INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Analytical performance prediction of parallel systems

Dawkins, William Price, Ph.D.

Rice University, 1993
RICE UNIVERSITY

Analytical Performance Prediction of Parallel Systems

by

William Price Dawkins

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

J. B. Sinclair
J. B. Sinclair, Associate Professor, Chairman
Electrical and Computer Engineering

J. Robert Jump, Professor
Electrical and Computer Engineering

Paul E. Pfeiffer, Professor
Computational and Applied Mathematics

Joseph R. Cavallaro, Assistant Professor
Electrical and Computer Engineering

Houston, Texas
April, 1993
Analytical Performance Prediction of Parallel Systems

by

William Price Dawkins

Abstract

The need for more computing power for many computer applications has increased beyond the capabilities of traditional von Neumann architectures. This has generated interest in using parallel computers to improve processing power. Because of the high cost of implementing parallel computers, accurate and efficient tools are needed to predict the performance of parallel computer designs to determine which designs should be built. This thesis presents an analytical technique for predicting the performance of parallel algorithms realized on parallel architectures. The technique models arbitrary non-deterministic task execution times, explicit precedence constraints, resource contention, and a variety of resource scheduling policies.

The program model used by this technique is a task graph. The nodes of a task graph represent parts of a parallel algorithm that require service from resources. The edges connecting nodes represent precedence constraints. Each task is assigned a random variable representing its execution time. The technique approximates arbitrarily distributed task execution time random variables with simple random variables. The different sequences of events that can occur due to resource contention are represented in a sequencing tree. A fully constructed sequencing tree represents all the possible sequences of events that can occur during the execution of a task graph. The technique allows the user to trade accuracy for efficiency by basing performance predictions on partial constructions of the sequencing trees.
The technique is implemented in a program called ES. ES predicts both the mean and standard deviation of the execution time of a task graph representing a parallel algorithm. ES predictions are compared to predictions of simulations and timings of real parallel algorithms executing on a real parallel computer. Two mergesort algorithms were implemented on a hypercube multicomputer. ES estimates of the mean execution times of these algorithms differ by at most 7.0% from the mean execution times measured on the hypercube. ES estimates of the execution time of an FFT algorithm are compared to estimates obtained from execution-driven simulations. The ES estimates are within ±0.15% of the estimates predicted by simulation.
Acknowledgments

I would like to express my thanks to the members of my thesis committee. Thanks to my advisor, Dr. J. B. Sinclair, who has given me guidance and patience over the last six years. Also, thanks to the other members of my committee, Dr. J. R. Jump, Dr. P. E. Pfeiffer, and Dr. J. R. Cavallaro. This research was supported by N.S.F. Award No. CCR-9006300 Parallel Algorithms for Hierarchical Memory Multiprocessors.

I would also like to thank Deirdre Hamilton, Roger Huang, Shridar Lakshamananmurthy, Ramakrishnan Rajamony, Haider Rizvi and the other students who have toiled with me over the years. Special thanks to Rajat Mukherjee and Sandhya Dwarkadas whom I have known since I first stepped foot on campus.

I would like to express my thanks to the many friends I have made in Houston. Thanks to Kevin Krouch, Amy Goldstein, Lynn Gann and Anne, Janis Arp, Glenda Arp, Earl Smalley, Sari Altman, Jennifer Elliott, Andrea Siskind, Piper Forse, Machelle Lerner, Dave Ramold, Lisa Faith Massey, and Mac and Leslie Pafford for their friendship. Words can not express my gratitude, and I will never forget them. Finally, I would like to thank Mike Duram, Jad DiCesare, Wayne Edmunds, John Skiptunas, Frank Veronic, Dutch, Randy, Ken, and anyone who ever went to Woodie’s Ice House.
# Table of Contents

## CHAPTER ONE
- 1.1 Motivation .................................................. 1
- 1.2 Problem Statement ........................................... 4
- 1.3 Organization of Thesis ....................................... 4

## CHAPTER TWO
- 2.1 Introduction .................................................. 6
- 2.2 Task Graphs ................................................... 7
  - 2.2.1 Series-Parallel Task Graphs ............................. 8
  - 2.2.2 Related Work Using Task Graphs ......................... 10
- 2.3 Distribution Approximation ................................... 13
  - 2.3.1 Related Work in Distribution Approximation .......... 14
  - 2.3.2 Moment Matching and Simple Random Variables ......... 16
  - 2.3.3 Advantages of Step Distributions ...................... 21
- 2.4 Sequencing Trees ............................................ 23
- 2.5 Construction of Sequencing Trees ......................... 26

## CHAPTER THREE
- 3.1 Introduction .................................................. 28
- 3.2 Simulation of Deterministic Task Execution Time Systems ... 28
- 3.3 Efficient Implementation of Sequencing Trees .............. 32
  - 3.3.1 Basic Sequencing Tree Construction Algorithm ....... 32
  - 3.3.2 Trading Accuracy for Efficiency ....................... 38
  - 3.3.3 Improved Sequencing Tree Construction Algorithm ..... 39
    - 3.3.3.1 Improved Algorithm Outline ....................... 40
    - 3.3.3.2 Explanation of Rules and Actions ................. 45
      - 3.3.3.2.1 Start Rule 1 .................................. 45
      - 3.3.3.2.2 Start Rule 2 .................................. 46
      - 3.3.3.2.3 Bookkeeping-After-Start Phase Action 1 ....... 47
      - 3.3.3.2.4 Bookkeeping-After-Start Phase Action 2 ....... 47
      - 3.3.3.2.5 Finish Rule 1 ................................ 48
      - 3.3.3.2.6 Finish Rule 2 ................................ 49
      - 3.3.3.2.7 Finish Rule 3 ................................ 52
      - 3.3.3.2.8 Bookkeeping-After-Finish Phase Action ....... 53
    - 3.3.3.3 Example of Improved Algorithm ................... 53
- 3.3.4 Simple Random Variables and the Sequencing Tree Construction Algorithms ....................... 56
- 3.3.5 Dependence of Random Variables ......................... 57
- 3.4 Summary ..................................................... 60

## CHAPTER FOUR
- 4.1 Introduction .................................................. 61
- 4.2 Comparisons Against Discrete-Event Simulations ............ 61
  - 4.2.1 Results of Tests on Partitioning Structured Task Graphs ... 63
  - 4.2.2 Results of Tests on Multi-Stage Linear Pipeline Structured Task Graphs ....................... 69
  - 4.2.3 The Efficiency of ES Compared to an Event-Driven Simulation ....................... 73
List of Figures

Figure 2.1: Example general task graph .................................................. 7
Figure 2.2: Series and parallel combinations of tasks ................................ 9
Figure 2.3: Example series connection of two task graphs ...................... 9
Figure 2.4: Non-series-parallel task graph ............................................... 9
Figure 2.5: k-stage coxian server ............................................................. 15
Figure 2.6: c.d.f. of a three-value simple random variable .................... 17
Figure 2.7: Approximation example, a three-valued step distribution
approximating a continuous distribution .............................................. 19
Figure 2.8: Example sequencing tree ....................................................... 24
Figure 2.9: Sequencing tree example ....................................................... 26
Figure 3.1: Example task system model .................................................. 29
Figure 3.2: Simulation of a deterministic task execution time system ....... 31
Figure 3.3: Basic algorithm outline ........................................................ 34
Figure 3.4: Basic algorithm flowchart ..................................................... 35
Figure 3.5: Example of basic algorithm .................................................. 37
Figure 3.6: Improved algorithm outline ................................................. 43
Figure 3.7: Improved algorithm flowchart .............................................. 44
Figure 3.8: Example task graph for Start Rule 2 ................................... 46
Figure 3.9: Example task graph for Bookkeeping-After-Start Phase Action 2 48
Figure 3.10: CONTENTION algorithm outline ....................................... 51
Figure 3.11: Example for Finish Rule 3 ................................................... 52
Figure 3.12: Improved algorithm example .............................................. 54
Figure 3.12: Improved algorithm example (cont.) ................................... 55
Figure 3.12: Improved algorithm example (cont.) ................................... 56
Figure 3.13: Common ancestor chain ..................................................... 58
Figure 3.14: Common series ancestor chain .......................................... 59
Figure 4.1: Accuracy test task graphs .................................................. 63
Figure 4.2: p.d.f. of triangular distribution ............................................. 64
Figure 4.3: Results for five-level partitioning task graphs with uniformly and
triangularly distributed task execution times ...................................... 66
Figure 4.4: Results for five-level partitioning task graphs with uniformly,
triangularly, and exponentially distributed task execution times ......... 68
Figure 4.5: Results for four-stage linear pipeline task graphs with uniformly
and triangularly distributed task execution times ................................ 71
Figure 4.6: Results for four-stage linear pipeline task graphs with uniformly,
triangularly, and exponentially distributed task execution times .......... 72
Figure 4.7: Varying confidence levels of the YACSIM simulations .......... 74
Figure 4.8: Example interconnection networks ...................................... 75
Figure 4.9: Mergesort task graph ............................................................ 77
Figure 4.10: Error of predicted mean execution time of mergesort ........... 79
Figure 4.11: Error of predicted mean execution time of mergesort with
sequential quicksort performed by sort tasks .................................... 80
Figure 4.12: Error of predicted mean of execution time of mergesort ....... 82
Figure 4.13: Error of predicted mean execution time of mergesort with
sequential quicksort performed by sort tasks .................................... 82
Figure 4.14: FFT task graph ................................................................. 85
Figure 4.15: Error of predicted execution time of 1024-point FFT algorithm .... 86
Figure 4.16: Task system with 18 tasks, six processors, and a single bus................. 87
Figure 4.17: Effect of varying cumulative terminal node probability level................. 89
Figure 4.18: Bottleneck detection........................................................................... 92
Figure 4.19: Communication network comparison ................................................. 93
CHAPTER ONE
Introduction

1.1 Motivation

The need for more computing power for many computer applications has increased beyond the capabilities of traditional von Neumann architectures. This has generated interest in using parallel computers to improve processing power. There are other methods for increasing processing power, such as the use of instruction buffers, cache memories, and pipelined execution. However, parallel architectures provide a method of using relatively inexpensive device technology at much higher performance ranges, making parallel architectures a cost effective way of enhancing performance [32].

The interest in parallel computers has inspired proposals for many new parallel architectures and new algorithms to exploit these new designs. It is often difficult to determine which design is the best for which applications. The cost of implementation and the time required to build new architectures prohibit building a new parallel computer just to test the capabilities of the design. Thus, accurate and efficient tools are needed to predict the performance of parallel computer designs before they are actually built.

The goal of this research is to achieve efficient and accurate predictions of parallel computers executing parallel algorithms. We require the performance prediction technique to be able to model:

- arbitrary task execution times
- explicit algorithm precedence and synchronization constraints
- resource contention
- a variety of scheduling policies for shared resources
A task is a part of a parallel program that requires service from some resource in the parallel system. The amount of service a task requires from a resource such as a processor, a disk, or a communication channel is the task execution time. Precedence and synchronization constraints are defined by the order in which different tasks of an algorithm must be completed. Resource contention occurs when two or more tasks of a parallel algorithm require the same resource (i.e., bus, disk, CPU) at the same time. A scheduling policy is defined as the order in which a shared resource services its customers, such as first-come-first-served or priority-based.

There have been many different methods used for the analytical performance prediction of computer systems. These methods include Markov chains [17, 31], queueing networks [5, 6, 20], and timed Petri nets [15]. All three methods have limitations. For example, the probability distributions used in continuous time Markov chains and timed Petri nets must be exponential (or Coxian in the more generalized case). Also, as the size of the system being modeled increases, the time required to solve the representative Markov chain or timed Petri net increases exponentially. Queuing networks are limited in that to make the solving of the model tractable certain assumptions must be made about the system being modeled. For instance, in separable or product-form queueing networks [5], if a resource in the system is modeled by a first-come-first-served (FCFS) queue, the service time distributions of all customers in the system at that resource must be identically distributed and exponential. Queuing networks are also unable to model precedence constraints, making them unsuitable for modeling parallel architectures executing specific parallel algorithms. Queuing networks and Markov chains are typically used to solve for steady-state behavior. This may be inappropriate when modeling parallel systems executing parallel algorithms where "steady-state" is not well-defined or applicable.
Several approximation techniques have been proposed that allow queueing networks to model FCFS queues servicing customers with non-exponential customer service time distributions [3, 13, 24]. Agrawal [2] presents several approximate queueing networks that overcome a number of restrictions of product-form queueing networks. However, none of these techniques allow the modeling of precedence constraints.

Other analytical techniques [1, 14, 20, 35] use various parameters to represent different aspects of the system, such as the number of processors, degree of parallelism in the algorithm, work per memory reference, and cache miss rate. These parameters are then used in analytical models that predict performance. Again, these techniques do not allow the modeling of explicit precedence and synchronization constraints.

Analytical techniques in general can be very efficient in making predictions. However, none of the techniques mentioned above (or any other known techniques) achieve all four of the modeling goals stated earlier. On the other hand, simulation allows the system being modeled to be described at virtually any level of detail. Precedence constraints, scheduling policies, and other features that prove difficult to incorporate in analytic models can be easily dealt with in a simulation. Simulation suffers from the drawback that it can be very slow. It is not uncommon for instruction-level simulations of uniprocessor systems to run one thousand times or more slower than the systems they model. Such detailed simulations of parallel systems on uniprocessors would be correspondingly slower. Furthermore, detailed simulation models can take a long time to implement, making analytical techniques more attractive, especially early in the process of making design decisions.

Ideally, the designer would use analytical techniques to quickly identify the good and bad designs, and then use detailed simulations to study and improve the better designs. This research is concerned with developing an efficient and accurate analytical technique incorporating all the modeling features stated earlier. The degree of accuracy is important because a more accurate analytical technique may allow the designer to reduce the number of
designs that would need to be simulated before the actual parallel system is built, thus shortening the overall time of the design process.

1.2 Problem Statement

The objective of this research is to develop an efficient and accurate technique for predicting the performance of specific parallel algorithms executing on specific parallel architectures. By accurate, we mean capable of making performance predictions that are useful in a variety of contexts, including comparisons of multiprocessor designs, studies of parallel program performance, and bottleneck detection. The technique must model arbitrary task execution times, explicit precedence and synchronization constraints, resource contention, and a variety of scheduling policies. The main performance metric with which we are concerned is the mean execution time of a parallel algorithm realized on a specific parallel architecture. A secondary performance metric is the standard deviation of the execution time. Using these metrics, the parallel computer architect can perform experiments evaluating the performance of different designs executing specific parallel algorithms. Such experiments can be used to identify bottlenecks between processors and communication networks. The designer can also compare the performance of different communication network structures. Tests could also determine the effects on performance of different mappings of parallel algorithm tasks to different processors.

1.3 Organization of Thesis

Chapter 2 describes the general approach of the analytical technique developed during this research. The chapter describes three methods used to achieve the four modeling goals stated earlier. The methods are task graphs, distribution approximation via moment matching, and sequencing trees. Chapter 2 also presents work related to these methods. Chapter 3 presents algorithms that combine the general methods described in Chapter 2 into
a technique for predicting the performance of parallel algorithms on parallel systems. We describe a basic algorithm that uses this technique and give an example of its operation. Improvements to this algorithm greatly improve the efficiency of the technique. We present an example of the use of the improved algorithm. Chapter 4 gives the results of experiments validating the accuracy of the technique. The technique is validated against discrete-event simulations, detailed simulations, and an actual parallel computer. The efficiency of the technique is also compared to the efficiency of the discrete-event simulations and the detailed simulations. Two sample applications of the technique are also given. One is a simple experiment for determining communication network bottlenecks. The second is a comparison of the performance of three parallel systems with different communication structures. Chapter 5 summarizes the thesis and presents directions for future work.
CHAPTER TWO

An Analytic Technique for Modeling Parallel Systems

2.1. Introduction

The previous chapter enumerated the modeling goals of the analytical technique we developed. Stated again, the technique must model arbitrary task execution times, explicit algorithm precedence and synchronization constraints, resource contention, and a variety of scheduling policies for shared resources. The analytical technique we have developed uses a combination of the following methods to realize these goals:

- task graphs
- distribution approximation via moment matching
- sequencing trees

Task graphs model the various precedence and synchronization constraints between specific tasks of a parallel algorithm. By using distribution approximation, our analytical technique can approximate arbitrary distributions efficiently. Sequencing trees are a form of decision tree which represent the various sequence of events that can occur due to resource contention and different resource scheduling policies. This chapter describes these methods in detail. The next chapter presents implementations of our technique that uses these three methods to make performance predictions of parallel algorithms executing on parallel systems.
2.2. Task Graphs

Many parallel algorithms consist of a set of tasks that can be represented by the nodes of a directed acyclic graph called a general task graph. Figure 2.1 shows an example of a general task graph.

![Example general task graph](image)

**Figure 2.1:** Example general task graph

We define a task as the computations between two successive synchronization or interaction points of a parallel algorithm. Once started, the computations of a task run to completion without interruption. Any interactions between tasks take place only at the beginning and end of the tasks [21]. The directed edges in a task graph represent precedence constraints. Precedence constraints define the explicit order in which tasks must execute. For example, task $b$ in Figure 2.1 may not begin execution until $a$ has completed. Task $a$ is a predecessor of task $b$, and $b$ is a successor of $a$. Tasks that have no predecessors are initial tasks, and tasks with no successors are final tasks. A task may not begin execution until all of its predecessors have completed. A task $j$ is a descendant of task $i$ if there is a path in the task
graph from $i$ to $j$. In that case, we also say that $i$ is an ancestor of $j$. In Figure 2.1, task $b$'s descendants are tasks $e$ and $g$. Task $e$'s ancestors are tasks $a$, $b$, and $c$.

Each task in the graph has a random variable associated with it that represents the amount of service time the task requires on some resource. These random variables are called task execution times. We assume that all task execution time random variables are independent of one another. Each task must be allocated a resource in the parallel system for its execution. The task must have exclusive use of this resource before it can execute. When two or more tasks are queued at the same resource, a scheduling policy associated with the resource determines which task will execute first. Task graphs with assigned task execution time distributions and resources for all tasks are called task systems.

Task graphs are best suited for use in modeling parallel algorithms with large-grain parallelism. Programs with large-grain parallelism are characterized by relatively long periods of computation between task interactions. For example, large-grain parallel programs are usually written as separate processes that exchange information, through message passing, after performing a number of sequential operations. This is in contrast to fine-grain parallelism, where parallelism is usually expressed in parallel programming constructs like parallel do-loops. In fine-grain parallelism, synchronization can occur as often as each processor finishes one sequential operation.

### 2.2.1. Series-Parallel Task Graphs

Several analytical techniques that use task graphs to model parallel programs limit the type of task graphs that can be studied to a special subclass of general task graphs called series-parallel task graphs [22, 23, 29]. Series-parallel task graphs are constructed by combining smaller series-parallel graphs (beginning with single nodes) in either series or parallel [29]. Figure 2.2 gives examples of simple series-parallel task graphs.
Figure 2.2: Series and parallel combinations of tasks

Figure 2.3: Example series connection of two task graphs

Figure 2.4: Non-series-parallel task graph
More formally, a finite directed graph is defined to be an ordered quadruple $G = (N, A, S, T)$ where

a) $N$ is a finite set of elements called tasks.

b) $A \subseteq N \times N$ is the set of precedence constraints.

c) $S \subseteq N$ is the set of all tasks with no predecessors. This is the set of initial tasks

d) $T \subseteq N$ is the set of all tasks with no successors. This is the set of final tasks.

Suppose $G_1 = (N_1, A_1, S_1, T_1)$ and $G_2 = (N_2, A_2, S_2, T_2)$ are graphs where $N_1$ and $N_2$ are mutually exclusive sets. A graph $G = (N, A, S, T)$ is the series connection of $G_1$ and $G_2$ if and only if

a) At least one of the two sets $T_1$ and $S_2$ is a singleton.

b) $N = N_1 \cup N_2$.

c) $A = A_1 \cup A_2 \cup (T_1 \times S_2)$.

d) $S = S_1, T = T_2$.

This allows series connections like that shown in Figure 2.3. A graph $G$ is the parallel combination of $G_1$ and $G_2$ if and only if

a) $N = N_1 \cup N_2$.

b) $A = A_1 \cup A_2$.

c) $S = S_1 \cup S_2, T = T_1 \cup T_2$.

Series-parallel graphs are defined as the class of graphs containing the unit graphs (consisting of one node) and having the property that any series or parallel connection of two graphs in the class is also in the class [29]. Figure 2.4 gives a simple example of a task graph that is not series-parallel.

2.2.2. Related Work Using Task Graphs

Robinson [28] uses task graphs to predict bounds on the mean execution time of parallel algorithms in the absence of resource contention. Robinson's bounds are valid
when all the task execution times of tasks on the same level of a task graph have independent and identically distributed (i.i.d.) random variables. Robinson gives the following lower and upper bounds on the mean execution time $E[T_G]$ of a task graph $G$ given the restrictions stated earlier.

$$\sum_{1 \leq j \leq L} \mu_j \leq E[T_G] \leq \sum_{1 \leq j \leq L} \left( \mu_j + \frac{m_j - 1}{\sqrt{2m_j - 1}} \sigma_j \right) \tag{2.1}$$

In the above equation, $L$ is the number of levels in the task graph $T_G$. $m_j$ is the number of tasks on level $j$, and these tasks have i.i.d. execution time random variables with mean $\mu_j$ and standard deviation $\sigma_j$. The upper bound in Eq. 2.1 is actually the mean execution time of a modified task graph where all tasks at a particular level synchronize at the end of execution [21].

Madala [21] expands on Robinson's bounds by using specific knowledge of the distributions of the task execution time random variables at each level. By knowing the types of the distributions (i.e., exponential, uniform, normal), Madala determines new bounds on mean execution time that are not based solely on the means and standard deviations of the tasks. Madala also presents a number of different approximations for mean execution times for task graphs with regular structures that are more accurate than the bounds of Eq. 2.1. However, like Robinson, Madala also ignores resource contention.

Thomason and Bay [33, 34] use a task graph to represent an algorithm and a queuing network to represent the resources in a system. Using these models, they develop a Markov chain where the states represent the tasks currently executing on the resources. The transition rates between states are the throughputs of the system resources in processing various combinations of tasks. Closed queuing network models are solved to find the throughputs. One limitation of this approach is exponential time complexity in the number
of tasks. The Markov chain developed from a task graph with $l$ tasks can have as many as $2^l - 1$ states [34]. This means the number of closed queueing networks (one for each state) that must be solved can also grow exponentially.

Mak and Lundstrom [22, 23] use an approach similar to Thomasian and Bay. They also use task graphs to represent algorithms and closed queueing networks to represent resources. However, Mak and Lundstrom use an iterative approach that does not require the generation of Markov chains to solve their model. This makes the method computationally more efficient. Since Markov chains are not generated, the number of queueing network models that must be solved do not grow exponentially as the size of task graphs increase. This approach and the Thomasian-Bay approach have the limitations imposed by the use of product-form queueing network models described in [5]. For example, no priority-based scheduling policies are allowed.

Both Mak-Lundstrom and Thomasian-Bay use closed queueing network models to account for the contention of tasks for resources. Such closed models may not be appropriate for predicting the performance of open systems. In solving closed queueing networks, the throughputs and residence times computed for the tasks in the system are calculated when the system is in steady state. The performance predictions determined by both these methods are not for steady state conditions. Once a task completes its service requirements, it leaves the system. The residence time of a task computed from a closed queueing network model may differ from its true residence time in an open system.

The approach developed by Sahner and Trivedi [29] allows task execution time distributions to be polynomial exponentials. Polynomial exponentials have the form

$$F(t) = \sum_i a_i t^k e^{b_i t}.$$  \hspace{1cm} (2.2)
The $k_i$'s are non-negative integers, and the $a_i$'s and $b_i$'s are real or complex numbers. The $k_i$'s, $a_i$'s and $b_i$'s must be chosen such that $F(t)$ is a legal cumulative distribution function (c.d.f.) of a non-negative random variable. This class is also the class of Coxian distributions described in [10, 11].

Sahner and Trivedi assign Coxian distributed execution time random variables to each task in a task graph. By performing a series of convolutions and multiplications of these distributions, they can determine the distribution of the execution time of the task graph. The major limitation of the Sahner-Trivedi approach is that it ignores resource contention. Another limitation of the Thomasian-Bay, Sahner-Trivedi and Mak-Lundstrom approaches is that they require that all task graphs be series-parallel. This restricts the types of algorithms they can model to algorithms that have a series-parallel structure. Our approach does not require the modeling of algorithms with series-parallel task graphs. This means our approach can be applied to any algorithm that can be represented by a directed acyclic graph.

2.3. Distribution Approximation

One goal of this research is to model tasks whose execution times are represented by arbitrarily distributed random variables. Other analytical techniques that predict the performance of task systems place limits on the types of distributions used for task execution times. Mak-Lundstrom and Thomasian-Bay allow only the distributions permitted for product-form queuing networks [5], including the limitations on scheduling policies associated with these distributions. For example, the techniques require that all tasks served by the same first-come-first-served (FCFS) queue have identical exponentially distributed execution times. Sahner-Trivedi use Coxian distributions [11] for task execution times but provide no method for approximating arbitrary distributions with the Coxian distributions.
Both Mak-Lundstrom and Thomasian-Bay assume that the residence time a task spends in a queueing network is exponentially distributed. The residence time is the time a task spends getting service at resources and waiting for service in queues. This assumption means that no matter what type of distribution is assigned to task execution time random variables, the total amount of time a task spends in the resource system is modeled with an exponential distribution.

Restrictions on task execution time distributions place limits on the applicability of these techniques. By allowing arbitrarily distributed task execution times, the range of systems that can be modeled greatly increases. It would be difficult to include all distributions in the task system modeling technique developed by this research. Our technique requires certain operations to be performed on distributions, such as multiplication and convolution. Developing efficient methods for performing these operations on all possible classes of distributions is a difficult problem. Instead, the approach we took was to approximate arbitrarily distributed random variables with a single class of random variables.

2.3.1. Related Work in Distribution Approximation

In [11], Cox described a type of distribution that is known as the phase or Coxian type. Eq. 2.2 is the c.d.f. of a Coxian distribution. Figure 2.5 shows a server consisting of a (possibly infinite) series of stages which has a Coxian service time distribution [4]. $\beta_i$ is the probability of a job entering stage $i$. $1 - \beta_i$ is the probability of a job exiting the server before entering stage $i$. $1 - \beta_1$ is the probability that a job bypasses all stages. The amount of time a job spends in stage $i$ is exponentially distributed with rate $\lambda_i$. The Laplace transform of a $k$-stage Coxian distribution is shown in Eq. 2.3.

$$f^*(s) = p_0 + \sum_{i=1}^{k} q_0 \cdots q_{i-1} p_i \prod_{l=1}^{i} (1 + s/\lambda_l)^{-1}$$ (2.3)
Any distribution with a rational Laplace transform can be represented exactly by a Coxian distribution. Furthermore, any distribution can be approximated arbitrarily closely by a Coxian distribution.

There have been several studies that used Coxian distributions to approximate arbitrary distributions. These include Marie [24] and Altio [4]. Given a random variable $T$ with a p.d.f. $f_T(t)$, the moments of $f_T(t)$ are given by

$$m_i = \int_{-\infty}^{\infty} t^i f_T(t) dt. \quad (2.4)$$

$m_0$ is always equal to one. If $f_T(t)$ adheres to certain constraints on the squared coefficient of variation, it is possible to approximate $f_T(t)$ with two-stage Coxian distributions that match a predetermined number of moments $f_T(t)$. The squared coefficient of variation $c$ is defined as $(\sigma/m_1)^2$. Marie presents a method to match the first three moments $m_0 - m_2$ of the original distribution, and Altio uses a different approach to match $m_0 - m_3$. Marie's approach requires that the distribution have $c > 0.5$. Altio's method has two conditions that an arbitrary distribution must meet to be approximated by a two-stage Coxian distribution with the same first four moments. These conditions are:
\[ \frac{m_3}{m_1} > \frac{3}{2} (c + 1)^2 \]  \hspace{1cm} (2.5)

\[ \frac{[(c+1)^2 - (3-c)^2]}{m_1^2(c+1)^2} \geq 0 \]  \hspace{1cm} (2.6)

The second condition is satisfied when \( c \) is greater than 1.0.

2.3.2. Moment Matching and Simple Random Variables

The techniques used by Altijk and Marie belong to a much larger class of approximation methods known as distribution approximation via moment matching. In these methods, arbitrary distributions are approximated by matching a predetermined number of their moments to the moments of the approximating distributions. We also use moment matching to approximate the distributions of task execution time random variables. Since task execution times are non-negative, we replace the lower limit of Eq. 2.4 with 0. We are also particularly concerned with task execution time random variables that have a finite upper bound. If a random variable \( T \) is bounded by some lower limit \( t_L > 0 \) and by some upper limit of \( t_U < \infty \), then the moments of its distribution are given by

\[ m_i = \int_{t_L}^{t_U} t^i f_T(t)dt. \]  \hspace{1cm} (2.7)

We call these distributions bounded distributions.

The true distributions of the execution times of the tasks in parallel algorithms are not usually known. Often it may be possible to infer the general shape of the distribution from knowledge of the tasks. We can then choose a distribution with a similar shape and
appropriate parameters. For instance, we may be able to determine that a task execution time has lower and upper bounds \( t_L \) and \( t_U \), and that any time between \( t_L \) and \( t_U \) is about as likely as any other time. We could model the unknown distribution with a uniform distribution, which is completely specified by \( t_L \) and \( t_U \). Similarly, a unimodal task execution time distribution with known lower and upper bounds could be modeled as a triangular or a truncated Gaussian distribution, with the choice perhaps depending on our information or assumptions about higher order moments. It may also be possible to determine estimators for the moments and use these estimators in place of the moments themselves. However, estimators for moments beyond \( m_2 \) or \( m_3 \) have notoriously high variances, which is likely to make this approach problematic in most practical situations.

We model arbitrary random variables with a type of random variable called \textit{simple random variables} [27]. We will discuss the advantages of using simple random variables in Section 2.3.3. Simple random variables have a finite number of values. Figure 2.6 shows a c.d.f. of a three-valued simple random variable \( T \). We call this type of distribution a \textit{step distribution}.

\begin{figure}[h]
\centering
\includegraphics[width=0.6\textwidth]{fig2_6.png}
\caption{c.d.f. of a three-value simple random variable}
\end{figure}

The equation of the c.d.f. shown in the above figure is
\[ F_T(t) = p_1 u(t-t_1) + p_2 u(t-t_2) + p_3 u(t-t_3), \]  
\[ (2.8) \]

and its corresponding probability mass function (p.m.f.) is

\[ f_T(t) = p_1 \delta(t-t_1) + p_2 \delta(t-t_2) + p_3 \delta(t-t_3). \]  
\[ (2.9) \]

\( t_1, t_2, \) and \( t_3 \) are the values that \( T \) can have. \( p_1, p_2, \) and \( p_3 \) are the probabilities that \( T \) has value, \( t_1, t_2, \) and \( t_3, \) respectively.

Figure 2.7 illustrates the approximation via moment matching technique for simple random variables. In this case a three-valued simple random variable \( T' \), approximates an arbitrarily distributed random variable \( T \). If we want the first four moments of \( F_T(t) \) to match the first four moments of \( F_T(t) \), the following equations must hold.

\[ m_0 = p_1 + p_2 + p_3 \]  
\[ (2.10) \]

\[ m_1 = p_1 t_1 + p_2 t_2 + p_3 t_3 \]  
\[ (2.11) \]

\[ m_2 = p_1 t_1^2 + p_2 t_2^2 + p_3 t_3^2 \]  
\[ (2.12) \]

\[ m_3 = p_1 t_1^3 + p_2 t_2^3 + p_3 t_3^3 \]  
\[ (2.13) \]

\( m_0 \) through \( m_3 \) are the first four moments of \( F_T(t) \), and \( T' \) has value \( t_1, t_2, \) and \( t_3 \) with probability \( p_1, p_2, \) and \( p_3, \) respectively. The equations are further constrained by the fact that \( 0 \leq p_1, p_2, p_3 \leq 1 \) and \( t_1 < t_2 < t_3. \)
Figure 2.7: Approximation example, a three-valued step distribution approximating a continuous distribution

If the distribution of $T$ were bounded and the values of $t_1$ and $t_3$ were set equal to $t_L$ and $t_U$ respectively, then not only would $F_T(t)$ have the same first four moments of $F_T(t)$ but also the same upper and lower limits. This presupposes that values for $p_1, p_2, p_3,$ and $t_2$ can be found that solve Eqs. 2.10 through 2.13.

Sinclair [30] showed that given $m_0 - m_3$, we can determine $p_1, p_2, p_3,$ and $t_2$ in Eqs. 2.10 through 2.13 as functions of $t_1$ and $t_3$, where $t_1$ and $t_3$ can have any values that satisfy the following constraints.

\[
t_1 \leq t_1(\text{max}) = m_1 + \sigma \left[ a_3 - (1 + a_3^2)^{1/2} \right] \quad (2.14)
\]

\[
t_3 \geq t_3(\text{min}) = m_1 + \sigma \left[ a_3 + (1 + a_3^2)^{1/2} \right] \quad (2.15)
\]

The momental skew $a_3$ is defined as

\[
a_3 = \frac{\mu_3}{2\sigma^3}. \quad (2.16)
\]
Define

\[ h(x, y) = x^2y + m_2y - 2m_1xy + 2m_2x - m_1x^2 - m_2 \]  \hspace{1cm} (2.17)

\[ n(x) = (m_2 - m_1^2)x^2 - (m_3 - m_1m_2)x + m_1m_3 - m_2^2 \]  \hspace{1cm} (2.18)

Eqs. 2.19 through 2.22 give the formulas for determining the values of \( p_1, p_2, p_3, \) and \( t_2 \) that exactly match the first four moments of \( F_T(t) \) for any \( t_1 \leq t_1(max) \) and any \( t_3 \geq t_3(min) \).

\[ p_1 = \frac{n(t_3)}{(t_3 - t_1)h(t_1, t_3)} \]  \hspace{1cm} (2.19)

\[ p_2 = \frac{[t_1t_3 - m_1(t_1 + t_3) + m_2]^3}{h(t_1, t_3)h(t_2, t_1)} \]  \hspace{1cm} (2.20)

\[ p_3 = \frac{n(t_1)}{(t_1 - t_3)h(t_3, t_1)} \]  \hspace{1cm} (2.21)

\[ t_2 = \frac{m_1t_1t_3 - m_2(t_1 + t_3) + m_3}{t_1t_3 - m_1(t_1 + t_3) + m_2} \]  \hspace{1cm} (2.22)

As stated earlier, \( t_1 \) and \( t_3 \) can have any values that satisfy Eqs 2.14 and 2.15. If \( F_T(t) \) is a bounded distribution, \( t_1 \) and \( t_3 \) could be chosen so that they equal \( t_L \) and \( t_U \) respectively. Then, the approximate distribution \( F_T(t) \) would match the bounds of \( F_T(t) \) as well as its first four moments. If the bounds of \( F_T(t) \) are not known or it is an unbounded distribution (\( t_L = 0, t_U = \infty \)), we use the following equations to determine \( t_1 \) and \( t_3 \).

\[ t_1 = t_1(max) - (1 - f)\delta_{min} \]  \hspace{1cm} (2.23)
\[ t_3 = t_3(\text{min}) + (1 - f)\delta_{\text{min}} \]  \hspace{1cm} (2.24)

\[ \delta_{\text{min}} = \min(t_3(\text{max}) - t_L, t_u - t_3(\text{min})) \]  \hspace{1cm} (2.25)

This is called a symmetric approximation because by varying \( f \) from 0 to 1 the positions of \( t_1 \) and \( t_3 \) are placed symmetrically around \( t_1(\text{max}) \) and \( t_3(\text{min}) \). The value of \( f \) has a strong influence on the accuracy obtained when using step distributions during the modeling of task graphs, especially with regard to estimates of the standard deviation of the task graph execution time [30]. The value of \( f \) that produces the most accuracy changes depending on the task graph being modeled.

The implementation presented in the next chapter uses the above equations to approximate arbitrary distributions with three-valued step distributions that match the first four moments of the arbitrary distributions. The user has the option of matching the bounds of the original distribution or using the symmetric approximation technique. The value of \( f \) in Eqs. 2.23 and 2.24 is also user definable. If \( f = 1 \), then \( t_1 = t_1(\text{max}), t_3 = t_3(\text{min}), t_2 \) will equal \( t_1 \) or \( t_3 \), and \( T' \) will be a two-valued simple random variable.

2.3.3. Advantages of Step Distributions

In our performance prediction technique, we match the first four moments of the original distributions of task execution time random variables. We also allow the option of modeling the limits of bounded distributions. We achieve this by using Eqs. 2.14 through 2.25 to approximate arbitrary distributions with step distributions. There is no restriction on the squared coefficient of variation \( c \) as in Marie [24] and Altıok [4].

There are several advantages to using step distributions. First, they are simple to represent with data structures by recording the values of the \( p_i \)'s and \( t_i \)'s of their cumulative distribution functions. Second, the convolutions and multiplications of step distributions re-
quired by our technique are computationally efficient. Eq. 2.28 shows the convolution of the distributions of two two-valued simple random variables whose p.d.f.s are given in Eqs. 2.26 and 2.27. Eq. 2.31 shows the multiplication of the distributions of two two-valued simple random variables whose c.d.f.s are given in Eqs. 2.29 and 2.30.

\[
f_A(t) = p_1 \delta(t - x_1) + p_2 \delta(t - x_2) \tag{2.26}
\]

\[
f_B(t) = m_1 \delta(t - y_1) + m_2 \delta(t - y_2) \tag{2.27}
\]

\[
f_A(t) \otimes f_B(t) = p_1 m_1 \delta(t - (x_1 + y_1)) + p_2 m_2 \delta(t - (x_1 + y_2)) + p_2 m_1 \delta(t - (x_2 + y_1)) + p_2 m_2 \delta(t - (x_2 + y_2)) \tag{2.28}
\]

\[
F_A(t) = p_1 u(t - x_1) + p_2 u(t - x_2) \tag{2.29}
\]

\[
F_B(t) = m_1 u(t - y_1) + m_2 u(t - y_2) \tag{2.30}
\]

\[
F_A(t) \cdot F_B(t) = p_1 m_1 u(t - \max(x_1, y_1)) + p_1 m_2 u(t - \max(x_1, y_2)) + p_2 m_1 u(t - \max(x_2, y_1)) + p_2 m_2 u(t - \max(x_2, y_2)) \tag{2.31}
\]

It should be noted that convolving and multiplying step distributions typically generates distributions with larger numbers of values. Convolving or multiplying an \(n\)-step distribution with an \(m\)-step distribution results in a step distribution with a maximum of \(nm\) steps. As more and more convolutions and multiplications are done, the number of steps in the distributions become larger and the calculations become slower. When the results of the convolutions and multiplications of simple random variables become large enough to degrade efficiency, we approximate the results with simple random variables that have smaller numbers of values by reapplying Eqs. 2.14 through 2.25.
Finally, the use of simple random variables facilitates the construction of the decision trees described in the next section. During the creation of these trees, it is necessary to calculate the probability of one task finishing before another task. Suppose tasks \( a \) and \( b \) have non-negative finish times represented by random variables \( A \) and \( B \), respectively. Assuming \( A \) and \( B \) are independent, the probability that task \( a \) finishes before task \( b \) is

\[
\Pr(A < B) = 1 - \int_0^\infty F_B(t) f_A(t) dt,
\]

and the conditional c.d.f. of \( A \) given \( A < B \) is

\[
\Pr(A \leq t \mid A < B) = \frac{F_A(t) - \int_0^t F_B(\tau) f_A(\tau) d\tau}{1 - \int_0^\infty F_B(t) f_A(t) dt}.
\]

When \( A \) and \( B \) are simple random variables, Eqs. 2.32 and 2.33 can be computed efficiently.

2.4. Sequencing Trees

Our approach uses sequencing trees to model the sequencing decisions that arise due to resource contention. Sequencing decisions occur when two or more tasks that could execute in parallel use the same resource, and there is a non-zero probability that their executions overlap. Each possible outcome of a sequencing decision can influence the activity that follows, and thus may change the sequence of events and affect execution time. Figure 2.8 gives an example of a sequencing tree.

Each node of the tree represents a state of the execution of a task graph. State is defined as the condition of the tasks (i.e., unexecuted, executing, completed), the condition of the individual resources (i.e., idle, running), and which tasks resources are currently ser-
The directed edges in the sequencing tree represent the outcomes of sequencing decisions. The $p_i$'s, called the branch probabilities, are the probabilities that a sequencing decision results in a particular outcome. Once a node is created, it is independent of other nodes until a sequencing decision requires the node to generate new nodes representing the outcomes of the decision.

In Figure 2.8, we calculate a random variable $T_A$ representing the execution time of a task graph until a sequencing decision is required (i.e., two or more tasks need the same resource). Next, we calculate the probabilities of each outcome (i.e., the probability of each task getting to the resource first). We then make a copy of the task graph for each possible decision outcome and adjust these copies to account for the result of making the decision. A simple example will follow later. $T_{A_i}$ is the random variable representing the execution time of outcome $A_i$ given $A_i$ occurs. Eq. 2.34 shows the expected value of $T$ in Figure 2.8 based on the assumption that $T_{A_1} - T_{A_n}$ are independent of one another and of $T_A$ (i.e., $F_{XY}(t,u) =$
\( F_X(t)F_Y(u) \) where \( X \) and \( Y \) can be any possible combination of the random variables \( T_{A_1}, T_{A_2}, \ldots, T_{A_n} \).

\[
E[T] = E[T_{A_1}] + p_1 E[T_{A_2}] + p_2 E[T_{A_3}] + \cdots + p_n E[T_{A_n}]
\]  \hspace{1cm} (2.34)

Applying this technique recursively for \( A_1 \) through \( A_n \), we can calculate the execution time of a task graph taking into account resource contention. The problem with this technique is that as the size of the task graph model increases, the sequencing tree can grow exponentially. To curb this exponential growth, we have developed a heuristic method for basing predictions on partial constructions of sequencing trees. The method allows the trading of accuracy for efficiency. It is described in Chapter 3.

Figure 2.9 shows an example of the partial construction of a sequencing tree. Suppose the program represented by the task graph in Figure 2.9 is executing on a three-CPU parallel processor. The number inside each node is the number of the CPU the task requires. The first task graph shows that task 1 has already executed on CPU 1, and tasks 2 and 3 are executing on CPUs 1 and 2, respectively. At this point a sequencing decision arises. It must be determined whether task 2 or task 3 finishes first because this will determine whether task 4 or task 5 gets CPU 3 first. \( p_1 \) is the probability that task 2 finishes before task 3. The branch of the decision tree below \( p_1 \) shows that task 4 got CPU 3 first and is executing. The task graph has an additional precedence constraint added to show that task 5 is now a successor of both task 2 and task 4. The other branch of the tree shows what happens if task 3 finishes before task 2.
2.5. Construction of Sequencing Trees

Our technique uses the three methods presented in the previous sections to predict the performance of a parallel system. A task graph is used to represent the various tasks and precedence constraints in a parallel algorithm. These tasks require the use of specific
resources of a parallel system, and the scheduling policies of the resources are known. Simple random variables approximate task execution time random variables. Using the information in the task system (task graph, task execution times, resources assignments, and scheduling policies), our technique constructs a sequencing tree that represents the sequence of events that can occur when the task system is executed. Using this tree, the technique can predict the mean and standard deviation of a task system’s execution time. The next chapter describes our technique in detail.
CHAPTER THREE
Sequencing Trees

3.1. Introduction

Simulation can be used to predict the execution times of task systems. However, when the execution times of tasks are non-deterministic, simulation is more difficult. A single simulation corresponds to one particular sequence of events that can occur due to the non-deterministic finishing order of tasks, and this sequence may be far from typical. A number of independent executions of the simulation model must be performed to obtain results that can be accepted with a given degree of confidence. This can be very inefficient.

The technique we present in this chapter uses the ideas presented in Chapter 2 to construct sequencing trees representing the sequence of events that may occur when non-deterministic task systems execute. The technique uses the sequencing trees to predict the execution time of non-deterministic task execution time systems. Our technique has several points in common with simulation. We describe a simulation of a deterministic task execution time system in Section 3.2. Expanding on the terminology of the example simulation described in Section 3.2, Section 3.3 describes our analytical technique for predicting the performance of non-deterministic task execution time systems.

3.2. Simulation of Deterministic Task Execution Time Systems

Simulation of deterministic task execution time systems can be accomplished using a global simulation time variable. At time zero, the simulator would determine which tasks can start execution, and these tasks are placed on the resources they require. The simulator then determines which task or tasks will finish first based on their task execution times. The finishing tasks are removed from their resources and the global simulation time variable
is incremented by the value of the finishing tasks' execution times. The simulator subtracts the incremented amount of time from the task execution times of the tasks still in execution. Successors of the finished tasks are checked to determine if they are eligible to begin execution. Eligible tasks start on the appropriate resources. The procedure is continued until all tasks have completed.

Figure 3.1 shows an example task system. The number $i$ inside a node is the task index of the task $t_i$ represented by the node. The names to the left of the nodes are the resources the tasks require. All the resources have FCFS scheduling policies.

![Figure 3.1: Example task system model](image-url)
If all the task execution times are constant, a simulation of the execution of the task system could be implemented in the following manner. A *ready list* (RL) contains all tasks whose predecessors have finished execution. A global time variable keeps track of the simulation time at any given point in the simulation. Suppose the following list gives the execution times of the tasks.

| t1 | 3 seconds |
| t2 | 2 seconds |
| t3 | 2 seconds |
| t4 | 4 seconds |
| t5 | 4 seconds |
| t6 | 4 seconds |
| t7 | 1 second  |
| t8 | 1 second  |
| t9 | 3 seconds |

In Figure 3.2, the rectangles represent the resources in the system. If a rectangle is empty, the resource is idle. If there is a task name in the rectangle, the resource is providing service to that task. The time under the task is the remaining amount of service the task needs to complete service. Figure 3.2 shows the initial steps in a simulation of the task system.

If the tasks in Figure 3.1 have non-deterministic task execution times the prediction of the task system's execution time by simulation becomes more complex. Each time the simulator determines that a task will acquire service from a resource, it must generate a random number from some distribution associated with the task that represents the amount of service the task requires during this execution. The simulation would continue as in the example shown in Figure 3.2 except that the task execution times are determined from distributions. Depending on the values of the task execution times, the sequence of events (i.e., the order in which tasks acquire resources) can change. To account for these different sequence of events, many simulations of executions of the task system must be made. The number of simulations can be quite large and depends on the required confidence level and confidence interval width. The analytical technique presented in the next section tries to in-
crease efficiency by eliminating the need for repeated executions by representing the different sequences of events as branches of a sequencing tree.

**Figure 3.2:** Simulation of a deterministic task execution time system
3.3. Efficient Implementation of Sequencing Trees

Instead of generating values for task service requirements from the distributions of task execution time random variables, our technique bases its calculations on the distributions of the random variables themselves. At a given point in the execution of a task graph, the technique computes distributions for random variables that represent the finish times of all tasks currently receiving service from resources. Since these finish times are non-deterministic, some mechanism must be provided to account for different sequences of events that can result from the non-deterministic finishing order of tasks. Sequencing trees account for these different sequences of events.

Each node of a sequencing tree holds the state of the task system for a given sequence of events. This includes the tasks in execution on different resources. Because the sequence of events can change depending on which task finishes first, the technique makes a sequencing decision whose outcomes represent the different sequences of events that occur when each task in execution finishes first. For each outcome node, the technique adjusts the distributions of the random variables representing the finish time of the finishing task and the task execution times of the executing tasks. This adjustment reflects the knowledge that one task finished before the others. The following sections describe two sequencing tree construction algorithms. The first algorithm is the basic method for sequencing tree construction. The second algorithm includes improvements made to increase efficiency by decreasing the size of the sequencing tree.

3.3.1. Basic Sequencing Tree Construction Algorithm

A task may be in one of three task states: UNEXECUTED, EXECUTING, or COMPLETED. A task in the UNEXECUTED state has not received service from the resource it requires. A task in the EXECUTING state is currently receiving service from its resource. A COMPLETED task has received all the service it needs from the resource it re-
quires. A resource may be in one of two states. If a resource is *IDLE*, it is not providing service to any task. A *RUNNING* resource is currently servicing a task. The *ready list* (RL) is a list of tasks whose predecessors are all in the *COMPLETED* state.

The random variables representing the start and finish times of a task \( t_i \) are referred to as \( ST_i \) and \( FT_i \), respectively. \( ST_i \) is equal to \( FT_j \) where \( t_j \) is the predecessor task that just entered the *COMPLETED* state and allowed \( t_i \) to enter the RL. If \( t_i \) is an initial task, \( ST_i = 0 \). \( FT_i \)'s distribution is equal to the convolution of the distributions of \( ST_i \) and \( T_i \), where \( T_i \) is the random variable for \( t_i \)'s task execution time.

Each node of a sequencing tree keeps track of the states of the tasks and the resources. Each also has its own RL. As each node is created, it is placed on a *node list*. Figure 3.3 contains a brief description of the algorithm that operates on the nodes of a node list.
Create initial node of sequencing tree.
{
    Initialize all resources to IDLE.
    Initialize all tasks to UNEXECUTED.
    Place all initial tasks (tasks that have no predecessors) on the RL.
    Place initial node on empty node list.
}
While (the node list is not empty)
{
    If (node's RL is empty)
    {
        Node is a terminal node with one EXECUTING task \( t_i \). The execution time of the task system provided that the sequence of events leading to this node occurs is \( FT_i \). Make a completion record consisting of \( FT_i \) and the probability that this sequence of events occurs.
    }
    Else
    {
        1. For each IDLE resource that is required by tasks on the RL, remove one task from the RL and place it on that resource. If more than one task require the use of the resource, make a scheduling decision based on the resource's scheduling policy to determine which task gets the resource and is removed from the RL.
        2. Change the state of each task placed on a resource from UNEXECUTED to EXECUTING. Change the state of each resource given a task from IDLE to RUNNING.
        3. Determine the probability of each RUNNING resource finishing service to its task first, and generate nodes on the sequencing tree to represent each of these outcomes.
        4. For each outcome generated in step 3, place the finishing task \( t_i \) in the COMPLETED state, place its resource in the IDLE state, and calculate \( FT_i \). Adjust the finish time distributions of the COMPLETED task and EXECUTING tasks to reflect the knowledge that one task finished. Update the RL of each outcome node.
        5. Place outcome nodes on the node list.
    }
}
Calculate moments of the task systems execution time from completion records.

Figure 3.3: Basic algorithm outline
When nodes are created as the result of sequencing decisions, they are added to the beginning of the node list, resulting in a depth-first construction of the sequencing. Depth-first construction requires less memory than breadth-first when implementing the algorithm as a computer program. Figure 3.4 gives a flowchart of the algorithm described in Figure 3.3.

![Flowchart](image)

**Figure 3.4:** Basic algorithm flowchart

A sequencing tree node that does not generate a sequencing decision is a *terminal node*. Terminal nodes occur when the RL is empty, there is only one RUNNING resource, and
the task \( t_i \) it is executing has no successors. Since \( t_i \) is the last uncompleted task, \( FT_i \) is the time it would take the task system to execute if the sequence of events leading to this node of the sequencing tree occurred. \( FT_i \) is the terminal node completion time \( TNCT_i \). The probability that a sequence of events leading to a given node \( j \) occurs is the node path probability \( NPP_j \). This probability is equal to the product of the branch probabilities of all branches in a sequencing tree leading from the initial node to the node \( j \) (again, assuming independence of all random variables). The Eq. 3.1 gives the expected execution time \( T \) of the execution of a task system, where \( n \) is the number of terminal nodes in the sequencing tree. \( E[TNCT_j] \) is the expectation of \( TNCT_j \).

\[
E[T] = \sum_{j=1}^{n} NPP_j E[TNCT_j]
\]  

(3.1)

The following is a partial example of how the algorithm would operate on the task graph shown in Figure 3.1 in the case of non-deterministic task execution times. As before, the rectangles represent resources. If a rectangle is empty, the resource it represents is idle. If a rectangle has a task name in it, the resource it represents is providing service to that task. Figure 3.5 shows the simulation up to the first sequencing decision. This decision represents the events of \( t_2 \) finishing before \( t_4 \) and vice versa. The outcome of this sequencing decision results in the algorithm creating two new nodes of the sequencing tree. The distribution of \( FT_i \) of the finishing task \( t_i \) in each outcome node is adjusted to reflect the knowledge that it finished before other executing tasks. New finish time distributions are also generated for the tasks that are still executing to reflect this knowledge. As the flowchart in Figure 3.4 shows, the algorithm is then applied to each of these nodes. Eventually, all branches of the sequencing tree will lead to nodes where the only task executing is \( t_9 \). These are the terminal nodes of the sequencing tree. \( FT_9 \) in each terminal node \( j \) will be \( TNCT_j \).
Figure 3.5: Example of basic algorithm

The algorithm described in this section assumes that the probability of two tasks finishing simultaneously is zero. If the distributions of the $FT$s are continuous, this as-
assumption is true. When using simple random variables, there may be a non-zero probability of two tasks finishing simultaneously. This probability is divided equally and added to the probabilities of one task finishing before the other. For example, suppose we have two tasks \( t_1 \) and \( t_2 \). The probability that \( t_1 \) and \( t_2 \) finish at the same time, \( \Pr(FT_1 = FT_2) \), is divided by two and added to the probability of \( \Pr(FT_1 < FT_2) \) and \( \Pr(FT_2 < FT_1) \).

### 3.3.2. Trading Accuracy for Efficiency

As stated earlier in Section 2.4, sequencing trees can grow exponentially as larger task systems are studied. To curb exponential growth, the algorithm allows the user to trade accuracy for efficiency by basing predictions on a partial construction of a sequencing tree. When inputting a task system specification to the algorithm, the user may specify a cumulative terminal node probability level with or without a terminal node limit. The algorithm will generate a sequencing tree until the sum of the path probabilities of the terminal nodes created exceed the cumulative terminal node probability level, or the number of terminal nodes created exceed the terminal node limit.

As new nodes of the sequencing tree are created, each outcome node has a probability associated with it that represents the probability that it is the outcome of a sequencing decision. At each sequencing decision, the algorithm always expands the sequencing tree below the outcome nodes with the higher probabilities first. Thus, the algorithm always constructs the sequencing tree in a depth-first manner. By expanding the higher probability branches first, the algorithm tries to generate the terminal nodes with the highest probabilities first.

When the cumulative terminal node probability level or terminal node limit is reached, the algorithm predicts the execution time of the task system based on the terminal nodes already created. For example, if the user inputs a cumulative terminal node probability level of 50%, a sequencing tree is constructed until the sum of the terminal node path
probabilities exceed 50%. The prediction of execution time of the task system returned by
the algorithm is qualified by the fact that only a subset of the possible sequences of events
is represented in the sequencing tree. The depth-first construction method attempts to make
sure the higher probability sequences of events are represented in the predictions returned
by the algorithm. However, there is no guaranty that the terminal nodes not generated by
the algorithm will have node path probabilities less that all the generated terminal nodes. In
effect, the search algorithm is a greedy algorithm which suffers from the usual problems of
this class of algorithms. The algorithm allows the user to adjust the tradeoff between accu-
curacy and efficiency by adjusting the cumulative terminal node probability level and terminal
node limit. The next chapter presents the results of experiments that examine the effect on
accuracy when these factors are varied.

3.3.3. Improved Sequencing Tree Construction Algorithm

The basic algorithm presented in the previous section has a potentially severe prob-
lem with efficiency. As the number of resources and the number of tasks using them con-
currently in the system grow, the possible number of outcomes of each sequencing decision
grows. Thus, the size of sequencing trees can grow exponentially as the number of re-
sources increase, depending on the structure of the task graph. For example, if four tasks
are using four resources concurrently, the basic algorithm creates four outcomes for a se-
quencing decision determining which task finishes first. If no other tasks start executing,
each of these outcome nodes will in turn create three outcome nodes as a result of sequenc-
ing decisions. This section describes an improved algorithm that improves constructing the
sequencing tree without loss of accuracy compared to the basic algorithm.

The algorithm improves efficiency by eliminating unnecessary sequencing decisions
leading to redundant nodes. Redundant means that they eventually lead to the same se-
quence of events. In the basic algorithm, we discovered that many of the outcomes of se-
quencing decisions eventually became redundant. For example, suppose we have a task system where two tasks are currently executing on two resources. If none of these tasks' descendants use the same resources, no more contention will occur. The basic algorithm generates outcome nodes from sequencing decisions for all tasks that are in the EXECUTING state at the same time. This increases the size of the sequencing tree with nodes that do not resolve any contention.

Furthermore, the basic algorithm does not allow task start time distributions to be computed from the maximum of task finish time distributions. For example, suppose tasks \( t_a \) and \( t_b \) are running on two separate resources, and \( t_c \) is the only successor of both \( t_a \) and \( t_b \). In the basic algorithm, two sequencing outcomes would be generated before \( ST_c \) would be computed. These outcomes would represent the event that \( t_a \) finished before \( t_b \) and vice versa. \( ST_c \) would be \( FT_b \) given \( t_a \) finished before \( t_b \), or \( FT_a \) given \( t_b \) finished before \( t_a \). The improved algorithm allows the \( ST_c \) to equal the maximum of \( FT_a \) and \( FT_b \), and no decision is required. The next section outlines the improved algorithm.

3.3.3.1. Improved Algorithm Outline

Several terms used in the following description are now defined. Unless otherwise redefined, the terms used in the description of the basic algorithm remain the same. First, a waiting list (WL) is a list of tasks that have at least one predecessor in the EXECUTING state, and all other predecessors are either in the EXECUTING or COMPLETED states. The resource set of a task is the set of all resources the task and its descendants require.

We define the following functions:

\[
\begin{align*}
\text{RS}(t_i) &= \text{resource set of task } t_i \\
\text{CONTENTION}(t_i, t_j) &= \text{returns TRUE if possible contention exists between the descendants of } t_i \text{ and } t_j
\end{align*}
\]
The improved algorithm tries to eliminate unnecessary sequencing decisions by repeatedly applying a four-phase loop to a task system. The loop is repeated until a sequencing decision is required to resolve any potential resource contention. The four phases are the start phase, the bookkeeping-after-start phase, the finish phase, and the bookkeeping-after-finish phase. The rules and actions of the different phases are only stated in this section. Explanations of the rules and actions are presented in Section 3.3.3.2.

The following is the list of start rules. The algorithm applies these rules during each start phase.

1. For each idle resource that is required by a task on the RL, remove one task and place it on that resource. If more than one task require the use of the resource, a scheduling decision, based on the resource's scheduling policy, is made to determine which task will start on the resource. The task that gets the resource starts. The other tasks are moved back to the WL.

2. The $ST$ of a task is equal to $\max(FT_1, ..., FT_k)$, where $t_i$ through $t_k$ are its predecessor tasks.

These are the actions the algorithm takes during each bookkeeping-after-start phase.

1. Update the WL.

2. Add any resource contention constraints (RCC).

When a task $t_i$ on the WL needs a resource that is currently executing a task $t_j$ that is not its predecessor, the algorithm places a resource contention constraint (RCC) between $t_i$ and $t_j$. An RCC is treated exactly like a precedence constraint, making $t_i$ a successor of $t_j$. Section 3.3.3.2.4 explains how RCCs remove some unnecessary sequencing decisions.

Next, we list the finish phase rules. Only one of these rules is applied during each finish phase.

1. If there are no tasks on the WL, calculate $FT$'s for all tasks currently in the EXECUTING state. This node of
the sequencing tree is a terminal node, and the terminal node completion time is the maximum of all the FTs just calculated.

2. Ignore all running tasks that have no successors on the WL. Calculate CONTENTION(t_i, t_j), where t_i and t_j are all possible combinations of the tasks in the EXECUTING state with successors, including RCC successors, on the WL. If CONTENTION(t_i, t_j) always returns FALSE, calculate the FTs of the tasks. Remove the tasks from their resources, placing the resources in the IDLE state and the tasks in the COMPLETED state.

3. Ignore all running tasks that have no successors on the WL. Calculate CONTENTION(t_i, t_j), where t_i and t_j are all possible combinations of the tasks in the EXECUTING state with successors, including RCC successors, on the WL. For each case in which CONTENTION(t_i, t_j) returns TRUE, add t_i and t_j to a list of tasks whose descendants can possibly contend for resources (possible contention list). Make sequencing decision between t_i through t_m, where t_i through t_m are the tasks in the possible contention list. Each outcome of the sequencing decision creates a new node of the sequencing tree that represents the event that each of the tasks t_i through t_m finishes before the others.

During the bookkeeping-after-finish phase the following action is taken.

1. Update RL.

In the next section we justify the rules and actions stated above with examples. The improved algorithm is outlined in Figure 3.6. Each node of a sequencing tree keeps track of the states of the tasks and the resources. Each node also has its own RL and WL. As with the basic algorithm, a node list is employed. As nodes are created, they are added to the head of the list so that the sequencing tree is created depth-first. Figure 3.7 gives a flowchart of the algorithm.
Create initial node of sequencing tree

[ Initialize RL with all tasks that have no predecessors.

Initialize all task states to UNEXECUTED.

Initialize all resource states to IDLE.

Place node on node list
]

While (the node list is not empty)
{
  Set variable FLAG to TRUE
  While (FLAG is TRUE)
  {
    1. Apply start phase rules.
    
    2. Apply bookkeeping-after-start phase actions.
    
    3. Apply appropriate finish phase rule.
      
    a. If rule 1 was used, mark sequencing tree node as terminal node and set FLAG to FALSE.
      
    b. If rule 3 was used, mark sequencing tree node as sequencing decision node and set FLAG to FALSE.
      
    4. Apply bookkeeping-after-finish phase actions.
  }
  If (node is a terminal node)
  {
    Calculate its path probability and completion time, and place in completion record.
  }
  Else
  {
    Make sequencing decision and generate outcome nodes. Finish task appropriate to each outcome and apply bookkeeping-after-finish phase actions to each outcome node.
    
    Place outcome nodes on node list
  }
}

Calculate results from completion records

Figure 3.6: Improved algorithm outline
Figure 3.7: Improved algorithm flowchart
3.3.3.2. **Explanation of Rules and Actions**

This section explains the actions taken by the algorithm. A detailed example of the entire algorithm follows in a later section.

3.3.3.2.1. **Start Rule 1**

1. For each idle resource that is required by a task on the RL, remove one task and place it on that resource. If more than one task require the use of the resource, a scheduling decision, based on the resource's scheduling policy, is made to determine which task will start on the resource. The task that gets the resource starts. The other tasks are moved back to the WL.

If a task is on the RL, the algorithm guarantees that all its predecessors' FTs have been computed and that the resource the task requires is IDLE. These predecessors include predecessors added due to RCCs. If no other task on the RL requires the same resource as a task, that task starts executing on the resource and no scheduling decision is required. If more than one task on the RL need the same resource, the algorithm has determined that these tasks entered the RL at the same time (they may not have entered the WL at the same time). If a scheduling decision is required, the scheduling policy of the resource decides which task acquires the resource first. The technique can support several scheduling policies including first-come-first-served, last-come-first-served, random, and priority. The technique cannot model processor sharing, preemptive resume, or time-sliced schedules. The algorithm moves the tasks that were denied a resource as the result of a scheduling decision back to the WL. These tasks become successors of the task that got the resource as the result of RCCs added during the bookkeeping after start phase. This action also guarantees that the RL will be empty after each start phase.
3.3.3.2.2. Start Rule 2

2. The $ST$ of a task is equal to $\max(FT_1, \ldots, FT_k)$, where $t_1$ through $t_k$ are its predecessor tasks.

Finish Rule 2, to be described later, allows more than one task to finish (i.e., tasks changed to COMPLETED state and $FT$s calculated) without making a sequencing decision determining their finish order and creating new nodes of the sequencing tree. A task $t_i$ that starts because of Start Rule 1 has a $ST_i$ that depends on the $FT$s of its predecessors. For example, Figure 3.8 shows a three-task task graph that requires the use of two resources.

![Figure 3.8: Example task graph for Start Rule 2](image)

Because of Finish Rule 2, $t_a$ and $t_b$ will finish during the same finish phase. The algorithm will calculate $ST_c$ from $FT_a$ and $FT_b$, and no sequencing decision is made. The basic algorithm required a sequencing decision to determine the sequence of events if $t_a$ finished before $t_b$ and vice versa.

The $ST_i$ of a task $t_i$ is equal to the maximum of the $FT$s of its predecessors $t_1$ through $t_k$ as shown in Eq. 3.2.

$$ST_i = \max(FT_1, \ldots, FT_k)$$

$$F_i(x) = \prod_{j=1}^{k} F_j(x) \quad (3.2)$$
F_j(x) is the distribution of the random variable FT_j. During the construction of the sequencing tree, the finishing order of some of the predecessor tasks t_1 - t_k may be determined as the results of sequencing decisions. The algorithm uses this information to remove some of the predecessor tasks' FTs from the calculation in Eq. 3.2. For example, suppose a task t_d has three predecessors t_a, t_b, and t_c. Suppose that at some point in the construction of the sequencing tree the finish order of t_b and t_c is determined sequencing decision. In one outcome of the sequencing decision, t_b will finish before t_c. The calculation of ST_d will involved only FT_d and FT_c because the algorithm knows that FT_c > FT_b. Determining which FTs may be removed from Eq. 3.2 is facilitated by the use of a finish list (FL). The FL is a list of all tasks that completed execution in the last finish phase. Only tasks in the FL have their FTs used in Eq. 3.2. A predecessor task that is not on the FL when the ST of one of its successors is calculated is guaranteed to finish before at least one of the other predecessors.

3.3.3.2.3. Bookkeeping-After-Start Phase Action 1

1. Update the WL.

After new tasks start during the start phase, the algorithm checks the successors of all the started tasks. If all the predecessors of these tasks are either in the EXECUTING or the COMPLETED state, the algorithm places the successor tasks on the WL. The WL facilitates the placement of RCCs.

3.3.3.2.4. Bookkeeping-After-Start Phase Action 2

2. Add any resource contention constraints (RCC).
RCCs eliminate sequencing decisions required when a task on the WL requires a resource that is currently running a task. Suppose we have the following task graph.

![Diagram](image)

**Figure 3.9:** Example task graph for Bookkeeping-After-Start Phase Action 2

During the first start phase, the algorithm would start $t_a$ and $t_b$. The first action of the bookkeeping-after-start phase would place $t_c$ on the WL. If no RCC is added, the algorithm would have to make a sequencing decision to determine whether task $t_a$ or $t_b$ finished first. In one outcome, $t_a$ would finish first, and $ST_c$ will equal $FT_b$. In the other outcome, $t_b$ would finish first. Although $t_c$'s predecessor has completed, $t_c$ may not start while the resource it requires is still in the RUNNING state. $t_c$ will start after $t_a$ completes execution on CPU_1. $ST_c$ will equal $FT_a$.

Because of an RCC between $t_a$ and $t_c$, the improved algorithm does not need to make a sequencing decision. The algorithm treats the RCC as a precedence constraint, making $t_c$ a successor of $t_a$. Using Finish Rule 2, the algorithm will finish $t_a$ and $t_b$ in the same finish phase. $ST_c$ is then calculated in the next start phase according to Start Rule 2. $ST_c$ is equal to the maximum of $FT_a$ and $FT_b$.

3.3.3.2.5. **Finish Rule 1**

1. If there are no tasks on the WL, calculate $FT$s for all tasks currently in the EXECUTING state. This node of the sequencing tree is a terminal node, and the terminal
node completion time is the maximum of all the $FT$s just calculated.

If there are no tasks on the WL, then all tasks currently executing are final tasks (tasks that have no successors), and the algorithm calculates the $FT$s of these tasks. The distribution of a task $t_i$'s $FT_i$ is equal to the convolution of the distribution of $ST_i$ and the distribution of the task execution time $T_i$. The execution time of an entire task system is equal to the maximum of the $FT$s of all the final tasks, provided that the sequence of events leading to this node of the sequencing tree occur. The expected completion time of a task system is calculated according to Eq. 3.1. The basic algorithm does not have the ability to take the maximum of distributions. Thus, sequencing decisions are required to determine the finishing order of the final tasks. This finish rule has the potential for removing many sequencing decisions from a sequencing tree. The number of sequencing decisions removed, if any, depends on the structure of the task graph and the number of resources in the system.

3.3.3.2.6. Finish Rule 2

2. Ignore all running tasks that have no successors on the WL. Calculate $CONTENTION(t_i, t_j)$, where $t_i$ and $t_j$ are all possible combinations of the tasks in the EXECUTING state with successors, including RCC successors, on the WL. If $CONTENTION(t_i, t_j)$ always returns FALSE, calculate the $FT$s of the tasks. Remove the tasks from their resources, placing the resources in the IDLE state and the tasks in the COMPLETED state.

If a task has no successors on the WL, then it either has no successors or its successors are also descendants of other tasks in the EXECUTING state. The algorithm can ignore this task while calculating which tasks may influence the sequence of events depending on their finish order. The result of the task finishing will not add any tasks to the WL, or move any tasks from the WL to the RL. The algorithm postpones changing the task to the
COMPLETED state and its resource to the IDLE state. The postponement allows RCCs to be established between the task and other tasks that require the same resource. From the remaining executing tasks, the algorithm determines if it needs to know the finish order of these tasks. The finish order is not important if there is no possibility of contention between the descendants of the EXECUTING tasks. The algorithm calls CONTENTION(t_i, t_j) to determine if there is any possibility of contention between the descendants of t_i and t_j. Figure 3.10 gives an outline of the CONTENTION function. If CONTENTION(t_i, t_j) returns FALSE for all possible combinations of t_i and t_j, the finish order of the EXECUTING tasks does not matter. The algorithm computes the FTs of the EXECUTING tasks and moves them to the COMPLETED state.

As an example, we look at the task graph shown in Figure 3.8. During the first start phase, t_a and t_b start. t_c is placed on the WL during the following bookkeeping-after-start phase. CONTENTION(t_a, t_b) returns FALSE because t_a and t_b have no non-common descendants. Thus, t_a and t_b may finish because ST_c will be calculated from both FT_a and FT_b. The task graph shown in Figure 3.9 would have exactly the same result. During the bookkeeping-after-start phase, an RCC would be added between t_a and t_c. The function CONTENTION(t_a, t_b) returns FALSE, and both t_a and t_b are finished.
CONTENTION(tᵢ, tⱼ)
{
    If ( \( RS(tᵢ) \cap RS(tⱼ) = \emptyset \) )
    {
        No descendants of tasks \( tᵢ \) and \( tⱼ \) use the same resources.
        There is no possibility of contention.
        Return FALSE.
    }

    Get set of non-common descendants of \( tᵢ \) and \( tⱼ \).
    Get set of common descendants of \( tᵢ \) and \( tⱼ \).
    If ( any of the non-common descendants use any of the resources
        the common descendants use )
        Possible contention exists between the descendants of
        tasks \( tᵢ \) and \( tⱼ \). Finish order of tasks is needed.
        Return TRUE.
    }

    If ( \( tᵢ \) or \( tⱼ \) has no descendants in the non-common set )
    {
        Finish order is not needed because all of one task's de-
        scendants are also descendants of the other task.
        Return FALSE.
    }

    If ( any of \( tⱼ \)'s non-common descendants use any of the same re-
        sources of task \( tᵢ \)'s non-common descendants )
        Possible contention exists between non-common descen-
        dants. Finish order of \( tᵢ \) and \( tⱼ \) is needed.
        Return TRUE.
    }
    Return FALSE
}

Figure 3.10: CONTENTION algorithm outline
3.3.3.2.7. **Finish Rule 3**

3. Ignore all running tasks that have no successors on the WL. Calculate CONTENTION(t_i, t_j), where t_i and t_j are all possible combinations of the tasks in the EXECUTING state with successors, including RCC successors, on the WL. For each case in which CONTENTION(t_i, t_j) returns TRUE, add t_i and t_j to a list of tasks whose descendants can possibly contend for resources (possible contention list). Make sequencing decision between t_i through t_m, where t_i through t_m are the tasks in the possible contention list. Each outcome of the sequencing decision creates a new node of the sequencing tree that represents the event that each of the tasks t_i through t_m finishes before the others.

Because Finish Rules 1 and 2 did not apply, the algorithm determines that there is a possibility for contention between the descendants of executing tasks. As explained in the previous section, the algorithm can ignore all running tasks that have no successors on the WL. When the algorithm tried to apply Finish Rule 2 at least one instance of CONTENTION(t_i, t_j) returned TRUE and thus, Finish Rule 3 is applied. For example, suppose we have the task graph shown in Figure 3.11.

![Diagram](image)

**Figure 3.11:** Example for Finish Rule 3

Suppose t_a and t_b are in the EXECUTING state and the algorithm is in the finish phase. The common descendant of t_a and t_b is t_c. Their non-common descendants are t_e.
and \( t_d \). Since \( t_c \) and \( t_d \) use CPU_3, the order in which they acquire the resource depends on the finishing order of \( t_a \) and \( t_b \). Thus, \( \text{CONTENTION}(t_a, t_b) \) returns TRUE and the algorithm makes a sequencing decision between \( t_a \) and \( t_b \) based on their \( FTs \). Two outcome nodes are generated representing the events of \( t_d \) finishing before \( t_b \) and \( t_b \) finishing before \( t_a \). For each outcome node, the distributions of \( FT_a \) and \( FT_b \) are adjusted to reflect the knowledge of the finish order.

3.3.3.2.8. Bookkeeping-After-Finish Phase Action

1. Update RL.

After the finish phase, the algorithm checks all the successors on the WL of the tasks that finished. Those tasks whose predecessors are all in the COMPLETED state are moved from the WL to the RL.

3.3.3.3. Example of Improved Algorithm

The following is a partial example of how the improved algorithm would predict the execution time of the task system shown in Figure 3.1. The example uses the same symbols as the example of the basic algorithm of Figure 3.5. The example also has a WL and an FL for each sequencing tree node. The example follows the algorithm's operation on the task system of Figure 3.1 to one terminal node. When a sequencing decision is reached, the example only shows one branch of the decision.
RL: $t_1$  
WL:  
CPU_1 | CPU_2 | CPU_3 | BUS  
--- | --- | --- | ---  
--- | --- | --- | ---  
t_1 starts (apply bookkeeping after start phase actions).

RL:  
WL: $t_2, t_3, t_4$  
CPU_1 | CPU_2 | CPU_3 | BUS  
--- | --- | --- | ---  
--- | --- | --- | ---  
t_4 finishes according to Finish Rule 2 (apply bookkeeping after finish phase actions).

RL: $t_2, t_3, t_4$  
WL:  
CPU_1 | CPU_2 | CPU_3 | BUS  
--- | --- | --- | ---  
--- | --- | --- | ---  
Scheduling decision made between $t_2$ and $t_3$ in $t_2$'s favor. $ST_2, ST_3 = FT_1$.

RL: $t_3, t_4$  
WL:  
CPU_1 | CPU_2 | CPU_3 | BUS  
--- | --- | --- | ---  
--- | --- | --- | ---  
t_4 is ignored because it has no successors on WL. $t_2$ finishes according to Finish Rule 2.

RL: $t_3, t_5$  
WL:  
CPU_1 | CPU_2 | CPU_3 | BUS  
--- | --- | --- | ---  
--- | --- | --- | ---  
t_5 and $t_5$ start. $ST_3, ST_5 = FT_2$.

RL: $t_6, t_7$  
WL:  
CPU_1 | CPU_2 | CPU_3 | BUS  
--- | --- | --- | ---  
--- | --- | --- | ---  
t_6 is again ignored. CONTENTION($t_6, t_7$) returns TRUE and a sequencing decision is required according to Finish Rule 3.

Figure 3.12: Improved algorithm example
Figure 3.12: Improved algorithm example (cont.)
If the sequence of events shown in the above example occurs, the execution time of the task graph is \( FT_9 \). The expected execution time of the task system is computed by Eq. 3.1.

3.3.4. Simple Random Variables and the Sequencing Tree Construction Algorithms

The sequencing tree construction algorithms presented earlier work with random variables instead of constants like the simulation technique presented in Section 3.2. Because of this, the algorithms need some mechanism to convolve and multiply the distributions of random variables. We use simple random variables to perform this task since they are closed over the operations of convolution and multiplication. Furthermore, Chapter 2 showed how simple random variables can approximate other random variables with arbitrary non-negative distributions.

As stated in Section 2.3.4, when the results of convolutions and multiplications have numbers of values that degrade efficiency, simple random variables with fewer numbers of
values are used to approximate these random variables. However, to improve efficiency, it was decided to store the distributions of all random variables as means, variances, skews, and, in the case of bounded distributions, minimums and maximums. When it is necessary to add distributions, the algorithm simply adds the means, variances, and skews of these distributions and does not convolve step distribution approximations. When bounded distributions are added, the minimum and maximum of the result are the sums of minimums and maximums of the operands, respectively. When multiplying distributions, the algorithm multiplies step distribution approximations determined from the central moments and the bounds. The mean, variance, skew, and the minimum and maximum values of the resulting step distribution are then stored as the distribution of the result.

3.3.5. Dependence of Random Variables

To be able to convolve or multiply distributions, the involved distributions are assumed to be independent. When calculating the probability of one task finishing before another, their FTs must also be independent. The algorithm only adds a task's ST and its task execution time T. These random variables are always assumed to be independent. However, when calculating distributions of STs by multiplying the distributions of the FTs of predecessor tasks, or finding the probability of one task finishing before another, the involved distributions may not be independent.

Mak and Lundstrom [22, 23] subtract the FT of the common ancestor chain from the FTs of tasks before performing calculations that assume independence. Figure 3.13, shows the common ancestor chain of \( t_b \) and \( t_c \). In Figure 3.13, \( FT_b = T_b + T_x + FT_a \) and \( FT_c = T_c + FT_a \). \( FT_a \) is subtracted from \( FT_b \) and \( FT_c \) before performing operations that assume independence. It should be noted that after Mak and Lundstrom perform this subtraction, the residence times (the time a task spends queued at a resource and getting service) of \( t_b \) and \( t_c \) are assumed to be independent. This is not really the case if \( t_b \) and \( t_c \) use the
same resource, because the residence times of \( t_b \) and \( t_c \) depend on each others' service demands as determined by queueing network models. Also, recall that the Mak-Lundstrom technique applies only to series-parallel task graphs.

Our technique does not require that task graphs be series-parallel. In fact, because of RCCs added by the algorithm, series-parallel task graphs may become non-series-parallel during the course of the prediction. The algorithm still needs to remove the dependence of two random variables before performing the operations mentioned earlier. The algorithm removes as much dependence as possible by subtracting the \( FT \) of the common series ancestor chain of two tasks before performing any operations that involve their \( FT \)s.

Figure 3.14 gives an example common series ancestor chain. In this example, \( FT_b = T_b + FT_x = T_b + T_x + FT_a \) and \( FT_c = T_c + \max(FT_x, FT_y) = T_c + \max(T_x, T_y) + FT_a \). \( FT_a \) is subtracted from \( FT_b \) and \( FT_c \) before performing operations that assume independence. As can be seen, both \( FT_b \) and \( FT_c \) are dependent on \( FT_x \). However, \( FT_x \) cannot be subtracted
from $FT_c$. $ST_c$ was calculated by taking the maximum of the $FT_x$ and $FT_y$, and thus $FT_x$ cannot be factored out by subtraction. $t_a$ is the closest common ancestor of both $t_b$ and $t_c$ whose dependence can be factored out by subtraction. $t_a$ is called a series ancestor of $t_b$ and $t_c$ because its contribution to both tasks $FT_i$s is the result of a series combination. $FT_a$ is the $FT$ of $t_b$ and $t_c$'s common series ancestor chain. After the subtraction of $FT_a$, $FT_b$ and $FT_c$ are assumed to be independent.

In any analytical performance prediction technique, certain assumptions must be made that can influence accuracy. In this case, we assume total independence when in fact dependence exists. We feel that this assumption will have less effect on our results than the assumptions that other techniques have made to avoid this problem (i.e., ignoring dependence, ignoring contention, limiting task graphs to series-parallel structure). We feel that this assumption is justified based on the results presented in Chapter 4.
3.4. Summary

This chapter presented the details of our performance prediction technique. The technique uses the methods of Chapter 2 to predict the performance of parallel programs executing on parallel systems. Explicit precedence constraints, resource contention, and a variety of resource scheduling policies are modeled. Two algorithms implementing the technique were presented. The second algorithm improves the efficiency of the first algorithm by removing sequencing decisions whose outcomes did not represent different sequences of events. A program called ES has been written that utilizes the improved algorithm. Chapter 4 presents results of tests that validate the accuracy of ES. Results for the efficiency of the program are also presented. Finally, examples of applications for the prediction technique are given.
CHAPTER FOUR
Accuracy, Efficiency, and Applications of Prediction Technique

4.1. Introduction

The improved algorithm described in the last chapter has been implemented in a program called ES. The program is written in C++ and C, and executes on a Sun SPARC 2 workstation. This chapter presents the results of experiments testing the accuracy and efficiency of our analytical performance prediction technique. The first section describes experiments comparing ES to discrete-event simulations. The comparisons were made for two task graph structures, one series-parallel and the other non-series parallel. We evaluate accuracy by determining the error between ES estimates and simulation estimates for the mean and standard deviation of the total execution times of the task systems. Other graphs compare the relative speeds of ES predictions and the simulations. ES predictions are also compared to execution times of algorithms realized on an Intel iPSC/860† hypercube parallel computer. The accuracy and efficiency of ES predictions are then compared to execution-driven simulations of specific parallel algorithms realized on parallel systems with fully connected and mesh connected interconnection networks. Next, the effects of limiting the size of the sequencing tree to improve efficiency are examined. Finally, the last section gives examples of applications for the performance prediction technique.

4.2. Comparisons Against Discrete-Event Simulations

We compared performance predictions obtained by ES against those obtained from an discrete-event simulator. The discrete-event simulator simulates the executions of task systems in which we assign distributions for the execution times of each task. The distribu-

† Use of the Intel iPSC/860 was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement Nos. CCR-8809615 and CDA-8619893 with support from the Keck Foundation.
tions are not derived from any observations of real programs but represent an artificial workload. Each task in the system is also assigned to a resource from which it must receive service. In these simulations, all the resources modeled are FCFS. The discrete-event simulator was written using YACSIM [16]. YACSIM is a simulation language developed by J.R. Jump of Rice University.

Given a task system specification, the simulator calculates an estimate for the mean execution time of the task system. The simulator begins by calculating a time for a single execution of the task system. During the simulation of a single execution, the simulator generates values for the execution times of individual tasks from random number generators. The random number generators generate these values from the distributions of the tasks' execution time random variables. The simulator records the value of the execution time of the task system for this single execution and begins to simulate another execution in which the values of the execution times of the individual tasks are again generated from the random number generators. The simulator calculates the estimate of the mean execution time for the task system by averaging the execution times of the individual simulated executions. The simulator continues simulating executions of the task system until the sample mean satisfies a given confidence level and confidence interval width.

ES has a task system input language used to specify the precedence constraints of tasks, the distributions of the execution times of tasks, and the resources the tasks require. The scheduling policies of the resources are also individually designated. The discrete-event simulator takes task system specifications written in the same input language as ES specifications. To perform a large number of experiments, two programs were written to randomly generate task system specifications that could be input into ES and the discrete-event simulator. These programs generate task graphs with two types of structures. The first program creates task graphs with a partitioning structure [25]. The second program
creates task graphs with a multi-stage linear pipeline structure [21, 22]. Figure 4.1 gives examples of task graphs with these structures.

(a) Five-level partitioning structure  
(b) Four-stage linear pipeline structure

Figure 4.1: Accuracy test task graphs

As can be seen, task graphs with partitioning structures are series-parallel. Task graphs with multi-stage linear pipeline structures are not series-parallel.

Given the number of resources in the task system, the task graph generating programs will randomly assign each of the tasks in a task graph to a resource. Furthermore, the programs randomly assign task execution times with different types of distributions to the tasks. The parameters of these distributions are also randomly determined. The resource and distribution assignments of any task system generated by these programs may not be representative of any real algorithm. However, we feel that these tests present a wide variety of problems to test our technique.

4.2.1. Results of Tests on Partitioning Structured Task Graphs

Figure 4.1 (a) shows a task graph with a five-level partitioning structure. Algorithms with partitioning structures apply a divide-and-conquer method to problem
solving. The task at the beginning of the graph divides the problem into two smaller halves and gives them to the following tasks. These tasks further divide the problem. Eventually, when the portions are the correct size (usually determined by factors such as the number of resources or maximized efficiency), tasks in the work level (i.e., the third level in Figure 4.1 (a)) solve the problem for the smaller portions. The tasks below the work level merge the results of the work level tasks until one solution is obtained. Many classes of algorithms, including many sorting and searching algorithms, have this structure [21].

For the first test, we generated sixty six task systems with the five-level partitioning structure. The task systems had between one and six FCFS resources. Eleven task systems were generated for each of the different numbers of resources. The tasks in these systems were randomly assigned to the resources. Each resource in the system is guaranteed to have a task assigned to it. With equal probability, the generating program randomly assigns each task either a uniformly \( U(t_L, t_U) \) or triangularly \( T(t_L, t_U) \) distributed task execution time. \( t_L \) and \( t_U \) are the upper and lower bounds of the distributions, respectively. Figure 4.2 gives a graphical representation of the p.d.f. of a triangularly distributed random variable \( T \).

![Figure 4.2: p.d.f. of triangular distribution](image)

\( t_U \) and \( t_L \) of each distribution are also randomly assigned. The graph generating program generates the value of \( t_L \) for a specific task execution time distribution from a random number generator with uniform distribution \( U(0.1, 0.5) \). \( t_U \) is generated from \( U(t_L, 0.5) \).
The graphs of Figure 4.3 present comparisons between ES's performance predictions and those determined from the YACSIM based simulator. The YACSIM simulations were run until sample means for the execution times of the task systems were obtained with 99.9% confidence intervals with widths of less than 1% of the sample mean. For these tests, ES generates the complete sequencing trees representing all the possible sequences of events that may occur during the executions of the task systems (i.e., cumulative terminal node probability level = 100%). The values of $t_1$ and $t_3$ of a three-step distribution approximation (Eqs. 2.19 - 2.22) of a given uniform or triangular distribution equal the values of $t_L$ and $t_U$ of the original distribution, respectively.

Eq. 4.1 defines percent error as represented in the graphs.

$$\frac{ES \, estimate - YACSIM \, estimate}{YACSIM \, estimate} \times 100 \quad (4.1)$$

The first graph of Figure 4.3 shows the errors between the ES-estimated mean task system execution times and those determined from YACSIM simulations. Each point on the graph represents the error between ES' estimate of the mean execution time of a task system and the YACSIM simulator's estimate for the same system. These errors are sorted in increasing order. The horizontal axis of the graph serves only to separate the values obtained for each of the sixty six test task systems. Since the errors are sorted in increasing order, a value at a given point on the horizontal axis will not correspond to the values at the same point on the horizontal axes of the standard deviation and speedup graphs. The maximum error for predicted mean obtained was 1.8%.
Figure 4.3: Results for five-level partitioning task graphs with uniformly and triangularly distributed task execution times

The second graph displays the errors between the ES-estimated standard deviation of the task system execution times and those determined from YACSIM simulations. Again, the errors are sorted in increasing order. The estimates for standard deviations were not as accurate as the estimates for means. The maximum relative error of the standard de-
viations was 54.0%. Potential sources of this error are the distribution approximations, violation of the independence assumptions, or because of the approximated distributions ES may assign a higher or lower probability to a sequence of events than its actual probability in the YACSIM simulation.

The third graph shows the speedup of ES over YACSIM for these tests. Speedup is defined as the time a YACSIM simulation requires to generate an estimate with a given confidence level for the mean execution time of a task system, divided by the time it tasks ES to estimate the mean and standard deviation of the execution time of the same task system. The speedups are also sorted in increasing order.

The graphs of Figure 4.3 showed how well ES estimates the execution time of task systems when all the tasks have bounded task execution time distributions. We performed another sixty six tests using five-level partitioning task graph structures. Again, the task systems had between one and six FCFS resources. Eleven task systems were generated for each of the different numbers of resources. This time the tasks had uniformly, triangularly, or exponentially distributed task execution times. The purpose was to see how well ES performed when some of the tasks had unbounded task execution time distributions. Again, the types of distributions and the resource assignments were randomly assigned to tasks. The bounds of the uniform and triangular distributions were assigned as in the earlier tests. The means of the exponential distributions were assigned a value returned from a random number generator with uniform distribution U(0.1, 0.5). Figure 4.4 shows the results from these tests. ES's step distribution approximations of exponential distributions used Eqs. 2.23 - 2.25 with $f = 0$ to obtain values for $t_1$ and $t_3$. Again, the cumulative terminal node probability level was 100%.  
Figure 4.4: Results for five-level partitioning task graphs with uniformly, triangularly, and exponentially distributed task execution times

As before all values are sorted in increasing order. As the first graph shows, on average the errors of ES's estimates for mean task system execution time increased from the previous tests, but were still quite small. The maximum error between an ES estimate of mean exe-
cution time and a YACSIM estimate was 9.4%. The increased error is attributed to modeling exponentially distributed random variables with simple random variables. The maximum standard deviation error was 48.5%. The greatest change was the large increase of speedups of ES over YACSIM. The reason for the increase is the fact that YACSIM must now perform a larger number of runs to find an average mean execution time that satisfies the constraints of the confidence intervals. The unbounded or exponential distributions are responsible for this increase. The confidence intervals we calculated were extremely tight. If the constraints on the confidence intervals were relaxed, the increase of ES speedup over YACSIM would not be as great. Section 4.2.3 presents results of experiments that examine the effects of relaxing the constraints of the YACSIM simulations on the relative speedup of ES.

Direct comparisons between ES and the related techniques presented earlier could not be accomplished. While the task graph structures used were series-parallel as required by Mak-Lundstrom, Thomason-Bay, Sahner-Trivedi, and Madala, other factors prevent these techniques from making accurate predictions. Both Mak-Lundstrom and Thomason-Bay require that all tasks using FCFS resources have exponentially distributed service requirements with the same mean. This is due to the limitations imposed by using product-form queueing network models in their approaches. The tests we performed did not have exponentially distributed task execution times with the same means. Sahner-Trivedi and Madala ignore contention for shared resources.

4.2.2. Results of Tests on Multi-Stage Linear Pipeline Structured Task Graphs

We now present the results of tests involving task systems with a four-stage linear pipeline structure as shown in Figure 4.1 (b). Applications using multi-stage linear pipelines can be found in wavefront array processor systems [18, 19]. An example algorithm with a multi-stage linear pipeline structure is QR decomposition [12, 18].
Again, we used special programs to generate the task systems analyzed. As with the partitioning structured task systems, tasks were randomly assigned task execution time distributions and resources. We performed two types of tests. Like the earlier tests, we used task systems in which the task execution times were either uniformly and triangularly distributed, or uniformly, triangularly and exponentially distributed. The bounds of the uniform and triangular distributions, and the means of the exponential distributions were assigned as in Section 4.2.1. The values of $t_1$ and $t_3$ of the three-step distribution approximations are also determined in the same manner as the values in Section 4.2.1.

Figure 4.5 shows the results when tasks were only assigned bounded distributions. All values in the graphs are sorted in increasing order. We analyzed sixty six task systems with four-stage linear pipeline structures. The task systems had between one and six FCFS resources. Eleven task systems were generated for each of the different numbers of resources. The YACSIM simulations were run until 99.9% confidence interval widths of less than 1% of the sample mean task system execution times were obtained. ES generates the complete sequencing trees of the execution of the task systems. The first two graphs show the error of ES's estimates of the means and standard deviations of task system execution times compared to those returned by the YACSIM simulations. Again, error is defined in Eq. 4.1. The maximum error of ES's estimated mean was 6.6%. This was an increase over the maximum error obtained for the five-level partitioning structure task systems with bounded task execution times. Because there were more tasks in the four-stage pipeline structure task graphs and the same number of resources, more tasks contended for shared resources. It is likely that the larger error came from inaccuracies in computing decision probabilities and in the increased dependencies among task finish times. The maximum standard deviation error was 57.1%. Finally, the speedup graph shows that ES was between 1 and 200 times faster than YACSIM at making predictions for task systems.
Figure 4.5: Results for four-stage linear pipeline task graphs with uniformly and triangularly distributed task execution times

Figure 4.6 shows the results from another sixty six task systems whose tasks could have task execution time distributions that were either bounded or unbounded. As before, the task systems had between one and six FCFS resources. Eleven task systems were generated for each of the different numbers of resources.
Figure 4.6: Results for four-stage linear pipeline task graphs with uniformly, triangularly, and exponentially distributed task execution times

As before, on average the errors of ES's estimated means and YACSIM's means were small. The maximum mean error was 9.3%, and the maximum standard deviation error was 16.7%. Like partitioning structure task graphs, adding exponential task execution time distributions increased the size of the mean errors. This suggests that the method we use for approximat-
ing arbitrary distributions is not as accurate for unbounded distributions as it is for bounded distributions. This is expected since we are approximating all distributions with a type of distribution that is bounded. Once more, we see the large increase in speedup of ES over YACSIM. This is again explained by the difficulty YACSIM has in obtaining tight confidence intervals when unbounded distributions are modeled.

Again, exact comparisons between ES and previously mentioned analytical performance prediction techniques are not possible. Mak-Lundstrom, Thomasian-Bay, and Sahner-Trivedi cannot model task graphs with the structure shown in Figure 4.1 (b) because the structure is non-series-parallel. Madala [21] presents methods of performance prediction for task graphs with this structure, but contention is ignored.

4.2.3. The Efficiency of ES Compared to an Event-Driven Simulation

For the test presented in Sections 4.2.1 and 4.2.2, ES generated the entire sequencing trees. The largest amount of time ES took to estimate the mean execution time of a task system was 7.08 seconds. As mentioned earlier, the efficiency comparisons presented in Sections 4.2.1 and 4.2.2 show high speedups of the time for ES to make estimates over the time YACSIM simulations take to generate results satisfying the constraints of the confidence level. We now present results of an experiment that shows how the efficiency of ES compares to YACSIM simulations in which the constraints of the confidence level were reduced. A single task system was analyzed by both ES and the YACSIM simulator. This task system was the one analyzed that resulted in the median speedup value of Figure 4.4. Figure 4.7 presents the speedup of ES over YACSIM simulations that had different levels of confidence. The confidence interval widths were set equal to two times the median value of the first graph of Figure 4.4, resulting in a confidence interval widths of 1.0124% of the sample means.
Figure 4.7: Varying confidence levels of the YACSIM simulations

Figure 4.7 shows that decreasing the confidence level required of the YACSIM simulations had a dramatic effect on the speed in which YACSIM returned an estimate for the mean execution time of a task system. However, even at a confidence level of 80%, the YACSIM simulator took 75.4 times longer to return an estimate than ES. The other curve of the graph shows that for this task system no matter what confidence level was chosen between 99.9% and 80%, the error between ES's estimate of the task system's mean execution time and YACSIM's does not vary significantly. YACSIM's estimate of the task system's mean execution time remained nearly constant as the confidence level varied between 99.9% and 80%. This trend may not be true when the YACSIM simulator simulates other task systems.

4.3. Interconnection Network Models

In the remainder of this chapter, the parallel systems studied had one of four interconnection network structures: fully connected, mesh, hypercube, and bus. Figure 4.8
gives an example of each of these networks. In the examples, processing nodes are designated by circles and communication links by lines. A processing node includes a processor, local memory, and message routing circuitry.

Figure 4.8: Example interconnection networks

Figure 4.8 (a) shows a 4 x 4 fully connected network. In a fully connected network, each node has a direct connection with every other node. Each processor has a single input and output port. The output port is connected to a demultiplexer that has direct links to all other processors. The input port of a processor has a multiplexer that is connected to the links coming from all other processors. While there is no contention for individual communication links, if two or more messages have the same destination processor, only one message
will be received at a time. The remaining messages are buffered at the destination's multiplexer. The multiplexers and demultiplexers are not shown in the figure.

Figure 4.8 (b) shows a 4 x 4 mesh of 16 nodes. In our mesh model, messages are routed to the destination node's column and then routed to the destination through nodes on the same row. When two or more messages contend for the same communication link, they are buffered at the node where the contention occurs. Processors are not interrupted to rout messages.

Figure 4.8 (c) shows an 8-node hypercube. In an n-node hypercube, each node is connected to \( \log_2 n \) other nodes. If the nodes are assigned unique binary addresses with value in the range \((0, 2^n - 1)\), each node \(a\) is connected to nodes whose addresses differ from the address of \(a\) by only one bit as shown in the figure. In our model, messages are routed through nodes in a most significant bit first method. This is referred to as dimension-ordered routing. For example, if a message is sent from node 000 to node 111, the message would travel in order through nodes 100 and 110 before reaching node 111. Messages travel through intermediate nodes without interrupting the processor on the node. If there is contention for communication links, contending messages are buffered at the intermediate nodes.

Figure 4.8 (d) shows four processors connected by a bus. In a bus connected parallel system, the same communication link is used to send messages between all processors. Only one message may be on the bus at a time.

4.4. Comparisons with an Intel iPSC/860 Hypercube

This section compares the mean execution times of a mergesort algorithm implemented on an Intel iPSC/860 hypercube parallel computer to those estimated by ES. Figure 4.9 shows the task graph of the mergesort algorithm analyzed.
The mergesort algorithm operates in the following manner. For an $n$ processor system, the *divide* task divides the unsorted data into $n$ equal parts. The task then sends the data over the interconnection network to the $n$ *sort* tasks. There is one sort task assigned to each processor. Each sort task sorts its local data and sends the sorted data to the merge task. After receiving all the sorted lists, the *merge* task combines the sublists into one sorted list.

The above task graph does not show the tasks associated with communication. One processor has the divide task, one sort task, and the merge task assigned to it. The other processors are assigned one sort task. When a task needs to send data to a task that is not assigned to the same processor, *communication* tasks are added to represent the communication. Communication tasks include both tasks assigned to the processors representing software overhead, and tasks assigned to network communication links representing the actual transmission of data.

We envision using ES as a tool of predicting the performance of various numbers of processors connected by various networks. If a user has access to a single processor system, he can use that system to gather timing data on various computation tasks of a parallel program. This data can then be used in the task system models given to ES that represent
the implementation of parallel programs on parallel systems using the same processor. In keeping with this idea, we computed the distributions of the tasks of the mergesort algorithm through experimentation on one processor of the iPSC/860. Simple programs were written to calculate the sample means, variances, and skews of the tasks. Because we could not find published timing information on message passing overhead, other measurements were taken to determine the software and hardware overheads of sending messages of various sizes. Through tests, it was determined that software overheads were orders of magnitude larger than hardware overheads, depending on message size. Thus, messages traveling through intermediate nodes did not see a significant increase in transmission time. Because of the high bandwidth of the communications links, ES models the communication structure of the iPSC/860 as a fully connected network. The data taken from the message passing experiments and the single processor measurements were used in the ES task system models of the mergesort algorithm executing on larger systems.

We studied the execution of a mergesort of 100,000 32-bit integers on various size subcubes of an iPSC/860, using up to sixteen processor subcubes. The mean execution times of these sorts were compared to estimates from ES. Figure 4.10 shows the error between the ES estimates of the mean execution times to those measured on the iPSC/860. Multiple sorts of different sets of random data were performed on the iPSC/860 so that the means of the task execution times and the total execution time were calculated with 99.9% confidence intervals with widths less than 1% of the sample means. ES constructs the entire sequencing trees before calculating its estimates. For the three-step distribution approximations ES uses to approximate the distributions of task execution time random variables, $f$ in Eqs. 2.23 and 2.24 is set to 1.
Figure 4.10: Error of predicted mean execution time of mergesort

The above graph shows that the error of the ES estimate increased as the number of processors being modeled increased. The maximum error was 4.4% at sixteen processors. We feel the increase in error was caused by an increase in software overhead for handling larger numbers of messages simultaneously that we are not modeling. ES estimated extremely small standard deviations of the execution times for all the experiments. The coefficients of variation for all the predictions were less than 0.003. The coefficients of variation measured on the iPSC/860 were all less than 0.000007. While the relative difference between the ES estimates of coefficient of variation and those measured on the iPSC/860 are quite large, the small values lead us to conclude that the execution time of the mergesort algorithm is nearly constant for any data set.

Figure 4.10 also shows estimates returned from the Mak-Lundstrom technique. The sample means of the single processor and message passing experiments were used as the means of task execution times. As can be seen, the Mak-Lundstrom predictions are less accurate than ES. The Mak-Lundstrom technique does not take into account the standard deviations and skews of individual task execution times. Mak-Lundstrom also gives estimates of the standard deviation of the completion time of the mergesort program. The co-
efficients of variations derived from the Mak-Lundstrom results ranged from 0.39 to 0.87, much greater than those determined from ES or the iPSC/860.

As another test, we replaced the sort algorithm used in the sort tasks of the mergesort algorithm with a more efficient sequential quicksort. Figure 4.11 shows the results from these experiments.

![Graph showing error of predicted mean execution time of mergesort with sequential quicksort performed by sort tasks.](image)

**Figure 4.11:** Error of predicted mean execution time of mergesort with sequential quicksort performed by sort tasks

The maximum error of the ES estimated mean execution time was 7.0% at sixteen processors. This was an increase over the experiments of Figure 4.10. This strengthens our hypothesis that the error was caused by not modeling some of the communication overhead. By using the quicksort algorithm, the total execution time of the mergesort was lowered. However, the time spent on communication remained the same. Thus, we expected to see an increase in the overall error. Again, the estimated and real coefficients of variation of mergesort execution times were quite small (less than 0.007). The error from the Mak-Lundstrom technique is again presented. Generally, the errors of these predictions were much higher. At the sixteen processor case, the Mak-Lundstrom technique had a lower error than ES. We do not know if this trend would continue for larger numbers of proces-
sors. However, Mak-Lundstrom's prediction of coefficient of variation varied from 0.40 to 0.78, several orders of magnitude greater than the ES estimates and the iPSC/860 measurements.

4.5. Comparisons Against Execution-Driven Simulation

This section compares ES estimates to estimates from execution-driven simulations. Execution-driven simulation uses the actual execution of real algorithms to drive simulation models of computer architectures. The execution-driven simulation environment used is called PARCSIM. PARCSIM, currently under development at Rice University, is the second generation of the Rice Parallel Processing Testbed [7, 8, 9, 12]. The next section compares ES predictions and PARCSIM predictions for a mergesort algorithm executing on crossbar connected processors. The following section studies predictions for a Fast Fourier Transform algorithm executing on mesh connected processors.

4.5.1. Results for Tests on the Mergesort Algorithm

The mergesort algorithm studied was the same algorithm whose task graph is shown in Figure 4.9. The interconnection network modeled was a fully connected network. Figure 4.12 shows the error of ES estimates for mean task system execution times compared to estimates generated from execution-driven simulations of a mergesort algorithm sorting 10,000 32-bit integers. The PARCSIM simulation simulates the mergesort realized on a crossbar connected array of IBM RS/6000 processors. The simulated processors had a clock speed of 40.0 MHz, and the communication links had transmission speeds of 10 Mbytes/sec. The sample means, variances, and skews of individual task execution times used in the ES models were determined from PARCSIM simulations of a single processor.
The task execution times of communication tasks were constant and determined mathematically from the transmission speed and message size. The PARCSIM simulations were executed with different random data sets until 95% confidence intervals with widths of less than 5% of the sample mean execution times of the algorithm were obtained. ES generates its estimates by constructing sequencing trees entirely.
Figure 4.13 shows the errors between the mean task system execution times estimated by ES and those obtained from PARCSIM simulations for the mergesort algorithm when the sorting algorithm used by the sort tasks was a more efficient quicksort. For both mergesort algorithms, the errors between the ES predicted mean and the PARCSIM generated mean were between 0% and 1.4%. Comparing the efficiency of ES and the PARCSIM simulations is not a direct comparison because the two programs execute on different systems. ES runs on a Sun SPARC 2 workstation with a clock speed of 40 MHz. The PARCSIM simulations execute on an IBM POWERstation 530 with a 25 MHz RS/6000 processor. However, all the ES predictions were completed in less than 0.6 seconds, while all the PARCSIM simulations took over 6000 seconds to complete. It should be noted that since the system being studied had a crossbar interconnection network, there was no contention for resources. Thus, ES only created one sequencing tree node for each prediction.

4.5.2. Results for Tests on Fast Fourier Transform Algorithm

The next algorithm studied is a decimation-in-time Fast Fourier Transform (FFT) algorithm (see pp. 599-605 [26]). The Discrete Fourier Transform (DFT) is

\[ X[k] = \sum_{n=0}^{N-1} x[n]W_N^{-kn} \quad 0 \leq k \leq N - 1. \] (4.2)

There are \( N \) points in the input sequence \( x[n] \), and \( W_N = e^{i(2\pi/N)} \). The decimation-in-time FFT algorithm implemented decomposes the DFT computation by forming smaller subsequences of the \( n \)-point input sequence \( x[n] \). \( n \) must be a power of two. When the algorithm is to be implemented on a \( p \) processor system (\( p \) must also be a power of two), the algorithm decomposes the original \( n \)-point DFT into \( p \) \( n/p \)-point DFTs. The decomposition part of the algorithm requires \( \log_2 p \) stages to complete. During each stage, each processor
exchanges information with one other processor. Thus, a four-processor implementation of
the algorithm requires each processor to exchange information with other processors twice.
Eq. 4.3 shows which processor \( P_j \) a processor \( P_i \) exchanges data with during first stage of
the decomposition.

\[
\text{distance} = \frac{p}{2} \\
j = i \oplus \text{distance}
\] (4.3)

\( \oplus \) is the bitwise exclusive OR operator. Before each subsequent stage, \( \text{distance} \) is halved.
Thus, when \( p = 4 \), \( P_0 \) communicates with \( P_2 \) during the first stage of the decomposition and
with \( P_1 \) during the second stage. After two stages, the decomposition is complete and \( P_0 \)
(and the other processors) computes an \( n/4 \)-point DFT.

Figure 4.14 shows a simplified task graph for the decimation-in-time FFT algorithm
implemented on four processors. The vertical dashed lines show how the tasks are divided
among the four processors. When a precedence constraint crosses a processor boundary, a
message must be sent over the system’s interconnection network. Tasks associated with
communication are not shown in Figure 4.14. All communication occurs during the de-
composition phase of the algorithm. As the figure shows, the task graph of the FFT algo-
rithm is non-series-parallel.

We compared ES estimates of the execution times of the FFT algorithm against ex-
cution times returned from PARCSIM simulations, where there are 1024 points in the input
sequence \( x[n] \). The parallel systems modeled had RS/6000 processors connected by a
mesh interconnection network (Figure 4.8 (b)). The simulated processors had a clock
speed of 40 MHz, and the communication links had transmission speeds of 1 Mbyte/sec.
The tasks in the FFT algorithm had constant execution times. The task execution times
used in the ES input specification were determined from PARCSIM simulations of a single
Figure 4.14: FFT task graph

processor. The task execution times of the communication tasks were determined mathematically from the transmission speed and message size.

Figure 4.15 shows the errors between the task system execution times estimated by ES and those obtained from the PARCSIM simulations. Because all task execution times were constant, there were no branches in the sequencing tree constructed by ES. The errors between ES and PARCSIM simulations were between −0.15% and +0.15%. Since all tasks had constant task execution times, no distribution approximation was needed. Thus, ES estimates were very accurate.
4.6. Trading Accuracy for Efficiency

ES's efficiency decreases proportionally with the size of the sequencing tree it must create. The size of the sequencing tree depends upon the number of tasks in the system, the number and structure of the precedence constraints, the number of resources in the system, and the task execution time distributions. As these factors are varied, the size of the sequencing tree can change. These factors are not independent. Generally, adding more precedence constraints, while keeping the number of tasks the same, reduces the different sequence of events that can occur if there were fewer precedence constraints. Adding more tasks to the system while keeping the same number of resources can increase the amount of contention between tasks for the same resource. This increases the number of sequences of events that must be represented as nodes of a sequencing tree.

ES allows the user to trade accuracy for efficiency by the depth-first sequencing tree construction method described in Section 3.3.2. For tests shown in the previous sections, ES could efficiently construct the full sequencing trees representing all sequences of events.
The longest it took for ES to make a prediction for any of the test task systems analyzed in the previous sections was 7.08 seconds.

To determine the finish order of \( n \) tasks executing concurrently, \( n! \) nodes of the sequencing tree must be constructed. Figure 4.16 shows a task system with six processors connected by a single FCFS bus communication network.

![Diagram](image)

**Figure 4.16**: Task system with 18 tasks, six processors, and a single bus

Task \( t_1 \) communicates with tasks \( t_8 \) through \( t_{12} \) on different processors by sending messages over the bus. These messages are represented by tasks \( t_2 \) through \( t_6 \). As \( t_8 \) through \( t_{12} \) complete, they send messages over the bus to \( t_{18} \). These messages are represented by \( t_{13} \) through \( t_{17} \). Since \( t_2 \) through \( t_6 \) start at the same time, the order in which the tasks get the bus is determined by the scheduling policy of the bus. Since the bus is FCFS, any ordering is valid. The order in which \( t_{13} \) through \( t_{17} \) use the bus depends of the FTs of \( t_8 \).
through \( t_{12} \). If the executions of \( t_8 \) through \( t_{12} \) all overlap, \( 5! \) or 120 nodes of the sequencing tree must be generated to determine their possible finish orders, which in turn determines the possible execution orders of \( t_{13} \) through \( t_{17} \). If more processors are added, and more tasks which use these processors concurrently are added, the size of the sequencing tree grows exponentially. Thus, ES's ability to efficiently estimate the execution time of the task system decreases rapidly with the increase in size of the task system. It is necessary to apply the heuristic depth-first construction algorithm presented in Section 3.3.2 to limit the size of the sequencing tree.

Figure 4.17 shows the effects on ES's estimate of mean execution time as the cumulative terminal node probability level varies for the task system shown in Figure 4.16. \( t_1 \) and \( t_{18} \) have exponentially distributed task execution times with a mean of 0.5. \( t_2 \) through \( t_6 \) and \( t_{13} \) through \( t_{17} \) have constant task execution times of 0.1. \( t_7 \) through \( t_{12} \)'s task execution times are exponentially distributed with means of 1.6667. A cumulative terminal node probability level of 100% represents construction of the entire sequencing tree. A cumulative terminal node probability level of 0% represents the construction of the sequencing tree depth-first to a single terminal node. The error shown is the error between ES estimates and estimates returned from a YACSIM simulation of the same task system as defined by Eq 4.1. The YACSIM simulation ran until the confidence level of the sample mean of the task system's execution time was 99.9% and the interval width was 1% of the sample mean.
Figure 4.17: Effect of varying cumulative terminal node probability level

The first graph shows the effect of varying the cumulative terminal node probability level on the error of ES's estimates of the mean and standard deviation of the task system's execution time. The error of the mean was between ±4% when the cumulative terminal
node probability level was greater than 20%. In this range, the error of ES's estimate of standard deviation stayed nearly constant at about 16%. At below 10%, the error of the ES estimates increased dramatically. The second graph shows the increase in efficiency as the cumulative terminal node probability level was decreased. The rate of speed increase as the cumulative node probability level is decreased will vary for different task systems.

Determining an appropriate cumulative terminal node probability level is problematic. Initially, the user can set the terminal node limit to $n$. This means that the construction of the sequencing tree will stop after $n$ terminal nodes are created. $n$ should be set low enough to insure ES terminates in an acceptable amount of time. This value depends on the task systems analyzed and the host computer of ES, and must determined by the user's experiences. When an estimate is returned, ES also reports the sum of the probabilities of the terminal nodes created. The user can decide if this cumulative probability is sufficiently high. If the cumulative probability is not high enough, the user can increase the value of $n$ to create more terminal nodes.

4.7. Example Applications

The previous sections presented experiments that tested the accuracy and efficiency of the prediction technique. We now present two sample applications for the technique. These examples show how the technique can be used to study design issues in parallel systems. The first example shows how the technique can be used to determine the bottleneck between processors and communication links in a parallel system executing a specific parallel algorithm. The second example shows how the technique can compare the performance of different interconnection structures.
4.7.1. Bottleneck Detection

When designing parallel systems, one design issue is balancing processing power with communication bandwidth. If the speed of the communication links is too slow, the processors may be idle, waiting for messages from other processors that are delayed by the speed of the communication network. Increasing processor speed will improve performance only minimally, if at all. A more effective way to improve the overall performance of the parallel system may be to use slower, cheaper processors and apply the cost savings to building a faster communication network. Conversely, if the communication links are very fast, resources may have been spent in developing a high speed communication network that does not improve the overall performance of the parallel system because processors, not communications, are the bottleneck.

As an example we study the mergesort algorithm whose task graph is shown in Figure 4.9. We study an array of 16 fully connected RS/6000 processors with clock speeds of 40 MHz. The task execution times of the tasks of the algorithm were determined from PARCSIM simulations of a single processor operating on a data set of 10,000 32-bit integers. Figure 4.18 shows the mean execution time of the task system as the communication speed was varied as estimated by ES. The cumulative terminal node probability level was set to 100% so that entire sequencing trees were constructed.

From the knee of the curve in Figure 4.18, it can be determined that the bottleneck between the communication network bandwidth and processor power occurred when the speed of the communication links was about 0.5 Mbytes/sec. When the link speed was below 0.5 Mbytes/sec, small increases in link speed resulted in relatively large decreases in the mean execution time of the mergesort algorithm. For example, increasing link speed from 0.1 to 0.3 Mbytes/sec reduced the mean execution time of the algorithm by 57.5%. Beyond the 0.5 Mbytes/sec mark, increases in link speed had a smaller effect on performance. Increasing the link speed from 0.5 to 1 Mbytes/sec decreased the mean execution
time of the mergesort algorithm by 27.9%. Increasing the link speed beyond 10 Mbytes/sec resulted in almost no increase in performance.

![Graph showing mean execution time vs communication link speed](image)

**Figure 4.18: Bottleneck detection**

By identifying such bottlenecks early in the design process, architects of parallel systems can determine where to spend the most effort and resources in the system design. As shown in the example above, if the processors are 40 MHz RS/6000s, a fully connected communication network with link speeds of greater than 10 Mbytes/sec does not significantly decrease the mean execution time of a mergesort of 10,000 32-bit integers. Further tests using a range of algorithms that represent the type of applications envisioned for the parallel system could help determine the smallest communication link speed needed to adequately utilize the processing power.

### 4.7.2. Architecture Comparisons

Another application for the analytical performance prediction technique implemented by ES is communication network comparison. By performing analytical modeling early in the design process, architects can determine whether or not costlier communication networks improve the performance of parallel systems constructed using identical processors.
As an example, we again look at the FFT algorithm used in Section 4.5.2. The same task execution times determined from PARCSIM simulations of a single RS/6000 processor were used. ES was used to estimate the execution time of the FFT algorithm implemented on bus, mesh, and hypercube connected processors. In general, the relative costs of the communication networks increase in order from bus to mesh to hypercube connected structures.

Figure 4.19 shows ES estimates of the execution times of a 1024-point FFT implemented on variously connected arrays of RS/6000 processors. The speed of the communication links is 1 Mbyte/sec and the speed of the processors is 40 MHz. Because all task execution times are constant, there are no branches in the sequencing tree constructed by ES.

![Communication network comparison](image)

**Figure 4.19:** Communication network comparison

It should be noted that at two processors, bus, mesh, and hypercube structures are identical. At four processors, mesh and hypercube structures are identical.
Of the systems modeled, the bus connected network had the worst performance. At 40 MHz processor clock speeds and 1 Mbyte/sec communication link speeds, the bus is saturated. Thus, increasing the number of processors decreases performance. The hypercube network had the best performance. At sixteen processors, the hypercube system is 19.7% faster than the mesh system. The communications of the FFT algorithm map perfectly to a hypercube connection network (i.e., each communication only travels over one link). Consequently, a system with a fully connected communication network would not compute an FFT faster than a system with a hypercube interconnection network.
CHAPTER FIVE

Conclusion

5.1. Summary

The goal of this research was to develop an efficient and reasonably accurate analytical technique for modeling specific parallel algorithms executing on parallel architectures. The technique models explicit precedence and synchronization constraints, arbitrary task execution times, resource contention, and a variety of resource scheduling policies.

Precedence and synchronization constraints are modeled by directed acyclic graphs called task graphs. Using task graphs, a range of parallel algorithms can be represented by dividing them into units of computation called tasks. These tasks are the nodes of the task graph. The explicit orders in which tasks must execute are represented as edges in the task graph. Task graphs are most appropriate for describing parallel algorithms with large-grain parallelism. Programs with large-grain parallelism are characterized by relatively long units of computation between task interactions. An example of task interaction is the exchange of data between tasks on different processors via message passing.

The modeling of arbitrary task execution times is accomplished by approximating the random variables representing task execution times by a single class of random variables called simple random variables. The approximation is accomplished by matching a predetermined number of moments of the original task execution time distribution with those of a distribution for a simple random variable with a given number of possible values. We match the first four moments, using a three-step distribution (corresponding to a simple random variable that can take on three values).

When there is a non-zero probability that the executions of two or more tasks that use the same resource will overlap, resource contention can occur. Resource contention is
resolved by representing the different possible sequences of events that can affect the order in which contending tasks use the resource. The different sequences of events are represented in a sequencing tree. The terminal nodes of a fully constructed sequencing tree represent all the sequences of events that can occur in the execution of a parallel algorithm on a parallel architecture. Estimates of the mean and standard deviation of the execution time of the parallel execution represented by a sequencing tree are determined from the probabilities of the sequences represented by the terminal nodes occurring and the distributions of the finish times of the sequences of events they represent.

Sometimes a sequence of events will result in several tasks becoming eligible to begin service on the same resource at the same time (i.e., the tasks are queued at the resource). In this case, the resource's scheduling policy decides the order in which the tasks acquire the resource. The analytical technique allows the modeling of a variety of scheduling policies including first-come-first-served, last-come-first-served, and priority. Any scheduling policy that can be implemented by scanning tasks in a list that reflects the order in which tasks entered a queue can be modeled, provided that once a task begins execution on a resource its execution cannot be interrupted or increased. Examples of policies that do not meet these criteria are time-slicing, last-come-first-served-preemptive-resume and processor sharing.

The technique is implemented in a program called ES that runs on a Sun SPARC 2 workstation. ES estimates of the mean execution times of parallel systems executing parallel algorithms are compared to mean execution times returned from discrete-event simulations, actual executions of parallel algorithms on a parallel architecture, and detailed simulations of executions of parallel algorithms on parallel architectures. For all the comparisons, the maximum difference between the ES estimates for mean execution time and those obtained by other means is 9.4%.

The other major issue in this approach is efficiency. The efficiency of ES was compared to discrete-event simulations. 264 task systems were analyzed by both ES and the
discrete-event simulator. These task systems either had a five-level partitioning structure or a four-stage linear pipeline structure. The number of resources in the systems and the task execution time distributions varied from system to system. When the discrete-event simulations were required to generate estimates with confidence levels of 99.9% with interval widths of 1% of the sample mean, ES was tens to thousands of times faster than the discrete-event simulations at making estimates of the execution time of the task systems. The effect on the relative speedup of ES over the discrete-event simulations if the confidence levels required of the simulations were reduced was studied. A single task system was analyzed for this experiment. When the confidence level was lowered to 80% with a confidence interval width 1.0124% of the sample mean, ES was still 75.4 times faster generating estimates than the discrete-event simulation. Direct comparisons of the efficiency of ES compared to the detailed simulator PARCSIM were not possible because the host systems of the respective techniques differ. However, for the mergesort tests performed all the ES estimates were returned in less than 0.6 seconds while all the simulations completed in over 6000 seconds.

For the tests performed, ES could efficiently construct sequencing trees that represented all the sequences of events that could occur during the modeled executions. However, for certain algorithm and architecture combinations, construction of the entire sequencing tree by ES will be inefficient. For example, suppose \( n \) tasks are executing concurrently on different resources and each of these tasks has a separate non-common descendant that requires the use of the same resource as the other descendants. Such an occurrence could happen when \( n \) concurrent tasks must return data via messages over a single bus to a single task. The order in which the descendants use the shared resource depends on the finishing order of the \( n \) parent tasks. The number of nodes of a sequencing tree needed to represent the different finish orders of \( n \) concurrent tasks is \( n! \). ES allows the user to trade accuracy for efficiency by basing estimates on a partial construction of the se-
sequencing tree. At each sequencing decision, ES always constructs the subtrees in order of decreasing branch probabilities. This is an attempt to generate the terminal nodes with the higher probabilities first. The user may instruct ES to construct a sequencing tree until the sum of the probabilities of the terminal nodes created reaches some predetermined amount or until the total number of terminal nodes created exceeds some predetermined limit. Any estimate returned by ES is qualified by the fact that only a subset of the possible sequences of events is represented. Our experience is that ES results are largely unaffected by this procedure to a point, and that estimates of means in particular are relatively insensitive to this procedure.

The technique implemented by ES should be useful to designers of parallel systems early in the design process. ES can be used to determine the bottleneck between communication bandwidth and processing power for specific parallel algorithms. We presented the results of an experiment in which different communication link speeds in a parallel system were modeled. The system modeled was a fully connected array of 40 MHz RS/6000 processors. The effects of increasing the communication link speed on the mean execution time of a mergesort of 10,000 32-bit integers was examined. From the experiments it was determined that the knee of the bottleneck curve occurred when the link speed was about 0.5 Mbytes/sec. When the communication link speed is less than 0.5 Mbytes/sec, relatively small increases in link speed can result in significant increases in performance.

ES can also be used to compare the effects of different interconnection structures on the execution times of parallel algorithms. We presented results from experiments of different communication network structure connections of RS/6000 processors executing an FFT algorithm. The three connection structures examined were bus, mesh, and hypercube connected arrays of up to sixteen processors. All processors had a 40 MHz clock speed and all communication links had a throughput of 1 Mbyte/sec. For the 1024-point FFT algorithm, the experiment showed that a bus connected system was communication bound,
and thus adding more processors degraded performance. The hypercube connected system had the best performance, being 19.7% faster than a mesh connected system when both systems had sixteen processors.

Other design issues are not suited for study with this technique. For example, the effects of different cache memories on overall performance are not easily studied. Also, it is not easy to model the fine details of different message routing methods such as wormhole routing with this technique. Basically, any fine-grained parallelism study is impractical.

5.2. Future Work

Directions for future work center on enhancements to the prediction technique and applications for the technique. Enhancements to the technique may be developed to allow the modeling of preemptive resource scheduling policies. Also, more intelligent methods could be developed for eliminating redundant nodes of the sequencing trees. For example, suppose that the only contention for a shared resource occurs near the end of a task graph, as in two communications contenting for the same bus. Also, suppose that the two communications share no common ancestors and that the ancestors of one communication use none of the same resources as the other's ancestors. As the technique is now implemented, the sequencing tree of must represent the finish order of all of the ancestors. However, if such a condition is recognized, the total execution time of the ancestors of one communication can be solved in isolation. These ancestors can be replace by a single task whose task execution time equal to the total execution time determined earlier. The same technique is then used to replace the ancestors of the other communication. The size of the sequencing trees constructed would be less than the size of the single sequencing tree where no replacement occurred. Currently, the user can do this manually.

Another way to improve the efficiency of the technique is to parallelize the algorithm implementing the technique. One method would be to assign a single process to manage
the list of the sequencing tree nodes created. Each processor of the host parallel system would have a single process that requests a node from the manager process. Once the node is received, the work process computes the sequence of events represented by the node until a sequencing decision is required. When this occurs, the work process generates the outcome nodes of the decision and sends them to the manager process. The work process then requests a new node on which to operate.

There are several ways in which this technique can be useful to investigate issues in parallel computing. The first is the design of parallel computers. Using the technique, the designer can quickly develop models of different parallel architectures executing different parallel programs. Using these models, the designer can compare the performance of different interconnection network structures on different classes of parallel programs. Once the better interconnection networks for the class of programs of interest are determined, detailed simulations can be used to further study and improve the design. By identifying the good and bad designs earlier using the analytical technique, the designer reduces the amount of time that must be spent on detailed simulation. Another design issue is identifying the performance range for a communication network that results in adequate utilization of processors of a given speed, or vice-versa. By knowing the approximate performance needed to balance the processor speed and the communication network speed, the designer can be sure to spend resources on the components that will affect performance.
References


