network. In Chapters 4 and 5, however, we assume that this communication cost is free (or as an approximation, that the average delay is incorporated into the processing requirement of each task); in Chapter 6 we bring the communication cost into the model.

The processors are identical in terms of their capabilities in processing speed and storage. Usually, the total number of processors is assumed to be a fixed constant, P. In some cases where we have enough processors to keep all executable tasks busy, we assume P is infinite.

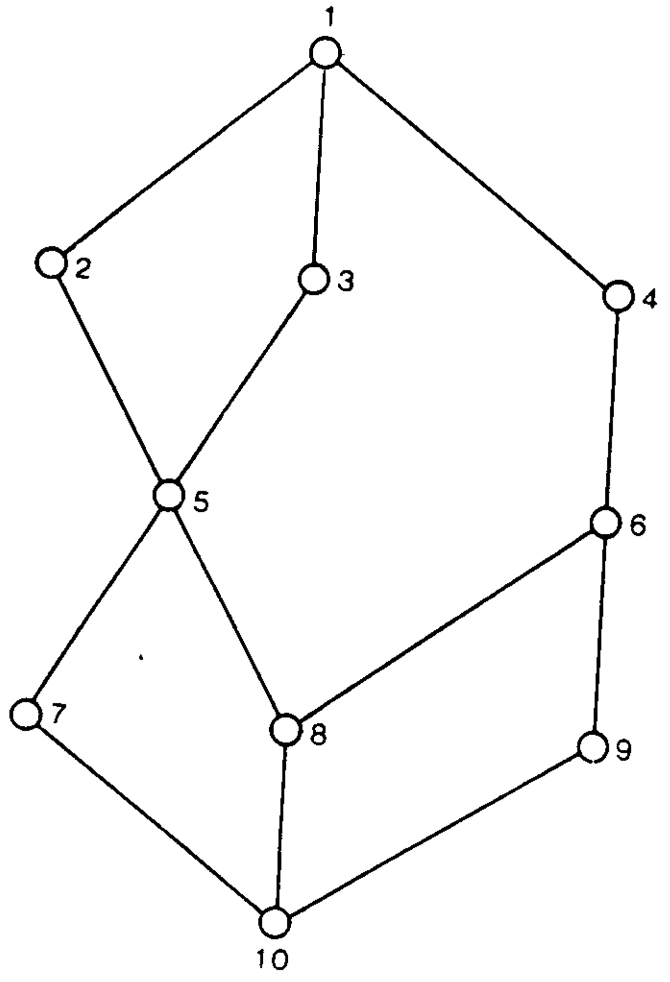
3.2 Process Graph

Each computer job is represented by a set of tasks, T, and a partial ordering of these tasks (given by a set of precedence relationships). We represent a task by a node and represent a precedence condition, where task i must be completed before task j, by a directed edge from i to j, denoted by (i,j). In the following, all directed edges point downward in the graphs; therefore, we will not put the arrows on the edges. In the following discussion, we distinguish neither between nodes and tasks nor between edges and precedence conditions.

We use the directed acyclic graph to represent the tasks in a computer job. Each node is a task that requires processing, and the edges (i,j) are used to prevent the starting of task j unless task i has been completed. Two tasks can be executed in parallel *if and only if* every predecessor of one task does not include the other task, and vice versa. The precedence relationship into a node is an 'and' type operator. Suppose $\{x_1,x_2, \cdots ,x_n\}$ are the nodes having edges into node x_0. Then x_0 may start execution only after all of the x_i, for i=1,2, $\cdots$ n, have completed execution. Without loss of generality, we assume there is only one starting node and one terminating node for each job. If there are several nodes with in-degrees of zero, we can add a new node with in-degree zero and which points to each of these nodes. Similarly, for several nodes with out-degrees of zero, we can add a new node with out-degree zero and then create new edges from all these nodes to this new node. The resulting directed acyclic graph is called a process graph. Figure 3.1 gives an example of a process graph.

Two parameters characterizing a process graph are its length and width. The *length* in a process graph, sometimes referred to as the total number of levels, is the number of tasks in the longest path between the starting and terminating nodes. We place a node j at level i if $i = \max_{u \epsilon U} \left[\delta_u\right]$ where δ_u is the number of tasks in path u from the initial node to node j, and U is the set of all paths from the initial node to node j. The *width* of a process graph is equal to $\max_i$ [*number of tasks in level i*]. In Figure 3.1, there are 5 levels and a width of 3.

Process graphs have a hierarchical structure so that each task in a process graph could, by itself, represent another process graph. This property can be useful in describing an operating system or a computer program. In an operating system, the nodes in a process graph could represent the jobs to be run. Within each job (or node), there is another structure of precedences which represents the execution order of the tasks. A similar situation exists in a program environment. Subroutines can be represented by a node in the process graph, and within each subroutine another process graph could exist with each node representing an executable block of statements. This hierarchical structure provides a useful tool in studying the scheduling problem at several different levels of complexity. When a more detailed schedule is required, a)



b) (1,2), (1,3), (1,4), (2,5), (3,5), (4,6), (5,7),
(5,8), (6,8), (6,9), (7,10), (8,10), (9,10)

Figure 3.1 a) a process graph
b) a list representing the precedence relations (i.e., directed edges) in a process graph in a).

each node of the process graph can be expanded into a finer process graph so that more detailed tasks could be individually scheduled. The opposite is also true; we can schedule the nodes, each of which represents a group of tasks in the original process graph.

Examples of process graphs representing some actual jobs include:

1. N x N matrix inversion

   After the initialization task, we can concurrently calculate the determinant and the N^2 cofactors. But each of the cofactors (say, i^{th} row and j^{th} column) is, in turn, computed from the (N-1)x(N-1) submatrix by eliminating the i^{th} row and j^{th} column of the original matrix. Therefore, each task may expand into more subtasks. This recursion stops when there is only a 2x2 matrix remaining in each of the subtasks. Figure 3.2 gives a typical process graph for the matrix inversion problem.

2. Shell Sort

   Shell Sort [KNUT73] sorts every h_t^{th} number in order. It then sorts h_{t-1}^{th}, h_{t-2}^{th}, $\cdots$ and h_1^{th} numbers in order at each iteration, respectively. The numbers h_t, h_{t-1}, $\cdots$ h_1 are intègers with $h_i > h_{i-1}$; so, the number of parallel sorts depends on how many numbers are to be sorted and the values of h_i, i=1, 2, ...,t. Figure 3.3 shows a typical process graph for the shell sort.

3. Quicksort

   Quicksort [KNUT73] uses the first number (the *key* number) in the list to divide the list into two parts. The left list contains all the numbers smaller than the key number; the right list contains all the numbers greater than the key number. These two lists can then be sorted independently by repeating the above procedure (i.e., use the first number of each list as the key number and sort each list into two more lists). This subdivision continues until there is only one or no task remaining in the subdivided list. Hence, a typical process graph might look like Figure 3.4. Because this sorting procedure depends on the value of the first number in the list, however, some unusual process graphs such as those in Figure 3.5 can result.

In later chapters, we will consider two cases of process graphs. In one case, the structure of the process graph for each job is fixed and known in advance. In the second case, each graph has a random structure; so, each job may have a different process graph.

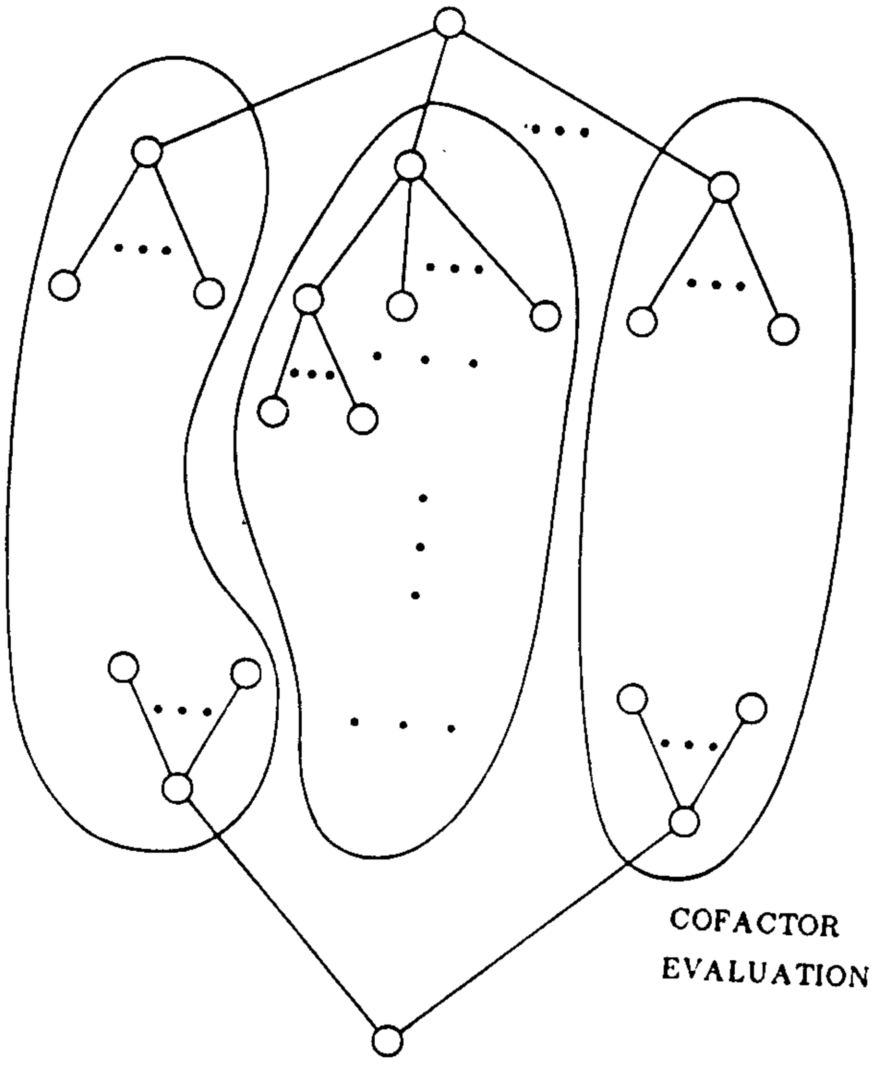


Figure 3.2 Matrix Inversion Process Graph

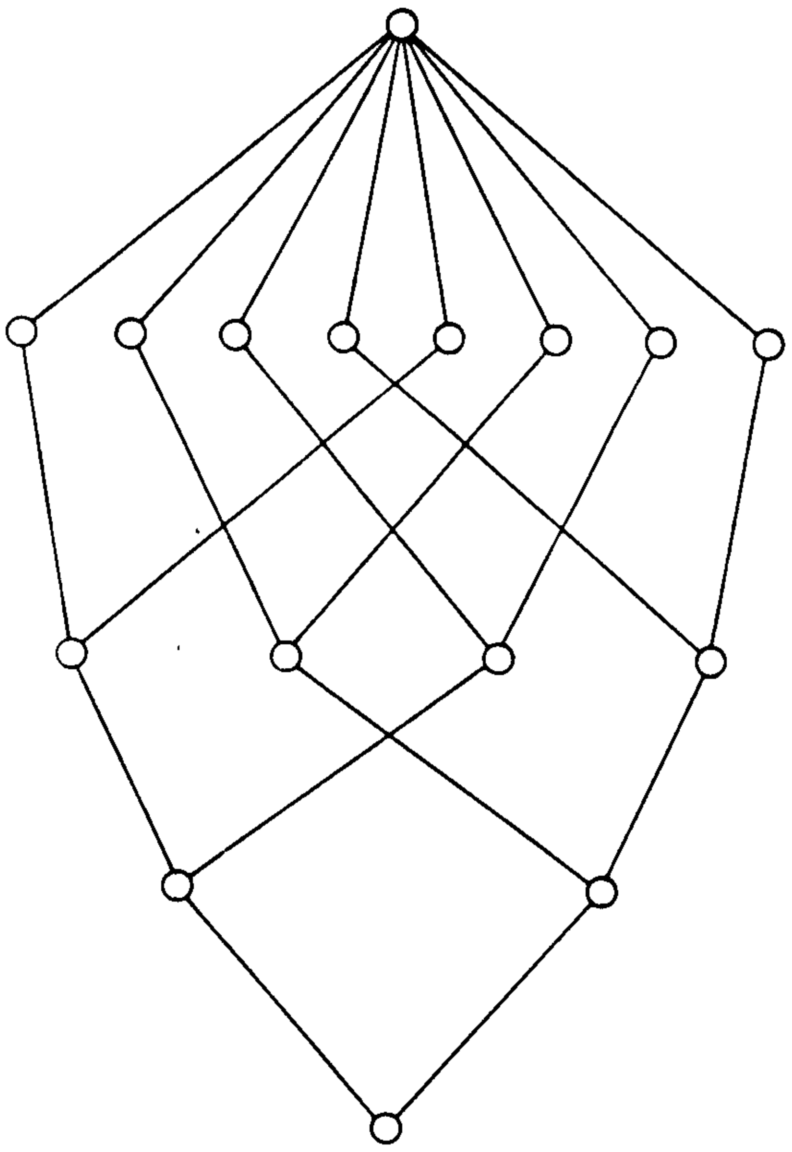


Figure 3.3 Shell Sort Process Graph

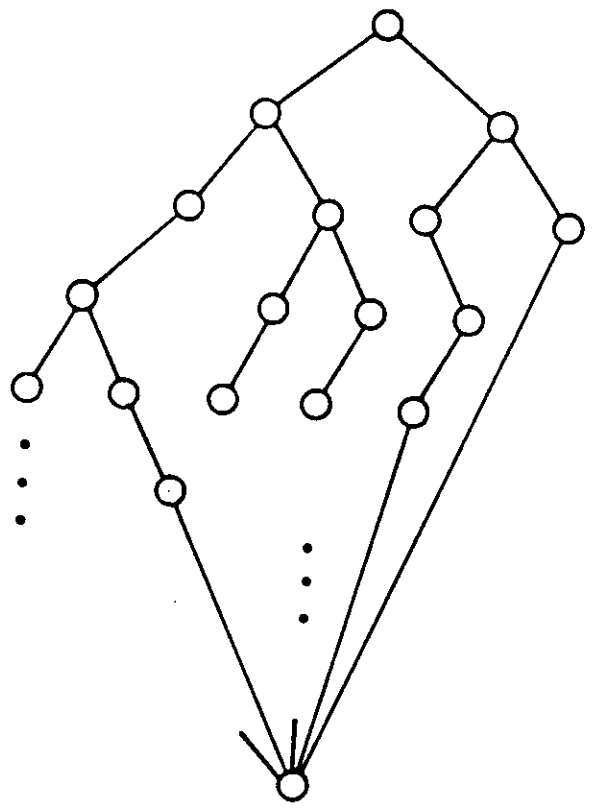


Figure 3.4 Quicksort Process Graph

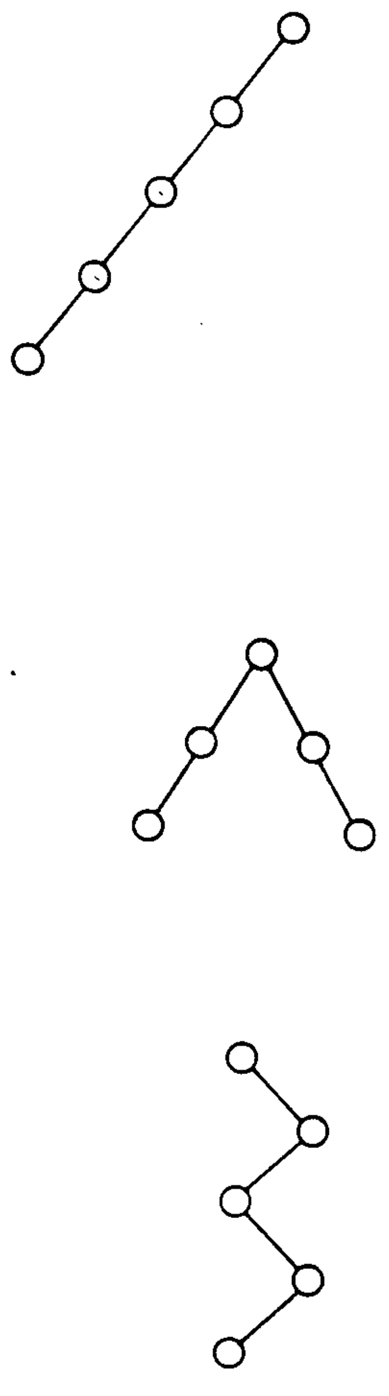


Figure 3.5 Unusual Quicksort Process Graph

3.3 Taxonomy

We have divided the task assignment and scheduling problems into sixteen cases. The parameters used for the classifications are:

1. Number of Jobs

   We can have a fixed number of jobs, k, at the start (time t=0), or we can allow jobs to arrive from a Poisson source with an average arrival rate of λ jobs/sec.

2. Types of Process Graph

   Each job in our problem can have an identical process graph, G, or each job can have a random process graph, G^*.

3. Processing Requirement of each Task

   Each task may have either a constant or a random processing requirement. In the latter case, the random task time for a fixed process graph can be sampled once at the beginning and used by all jobs or can be sampled each time the task is being processed. If the random sampling is done only once, it can be reduced to the constant processing requirement case by transforming each task into a chain of tasks, the length of each equal to the service time requirement and each task in the chain having one normalized time unit of service demand.

4. Number of Processors

   The number of processors is given by P. If there are enough processors so that each task can be processed whenever needed, then P can be treated as infinite.

With the terms defined above, we can summarize the sixteen cases with the taxonomy tree in Figure 3.6.

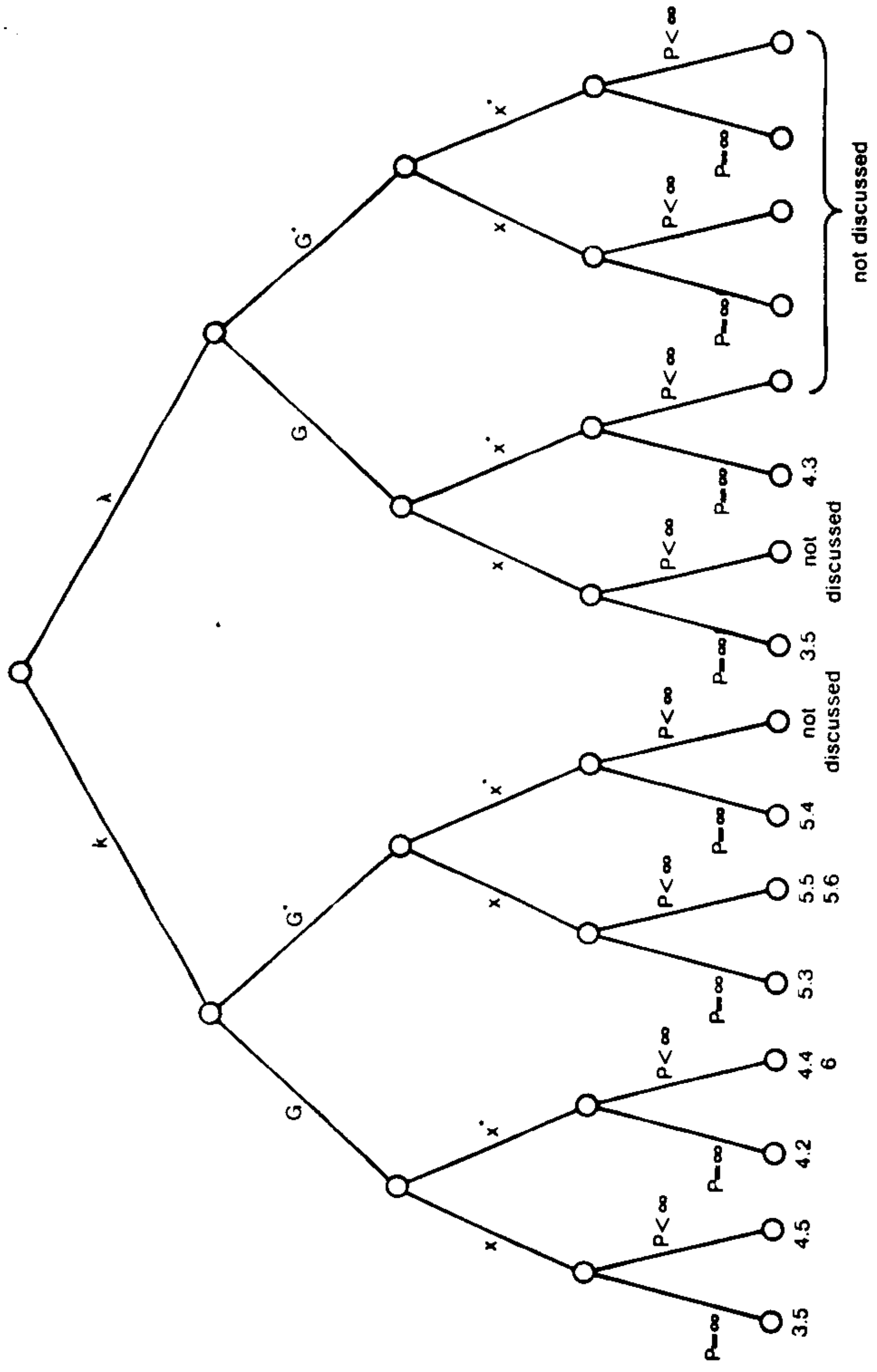


Figure 3.6 Taxonomy Tree

At the first level, we distinguish whether there is a fixed number (k) of jobs at time t=0 or whether jobs keep on arriving (at rate of λ) after time t=0. The second level deals with the type of process graph for each job. All jobs have either the same process graph (G) or random process graphs (G^*). The next level separates jobs with constant task time (x) from jobs with random task time (x^*) service demands. We also include a fourth level for any situation in which the number of processors, P, is greater than the maximum number of concurrent tasks that demand processors.

For all these cases we are interested in a parameter we call the *concurrency measure*, σ, which is defined to be the average system time required using P processors ($S(P)$) divided by the average system time required using one processor ($S(1)$), that is

$$\sigma = \frac{S(P)}{S(1)}$$

$\frac{1}{\sigma}$ measures just how much parallel processing is possible for a particular job, that is, $\frac{1}{\sigma}$ is the "speedup" factor. Note further that $\frac{1}{\sigma P}$ is the efficiency of each processor. In our notation, when $P = \infty$, we really mean that we have a large number of processors, say P^*, (i.e., maximum width of a process graph multiplied by the number of jobs) instead of an infinite number of processors; thus, the efficiency $\frac{1}{\sigma P}$ is not to be interpreted as zero for "$P = \infty$" but rather the efficiency is $\frac{1}{\sigma P^*}$.

$\frac{1}{\sigma}$, as defined above, measures the parallelism within a particular job. Furthermore, if k jobs, on the average, are running in a P-processor environment, then the speedup is $\frac{\bar{k}}{\sigma}$. For all cases where we have a fixed number of jobs at time zero and $P = \infty$, $\bar{k} = k$; the speedup over all k jobs is then $\frac{k}{\sigma}$. For all cases where jobs are arriving from a Poisson source (with a mean arrival time of $\frac{1}{\lambda}$), $\bar{k} = \lambda\ S(P)$; the speedup over a time span is $\frac{\lambda\ S(P)}{\sigma} = \lambda\ S(1)$.

A large speedup may appears as a good architecture, but the efficiency of the processors is also important. It is easy to get an efficiency of 1, but this system will be very slow. This tradeoff is studied when we discuss the issue of power (in Section 5.5). Note also that the maximum speedup is

$$\frac{1}{\sigma} = \frac{S(1)}{S(P)} \leq P$$

Some of the other parameters not considered are

- task interruptibility (preemption)

- homogeneous versus heterogeneous processors

Preemption is not considered since the communication overhead required for each preemption may be too large in a distributed processing environment. For simplicity, we have assumed homogeneous processors. If heterogeneous processors are used, all assignments must be optimized so that the speed of the processors with respect to each job must be considered.

3.4 Notation

Following is a partial list of notations used throughout the rest of this dissertation. Additional notation will be introduced as used.

k	number of jobs present at the beginning (time $t=0$)
λ	arrival rate of jobs from a Poisson source
G	fixed process graph
G^*	random process graph
x	constant processing time of a task
x^*	random processing requirement of a task
P	number of processors in the system
r,L	number of levels in a process graph
w	width of a process graph
N	total number of tasks in a process graph
S	average system time required to complete a job
σ	concurrency measure

3.5 Cases Studied

Many performance objectives are available:

- minimize the completion time of the slowest job;

- minimize the number of processors required;

- minimize the average system time;

- minimize the processor idling time or maximize the processor utilization.

These objectives can be used in combination or by themselves. In our work, we have chosen to use the minimization of the average system time (referred to in some literature as the flow time) as the performance objective.

Of the sixteen cases shown in Figure 3.6, eight of them have a limited number of processors ($P < \infty$). Therefore, these cases require scheduling of the tasks in each job. We defer most of the scheduling problem to the references cited in Chapter 2 and concentrate on the cases where we can assume that enough processors exist for all jobs and tasks that demand them.

Two of the cases, k, G, x, $P = \infty$ and λ, G, x, $P = \infty$, are very simple to analyze. All the parameters are deterministic; therefore, all the measures can be easily calculated. For both systems, we find the average system time, S (in fact, a constant for all jobs), by multiplying the number of levels, r, by the task processing time: $S = r\ x$. Hence, $\sigma = \frac{r}{N}$. For the arrival system, by Little's Result [LITT61], the average number of jobs in the system is $\bar{k} = \lambda\ S = \lambda\ r\ x$. Since we have an M/D/$\infty$ system, we also know the distribution of the number of jobs in the system, P_k, to be

$$P_k = \frac{(\lambda\ S)^k}{k!}\ e^{-\lambda\ S}$$

In all cases in which it can be assumed that $P = \infty$ and x is constant, the precedence relationships given in G are of no consequence except to ascertain the number of levels. For the constant service time cases, for every x units of service time, one level of the process graph will be completed regardless of whether a node on the previous level has precedence over this node. The assignment problem is also simplified by assigning all processors required to all tasks on the same level of the process graph for x units of process time. Because we have enough processors to keep any number of jobs active concurrently, the number of jobs is irrelevant. Any job, either k jobs at time $t = 0$ or those arriving from a Poisson source, will spent $r\ x$ units of time in the system before departing. Therefore, we need to study just one job in order to find all the system parameters.

In Chapter 4 we study the cases i) k, G, x^*, $P = \infty$, ii) k, G, x, $P < \infty$, iii) k, G, x^*, $P < \infty$ and iv) λ, G, x^*, $P = \infty$. In Chapter 5 we concentrate on random process graphs for cases i) k, G^*, x, $P{=}\infty$, ii) k, G^*, x^* $P{=}\infty$, and iii) k, G^*, x, $P{<}\infty$. In Chapter 6 we study the communication overhead with the case k, G, x^*, $P{<}\infty$.

In the taxonomy tree of Figure 3.6, the section associated with a particular case is shown on the bottom. The other cases are left for future research.

CHAPTER 4
Fixed Process Graphs

4.1 Introduction

In this chapter, we explore cases where the process graph is fixed (i.e., given). The service time for each task is random in Section 4.2, 4.3 and 4.4 while in Section 4.5 we assume it is constant. The number of processors is assumed to be infinite; so the results obtained are independent of the number of jobs. In Section 4.2.1 we first obtain the average system time for the case of exponentially distributed service time for each task. The process graph is first converted into a Markov chain; the equilibrium state probabilities of each state in the chain are then obtained. From the average system time we find the concurrency measure for a specific process graph. We use bounds on the average system time to get an approximation of the concurrency measure in those cases where the exact concurrency measure becomes difficult. Section 4.2.2 describes how the bounds are obtained. In Section 4.3 we consider the arrivals of jobs to the system instead of a fixed number of jobs. In Section 4.4 we consider a finite number of processors, and using Stochastic Petri Net theory and the notion of power, we find the optimum number of processors that should be assigned to each job. Section 4.5 deals with the assignment of tasks to processors. We look at the ratio of the average system time when the best scheduling algorithm is used versus that when the worst scheduling algorithm is used.

With either the exact concurrency measure or bounds on the concurrency measure, we have characterized a process graph. The average execution time, the average width and the speed-up that is possible for this process graph can all be derived from the concurrency measure.

4.2 Fixed Number of Jobs (k, G, x^{*}, $P = \infty$)

4.2.1 The Exact Average System Time

In order to find the average system time of a process graph, we must be able to compute the average concurrency of the tasks. Towsley [TOWS78] introduced a model of parallel processing for CPU and I/O overlapping. In this model, after a CPU task terminates (say task CPU_1), it initiates another CPU task along with an I/O task (i.e., CPU_2 and I/O) – see Figure 4.1a.

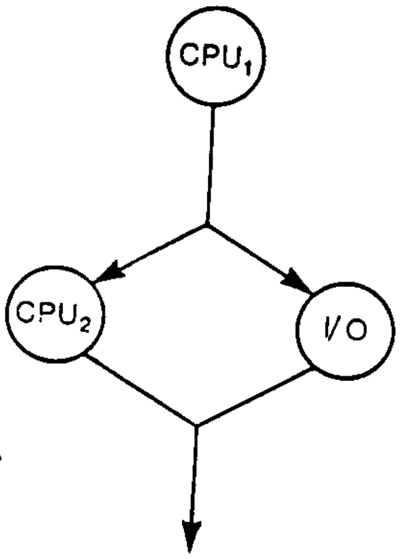


Figure 4.1a CPU and I/O Overlap

If we now represent this system behavior by a Markov state transition diagram (Figure 4.1b), a job may be in any of four states:

1) CPU_1	the CPU is executing task CPU_1.
2) CPU_2	the CPU is executing task CPU_2 alone.
3) I/O	the I/O task is executing.
4) CPU_2-I/O	the CPU is executing task CPU_2 in parallel with the execution of the I/O task.

The time spent in each state is selected from an exponential distribution with the mean service time for CPU_1, CPU_2, and I/O equal to $\frac{1}{\mu_1}$, $\frac{1}{\mu_2}$, and $\frac{1}{\lambda}$, respectively.

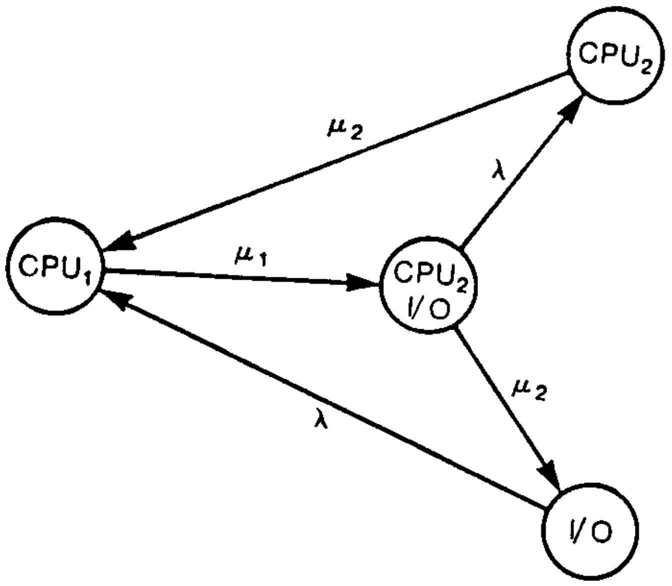


Figure 4.1b Markovian State Transition Diagram

In this section, we use Towsley's approach in our concurrency problem by converting process graphs into Markovian state transition diagrams. Two methods of obtaining the average system time are then discussed.

4.2.1.1 Converting the Process Graph into a Markovian State Transition Diagram

A Markovian state transition diagram, $M(G)$, is generated for process graph G where each state in the Markov chain represents a specific set of tasks in G that can be executed in parallel. Let C_α represent a state in the Markov state transition diagram where α is the set of tasks that are executed concurrently. Also, let $|\alpha|$ represent the number of tasks in the set α.

The chain starts with state C_f where f is the initial task in G. For each state C_α in the chain, it will go to $|\alpha|$ other states, each branch corresponding to the termination of one of the tasks in α. The state $C_{\alpha'}$ at the end of one of these branches has the set of active tasks $\{\alpha'\}$, where α' includes the tasks in α minus the completed task plus the activation of several other tasks if any, due to the termination of this task. The exact algorithm is given in Algorithm

CPM (i.e., Convert Process graph to Markov chain) in Figure 4.2, where the procedure for obtaining the Markov Chain from a process graph is described. Figures 4.3a and 4.3b are examples of this algorithm.

ALGORITHM CPM

1. For the initial task i, we create an initial state with one active task i.
   Mark this state 'unlabeled.'

2. Select one of the unlabeled states C_ω, and mark it 'labeled.'
   Suppose there are x active tasks, $\phi_1, \phi_2, \cdots, \phi_x$, in this state C_ω.
   For each ϕ_i create a branch with the branch label of ϕ_i;
   this label corresponds to the termination of task ϕ_i.
   If we traverse back from state C_ω to the initial tasks,
   the tasks on the branches of this path form the set of completed tasks.
   By adding ϕ_i to this set, we can check the process graph for new tasks, if any,
   which become active; call this set β_i.
   The branch with label ϕ_i goes into state $C_{\omega'}$ where

   $$\omega' = \omega - \phi_i + \left\{\beta_i\right\}.$$

   If $C_{\omega'}$ does not exist, we create this state and mark it 'unlabeled.'

3. If any state is not marked 'labeled,' go to step 2.

4. Create a branch from the terminating state to the initial state;
   stop.

Figure 4.2 Algorithm CPM

We find that there are as many levels in $M(G)$ as there are tasks in the process graph G. This also equals the number of states visited in $M(G)$ before a job cycles back to the first state C_f in the Markovian state transition diagram.

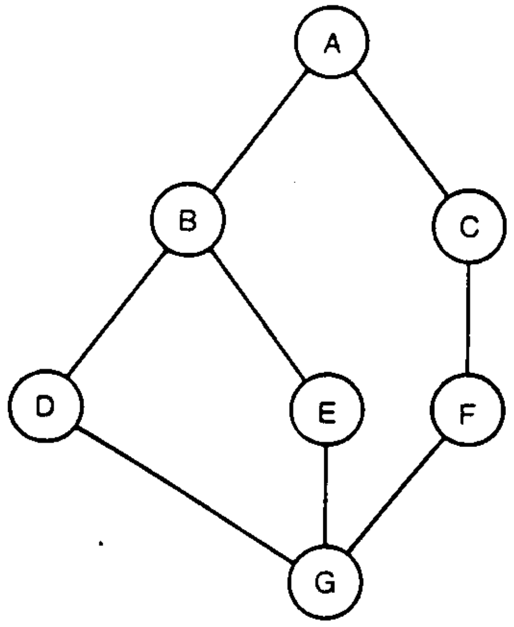


Figure 4.3a Process Graph

4.2.1.2 The Average System Time

For a state C_α, the rate of leaving this state is $\mu_1+\mu_2+\cdots+\mu_{|\alpha|}$, where $\frac{1}{\mu_i}$ is the mean of the exponential service time of task $i\epsilon\alpha$. Therefore, the mean time spent in this state is

$$\frac{1}{\sum_{i=1}^{|\alpha|} \mu_i}$$

Let us now assume that $\mu_i = \mu$ for all tasks i. Therefore, the mean time a job stays in state C_α is $\frac{1}{|\alpha|\mu}$.

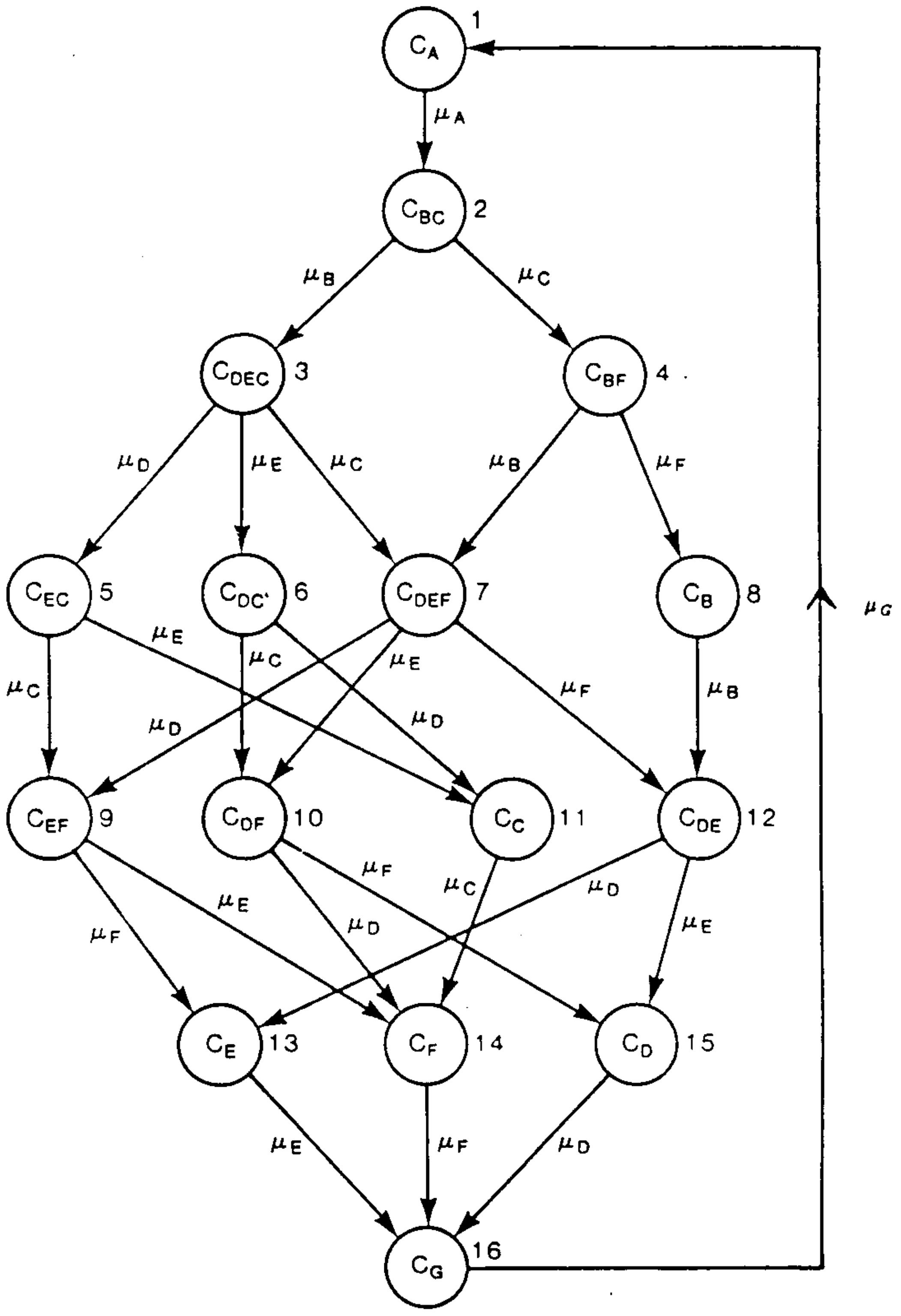


Figure 4.3b Markovian State Transition Diagram

Due to the memoryless property of the exponential service time distribution, each task in C_α has the same mean service time regardless of whether a specific task had been processed in another state $C_{\alpha'}$. Hence,

$$Prob\left[task\ i\ completes\ first \mid i \epsilon \alpha\right] = \frac{1}{|\alpha|}$$

for all $i \epsilon \alpha$.

Starting from the initial state, there are many paths a job can traverse before reaching the terminating state in a Markovian state transition diagram. Since we know the probability of traversing each branch, the probability that a specific path has been taken can be calculated. Suppose the path taken proceeds through the following states:

$$C_{\alpha_1}, C_{\alpha_2}, \cdots, C_{\alpha_N}.$$

The probability of taking this path is $\prod_{i=1}^{N} \frac{1}{|\alpha_i|}$.

Suppose there are r different paths from the initial state to the terminating state in the Markov state transition diagram. If it takes an average of T_i units of time to complete path i with probability p_i this path is chosen, then

$$S(P) = \sum_{i=1}^{r} T_i\, p_i \qquad (4.1)$$

The total number of paths from the initial state to the terminating state is enumerable. By summing the product of the total average time spent in each state in a path and the probability of taking this path, we are then able to find the average system time of the process graph represented by this Markovian transition state diagram.

Take the example shown in Figure 4.3b, the average time for path

$$C_A, C_{BC}, C_{DEC}, C_{DC}, C_{DF}, C_F, C_G$$

including return to C_A (i.e., a cycle) is

$$\frac{1}{\mu}\left[1+\frac{1}{2}+\frac{1}{3}+\frac{1}{2}+\frac{1}{2}+1+1\right] = \frac{1}{\mu}\frac{29}{6}$$

and the probability of taking this path is $\frac{1}{2}\frac{1}{3}\frac{1}{2}\frac{1}{2} = \frac{1}{24}$. Summing over the product of average path time and the probability of taking this path over all possible paths, we obtain an average system time of $\frac{1}{\mu}5.0556$.

From a simulation of this system, we obtain a value of $\frac{1}{\mu}$ 5.09 for the average system time, a result which is in very close agreement with the predicted value.

From the above calculation, we see that the average system time is greater than $\frac{4}{\mu}$ which is the number of levels in the process graph (Figure 4.3a) multiplied by the average service time of a task. The reason for this difference is that task G must wait for the completion of its 3 predecessor tasks (tasks D, E, and F) before it may start. Thus the time to process nodes D, E, and F in parallel (even assuming that they begin to be processed at the same point in time) is greater than $\frac{1}{\mu}$ since we must wiat for the slowest of the three to complete (and this will exceed the average task time for each -- see Equation 4.4 below). We are seeing the cost (in increased system time) due to the dependencies among the paths from initial node to terminating node.

4.2.1.3 The Concurrency Measure

The concurrency measure can be calculated from the average system time as

$$\sigma = \frac{S(P)}{S(1)}$$

Substituting Equation (4.1) into the concurrency expression, we get,

$$\sigma = \frac{\sum_{i=1}^{r} T_i p_i}{\frac{N}{\mu}}$$

We can find σ by another method. We have transformed the process graph into a Markov state transition diagram. If, in addition, we have a branch going from the terminating state back up to the initial state, we then have a discrete state continuous time ergodic Markov chain. The equilibrium state probabilities can be solved by the balance equations, which equate the rate into a state to the rate out of the same state. In addition, we need $\sum_i \pi_i = 1$ where π_i is the equilibrium probability at state i. Even though there might be a large number of states in the Markov chain, due to the characteristics of process graphs, the balance equations will form a lower triangular matrix, and it is easy to express all the state equilibrium probabilities in terms of π_1. Then, by using the $\sum_i \pi_i = 1$ equation, we find π_1 which, in turn, gives us all the equilibrium state probabilities. Figure 4.4 gives an example of the balance equations in matrix form for the process graph given in Figure 4.3b, assuming $\mu_i = \mu$ for all i.

$$\begin{pmatrix}
\mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \mu & \mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \mu & 0 & \mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \mu & \mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \mu & \mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \mu & \mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu & 0 & 0 & \mu & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu & \mu & \mu & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu & 0 & \mu & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu & \mu & \mu
\end{pmatrix}
\begin{pmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \\ \pi_4 \\ \pi_5 \\ \pi_6 \\ \pi_7 \\ \pi_8 \\ \pi_9 \\ \pi_{10} \\ \pi_{11} \\ \pi_{12} \\ \pi_{13} \\ \pi_{14} \\ \pi_{15} \end{pmatrix}
=
\begin{pmatrix} 2\,\mu\,\pi_2 \\ 3\,\mu\,\pi_3 \\ 2\,\mu\,\pi_4 \\ 2\,\mu\,\pi_5 \\ 2\,\mu\,\pi_6 \\ 3\,\mu\,\pi_7 \\ \mu\,\pi_8 \\ 2\,\mu\,\pi_9 \\ 2\,\mu\,\pi_{10} \\ \mu\,\pi_{11} \\ 2\,\mu\,\pi_{12} \\ \mu\,\pi_{13} \\ \mu\,\pi_{14} \\ \mu\,\pi_{15} \\ \mu\,\pi_{16} \end{pmatrix}$$

Figure 4.4 Balance Equation

Let us denote the matrix multiplication in Figure 4.4 by

$$A\,\Pi = B\,\Pi'$$

A is a $K - 1 \times K - 1$ square matrix (where K is the number of states), Π is a column vector of π_i's ($i = 1, 2, \cdots, K - 1$). B is a row vector of the rate out of states $2, 3, \cdots, K - 1$, and Π' is a column vector of π_i 's ($i = 2, 3, \cdots, K$). A 'μ' in the i^{th} row and j^{th} column in matrix A represents an edge with a label μ from state j into state $(i + 1)$ in Figure 4.3b. Therefore the matrix product of the i^{th} row of A by Π results in the rate going into the state $(i + 1)$. The i^{th} entry in B represents the number of edges leaving state $(i + 1)$ multiplied by μ. Thus, multiplying the i^{th} entry in B by the i^{th} entry in Π' results in the rate out of state $(i + 1)$. Hence, $A\,\Pi = B\,\Pi'$ equates the rate in and rate out of state i, for $2 \leq i \leq 16$. For example, the 6th row of $A\,\Pi$ contains $\mu\,\pi_3 + \mu\,\pi_4$ which is the rate into state 7, and the 6th row of $B\,\Pi'$ contains $3\,\mu\,\pi_7$ which is the rate out of state 7.

Once we have found the π_i, we can proceed as follows to obtain the concurrency measure.

$$\sigma = \frac{S(P)}{S(1)} = \frac{\dfrac{N}{\sum_k o_k \pi_k} \dfrac{1}{\mu}}{\dfrac{N}{\mu}}$$

where o_k is the number of active tasks in state k and $\sum_k o_k \pi_k$ is the average number of tasks being executed over the entire execution period. Hence,

$$\sigma = \frac{1}{\sum_k o_k \pi_k} \tag{4.2}$$

This is the main result of this section.

From renewal theory, we also know that $S(P)$ is the mean recurrence time. Thus, $S(P) = \frac{1}{\pi_j \mu}$ where π_j is the equilibrium probability of either the initial or the terminating state. Hence,

$$\sigma = \frac{\dfrac{1}{\pi_1 \mu}}{\dfrac{N}{\mu}} = \frac{1}{\pi_1 N}$$

A simple example is given next. Figure 4.5 is a process graph with its Markov chain shown in Figure 4.6. The set of balance equations are

$$\mu \pi_5 = \mu \pi_1$$

$$\mu \pi_1 = 2\mu \pi_2$$

$$\mu \pi_2 = \mu \pi_3 = \mu \pi_4$$

$$\sum_{i=1}^{5} \pi_i = 1$$

The solution is

$$\pi_1 = \pi_5 = \frac{2}{7}$$

and

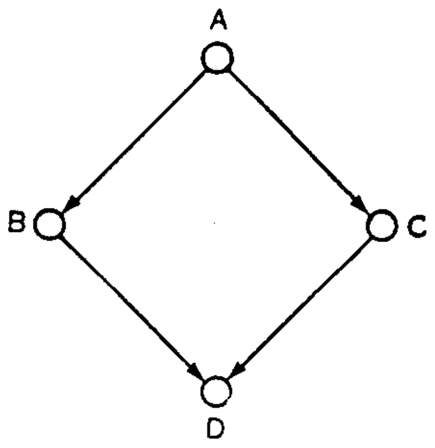


Figure 4.5 Process Graph

$$\pi_2 = \pi_3 = \pi_4 = \frac{1}{7}$$

Hence,

$$\sum_{k=1}^{5} o_k \pi_k = \frac{8}{7}$$

$$or\ \sigma = \frac{7}{8}$$

There are two paths from state 1 to state 5, each with equal probability of being chosen. The average path time for both paths is $\left[1+\frac{1}{2}+1+1\right]\frac{1}{\mu}$. Hence,

$$\frac{S(P)}{S(1)} = \frac{\frac{7}{2}\frac{1}{\mu}}{4\frac{1}{\mu}} = \frac{7}{8}$$

which agrees with the σ calculated above.

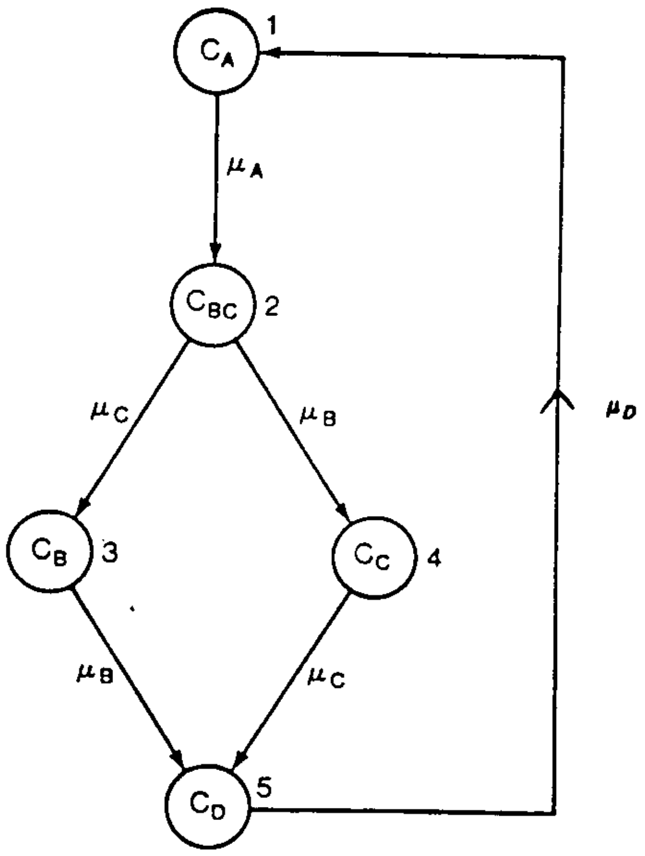


Figure 4.6 Markov Chain

The small speedup ($\frac{1}{\sigma} = \frac{8}{7}$) is due to the serial nature of the process graph in this example. Only two of the four tasks may be processed in parallel and only for the duration of $t = \min[\, t_1 , t_2 \,]$ where t_i, $i = 1, 2$, is the random processing time of one of the tasks that can be executed in parallel.

4.2.2 Bounds on the Average System Time

In Section 4.2.1, we found the exact concurrency measure (Equation 4.2) for a fixed process graph with random task times. Although the algorithm used in obtaining σ is not difficult to carry out, it is cumbersome to either calculate $S(P)$ by going through all the paths in the Markovian state transition diagram or to solve for the equilibrium state probabilities from a set of balance equations derived from the Markov chain. If the exact concurrency measure is not required, we may use upper and lower bounds as substitute measurements for the concurrency; they are usually much more easily obtained than the exact solution.

In order to find an upper bound, we "synchronize" the execution at each level by forcing all the tasks in the next level to wait for the slowest task in the current level to complete before they all start executing. We call the time between the synchronization of two neighboring levels the "forced synchronization time" (FST). If we sum up the FST at each level of a process graph, an upper bound for $S(P)$ is obtained. For a lower bound, we just find the average time required to execute the tasks in the longest path from the initial node to the terminating node. Since this is the minimum time required for any job with this process graph, we have a lower bound. In the following sections, we describe the exact procedures for finding these bounds.

4.2.2.1 Blocking Time of Predecessor Tasks

In this section, we find the average time contributed by the "blocking nodes." Blocking nodes of a specific node i in the process graph are the nodes that have precedence relationships into node i. Since a task cannot start execution until all of its predecessors have been completed, we would like to find the average time required for the completion of all its predecessors in the previous level (assuming they all started at the same time).

Each node, i, in G has several precedences entering it and several precedences exiting it. During the processing of a task, out-degrees do not influence the completion time of this task, but in-degrees do. Suppose there are n precedences entering this node; the task can't start execution until all n tasks are done. In other words, assuming that all these n tasks are begun at the same time, the effective execution time of these n tasks with respect to node i is equivalent to the max(t_1, t_2, ... , t_n), where t_j, j=1,2, $\cdots$ n is the random service time of task j. We find the probability distribution function of this max as follows:

$F^n_{\max}(t) = Prob\ [completion\ time\ of\ n\ tasks \leq t]$

$$= Prob\,[t_1 \le t] * Prob\,[t_2 \le t] * \cdots * Prob\,[t_n \le t]$$

$$= [F(t)]^n$$

where $F(t)$ is the probability distribution function for the service time of one task. The equality is due to the independence of service times of the tasks. From probability theory, we derive the expected time for finishing n tasks from

$E\,[\ \textit{completion time of n tasks}\ ]$

$$= \int_0^\infty \left[1 - F_{\max}^n(t) \right] dt$$

$$= \int_0^\infty \left[1 - F(t)^n \right] dt \qquad (4.3)$$

Since $F(t)^n \le F(t)$, E [completion time of n tasks] $\ge$ E [completion time of one task]. If the service time is exponentially distributed with an average service time of $1/\mu$, then Equation (4.3) becomes

$$E \left[\begin{array}{l} \textit{completion time of n tasks with} \\ \textit{exponentially distributed service times} \end{array} \right]$$

$$= \int_0^\infty \left[1 - (1 - e^{-\mu t})^n \right] dt$$

$$= \frac{1}{\mu} \sum_{j=1}^{n} \frac{1}{j} \qquad (4.4)$$

Note that for $n >> 1$, Equation (4.4) is approximately $\dfrac{\ln n + \Phi}{\mu}$ (where Φ is Euler's constant $= .57721...$) Equation (4.4) can also be obtained from the following equivalent queueing system:

> A service center with n servers, each server having an exponential service time distribution with mean of $1/\mu$;
>
> No waiting room allowed in the system and servers starting execution when there are n

customers in the system;

Once the servers start execution, no new arrivals allowed to replace the departed customers.

The time, S, to complete all n customers measured from the start of execution is

$$S = \frac{1}{n\mu} + \frac{1}{(n-1)\mu} + \cdots + \frac{1}{\mu}$$

$$= \frac{1}{\mu} \sum_{j=1}^{n} \frac{1}{j}$$

The equivalence can be shown as follows:

The n customers in the queueing system are equivalent to the n blocking nodes;

One server becomes inactive after each customer departs from the queueing system and this is the same as the completion of one blocking task;

The service time of each customer is exponentially distributed with a mean service time of $\frac{1}{\mu}$, and the time required to complete a blocking task is also exponentially distributed with the same mean service time of $\frac{1}{\mu}$.

Hence, the average time required to empty this n-server queueing system is the same as the average time needed to complete the n blocking tasks if all of them start execution at the same time.

From equation (4.4), we note that without the blocking effect, each task of a process graph requires an average service time of $\frac{1}{\mu}$. However, with the blocking effect, the average time to complete a task becomes approximately $\frac{\ln n}{\mu}$ where n is the number of tasks blocking this task. Thus $\sigma = \frac{S(n)}{n} = \frac{\ln n}{n}$ giving a speedup of $\frac{n}{\ln n}$; this falls short of the maximum possible speedup which is equal to n. The reason for this poor speedup is clearly the blocking effect.

Since we know the in-degrees of each node in the process graph, the average time required to wait for the completion of all the precedence tasks for a specific task, i, can be calculated from Equation (4.4).

4.2.2.2 Bounds for Structured Process Graphs

We will first study "structured process graphs," which are defined as having the following properties:

1. All sons of a node, i, must merge back into one single node, j, before reaching the terminating node, and

2. only node i can have a direct precedence relationship into each son of node i; no other nodes may have direct precedence relationships into sons of node i.

With the above properties, each son may be replaced with a set of tasks with the same properties. A structured program is a good analogy of this process graph. In a program, which must be entered at one specific point and exited at another, several parallel blocks of code may be executed simultaneously, but each block must be entered and exited from the specific points. Property 2 above states that no 'GOTO' statements may direct the execution out of or into a block of code. Figure 4.7a) shows a structured process graph, while the edge e in Figure 4.7b) violates the property of a structured process graph.

Within each structured process graph, we can divide the tasks (other than the starting and terminating nodes) into several mutually exclusive sets, where each set of nodes, together with the starting and terminating nodes, forms a structured sub-process graph. Property 2 prevents the precedences from a node in one set of tasks leading into a node in another set. We call each set of the tasks a 'group-path,' and we let m denote the number of group-paths in a structured process graph. Algorithm GP below describes a method for finding all group-paths for a given structured process graph.

This algorithm begins at the starting node of G and, by keeping track of the nodes diverging out of each task, looks for nodes that are in a single group-path. If tasks eventually merge back after diverging out of a single node, they are considered as one group-path. If tasks do not merge back before the terminating node, they will be considered as separate group-paths.

By substituting other tasks for the starting and terminating nodes in Algorithm GP, we can find the sub-group-paths within the process graph.

Nodes within the same group-path have the same PATHNUMBER in the algorithm below:

Algorithm GP:

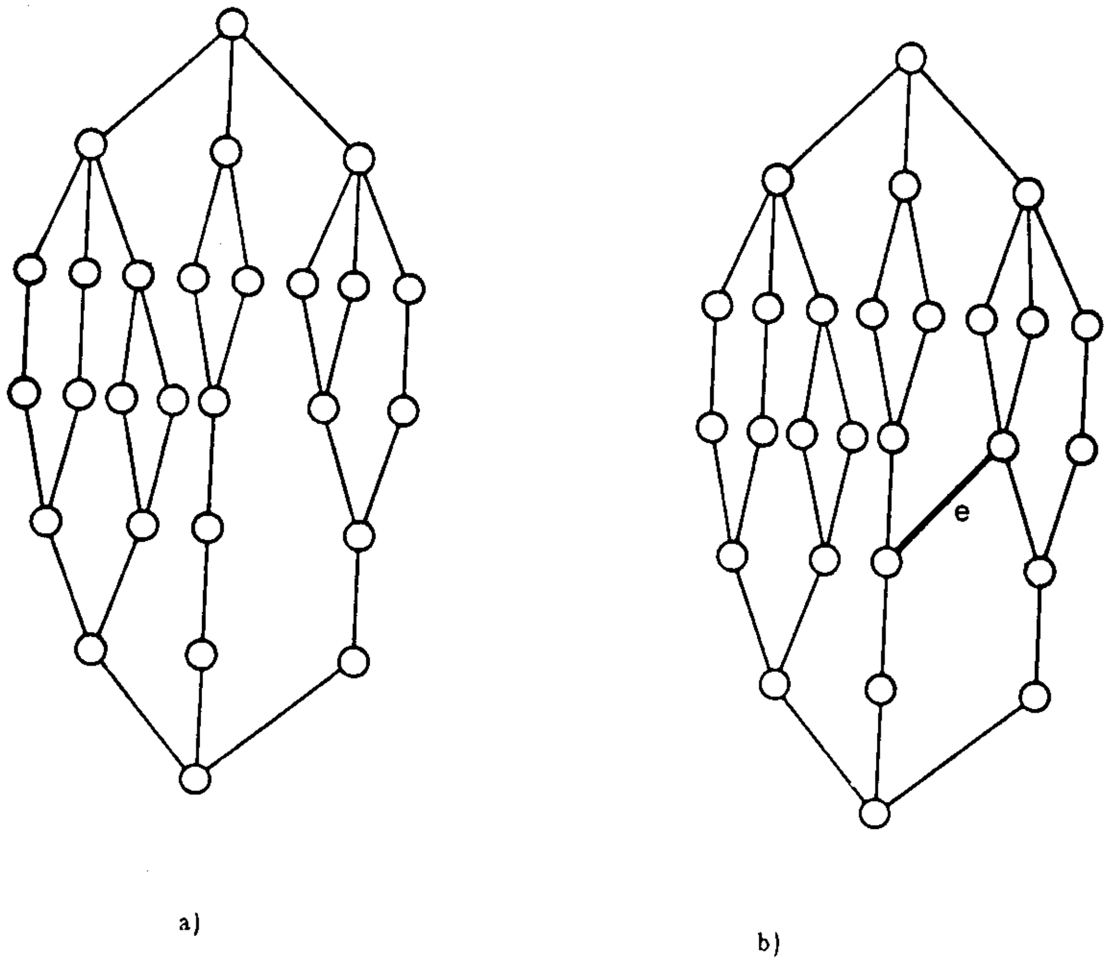


Figure 4.7 a) A structured process graph b) A non-structured process graph

```
STEP0   Mark all nodes 'NEW.'
        PATHNUMBER ← 1
        PATH($v$) ← 0 for all nodes $v$
        STACK ← 'empty'
        STORAGE ← 'empty'
        $v \leftarrow$ the starting node of $G$.

STEP1   Mark $v$ 'OLD.'

        Store the sons of $v$ on the STACK.

STEP2   Select a task from the top of STACK.

        Call this task $v$.

        If STACK is empty, STOP.

STEP3   If $v$ is 'NEW'.

                Put $v$ in STORAGE.

                Mark it 'OLD.'

                Put all sons of $v$ on STACK.

                GO TO STEP5.

STEP4   If $v$ is 'OLD.'

                PATH($w$) $\leftarrow$ PATH($v$) for all nodes $w$ in STORAGE

                STORAGE $\leftarrow$ 'empty'

                GO TO STEP2.

STEP5   If $v$ is the terminating node,

                mark it 'NEW' again (since the terminating node
                        is not considered to be in any one
                        particular group-path)

                PATH($w$) $\leftarrow$ PATHNUMBER for all nodes $w$ in STORAGE

                STORAGE $\leftarrow$ 'empty'

                PATHNUMBER $\leftarrow$ PATHNUMBER + 1.

STEP6   GO TO STEP2.
```

For m=1, we know that there does not exist any individual path that is not coupled with some other part of the G. Instead, many precedence relationships exist between two neighboring levels. Because of this, the line of active tasks in G is roughly the same as the levels of G since blocking prevents any tasks from becoming active if they are several levels ahead of the active tasks.

From this analysis, we can immediately find an upper bound to the average system time of G. If we force the execution to complete one level of G at a time, no tasks in the following level are allowed to start, even if there are no tasks on the current level to block this task. The average service time for each level is the average time required to process the slowest task, i.e., the task with the largest number of blocking nodes from the previous level. We introduce a new parameter $d_{i\max}$, which gives the largest in-degree per task for all tasks on level i of G. From our definition of the process graph, we have $d_{1\max}=0$ and $d_{2\max}=1$ for all G.

Theorem 4.1

Given a process graph, G, which has one group path $(m=1)$, r levels and $\{d_{i\max} \mid i=1,2,3 \cdots r\}$, S_{UB}, an upper bound for the average system time of G, is equal to the sum of the average times required to process the node with the largest in-degree at each level.

$$S_{UB} = \sum_{i=1}^{r} \begin{bmatrix} \textit{average time required to process} \\ \textit{a node with in-degree of } d_{i\max} \end{bmatrix}$$

$$= \frac{1}{\mu} \sum_{i=1}^{r} \sum_{j=1}^{d_{i\max}} \frac{1}{j}$$

If $d_{i\max} = 0$, the average processing time is defined to be the average service time of one task. (The sum on j can be approximated by $\ln d_{i\max}$ when $d_{i\max} >> 1$).

Proof:

$d_{i\max}$ is the largest in-degree for level i of G. If we sum the average times required to process tasks t_i for $i=1,2, \cdots ,r$ and the in-degree of t_i is $d_{i\max}$, the resulting average time equals that of a process graph with a path, p, from the starting to the terminating nodes, where each node on this path at i^{th} level has an in-degree equal to $d_{i\max}$. If any one of the $\{d_{i\max}\}$ is not on the same path, we must show that the resulting average system time is no greater than S_{UB}

Suppose the maximum in-degree node on level j of G is on path p' rather than path p (see Figure 4.8). Let i_1 denote the node in the j^{th} level for path p, i_2 the node in the $j+1^{th}$ level for path p and i_3 the node in the j^{th} level for path p'. Since the number of blocking nodes for node i_1 (that is d_{i_1}), is less than or equal to the number of blocking nodes for node i_3 (that is d_{i_3}), the average time required to complete node i_1 is smaller than or equal to the average time

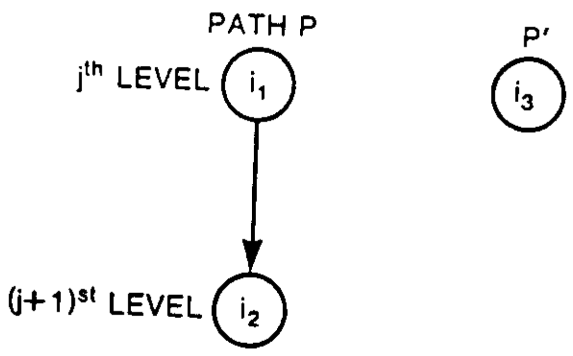


Figure 4.8 Maximum In-degree Nodes

required to complete node i_3:

$$\frac{1}{\mu}\sum_{i=1}^{d_{i_1}}\frac{1}{i} \le \frac{1}{\mu}\sum_{i=1}^{d_{i_3}}\frac{1}{i} \qquad \text{for } d_{i_1} \le d_{i_3} = d_{i\,\max}$$

Therefore, the average time to complete path p with node i_1 in the j^{th} level instead of node i_3 is smaller than or equal to S_{UB}. Similarly, if the nodes i_{j_1}, i_{j_2}, $\cdots$, and i_{j_α} on levels j_1, j_2, $\cdots$, and j_α of path p are not the maximum in-degree nodes, then

$$\frac{1}{\mu}\sum_{l=1}^{\alpha}\sum_{r=1}^{d_{i_{j_l}}}\frac{1}{r} \le \frac{1}{\mu}\sum_{l=1}^{\alpha}\sum_{r=1}^{d_{j_l\,\max}}\frac{1}{r}$$

because $d_{i_{j_l}} \le d_{j_l\,\max}$ for $l = 1, 2, \cdots , \alpha$.

Hence, S_{UB} is an upper bound for the average system time of G.

III

A simple lower bound on the average system time is just the average time required by a task multiplied by the number of levels.

Theorem 4.2

A lower bound of the average system time of a process graph G, given r levels, is the average service time of one task multiplied by r.

$$S_{LB} = r \frac{1}{\mu}$$

Proof:

Since each level is defined as having at least one task in it, the minimum time required to process one level is the service time of one task. For r levels, the minimum average system time cannot be lower than the average service time of one task multiplied by r.

|||

The upper and lower bounds obtained in the above two theorems are for different process graphs with specific sets of $\{d_{i\max}|\forall i\}$; however, each set of $\{d_{i\max}|\forall i\}$ can represent a number of different process graphs. Indeed, if we do not limit the number of tasks per process graph, there could be an infinite number of process graphs generated from each $\{d_{i\max}|\forall i\}$. In the proofs of both theorems, the number of tasks and the structure of the task graphs were not used. In other words, provided the same sets of $\{d_{i\max}|\forall i\}$ in both theorems, we have a class of process graphs with the same upper and lower bounds.

For a structured process graph with $m=1$ group-path, the actual progress of active tasks sometimes closely follows the physical levels of the process graph. This is caused by the precedence relationships between adjacent levels (which forced the synchronization at each level). Hence, the average system time is often close to the upper bound. This fact has also been verified by simulation. For $m>1$ group-paths, since there are no precedence relationships between the group-paths, the line of active tasks progresses at a different rate for each group-path, depending on the random service time requirement of each task.

If a process graph has more than one group-path, the method described above for obtaining bounds for $m=1$ must be improved to show the influence of the number of group-paths. To classify a general structured process graph, we require the maximum in-degrees for each level of all group-paths: $\{ \{d_{i1}|\forall i\}, \{d_{i2}|\forall i\}, \ldots , \{d_{im}|\forall i\} \}$. The order of maximum in-degrees per level in $\{d_{i\max}|\forall i\}$ for $m=1$ does not influence the bounds since any permutation will produce a process graph with a similar bound. By the same argument, the order of the sets of the maximum in-degrees for each group-path does not influence the bounds.

As has been discussed in the case of $m = 1$, the average system time is usually close to the upper bound; therefore, we will use the forced synchronization per level to *approximate* the average process time required for *each group-path*. For a group-path j and level i with maximum in-degree node having a value of d_{ij}, the probability distribution function for the service time of this level is $[F(t)]^{d_{ij}}$, and the probability density function is

$$f_{ij}(t) = d_{ij}\,[F(t)]^{d_{ij}-1}\,\frac{d\,F(t)}{d\,t} \tag{4.5}$$

To obtain the approximate average system time for the multi-path process graph, we can reduce each group-path into one 'super-node.' The probability density and probability distribution functions of the service time for each of the super-nodes j are

$$f_j(t) = f_{1j}(t) \otimes f_{2j}(t) \otimes \cdots \otimes f_{(r-1)j}(t) \tag{4.6}$$

where $\otimes$ represents the convolution operator, and

$$F_j(t) = \int_0^t f_j(\tau)\, d\tau \tag{4.7}$$

respectively, where each $f_{ij}(t)$ is a probability density function represented by Equation (4.5).

Looking at the terminating node, it has m 'super-nodes' "blocking" it. This is the exact analogy of Equation (4.3) with $F(t)$ replaced by the $F_j(t)$ of Equation (4.7):

$$E\begin{bmatrix}\textit{the average time required to process a}\\ \textit{node with in-degrees of m group-paths}\end{bmatrix}$$

$$= \int_0^\infty \left[1 - \prod_{j=1}^{m} F_j(t)\right] dt \tag{4.8}$$

The convolution in Equation (4.4) is a tedious task. However, according to the central limit theorem [PAPO65], as we add up a large number of independent random variables, the probability density function of the resulting sum is close to a normal density function, with the average of the sum equal to the sum of each level's mean process time and the variance of the sum equal to the sum of the variances of each level. We assume that the service time of the tasks between the levels are independent of each other. For each group path, i, we approximate the probability density function of its processing time by the normal density function with mean a_i and variance b_i, where a_i is the sum of the average times needed if the forced synchronization at each level in the group path is used and b_i is the sum of the corresponding variances of the average processing times at each level.

Using this approximation, Equations (4.6) and (4.7) become

$$f_i(t) \approx \frac{1}{\sqrt{2\pi b_i}} e^{-(t-a_i)^2/2b_i} \tag{4.9}$$

and

$$F_i(t) \approx \int_{-\infty}^{t} \frac{1}{\sqrt{2\pi b_i}} e^{-\frac{(x-a_i)^2}{b_i}/2} dx \tag{4.10}$$

respectively. Substituting $F_i(t)$ into Equation (4.8), we may then calculate the upper bound, $S_{UB}^{(m)}$, of the average system time (Equation (4.8)) required to complete a process graph with m group-paths as

$$S_{UB}^{(m)} = \int_0^{\infty} \left\{ 1 - \prod_{i=1}^{m} \left[\int_{-\infty}^{t} \frac{1}{\sqrt{2\,\pi\, b_i}} e^{-\frac{(x-a_i)^2}{b_i}/2} dx \right] \right\} dx$$

4.2.2.3 Bounds for Non-structured Process Graphs

For a non-structured process graph, it is harder to obtain an improved expression for the average system time. Although the lower bound expression on the average system time is still the same as that of the structured process graph, we have been unable to find an upper bound; this is due to the complicated coupling of the tasks, which makes it almost impossible to find group paths.

4.2.2.4 Tightness of the Bounds ($m = 1$)

The upper bound is obtained by summing the FST at each level in the process graph. For a given number of tasks, N, and number of levels, r, we know the lower bound to be $\frac{1}{\mu} r$. By obtaining the upper bound on the worst arrangement with N and r, we know approximately how tight the bounds are. Since we distribute one node each for the initial and terminating tasks and $N - 2$ nodes among the remaining $r - 2$ levels and assume that precedence relationships exist between all nodes of the adjacent levels (this being a process graph with any two adjacent levels forming a complete bipartite graph), we are looking for:

$$S_{MUB} \overset{\Delta}{=} \max S_{UB}$$

$$= \max \frac{1}{\mu} \left[2 + \sum_{j=1}^{r-2} \sum_{i=1}^{n_j} \frac{1}{i} \right]$$

subject to $\sum_{j=1}^{r-2} n_j = N-2$.

This maximum occurs when $n_j = \frac{N-2}{r-2}$ for all $j = 2, 3, 4, \cdots r-2$. If $\frac{N-2}{r-2}$ is not an integer, then $n_j = \left\lceil \frac{N-2}{r-2} \right\rceil$ for $j = 2, 3, \cdots, x$ where $x =$ remainder of $\frac{N-2}{r-2}$ and $n_j = \left\lfloor \frac{N-2}{r-2} \right\rfloor$ for $j = x+1, x+2, \cdots, r-2$. The ratio of the maximum upper bound S_{MUB} to the lower bound gives us a measure of the tightness of the bounds.

For example, for $N = 6$, $r = 4$,

$$S_{MUB} = \frac{1}{\mu} \left[2 + (r-2) \sum_{i=1}^{2} \frac{1}{i} \right]$$

$$= \frac{1}{\mu} 5$$

$$S_{LB} = \frac{1}{\mu} 4$$

Thus, $\frac{S_{MUB}}{S_{LB}} = 1.25$, which indicates that the upper and lower bounds are very close to each other, but for a larger process graph such as $N = 102$, $r = 22$,

$$\frac{S_{MUB}}{S_{LB}} = \frac{\left[2 + 20 \sum_{i=1}^{5} \frac{1}{i} \right] \frac{1}{\mu}}{22 \frac{1}{\mu}} = 2.17 \, ,$$

the bounds are further apart.

Of course, we are comparing the maximum upper bound possible, but as N becomes large, the ratio of the upper to lower bounds also gets larger. The exact upper bound depends on the number of levels and the number of precedence relationships in a given process graph.

Where, between the two bounds, does the exact average system time lie? This depends on how tightly the tasks are coupled to each other in the process graph. The more tightly they are coupled, the closer the average system time is to the upper bound, and vice versa. Figure 4.9 shows a process graph with average system time close to the upper bound.

4.3 Arrivals of Jobs (λ, G, x', $P = \infty$)

In the previous section (Sections 4.2), we obtained the average system time and bounds on the average system time of a job. Since we have assumed that there is an infinite number of processors, all jobs start execution at time zero. In this section, we assume that the jobs arrive from a Poisson source. As soon as a job arrives at the system, it starts execution (again this is due to $P = \infty$). Thus, the results obtained in Section 4.2 can also be applied to this case. In addition, from Little's result, we have, on the average, $\bar{k} = \lambda \; S(P)$ jobs in the system, where $S(P)$ is the average system time obtained in Section 4.2.1.2, and we have the bounds on the average system time as

$$\bar{k}_{UB} = \lambda \; S_{UB}$$

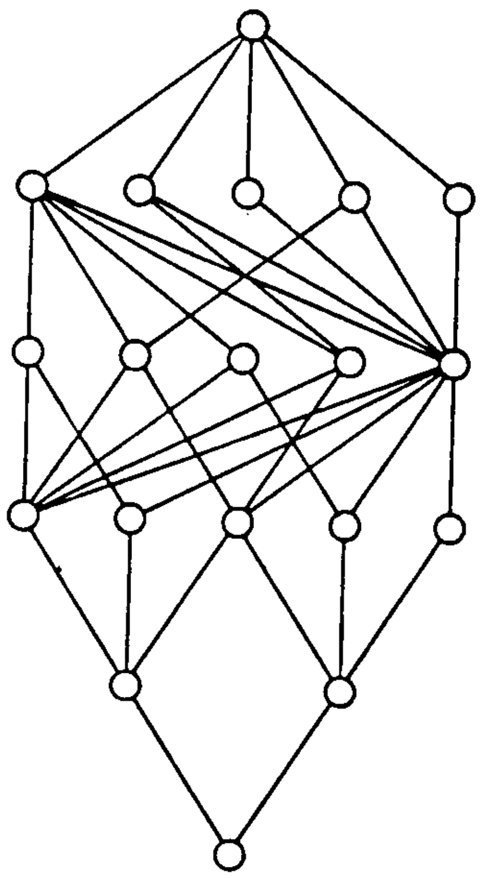
and

$$\bar{k}_{LB} = \lambda \; S_{LB}$$

where S_{UB} and S_{LB} are the bounds obtained in Section 4.2.2.

4.4 Stochastic Petri Nets (k, G, x', $P < \infty$)

In this section, we limit the number of processors to $P < \infty$, and the Stochastic Petri Net (SPN) [MOLL81] model is used to find the average utilization of these P processors given k jobs, a fixed process graph and task service times which are exponentially distributed. A process graph can easily be transformed into a Petri Net, as was shown in Section 2.3.1. To the resulting Petri Net we add a "place" called "Processor Available," with P tokens in it, and another "place" called "Unexecuted Jobs," with k tokens in it. Initially, all other places have no tokens in them. We add an edge from the "Processor Available" place to each transition requiring a processor and another edge from each transition finished using the processor to the 'Processor Available' place. Figure 4.10 gives an example of how we transform a process graph into such a Petri Net.



$$\frac{1}{\mu} = 50$$

$$S_{LB} = 300$$

$$S_{UB} = 494.95$$

$$S = 472.9$$

Figure 4.9

Process Graph with Average System Time Close to the Upper Bound

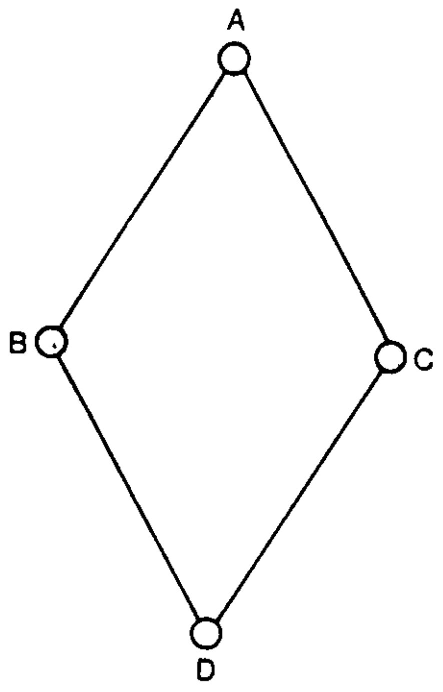


Figure 4.10a Process Graph

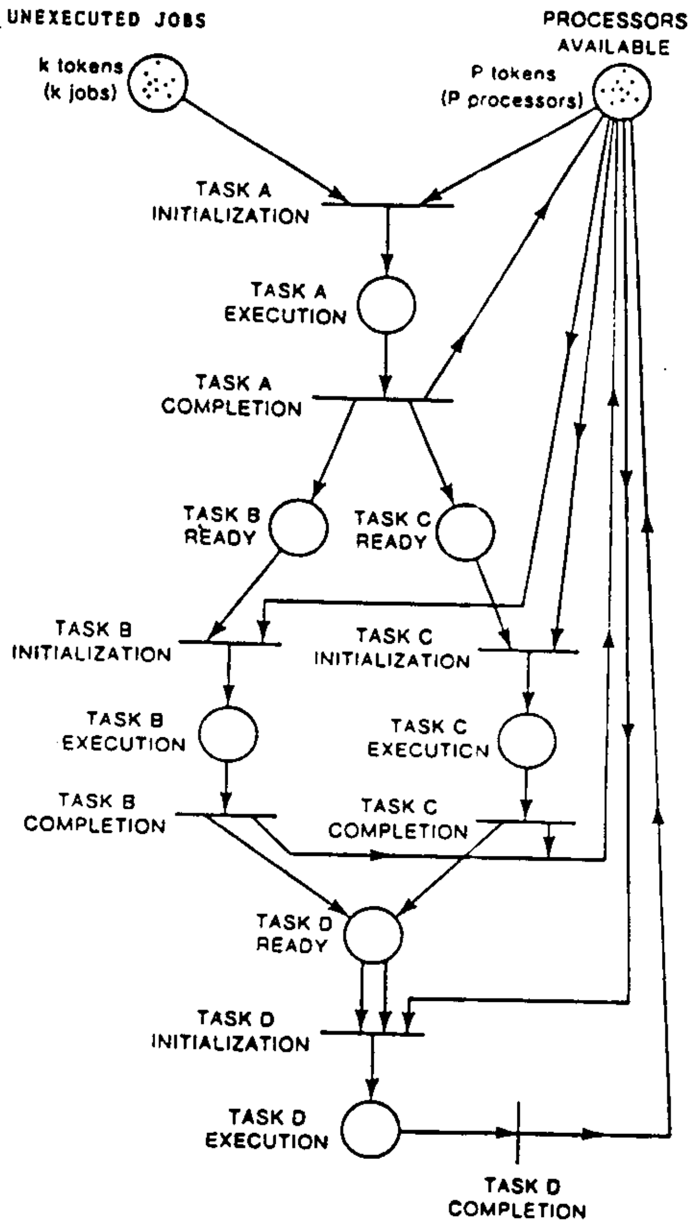


Figure 4.10b Petri Net

When all tokens in the "Unexecuted Jobs" place have been used up, and no other tokens remain in any place except the 'Processor Available' place, the Petri Net said to have reached the recurrent state. From the analysis provided by the Stochastic Petri Net, we can find the average number of tokens, I, in the "Processor Available" place, which also indicates the average number of idling processors. Hence, the average utilization of the P processors is

$$\rho = 1 - \frac{I}{P}$$

Since P is limited, we do have a scheduling problem; however, a SPN does not allow the assignment of a specific task to a processor. The assignment depends on which transition requiring a processor fires next. Thus, the performance obtained with a SPN analysis lies between the best and worst assignment results.

If, instead the value of P is the design parameter, we may use the definition of power [KLEI79] to find the optimal number of processors for a specific process graph and number of jobs. Power is defined to be the utilization of the processors divided by the normalized average system time. A SPN provides the values of both of these variables for a specific value of P. We can therefore plot power versus the number of processors to find that number P at which the power will be maximized.

4.5 Task Assignment (k, G, x, $P<\infty$)

In this section, we find bounds on the average system time by developing algorithms that will give the best and worst scheduling in terms of the average system time.

If the ratio of these two bounds is not large, perhaps random scheduling of the tasks to the processors could then be allowed. Random assignment has the advantage of no overhead being needed to schedule tasks. Whenever a processor is available, it will just grab any task that is ready to be executed. We know the performance of the system must fall between the two bounds.

First, we assume the shape of the process graph to be bounded by a diamond as in Figure 4.11.

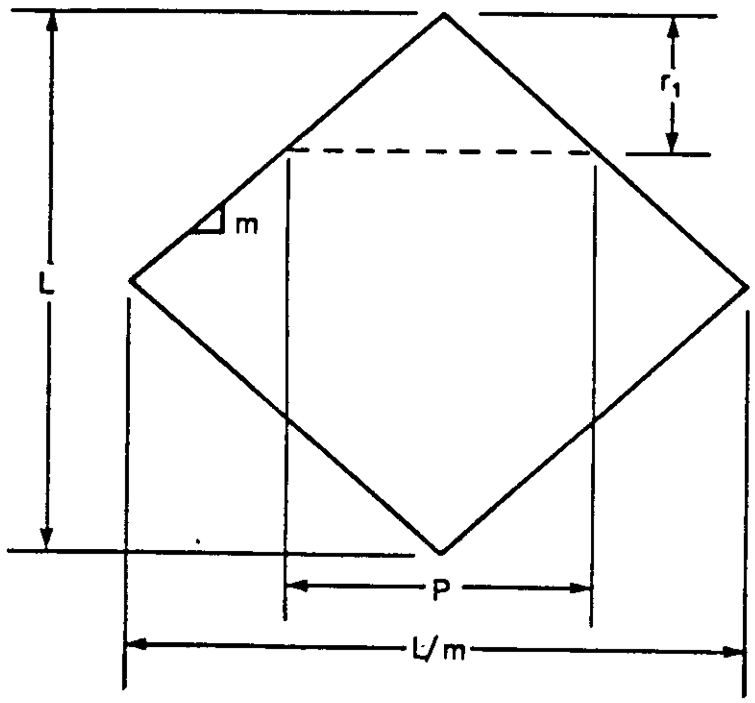


Figure 4.11 Diamond-shaped Process Graph

This type of process graph can be characterized by two parameters: L and m, where L is the number of levels in the process graph and m is the slope of the diamond enveloping the boundary tasks. We assume a continuum of tasks within the diamond.

Since the service time of tasks are constant, we normalize the service time of each task to one unit of time.

From [COFF76], we know that the assignment which minimizes the average system time is the shortest expected remaining processing time first assignment. This is the Depth-first Assignment Algorithm, where all available processors will be assigned to the tasks in a job that is closest to being completed. In other words, we are trying to complete jobs as fast as possible.

On the other hand, if we want to maximize the utilization of the processors, then the longest expected remaining time first assignment is used. This is the Breadth-first Assignment Algorithm, where all available processors are assigned to the jobs that have the least amount of processing to be done. In this assignment, we are trying to process all jobs at the same time, so that all the jobs complete at times very close to each other. We are interested in finding the ratio of the average system time obtained from these two assignments,

$$\psi = \frac{S_b}{S_d}$$

where S_b is the average system time using Breadth-first Assignment and S_d is the average system time using Depth-first Assignment.

If we assume all jobs depart at the same instant as the last job when calculating S_b, then

$$S_b = 2r_1 + \frac{2\left(\frac{\frac{L}{m}\frac{L}{2}}{2} - \frac{\frac{r_1 2}{m} r_1}{2} \right) k}{P}$$

where $r_1 = \frac{Pm}{2k}$ and $P \leq \frac{L}{m}k$. Simplifying the above expression, we get

$$S_b = \frac{Pm}{2k} + \frac{kL^2}{2Pm}$$

If we provide a maximum number of processors, $P = \frac{L}{m}k$, then

$$S_b = \frac{\left(\frac{L}{m}k\right) m}{2k} + \frac{kL^2}{2\left(\frac{L}{m}k\right) m}$$

$$= \frac{L}{2} + \frac{L}{2} = L$$

that is, it takes L units of service time to complete all k jobs. This is what we expect, since each job has $\frac{L}{m}$ processors, which is equivalent to $P = \infty$. Thus, each job takes L units of time to complete and all k jobs run in parallel.

If we let $P = 1$, then

$$S_b = \frac{m}{2\,k} + k\,\frac{L^2}{2\,m}$$

Since there are $\frac{L^2}{2\,m}$ tasks in each job, it takes $k\left(\frac{L^2}{2\,m} - 1\right) + i$ units of time to complete the i^{th} job. Thus, the average system time for these k jobs is

$$S = \frac{\sum_{i=1}^{k} k\left(\frac{L^2}{2\,m} - 1\right) + i}{k}$$

$$= \frac{k^2\left(\frac{L^2}{2\,m} - 1\right) + \frac{k\,(\,k+1\,)}{2}}{k}$$

$$= k\,\frac{L^2}{2\,m} - \frac{k}{2} + 1$$

The difference between S and S_b is due to the assumption in calculating S_b that all jobs depart at the same instant as the last job; this assumption is pessimistic, as we see, and so it may be made in obtaining our bound. Thus, for $k \geq 2$, $S_b > S$.

For example, if we let $P = 1$, $m = 1$, $k = 10$, and $L = 5$, then

$$S_b = \frac{m}{2\,k} + \frac{k\,L^2}{2\,m} = 125 + \frac{1}{20}$$

and

$$S = k\,\frac{L^2}{2\,m} - \frac{k}{2} + 1 = 121$$

As predicted, $S < S_b$. By changing the value of P to 50 while keeping all other parameters in the above example the same, we get

$$S_b = \frac{50}{(\,2\,)(\,10\,)} + \frac{(\,10\,)(\,25\,)}{(\,2\,)(\,50\,)} = 5$$

which is exactly the average system time, S, using the Breadth-first Assignment.

As for S_d, the least average system time can be obtained by considering the process graph as a rectangular shape since, for this shape of process graph, the utilization of the processors is at the maximum. The total number of tasks in a process graph is $\frac{L^2}{2m}$; so, the width (average number of tasks per level) of the rectangular process graph is $\frac{L}{2m}$. Thus, the largest number of jobs that can be processed at the same time by P processors is

$$k' = \begin{cases} \dfrac{P}{\dfrac{L}{2m}} & \dfrac{P}{\dfrac{L}{2m}} < k \\ k & \dfrac{P}{\dfrac{L}{2m}} \geq k \end{cases}$$

Thus,

$$S_d = \frac{\sum_{i=1}^{\lfloor j \rfloor} ik'L + (j - \lfloor j \rfloor)k'(\lfloor j \rfloor + 1)L}{jk'}$$

where $j = \dfrac{k}{k'} = \dfrac{k}{Min\left[k, \dfrac{P}{L/2m}\right]}$. In the numerator, the summation term represents that every L units of time, k' jobs are completed; the second term in the numerator represents the time required, $(\lfloor j \rfloor + 1)\, L$, by the last $(j - \lfloor j \rfloor)\, k'$ jobs. Hence,

$$S_d = \frac{\dfrac{\lfloor j \rfloor(\lfloor j \rfloor + 1)}{2} + (j - \lfloor j \rfloor)(\lfloor j \rfloor + 1)L}{j}$$

If we assume $\dfrac{P}{L/2m} < k$, then $j = \dfrac{k}{\dfrac{P}{L/2m}} = \dfrac{kL}{2Pm}$. If j is an integer, then $S_d = \dfrac{j+1}{2}\, L$, and

$$\psi = \frac{S_b}{S_d} = \frac{\dfrac{kL^2}{2Pm} + \dfrac{Pm}{2k}}{\dfrac{j+1}{2}\, L}$$

$$= \frac{\dfrac{kL}{2Pm} + \dfrac{Pm}{2kL}}{\dfrac{kL}{4Pm} + \dfrac{1}{2}} .$$

Otherwise,

$$\psi = \frac{\dfrac{kL^2}{2Pm} + \dfrac{Pm}{2k}}{\dfrac{\dfrac{\lfloor j \rfloor(\lfloor j \rfloor+1)}{2} + (j-\lfloor j \rfloor)(\lfloor j \rfloor+1)L}{j}}$$

If we assume $\dfrac{P}{L/2m} \geq k$, then $j = 1$, and $S_d = L$, or

$$\psi = \frac{\dfrac{kL^2}{2Pm} + \dfrac{Pm}{2k}}{L}$$

$$= \frac{kL}{2Pm} + \frac{Pm}{2kL}$$

For example, Figure 4.12 shows ψ versus P for $L = 10$, $k = 5$ and $m = 1$, and Figure 4.13 shows ψ versus P for $L = 10$, $k = 5$ and $m = 2$. We observed, in both Figures 4.12 and 4.13, that the value of ψ, after falling initially, will rise slightly before monotonically decreasing again. The cause of this rise is from the assumption of a rectangular process graph in calculating S_d. Since the rectangular process graphs have a constant width of $L/2m$, when the value of P reaches a multiple of $L/2m$, an additional job can depart at every L time step. This fact decreases the value of S_d faster than the value of S_b is decreasing whenever the value of P is close to a multiple of $L/2m$. After $P > \dfrac{k}{L/2m}$, $S_d = L$, then this effect disappears.

The next theorem gives the asymptotic behavior of ψ as P and k become large.

Theorem 4.8
As k and P become large, $\psi < 2$.

Proof:

Case I $\quad \dfrac{P}{L/2m} < k$

$$\psi = \frac{S_b}{S_d}$$

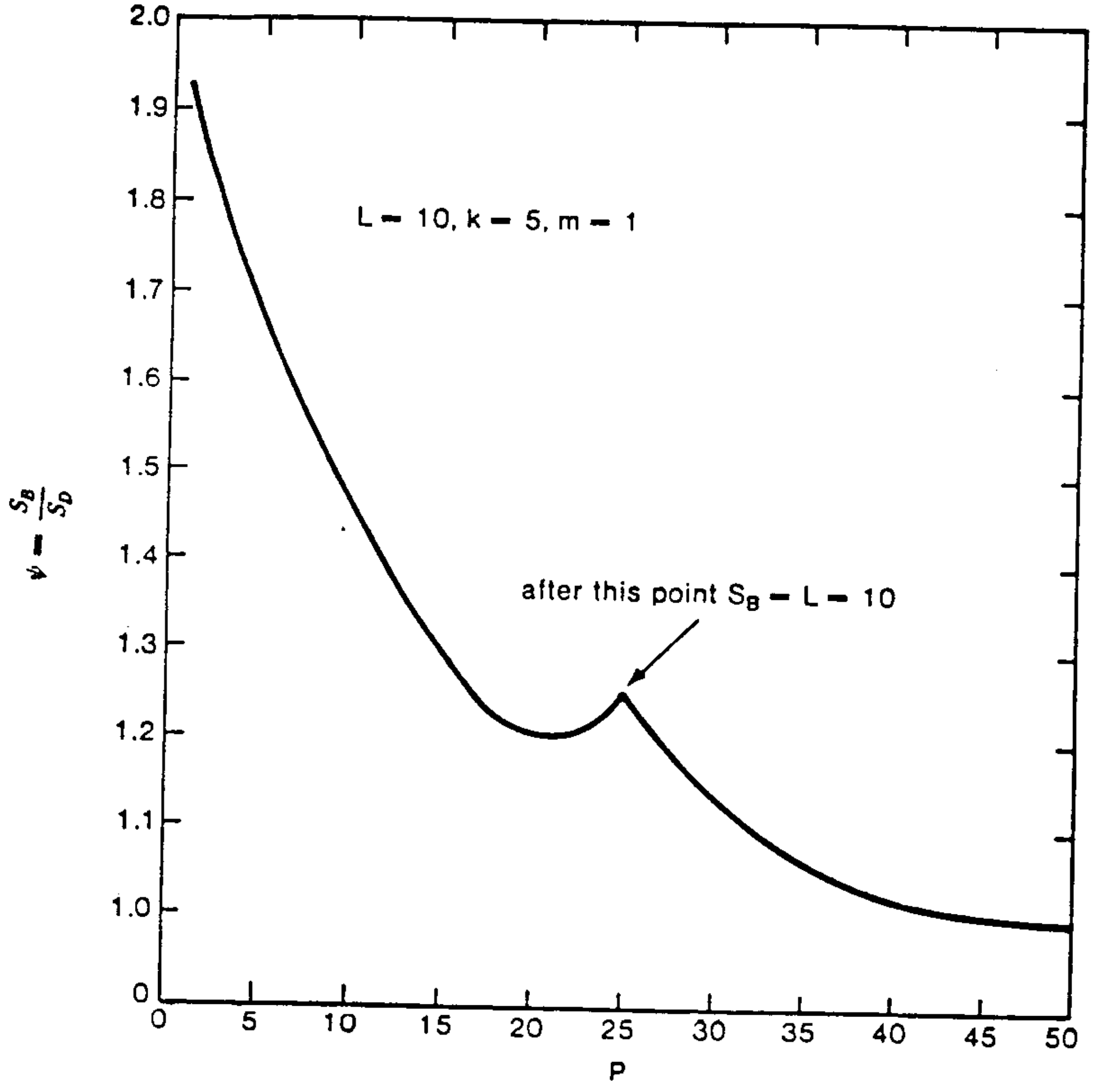


Figure 4.12 ψ versus P for $L = 10$, $k = 5$ and $m = 1$

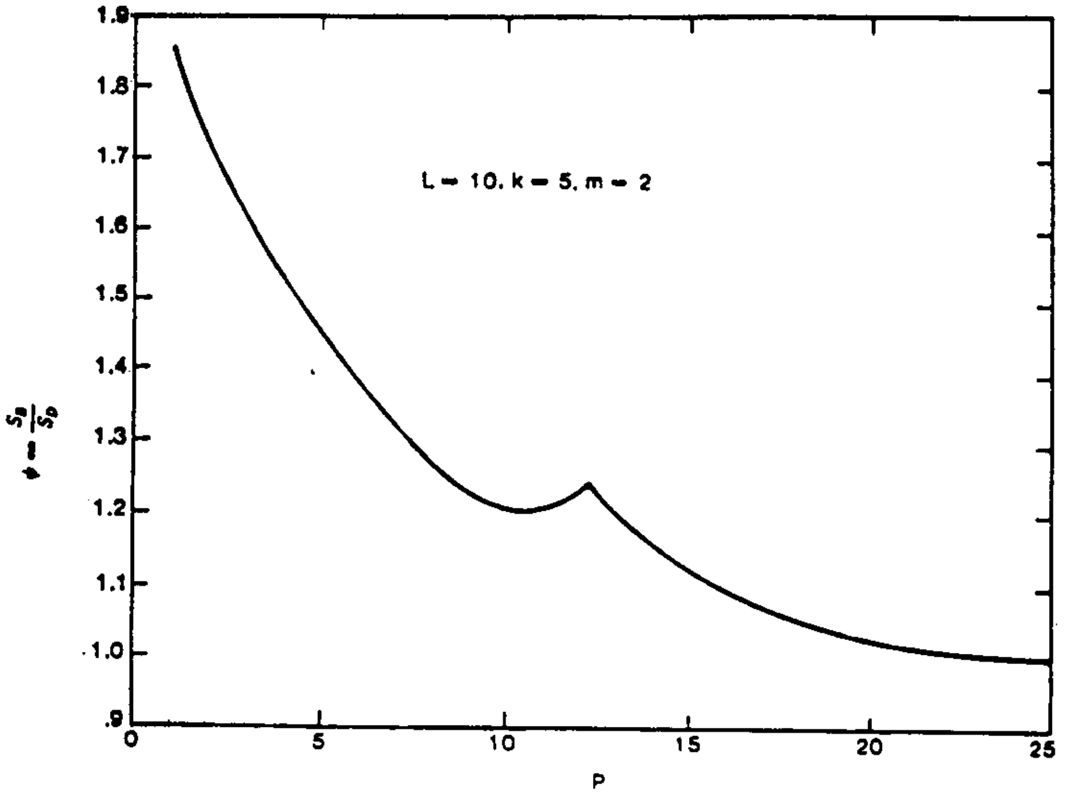


Figure 4.13 ψ versus P for $L = 10$, $k = 5$ and $m = 2$

$$\ldots = \frac{\frac{kL^2}{2Pm}+\frac{Pm}{2k}}{\frac{j+1}{2}L}$$

where $j = \dfrac{k}{\frac{P}{L/2\ m}} = \dfrac{kL}{2Pm}$ or

$$\psi = \frac{\frac{kL^2}{2Pm}+\frac{Pm}{2k}}{\frac{kL^2}{4Pm}+\frac{L}{2}}$$

$$= 2 + \frac{\frac{Pm}{2k} - L}{\frac{kL^2}{4Pm} + \frac{L}{2}}$$

Since we have the assumption of $\dfrac{P}{L/2m} < k$ or $\dfrac{Pm}{2k} < \dfrac{L}{4} < L$; therefore, $\dfrac{Pm}{2k} - L < 0$. Thus, $\psi < 2$.

Case II $\quad \dfrac{P}{L/2m} \geq k$

For this case, we know $S_d = L$. Therefore, $\psi = \dfrac{Pm}{2kL} + \dfrac{kL}{2Pm}$. If we let $x = \dfrac{2Pm}{Lk}$, then

$$\psi = \frac{1}{4}x + \frac{1}{x} \tag{4.11}$$

Plotting ψ versus x, we obtain Figure 4.14.

If we take the derivative of ψ with respect to x (approximating x by a continuous variable) and set the result to zero, we can find the x at which ψ is at the minimum:

$$\frac{d\,\psi}{d\,x} = \frac{1}{4} - \frac{1}{x^2} = 0$$

The minimum occurs at $x = 2$. Thus, since $x = 2$ is an integer, we have also found the minimum of ψ as the integer. As long as $x = \dfrac{2pm}{Lk} < 8$, then $\psi < 2$, or $P < \dfrac{4Lk}{m} = 4k\dfrac{L}{m}$. In other words, as long as the number of processors is fewer than four times the number of jobs multiplied by the widest part of the process graph $(\dfrac{L}{m})$, then $\psi < 2$. Usually, we use only, at most, $k\left\lceil \dfrac{L}{m} \right\rceil$ processors. So, if we let $P \leq k\dfrac{L}{m}$, then $x = \dfrac{2Pm}{Lk} \leq \dfrac{k\frac{L}{m}2m}{Lk} = 2$. Also, since

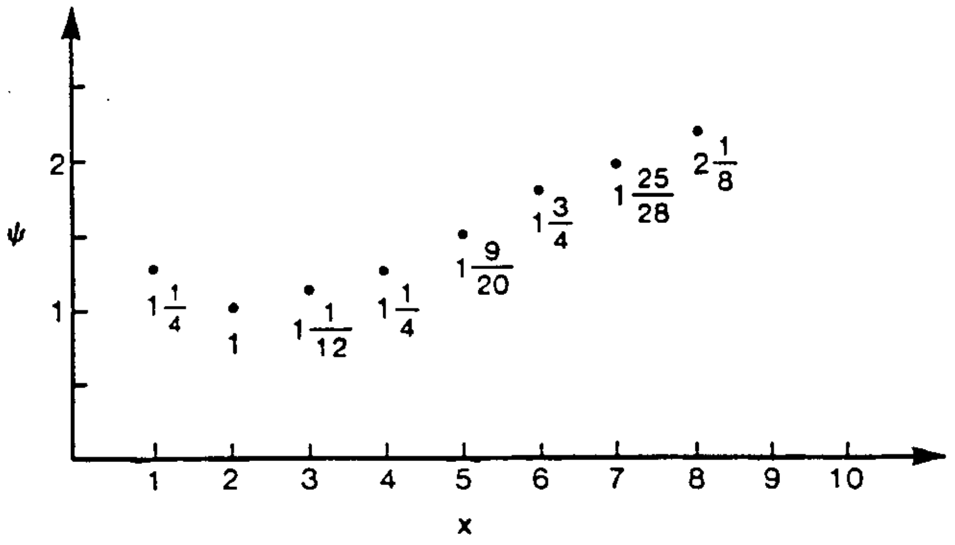


Figure 4.14 ψ versus x

we have assumed $P \geq \frac{kL}{2m}$, $x = \frac{2Pm}{Lk} > \frac{\frac{kL}{2m}Lm}{Lm} > 1$. Again, we see from Figure 4.14 that between $x = 1$ and $x = 2$, $\psi \leq 1\frac{1}{4}$.

|||

From Case II of the proof in Theorem 4.3, we know that, for $\frac{P}{L/2m} \geq k$, $\psi = \frac{Pm}{2kL} + \frac{kL}{2Pm}$. Therefore, in Figures 4.12 and 4.13, the values of ψ for $P \geq \frac{kL}{2m}$ are defined by the same expression as Equation 4.11. If we extend this curve backward for smaller values of P until $\psi = 2$, we obtain an easier bound on ψ. This is shown in Figure 4.15. The values of P' at which $\psi = 2$ intercept this curve can be calculated from the following expression:

$$\psi = 2 = \frac{P'\,m}{2kL} + \frac{kL}{2P'\,m}$$

or

$$\frac{P'^{\,2}m}{2kL} - 2P' + \frac{kL}{2m} = 0$$

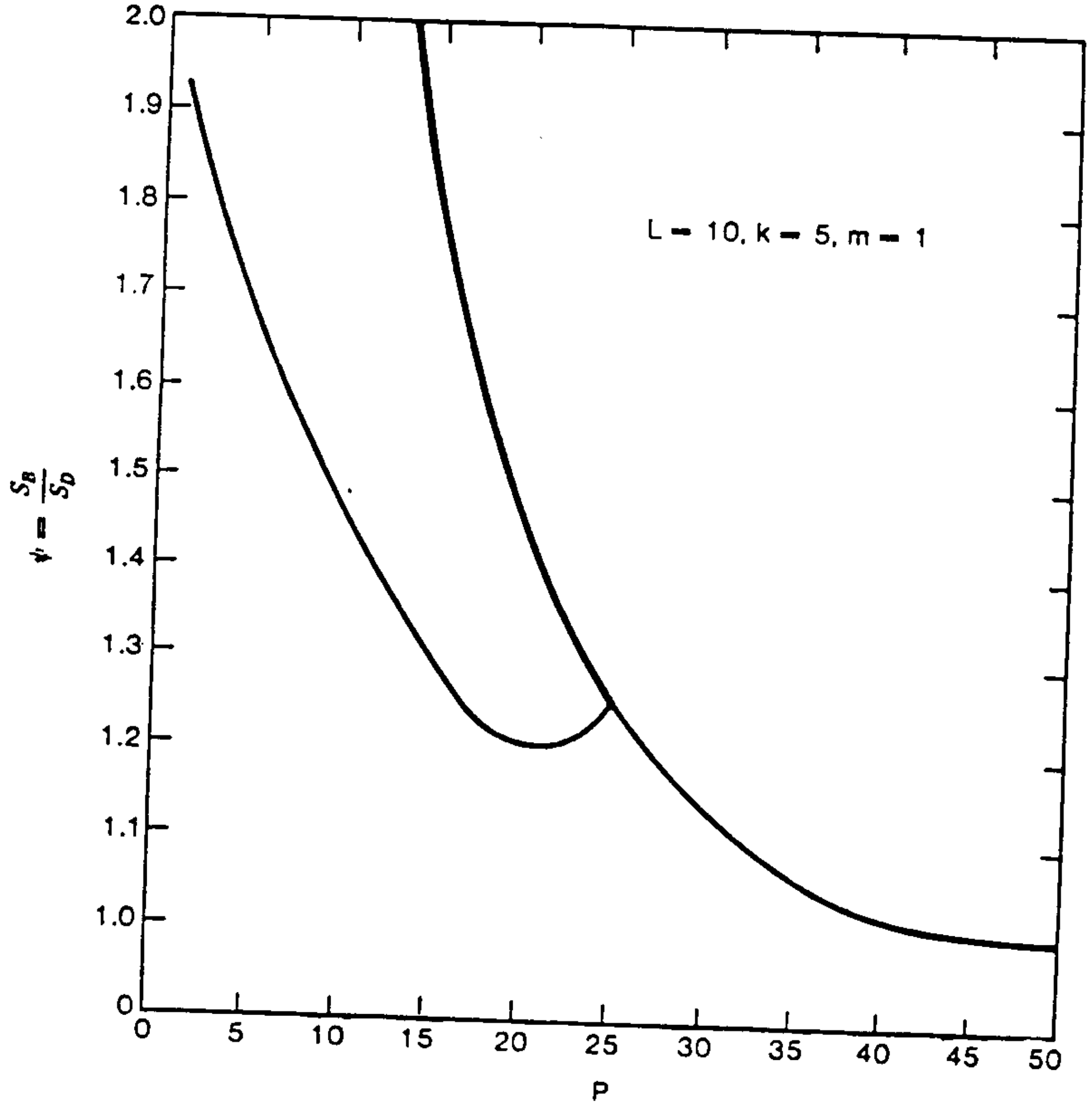


Figure 4.15 ψ versus P

Solving for P' we get

$$P' = 0.268\frac{kL}{m}$$

Therefore, for $1 \le P \le 0.268\frac{kL}{m}$, the upper bound on ψ is 2, and for $0.268\frac{kL}{m} \le P \le \frac{kL}{m}$, the upper bound on ψ is the polynomial $\frac{Pm}{2kL} + \frac{kL}{2Pm}$. This fact is given in the next Corollary.

Corollary 4.4 The upper bound of the ratio ψ is

$$\psi \le \begin{cases} 2 & 1 \le P \le 0.268\frac{kL}{m} \\ \frac{Pm}{2kL} + \frac{kL}{2Pm} & 0.268\frac{kL}{m} \le P \le \frac{kL}{m} \end{cases}$$

Figure 4.16 shows a plot of ψ versus P.

Thus, we see that the ratio of the average system time for the worst assignment to the best assignment is given by Corollary 4.4. This ratio is quite small. Hence, if we do allow random scheduling of the tasks to the processors, the resulting average system time will be bounded relatively tightly by S_d and S_b.

4.6 Discussion

In this chapter, we discussed two methods for obtaining the average system time and the concurrency measure of a fixed process graph with randomly distributed service time of tasks. These results apply to both a fixed number of jobs at time zero and an arrival of jobs from a random source because the number of processors is assumed to be infinite. In the process of finding the average system time, however, either an enumeration algorithm must be used or a system of a large number of equations must be solved. Both methods are time consuming when the number of tasks in the process graph becomes large.

We can, however, use the upper and lower bounds on the average system time as a rule of thumb in approximating the concurrency measure. A relatively easy way to calculate both bounds has been presented.

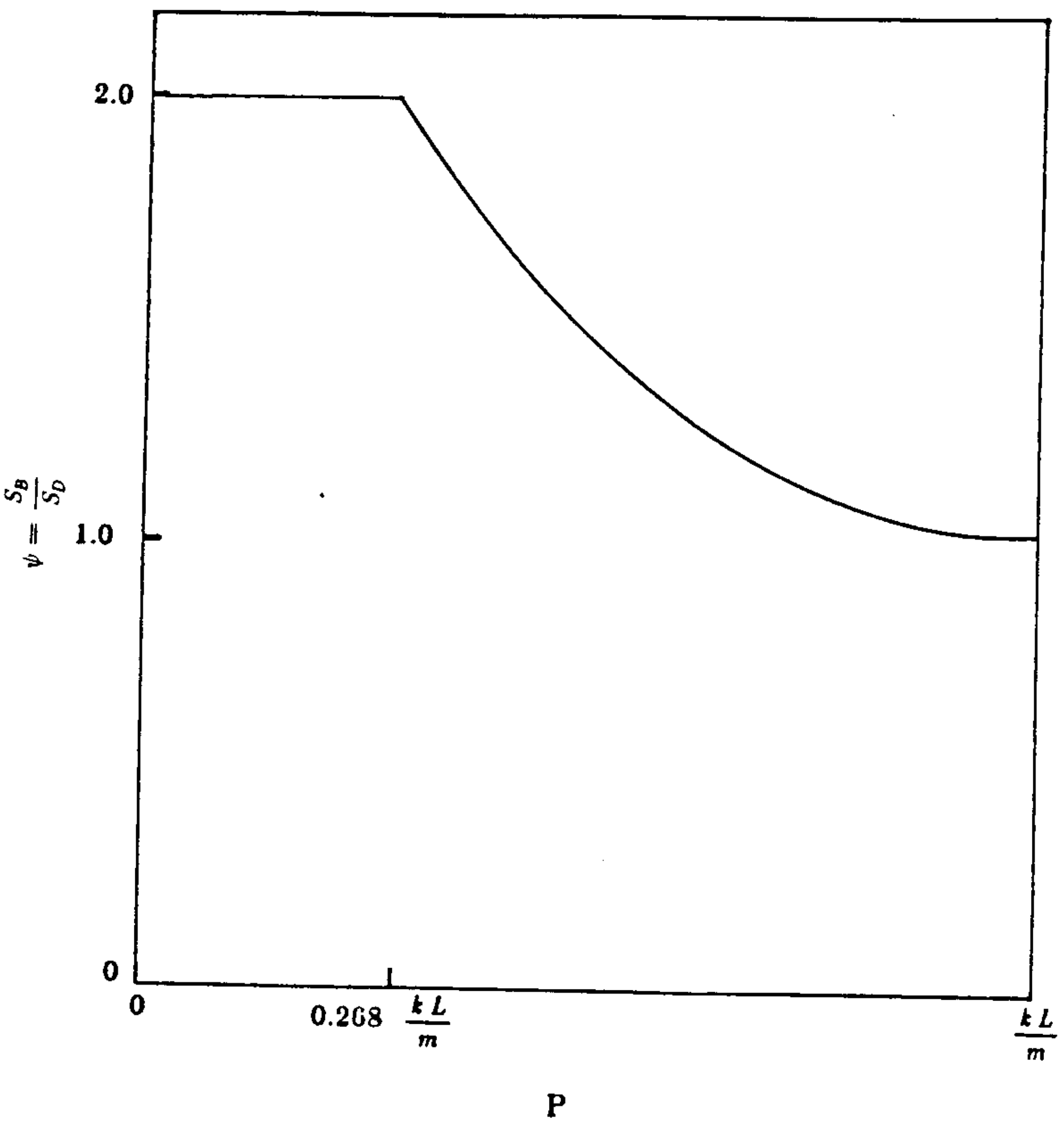


Figure 4.16 The Upper Bound for the Ratio ψ

A Stochastic Petri Net model was used to find the average utilization of processors for the case of a fixed number of jobs, fixed process graph, random task service time and limited number of processors.

Two scheduling algorithms were analyzed to find the ratio of the worst algorithm to the best algorithm in terms of the average system time. We found this ratio to be less than two for diamond-shaped process graphs.

CHAPTER 5
Random Process Graphs

5.1 Introduction

In Chapter 4 we studied cases where the process graph was considered to be fixed. Therefore, the analysis and the system parameters obtained are good for only one particular process graph. When we change to a different process graph or try to predict the general system behavior of the other process graphs, the results from a single process graph are often not very helpful.

In Sections 5.3 and 5.4, we obtain bounds on the average system time of a random process graph, with N tasks and enough processors so that a processor will always be available any time a task demands it. With these bounds on the average system time, we can find the bounds on the speedup achievable when we use multiprocessors to process a fixed number of jobs. The speedup is defined as the inverse of the concurrency measure and the concurrency measure is still defined as the average system time using P processors divided by the average system time using only one processor. In Sections 5.5 and 5.6 we will assume that the number of processors is limited.

In our first model in Section 5.4, we will assume that the number of precedences is arbitrary; in our second model, this parameter is fixed to a constant. In the former case, only the *arrangement* of the tasks in the process graph is studied.

In this chapter (except in Section 5.4.2), we assume that the N tasks do not include the initial and the terminating tasks, and the number of precedence relationships, M, (if given) does not include the precedence relationships between the initial task to the next level tasks and from any task into the terminating task. The resulting upper and lower bounds are known to be two average task service times smaller than the actual bounds. This change, while not affecting any of the results, does allow a clearer explanation without worrying about the two additional tasks at the boundary.

In Section 5.2.1 we show that the number of arrangements of the tasks for a process graph with a fixed number of tasks, N, is $2^{(N-1)}$ and that the number of arrangements for a particular level, r, is $\binom{N-1}{r-1}$. From the construction algorithm described in Section 5.2.2, we show that the number of arrangements for an N-task, r-level process graph forms a Pascal tree. Since the number of arrangements is Gaussian distributed with respect to the number of levels in a process graph as N becomes large (this is proved in Section 5.2.3), most of the arrangements will have a number of levels which (percentage-wise) is close to the average level, namely $\left\lfloor \frac{N+1}{2} \right\rfloor$. The Chernoff bound, introduced in Section 5.2.4, will be used for the probabilistic argument in Sections 5.3 and 5.4.1, in which we obtain upper and lower bounds on the average system time for a randomly selected process graph.

With the number of precedence relationships (edges) and the number of levels added as additional parameters, we find tighter bounds in Section 5.4.2. The two upper bounds obtained are compared in the Section 5.4.3.

In Section 5.5, the issue of trading off between the utilization of processors and the average system time is discussed. Finally, in Section 5.6, we briefly look at the bounds when the number of the processors is limited to a finite number.

5.2 Some Properties of Random Process Graphs

5.2.1 Total Number of Arrangements with N Tasks

Process graphs with N tasks can have $r=1,2,3,\cdots,N$ levels. The only constraint on the arrangement of the tasks is that each of the r levels must contain at least one task. Therefore, we can replace the question, 'How many ways can we distribute N tasks in r levels with each of the level containing at least one task' by the following simpler question, 'How many ways can we distribute $(N-r)$ tasks in r levels.'

This is a combinatorics problem. We know that the number of combinations of x distinct objects taken y at a time with repetition allowed is

$$\binom{x+y-1}{x-1} = \binom{x+y-1}{y}$$

In terms of the number of tasks and levels, we wish to find the number of combinations of r levels taken $(N-r)$ at a time. An intuitive way of looking at this is to observe that we are selecting a particular level for each of the $(N-r)$ tasks. Therefore, letting $x = r$ and $y = N-r$, we have

$$\binom{r+(N-r)-1}{N-r} = \binom{r+(N-r)-1}{r-1} = \binom{N-1}{r-1}$$

as the number of arrangements.

When summing the number of arrangements over all levels, we get the total number of arrangements for the N-task process graphs:

$$\sum_{r=1}^{N}\binom{N-1}{r-1} = \sum_{i=0}^{N-1}\binom{N-1}{i} = 2^{(N-1)}$$

5.2.2 A Method of Constructing All Arrangements of Process Graphs with N Tasks

After we present a method that constructs the arrangements of process graphs by using a recursive algorithm, we then prove that this algorithm generates all arrangements for the process graphs of N tasks. From this construction method, we will see that the number of arrangements for an N-task and r-level process graph forms a Pascal tree (from which we can also obtain the number of arrangements for a process graph with N tasks).

The construction method is shown in Algorithm C:

Algorithm C

1. For $N=1$, there is just one arrangement

2. For $N \geq 2$ tasks and r levels, we add to all the arrangements (possibly none) with $(N-1)$ tasks and $(r-1)$ levels, one task at a new level, the r^{th} level; we also add one task to the r^{th} level of all the arrangements (possibly none) with $(N-1)$ tasks and r levels.

3. Repeat Step 2 for each level $r = 2, 3, \cdots, N$ to obtain all the arrangements for the N-task process graph.

Note that, in order to construct the arrangements for the N-task process graphs, we must also construct all the i-task process graphs where $i < N$.

Figure 5.1 shows examples of constructing

a. all 2 arrangements of the 2-task process graphs from the one 1-task process graph, and

b. all 10 arrangements of the 6-task, 3-level process graphs from the 4 arrangements of 5-task, 2-level process graphs and 6 arrangements of the 5-task, 3-level process graphs.

From Figure 5.1, we observe an interesting property of the arrangements for process graphs. That it, for any arrangement R, there exists another arrangement R' which is symmetric to R such that if in arrangement R, we let n_i representing the number of tasks in level i, then in arrangement R' the number of tasks in i^{th} level, n'_i, is

$$n'_i = n_{L-i+1} \qquad \text{for } i = 1, 2, \cdots, L$$

where L is the total number of levels in R and R'.

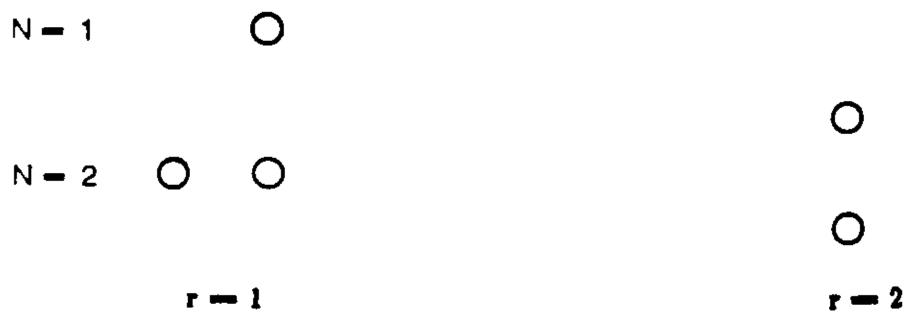


Figure 5.1a Arrangements of Process Graphs with 2 Tasks

From Section 5.2.1 above, we know that, with N tasks there are $\binom{N-1}{r-1}$ arrangements with r levels. The above algorithm constructs

$$\binom{(N-1)-1}{(r-1)-1} + \binom{(N-1)-1}{r-1} = \binom{N-1}{r-1}$$

arrangements. Hence, if we can show that all of the $\binom{N-1}{r-1}$ arrangements are unique, then we shall have obtained all the arrangements with N tasks and r levels.

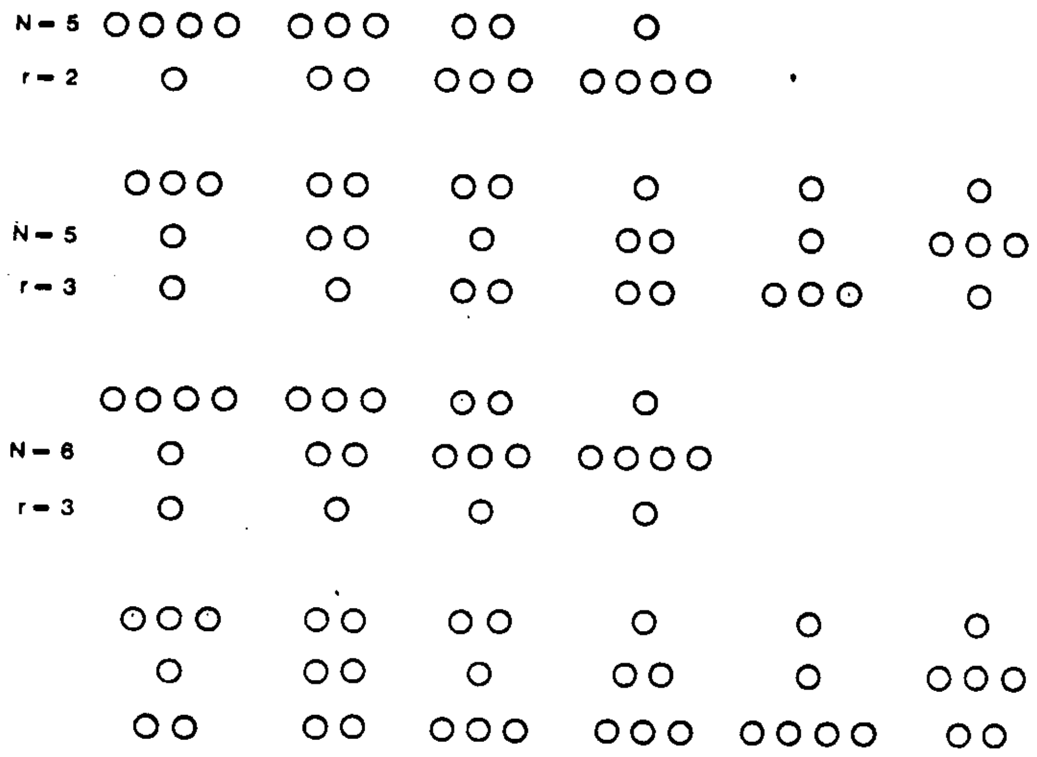


Figure 5.1b Arrangements of Process Graphs with 6 Tasks and 3 Levels

Lemma 5.1
All arrangements created for process graphs with N tasks and L levels using the Algorithm C are unique.

Proof
Assume all unique arrangements with $(N-1)$ tasks and $r=1,2, \cdots ,(N-1)$ levels. To construct the arrangements with N tasks and L levels, where $1 \leq L \leq N$, we add a task to the new L^{th} level for all arrangements with $(N-1)$ tasks and $(L-1)$ levels, and we add a task to the bottom level of all arrangements with $(N-1)$ tasks and L levels. The arrangements for the former case will have only one task at the L^{th} level, while the arrangements from the latter case will have more than one task in the L^{th} level. Hence, between these two cases, no two arrangements can be identical.

We know from the assumption that all the arrangements with $(N-1)$ tasks are unique; therefore, the resulting arrangements after adding a new level, the L^{th} level, with one task in it, should still be unique in the former case; the resulting arrangements after adding a new task to the L^{th} level should also be unique within the latter case. Thus, all $\binom{N-1}{r-1}$ arrangements obtained by our algorithm are unique.

|||

Theorem 5.2

All arrangements created by Algorithm C for N-task process graphs are unique.

Proof.

From Lemma 5.1 we know that the arrangements are unique within each level r. Since arrangements in different levels cannot be similar to each other (this is due to the constraint that each level must have at least one task), no two arrangements in the $2^{(N-1)}$ arrangements created by Algorithm C are similar to each other.

|||

From this construction method, we see that the number of arrangements for N tasks and r levels actually forms a Pascal tree. In the next section, we show that, as N becomes large, the distribution of the number of arrangements with respect to the number of levels is Gaussian.

5.2.3 Distribution of the Number of Arrangements

The number of arrangements for an N-task process graph with respect to each level is a binomial number. If we analyze the distribution of the tasks as a random variable, Y, such that if $Y_i = 1$, a new task is added to a new level; if $Y_i = 0$, it is added to an existing level. In a Pascal tree, this is equivalent to going either to the left (i.e. the number of levels remains the same) or to the right (i.e. the number of levels increases by one) of the current location in the next level of the Pascal tree.

We define a random variable, $Y = \sum_{i=1}^{N} Y_i$, where

$$Y_i = \begin{cases} 1 & \textit{with probability } q \\ 0 & \textit{with probability } p = 1-q \end{cases}$$

We let $p=q=\frac{1}{2}$. When we sum N such random variables, we obtain the Bernoulli distribution

$P_N(k) = \binom{N}{k} p^{N-k} q^k$, which has the mean of $N\,q = N/2$ and a variance of $N\,p\,q = N/4$ when $p = \frac{1}{2}$. In terms of our parameters, when $Y_i = 1$, we add the new node i to a new level of the arrangement; when $Y_i = 0$, we add the new node i to one of the existing levels. Summing N of the random variables Y_i, we have an arrangement with Y levels. This distribution is the same as the distribution for the number of arrangements with respect to the number of levels in a process graph with N tasks. If we normalize this distribution so that the mean is zero and the variance is one, then the characteristic function[†] is given by [MISE64]

$$\phi(\omega) = \left(pe^{-\frac{i\omega}{\sqrt{N}}} + qe^{\frac{i\omega}{\sqrt{N}}} \right)^N$$

$$= \left(\frac{1}{2}e^{-\frac{i\omega}{\sqrt{N}}} + \frac{1}{2}e^{\frac{i\omega}{\sqrt{N}}} \right)^N$$

where $i = \sqrt{-1}$.

Since $e^{ia} = 1+ia-\frac{a^2}{2}+\cdots,$

$$\phi(\omega) = \left\{ \frac{1}{2}\left[1+i\frac{\omega}{\sqrt{N}} - \frac{\left(\frac{\omega}{\sqrt{N}}\right)^2}{2} + \cdots \right] + \frac{1}{2}\left[1+\frac{-i\omega}{\sqrt{N}} - \frac{\left(-\frac{\omega}{\sqrt{N}}\right)^2}{2} + \cdots \right] \right\}^N$$

[†] Let X be a random variable with probability distribution function $F(x)$. The characteristic function of $F(x)$ is the function ϕ defined for real ω by

$$\phi(\omega) = \int_{-\infty}^{\infty} e^{i\omega x}\, dF(x) = u(\omega)+iv(\omega)$$

where $i = \sqrt{-1}$,

$$u(\omega) = \int_{-\infty}^{\infty} \cos\omega x\, dF(x)$$

and

$$v(\omega) = \int_{-\infty}^{\infty} \sin\omega x\, dF(x)$$

$$= \left[1 - \frac{\omega^2}{2N} + O\left(\frac{1}{N}\right)\right]^N$$

where $O(x)$ denotes any function which goes to zero faster than x, that is, $\lim_{x \to 0} \left[\frac{O(x)}{x}\right] = 0$. Take the limit as N approaches infinity to obtain

$$\lim_{N \to \infty} \phi(\omega) = e^{-\frac{\omega^2}{2}}$$

But $e^{-\frac{\omega^2}{2}}$ is the characteristic function of a normalized (i.e., mean = 0 and variance = 1) Gaussian distribution. Thus, we have shown that the number of arrangements with respect to the number of levels is Gaussian distributed.

5.2.4 Chernoff Bound on the Tail Probability

Suppose we want to know the probability that a randomly selected arrangement has more than y levels. The Chernoff bound [KLEI75] gives us a very good bound on this tail probability.

First, from [KLEI75] we find the moment generating function for the sum of N Bernoulli trials. For $N = 1$,

$$M(v) = \frac{1}{2} + \frac{1}{2}e^v$$

which indicates that, with probability $\frac{1}{2}$, we add a new level (e^{1v}), and with probability $\frac{1}{2}$, we don't add a new level (e^{0v}). We now define the semi-invariant generating function

$$\gamma(v) = \ln M(v) = \ln\left(\frac{1}{2} + \frac{1}{2}e^v\right)$$

The Chernoff bound for the tail of our density function is given by [KLEI75] as

$$P[Y \geq y] \leq e^{-vy + N\gamma(v)}$$

Since this inequality is good for any value of $v \geq 0$, we should choose v, as in [KLEI75], to create the tightest possible bound. This is done by differentiating the exponent and setting it to zero. We then find the optimum relationship between v and y as

$$y = N\gamma^{(1)}(v)$$

Hence, we have

$$Prob[Y \geq y] \leq e^{N\left[\gamma(v) - v\gamma^{(1)}(v)\right]}$$

We let $y = \frac{N}{2} + N\epsilon$, and $0 < \epsilon < \frac{1}{2}$. In order to find the optimum relationship between v and y, we let

$$y = N\gamma^{(1)}(v)$$

$$= N\frac{\frac{1}{2}e^{v}}{\frac{1}{2}+\frac{1}{2}e^{v}} = N\frac{e^{v}}{1+e^{v}}$$

Solving for v, we get

$$v = \ln\frac{y}{N-y}$$

$$= \ln\frac{\frac{1}{2}+\epsilon}{\frac{1}{2}-\epsilon}$$

Thus,

$$Prob[Y \geq y] = Prob[Y \geq N\gamma^{(1)}(v)]$$

$$\leq e^{N\left[\ln\left(\frac{1}{2}+\frac{1}{2}e^{v}\right) - v\frac{e^{v}}{1+e^{v}}\right]}$$

Figure 5.2a shows several curves of P versus ϵ with various values of N where P is the upper bound on the tail probabilities such that $Prob\left[\,Y > \left(\frac{N}{2} + N\epsilon\right)\right] \leq P$. We note that, as N increases, the probability is concentrated closer to $y = \left\lfloor \frac{N+1}{2} \right\rfloor$ and the bound on the tail probability falls faster. Figure 5.2b shows the probability of a randomly selected process graph having more than y levels.

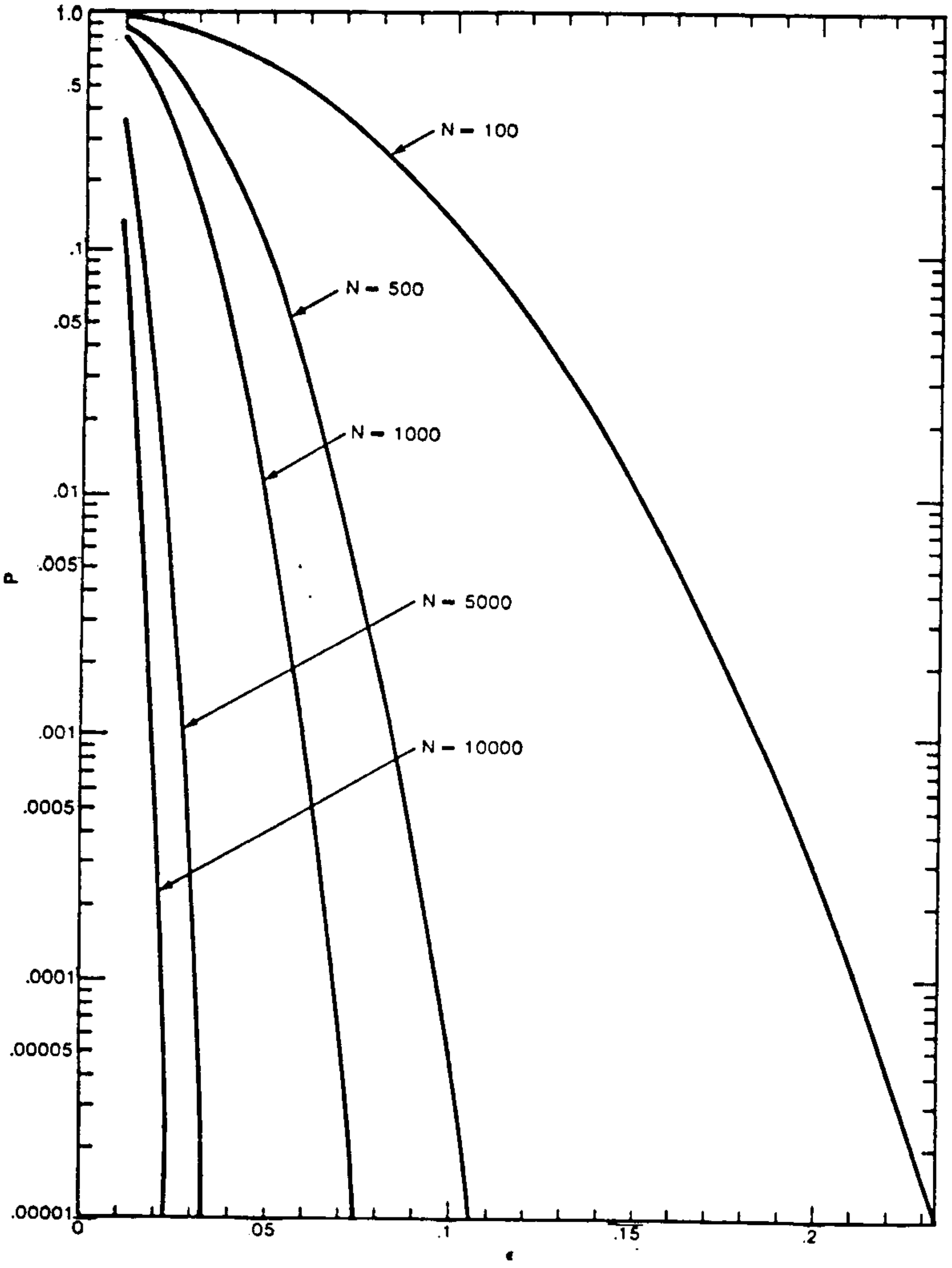


Figure 5.2a Chernoff Bound

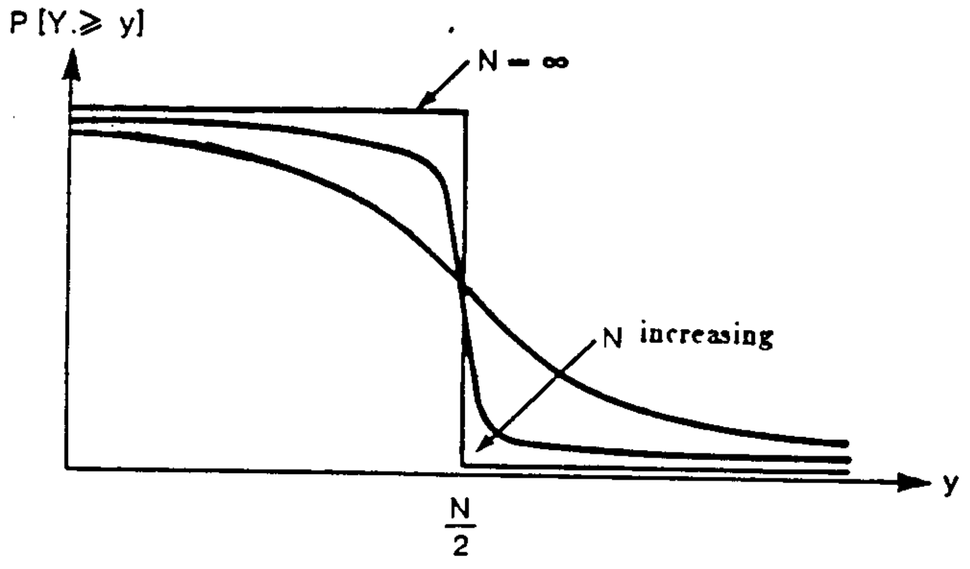


Figure 5.2b Prob [$Y \geq y$]

5.2.5 Generation of Random Process Graphs

Random process graphs can be produced by at least the following four methods. In the first method, we are given the total number of levels for all graphs and a probability distribution of the width of each level. After selecting a random number of tasks for each level, precedence relationships are created by randomly connecting the nodes of the adjacent levels with the direction of all edges pointing toward the terminal node.

For the second method, we initially produce a connected undirected random graph with a given number of tasks and edges. We next select a given node as the initial task and assign edge directions to the nodes one hop away. This is followed by assigning edge directions from nodes that are one hop away to nodes that are two hops away. This procedure is repeated until the edge of the node farthest away from the initial node has been assigned a direction. Any remaining undirected edges can have either direction. Then, in order to conform with a normal process graph, we add an additional terminal node. An edge will be added to this terminal node from all nodes with an out-degree of zero.

If the number of precedence relationships are large, the resulting process graph will generally have a large number of levels. This is due to the constraint that no precedence relationships are allowed between any two tasks on the same level. If this edge does exist, one of the nodes is pushed down to the following level. In fact, if the number of edges equals $\frac{N(N-1)}{2}$, only one process graph can be generated -- a linear chain of N tasks.

Drawing from the theory of branching process [HARR63], the third method uses the Galton-Watson branching process to create a random graph. In this process, level one has one task. Then, for each task at level i, it has the probability P_k to create k new tasks at the $(i+1)^{st}$ level, where $k=0,1,2,\cdots$. An edge connects each of the new tasks with the creating task. If a task creates no new task, it is extinguished, and it has a precedence relationship to any task on the next level. A difficulty with this method is that there is a possibility that the process graph will have an infinite number of levels. If a finite-level process graph is found, it is, by our definition, a structured process graph.

Finally, Dodin [DODI81] proposed another method of creating random process graphs with N nodes and M edges. In his method, the adjacency matrix that represents the precedence relationships is created in the following two ways:

1. Deletion Method
   Create an adjacency matrix with the upper triangle full of 1's. Randomly delete $\frac{N(N-1)}{2} - M$ edges on the condition that there exist at least one edge into and one edge out of any node.

2. Addition Method
   Distribute one edge to nodes (1,2) and one edge to nodes (N-1, N), and randomly distribute the remaining $M - 2$ edges to the upper triangle of the adjacency matrix on the condition that there exist at least one edge into and one edge out of any node.

From this adjacency matrix, a process graph is generated by mapping all the edges in the matrix onto a set of N nodes enumerated from 1 to N. Because only the upper triangle of the adjacency matrix can have 1's, the resulting directed graph is also guaranteed to be acyclic.

5.3 Fixed Task Service Time (k, G^*, x, $P = \infty$ and λ, G^*, x, $P = \infty$)

We have shown in the last section that, as N becomes large, the probability that a random process graph (i.e., given only N, not M or L, where M is the number of precedence relationships and L is the number of levels in the process graph) will have y levels, where $N(\frac{1}{2} - \delta) \leq y \leq N(\frac{1}{2} + \delta)$ for small δ, approaches 1. Since we are assuming an infinite number of processors and constant task service times, the actual shape of the process graph will not affect the system time as long as the number of levels is given (i.e., close to $\frac{N}{2}$ percentage-wise). All tasks in a given level will be executed concurrently when the tasks in their previous level have been completed. Hence, the system time of a random process graph with N tasks and constant task service time requirements depends solely on the number of levels in the process graph. Assuming the constant service time is normalized to one unit of time, we have

$$\lim_{N\to\infty} Prob\left[N\left(\frac{1}{2} - \delta\right) \leq S \leq N\left(\frac{1}{2}+\delta\right)\right] = 1$$

for arbitrary small δ.

If jobs arrive from a Poisson source with a mean arrival rate of λ, we have an $M/G/\infty$ [KLEI75] system. As in the constant number of jobs case above, the system time S approaches $\frac{N}{2}$ as N becomes large. Hence, from $M/D/\infty$ results, we have the equilibrium probability that k jobs are in the system as

$$P_k = \frac{(\lambda S)^k}{k!} e^{-\lambda S}$$

The concurrency measure in both cases (k or λ) is the measurement of the average width, $\overline{w}$, of random process graphs. Since

$$\overline{w} = \frac{N}{\overline{L}}$$

where $\overline{L}$ is the average number of levels in random process graphs and we know

$$\lim_{N\to\infty} Prob\left[N(\frac{1}{2} - \delta) \leq \overline{L} \leq N(\frac{1}{2} + \delta)\right] = 1$$

for small δ, we have $\lim_{N\to\infty} \overline{w} = 2$ or $\lim_{N\to\infty} \sigma = \frac{1}{2}$.

Hence, as the number of tasks per job becomes large, the concurrency measure takes on the value of one half.

5.4 Random Task Service Times (k, G^*, x^*, $P = \infty$)

5.4.1 Bounds on the Average System Time without the Number of Precedence Relationships

In this section, an upper bound and a lower bound on the average system time are found for the random process graph with N tasks and task service times that is exponentially distributed.

5.4.1.1 Upper Bound

We wish to find an upper bound on the average system time of an N-task process graph. From the Chernoff bound we know the probability of a randomly chosen process graph having more than y levels in its arrangement. Thus, for a specific y, if we can find an upper bound, then this bound should be correct for any process graph with probability of $1\text{-}Prob[Y \geq y]$. In this section we obtain an upper bound for process graphs with the number of levels equal to or less than y. As $N \rightarrow \infty$ we can let y be arbitrarily close to (but greater than) the mean number of levels, m, and the probability that the average system time of any randomly selected process graph will be smaller than this upper bound will approach one.

The following two lemmas provide an upper bound on the average system time for arrangements with less than or equal to y levels where $m \leq y \leq N$.

Lemma 5.8

Given N tasks, and y levels for a process graph, if we assign $\dfrac{N}{y}$ tasks to each level, then the resulting *forced synchronization time* (FST) is the maximum average system time with respect to the other arrangements of the tasks with y levels.

Forced synchronization time is defined to be the time required to process a process graph such that each task in a given level is being blocked by all the tasks in the previous level. In other words, we are forcing the tasks to be executed one level at a time (with all tasks in a given level waiting for the slowest task in the previous level to complete before they all start execution).

Proof of Lemma 5.3

Since the tasks have been assumed to have exponential service times with a mean of $\frac{1}{\mu}$, the blocking time of a task with d tasks blocking it is (see Section 4.3.1)

$$\int_0^\infty \left[1-(1-e^{-\mu t})^d\right] dt = \frac{1}{\mu}\sum_{i=1}^{d}\frac{1}{i}$$

With n_i tasks in each level i, $i = 1,2, \cdots ,y$, we have the upper bound on the total average system time S as

$$S_{UB} = \frac{1}{\mu}\sum_{i=1}^{n_1}\frac{1}{i} + \frac{1}{\mu}\sum_{i=1}^{n_2}\frac{1}{i} + \cdots + \frac{1}{\mu}\sum_{i=1}^{n_y}\frac{1}{i}$$

subject to the conditions:

$$\sum_{j=1}^{y} n_j = N$$

and

$$n_j > 0 \quad \forall j$$

We must show that $n_j = \frac{N}{y}$, for each level j, maximizes S_{UB}.

Assuming $n_j = \frac{N}{y}$, for all j, gives the maximum S_{UB},

$$S_{UB} = \frac{1}{\mu}\sum_{j=1}^{y}\sum_{i=1}^{n_j}\frac{1}{i}$$

$$= \frac{1}{\mu} y \sum_{i=1}^{N/y}\frac{1}{i}$$

Suppose there exists another arrangement such that its FST S^* is larger than S_{UB}. Let

$$n_1^*,\ n_2^*,\ \cdots ,\ n_s^*,\ n_{s+1}^*,\ \cdots ,\ n_t^*,\ n_{t+1}^*,\ \cdots ,n_y^*$$

be the arrangements, where

$$n_i^* > \frac{N}{y} \qquad \text{for } 1 \le i \le s$$

$$n_i^* = \frac{N}{y} \qquad \text{for } s+1 \le i \le t$$

$$n_i^* < \frac{N}{y} \qquad \text{for } t+1 \le i \le y$$

then

$$S^* = S_{UB} + \frac{1}{\mu}\sum_{i=1}^{a}\frac{1}{\frac{N}{y}+w_i} - \frac{1}{\mu}\sum_{i=1}^{b}\frac{1}{\frac{N}{y}-w_i}$$

where $w_i \ge 1 \ \forall i$, $a = \sum_{k=1}^{s} n_k^* - s\frac{N}{y}$ is the total number of additional nodes added to levels 1 to s, and $b = (y-t)\frac{N}{y} - \sum_{k=t+1}^{y} n_k^*$ is the total number of nodes taken out of levels $(t+1)$ *to* y. Since the total number of tasks remain constant, each additional task over $\frac{N}{y}$ for a level i where i is between 1 and s, one task must be taken out of another level j where j is between $t+1$ and y. Therefore, $a=b$. Now,

$$\frac{1}{\frac{N}{y}+1} \ge \frac{1}{\frac{N}{y}+w_i} \qquad w_i \ge 1$$

and

$$\frac{1}{\frac{N}{y}-1} \le \frac{1}{\frac{N}{y}-w_i} \qquad 1 \le w_i \le \frac{N}{y}$$

Thus,

$$S^* \le S_{UB} + a\frac{1}{\mu}\left(\frac{1}{\frac{N}{y}+1}\right) - b\frac{1}{\mu}\left(\frac{1}{\frac{N}{y}-1}\right)$$

$$= S_{UB} + a\frac{1}{\mu}\left(\frac{1}{\frac{N}{y}+1} - \frac{1}{\frac{N}{y}-1}\right)$$

Since $\frac{1}{\frac{N}{y}+1} < \frac{1}{\frac{N}{y}-1}$, we have $S^* < S_{UB}$ which contradicts the assumption. Hence, we have shown that the arrangement

$$n_j = \frac{N}{y} \qquad \forall j$$

gives us the maximum FST.

In the cases when n_j is not an integer in the above lemma, the arrangement that gives the maximum FST is constructed as follows. Let $u = \textit{Remainder of } \frac{N}{y}$, $w = \textit{Integer part of } \frac{N}{y}$, then $n_j = w+1$ for $j = 1, 2, , \cdots , u$ and $n_j = w$ for $j = u+1, \cdots , y$. From the proof of the above lemma, we know

$$\frac{1}{\mu}\, u \sum_{j=1}^{w+1} \frac{1}{j} + \frac{1}{\mu}\,(y-u) \sum_{j=1}^{w} \frac{1}{j} \le \frac{1}{\mu}\, y \sum_{j=1}^{\frac{N}{y}} \frac{1}{j}$$

Therefore, the arrangement $n_j = \frac{N}{y}$ $\forall j$ still gives us the maximum FST for any process graph having y levels even if there is a possibility that no process graph can have fractional nodes in a level.

Lemma 5.4

Given the FST calculated in Lemma 5.3 for an N-task and y-level process graph, it is also the maximum FST for any process graph with less than y levels.

Proof

This lemma can be formulated as the following nonlinear optimization problem

$$\underset{y_1}{Max}\ S = y_1 \left(\frac{1}{\mu} \sum_{i=1}^{\frac{N}{y_1}} \frac{1}{i} \right)$$

$$\textit{subject to}\ \ y_1 \le y$$

We know that the harmonic series [KNUT73a], $\sum_{i=1}^{n} \frac{1}{i}$, can be approximated by

$$\sum_{i=1}^{n} \frac{1}{i} = \ln n + \Phi + \frac{1}{2n} - \frac{1}{12n^2} \cdots$$

where Φ is the Euler's constant ($= 0.57721 \cdots$). Therefore,

$$S \cong \frac{1}{\mu} y_1 \left[\ln \frac{N}{y_1} + \Phi + \frac{1}{2\frac{N}{y_1}} - \frac{1}{12\left(\frac{N}{y_1}\right)^2} \right]$$

Now, we assume y_1 is continuous,

$$\frac{\partial S}{\partial y_1} = \frac{1}{\mu} y_1 \left(\frac{-\frac{N}{y_1^2}}{\frac{N}{y_1}} + \frac{1}{2N} - \frac{y_1}{6N^2} \right)$$

$$+ \frac{1}{\mu} \left(\ln \frac{N}{y_1} + \Phi + \frac{y_1}{2N} - \frac{y_1^2}{12N^2} \right)$$

$$= \frac{1}{\mu} \left(\Phi - 1 + \ln \frac{N}{y_1} + \frac{y_1}{N} - \frac{y_1^2}{4N^2} \right)$$

For $1 \le y_1 \le N$, we find that $\frac{\partial S}{\partial y_1} > 0$, or the slope of S versus y_1 is positive in the region $1 \le y_1 \le N$. Since at $y_1 = 1$,

$$S = \frac{1}{\mu} y_1 \sum_{i=1}^{\frac{N}{y_1}} \frac{1}{i} = \frac{1}{\mu} \sum_{i=1}^{N} \frac{1}{i}$$

we have $S > 0$ as an increasing function with respect to y_1. The condition $y_1 \le y$ implies that the maximum S occurs at $y_1 = y$. Therefore, the maximum FST obtained for an N-task process graph with y levels is also the maximum FST for all N-task process graphs with less than y levels.

|||

The next theorem follows as the result of the two above lemmas.

Theorem 5.5

An upper bound for the average system time of an N-task process graph with y levels or less and exponential task times with mean $\frac{1}{\mu}$ is

$$S_{UB}(y) = y \frac{1}{\mu} \sum_{i=1}^{\frac{N}{y}} \frac{1}{i}$$

This upper bound is good only for process graphs with less than or equal to y levels. But since the number of arrangements in a process graph is Gaussian distributed, we expect that for $y > \frac{N}{2}$, as $N \rightarrow \infty$, the probability that a randomly chosen process graph has a higher average system time than $S_{UB}(y)$ gets smaller. In fact, we show in the next theorem that as $N \rightarrow \infty$, and $y = \frac{N}{2} + N\delta$ for any small positive δ, the tail probability $Prob[Y \geq y]$, or the probability that a process graph has more than y levels, approaches zero.

Theorem 5.6

For $y = \frac{N}{2} + N\delta$, where δ is a real positive number, and Y is the number of levels in a randomly selected process graph, we have

$$\lim_{N \rightarrow \infty} Prob[Y \geq y] \rightarrow 0$$

Proof:

The Chernoff bound gives us

$$Prob[Y \geq y] \leq e^{N\left[\ln\left(\frac{1}{2}+\frac{1}{2}e^v\right) - v\frac{e^v}{1+e^v}\right]}$$

In order for $\lim_{N \rightarrow \infty} Prob[Y \geq y] \rightarrow 0$ we must show

$$\ln\left(\frac{1}{2}+\frac{1}{2}e^v\right) < v\frac{e^v}{1+e^v}$$

Since $v = \ln\frac{\frac{1}{2}+\epsilon}{\frac{1}{2}-\epsilon}$ and since $0 < \epsilon < \frac{1}{2}$ (from Section 5.2.4), we have $v > 0$.

Let $z = e^v$, or $v = \ln z$, where $z > 1$. The inequality we must show becomes

$$\ln\left(\frac{1}{2}+\frac{1}{2}z\right) < \frac{vz}{1+z}$$

$$(1+z)\ln\left(\frac{1}{2}+\frac{1}{2}z\right) < z\ln z$$

Taking the exponential of both sides, we get

$$\left(\frac{1}{2}+\frac{1}{2}x\right)^{(1+x)} < x^x$$

For $x>1$, x^x is always greater than $\left(\frac{1}{2}+\frac{1}{2}x\right)^{(1+x)}$.

Hence we have shown $\lim_{N\to\infty} Prob[Y \geq y] \to 0$ for any $\delta>0$.

|||

With N a large number, we can then state that the upper bound on the average system time is

$$\lim_{N\to\infty} Prob\left[N_L \frac{1}{\mu} \sum_{i=1}^{\frac{N}{N_L}} \frac{1}{i} \leq S_{UB} \leq N_U \frac{1}{\mu} \sum_{i=1}^{\frac{N}{N_U}} \frac{1}{i}\right] \to 1$$

where $N_L = N(\frac{1}{2} - \delta)$, $N_U = N(\frac{1}{2} + \delta)$ and δ is any arbitrarily small number. That is,

$$S_{UB} \cong \frac{3}{4}\frac{N}{\mu}$$

5.4.1.2 Lower Bound

For an N-task process graph, if we are given the number of levels y, then we know that the minimum amount of average processing time is $y\frac{1}{\mu}$ where $\frac{1}{\mu}$ is the average processing time of a task. With respect to all arrangements of the process graph with N tasks, this average processing time will be a lower bound with probability $1-Prob[Y \leq y]$. Since the number of arrangements with respect to the number of levels is Gaussian distributed with mean $\frac{N}{2}$, from the symmetry, we have $Prob[Y \leq \frac{N}{2}-y] = Prob[Y \geq \frac{N}{2}+y]$. Hence, all the properties of the Chernoff bound discussed in the last section can be applied here also. Specifically, we can let $N\to\infty$, for arbitrary small δ, and $\delta>0$, then

$$\lim_{N\to\infty} Prob\left[N\left(\frac{1}{2}-\delta\right)\frac{1}{\mu} \leq S_{LB} \leq N\left(\frac{1}{2}+\delta\right)\frac{1}{\mu}\right] \to 1$$

This is true since the tail probability approaches zero as N becomes very large. Thus, $S_{LB} \cong \frac{N}{2\mu}$.

5.4.1.3 Discussion

In last two sections, we see that for $N >> 1$, the average system time for the case k, G^*, x^*, $P = \infty$ is bounded by

$$\frac{1}{2}\,\frac{N}{\mu} \leq S \leq \frac{3}{4}\,\frac{N}{\mu}$$

with high probability. In terms of speedup, it is bounded by

$$1\,\frac{1}{3} \leq \frac{1}{\sigma} \leq 2\,.$$

So, on the average, the best speedup we can achieve is two and the least speedup is $1\,\frac{1}{3}$.

5.4.2 Upper Bound with a Fixed Number of Precedence Relationships

We have studied random process graphs without considering the number of precedence relationships in the previous section. We have obtained some general properties of the arrangements of the tasks for process graphs and bounds on the average system time. The upper bound and the lower bound obtained are probabilistic such that as the number of tasks becomes large, the more certain we are regarding these bounds. However, if we now include the number of precedence relationships, we can improve these bounds

In this section, the number of precedence relationships are introduced into the model. We will obtain a tighter bound using the number of precedence relationships, the number of levels and the number of tasks in a random process graph as parameters. We first develop the idea of *minimally connected process graphs.* The number of edges required for the minimally connected process graph are studied as well as how additional edges can be added to it. Next, an algorithm is presented which gives a construction method for a process graph G^* with N nodes, M edges and L levels. We will prove that the forced synchronization time (FST) obtained from process graph G^* is indeed an upper bound on the average system time of all random process graphs with N nodes, M edges and L levels.

5.4.2.1 Minimally Connected Process Graph

Given any arrangement, the M edges can connect only a limited set of tasks. No edges, for example, are allowed to connect any tasks within the same level of a process graph. In Figure 5.3, we see the 26 legal places where the precedence relationships can be placed in a particular 3 level, 9 task graph: six positions between the first and the second level, eight between the second level and the third level, and twelve between the first level and the third level.

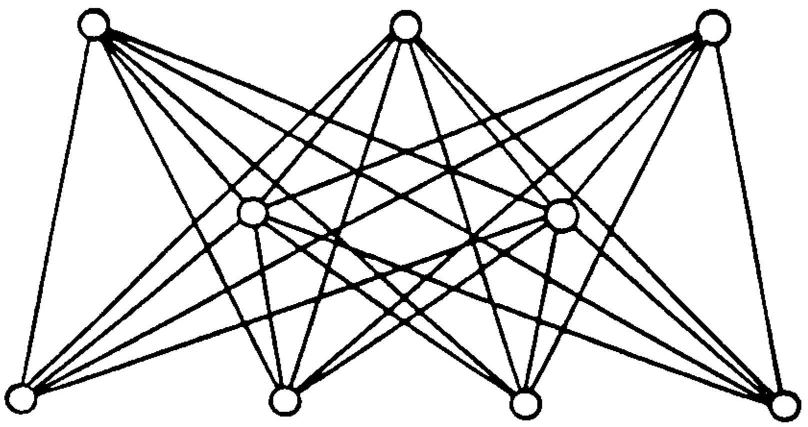


Figure 5.3 Legal Places for Precedence Relationships

If we allow the M edges to be randomly distributed among all these legal places, we often find that the process graph is not even connected. Even with a large enough number of edges, such as $M > N \log N$, there is no guarantee the resulting process graph is connected (although it is highly likely).

For purposes of calculating bounds on the average system time, the underlying arrangement must be connected such that each node is maintained at its proper level in the process graph. A node, j, is said to be in i^{th} level if there exists a shortest path from the initial node to node j such that the number of nodes in this path equals i. We define M_c to be the *minimum* number of edges required to fix all the nodes of a particular process graph in this proper level. All edges are between nodes of the adjacent levels instead of between nodes of non-adjacent levels because of our definition of level and because of the following Lemma.

Lemma 5.7

The blocking time of an edge between two neighboring levels is greater than the blocking time of the same edge between two levels not neighboring each other.

Proof.

Suppose node i at level r is being blocked by node j which is at level $r-2$. If there exist two other edges (j,k) and (k,i) for any node k at level $r-1$, then edge (j,i) presents no blocking to node i (See Figure 5.4) for the following reasons.

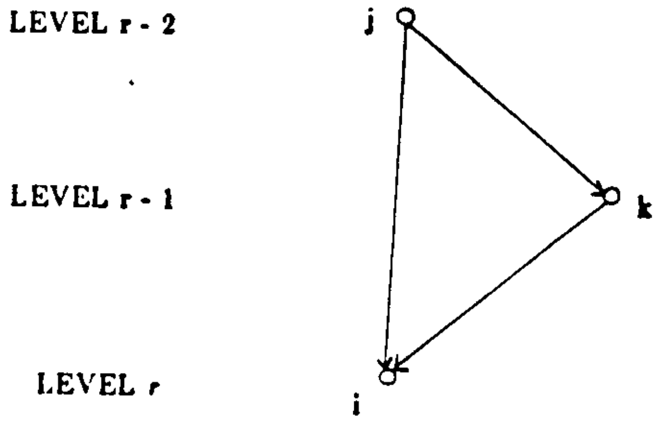


Figure 5.4 Reduced Blocking Effect

As soon as node j is completed, node k starts execution. Since node k is still blocking node i, the release of the blocking from node j to node i does not allow node i to start execution. If no such indirect blocking edges exist, we notice the fact that the blocking effect of the edge (j,i) is reduced partially by the average task time of other tasks in level $r-1$ which do block node i. Hence, the blocking due to the edge (j,i) is not worse than any edge (k,i) if node k belongs to level $r-1$.

Since we will be looking for an upper bound on the average system time with a limited number of precedence relationships and since the edges between adjacent levels result in greater blocking, we assume all edges are between two adjacent levels.

Figure 5.5 shows some examples of minimally connected process graphs. Each edge in Figure 5.5 is necessary in order to fix the nodes in their proper levels within the process graph. By deleting any one edge, the resulting process graph will be either disconnected or at least one node is no longer in a path from the initial node to the terminating node.

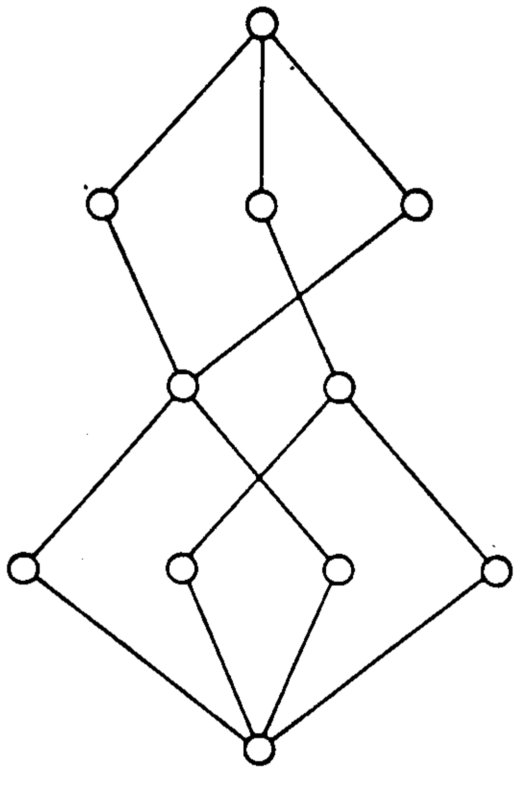


Figure 5.5 Minimally Connected Process Graph

From Lemma 5.7, we can calculate the exact value of M_c for a process graph. Since we are looking for an upper bound of the average system time, we place all the edges between nodes in adjacent levels. Define n_i to be the number of tasks in level i, for $i = 1, 2, \cdots, L$.

For any two adjacent levels, say levels i and $i+1$, if $n_i \geq n_{i+1}$, then there must be at least n_i edges between i^{th} level and $(i+1)^{th}$ level. Otherwise, at least one of the nodes, say node j, in the i^{th} level will have no edge leaving it and the path from the initial node toward the terminating node stops at node j. This contradicts the definition of a task in a process graph. Therefore, there must be at least n_i edges between levels i and $i+1$.

If, on the other hand, $n_i < n_{i+1}$, then there must be at least n_{i+1} edges between i^{th} level and $(i+1)^{th}$ level. If there are less than n_{i+1} edges, then at least one of the nodes, say node j, in $(i+1)^{th}$ level has no edges entering it. It cannot be in any path from the initial node to the terminating node. This contradicts the definition of a task in a process graph, therefore, there must be at least n_{i+1} edges between levels i and $i+1$.

By selecting the larger of n_i and n_{i+1} to be the minimum number of edges between levels i and $i+1$, we have enough edges to keep all nodes in the i^{th} and $(i+1)^{th}$ levels properly defined. Summing over all levels, we have

$$M_c = \max\{n_1, n_2\} + \max\{n_2, n_3\} + \cdots + \max\{n_{L-1}, n_L\}$$

The rules for making the minimally connected process graph are:

Let levels i and $i+1$ be adjacent in the process graph,

1. $n_i < n_{i+1}$

   For each node in level i we assign an integer $1, 2, 3, \ldots, n_i$ and for each node in level $i+1$ we also assign an integer $1, 2, 3, \ldots, n_{i+1}$. Let j represent a node in level $i+1$ and s equal to the remainder of $\frac{j}{n_i}$. Make a connection between node s of level i and node j of level $i+1$.

2. $n_i = n_{i+1}$

   For each node in level i, connect it to any node in level $i+1$ with indegree of zero.

3. $n_i > n_{i+1}$

   For each node in level i and $i+1$, we assign an integer $1, 2, \ldots, n_i$ and $1, 2, 3, \ldots, n_{i+1}$

respectively. Let j represent a node in level i, and s equal to the remainder of $\frac{j}{n_{i+1}}$. Make a connection between node j of level i and node s of level i+1.

In all the cases above, each node in the i^{th} level has at least one edge going to some node in the $(i + 1)^{th}$ level and each node in the $(i + 1)^{th}$ level has at least one edge entering it from some node in the i^{th} level. By extending this method to all levels, all nodes, except the initial and terminating nodes, have at least one edge entering and one edge leaving it. Hence each node is on a path between the initial and terminating node and each node is held in its proper level in the process graph. Thus, we have a method for creating minimally connected process graphs.

After we place the M_c edges into the arrangement of a process graph, there are still many pairs of nodes between the adjacent levels where an edge can be placed. We call each of these pairs as the *empty edge slot* (EES). All M - M_c edges are placed randomly into the EESs.

Two lemmas relating the values of M_c with the other parameters of a process graph are:

Lemma 5.8

For any process graph with N nodes and L levels, the maximum number M_c occurs in process graphs which have node arrangements such that the levels with more than one node are separated with at least one level which has only one node. In these cases,

$$Max\ M_c = 2N - L - 1 \tag{5.1}$$

Proof.

We are maximizing

$$M_c = \sum_{i=1}^{L-1} Max\left(n_i,\ n_{i+1}\right)$$

with respect to the numbers $\left\{ n_i \right\}$

$$Subject\ to\ \sum_{i=1}^{L} n_i = N$$

Consider 4 adjacent levels j-1, j, j+1, and j+2, such that n_{j-1}, n_j, n_{j+1}, n_{j+2} are respectively the number of nodes in each level. We assume $n_j > n_{j+1}$, then there are four cases relating the values of n_{j-1} and n_j and the values of n_{j+1} and n_{j+2} (The arguments for the case $n_j \leq n_{j+1}$ is similar to the following discussion). We must show that the arrangement with $n_j' = n_j + n_{j+1} - 1$ and $n_{j+1}' = 1$ with other levels having the same number of nodes gives us a higher M_c.

Case I. $n_j \geq n_{j-1}$ and $n_{j+1} \geq n_{j+2}$.

In calculating Max M_c for these four levels, we have Z edges between these three pairs of levels, where

$$Z = n_j + n_j + n_{j+1}$$

But if we change the arrangement to $n_{j-1}, n_j + n_{j+1} - 1, 1, n_{j+2}$ then the number of edges becomes

$$Z' = n_j + n_{j+1} - 1 + n_j + n_{j+1} - 1 + n_{j+2}$$

But Z' is larger than Z since

$$Z' - Z = 2n_j + 2\,(n_{j+1} - 1) + n_{j+2} - (2n_j + n_{j+1})$$

$$= n_{j+1} - 2 + n_{j+2} \geq 0$$

Thus, the arrangements of $n_j' = n_j + n_{j+1} - 1$ and $n_{j+1}' = 1$, while fixing the values of other n_i's, will give a higher or equal value of M_c.

Case II. $n_{j-1} > n_j > n_{j+1} > n_{j+2} \geq 1$

For this case, we have

$$Z = n_{j-1} + n_j + n_{j+1}$$

and by shifting $(n_{j+1} - 1)$ nodes from $(j + 1)^{st}$ level to j^{th} level, we have

$$Z' = \begin{cases} n_{j-1} + n_j + (n_{j+1} - 1) + n_{j+2} & \text{if } n_{j-1} > n_j + n_{j+1} - 1 \\ 2(n_j + n_{j+1} - 1) + n_{j+2} & \text{if } n_{j-1} \leq n_j + n_{j+1} - 1 \end{cases}$$

and

$$Z' - Z = \begin{cases} n_{j+2} - 1 \geq 0 & \text{if } n_{j-1} > n_j + n_{j+1} - 1 \\ n_j + n_{j+1} - 2 + n_{j+2} - n_{j-1} \geq 0 & \text{if } n_{j-1} \leq n_j + n_{j+1} - 1 \end{cases}$$

Hence, $Z' \geq Z$ or the new arrangement will give a possibly higher value of M_c.

Case III. $1 \leq n_{j-1} < n_{j+1} < n_j < n_{j+2}$

The argument for this case is similar to Case II above.

Case IV. $n_{j-1} > n_j > n_{j+1} > 1$ and $n_{j+2} > n_{j+1}$

In this case,

$$Z = n_{j-1} + n_j + n_{j+2}$$

and by shifting $(n_{j+1} - 1)$ nodes from $(j + 1)^{th}$ level to j^{th} level, we obtain

$$Z' = \begin{cases} n_{j-1} + n_j + n_{j+1} - 1 + n_{j+2} & \text{if } n_{j-1} > n_j + n_{j+1} - 1 \\ 2\, n_j + 2\, n_{j+1} - 2 + n_{j+2} & \text{if } n_{j-1} \leq n_j + n_{j+1} - 1 \end{cases}$$

Therefore,

$$Z' - Z = \begin{cases} n_{j+1} - 1 > 0 & \text{if } n_{j-1} > n_j + n_{j+1} - 1 \\ n_j + 2\, n_{j+1} - 2 - n_{j-1} > 0 & \text{if } n_{j-1} \leq n_j + n_{j+1} - 1 \end{cases}$$

or the new arrangement gives a higher value of M_c.

So for all four cases, we can always obtain a larger *Max* M_c by shifting nodes from the $(n_{j+1})^{th}$ level to the n_j^{th} level such that $n_j' = n_j + n_{j+1} - 1$ and $n_{j+1}' = 1$.

To see that Max M_c occurs when the levels with more than one node are separated by at least one level which has only one node, we consider four adjacent levels again. Suppose the four adjacent levels $j - 1$, j, $j + 1$, and $j + 2$ have 1, n_j, 1, n_{j+2} nodes respectively where $n_j > 1$ and $n_{j+2} \geq 1$. In calculating Max M_c for these four levels, we have a sum of

$$Z = n_j + n_j + n_{j+2}$$

By shifting one node from j^{th} level to $(j + 1)^{th}$ level, this summation becomes

$$Z' = (n_j - 1) + (n_j - 1) + Max\,(2,\, n_{j+2})$$

$$= 2\, n_j - 2 + Max\,(2,\, n_{j+2})$$

and

$$Z - Z' = 2 + n_{j+2} - Max\,(2,\, n_{j+2}) > 0$$

The value of M_c is smaller when we shift one (or more) node from the j^{th} level to the $(j + 1)^{th}$ level. Hence, Max M_c occurs when the levels with more than one node are separated by at least one level which has only one node.

The simplest arrangement of the N-node and L-level process graph with this property has $N-(L-1)$ nodes in one level between the second and the $(L-1)^{th}$ level and one node in each of the remaining $(L-1)$ levels. With this arrangement, we see that

$$Max\, M_c = (L-3) + 2(N-L+1)$$

or

$$Max\, M_c = 2N - L - 1$$

|||

Lemma 5.9

The minimum M_c occurs when the nodes are distributed evenly among all levels. Let

$$z = Remainder\ of\ \frac{N-2}{L-2}$$

$$x = \left\lceil \frac{N-2}{L-2} \right\rceil$$

$$y = \left\lfloor \frac{N-2}{L-2} \right\rfloor$$

Then

$$Min\, M_c = \left[(z+1)x + (L-z-2)y\right]$$

Proof.

We are minimizing

$$Min\, M_c = \sum_{i=1}^{L-1} \max(n_i, n_{i+1})$$

with respect to the numbers $\{ n_i \}$

$$Subject\ to\ \sum_{i=1}^{L} n_i = N$$

Suppose the $Min\, M_c$ occurred in an arrangement where four adjacent levels $j-1$, j, $j+1$, and $j+2$ have an equal number of nodes,

$$n_{j-1} = n_j = n_{j+1} = n_{j+2}$$

Now we show that by moving a node from the j^{th} level to the $(j+1)^{th}$ level, the value of M_c for the resulting arrangement will be larger.

In calculating Max M_c for these four levels, we have a sum of

$$Z = n_{j-1} + n_j + n_{j+1} = 3\ n_j$$

By moving one node from j^{th} level to $(j+1)^{th}$ level, we have a different sum of

$$Z' = n_{j-1} + (n_j + 1) + (n_j + 1) = 3\ n_j + 2 > Z$$

The same is true in cases

1. $n_{j-1} = n_j = n_{j+1} = x$ and $n_{j+2} = y$

2. $n_{j-1} = n_j = x$ and $n_{j+1} = n_{j+2} = y$

3. $n_{j-1} = x$ and $n_j = n_{j+1} = n_{j+2} = y$

Hence, we see that if we try to assign an 'equal' number of nodes to each level for all levels, we obtain the minimum M_c. Thus, besides one node in the initial level and one node in the terminating level, each of the first z successive levels will have x nodes and each of the other $(L - z - 2)$ levels will have y nodes. Finally, we have

$$Min\ M_c = \Big[(z+1)x + (L-z-2)y\Big]$$

|||

5.4.2.2 An Upper Bound

In Section 5.4.1.1, an upper bound for the average system time was obtained from the arrangement with an equal number of nodes per level. Indeed, this is also true in the case of process graphs with a fixed number of edges. We show this in the next theorem after giving an algorithm (Algorithm A) which constructs an N-node process graph with M edges in L levels.

ALGORITHM A

STEP 1: Distribute one node for the initial task and one node for the terminating task.

STEP 2: Let

$$z = Remainder\ of\ \frac{N-2}{L-2}$$

$$x = \left\lceil \frac{N-2}{L-2} \right\rceil$$

$$y = \left\lfloor \frac{N-2}{L-2} \right\rfloor$$

For each of the levels 2, 3, ... , $z+1$, place x nodes in them, and for each of the levels $z+2$, $z+3$, ... , $L-1$, place y nodes in them.

STEP 3: Use $M_c = (z+1)x + (L-z-2)y$ edges to minimally connect this arrangement of tasks.

STEP 4: Randomly distribute the remaining edges, $M - M_c$, uniformly among the EES's.

Theorem 5.10

If we construct a process graph with N nodes, M edges, and L levels according to ALGORITHM A, and assuming each task has an exponential service time distribution with mean $\frac{1}{\mu}$, then an upper bound on the average system time is

$$S_{UB} = \frac{1}{\mu}\left[\sum_{j=1}^{L} \sum_{i=1}^{t_j} \frac{1}{i}\right]$$

where

$$t_j = the\ maximum\ number\ of\ blocking\ edges\ into\ a\ node\ in\ level\ j$$

$$= \min\left\{\left\lceil b_j + (M-M_c)\frac{EES_j}{EES_T}\right\rceil, n_{j-1}\right\}$$

$$b_j = \max_q \left[b_q \mid q\ is\ a\ node\ in\ level\ j\right]$$

$$b_q = indegree\ of\ node\ q\ from\ the\ minimally\ connected\ process\ graph$$

EES_j = *number of empty edge slots between levels* $j-1$ and j

EES_T = *total number of empty edge slots*

n_{j-1} = *number of nodes in level* $j-1$

$n_1 = 1$

Proof.

Case 1) For $M \geq \max M_c$

In this case, there are enough edges such that b_j equals the number of nodes in the previous level, n_{j-1}, for all levels $j \geq 2$. Therefore $t_j = n_{j-1}$ for all $j \geq 2$. The proof that the arrangement generated by ALGORITHM A gives the largest FST is similar to the proof of Lemma 5.3 of the last section. Therefore, we only need to show that with $M \geq \max M_c$ we have enough edges for each level to force the maximum indegree of a node in that level to be equal to the number of nodes in the previous level (i.e. $b_j = n_{j-1}$). The number of extra edges is

$$M - M_c$$

$$\geq Max\, M_c - Min\, M_c$$

$$= (2N-L-1) - \left[(z+1)\left\lceil \frac{N-2}{L-2} \right\rceil + (L-z-2)\left\lfloor \frac{N-2}{L-2} \right\rfloor\right]$$

where z is the remainder of $\frac{N-2}{L-2}$. Since for z not equal to zero, the value of Min M_c becomes smaller, we shall let $z = 0$.

The inequality we wish to prove is then

$$1 + \frac{(2N-L-1) - (L-1)\frac{N-2}{L-2}}{L-3} \geq \frac{N-2}{L-2}$$

The left hand side represents the number of additional edges remaining after $Max\, M_c - Min\, M_c$ have been distributed equally between the $L-2$ levels plus one edge representing the minimal connection of the node. The right hand side represents the number of nodes per level. Multiplying both sides by $(L-3)$, we get

$$2N\text{-}L\text{-}1\text{-}(L\text{-}1)\frac{N\text{-}2}{L\text{-}2} \geq \left\lceil \frac{N\text{-}2}{L\text{-}2} \right\rceil (L\text{-}3) - (L\text{-}3)$$

$$2N \geq \left\lceil \frac{N\text{-}2}{L\text{-}2} \right\rceil (2L\text{-}4) + 4$$

$$= \frac{2NL - 4N}{L\text{-}2} = 2N$$

Which clearly is true. In other words, we have enough edges to force the maximum blocking for each level. Hence the FST obtained from ALGORITHM A does give an upper bound for the average system time.

Case 2) $Min\ M_c \leq M \leq Max\ M_c$

Let

$$f(k) = \sum_{j=1}^{k} \frac{1}{j} \qquad k \geq 1$$

$$f(0) = 1$$

We are maximizing $\sum_{j=1}^{L} f(d_j)$ subject to $\sum_{q=1}^{L} d_q \leq constant$, where d_q is the maximum indegree at level q. From the proof of Lemma 5.3, we find that the arrangement of d_q that gives the maximum sum of $f(d_q)$ is

$$d_q = \frac{constant}{L\text{-}2} \qquad q = 3,4,\cdots,L\text{-}1$$

and $d_1 = 0$ and $d_2 = 1$. This arrangement is exactly what ALGORITHM A generated. So for $Min\ M_c \leq M \leq Max\ M_c$, S_{UB} obtained from the FST of the arrangement generated by ALGORITHM A is also an upper bound.

|||

Figure 5.6 gives the minimally connected process graphs for some of the arrangements with $N = 10$ and $L = 6$. Table 5.1 shows simulation results and the predicted upper bound for the average system time with $M = 12$ for the process graphs with the arrangements shown in Figure 5.6. Since the arrangement of nodes in Case I in Figure 5.6 requires a minimum of 13 edges to be minimally connected and $M < 13$, no process graphs can be created with 12 edges, and we cannot find its average system time.

Case	Simulated Average System Time	Calculated Upper Bound
I	not feasible	378.7
II	364.24	378.7
III	362.98	378.7
IV	367.53	378.7

Table 5.1

5.4.3 Comparison of The Two Upper Bounds

Two upper bounds have been obtained with different parameters. In Section 5.4.1.1, an upper bound was obtained through the probabilistic method on the likelihood of a process graph having a certain number of levels or less. In Section 5.4.2.2, another upper bound was obtained with a fixed number of nodes, edges, and levels. Since the latter method uses more information, its bound should be tighter than the bound obtained by the former method.

Let S_{UB} represent the upper bound obtained in Section 5.4.1.1 and S_{UBM} represent the upper bound obtained in Section 5.4.2.2.

Theorem 5.11

$S_{UBM} \leq S_{UB}$ if $L \leq \left\lceil \frac{N}{2} \right\rceil$.

Proof.

Case 1) $L < \left\lceil \frac{N}{2} \right\rceil$

From the calculation of S_{UB}, we know that S_{UB} is the largest FST for any process graph having $\left\lceil \frac{N}{2} \right\rceil$ levels.

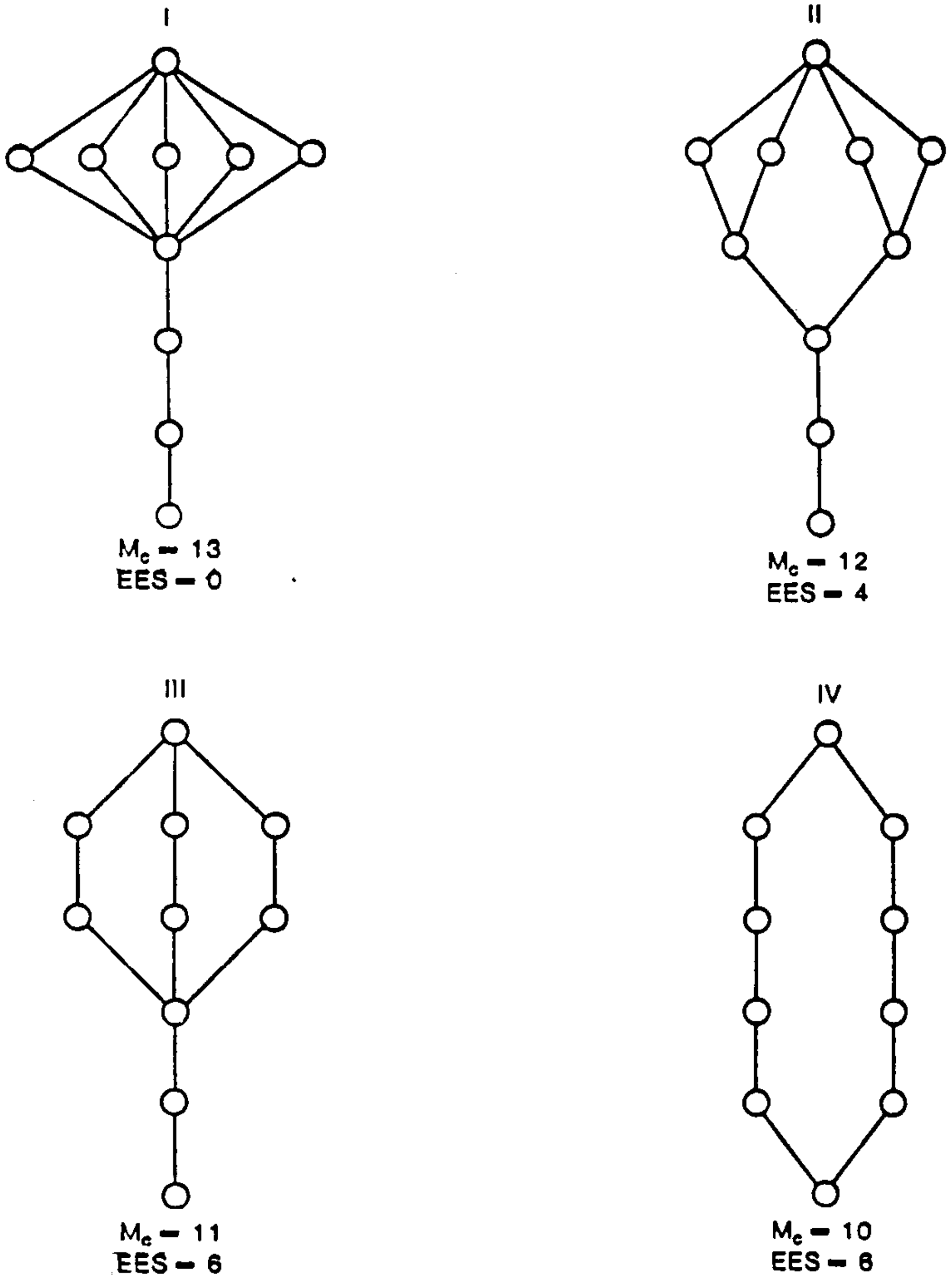


Figure 5.6 Minimally Connected Process Graph with $N = 10$ and $L = 6$

Since the number of levels is smaller than $\left\lceil \frac{N}{2} \right\rceil$, we have proved in Lemma 5.4 that S_{UB} is also the maximum FST for all levels less than $\left\lceil \frac{N}{2} \right\rceil$. Hence, $S_{UBM} \leq S_{UB}$.

Case 2) $L = \left\lceil \frac{N}{2} \right\rceil$

In this case, the arrangements used to obtain both bounds are the same. If the number of edges, M, is greater than or equal to max M_c, S_{UBM} is obtained with max FST of this arrangement or

$$S_{UBM} = S_{UB}$$

If $M <$ max M_c, there will not be enough edges at every level for the maximum FST. Thus,

$$S_{UBM} < S_{UB}$$

From the above two cases, we conclude $S_{UBM} \leq S_{UB}$ if $L \leq \left\lceil \frac{N}{2} \right\rceil$.

!!!

When $L > \left\lceil \frac{N}{2} \right\rceil$, it is likely that $S_{UBM} > S_{UB}$. The upper bound S_{UBM} uses more inoformation but it is a worse bound. This implies we did not use the information optimally in calculating S_{UBM}. The inequality ($S_{UBM} > S_{UB}$) is not always true because with different parameters of N, M, and L, it is possible to construct counter examples. Also note that since $L \cong \frac{N}{2}$ due to the law of large numbers, the case ($L >> \left\lceil \frac{N}{2} \right\rceil$) is not likely to occur.

5.5 Trade Off between Average System Time and Utilization of Processors for a Diamond-shaped Process Graph (k, G', z, $P < \infty$)

Given a process graph, if we have enough processors such that the number of processors, P, assigned to one job is greater than $\underset{i}{Max}\left\{ n_i \right\}$, then the average system time is just the time required to process L levels of nodes. The utilization of the P processors is, however, very low. In order to utilize each processor more, on the other hand, the average system time will increase. In this section, we will study this trade off by using the notion of "power" defined in Chapter 4. The average task time is assumed to be constant. So we define

$$Power = R = \frac{\rho}{S / \overline{X}}$$

where ρ is the efficiency of the P processors, S is the average system time, and $\overline{X}$ is the constant task service time (i.e., $\frac{S}{\overline{X}}$ is the normalized average system time).

If we can assume that the shape of process graphs is bounded by a parallelgram (as in Figure 5.7) which can be characterized by two parameters, L and m (where L is the number of levels in the process graph and m is the slope of the diamond enclosing the boundary tasks in this process graph). If the number of levels, L, is even, then we assume the $\left(\frac{L}{2}\right)^{th}$ level and the $\left(\frac{L}{2}+1\right)^{th}$ level have the same number of tasks.

Let $n(l)$ denote the number of tasks on the l^{th} level, and let $n(1) = 1$. Then, we know

$$n(l) - n(l-1) = 2\frac{1}{m} \qquad \begin{cases} 1 \le l \le \frac{L}{2} \text{ and } L \textit{ is even} \\ 1 \le l \le \frac{L+1}{2} \text{ and } L \textit{ is odd} \end{cases}$$

$$n(l) - n(l+1) = 2\frac{1}{m} \qquad \begin{cases} \frac{L}{2} < l \le L \quad \text{and } L \textit{ is even} \\ \frac{L+1}{2} < l \le L \quad \text{and } L \textit{ is odd} \end{cases}$$

Hence, for an even number of levels L,

$$n(l) = \begin{cases} 1 + \frac{2}{m}(l-1) & 1 \le l \le \frac{L}{2} \\ 1 + \frac{L}{m} & l = \frac{L}{2} + 1 \\ 1 + \frac{2}{m}\left(\frac{L}{2} - 1\right) - \frac{2}{m}\left(l - \frac{L}{2} - 1\right) & \frac{L}{2} + 1 < l \le L \end{cases}$$

and for an odd number of levels L,

$$n(l) = \begin{cases} 1 + \frac{2}{m}(l-1) & 1 \le l \le \frac{L+1}{2} \\ 1 + \frac{2}{m}\left(\frac{L+1}{2} - 1\right) - \frac{2}{m}\left(l - \frac{L+1}{2}\right) & \frac{L+1}{2} < l \le L \end{cases}$$

For example, let $L = 9$ and $m = 2$, then

$$n(l) = 1 + \frac{2}{m}(l-1) = l \qquad \text{for } 1 \le l \le \frac{L+1}{2}$$

and

$$n(l) = 1 + \frac{2}{m}\left(\frac{L+1}{2} - 1\right) - \frac{2}{m}\left(l - \frac{L+1}{2}\right)$$

$$= 10 - l \qquad \text{for} \frac{L+1}{2} < l \le L$$

Therefore, given m and L, the total number of tasks N in a process graph is

$$N = \begin{cases} 2\sum_{i=1}^{\frac{L}{2}}\left[1 + \frac{2}{m}(i-1)\right] & L \text{ even} \\ 2\sum_{i=1}^{\frac{L-1}{2}}\left[1 + \frac{2}{m}(i-1)\right] + \left[1 + \frac{2}{m}\left(\frac{L+1}{2} - 1\right)\right] & L \text{ odd} \end{cases}$$

$$= \begin{cases} L\left(1 - \frac{2}{m}\right) + \frac{L}{m}\left(\frac{L}{2} + 1\right) & L \text{ even} \\ (L-1)\left(1 - \frac{1}{m}\right) + \frac{1}{m}(L-1)\left(\frac{L+1}{2}\right) + 1 & L \text{ odd} \end{cases}$$

Next, we define

$B =$ *the width of the process graph at the widest level*

$$= \begin{cases} 1 + \frac{L}{m} - \frac{2}{m} & L \text{ even} \\ 1 + \frac{L+1}{m} - \frac{2}{m} & L \text{ odd} \end{cases}$$

$H =$ the number of levels in a process graph before all P processors become busy

$$= \left\lfloor \frac{P-1}{2/m} \right\rfloor + 1 = \left\lfloor \frac{(P-1)\,m+2}{2} \right\rfloor$$

Figure 5.7 shows a typical diamond-shaped process graph.

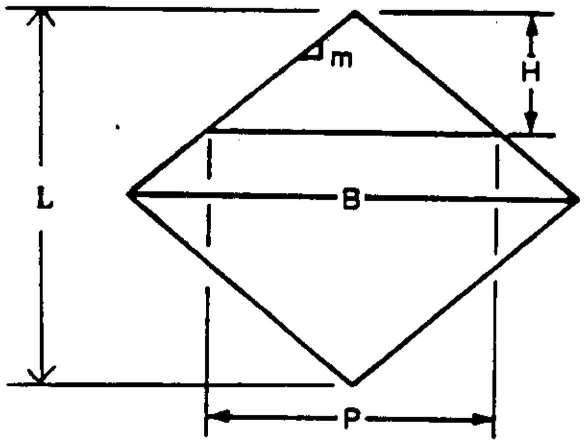


Figure 5.7 Diamond-shaped Process Graph

We find

$$\rho = \frac{2\,H\,\rho_1 + \left\lceil \frac{V}{P} \right\rceil}{2\,H + \left\lceil \frac{V}{P} \right\rceil}$$

where

$\rho_1 \triangleq$ *efficiency averaged over all the time when not all P processors are busy*

$$= \frac{1}{H} \sum_{i=1}^{H} \frac{1 + (i-1)\frac{2}{m}}{P}$$

$$= \frac{1}{P}\left[1 - \frac{2}{m} + \frac{H+1}{m}\right]$$

and

$$V = N - 2 \sum_{i=1}^{H} \left[1 + (i-1)\frac{2}{m}\right]$$

$$= N - 2H\left(1 - \frac{2}{m}\right) - \frac{2}{m} H(H+1)$$

Note that when $P = 1$, then $H = 1$ and we expect $\rho = 1$; this can be verified from the above expressions. Furthermore, if $P = B$, we have for even number of levels L,

$$\rho = \rho_1$$

$$= \frac{1}{P}\left[1 - \frac{2}{m} + \frac{H+1}{m}\right]$$

$$= \frac{1}{1 - \frac{2}{m} + \frac{L}{m}}\left[1 - \frac{2}{m} + \frac{\frac{L}{2}+1}{m}\right]$$

$$= \frac{2m + L - 2}{2m + 2L - 4}$$

and for L odd, we have

$$\rho = \frac{2H\rho_1 + \left\lceil \frac{V}{P} \right\rceil}{2H + \left\lceil \frac{V}{P} \right\rceil}$$

where

$$V = (L-1)(1-\frac{1}{m}) + \frac{1}{m}(L-1)\left[\frac{L+1}{2}\right] + 1$$

$$- 2\,\frac{L+1}{2}\left[1-\frac{2}{m}\right] - \frac{2}{m}\,\frac{L+1}{2}\left[\frac{L+1}{2}+1\right]$$

$$= \frac{1}{m}\left[1 - L - m\right]$$

and

$$P = 1 + \frac{2}{m}\left[\frac{L+1}{2} - 1\right]$$

$$= \frac{m+L-1}{m}$$

Thus,

$$\rho = \frac{2H\rho_1 - 1}{L} \quad .$$

$$= \frac{2\,\frac{L+1}{2}\left[\frac{1}{1+\frac{L-1}{m}}\left[1-\frac{2}{m}+\frac{\frac{L+1}{2}+1}{m}\right]\right] - 1}{L}$$

$$= \frac{2m + L - 2 + \frac{1}{L}}{2m + L - 2}$$

For both cases of ρ, we see that

$$\lim_{L\to\infty} \rho \to \frac{1}{2}$$

This can be observed from the fact that the area occupied by the diamond is exactly half the area of a rectangle with L levels long and $P = B$ processors wide. The reason that ρ is not exactly one half for small L is due to the fact that a full task might not be able to exist on the boundary of the diamond.

As an approximation, we assume each level has a continuum of tasks. We now find the number of processors that maximizes power R.

Let i be the index of the levels in G and let n_i be the number of tasks in level i, then

$$n_i = \frac{2i}{m}$$

$$B = \frac{L}{m}$$

$$H = \frac{Pm}{2}$$

We find

$$\rho = \frac{\left[\begin{array}{l} \textit{Sum of the fraction of P processors} \\ \textit{kept busy during the processing time} \end{array}\right]}{\textit{total time required to process this job}}$$

$$= \left[\frac{1}{H + \dfrac{2\left[\dfrac{B\frac{L}{2}}{2} - \dfrac{PH}{2}\right]}{P} + H}\right] \left\{\frac{\sum_{i=1}^{H} n_i}{P} + 2\frac{\dfrac{B\frac{L}{2}}{2} - \dfrac{PH}{2}}{P} + \frac{\sum_{i=1}^{H} n_i}{P}\right\}$$

$$= \frac{\left[\dfrac{2}{mP}\sum_{i=1}^{H} i\right] 2 + \dfrac{B\frac{L}{2} - PH}{P}}{2H + \dfrac{B\frac{L}{2} - PH}{P}}$$

$$= \frac{\dfrac{2H(H+1)}{Pm} + \dfrac{L^2}{2mP} - H}{H + \dfrac{L^2}{2mP}}$$

$$= \frac{1 + \dfrac{L^2}{2mP}}{\dfrac{Pm}{2} + \dfrac{L^2}{2mP}}$$

and

$$S = \frac{Pm}{2} + \frac{L^2}{2mP}$$

Thus,

$$R = \frac{\rho}{S} = \frac{1 + \frac{L^2}{2mP}}{\left[\frac{Pm}{2} + \frac{L^2}{2mP}\right]^2}$$

To find the number of processors that gives the maximum power, we take the partial derivative of R with respect to P,

$$\frac{\partial R}{\partial P} = 0$$

$$= \frac{1}{\left[\frac{Pm}{2} + \frac{L^2}{2mP}\right]^4} \left\{ \left[\frac{Pm}{2} + \frac{L^2}{2mP}\right]^2 \left[\frac{-L^2 2m}{(2mP)^2}\right] \right.$$

$$\left. - \left[1 + \frac{L^2}{2mP}\right] 2 \left[\frac{Pm}{2} + \frac{L^2}{2mP}\right] \left[\frac{m}{2} + \frac{-L^2 2m}{(2mp)^2}\right] \right\}$$

$$= \frac{1}{\left[\frac{Pm}{2} + \frac{L^2}{2mP}\right]^3} \left\{ \left(\frac{P\,m}{2} + \frac{L^2}{2\,P\,m}\right) \left(\frac{-\,L^2}{2\,P\,m}\right) \right.$$

$$\left. - \left(1 + \frac{L^2}{2\,P\,m}\right) \left(m - \frac{L^2}{m\,P^2}\right) \right\} \tag{5.2}$$

and we also need to show $\frac{\partial^2 R}{\partial P^2} < 0$.

$$\frac{\partial^2 R}{\partial P^2} = \frac{1}{\left[\frac{Pm}{2} + \frac{L^2}{2mP}\right]^6}$$

$$\left\{ -\frac{3}{8}\, m^4 P^2 - \frac{6}{32}\, L^2 m^3 P - \frac{1}{4}\, m^2 L^2 - \frac{3}{4}\, \frac{L^4}{P^2} + \frac{12}{32}\, \frac{L^6}{m P^3} \right.$$

$$\left. -\frac{3}{4}\, \frac{L^6}{m^2 P^4} - \frac{5}{8}\, \frac{L^8}{m^5 P^6} - \frac{6}{32}\, \frac{L^{10}}{m^5 P^7} \right\} \tag{5.3}$$

For $1 \le P \le \frac{L}{m}$, Equation (5.3) is less than zero.

Simplifying Equation (5.2), we have

$$\frac{-L^2 m}{8} - \frac{L^4}{4mP^2} - \frac{L^6}{8m^3P^4} - \frac{Pm^2}{2} - \frac{L^2 m}{4} + \frac{L^4}{2P^3m^2} + \frac{L^6}{4m^3P^4} = 0$$

$$-\frac{m^2}{2}P^5 - \frac{3mL^2}{8}P^4 - \frac{L^4}{4m}P^2 + \frac{L^4}{2m^2}P + \frac{L^6}{8m^3} = 0$$

For any value of m and L, we can numerically solve this equation for P and this is the optimal number of processors required per job to achieve the maximum power.

If we are simply given an S versus ρ curve, we can find the optimal number of processors. First, we solve for P from the expression for ρ for a specific value of ρ',

$$\frac{Pm}{2}\rho' + \frac{L^2}{2mP}\rho' - 1 - \frac{L^2}{2mP} = 0$$

$$\frac{m\rho'}{2}P^2 - P + \frac{L^2}{2m}(\rho' - 1) = 0$$

$$P = \frac{1 + \sqrt{1 - 4\left(\frac{m\rho'}{2}\right)\frac{L^2}{2m}(\rho' - 1)}}{m\rho'}$$

$$= \frac{1 + \sqrt{1 + L^2\rho'(1 - \rho')}}{m\rho'}$$

We now substitute this P back into the expression for S,

$$S = \frac{m}{2}\left[\frac{1+\sqrt{1+L^2\rho'(1-\rho')}}{m\rho'}\right] + \frac{L^2}{2m\left[\frac{1+\sqrt{1+L^2\rho'(1-\rho')}}{m\rho'}\right]}$$

$$= \frac{1+\sqrt{1+L^2\rho'(1-\rho')}}{2\rho'} + \frac{L^2}{\left(\frac{2}{\rho'}\right)\left(1+\sqrt{1+L^2\rho'(1-\rho')}\right)}$$

Interestingly, S is not a function of m when optimal number of processors are used. For a specific utilization ρ' we obtain the same S regardless of the value of the slope m.

As we know [KLEI79] the maximum power is achieved at the point where the S versus ρ curve intersects a straight line from the origin approaching from the right (see Figure 5.8).

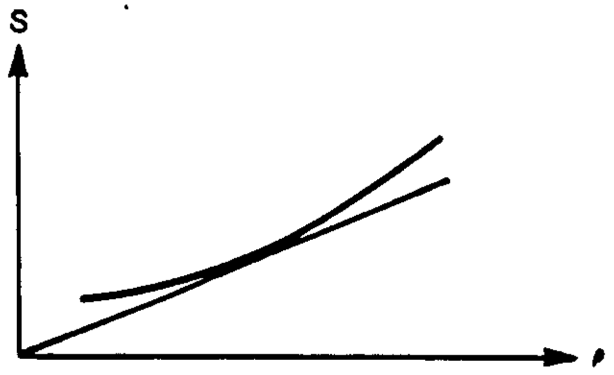


Figure 5.8 S versus ρ

From this method we can find the ρ^* at the intersection and substitute back into the expression for P to obtain the optimal number of processors

$$P^* = \frac{1+\sqrt{1+L^2\rho^*(1-\rho^*)}}{m\rho^*}$$

The S versus ρ curve does not start from $\rho = 0$ because the minimum ρ occurs when $P = B$ or

$$Min\ \rho = \frac{1 + \frac{L^2}{2mB}}{m\frac{B}{2} + \frac{L^2}{2}mB}$$

$$= \frac{1 + \frac{L}{2}}{\frac{L}{2} + \frac{L}{2}}$$

$$= \frac{1}{L} + \frac{1}{2}$$

In addition, because no jobs can be finished in less than L normalized units of time, we have $S \geq L$.

Figures 5.9-5.13 show some examples of the average system time versus utilization curves for several values of L. Table 5.2 compares the optimal value of P^* solved from the equation with the value calculated by using the ρ^* obtained from the Figures 5.9-5.13.

L	ρ^*	P^*	P^* (exact solution with $m = 1$)
10	0.82	$6.06\ \frac{1}{m}$	6.284
20	0.78	$11.98\ \frac{1}{m}$	12.047
30	0.76	$18.23\ \frac{1}{m}$	17.824
40	0.76	$23.83\ \frac{1}{m}$	23.594
50	0.75	$30.232\ \frac{1}{m}$	29.369

Table 5.2

The P^*'s obtained from Figures 5.9-5.13 theoretically should be the same as P^*'s calculated from the equation and they are very close to each other. But more interestingly is the fact that P^* is *approximately* equal to $0.6\frac{L}{m}$. In other words, we should provide a number of processors for each job equal to six tenths of the number of levels L divided by the slope of the process graph.

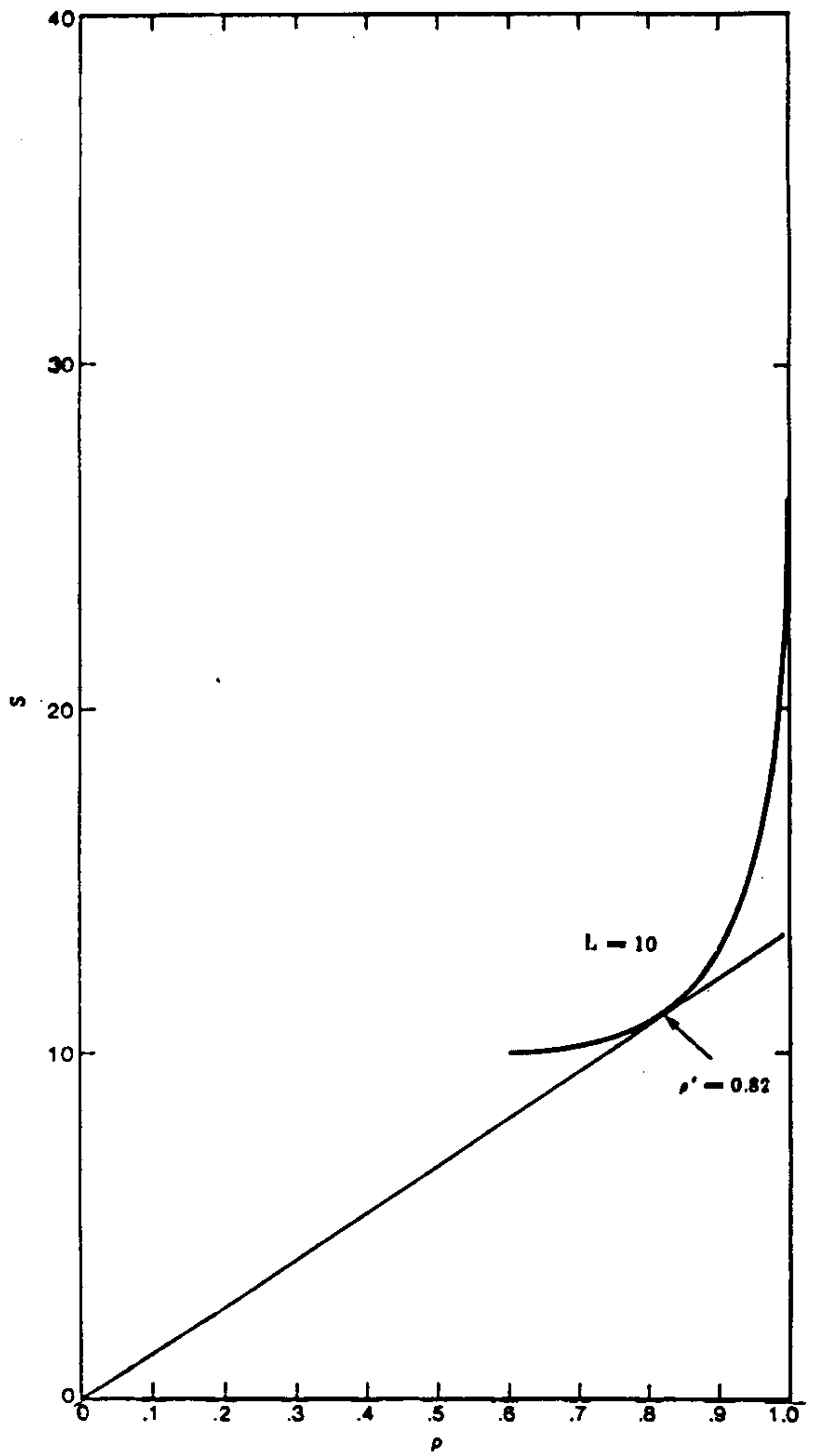


Figure 5.9 S versus ρ with $L = 10$

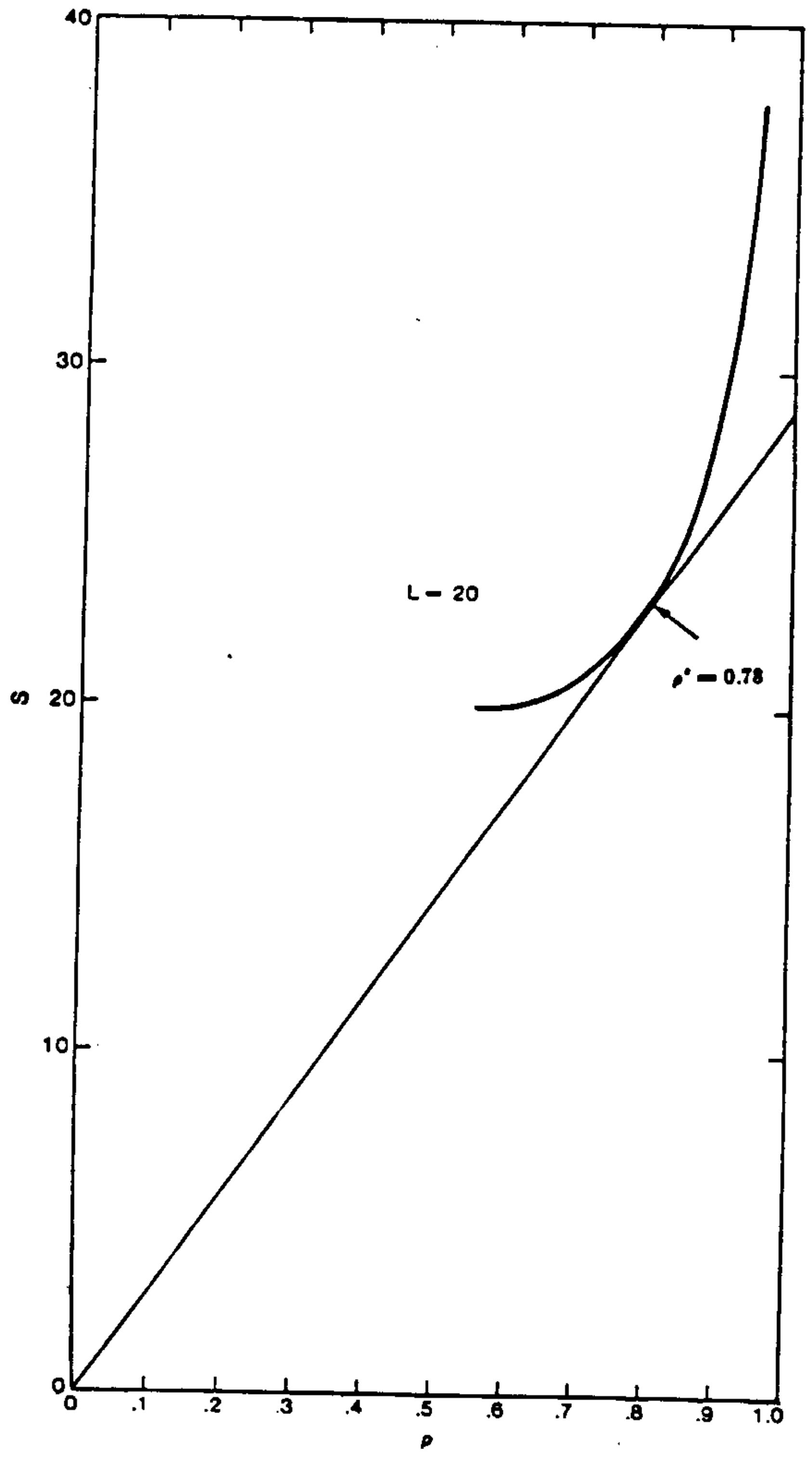


Figure 5.10 S versus ρ with $L = 20$

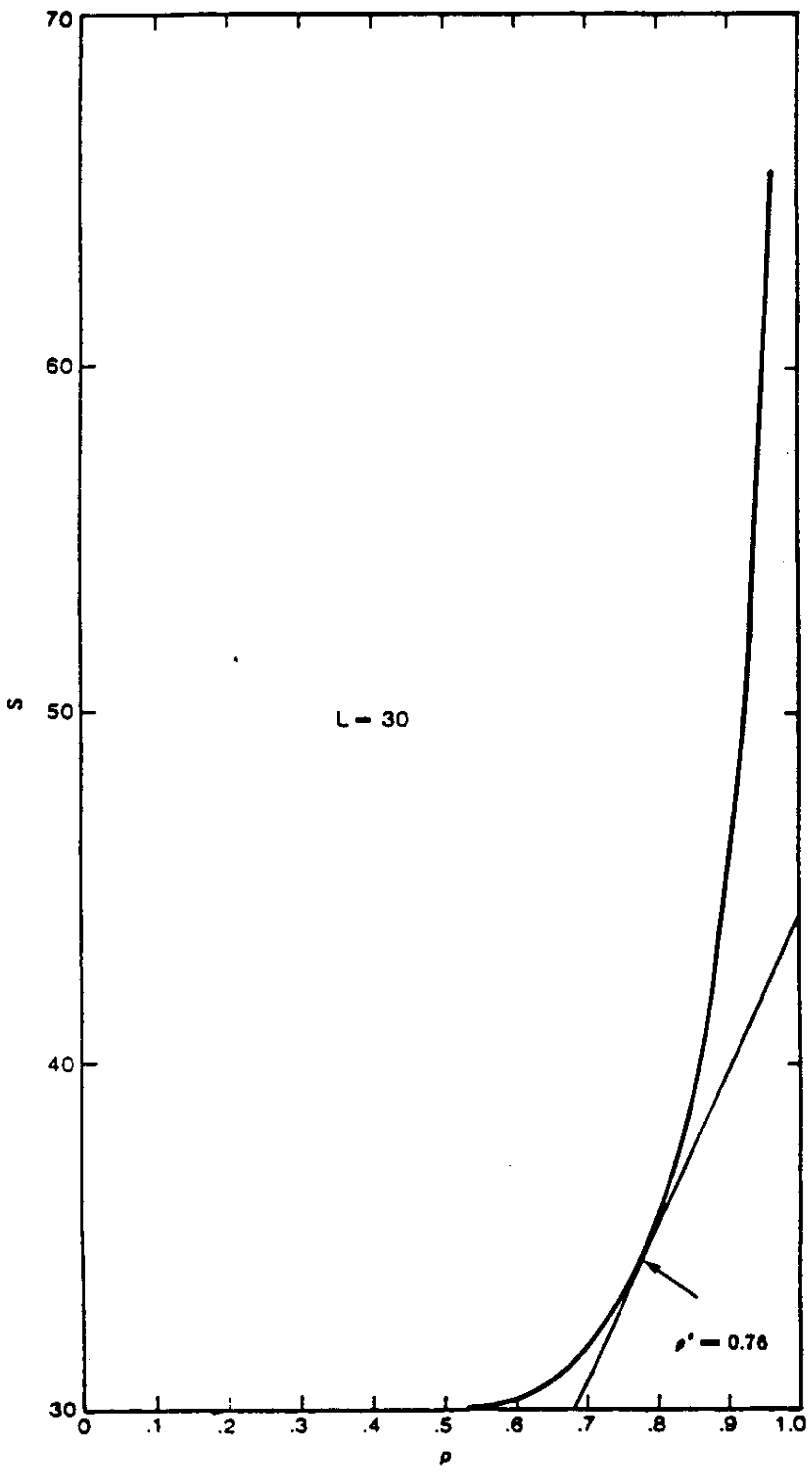


Figure 5.11 S versus ρ with $L = 30$

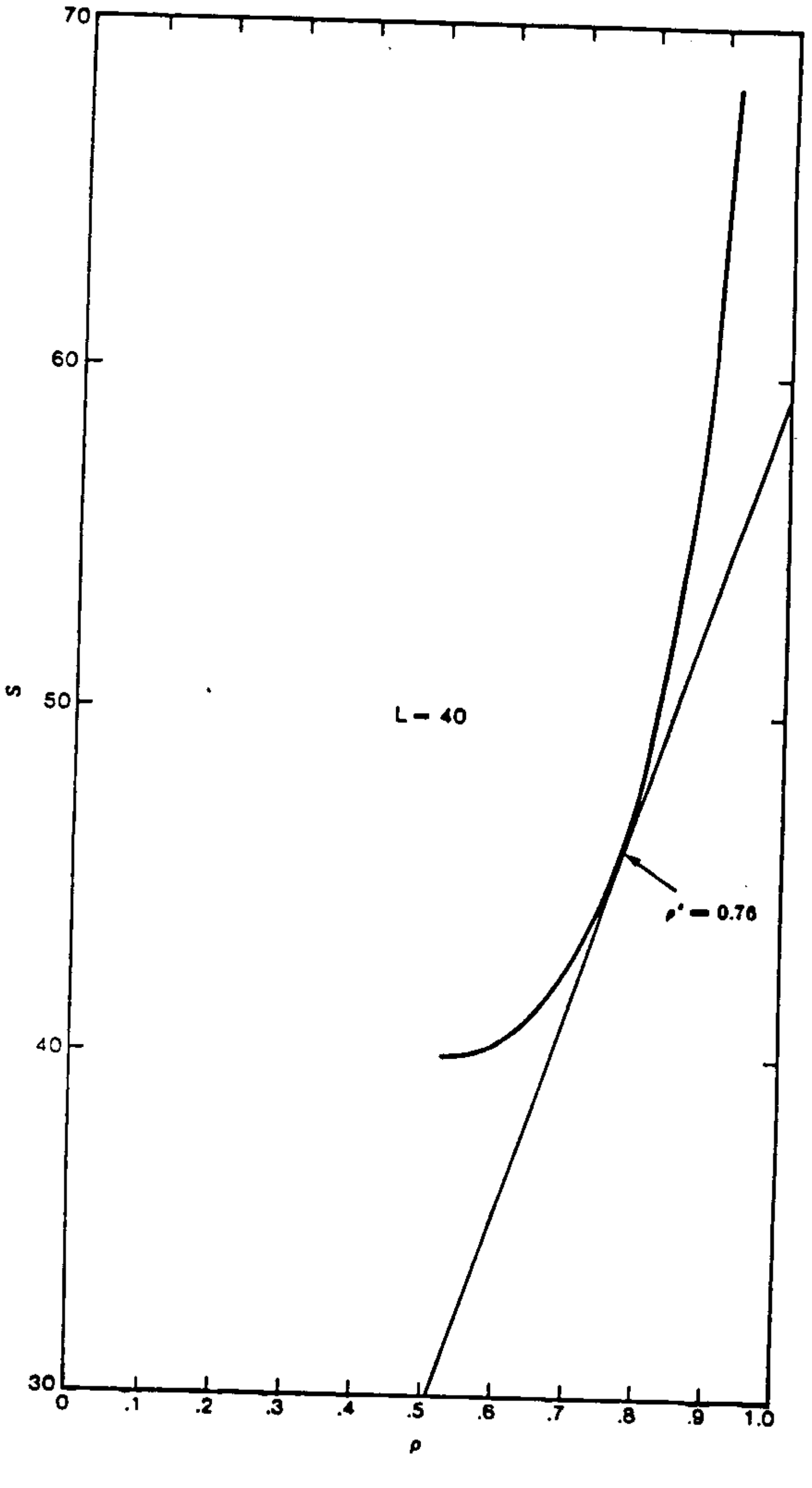


Figure 5.12 S versus ρ with $L = 40$

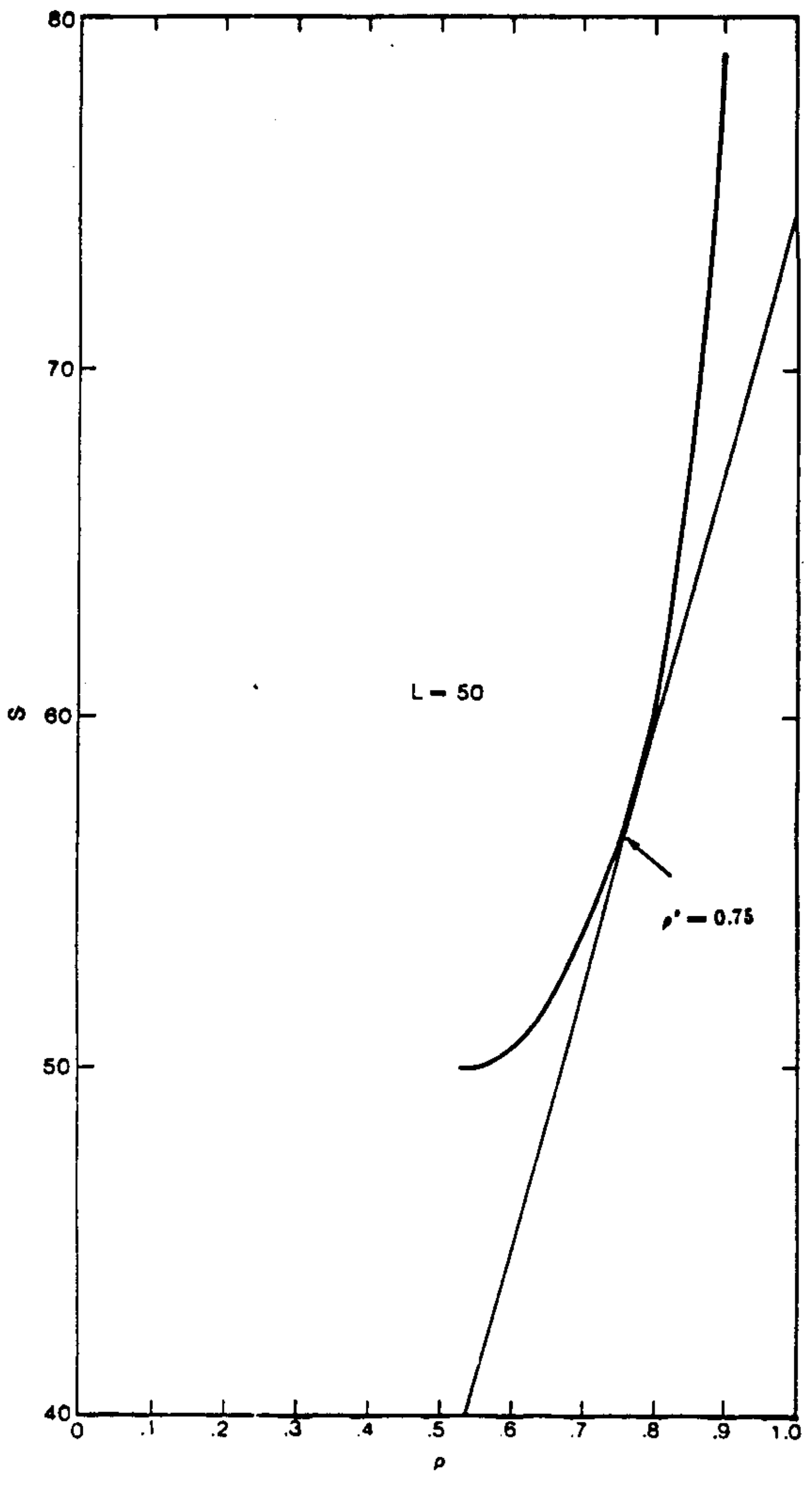


Figure 5.13 S versus ρ with $L = 50$

Substituting the value of P^* into the expression for the average system time,

$$S^* = \frac{P^* m}{2} + \frac{L^2}{2\, m\, P^*},$$

we have an approximate average system time

$$S^* = \frac{0.6\, L}{2} + \frac{L^2}{2\,(0.6\,)\, L}$$

$$= 1.133\, L$$

Note that if we do not care about the low utilization of processors, then, by using maximum number of processors, the average system time is $S = L$. The slightly larger S^* is the trade off between the utilization of processors and the average system time of jobs. Furthermore, we note that the approximate concurrency measure is

$$\sigma = \frac{S\,(P^*)}{S\,(1\,)}$$

$$= \frac{1.133\, L}{\frac{L^2}{2\, m}}$$

$$= \frac{2.266\, m}{L}$$

5.6 Bounds on the Average System Time with a Limited Number of Processors (k, G^*, x, $P < \infty$)

In this section, we limit the number of processors to a constant P. With a finite number of processors, we have the problem of scheduling tasks. When more than one task demands a processor and only one processor is available, we are forced to pick one task to be processed. The method of selecting which task to be processed next in general effects the average system time of the job.

Assuming unit task processing times, a lower bound on the average system time can be easily calculated as

$$S_{LB} = \max\left[L, \frac{Nk}{P}\right]$$

where L is the number of levels, N is the number of tasks per process graph, k is the number of jobs, and P is the number of processors.

A very loose upper bound can be obtained by using the Longest Expected Processing Time First assignment algorithm [COFF76]. Assume $P < k$, and that we assign only one processor to a job. Whenever a job is completed, the processor looks for another unstarted job to process. Idled processors are not allowed to assist other jobs. Since only one processor works on a single job, the structure of the random process graph does not influence the execution time. Each job will take the same amount of processing time of $N\,x$ units.

Let $s = \left\lfloor \frac{k}{P} \right\rfloor$ and $t = \textit{remainder of } \frac{k}{P}$, then an upper bound is

$$S_{UB} = \frac{1}{k}\Big[NP + (2N)P + \cdots + (sN)P + (s+1)Nt\Big]$$

If we allow random tasks time, with the distribution F(t) and mean of $\overline{X}$, and if we require synchronization of P jobs (i.e. all P jobs must all finish before starting another P jobs), then we have the Longest Expected Remaining Processing Time First assignment algorithm [COFF76] which gives

$$S_{UB} = \Big[M_1 P + (M_1 + M_2)\,P + \cdots$$

$$+ (M_1 + M_2 + \cdots + M_s)\,P$$

$$+ (M_1 + M_2 + \cdots + M_s + M_t)\,P\Big]\ \frac{1}{k}$$

where

$$M_\alpha = \max_{1 \le i \le P} \left\{ \sum_{j=1}^{N} X_{ij} \right\} \qquad \alpha = 1, 2, \cdots, s$$

and *all* M_α *have the same distribution*

$$M_t = \max_{1 \le i \le t} \left\{ \sum_{j=1}^{N} X_{ij} \right\}$$

X_{ij} = *random task time* for *task j of job i*

From the Law of Large Numbers,

$$\lim_{N\to\infty} Prob\left[\left|\frac{\sum_{j=1}^{N} X_{ij}}{N} - \overline{X}\right| < \epsilon\right] = 1 \qquad \epsilon>0$$

Hence,

$$\lim_{N\to\infty} Prob\left[S_{UB} = \frac{N\overline{X}P + 2N\overline{X}P + \cdots + N\overline{X}P + (z+1)N\overline{X}y}{k}\right] = 1$$

or

$$\lim_{N\to\infty} Prob\left[S_{UB} = \frac{\sum_{i=1}^{z} iN\overline{X}P + (z+1)N\overline{X}y}{k}\right]$$

$$= \lim_{N\to\infty} Prob\left\{S_{UB} = \frac{\left[\frac{z(z+1)}{2}P + (z+1)y\right]N\overline{X}}{k}\right\} = 1$$

The bounds obtained in this section are very loose. The minimum average system time is known to be achieved by the Shortest Expected Remaining Processing Time First (SERPT) [COFF76] scheduling algorithm. But with random process graphs, we don't know the exact structure of each process graph in order to apply the SERPT.

5.7 Discussion

In this chapter, we have attempted to observe some properties of the arrangements of random process graphs. We found a method to construct all the arrangements of the tasks for a process graph with N tasks. The distribution of the number of arrangements with respect to the number of levels was shown to be Gaussian. The tail probability of this distribution was bounded by the Chernoff bound. Next, an upper bound and a lower bound for the average system time were obtained for a specific number of levels in a process graph. As N becomes large, the probability that the average system time of a randomly chosen process graph is between the upper bound and the lower bound calculated near the mean number of levels approaches one.

The number of precedence relationships and the number of levels were added to the model next. The bounds for this case were found and compared to the previous bounds.

We used the notion of power to study the trade off between the utilization of the processors and the average system time. A very loose upper bound was presented for the case where the number of processors is finite.

CHAPTER 6
Process-communication Graphs

In the previous two chapters, we have assumed that the cost of sending data between processors is free and that the communication between them can be achieved instantaneously. In reality, there is always some delay occurred when communicating between the processors.

Gentleman [GENT78] found that in a multiprocessor environment, even though data paths are provided to move data between processors, data from one processor is only immediately available to a small number of other processors, and in general, moving data from one processor to another requires several submoves. Gentleman uses the matrix multiplication on ILLIAC IV, where processors are connected in a two dimensional rectangular grid, as an example. For the multiplication of two N by N matrices, at least $\left(\frac{N^2}{2}-\frac{1}{4}\right) - \frac{1}{2}$ data movements are required. Hence, we cannot ignore the communication cost in general.

To minimize the communication cost, we will try to assign as many tasks as possible on a processor. Of course, there will be no communication cost if all the tasks are assigned to a single processor. On the other hand, we are looking for maximum concurrency which will tend to use as many processors as possible to execute the tasks. A compromise must then be made to balance between these two opposing objectives. Consequently, we can no longer assume that there are an infinite number of processors. With a limited number of processors, the need for task assignment comes back.

In this chapter, we still use the process graph discussed in Chapter 3. Except, we will add in the communication tasks that represent the communications required between the processors. We will look at how the number of processors will affect the average system time and how we can obtain the average system time by converting the process-communication graph into a Markov Chain (similar to Algorithm CPM in Chapter 4, but with some differences due to the limited number of processors). We do not address the task assignment problem specifically. Instead a simple rule of thumb is used in deciding which processor a task should reside in and where the communication tasks are added in the process graph.

6.1 Communication Tasks (k, G, x', $P < \infty$)

We have assumed in Chapters 3, 4 and 5 that the tasks in a process graph may be assigned to any processors and that there is no communication cost of passing data between tasks due to the contention on the communication bus or the physical distance between any pair of processors. In this section, we explore a way to represent this communication cost in the model discussed in Chapter 4.

We assume that the process graph is fixed and that the task service time is exponentially distributed with a mean of $\frac{1}{\mu}$ sec. If each task is residing on a different processor, then there exists a communication delay between any two tasks if there is a precedence relationship between them. We will treat this communication cost as another task whose service time is exponentially distributed (with a mean of $\frac{1}{\mu_c}$ sec.). For example, Figure 6.1b is the process graph obtained when we add communication tasks to the process graph of Figure 6.1a.

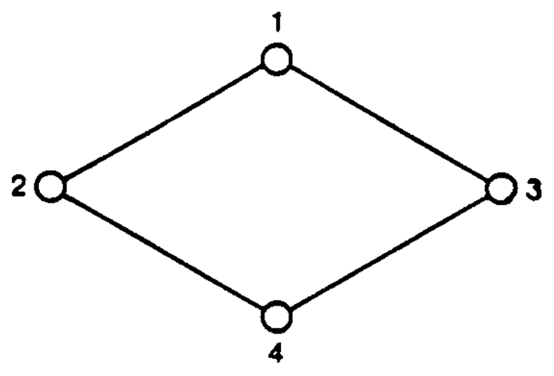


Figure 6.1a Process Graph

A communication task S_{ij} represents the communication between a processor where task i is residing and a processor where task j is residing. We will call the process graph with the communication tasks added the "process-communication graph".

Of course, if a task t has several subtasks $t_1, t_2, \cdots, t_\alpha$, where $\alpha \geq 2$, we can assign one of the subtasks on the same processor where task t was executing. We assume for now that the subtask selected to reside on the same processor with task t is picked at random, and when two tasks reside on the same processor, the communication cost between these two tasks is zero.

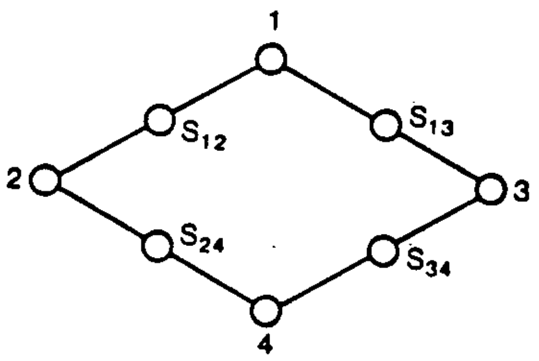


Figure 6.1b Process-communication Graph

In the example showd in Figure 6.1b, we can let tasks 1 and 2 reside on one processor and let tasks 3 and 4 reside on another processor. The resulting process-communication graph is shown in Figure 6.2.

Since the communication task is treated just like a regular task, we see that there could be concurrent execution of regular tasks and a communication task. If we assume multiple communication busses, then more then one communication task could execute in parallel also. In Section 6.3, we will examine the case when only one communication task is allowed to execute at any given time.

The exact communication time requirement depends upon the access protocols, the communication bandwidth, the volume of data to be transmitted and the physical locations of the processors requiring the communication. Lee [LEE77], discussed in Section 2.3.6, approximated the communication cost by multiplying the volume of data to be transmitted by the distance of the two processors. His method assumed that the volume of data transmitted between two processors is fixed and the distance between two processors is a constant multiplied by the number of hops or a linear function of the physical distance.

In this chapter, we let $\mu_c = a\mu$ where a is a real number greater than zero. If $a > 1$, the communication task takes less time on the average than the regular task. In particular, when $a = \infty$, communication cost is then zero.

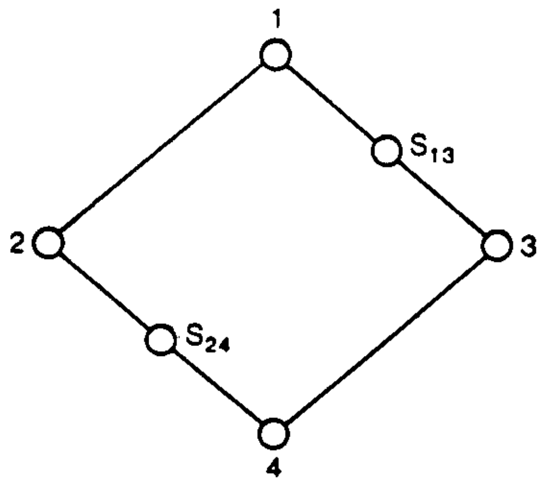


Figure 6.2 Process-communication Graph

Using Algorithm CPM (Section 4.2.2), we can convert a process-communication graph into a Markov Chain to solve for the average system time. Figure 6.3 is the Markov chain for the process-communication graph of Figure 6.2. Recall that C_α is a state where all the tasks in α are executing concurrently.

We wish to solve for the a such that the resulting average system time equals to $S(P=1)$. We denote such a as a_s. Note that with $a = a_s$, the concurrency measure σ equals one. This will allow us to separate out those systems which yield a net improvement when parallel processing is introduced.

Using the balance equations and the fact that the sum of all equilibrium state probabilities equals to one, we obtain the expression for a_s for the example shown in Figure 6.3 as

$$\sigma = 1 = \frac{S(a_s)}{S(P=1)} = \frac{\frac{3+9a_s+7a_s^2}{2a_s+2a_s^2}\frac{1}{\mu}}{4\frac{1}{\mu}}$$

Solving for a_s, we get $a_s = 2.3027$. In other words, if $\mu_c > 2.3027\mu$, for the job represented by Figure 6.1a, multiprocessing is still faster than the single processor (even with the communication delay). But if $\mu_c \leq 2.3027\mu$, then it is better to process this job on a single processor.

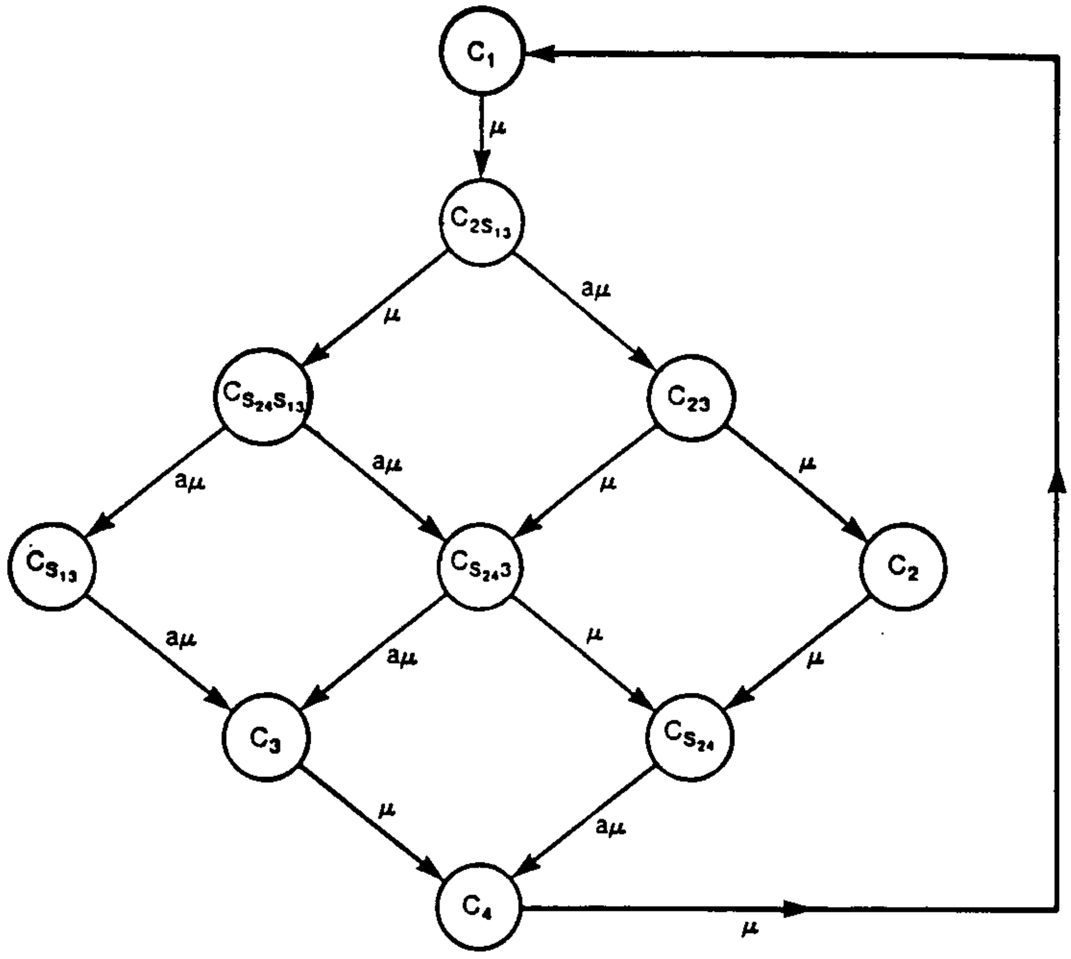


Figure 6.3 Markov Chain of Figure 6.2

6.2 Limited Number of Processors Per Job (k, G, x^*, $P < \infty$)

If there are limited number of processors and the number of processors, P, assigned to a job G is smaller than the width of G, then the scheduling of tasks to processors is required. Whenever there is more than one task waiting for an available processor, we must consider the communication costs when deciding which task should be processed on this processor. If the available processor does not have the data necessary for executing a task, the data must be transmitted from another processor. Thus, we have a difficult optimization problem of assigning tasks to the processors. In this section, we discuss some simple rules of thumb in assigning tasks. These rules most likely will not produce the best assignment in terms of minimizing the average system time, but it provides a basis for analyzing the effect of having different number of processors for a specific process graph.

The rules of thumb for assigning tasks to processors are

1. For a task i, if there is only one task j such that there is a precedence relationship (i,j), then assign task j to the same processor where task i is processed.

2. For a task i, if there is a set of tasks α such that there is a precedence relationship (i,j) where $j \epsilon \alpha$, and there is only one available processor, assign all the tasks α to the same processor where task i is processed.

3. For a task i, if there is a set of tasks α such that there is a precedence relationship (i,j) where $j \epsilon \alpha$, and there is more than one processor available, supposing there are P' processors and x tasks in the set α, then

   i. for $x \leq P'$
   assign one task $j \epsilon \alpha$ to the processor where task i is processed, create one communication task for each of the x-1 tasks and assign each of them to a separate processor.

   ii. for $x > P'$
   assign $(x-P' +1)$ tasks to the processor where task i is processed, create $(P' -1)$ communication tasks and assign the rest of the tasks to each of the $(P' -1)$ processors.

4. For a task i, if it has only one edge (i, j) leaving it and task j has been assigned to another processor p, then create a communication task to transfer the data to processor p.

Figures 6.4-6.6 show a simple example of the task assignment.

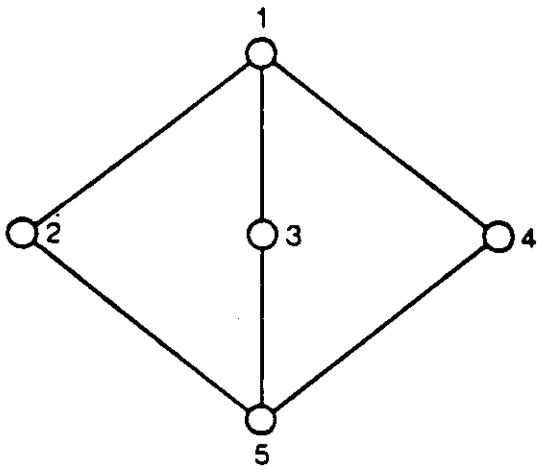


Figure 6.4 Process Graph

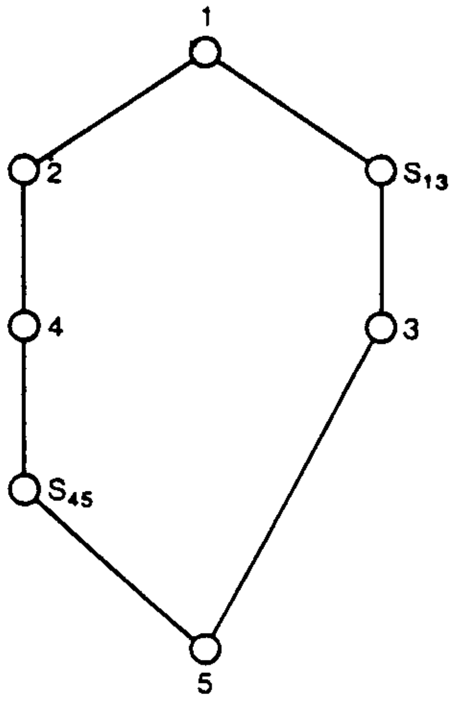


Figure 6.5 Process-communication Graph with $P = 2$

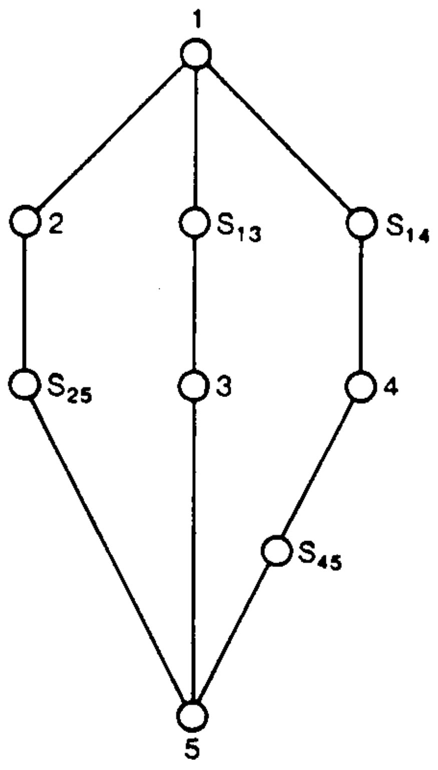


Figure 6.6 Process-communication Graph with $P = 3$

Figure 6.4 is the original process graph. Figure 6.5 shows the assignment for the $P = 2$ case, and Figure 6.6 shows the assignment for the $P = 3$ case. In Figure 6.5, tasks 1, 2 and 4 reside on one processor while tasks 3 and 5 reside on the other processor. In Figure 6.6, tasks 1 and 2 reside on the first processor; tasks 3 and 5 reside on the second processor; task 4 resides on the third processor. If we draw circles around the task assignments in the original process graph of Figure 6.4, any precedence relationship that connects tasks across these circles indicates that a communication task is required.

When the communication cost is assumed to be zero ($a = \infty$), we expect the μS versus P figure to look like Figure 6.7. When $P = 1$, $\mu S = N$ where N is the number of tasks in the process graph, and as P increases, the value of μS will approach N^* where $\frac{N^*}{\mu}$ is the average system time $S(P = \infty)$ obtained in Chapter 4.

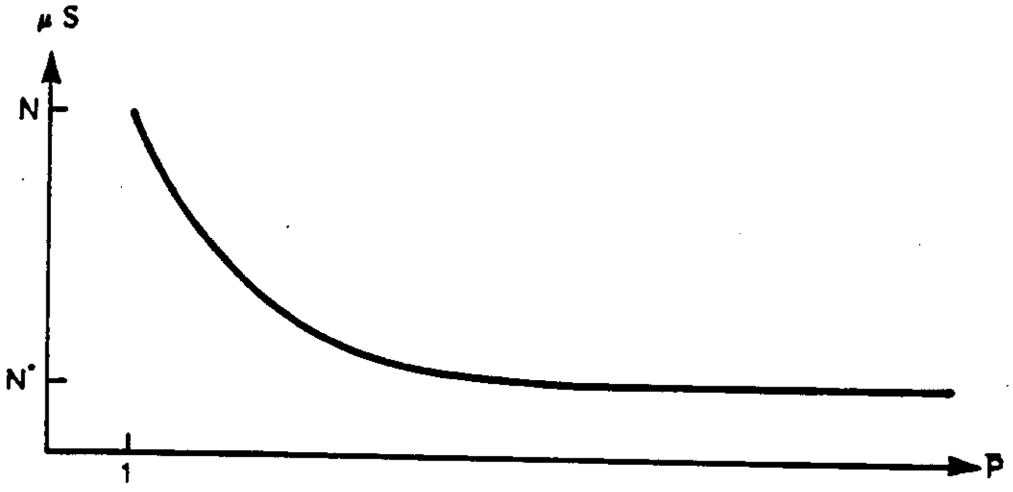


Figure 6.7 μS versus P

In general as the cost of the communication increases (smaller a), the average system time will increase also. Figure 6.8 is a typical example of a family of curves for μS versus P. For a specific value of P, let us say P', there will be a specific $a = a_s$ such that the value of μS at these values of a_s and P' equals to N. In other words, the advantage of multiprocessing on P' processors is erased by the communication cost (with mean time of $\frac{1}{a_s \mu}$) between processors. As a becomes smaller than a specific a^*, the communication cost becomes too large for any multiprocessing at all, and the normalized average system time is greater than N.

Figures 6.9-15 show an example of the process graph (Figure 6.9), the process-communication graphs with various values of P, $P = 2,3,4,5,6$, (Figures 6.10-14), and the resulting family of μS curves versus P (Figure 6.15). Of course these process-communication graphs are for a specific assignment of tasks to processors. But the behavior of the curves is likely to be similar for all other assignments.

For some of the curves, a horizontal line intersects the μS curve at two points as in Figure 6.16. This implies that the average system time is the same whether we use P_1 processors or P_2 processors. We are interested, then, in the issue of efficiency of the processors. Since $P_2 > P_1$, and in both cases the same amount of work is completed in the same average system time, the P_2 processors must be idling more (waiting for the communication tasks to complete).

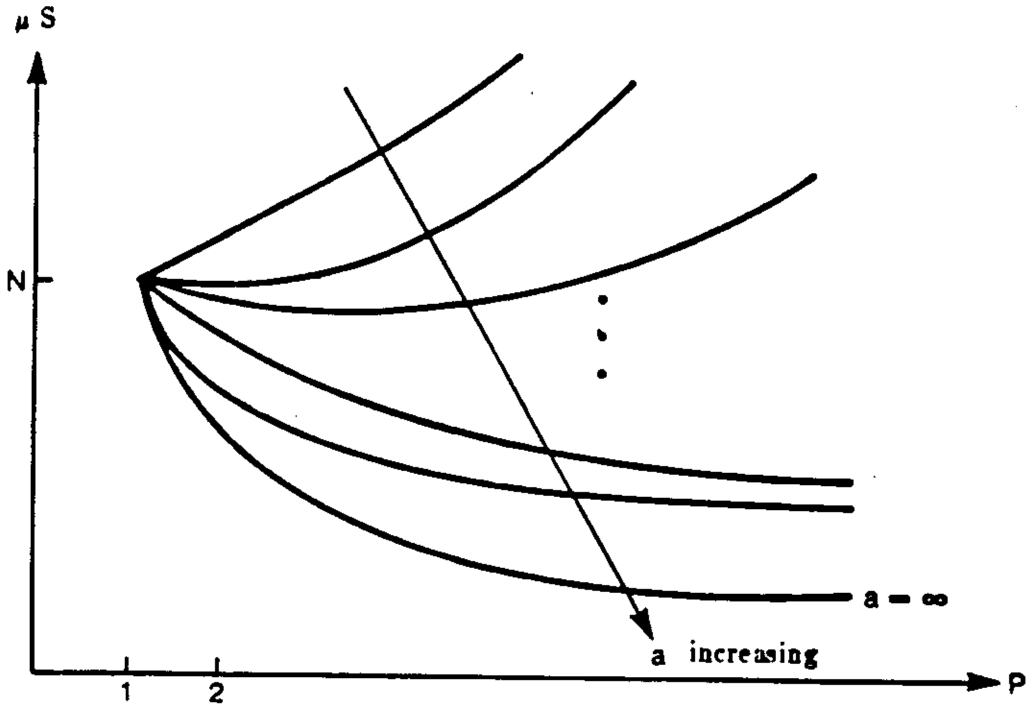


Figure 6.8 μS versus P with a Family of a

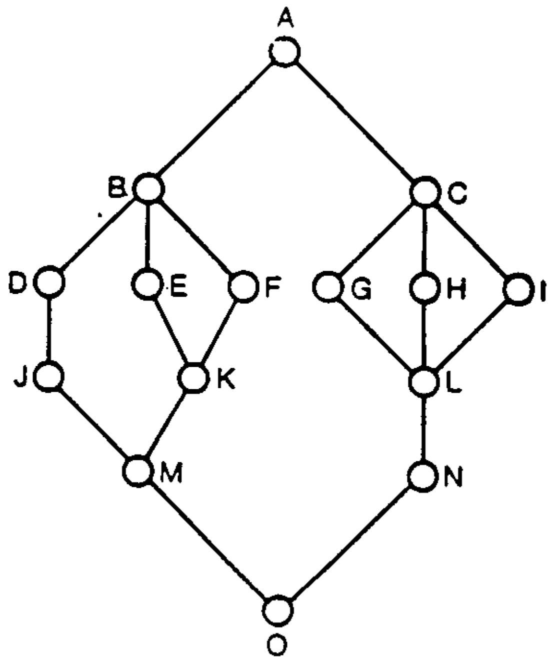


Figure 6.9 A Process Graph

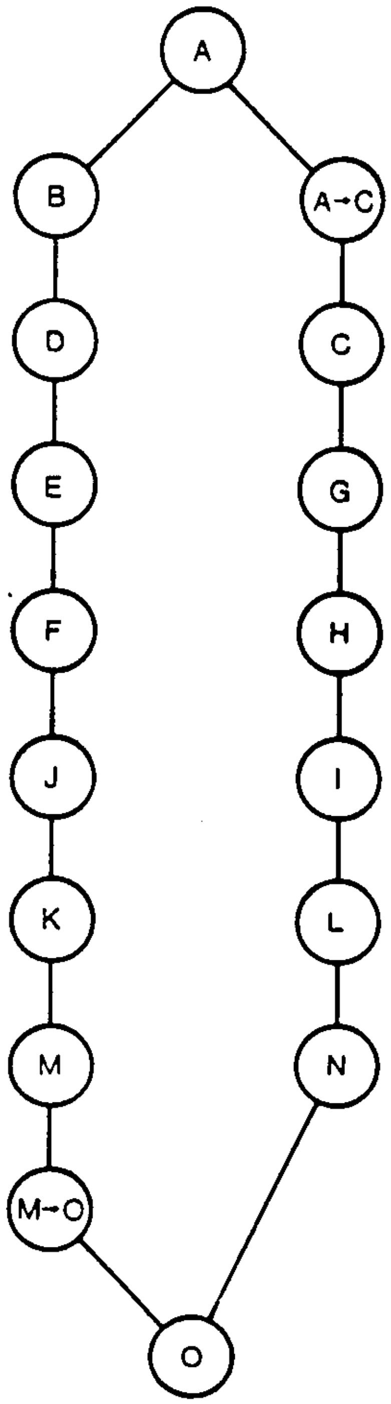


Figure 6.10 Process-communication Graph with $P = 2$

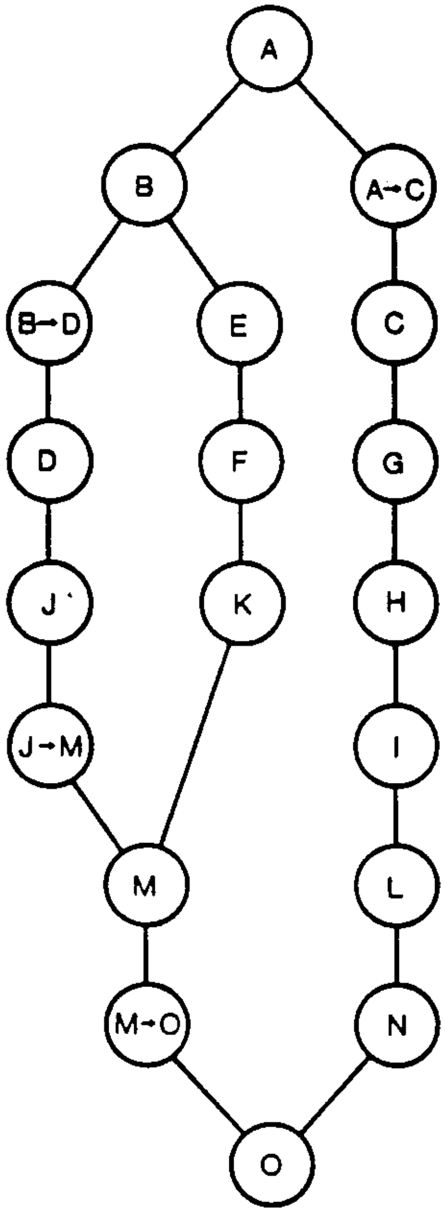


Figure 6.11 Process-communication Graph with $P = 3$

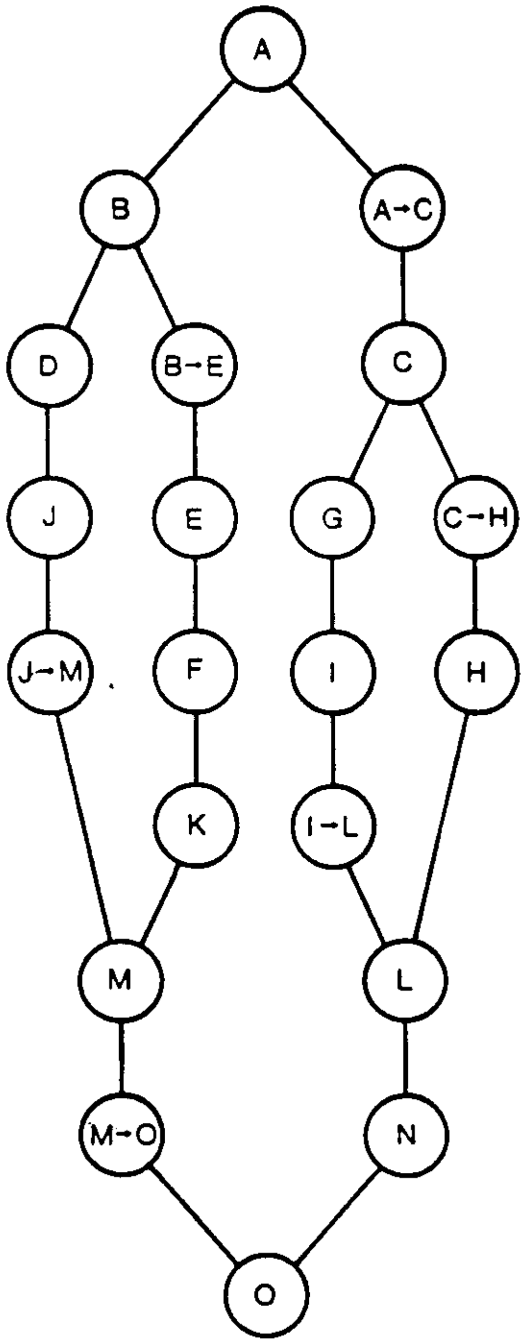


Figure 6.12 Process-communication Graph with $P = 4$

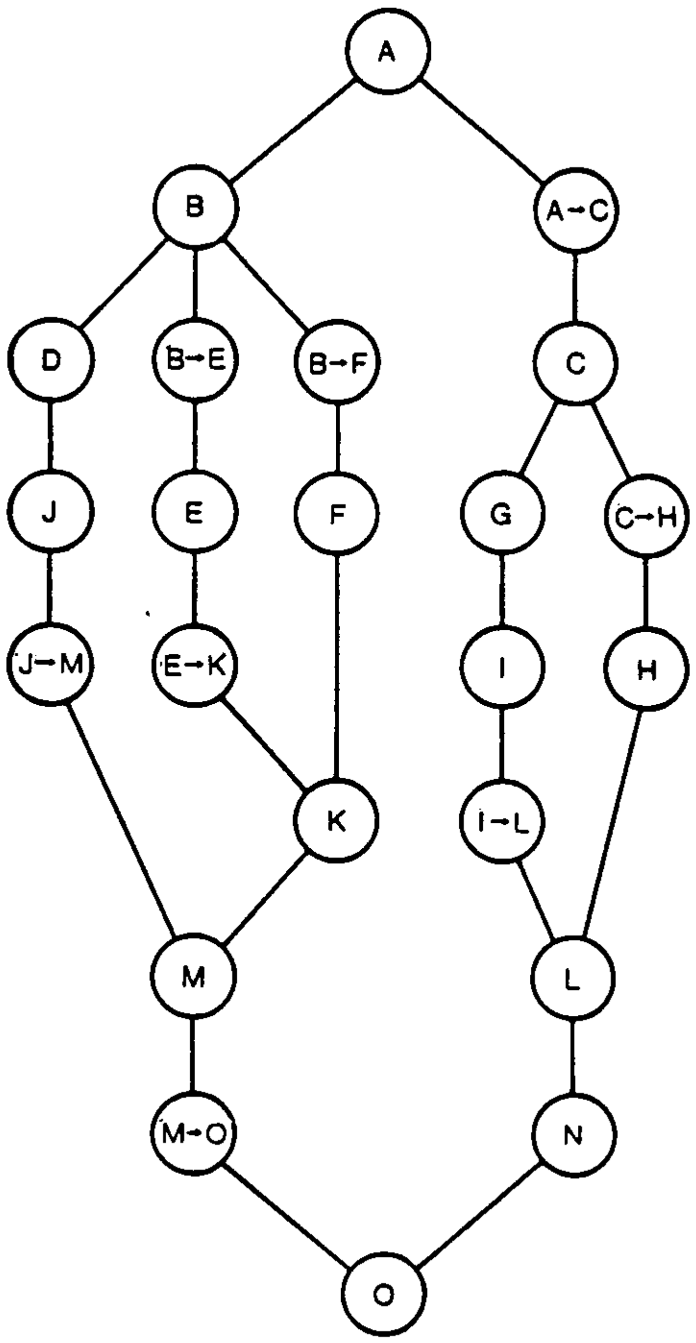


Figure 6.13 Process-communication Graph with $P = 5$

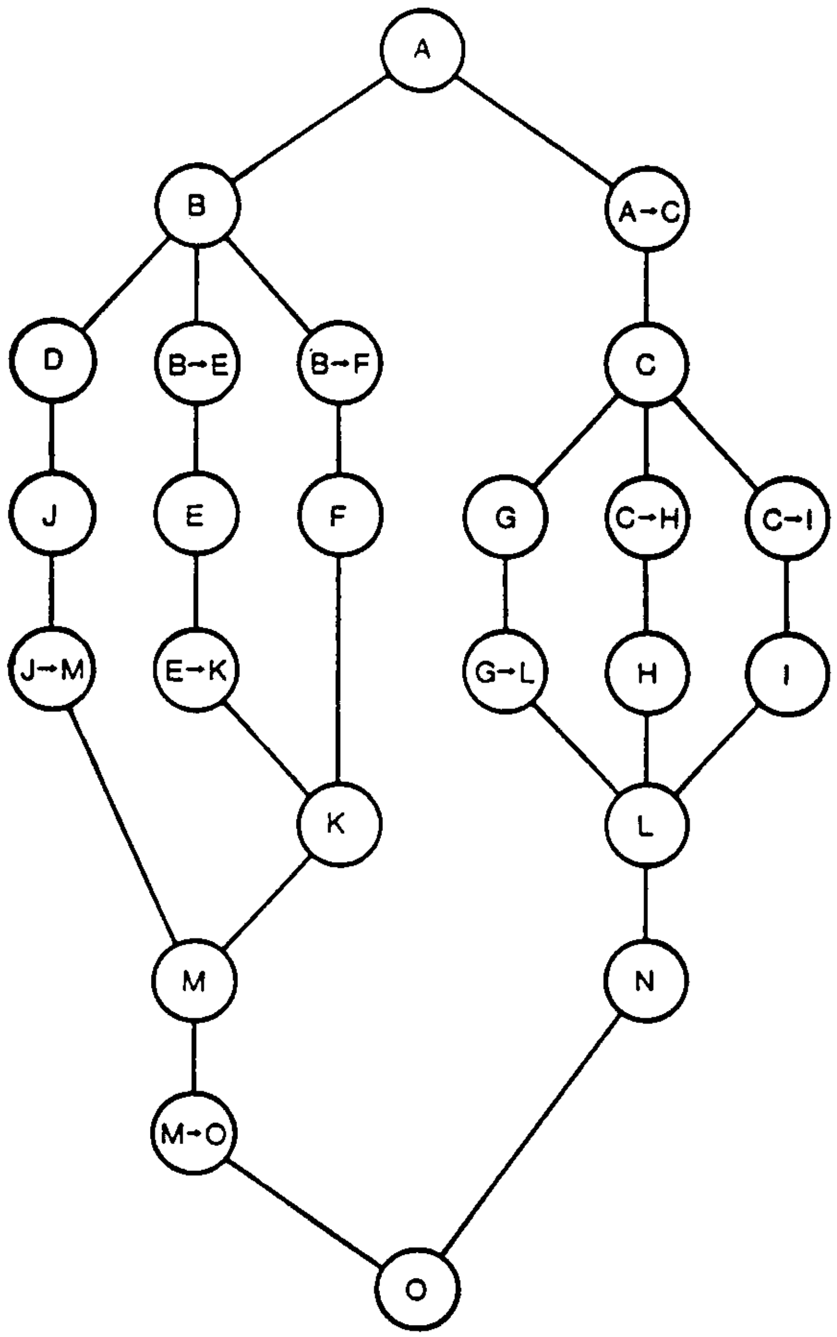


Figure 6.14 Process-communication Graph with $P = 6$

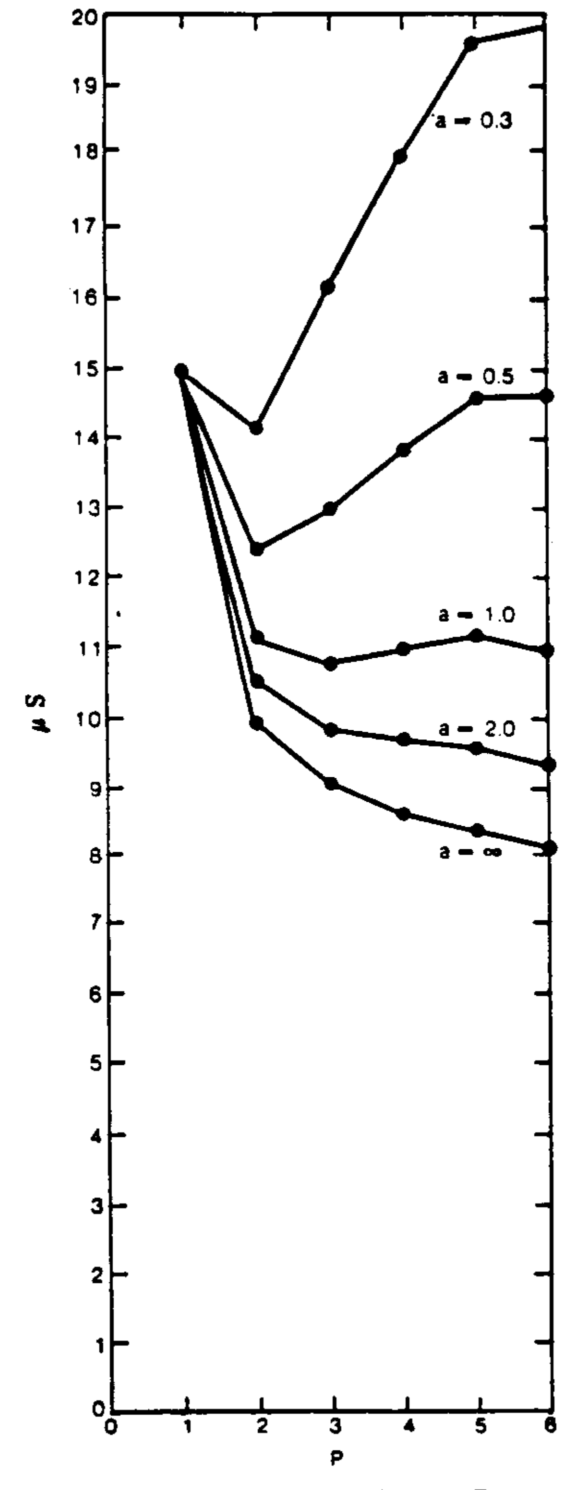


Figure 6.15 μS versus P

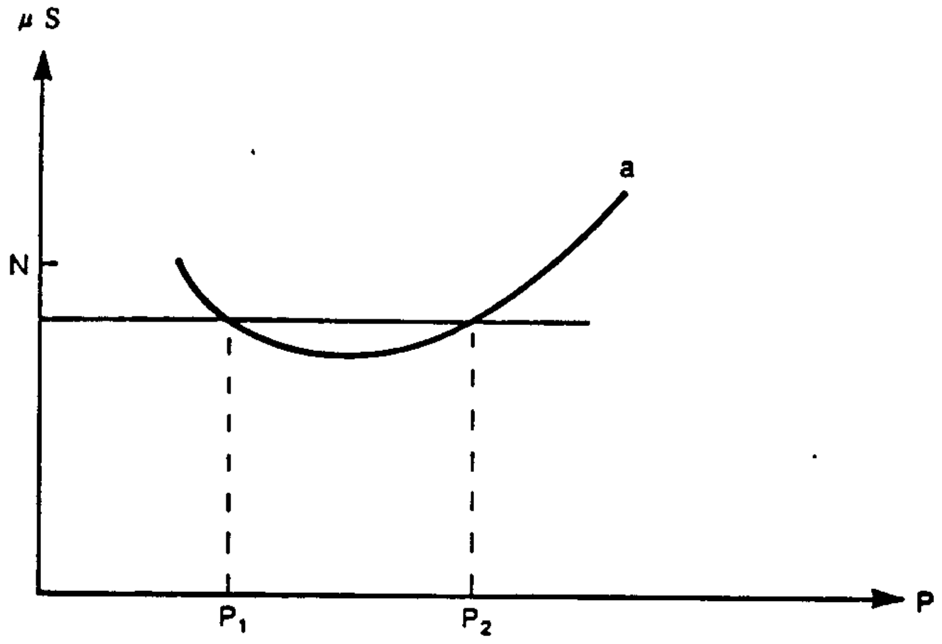


Figure 6.16 μS versus P with a specific value of μS^*

Let us denote the average system time when $P = 1$ to be S_1 and the average system time with P processors and with $\mu_c = a\mu$ to be $S_P(a)$. The total work to be done is constant and equals $N \frac{1}{\mu}$ second which is also S_1. Let η denotes the efficiency of the processors where η is defined to be $\frac{S_1}{P\, S_P(a)}$ which is the total useful work done divided by the time necessary for the P processors to process this amount of useful work multiplied by P. If we define $\frac{S_1}{S_P(a)} = C(a)$, where $C(a)$ is a constant for a specific a, then

$$\eta = \frac{C(a)}{P}$$

Therefore, we can see the trend that the efficiency of the multiprocessors decreases as the number of processors increases.

6.3 One Communication Bus (k, G, x^*, $P < \infty$)

In the last two sections, we have assumed that there is more than one communication bus such that whenever a processor needs to send data to another processor, it will be transmitted without having to wait for a communication channel. In this section, we assume that there is only one communication bus such that only one processor may be transmitting at any given time. We also assume that communication tasks are perfectly scheduled such that if there is more than one communication task required to be transmitted, then each of them will be transmitted in turn and each of them knows who is to be transmitted next (no collisions).

To obtain the average system time, we again convert the process-communication graph into a Markov chain. The states of the Markov chain are represented as $C_\alpha\, D_\beta$ where the set α contains tasks to be executed in parallel, and the set β contains all the communication tasks waiting to be transmitted. If the set β is not empty, then one of the tasks in α must be a communication task. After the communication task in α finishes, we can activate one of the tasks in β. The conversion algorithm is same as Algorithm CPM of Section 4.2.1, with the modification that each time a task i completes the execution, we add the following two tasks:

1. as the result of the completion of task i, if there is a set of communication tasks becoming active, concatenate them into the set β

2. if task i was a communication task, and if the set β is not empty, bring one of the task in set β into the set α.

After the Markov chain is constructed, we can solve for the equilibrium state probabilities and the average system time using the same method studied in Chapter 4. We expect that the average system time with one communication bus is greater than the multi-communication busses because of the delay caused by the non parallel processing of the communication tasks.

For example, Figure 6.18 is one of the Markov chains that can be converted from the process communication graph of Figure 6.17 where tasks 3, 4, 5, and 8 are the communication tasks. After the task 2 in the state C_{23} D_4 completes, the communication task 5 is concatenated to the set β, communication task 4 cannot be moved to the set α because task 3 is a communication task and there is only one communication bus available; after the communication task 3 in the state C_{23} D_4 completes, one of the communication tasks in the set β can be activated, and task 6 can also start execution (since it has received the information from the communication task 3.)

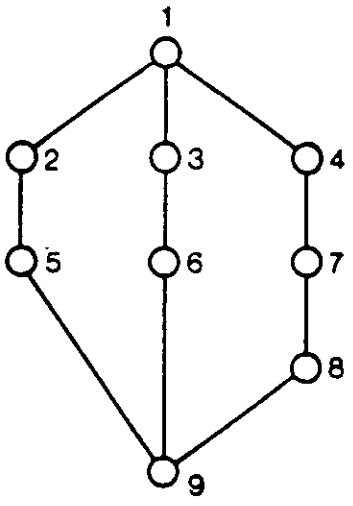


Figure 6.17 A Process Communication Graph

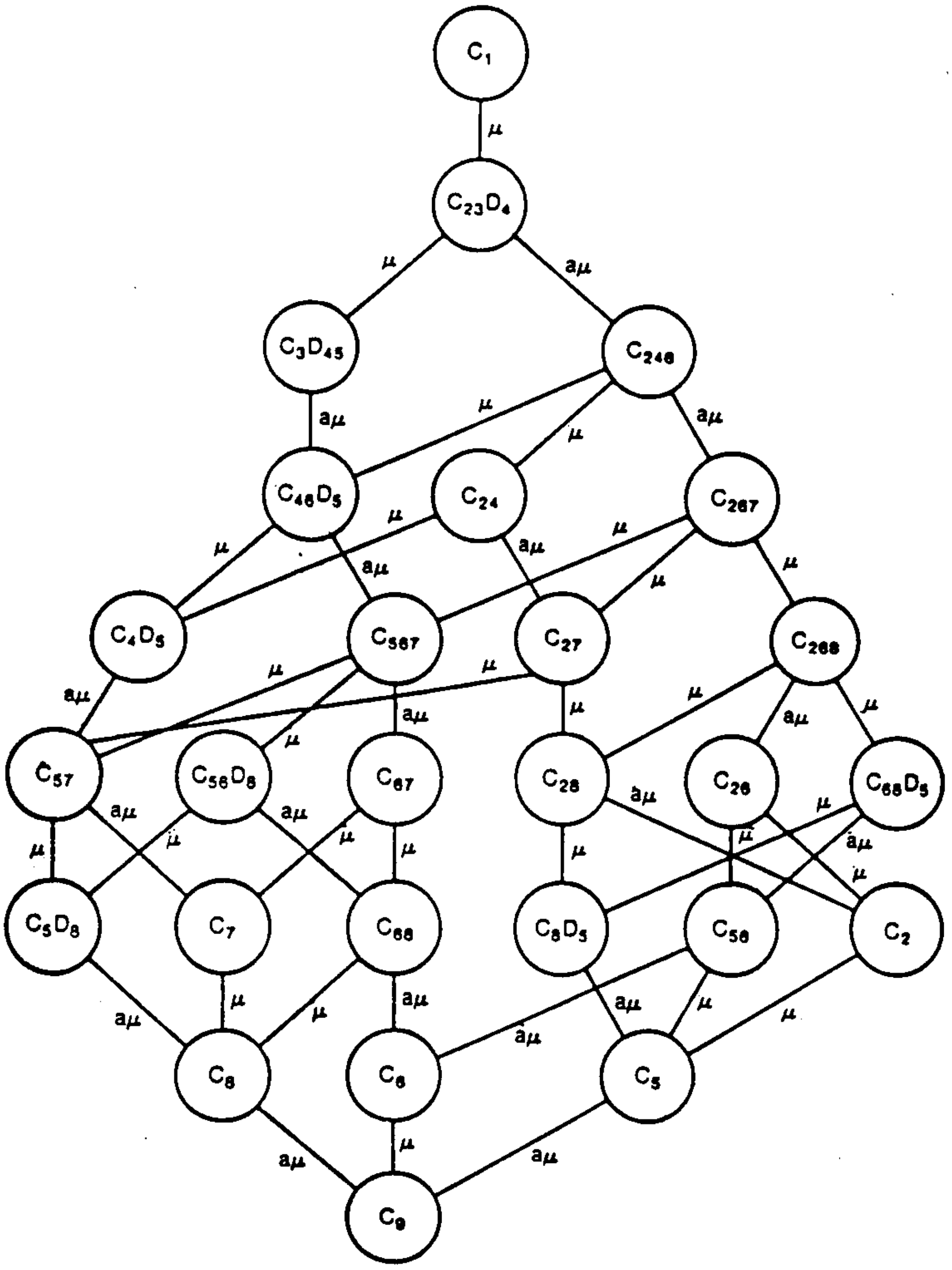


Figure 6.18 Markov Chain Generated from Figure 6.17

6.4 Discussion

In this chapter, we added the communication overhead to the process graphs. This overhead is represented as new nodes in a process graph. The resulting graph is called the process-communication graph and it may be converted into a Markov Chain to obtain the average system time.

When the number of processors is limited to P, we presented some rules of thumb on how tasks should be assigned to processors and where communication tasks must be added. This assignment was by no means the 'optimal' assignment. It was used so that we could analyze the processor communication graph and study a few facts regarding it. For example, we found the behavior of the μS curve when plotted against P and how it behaves when the parameter a increases. We also found the efficiency of the P processors which worked on a job.

Finally, we studied the communication problem when only one communication bus is allowed. The difference between this case and the two previous cases was when converting a process-graph into a Markov Chain, we cannot let more than two communication tasks to be executed at the same time. The additional communication tasks were kept in a first come first served queue.

CHAPTER 7
Extensions of This Work

Multi-processing is an attractive idea to speedup the processing of computer jobs. However, not all the jobs are suitable to be processed by multi-processors. For some jobs, because of the sequential nature of processing within them, multi-processing provides very little or no speedup. When the tasks within a job are assigned to more than one processor, the communication overhead between processors also reduces the maximum concurrency. Hence, we must study each case before concluding the feasibility of using multi-processors.

In this dissertation, we utilize the Graph Models of Behavior, described in Chapter 2, to model a computer job. This model defines the relationships among the tasks within a job. However, Graph Models of Behavior has the deficiencies of not able to model loops or recursions in a computer job.

Four parameters, discusses in Chapter 3, are used to characterize the concurrency problem into 16 cases. We have, in Chapters 4, 5 and 6, looked into 9 of these cases. For some cases, we are able to find the solutions; for other cases, we are able to use approximations to study the behavior of the system.

Several extensions, besides the 7 cases not studied in this dissertation, are:

- Instead of either having a bulk arrival at time zero and no arrivals after time zero, or having continuous arrival from a Poisson source, we can have another system in which there is a bulk arrival at time zero plus job arrivals after time zero.

- Instead of homogeneous processors, we can have heterogeneous processors. In the latter case, then, the assignment algorithms must also utilize each processor optimally.

- In Chapter 6, we assumed that the communication time among the processors has the same distribution. This is usually not true in real multi-processor environments. Processors further apart generally take longer time to communicate than processors close to each other. A more detailed model must be developed to take this communication delay characteristics into consideration.

References

[ABDE78] Abdel-Wahab, H.M., ad T. Kameda, 'Scheduling to Minimize Maximum Cumulative Cost Subject to Series-Parallel Precedence Constraints,' *Operations Research,* Vol. 26, No. 1, pp. 141-158, January-February 1978.

[AMDA67] Amdahl, G.M., 'Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,' *AFIPS Conference Proceedings,* Vol. 30, pp. 483-487, 1967.

[AVIZ81] Avizienis, A., 'Fault Tolerance by Means of External Monitoring of Computer Systems,' *AFIPS Conference Proceedings,* Vol. 50, pp.27-40, 1981.

[BAER68] Baer, J.L., and D.P. Bovet, 'Compilation of Arithmetic Expressions for Parallel Computations,' *Proceedings of IFIP Congress,* pp. 340-346, 1968.

[BAER69] Baer, J.E. and G. Estrin, 'Bounds for Maximum Parallelism in a Bilogic Graph Model of Computations,' *IEEE Transactions on Computers,* Vol. C-18, No. 11, pp. 1012-1014, November 1969.

[BAER70] Baer, J.L., D.P. Bovet, and G. Estrin, *Journal of the Association for Computing Machinery,* Vol. 17, No. 3, pp. 543-554, July 1970.

[BAKE78] Baker, Kenneth R., and Linus E. Schrage, 'Finding an Optimal Sequence by Dynamic Programming: An Extension to Precedence-Related Tasks,' *Operations Research,* Vol. 26, No. 1, pp. 111-120, January-February 1978.

[BARN68] Barnes, G., R. Brown, M. Kato, D. Kuck, P. Slotnick, and R. Stoker, *IEEE Transactions on Computers,* Vol. C-17, No. 8, pp. 746-757, August 1968.

[BEAT72] Beatty, J.C., 'An Axiomatic Approach to Code Optimization for Expressions,' *Journal of the Association for Computing Machinery,* Vol. 19, No. 4, pp. 613-640, October 72.

[BOVE68] Bovet, D.P., 'Memory Allocation in Computer Systems,' Ph.D. dissertation, Dep. Eng., University of California, Los Angeles, 1968.

[BRUN74] Bruno, J., E.G. Coffman, Jr., and R. Sethi, 'Scheduling Independent Tasks to Reduce Mean Finishing Time,' *Communications of the ACM,* Vol. 17, No. 7, pp. 382-387, July 1974.

[BRUN81] Bruno, J., P. Downey, and G.N. Frederickson, Flow Time or Makespan,' *Journal of the Association for Computing Machinery,* Vol. 28, No. 1, pp. 100-113, January 1981.

[BUX81] Bux, W., et al, 'A Reliable Token Ring System for Local Area Communication,' *National Telecommunication Conference,* pp. A2.2, 1981.

[CAPE79] Capetanakis, J.I., 'The Multi-Access Tree Protocol,' *IEEE Transactions on Communications,* Vol. COM-27, pp.1476-1484, October 1979.

[COFF76] Coffman, E.G., *Computer and Job-shop Scheduling Theory,* John Wiley and Sons, 1976.

[COFF79] Coffman, E.G., Jr., and Kimming So, 'On the Comparison Between Single and Multiple Processor Systems,' Department of Computer Science, University of California, Santa Barbara, August 1979.

[COHN78] Cohn, Harry, and Anthony Pakes, 'A Representation for the Limiting Random Variable of a Branching Process with Infinite Mean and Some Related Problems,' *J. Appl. Prob.,* Vol. 15, pp. 225-234, 1978.

[DEMP81] Dempster, M.A.H., et al, 'Analytical Evaluation of Hierarchical Planning Systems,' *Operations Research,* Vol. 29, No. 4, pp. 707-716, July-August 1981.

[DHAL78] Dhall, Sudarshan K., and C.L. Liu, 'On a Real-Time Scheduling Problem,' *Operations Research,* Vol. 26, No. 1, pp.127-140, January-February 1978.

[DODI81] Dodin, Bajes, and Naman, 'Random Network Generation,' University of N. Carolina, OR Report No. 179, June 1981.

[ELDE80] El-Dessouki, Ossama I., and Wing H. Huen, 'Distributed Enumeration on Between Computers,' *IEEE Transactions on Computers,* Vol. C-29, No. 9, pp. 818-825, September 1980.

[ENSL77] Enslow, Philip, Jr., 'Multiprocessor Organization - A Survey,' *Computing Surveys,* Vol. 9, No. 1, pp. 122-126, March 1977.

[ESTR63] Estrin, G., and R. Turn, 'Automatic Assignment of Computations in a Bariable Structure Computer System,' *IEEE Transactions,* Vol. EC-12, pp. 756-773, December 1963. [FELL67] Feller, William, *An Introduction to Probabilisty Theory and Applications,* Wiley, 1967.

[FERN72] Fernandez, E., 'Activity Transformations on Graph Models of Parallel Computations,' Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, October 1972.

[FORD62] Ford, L.R., and D.R. Fulkerson, *Flows in Networks,* Princeton University Press, 1962.

[FOSC81] Foschini, G.J., B. Gopinath, and J.F. Hayes, 'Optimum Allocation of Servers to Two Types of Competing Customers,' *IEEE Transactions on Communications,* Vol. COM-29, No. 7, pp. 1051-1055, July 1981.

[GALL68] Gallager, R., *Information Theory and Reliable Communication,* John Wiley & Sons, New York, 1968.

[GENT78] Gentleman, W. Morven, 'Some Complexity Results for Matrix Computations on Parallel Processors,' *JACM,* Vol. 25, No. 1, pp. 112-115, January 1978.

[GITT77] Gittins, J.C., and K.D. Glazebrook, 'On Bayesian Models in Stochastic Scheduling,' *J. Appl. Prob.,* Vol. 14, pp. 556-565, 1977.

[GITT79] Gittins, J.C., 'Bandit Processes and Dynamic Allocation Indices,' *J. R. Statist. Soc. Ser. B,* Vol. 41, No. 2, pp. 148-177, 1979. [GLAZ76] Glazebrook, P. Nash, 'On Multi-server Stochastic Scheduling,' *J. R. Statist. Soc. Ser. B,* Vol. 38, No. 1, pp. 67-72, 1976.

[GLAZ80] Glazebrook, K.D., 'On Stochastic Scheduling with Precedence Relations and Switching Costs,' *J. Appl. Prob.,* Vol. 17, pp. 1016-1024, 1980.

[GLAZ81] Glazebrook, K.D., and J.C. Gittins, 'On Single-Machine Scheduling with Precedence Relations and Linear or Discounted Costs,' *Operations Research,* Vol. 29, No. 1, pp. 161-173, January-February 1981.

[GONZ72] Gonzalez, Mario J., and C.V. Ramamoorthy, 'Parallel Task Execution in a Decentralized System,' *IEEE Transactions on Computers,* Vol. 21, No. 12, pp. 1310-1322, December 1972.

[GOTT82] Gottlieb, Allan, and J.T. Schwartz, 'Networks and Algorithms for Very-Large-Scale Parallel Computation,' *Computer,* Vol. 15, No. 1, pp.27-36, January 1982.

[HARR63] Harris, Theodore E., *The Theory of Branching Processes,* Prentice-Hall, 1963.

[HASE75] Hasen, P. Brian, 'The Programming Language Concurrent Pascal,' *IEEE Transactions on Software Engineering,* Vol. SE-1, No. 2, pp. 199-207, June 1975.

[HASE77] Hansen, P. Brian, *The Architecture of Concurrent Programs,* Prentice-Hall, Englewood Cliffs, N.J., 1977.

[HAYN82] Haynes, Leonard S., 'Highly Parallel Computing,' *Computer,* Vol. 15, No. 1, pp. 7-8, January 1982.

[HOFR78] Hofri, M. and C.J. Jenny, 'On the Allocation of Processes in Distributed Computing Systems,' IBM Research Report RZ905, April 1978.

[HOLT78] Holt, Graham, Lazowska, and Scott, *Structured Concurrent Programming with Operating System Applications,* Addison Wesley, Reading, Massachusetts, 1978.

[HORO76] Horowitz, Ellis, and Sartaj Sahni, 'Exact and Approximate Algorithms for Scheduling Nonidentical Processors,' *Journal of the ACM,* Vol. 23, No. 2, pp. 317-327, April 1976.

[IBAR77] Ibarra, Oscar H., and Chul E. Kim, 'Heuristic Algorithms for Acheduling Independent Tasks on Nonidentical Processors,' *Journal of the ACM,* Vol. 24, No. 2, pp. 280-289, April 1977.

[JAFF80] Jaffe, Jeffrey, 'Bounds on the Scheduling of Typed Task Systems,' *SIAM J. Comput.,* Vol. 9, No. 3, pp. 541-551, August 1980.

[JENN77] Jenny, C.J., 'Process Partitioning in Distributed Systems,' *Proceedings of National Telecommunications Conference,* pp. 31:1, 1977.

[KELL73a] Keller, Robert M., 'Parallel Program Schemata and Maximal Parallelism I: Fundamental Results,' *JACM,* Vol. 20, No. 3, pp. 514-537, July 1973.

[KELL73b] Keller, Robert M., 'Parallel Program Schemata and Maximal Parallelism II: Construction of Closures,' *JACM,* Vol. 20, No. 4, pp. 696-710, October 1973.

[KIES81] Kiesel, W., and P.J. Kuehn, 'CSMA-CD-DR: A New Multi-Acess Protocol for Distributed Systems,' *National Telecommunication Conference, pp. A2.4, 1981.*

[KLEI75] Kleinrock, L., *Queueing Systems, Vol. I: Theory,* John Wiley & Sons, New York, 1975.

[KLEI79] Kleinrock, L., 'Power and Deterministic Rules of Thumb for Probabilistic Problems in Computer Communications,' Conference Record, *International Conference on Communications,* pp. 43.1.1-43.1.10, June 1979.

[KNUT73b] Knuth, Donald, *The Art of Computer Programming -- Sorting and Searching,* Addison-Wesley, Reading, Massachusetts, 1973.

[KOZD80] Kozdrowicki, Edward W., and Douglas J. Thies, 'Second Generation of Vector Supercomputers,' *Computer,* Vol. 13, No. 11, pp. 71-83, November 1980.

[KUCK72] Kuck, David, et al, 'On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup,' *IEEE Transactions on Computers,* Vol C-21, No. 12, pp. 1293-1310, December 1972.

[KUCK74] Kuck, D., et al, 'Measurements of Parallelism in Ordinary FORTRAN Programs,' *Computer,* pp. 37-46, January 1974.

[KUCK77] Kuck, David J., 'A Survey of Parallel Machine Organization and Programming,' *ACM Computing Surveys,* Vol. 9, No. 1, pp. 29-59, March 1977.

[KUNG82] Kung, H.T., 'Why Systolic Architecture,' *Computer,* Vol. 15, No. 1, pp. 37-46, January 1982.

[LAM74] Lam, Simon S., 'Packet Switching in a Multi-Access Broadcast Channel with Application to Satellite Communication in a Computer Network,' Ph.D. dissertation, Computer Science Department, University of California, Los Angeles, April 1974.

[LANS78] Lenstra, J.K., and A.H.G. Rinnooy Kan, 'Complexity of Scheduling under Precedence Constraints,' *Operations Research,* Vol. 26, No. 1, pp. 22-35, January-February 1978.

[LEE77] Lee, Robert P., 'Optimal Task and File Assignment in a Distributed Computing Network,' PhD dissertation, Computer Science Department, University of California, Los Angeles, 1977.

[LELA77] Le Lann, Gerald, 'Distributed Systems - Towards a Formal Approach,' *IFIP Congress Proceddings*, pp. 155-160, 1977.

[LELA82] Leland, Will E., and Marvin H. Solomon, 'Dense Trivalent Graphs for Processor Interconnection,' *IEEE Transactions on Computers*, Vol. C-31, No. 3, pp. 219-222, March 1982.

[LITT61] Little, J., 'A Proof of the Queueing Formula $L = \lambda W$,' *Operations Research*, Vol. 9, No. 2, pp. 383-387, March 1961.

[LIU73] Liu, C.L., and James W. Layland, 'Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,' *Journal of the ACM*, Vol. 20, No. 1, pp. 46-61, Jamuary 1973.

[MAKA81] Makam, S., 'Design Study of a Fault-Tolerant Computer System to Execute N-Version Software,' Ph.D. Dissertation, UCLA Computer Science Department, June 1981.

[MART66] Martin. David F.,.'The Automatic Assignment and Sequencing of Computations on Parallel Processor Systems,' Ph.D. dissertation in Engineering, University of California, Los Angeles, 1966.

[MART67a] Martin, David, F., and G. Estrin, 'Experiments on Models of Computations and Systems,' *IEEE Transactions*, Vol. EC-16, pp. 59-69, February 1967.

[MART67b] Martin, David F., and G. Estrin, 'Models of Computational Systems - Cyclic to Acyclic Graph Transformations,' *IEEE Trasnactions on Electronic Computers*, Vol. EC-16, No. 1, pp. 70-79, February 1967.

[MART67c] Martin, David F., and G. Estrin, 'Models of Computations and Systems - Evaluation of Vertex Probabilities in Graph Models of Computations,' *J. ACM*, Vol. 14, pp. 281-299, April 1967.

[MART69] Martin, David F., and G. Estrin, 'Path Length Computations on Graph Models of Computations,' *IEEE Transactions on Computers*, Vol. C-18, No. 6, pp. 530-536, June 1969.

[METC76] Metcalfe, R.M., and D. R. Boggs, 'Ethernet: Distributed Packet Switching for Local Computer Networks,' *Communication of the ACM*, Vol. 19, No. 7, July 1976.

[MILL73] Miller, Raymond E., 'A Comparison of Some Theoretical Models of Parallel Computation,' *IEEE Transactions on Computers,* Vol. C-22, No. 8, pp. 710-717, August 1973.

[MOLL81] Molloy, Michael K., 'On the Integration of Delay and Throughput Measures in Distributed Processing Models,' Ph.D. dissertation, Computer Science Department, University of California, Los Angeles, 1981.

[MORR61] Morrison, P., and E. Morrison, *Charles Babbage and his Calculating Engines,* Dovers, Inc., N.Y., 1961.

[MUNT69] Muntz, Richard R., and E.G. Coffman, Jr., 'Optimal Preemptive Scheduling on Two-Processor Systems,' *IEEE Transactions on Computers,* Vol. C-18, No. 11, pp.1014-1020, November 1969.

[MURA71] Muraoka, Y., 'Parallelism Exposure and Exploitation in Programs,' Ph.D. dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, Report 71-424, February 1971.

[NG80] Ng, Y.W., and A. Avizienis, 'A Unified Reliability Model for Fault Tolerant Computers,' *IEEE Transactions on Computers,* Vol. C-29, No. 11, pp. 1002-1011, November 1980.

[OUST80] Ousterhout, John K., Donald A. Scelza, and Pradeep S. Sindhu, *Communications of the ACM,* Vol. 23, No. 2, pp. 92-105, February 1980.

[PAKE76] Pakes, A.G., 'Some Limit Theorems for a Supercritical Branching Process Allowing Immigration,' *J. Appl. Prob.,* Vol. 13, pp. 17-26, 1976.

[PAPO65] Papoulis, Anthanasios, *Probability, Random Variables, and Stochastic Processes,* McGraw-Hill, 1965.

[PARR83] Parrella, Michael, 'All Nodes Are "Equal" in Distributed Data Processing,' *System & Software,* p. 83, June 1983.

[PETE81] Peterson, James L., *Petri Net Theory and the Modeling of Systems,* Prentice-Hall, 1981.

[POPE81] Popek, G., B. Walker, J. Chow, D. Dewards, C Kline, G. Rudisin, and G. Thiel, 'Locus: A Network Transparent, High Reliability Distributed System,' *Proceedings of the Eighth Sysmposium on Operating Systems and Principles,* pp. 169-177, December 1981.

[POTT83] Potter, J. L., 'Image Processing on the Massively Parallel Processors,' *Computer,* Vol. 16, No. 1, pp. 62-67, January 1983.

[PREP81] Preparata, Franco P. and Jean Vuillemin, 'The Cube-Connected Cycles: A Versatile Netwrok for Parallel Computation,' *CACM,* Vol. 24, No. 5, pp. 300-309, May 1981.

[PRIC81] Price, Camille C., 'The Assignment of Computational Tasks Among Processors in a Distributed System,' *NCC,* pp. 291-296, 1981.

[PRIC83] Price, Camille C., 'Task Assignment Using A VLSI Shortest Path Algorithm,' Department of Computer Science, Austin State University, Nacogdoches, Texas, April 1983.

[RAMA69] Ramamoorthy, and M.J. Gonzalez, 'A Survey of Techniaues for Recognizing Parallel Processable Streams in Computer Programs,' *AFIPS, FJCC,* pp. 1-5, 1969.

[RAMA72] Ramamoorthy, C. V., K. M. Chandy, and Mario J. Gonzalez, Jr., 'Optimal Scheduling Strategies in a Multiprocessor System,' *IEEE Transactions on Computers,* Vol. C-21, No. 2, pp. 137-146, February 1972.

[RAMA80] Ramamoorthy, C. V. and Gray S. Ho, 'Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets,' *IEEE Transactions on Software Engineering,* Vol. SE-6, No. 5, pp. 440-449, September 1980.

[RAMI79] Ramirez, R.J., and N. Santoro, 'Distributed Control of Updates in Multiple-Copy Databases: A Time Optimal Algorithm,' *Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Netwrok,* pp. 191-234, 1979.

[RAO79] Rao, Gururaj, Harold Stone, and T.C. Hu, 'Assignment of Tasks in a Distributed Processor System with Limited Memory,' *IEEE Transactins on Computers,* Vol. C-28, No. 4, pp. 291-299, April 1979.

[ROBI79] Robison, J.T., 'Some Analysis Technique for Asynchronous Multiprocessor Algorithms,' *IEEE Trans. Software Eng.,* Vol. SE-5, pp.24-31, January 1979.

[ROSE83] Rosenfeld, Azriel, 'Parallel Image Processing Using Cellular Arrays,' *Computer,* Vol. 16, No. 1, pp. 14-20, January 1983.

[RUSH83] Rushby, John, and Brian Randell, 'A Distributed Secure System,' *Computer*, Vol. 16, No. 7, pp. 55-67, July 1983.

[RUSS69] Russell, E.C., 'Automatic Program Analysis,' Ph.D. dissertation, Dep. Eng., University of California, Los Angeles, 1969.

[SAHN76a] Sahni, Sartaj K., 'Algorithms for Scheduling Independent Tasks,' *JACM*, Vol. 23, No. 1, pp. 116-127, January 1976.

[SAHN76b] Sahni, Sartaj, amd Teofilo Gonzalez, 'P-Complete Approximation Problems,' *JACM*, Vol. 23, No. 3, pp. 555-565, July 1976.

[SCHE83] Schell, Roger R., 'A Security Kernel for a Multiprocessor Microcomputer,' *Computer*, Vol. 16, No.7, pp. 47-53, July 1983.

[SCHU81] Schuman, Stephen A., and Edmund M. Clarke, Jr., 'Programming Distributed Applications in ADA: A First Approach,' *Proceedings of the 1981 International Conference on Parallel Processing*, pp. 38-49, 1981.

[SEDG78] Sedgewick, R., 'Implementing Quicksort Programs,' *CACM*, Vol. 21, No. 10, pp. 847-857, October 1978.

[SENE73] Seneta, E., 'The Simple Branching Process with Infinite Mean,' *J. Appl. Prob.*, Vol. 10, pp. 206-212, 1973.

[SENE74] Seneta, E., 'Regulaly Varying Functions in the Theory of Simple Branching Processes,' *Adv. Appl. Prob.*, Vol. 6, pp. 408-420, 1974.

[SEVC74] Sevcik, Kenneth C., 'Scheduling for Minimum Total Loss Using Service Time Distributions,' *JACM*, Vol. 21, No. 1, pp. 65-75, January 1974.

[SIDN75] Sidney, Jeffrey B., 'Decomposition Algorithms for Single-Machine Sequencing with Precedence Relations and Deferral Costs,' *Operations Research*, Vol. 23, No. 2, pp. 283-298, March-April 1975.

[SIEW78a] Siewiorek, D.P., et al, 'A Case Study of Cmmmp, *Cm**, C.vmp, Part I: Experience with Fault-Tolerance in Microprocessors Systems,' *Proceedings of IEEE*, Vol. 66, No. 10, pp. 1178-1199, October 1978.

[SIEW78b] Siewiorek, D.P., et al, 'A Case Study of Cmmmp, *Cm**, C.vmp, Part II: Predicting and Calibrating Reliability of Multiprocessors Systems,' *Proceedings of IEEE*, Vol. 66, No. 10, pp. 1200-1220, October 1978.

[SIMO71] Simon, Richard, and Richard Lee, 'On the Optimal Solutions to AND/OR Series-Parallel Graphs,' *JACM,* Vol. 18, No. 3, pp. 354-372, July 1971.

[STON77] Stone, Harold S., 'Multiprocessor Scheduling with the Aid of Network Flow Algorithms,' *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, pp. 85-93, January 1977.

[THAN81] Thanawastien, S., and V.P. Nelson, 'Interference Analysis of Shuffle/Exchange Networks,' *IEEE Transactions on Computers,* Vol. C-30, No. 8, pp. 545-556, August 1981.

[UPFA82] Upfal, Eli, 'Efficient Schemes for Parallel Communications,' *Symposium on Principles of Distributed Computing,* pp. 55-59, August 1982.

[VANT81] Van Tilborg, Andre, and Larry D. Wittie, 'Distributed Task Force Scheduling in Multi-microcomputer Networks,' *National Computer Conference,* pp. 283-289, 1981.

[WEBE78a] Weber, Richard R., 'On Optimal Assigment of Customers to Parallel Servers,' *J. Appl. Prob.,* Vol. 15, pp. 406-413, 1978

[WEBE78b] Weber, Richard R., and Peter Nash, 'An Optimal Strategy in Multi-server Stochastic Scheduling,' *J. R. Statist. Soc. Ser. B,* Vol. 40, No. 3, pp. 322-327, 1978.

[WEI82] Wei, Martin, and Howard A. Scholl, 'An Expression Model for Extraction and Evaluation of Parallelism in Control Structures,' *IEEE Transactions on Computers,* Vol. C-31, No. 9, pp. 851-863, September 1982.

[WITT80] Wittle, P., 'Multi-armed Bandits and the Gittins Index,' *J. R. Statist. Soc. Ser. B,* Vol. 42, No. 2, pp. 143-149, 1980.

[WINS77] Winston, Wayne, 'Optimality of the Shortest Line Discipline,' *J. Appl. Prob.,* Vol. 14, pp. 181-189, 1977.

[WULF80] Wulf, W.A., R. Levin, and S.P. Harbison, *Hydra: An Experimental Operating System,* McGraw-Hill, New York, 1980.