INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript 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. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

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.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 8900259

**Performance of synchronous parallel algorithms with regular
structures**

Madala, Sridhar, Ph.D.

Rice University, 1988

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

RICE UNIVERSITY

PERFORMANCE OF SYNCHRONOUS PARALLEL ALGORITHMS
WITH REGULAR STRUCTURES

by

SRIDHAR MADALA

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

DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE:

Dr. James B. Sinclair, (Director)
Associate Professor
Electrical and Computer Engineering

Dr. J. Robert Jump
Professor
Electrical and Computer Engineering

Dr. Hans-Juergen K.H. Boehm
Assistant Professor
Computer Science

Houston, Texas

May, 1988

**Performance of Synchronous Parallel Algorithms
with Regular Structures**

by

Sridhar Madala

Abstract

The ability to model the execution and predict the performance of parallel algorithms is necessary if parallel computer systems are to be designed and utilized effectively. The performance of a parallel algorithm depends on many factors, some of which are inherent to the algorithm, and some of which depend on implementation on a specific architecture. One factor which is inherent to the structure of the algorithm is the delay due to synchronization requirements when one task has to await the completion of another task.

This thesis is concerned with the problem of evaluating the performance of parallel algorithms on MIMD computer systems when tasks in the algorithms have non-deterministic execution times. It addresses the following questions: How does non-determinism affect the synchronization delays and ultimately the performance of parallel algorithms? Is it possible to predict this effect independent of implementation on a specific architecture? How can such predictions be validated?

The approach adopted is based on the view of a parallel algorithm as a collection of tasks with constraints on their relative order of execution. Important classes of parallel algorithms which exhibit regularity in their task graph structures are identified and the performance of these

structures is predicted using results from order statistics and extreme value theory. These algorithm classes are those whose task graphs have multiphase, partitioning, and pipeline structures. The Rice Parallel Processing Testbed (RPPT), a program-driven simulation tool for the performance evaluation of parallel algorithms on parallel architectures, is used along with distribution-driven simulation to validate the results.

Variations in the execution times of tasks generally increase the length of synchronization delays and adversely affect the performance of a parallel algorithm. In the case of an algorithm with a regular task graph structure it is possible to quantify the performance degradation due to non-determinism under certain assumptions about the task execution times and the number of processors available to execute the tasks. In particular, it is possible to place bounds on and approximate the mean execution time of the multiphase, partitioning, and pipeline structures. Distribution-driven simulations and results from sorting algorithms run on the RPPT indicate that these bounds and approximations, which are based on independence assumptions about task execution time distributions, are robust even in the presence of dependencies among task execution times.

To my parents,

Hymavathi and Venkateswara Rao,

and my brother Anil Kumar

Acknowledgements

It is indeed a pleasure to acknowledge the debt I owe to Bart Sinclair, who patiently endured my meanderings in search of a research topic, nudged me onto the right track, and painstakingly translated my scribbles into English. Were it not for his persistency in 1985 my education would have stopped short of a Ph.d. Dr. Jump has been an inspiration throughout the RPPT project, demonstrating by example how to distinguish between the trivial and the crucial. To him and Dr. Boehm I owe thanks for being on my thesis committee.

Fellow travelers on the long road towards a doctorate, Richard Covington, Kshitij Doshi, and Hemant Tagare made the journey a pleasant one with their deep friendship. I owe a great deal to Rick Covington for letting me "borrow" ideas from CSIM, and for his constant help and companionship.

Varun Mehta, Sandhya Dwarakadas, and Steve Ingels suffered through many versions of the RPPT and Concurrent C. What I called "features" they insisted on calling "bugs", and more often than not they were right. I thank them for their patience and for their friendship.

Table of Contents

1. Introduction	1
1.1 Introduction	1
1.2 Motivation	2
1.3 Approach To Problem	3
1.4 Task Graph Structures	6
1.5 Thesis Results	8
1.6 Overview of Thesis	12
2. Previous Work	14
2.1 Introduction	14
2.2 Classification of Parallel Algorithms	14
2.3 Measuring the Performance of Parallel Algorithms	16
2.4 Factors Affecting the Performance of Parallel Algorithms	17
2.5 Queueing and Petri Net Models	18
2.6 Accounting for Non-Determinism	19
2.7 Summary	21
3. Rice Parallel Processing Testbed	22
3.1 Introduction	22
3.2 Simulation Techniques	22
3.3 Overview of the RPPT	24
3.4 Concurrent C	28
3.5 RPPT Design Issues	32
3.6 Conclusion	34
4. Multiphase Algorithms	35
4.1 Introduction	35
4.2 Model and Assumptions	38
4.3 Previous Work	39
4.4 Bounds on Mean Execution Time	39
4.4.1 Lower Bound	40
4.4.2 Distribution-Free Upper Bound	41
4.4.3 Upper Bounds for Specific Distributions	41
4.4.3.1 Exponential Distribution	42
4.4.3.2 Uniform Distribution	42

4.4.3.3 Normal Distribution	43
4.5 Scheduling Policies	43
4.5.1 Static Scheduling	44
4.5.2 Dynamic Scheduling	45
4.5.3 Dynamic Scheduling with Overhead	48
4.6 Evaluation	49
4.7 Conclusion	51
5. Partitioning Algorithms	53
5.1 Introduction	53
5.2 Model and Assumptions	56
5.3 Previous Work	57
5.4 Analysis of Partitioning Structures	58
5.4.1 Bounds for Specific Distributions	58
5.4.2 Bounds for Dependent Task Times	59
5.4.3 Independent Paths Approximation	60
5.4.4 Independent Paths Normal Approximation	61
5.4.5 Type I Iterative Approximation	61
5.5 Performance Comparison	64
5.5.1 Stochastic, Independent Task Execution Times	64
5.5.2 Program-Driven Task Execution Times	68
5.6 Analysis of a Parallel Mergesort Algorithm	70
5.7 Conclusion	71
6. Pipeline Algorithms	73
6.1 Introduction	73
6.2 Model and Assumptions	74
6.3 Previous Work	75
6.4 Linear Pipeline Structures	76
6.4.1 Deterministic Time Lower Bound	78
6.4.2 Disconnected Chains Lower Bound	79
6.4.3 Independent Paths Approximation	81
6.4.4 Robinson's Upper Bound	82
6.4.5 Evaluation of Bounds	82
6.5 Two-Dimensional Pipeline Structures	88
6.5.1 Deterministic Time Lower Bound	88
6.5.2 Disconnected Chains Lower Bound	90
6.5.3 Longest Wavefront Upper Bound	90
6.5.4 Robinson's Upper Bound	91
6.5.5 Evaluation of Bounds	92
6.6 Conclusion	92

7. Conclusion	96
7.1 Results Obtained	96
7.2 Discussion	99
7.3 Future Research	100
References	102

List of Figures

Figure 1.1: An example task graph	4
Figure 1.2: Task graph for a multiphase structure	7
Figure 1.3: Task graph for a partitioning algorithm	9
Figure 1.4: Linear pipeline structure	10
Figure 3.1: Preprocessing undergone by a Concurrent C program	26
Figure 3.2: User's view of the RPPT	27
Figure 4.1: Task graph for a multiphase structure	36
Figure 4.2: Evaluation of bounds for dynamic scheduling with overhead	50
Figure 5.1: Task graph for a partitioning algorithm	54
Figure 5.2: Comparison of bounds and estimates for exponential distribution	65
Figure 5.3: Comparison of bounds and estimates for uniform and beta distributions	66
Figure 5.4: Quicksort of 8192 integers (RPPT simulation)	69
Figure 5.5: Mergesort of 8192 integers (RPPT simulation and IPN analysis)	69
Figure 6.1: Four-stage linear pipeline	77
Figure 6.2: Task graph for a four-stage linear pipeline	77
Figure 6.3: Modified task graph for linear pipeline	80
Figure 6.4: Comparison of bounds for linear pipeline, uniform distribution	83
Figure 6.5: Comparison of bounds for linear pipeline, exponential distribution	84
Figure 6.6: Comparison of bounds for linear pipeline, beta distribution	85
Figure 6.7: 3x3 wavefront array processor	86
Figure 6.8: Task graph for a 3x3 wavefront array processor	89
Figure 6.9: Comparison of bounds for 4x4 wavefront array processor	93

CHAPTER 1

Introduction

1.1. Introduction

The need for high performance computer systems has focused attention on faster switching and storage devices and on parallel processing. Although improved device technology increases the performance of computer systems, physical limitations and the need to build faster systems with existing technology leads designers to exploit the capabilities of multiple computing elements.

If parallel systems are to be designed effectively and utilized efficiently we must understand and be able to predict the performance of parallel architectures, parallel algorithms, and the interaction between architectures and algorithms.

This thesis is concerned with the problem of evaluating the performance of parallel algorithms on multiple-instruction, multiple-data stream (MIMD) computer systems when the execution times of various sections of the parallel algorithms are non-deterministic.

Parallel algorithms for MIMD systems, or multiprocessors, are typically structured as collections of processes that cooperate to solve a problem. These processes generally synchronize various parts of their computation with computations of other processes. A *task* is the computation between two successive synchronization or interaction points of a process. By definition tasks, once started, run to completion without interruption and any interaction between tasks take place only at the beginning and end.

The performance of a parallel algorithm depends on many factors, some of which are inherent to the algorithm itself, and some of which depend on implementation of the algorithm

on a specific architecture. One significant factor is the delay introduced due to synchronization requirements which are inherent to the structure of the algorithm itself. Variations in the execution times of tasks may increase the length of synchronization delays and adversely affect the performance of a parallel algorithm.

Our goal is to determine the effect of variations in task execution times on synchronization delays and consequently on the total execution time of several important classes of parallel algorithms. We focus in particular on the problem of quantifying the performance degradation due to non-deterministic task execution times for these classes. Further, we address the question of the appropriate means for validating any such performance predictions.

1.2. Motivation

The execution times for tasks are often non-deterministic or are best modeled stochastically. This may be due to various reasons such as queueing delays due to contention for resources such as memory and communication channels, because of data dependent computation times, or even a result of randomness in the nature of the algorithm itself. For example, in a shared memory multiprocessor the time to execute a task running on one processor depends on the access patterns to shared memory by tasks running on other processors. The exact time to sort a set of numbers depends on the actual numbers to be sorted. Search algorithms may randomly select the next path to search. Sometimes it might not be possible or feasible to get complete information about the execution time of a task. In all these cases task execution times are best characterized by probability distributions.

Variations in execution times generally result in unpredictable (or non-deterministic) synchronization delays where one task has to await the completion of other tasks. These synchronization delays are inherent in the structure of the algorithm and limit the potential speedup. It is

important to be able to determine the effects on performance of synchronization delays for several reasons. First, this gives a lower bound on execution time (an upper bound on speedup) that is independent of architectural features such as the number of processors, the structure of the interconnection network, and the communication bandwidth. Also, in those cases in which a task that becomes ready to execute always finds an available processor, if the interprocessor communication times are negligible or deterministic, they can be included as part of the task execution times to obtain accurate execution time estimates for the entire program. Finally, we can compare the performances of algorithms with different synchronization structures which may in turn lead us to better algorithm design.

1.3. Approach To Problem

Our view of a parallel computation is that of a collection of tasks with constraints on when a task can execute relative to other tasks. These tasks and their interrelationships are often depicted in a task graph or dependency graph. Our terminology for describing task graphs is drawn mainly from scheduling theory [1]. Fig. 1.1 shows an example dependency graph. Nodes in the graph represent tasks and edges represent dependencies between tasks. A task *a* is a *predecessor* of a task *b* and *b* is a *successor* of *a* if there is a directed edge from *a* to *b*. Tasks without predecessors are called *initial* tasks and tasks without successors are called *final* tasks. A task cannot start until all its predecessors have completed execution. The *level* of a task is the length of the longest path from an initial task to that task. The execution time for the entire graph is the maximum time from the start of any initial task to the completion of all tasks.

Predicting the effect of non-determinism on a general parallel computation with an arbitrary task graph is a difficult problem. We consider the simpler problems of evaluating the performance of classes of parallel algorithms whose task graph structures show some regularity. Since our aim is to isolate the effects of non-determinacy on the performance of parallel

Figure 1.1: An example task graph

algorithms, we ignore other factors such as communication costs that could also affect the performance.

Our problem now can be more accurately described as follows: Given that we have a parallel algorithm whose task graph exhibits a regular structure, and given that the execution times of the tasks are described by probability distributions, what can we conclude about the performance of the algorithm independent of its implementation on any particular architecture?

Before we can evaluate the performance of an algorithm, we need to decide on appropriate measures of performance. The most common measures of performance of a parallel algorithm are the execution time and the speedup relative to a sequential implementation of the algorithm. We will use these as our primary measures of performance with one difference. Since task execution times are non-deterministic, the execution time and speedup for the algorithm will themselves be stochastic and hence it is more appropriate to consider the distribution and the moments of these performance measures.

Our primary analytical tools for evaluating the performance of these structures will be based on results from order statistics and extreme value theory. We will show that, given the regularity of the task graph structures and under certain independence assumptions about the probability distributions of the task execution times, our methodologies lead to accurate predictions and bounds on average execution time and average speedup.

Analysis and analytical predictions based on models of parallel computations are of questionable value unless they are validated by comparing the predictions to other, reliable performance measures. Since our model of a parallel computation is a collection of tasks with non-deterministic execution times we will use stochastic simulation to verify our analysis. Stochastic simulation uses probability distributions to drive an architectural model and is a relatively quick and efficient way to verify our predictions.

Stochastic or distribution-driven simulation, however, is not adequate when dealing with more complex task graph structures since it cannot capture all the nuances of the execution time behavior of a task. Either direct implementation and measurement on a parallel system or a more accurate method of simulation is required. We choose the second alternative and use a program-driven simulator, the Rice Parallel Processing Testbed [2], to validate the analysis of algorithms with a partitioning structure.

1.4. Task Graph Structures

We now describe the parallel algorithm classes chosen for our study. Parallel algorithms can be classified based on the structure of their task graphs [3]. Experience shows that most parallel algorithms belong to one of only a small number of classes [4]. Examples of classes of task graphs are those representing asynchronous, synchronous, multiphase, partitioned, and pipelined parallel algorithms. Algorithms with asynchronous structures are the simplest of all and consist of a collection of tasks without any synchronization constraints among them. Since we are interested in algorithms with synchronization requirements we will not consider those with asynchronous structures. We will concentrate on three frequently encountered structures: multiphase, partitioning, and pipeline.

Parallel algorithms with a multiphase structure have alternating serial and parallel phases. Fig. 1.2 shows the task graph for one iteration of a multiphase structure. A single master task is active during the serial phase, and several slave tasks are active during the parallel phase. Algorithms with structures similar to the multiphase structure but with non-existent or negligible master tasks have been variously termed synchronous or synchronized iterative algorithms.

Task graphs of algorithms based on a recursive use of the divide and conquer strategy often exhibit a partitioning structure. These algorithms typically consist of three phases: a divide

Figure 1.2: Task graph for a multiphase structure

phase during which work is successively divided or partitioned; a work phase during which computation is performed on the partitions; and a merge phase during which results from the previous steps are combined. A classic example of such an algorithm is the quicksort algorithm which partitions an array of elements to be sorted into two subarrays, each of which is subdivided recursively until the number of elements in a subarray is below a threshold. The work phase consists of sorting the elements in the subarray. The merge phase is either non-existent (if the partitioning and sorting are done in place) or trivial (if the partitioning and sorting are done on copies). The mergesort is a similar algorithm with a non-existent or trivial divide phase and non-trivial work and merge phases. Fig. 1.3 shows a two-stage partitioning task graph structure.

Often a computation is divided into a number of subcomputations or stages, with data for a particular task being processed by each stage in turn in a linear fashion. One example is the four stage linear pipeline of Fig. 1.4. The simple linear pipeline is often extended to two dimensions, as in the case of systolic arrays and wavefront array processors. We refer to parallel algorithms of this type as having a pipelined structure.

1.5. Thesis Results

The results of this research are the development of accurate performance models for three important classes of parallel algorithms with regular task graph structures and non-deterministic task execution times, analytic techniques for evaluating the performance of these algorithms using results from order statistics and extreme value theory, and the design and development of significant components of a program-driven simulation tool for the performance evaluation of parallel systems.

In the case of the multiphase structure we show that non-deterministic task times result in sublinear speedup and derive approximations to the speedup that are asymptotically correct for

Figure 1.3: Task graph for a partitioning algorithm

a) Four-stage linear pipeline

b) Task graph for a four-stage linear pipeline

Figure 1.4: Linear pipeline structure

specific task time distributions, namely the exponential, uniform, and normal distributions. In the absence of information about the nature of the task time distribution we derive distribution-free bounds that depend only on knowledge of the mean and variance of the task times. In addition, we derive bounds on the mean execution time for the case when the number of processors is smaller than the number of tasks. Our results show that dividing the parallel phase into many more tasks than the number of processors and assigning them statically to processors does not improve the performance over the case where there are as many tasks as there are processors. However, a dynamic assignment of tasks to processors results in near linear speedup when there are many more tasks in the parallel phase than there are processors. We also derive the optimum number of subtasks in the parallel phase of the multiphase structure that minimizes the bounds on the execution time under dynamic scheduling when there is constant scheduling overhead.

For the partitioning structure we derive bounds on the mean execution time for three different task time distributions, the exponential, uniform, and normal distributions. We also derive distribution-free bounds under the assumption that the tasks at a level are dependent. We then take advantage of the regularity of the partitioning structure and use the number of different paths in the structure to derive an upper bound on the mean execution time. A variation of this approach leads to the independent paths normal approximation that is highly accurate and robust even in the presence of dependencies among the task times. This approximation is used to derive an analytical expression for the mean execution time of a parallel mergesort algorithm. We also develop an iterative approximation for the mean execution time for task time distributions whose extreme values converge to the type I extremal distribution.

We deal with two pipeline structures, the linear pipeline and the two-dimensional array. For the linear pipeline we derive lower and upper bounds and approximations on the mean execution time. For the two-dimensional array we derive bounds using methods similar to those used for the linear pipeline. In addition, we derive an upper bound on the mean execution time

by placing a restriction on the regularity of the wavefronts in the two-dimensional array.

Methods developed for predicting the performance of the three different regular task graph structures share some similarities and indicate approaches that might be taken when studying other regular structures. In particular the methods based on the number of different paths in a task graph structure and the method for iteratively computing the execution time of a partitioning structure are applicable for other regular structures as well.

Other contributions made during the research relating to this thesis include the design and development of significant components of the RPPT, the program-driven simulation tool mentioned earlier. These components are: 1) Concurrent C, an extension of C for parallel programming, 2) a library of primitives for the simulation of concurrent constructs in an architecture independent manner, and 3) TRAPP, a window-based tracer for parallel programs.

1.6. Overview of Thesis

The rest of the thesis is organized as follows. In Chapter 2 we discuss previous research in the area of performance evaluation of parallel algorithms and systems as it relates to our work. Our emphasis is on research that deals with performance prediction of parallel algorithms with non-deterministic task execution times. We give an overview of the Rice Parallel Processing Testbed in Chapter 3, and discuss some of its components in detail. Chapters 4, 5, and 6 deal with the multiphase, partitioning, and pipeline structures, respectively. In Chapter 4 we derive bounds on the mean execution time of algorithms with a multiphase structure and give some results for two different scheduling disciplines for the case where there are more parallel tasks than processors. In Chapter 5 we derive bounds and approximations for the mean execution time of algorithms with the partitioning structures. We deal with linear and two-dimensional pipeline structures in Chapter 6. Chapter 7 summarizes the major results of this research, suggests possi-

ble extensions of the analysis, and discusses the applicability of the methods developed to other classes of parallel algorithms and to other parallel system models.

CHAPTER 2

Previous Work

2.1. Introduction

Research relating to the performance of parallel algorithms has focussed on four major areas: classification of parallel algorithms, measurement of the performance of specific algorithms on particular parallel systems, analytical modeling of the performance of classes of algorithms, and computational complexity studies. In this chapter, we will describe some of the major results in these areas and, where appropriate, indicate how these results relate to the work presented in this thesis. Other research efforts relevant to the methodologies used in our work, particularly in the areas of simulation tools and parallel programming languages are discussed in Chapter 3.

2.2. Classification of Parallel Algorithms

Parallel algorithms have been broadly classified as asynchronous and synchronous algorithms by Kung [5]. Asynchronous algorithms are those in which processes do not explicitly wait for each other. These algorithms are characterized by the ability of processors to work with the most recently available data. An example of such an algorithm is the asynchronous Jacobi method for the solution of partial differential equations [4]. In this algorithm each process uses the latest values it can find in computing the new values on a grid. Convergence of such algorithms is difficult to prove. Baudet [6] has studied the performance of asynchronous algorithms for multiprocessors. Quinn [7] refers to asynchronous algorithms as relaxation algorithms.

Synchronous algorithms are those in which processes explicitly wait for other processes. Most parallel algorithms have some synchronization requirements. Mohan [3] has used the task graph model of scheduling theory [1] and has developed a classification of parallel algorithms based on the structure of their task graphs or dependency graphs. A task graph of a parallel computation is a representation of the tasks and their relative order of execution. He classifies algorithms into asynchronous, synchronous, multiphase, partitioning, and pipeline algorithms. Mohan's asynchronous class is the same as Kung's asynchronous class of algorithms, and Kung's synchronous class encompasses the rest of the classes described by Mohan.

Multiphase algorithms consist of alternating serial and parallel phases. A single task is active during the serial phase and several tasks are active during the parallel phase. The synchronous class of algorithms described by Mohan is a multiphase algorithm with the execution time of the serial phase set to zero.

Task graphs of algorithms based on a recursive use of the divide and conquer strategy exhibit the partitioning structure. These algorithms typically consist of three phases: a divide phase during which work is divided or partitioned, a work phase during which computation is performed on the partitions, and a merge phase during which results from the previous steps are combined.

Often a computation is divided into a number of subcomputations or stages, with the data for a particular job being processed by each stage in turn in a linear fashion. This simple linear pipeline can be extended into two or more dimensions as in the case of systolic arrays and wave-front arrays. Parallel algorithms of this type are referred to as pipeline algorithms.

Nelson and Snyder [8] have given some programming paradigms or strategies of problem solving for nonshared memory parallel computers. These paradigms, the compute-aggregate-broadcast, divide and conquer, and pipeline and systolic computation are similar to the multi-

phase, partitioning, and pipeline structures respectively.

We use Mohan's classification in this thesis. Since our concern is with algorithms that have synchronization requirements we will not discuss asynchronous or relaxation algorithms any further. We study the rest of the structures described by Mohan in this thesis.

2.3. Measuring the Performance of Parallel Algorithms

The most accurate method to determine the performance of a parallel algorithm on an architecture is to implement the algorithm and measure the performance. This methodology is of course not applicable when the parallel system is unavailable or is yet to be built. The other drawback of this approach is that it may not be possible to obtain required information because of inadequate instrumentation.

Deminet [9] reports the performance of four parallel algorithms, quicksort, partial differential equation solver, network simulator, and fast fourier transform, implemented on a loosely coupled shared memory multiprocessor with 50 processors, the Cm*. These algorithms were chosen to be representative of different classes of applications with different attributes. These attributes were, flexible or fixed number of processes, single or multiple processes per processor, distinguishable or indistinguishable processes, functionally identical or different processes, static or dynamic assignment of work to processes, and shared or private data structures. Deminet concludes that when the structure of an algorithm corresponds well with the structure of the computer, a close to linear speedup may be achieved. Felten *et al.* [10] discuss the performance of a parallel algorithm for solving the traveling salesman problem implemented on a 64-node hypercube. Gehring *et al.* [4] report on a number of experimental studies of parallel algorithms on the Cm*. These include algorithms for integer programming, speech recognition, matrix multiplication, VLSI design rule checking, parallel production systems, among others.

Jones and Schwarz [11] report on their experiences in dealing with three different multiprocessor systems, the C.mmp, the Cm*, and the Pluribus.

Performance measurements, in general, are confined to a specific algorithm on a particular parallel system thus making it difficult to reach conclusions about the performance of all parallel algorithms or at least a class of parallel algorithms. Generalization of results to a class of algorithms requires us to develop performance models.

2.4. Factors Affecting the Performance of Parallel Algorithms

Developing realistic models requires us to identify and incorporate in our models, characteristics of parallel algorithms that have a significant impact on their performance. The performance of a parallel algorithm can be affected by many factors. Link and Agerwala [12] discuss the importance of interprocess communication in developing realistic models of parallel algorithm performance. Levitan [13] claims that communication behavior of the machine-algorithm system is the dominant factor in run time performance. Mohan [3] identifies some attributes of parallel programs that have significant effect on performance and studies these in detail. They are logical and physical parallelism, task granularity, synchronism, resource sharing, and parallel structure. Physical parallelism is the number of processors available for execution while logical parallelism is the maximum number of parallel streams in a computation, *i.e.*, the maximum number of processors that can be gainfully employed. Granularity refers to the size of the elements that constitute the computation. Task granularity in particular refers to the the average execution time of a task. Synchronism refers to the relation in time between the tasks in a program. In Mohan's terminology a program is considered synchronous if its tasks (roughly) execute in lock step. Resource sharing refers to the sharing of system resources, both hardware and software, by the different processes. By parallel structure Mohan means the task graph structure of the program.

Barlow [14] discusses three features considered important to the performance of a parallel algorithm. These are the degree of parallelism in the program, access to shared data space, and access to synchronization tool and protected data structures. Jamieson [15] gives six important characteristics of parallel algorithms that have a significant impact on the performance. These are, nature of parallelism, degree of parallelism, uniformity of operations, synchronization requirements, static/dynamic character of the algorithm, and data dependencies.

The important attributes that recur in the work of other researchers are the degree or amount of parallelism (logical parallelism in Mohan's parlance), synchronization requirements between tasks, and communication costs. Our concern is with the synchronization requirements inherent in the structure of the algorithm. In most of the analysis in this thesis, the logical parallelism or the maximum number of tasks that can be executed simultaneously is assumed to be less than or equal to the physical parallelism or the number of available processors. We also ignore communication costs in our work by assuming that they can be incorporated into task execution times.

2.5. Queueing and Petri Net Models

Direct measurement of the performance of an algorithm is not always feasible. In such circumstances one has to resort to either modeling or simulation to determine the performance of the algorithm. Petri Net models and queueing networks have been used by a number of researchers to model the performance of parallel computer systems in general and parallel algorithms in particular. Marsan *et al.* [16] use generalized stochastic Petri Nets or GSPNs to model the synchronization of tasks in single and two-processor systems. Heidelberger and Trivedi [17, 18] develop queueing models and give approximate solution methods for parallel processing systems in which jobs subdivide into two or more tasks at some point during execution. Thomasian and Bay [19] present a procedure to determine the mean completion time and related

performance measures for a set of tasks with precedence relationships in their execution sequence. Towsley *et al.* [20] develop approximate queueing models for internal parallel processing by individual programs in a multiprogrammed system.

2.6. Accounting for Non-Determinism

The execution times of tasks for some algorithms are deterministic *i.e.*, they are either fixed or can be predicted with a simple analytical relation. Under such circumstances it is relatively easy to evaluate their performance. We will refer to models for such algorithms as deterministic models.

Deterministic models for predicting multiprocessor performance on iterative algorithms are developed by Vrsalovic *et al* [21, 22]. Each iteration of the algorithm consists of some amount of access to global data and some amount of local computation. They consider the manner in which the access and local computation change when a task is partitioned into subtasks. This decomposition or change need not necessarily be linear. In particular, they discuss the effects of several non-linear decompositions on the performance of the iterative algorithm. Their analysis agrees well with the results for three algorithms running on the Cm*.

Predicting the performance of parallel algorithms becomes more difficult when non-determinism has to be taken into account. Several researchers have drawn on results from order statistics in dealing with algorithms which have non-deterministic task execution times. Dubois and Briggs [23] develop approximate analytical models to predict the performance of synchronized iterative algorithms and use these to compare the effectiveness of three typical multiprocessor systems. They study the effect of synchronization, memory conflicts, and interprocessor communication. Each iteration of a synchronized iterative algorithm consists of several processes computing in parallel. All the processes get new values and synchronize before they

start on the next iteration. These algorithms are similar to the multiphase structures we will be considering. Briggs and Dubois do not consider the effect of scheduling strategies or the effect of reduced physical parallelism on the performance. We will deal with both these issues when analyzing the multiphase algorithms in Chapter 4.

Robinson [24] gives upper and lower bounds on the average execution time of a task graph when the task execution times are non-deterministic. His bounds are applicable under the assumptions that task execution times at a particular level in the task graph are i.i.d. random variables, and that there are enough processors to execute all ready tasks. The architectural model used by Robinson is a multiprocessor with identical processors. Communication and contention times are either ignored or assumed to have been incorporated into the task execution times. We will use the bounds developed by Robinson to evaluate the performance of our techniques. Robinson's approach is discussed in more detail in Chapter 5 when dealing with partitioning algorithms.

Weide [25] gives an analytical model to explain the anomalous behavior of certain types of algorithms. This behavior, observed in practice on the C.mmp, indicates that in the absence of overheads, it is beneficial to create more tasks than there are processors under a round robin scheduling discipline. Weide deals with algorithms with the property that they complete execution when all independent subtasks complete execution, *i.e.* synchronized iterative or multiphase algorithms. We will later show that our results for multiphase algorithms under different scheduling disciplines complement Weide's results.

Mohan [3] classifies parallel algorithms based on the structure of their task graphs and studies several common structures in detail. These structures include the asynchronous, synchronous, multiphase, partitioning, and pipeline structures. His results are obtained using a hybrid simulation tool PEP that is capable of simulating the execution of an arbitrary task graph.

Mohan studies effect of various attributes of parallel algorithms such as parallelism, task granularity, synchronism, and resource sharing. His results are based on simulation and he does not provide any analytical results for the case when the task times are non-deterministic.

Kruskal and Weiss [26] analyze the problem of allocating tasks to a multiprocessor to minimize the execution time for multiphase like structures. They assume that the task times are i.i.d. with an increasing failure rate function and derive the optimum number of tasks to be allocated at a time to each processor. Under certain assumptions on the task execution time distributions and scheduling overheads we get similar results as will be seen later in Chapter 4.

2.7. Summary

This thesis is an extension of the results by Mohan [3] and Robinson [24], and a corroboration in part of results by Weide [25] and Kruskal and Weiss [26]. We will use Mohan's classification and extend his simulation based study by giving analytical results for the case when the task execution times are non-deterministic. Robinson's work is applicable for general task graphs. We will show that in general it is possible to obtain sharper bounds and approximations than those obtained by Robinson, when the task graph structures are regular. Analytical results for the performance of partitioning and pipeline structures for the case when the task times are non-deterministic have not been reported in the literature. The multiphase type of synchronization is the simplest type of synchronization possible and has been studied by other researchers. Our results for the multiphase will in general complement those obtained by previous researchers.

CHAPTER 3

Rice Parallel Processing Testbed

3.1. Introduction

Analytical models need to be validated before they can be relied on. The best way to validate a model is of course to compare the predictions obtained from the model with results from an existing system. However, validation can be difficult even with the availability of a real parallel system. This could be because of lack of proper instrumentation to measure the parameters of interest or because the act of measurement might perturb the quantities being measured. In the absence of an existing system or because of difficulties with measurement one has to resort to simulation to predict the performance of complex systems.

Validation of the performance models we develop in this thesis requires knowledge about probability distributions of execution times. Measurements on real parallel systems without adequate hardware instrumentation would perturb these times and would also effect the execution time for the entire algorithm. In addition, real systems will not offer the flexibility in terms of reconfiguration as will software based simulation tools. For these reasons, we chose to validate our models with stochastic simulations and with the Rice Parallel Processing Testbed (RPPT) [2]. The RPPT is a software based simulation tool for the performance evaluation of parallel algorithms on parallel architectures. We describe the RPPT in this chapter.

3.2. Simulation Techniques

Two widely used approaches to the simulation of computer systems are distribution-driven and trace-driven discrete event simulation. Distribution-driven simulations, which are also

known as stochastic simulations, use a statistical model of the programs to be executed to drive the simulation. The premise of this methodology is that the branching and other data-dependent behavior of a program can be modeled as a random process. The main advantage is that the distribution-driven simulations are faster than simulations based on other approaches. The chief disadvantage is the potential inaccuracy of the results because a random process might not accurately capture the behavior of a complex program.

In trace-driven simulations, a trace of instructions is used to drive a model of the architecture of interest. This trace is typically collected by executing portions of the program to be simulated on a serial computer that is identical to or very similar to the processors in the parallel system. This approach, although more accurate than distribution-driven simulation, is more expensive computationally since execution has to be simulated on an instruction by instruction basis. Another drawback of this approach is that the trace generated by a serial execution might be inappropriate to drive the simulation for concurrent execution [27]. This is because the ordering of events in the trace will only be an approximation to the ordering that will occur on a specific parallel system.

One way to overcome the drawback of using an inaccurate trace is to drive the architectural model with the trace and use feedback from the model to control the generation of trace. This pseudo-concurrent execution of the program on the architecture would accurately reflect the ordering of events. An instruction by instruction simulation would then model the parallel system accurately. An example of this approach, which we will refer to as instruction-driven simulation, is the PSIMUL [28] simulator for simulating parallel programs. Other examples of this approach are reported by Axelrod *et al.* [29], Dubois *et al.* [27], and Williams and Bobrowicz [30]. Instruction by instruction simulation results in a large amount of overhead and simulators based on this approach are slow, increasing the ratio of time to simulate the program to the time to execute it directly by factors of several hundred [28].

The RPPT uses pseudo-concurrent execution of a parallel program to drive the simulation but avoids the high overhead of instruction-by-instruction simulation by directly executing the instructions on the simulation host instead of simulating them. This approach, called execution-driven simulation, results in accurate modeling of the behavior of a parallel program with relatively low overhead. The Computer Architect's Workbench developed at Stanford uses a similar approach, but only in the context of single processor systems [31]

We describe the RPPT in some detail in the rest of the chapter. The next section contains an overview of the RPPT and describes its various modules. The RPPT is implemented on top of Concurrent C [32], an extension of C for parallel programming. Concurrent C is also the parallel language for specifying programs to the RPPT. We describe Concurrent C in Section 3. This is followed in Section 4 by a discussion of some of the design and implementation issues that had to be resolved in building the RPPT.

3.3. Overview of the RPPT

The RPPT is a software tool based on the execution-driven simulation approach for the performance evaluation of parallel algorithms on parallel architectures. Both a program and an architecture are specified by the user. Therefore, the RPPT can be used to investigate the performance of different implementations of an algorithm on a fixed architecture, the relative performance of different architectures executing the same program, or variations in performance over a range of different algorithms and architectures.

The RPPT is composed of the following basic components:

- 1) **Concurrent C:** An extension of the C programming language obtained by the addition of procedures to create and control processes, synchronize processes, and pass information between processes [32].

- 2) **C Simulation Package (CSIM):** A process level discrete event simulator obtained by extending Concurrent C with a set of procedures for event queue handling and statistics collection [33].
- 3) **Architecture Simulation Preprocessor (ASIMP):** A source code translator that modifies Concurrent C programs by inserting simulation primitives used to simulate architecture delays that are due to interprocess communication and synchronization between processes on different processor modules.
- 4) **Timing Profiler (TPROF):** An assembly language program analyzer used to estimate the time required to execute basic blocks of a program, and to modify the program to accumulate these times during runtime.
- 5) **Simulation Primitives:** A set of routines that duplicate the functionality of Concurrent C operations and simulate architecture delays and program execution delays.
- 6) **Simulation Tool Interface (SIMTOOL):** A menu-driven, window-oriented user interface that simplifies the task of combining a program, an architectural model, and a mapping, and allows users to specify program and/or architecture specific parameters [34].
- 7) **Parallel Tracer/Debugger (TRAPP):** A runtime, window-based interface that allows the user to monitor the progress of a simulation [35].

Figs. 3.1 and 3.2 illustrate the relations among these components and the architecture and program supplied by the user. Fig. 3.1 illustrates the preprocessing undergone by a Concurrent C program before an executable simulation model is produced. First, the Concurrent C source program is modified by ASIMP, which replaces calls to Concurrent C primitives by calls to simulation primitives that duplicate the functionality of the original calls and simulate delays due to program execution and interprocessor communication on the interconnection network.

Figure 3.1: Preprocessing undergone by a Concurrent C program

Figure 3.2: User's view of the RPPT

The output of ASIMP is then sent to a standard C compiler to generate assembly language output. The assembly output is then passed through TPROF which estimates the time required to execute straight line segments of code, *i.e.*, basic blocks, and inserts extra instructions at basic block boundaries to accumulate the estimated execution time during runtime. TPROF is an extension and generalization of the BB profiler developed at Bell Labs [36]. The output of TPROF is assembled and linked with precompiled modules of Concurrent C routines, CSIM routines, simulation primitives, and an architectural model to produce the final executable simulation model.

The process of combining the program and architectural modules and libraries is simplified by the use of SIMTOOL, a menu-driven, window-oriented, user friendly interface. Fig. 3.2 represents the user's view of RPPT. The user must select a Concurrent C program and an architecture model from libraries maintained by the RPPT and input program and/or architecture specific parameters to SIMTOOL, which combines these to produce an executable simulation model. The progress of a simulation can be monitored by using TRAPP to display information about important events in the lifetime of the processes, one process per window on a sun workstation. Upon completion of simulation information about the performance of the program and the architecture is output. At present this information includes execution time for the program, utilization information for processors, various categories of waiting times such as *semaphore wait* and *message wait* for processes, and traffic and utilization information for the interconnection network of the architecture.

3.4. Concurrent C

Many languages have been devised to facilitate the programming of parallel systems. Some of these languages, such as CSP, Prolog, and Occam, have been explicitly created for parallel programming and are unrelated to any existing sequential languages. Another approach

to developing parallel languages is to start with a well-known sequential language such as Fortran, Pascal, or C and modify it by adding new data types or constructs. Examples of this approach include Concurrent Pascal [37], Refined C [38, 39], and Parallel-C [40]. More recent examples include the extensions of Lisp and C to program the Connection Machine.

Concurrent C is an extension of the C language for parallel programming. It is a pseudo-concurrent language in the sense that it runs on a uniprocessor while giving the user the illusion of multiple simultaneously executing processes, much like an operating system on a multiprogrammed computer.

Concurrent C provides facilities for dynamic process creation, process control, semaphore handling and synchronous and asynchronous message passing. The extensions to C to provide these facilities are implemented as a set of library routines thus obviating the need for a special compiler for the language. The following is a list of the most important of these routines. Their functions are described below.

Process Creation:

Create, CreateChild, Split, Fork

Parent-Child Synchronization:

Join

Process Control:

Suspend, Activate, Transfer, Nice

Semaphore Handling

P, V, InitSemaphore

Asynchronous Message Passing:

SendMessage, ReceiveMessage

Synchronous Message Passing:

SendRequest, ReceiveRequest, Reply

Logical Nodes:

SetNode, PushNode, PopNode, LogicalToPhysical, WhereIs

Ready List Management:

Block, Unblock

Independent processes can be created in Concurrent C by means of the routines Create(), and Split(). Create() creates a new process but does not start executing it whereas Split() creates a new process and activates it immediately. Dependent processes that are related to the creating process by a parent-child relationship can be created by using CreateChild(), and Fork(). CreateChild() just creates a process whereas Fork() creates and activates a new process, analogous to Split().

Related processes can be synchronized, at the discretion of the user by means of a Join(). A process executing a Join() will not proceed past the Join() until all its child processes have completed execution.

A process can stop its own execution by calling Suspend() and suspended processes can be activated by calling Activate(). The Transfer() routine is a coroutine type of control transfer from the executing process to another process. This routine is the basic transfer mechanism by which process scheduling is implemented in Concurrent C, and hence its use is not recommended in parallel programs. Since scheduling in Concurrent C is not preemptive, control is not transferred to another process until the executing process suspends itself. The Nice() routine is a way to permit another process to get a chance to execute.

Concurrent C provides P() and V() operations to handle semaphores. These are standard operations to acquire and release semaphores typically found in operating systems. Semaphores need to be initialized before use by means of InitSemaphore().

Concurrent C implements both asynchronous and synchronous message passing. The routines SendMessage() and ReceiveMessage() implement asynchronous buffered message passing. A ReceiveMessage() call can be a blocking or a non-blocking call to permit users to check for waiting messages. The routines SendRequest(), ReceiveRequest(), and Reply() implement synchronous message passing. These routines can be used to model synchronous client-server operations typically found in distributed systems such as the V operating system.

A major difference between Concurrent C and other parallel languages is the notion of logical and physical nodes. A physical node can be thought of as a processor with some local memory and capable of running processes (a node of a multiprocessor or a distributed operating system). The concept of logical nodes has been provided to facilitate the writing of programs in an architecture-independent manner. Each process in Concurrent C is associated with a logical node at creation time. This association can be changed at any time to model process migration. The user can give a mapping from logical to physical nodes, and several logical nodes can map to the same physical node. Concurrent C maintains a stack of node associations to permit modeling of temporary process migration by means of the PushNode(), and PopNode() routines.

Concurrent C maintains a list of processes ready to run associated with each physical node and the user can control the availability of these nodes by means of the Block() and Unblock() routines.

As mentioned earlier the extensions to the C language to create Concurrent C are implemented as a set of routines. This implies that users can write normal C programs with embedded Concurrent C calls and use a standard compiler and linker to create an executable pseudo-

concurrent program. Any C routine can be invoked as a process. A stack is maintained for each process in Concurrent C and control transfer between processes is carried out by means of a coroutine type of transfer that switches stack pointers. This stack switching requires the use of a few machine dependent statements and thus Concurrent C is not totally portable. However, machine dependency is small and implementations on standard architectures are relatively straightforward. Concurrent C at present runs on Vax computers, Pyramid systems, and systems such as Sun workstations based on the 68000 family of microprocessors.

3.5. RPPT Design Issues

We now discuss some of the issues involved in the design of the RPPT. Our primary concern in this section is with the implementation of the simulation primitives that account for delays due to program execution and interprocessor communication.

Since the RPPT is meant to be a performance evaluation tool that permits arbitrary architecture-program combinations to be evaluated it is imperative to decouple the architecture and algorithm in a clean way so that different programs could be simulated onto a single architecture or vice versa. This is achieved by implementing all architecture contention, queueing, and data transfer delays in a single routine, `UserSend()`. Simulation primitives detect the need for a remote operation at runtime and make calls to `UserSend()` at the appropriate places. For example, processes that were on the same node do not require the use of the interconnection network for a message passing operation, whereas those on different nodes do. In the later case, simulation primitives detect the fact that processes are on different nodes and call `UserSend()` to simulate the data transfer. Thus modeling a new architecture reduces to implementing a new `UserSend()` routine. This can be as simple as an FCFS queue to model a simplified single bus multiprocessor, or as complex as the multiple queues and processes required to implement a multistage interconnection network.

The RPPT is a simulation tool based on the concept of execution-driven simulation. In this approach estimates of execution time for straight line pieces of code are made at compile time and extra code is inserted to accumulate these estimates during runtime. RPPT simulation primitives use such estimates to advance simulation time to account for time delay due to code execution and to interleave execution of several processes. A basic assumption for RPPT applications is that all interactions between two user processes are explicit, either through access to shared variables or through the invocation of interprocess communication mechanisms. Simulation time, therefore, does not need to be advanced until currently executing process performs such an interaction. Thus the overhead associated with instruction-by-instruction time updates present in instruction-driven simulations is not incurred. This is one of the major reasons for the improved efficiency of the RPPT over such tools.

The cost of implementing parallel constructs or primitives such as process creation is an often ignored overhead in determining the performance of parallel systems using either analysis or simulation. The RPPT models this overhead by profiling the simulation primitives themselves using TPROF to provide timing estimates. These estimates are used along with the timing estimates for user code to advance simulation time. Since there is no reason to assume that these estimates would accurately reflect the implementation of primitives on real systems, the RPPT allows the user to scale and adjust the estimates. Thus, the RPPT allows the user a great deal of flexibility in modeling various operating system overheads. For example, the user can test the sensitivity of the performance of an algorithm to the implementation of a specific operating system primitive on a specific architecture.

The simulation primitives are also an appropriate place to collect statistics on the behavior of processes. Statistics routines have been implemented at present to completely account for the lifetime of a process. Upon completion of simulation, RPPT generates a detailed table of statistics for all processes and nodes. The time a process spent idling is classified into several types,

and it is easy to determine how much time a process is spending waiting for messages, semaphores, etc. Statistics collection about the performance of the interconnection network is done inside the `UserSend()` routine.

3.6. Conclusion

The Rice Parallel Processing Testbed is a software tool based on execution-driven simulation for the performance evaluation of parallel systems. The RPPT can be used to study the implementation of several algorithms on a single architecture or to study the implementation of an algorithm on several architectures and is an efficient tool with slowdown factors of 5 to 10, as opposed to the 200 to 300 often seen in instruction-driven simulators.

The RPPT is currently undergoing extensive validation at Rice University and preliminary results show it to be an accurate tool. Several architectural models such as a hypercube, a LAN based distributed system, and a single bus multiprocessor are available at present and several more are being implemented. A host of parallel algorithms have been written in Concurrent C and tested under RPPT. These include algorithms for sorting, root finding, eigenvalue evaluation, FFT, and matrix multiplication among others.

The author was involved in the design and implementation effort of RPPT. In particular, this involvement included the design and implementation of Concurrent C, the design and implementations of portions of the simulation primitives, the design and implementation of statistics collection routines, and the design and partial implementation of TRAPP.

Validation of performance models is difficult, even with the availability of real parallel systems. The RPPT is a flexible, accurate, and relatively inexpensive tool for determining the performance of parallel systems. We chose to validate our models using stochastic simulation and by execution-driven simulation using the RPPT.

CHAPTER 4

Multiphase Algorithms

4.1. Introduction

A frequently encountered structure in parallel programs is that of a master task followed by several slave tasks. Typically, the master task partitions the work to be done among the slave tasks and waits until all the slave processes have completed execution before starting another iteration. The program is in a serial phase during the execution of the master task and is in a parallel phase during the execution of the slave tasks. The task graph or the dependency graph of such an algorithm has a characteristic structure known as the multiphase structure [3], shown in Fig. 1. In this chapter we analyze multiphase structures with non-deterministic task execution times and derive asymptotic bounds for the average execution time when there is no contention for processors. In addition we derive bounds for the average execution times under static and dynamic scheduling policies.

An example of a multiphase algorithm is the parallel PDE solver using the synchronous Jacobi method. In this algorithm each slave process gets a portion of the grid and computes new values. The master process waits for all the slave processes to finish the current iteration and starts another iteration if global convergence has not been achieved. The kind of synchronization required for multiphase structures is termed barrier synchronization. Algorithms with structures similar to the multiphase structure but with non-existent or negligible master tasks have been variously known as synchronous algorithms [3] or synchronous iterative algorithms [23].

If the execution times of the tasks are deterministic the computation of the execution time of a multiphase algorithm is trivial. However, task execution times in real programs are often

Figure 4.1: Task graph for a multiphase structure

non-deterministic and are best modeled probabilistically. This may be because of queueing delays due to contention for resources such as memory or communication channels, or because of data-dependent computation times. It is important to account for such non-determinacies in developing realistic models of parallel algorithms.

Our aim is to determine of effect of multiphase synchronization on the performance of parallel algorithms. We will show that, given some information about the probability distributions of the task execution times times, it is possible to bound or approximate the average execution time of the multiphase structure under certain independence assumptions.

The rest of the chapter is organized as follows. In Section 2 we describe the architectural model for our analysis and state the assumptions required to ensure a solution. We then describe previous research related to performance analysis of multiphase like structures in Section 3. This is followed in Section 4 by the analysis of the multiphase structure for the case when there is no contention for processors. Here we derive upper and lower bounds on the mean execution time of a multiphase structure. Next in Section 5 we consider the case when there is contention for processors and consequently there is a need for scheduling policies. We derive bounds on the mean execution time for two different policies, static scheduling and dynamic scheduling. In addition, we derive the optimum number of subtasks required to minimize bounds on mean execution time under dynamic scheduling when there is scheduling overhead. In Section 6 we evaluate the bounds and derivation of optimum subtasks using simulation. We conclude in Section 7 with a summary and a discussion of how our results relate to those obtained by previous researchers.

4.2. Model and Assumptions

Our architectural model is a simple multiprocessor with n identical processors operating asynchronously and in parallel. Each processor can execute a single task at a time and a task once started runs to completion without interruption. We make the following assumptions about tasks.

- 1) Communication and contention costs are either negligible or are incorporated into the task execution times.
- 2) The execution time of a task is a random variable, with known probability distribution or at least known mean and variance.
- 3) The execution times for tasks are independent of each other and the execution times for all the slave tasks are independent and identically distributed (i.i.d.).
- 4) When a task is partitioned into n subtasks, the mean and variance of the subtasks are smaller by a factor of n .

Assumption 1 is necessary to isolate the effect of non-deterministic execution times on synchronization delays. It may not be possible to have complete information about the probability distribution of a task execution time, but the mean and variance can often be estimated from performance measurements when dealing with real systems. The assumption that all the slave tasks are i.i.d. is usually justified since the slave tasks typically do similar computations albeit on different data. Assumption 4 is reasonable for programs where the subtasks each operate on an equal share of the data and the execution time is linearly related to the amount of data handled.

4.3. Previous Work

Mohan [3] classified parallel algorithms based on the structure of their task graphs and studied the performance of several important canonical structures using a hybrid simulation tool. Mohan studied the multiphase structure in particular but did not give any analytical results for the case when the task execution times are non-deterministic. Dubois and Briggs have developed analytical models to predict the performance of synchronized iterative algorithms and use these to compare the effectiveness of three different architectures. They do not consider the case when there is contention for processors [23]. Weide [25] has studied the anomalous behavior of multiphase like algorithms under a processor sharing scheduling discipline. He concludes that when task switching overhead is small or non-existent it is beneficial to create more tasks than there are processors. We will later show that our results under different scheduling disciplines complement Weide's results. Kruskal and Weiss [26] have analyzed the problem of allocating tasks to a multiprocessor to minimize the execution time for multiphase like structures. They assume that the task times are i.i.d. with an increasing failure rate function and derive the optimum number of tasks to be allocated at a time to each processor. We will later show that under the assumption of normally distributed task execution times we obtain similar results.

4.4. Bounds on Mean Execution Time

We start our analysis under the assumption that there is no contention for processors. If a task is ready to execute it will always find a free processor *i.e.*, the number of processors n is greater than or equal to the number of slave tasks. Under these conditions there is no need for a scheduling policy. We first derive a simple lower bound on the mean execution of the multiphase structure. This is followed by the derivation of a distribution-free upper bound and distribution-specific upper bounds for some common distributions.

4.4.1. Lower Bound

Let the number of iterations in the algorithm be I . Let the execution time of the serial phase for the i th iteration be a random variable S_i and let the execution time of the parallel phase on a single processor for the i th iteration be a random variable P_i . Assuming that the $\{S_i\}$ are identically distributed and that the $\{P_i\}$ are also identically distributed, the average execution time on a single processor is given by

$$E[T_1] = I(\mu_s + \mu_p)$$

where μ_s and μ_p are the mean execution times of the serial and parallel phases, respectively. In the parallel algorithm we partition each parallel phase into n tasks such that the execution time for the j th task in the i th iteration is a random variable Y_{ij} with the mean and variance given by (from our additional assumption)

$$\mu_y = \frac{\mu_p}{n} \quad \sigma_y = \frac{\sigma_p}{n^{1/2}}$$

The execution time for the entire algorithm on n processors is given by

$$T_n = \sum_{i=1}^I \left\{ S_i + \max(Y_{ij} \mid 1 \leq j \leq n) \right\}$$

Taking expectations we get

$$E[T_n] = I \mu_s + I E[\max(Y_{ij} \mid 1 \leq j \leq n)] \quad (4.1)$$

For any set of random variables $\{X_i\}$

$$E[\max(X_i)] \geq \max(E[X_i]) \quad (4.2)$$

Using (4.2) on (4.1) we get

$$E[T_n] \geq I \mu_s + I \frac{\mu_p}{n}$$

Defining speedup as the ratio of the average execution time on one processor to the average execution time on n processors, *i.e.*,

$$Speedup = \frac{E[T_1]}{E[T_n]}$$

we get

$$Speedup \leq \frac{\mu_s + \mu_p}{\mu_s + \frac{\mu_p}{n}}$$

which is always bounded from above by n .

4.4.2. Distribution-Free Upper Bound

The lower bound on the mean execution time derived from Eq. 4.1 used a rather weak inequality about the behavior of the expectation of maximum of several random variables. Under the assumption that the set of random variables is i.i.d. it is possible to use stronger results from order statistics to place an upper bound on the mean execution time. If m i.i.d. random variables $X_1, X_2, \dots, X_m$ have mean μ and standard deviation σ then (see pages 57-59 of [41])

$$E[\max(X_i)] \leq \mu + \frac{(m-1)}{(2m-1)^{1/2}} \sigma \quad (4.3)$$

Using (4.3) on (4.1) we get

$$E[T_n] \leq I \mu_s + I \left\{ \frac{\mu_p}{n} + \frac{n-1}{(2n-1)^{1/2}} \frac{\sigma_p}{n^{1/2}} \right\}$$

This bound is independent of the nature of the task execution time distribution and is tight in the sense that there exist distributions that attain the bound. In the absence of additional information about the random variables this bound cannot be improved.

4.4.3. Upper Bounds for Specific Distributions

In obtaining the upper bound in the previous section we assumed that we had information about the mean and variance of task execution times in the parallel phase. With additional information about the nature of the probability distribution of the task execution times it is possible to improve on this bound. We now use results from extreme value theory [42] to derive asymptotic

upper bounds for three commonly used distributions: exponential, uniform, and normal.

4.4.3.1. Exponential Distribution

If m random variables X_i are i.i.d. with an exponential distribution

$$E[\max(X_i)] \approx \mu_x (\log m + \gamma) \quad (4.4)$$

where μ_x is the mean and γ is Euler's constant (0.5772...). This is an asymptotic approximation that becomes increasingly accurate for large values of m . Using this we obtain

$$E[T_n] \approx I \mu_s + I \left\{ \frac{\mu_p}{n} (\log n + \gamma) \right\}$$

and

$$Speedup \approx \frac{\mu_s + \mu_p}{\mu_s + \frac{\mu_p}{n} (\log n + \gamma)}$$

If the serial phase has a negligible execution time this reduces to

$$Speedup \approx \frac{n}{\log n}$$

This limitation on the speedup is entirely due to synchronization requirements inherent in the algorithm and is not an artifact of implementation penalties on a particular architecture.

4.4.3.2. Uniform Distribution

If m random variables X_i are i.i.d. with a uniform distribution between a and b , $a < b$

$$E[\max(X_i)] \approx b - \frac{b-a}{n} \quad (4.5)$$

This gives us

$$Speedup \approx \frac{\mu_s + \frac{n(b+a)}{2}}{\mu_s + b - \frac{(b-a)}{n}}$$

In the absence of a serial phase this reduces to

$$\frac{n(b+a)}{2b}$$

which is linear in n . For the special cases where a is zero or $b \gg a$ we get

$$Speedup \approx \frac{n}{2}$$

4.4.3.3. Normal Distribution

If m random variables X_i are i.i.d with a normal distribution

$$E[\max(X_i)] \approx \mu_x + \sigma_x \left\{ (2 \log m)^{1/2} - \frac{\log \log m + \log 4\pi}{2(2 \log m)^{1/2}} + \frac{\gamma}{(2 \log m)^{1/2}} \right\} \quad (4.6)$$

Using 4.6 we get

$$E[T_n] \approx \mu_p + \frac{\sigma_p}{n^{1/2}} \left\{ (2 \log n)^{1/2} - \frac{\log \log n + \log 4\pi}{2(2 \log n)^{1/2}} + \frac{\gamma}{(2 \log n)^{1/2}} \right\}$$

In the absence of a serial phase and for large n the speedup can be approximated by

$$Speedup = \frac{1}{c_v} \left[\frac{n}{2 \log n} \right]^{1/2}$$

where c_v is the coefficient of variation of the execution of the parallel phase when run on a single processor. Again, this limitation on speedup is entirely due to the synchronization delays caused by non-determinacies in the task execution times. Kruskal and Weiss [26] get similar results under the assumption that distribution has an increasing failure rate function.

Asymptotic expressions for the mean of the maximum of i.i.d random variables are not available for all distributions. The beta distribution is an example of such a distribution. In such cases we have to rely on the distribution independent generic upper bound derived earlier.

4.5. Scheduling Policies

In deriving the bounds so far we have assumed that there are enough processors available so that tasks ready to run do not have to wait for free processors. We now consider the case

when there are more tasks than processors. There is contention for processors under these circumstances and the manner in which tasks are assigned to processors, *i.e.*, the scheduling policy, affects the performance of the multiphase structure. Tasks can be preassigned to processors statically before runtime or they can be assigned dynamically at runtime. As we mentioned earlier, Weide [25] has analyzed this case for a processor scheduling discipline.

We first derive an upper bound on the mean execution time of the multiphase structure under a static scheduling policy. This is followed by an analysis for a dynamic scheduling policy. We then consider the case of dynamic scheduling under overhead.

4.5.1. Static Scheduling

Let the parallel phase be partitioned into kn tasks where n is the number of processors with the execution time for the i th task being a random variable Y_i . Let the (Y_i) be i.i.d. with the mean and standard deviation of the tasks given by

$$\mu_y = \frac{\mu_p}{kn} \quad \sigma_y = \frac{\sigma_p}{(kn)^{1/2}}$$

We will assume that the number of tasks is a multiple of the number of processors, *i.e.*, k is an integer. In static scheduling each processor is preassigned k tasks before the start of execution. Let Z_j be the sum of the execution times of the k tasks on the j th processor. The execution time for the parallel phase on n processors is given by

$$T_{pn} = \max (Z_j \mid 1 \leq j \leq n)$$

Using the bound represented by (4.3)

$$E[T_{pn}] \leq \frac{\mu_p}{n} + \frac{n-1}{(2n-1)^{1/2}} \frac{\sigma_p}{n^{1/2}} \quad (4.7)$$

For large values of k , the distribution of Z_j will tend to a normal distribution permitting us to use the asymptotic approximation represented by (4.6). This leads to

$$E[T_n] \approx \frac{\mu_p}{n} + \frac{\sigma_p}{n^{1/2}} \left\{ (2 \log n)^{1/2} - \frac{\log \log n + \log 4\pi}{2(2 \log n)^{1/2}} + \frac{\gamma}{(2 \log n)^{1/2}} \right\} \quad (4.8)$$

This expression is independent of k and implies that a static or preassigned scheduling will result in sublinear speedup as long as the task execution times have a nonzero variance.

4.5.2. Dynamic Scheduling

In general, static scheduling does not take advantage of the variations in task execution times to balance the load among the processors. In dynamic scheduling, processors are assigned new tasks from a central pool whenever they become idle. We will assume that there is no execution time overhead because of runtime allocation and derive upper bounds on the average execution time under this scheduling policy. We consider the case where there is overhead in the next section.

Again we partition the parallel phase of the algorithm into kn tasks as in the case of static scheduling. However, in this case k need not be an integer. Any idle processor picks up a task, if any are available, from a central pool and starts execution. Let T_f be the time at which the first processor to complete its tasks in a phase becomes idle, let T_l be the time at which the last processor to complete execution becomes idle and let d be the difference between the two.

$$T_l = T_f + d$$

If d is greater than zero some processors must have been idle for some portion of the time out of T_l . Thus

$$n T_l \geq \sum_{i=1}^{kn} Y_i$$

Taking expectations we get

$$E[T_l] \geq \frac{\mu_p}{n}$$

Also

$$nT_f \leq \sum_{i=1}^{kn} Y_i$$

No new tasks could have started execution after T_f because if there were any new tasks to be executed they would have been scheduled on the processor that became idle at T_f . The difference d is less than the maximum of the $n-1$ tasks that were running on the other processors at time T_f .

$$d \leq \max (Y_i \mid 1 \leq i \leq n-1)$$

which results in

$$T_l \leq \frac{\sum_{i=1}^{kn} Y_i}{n} + \max (Y_i \mid 1 \leq i \leq n-1)$$

Taking expectations and using the bound represented by Eq. (4.3) we get

$$\frac{\mu_x}{n} \leq E[T_l] \leq \frac{\mu_x}{n} \left[1 + \frac{1}{k} \right] + \frac{n-2}{(2n-3)^{1/2}} \frac{\sigma_x}{(kn)^{1/2}} \quad (4.9)$$

For large k the upper bound approaches the lower bound, indicating a linear speedup. Thus fine grain task size with no scheduling overhead can give near linear speedup under a dynamic scheduling policy.

The upper bound can be made tighter if the distribution of the $\{Y_i\}$ is known. For example, if the $\{Y_i\}$ are distributed exponentially we can use (4.4) to obtain

$$E[T_l] \leq \frac{\mu_x}{n} + \frac{\mu_x}{kn} (\log (n-1) + \gamma) \quad (4.10)$$

If we assume that the task execution times are i.i.d. with a normal distribution, it is possible to use (4.6) to derive an upper bound on d to obtain results which are very similar to those obtained by Kruskal and Weiss [26].

We can also derive a different upper bound based on the following approach in which we assume that k is significantly greater than 1. Consider the time period from T_f to T_l . No more than $n-1$ tasks could have been executed during this period. Thus at least $kn-(n-1)$ tasks must

have completed execution by T_f .

$$nT_f \geq \sum_{i=1}^{kn-n+1} Y_i$$

Taking expectations we get

$$E[T_f] \geq \frac{1}{n} \frac{\mu_x}{kn} (kn-n+1)$$

The difference d between T_l and T_f will be maximum if all the work during that period is done on a single processor. This gives

$$d \leq \sum_{i=1}^{kn} Y_i - n T_f$$

From this we get

$$T_l \leq \sum_{i=1}^{kn} Y_i - (n-1) T_f$$

Taking expectations and using the lower bound on $E[T_f]$

$$E[T_l] \leq \mu_x \left[1 - \frac{(n-1)(nk-n+1)}{n^2 k} \right] \quad (4.11)$$

This bound is not very tight for small values of n or k but otherwise is an improvement over the general bound as will be seen in the next section.

We now derive an approximation for the mean execution time. If we assume that the completion times for the processors are uniformly distributed between T_f and T_l we can make the following approximation for T_l

$$n T_f + n \frac{d}{2} = \sum_{i=1}^{kn} Y_i$$

Thus

$$T_l = T_f + d = \frac{\sum_{i=1}^{kn} Y_i}{n} + \frac{d}{2} \quad (4.12)$$

This is neither an upper nor a lower bound, but we expect it to be increasingly accurate as k gets

large.

4.5.3. Dynamic Scheduling with Overhead

So far we have assumed that partitioning the parallel phase into tasks would result in the work being shared among all the tasks with no overhead. In reality, partitioning usually entails some overhead. This could be due to a number of reasons such as initialization required for each task, duplicated computations among the tasks, and queueing delays due to contention for resources by several tasks. With a constant parallelization overhead of c for each task, the mean execution time per task is now given by

$$E[Y_i] = \frac{\mu_p}{kn} + c$$

which leads to the distribution-free bound

$$E[T_i] \leq \frac{\mu_x}{n} \left[1 + \frac{1}{k} \right] + \frac{n-2}{(2n-3)^{1/2}} \frac{\sigma_x}{(kn)^{1/2}} + kc + c \quad (4.13)$$

Increasing k reduces the difference d relative to T_i thus improving speedup. On the other hand an increase in k increases the execution time because of the overhead per task. To find the optimal k that will minimize the upper bound we can take the first derivative of the bound with respect to k and set it to zero, giving us

$$c - \frac{\mu_x}{k_{opt}^2 n} - \frac{\sigma_x}{8^{1/2} k_{opt}^{3/2}} = 0 \quad (4.14)$$

The solution to the above equation gives the optimum value of k that minimizes the upper bound. Note that this does not necessarily minimize the average execution time.

The optimum value of k is easier to derive when the $\{Y_i\}$ have a distribution whose extreme value behavior is analytically expressible as in Eqs. (4.4), (4.5), and (4.6). For examples, if the $\{Y_i\}$ are distributed exponentially

$$k_{opt} = \left\{ \frac{\mu_x (\log(n-1) + \gamma)}{n c} \right\}^{1/2} \quad (4.15)$$

If overhead is taken into account in deriving the bound represented by Eq. (4.11), we get

$$k_{opt} = \left(\frac{n-1}{n} \right) \left(\frac{\mu_x}{c} \right)^{1/2} \quad (16)$$

In the next section, we will compare the predicted values of k_{opt} which minimize the bounds on the expected execution times with the values at which the mean execution time itself is minimized.

4.6. Evaluation

In evaluating the execution time bounds for dynamic scheduling with constant partitioning overhead, we are interested not only in how close the bounds are to the average execution time, but also in how close the values of k that minimize the bounds are to the value of k that minimizes the average execution time. We used distribution-driven simulation to find the average execution time as a function of k and also the value k at which the minimum occurred. We assumed $n=10$, exponentially distributed parallel phase execution times with $\mu_x=1.0$, and $c=0.01$. Fig. 4.2 shows the simulation results as well as the bounds predicted for this case.

The tightest bound is the distribution-specific bound. This will be true in general since the bound is based on more information about the distribution than any of the other bounds. The next best bound is the one given by Eq. (4.11) after correction for overhead. The distribution-free bound based on just the mean and variance of the task times gives the poorest performance. The optimum values of k were computed as in Eqs. (4.14), (4.15), and (4.16). The distribution-specific bound predicts a k_{opt} of 5.25 which is in excellent agreement with the actual optimum value of 5 given by simulation. The bound based on Eq. (4.11) predicts a k_{opt} of 9 while the distribution-free bound predicts a k_{opt} of 11.36.

mean = 1.0; c = 0.01; n = 10;

Optimum K for Distribution Specific bound = 5.25

Optimum K for LargeK bound = 9.00

Optimum K for Distribution Free bound = 11.36

Actual optimum K by simulation = 5

Figure 4.2: Evaluation of bounds for dynamic scheduling with overhead

Simulations for other values of n , μ_p , and c , indicate that this is the typical behavior to be expected of these bounds. In general, the distribution-specific bound is very tight and predicts the optimum value of k to within one. The other two bounds are loose and in general overestimate the optimum value of k , especially for large values of μ_p and small values of c .

4.7. Conclusion

Multiphase algorithms with non-deterministic task execution times show sublinear speedup due to the synchronization requirements inherent in the multiphase structure. This performance degradation is independent of any penalties that could result due to implementation on a specific architecture.

We have derived upper and lower bounds for the mean execution times of multiphase algorithms when the mean and variance of the task execution times are known and there are at least as many processors as parallel tasks. We have also derived approximations to the mean execution time that are asymptotically accurate for large n . When the number of tasks in a parallel phase exceeds the number of processors available, scheduling policies are of importance. We have derived upper and lower bounds for the mean execution time for static and dynamic scheduling.

We have shown that under dynamic scheduling without overhead it is possible to get near linear speedup by partitioning the the work for the parallel phase into a large number of sub-tasks. These results complement those obtained by Weide [25] who concludes that with a processor sharing scheduling policy it is better to have more tasks than processors. He also concludes that with a processor sharing discipline if the task switching overhead is below a certain value it is better to create as many processes as possible. Our analysis shows that when there is a scheduling or a partitioning overhead for dynamic scheduling there is an optimum number of

tasks that minimizes the bounds on the average execution time and in general the average execution time itself.

CHAPTER 5

Partitioning Algorithms

5.1. Introduction

A common approach to solving problems is to partition the problem into smaller parts, find solutions for the parts, and then combine the solutions for the parts into a solution for the whole. This divide-and-conquer strategy, applied recursively, is the basis for several classes of parallel algorithms, including a number of sorting and searching algorithms. These algorithms typically consist of three phases: a divide phase during which work is partitioned, a work phase during which computation is performed on the partitions, and a merge phase during which results from the previous steps are combined. Task graphs of such algorithms have a characteristic structure known as the partitioning structure [3]. In this chapter we analyze partitioning task structures with non-deterministic task execution times and present bounds and approximations for the mean execution time of such structures.

A classic example of an algorithm with the partitioning structure is the quicksort algorithm which partitions an array of elements to be sorted into two subarrays, each of which is subdivided recursively until the number of elements in a subarray is below a threshold. The work phase consists of sorting the elements in the subarray. The merge phase is either non-existent (if the partitioning and sorting are done in place), or trivial (if the partitioning and sorting are done on copies). The mergesort is a similar algorithm with a non-existent or trivial divide phase and non-trivial work and merge phases.

Fig. 1 shows a two-stage partitioning task graph structure. Each node in the graph represents a computational task and each edge represents a dependency between tasks. A task a

Figure 5.1: Task graph for a partitioning algorithm

is said to be a predecessor of a task b if there is a directed edge from a to b . Tasks without predecessors are called initial tasks and tasks that are not the predecessors of any task are called final tasks. A task cannot start until all its predecessor tasks have completed execution and once started a task runs to completion without interruption. The level of a task is the length of the longest path from an initial task to that task. The execution time for the graph is the time from the start of an initial task to the completion of all the tasks. The number of stages in a partitioning structure is the number of divide levels or the number of merge levels. The branching factor is the number of successors to each divide task or equivalently the number of predecessors to each merge task. Many algorithms have a small constant branching factor, usually two or three. The task graph in Fig. 1 has two stages, five levels and a constant branching factor of two.

If the execution times of each of the tasks in a partitioning structure are deterministic, the computation of the execution time for the entire graph is trivial. However, the task execution times in real programs are often non-deterministic because of queueing delays due to contention for resources such as memory or communication channels, and because of data-dependent computation times.

Non-deterministic execution times generally result in synchronization delays where one task has to await the completion of other tasks. Synchronization delays and communication costs are considered to be the most important factors effecting the performance of parallel algorithms [12]. Our goal is to determine the effect of non-deterministic task execution times on the total execution time of the algorithm. We will show that, given information about the nature of task execution times, it is possible to make accurate statements about the mean execution time of a parallel algorithm with the partitioning structure by drawing on results from extreme order statistics.

The rest of the chapter is organized as follows. In the next section we describe the architectural model and the assumptions on task execution times on which our analysis will be based. In Section 3 we discuss related previous research. We analyze the partitioning structure in Section 4, where we present five methods for bounding and approximating the mean execution time. In Section 5 we compare the performance of the methods we present with previous approaches and evaluate the accuracy and applicability of the methods using stochastic and program-driven simulations. We use one of the methods developed to derive an analytical expression for the mean execution time of a parallel mergesort algorithm in Section 6.

5.2. Model and Assumptions

The architectural model to be used for our analysis is a simple multiprocessor with n identical processors operating asynchronously and in parallel. Each processor can execute a single task at a time and a task once started runs to completion without interruption. We will assume that there are enough processors so that a task that is ready to execute does not have to wait for a processor, *i.e.*, the maximum number of ready tasks at any time is less than or equal to n . We make the following assumptions about tasks.

- 1) Communication and contention costs are either negligible or are incorporated into the task execution times.
- 2) The execution time of a task is a random variable, with known probability distribution or at least known mean and variance.
- 3) The execution times for tasks are independent of each other and the execution times for tasks at a particular level in the structure are identically distributed (i.i.d.).

Assumption 1 is necessary in order to isolate the effect of non-deterministic execution times on synchronization delays. It may not be possible to have complete information about the

probability distribution of a task execution time, but the mean and variance can often be experimentally estimated from performance measurements when dealing with real systems. The assumption that tasks at a level are identically distributed is usually justified since all tasks at a particular level in algorithms with regular structures do identical computations albeit on different data. However, more often than not tasks at a particular level are not independent of each other. We will later show that violating the independence assumptions is not critical.

5.3. Previous Work

Robinson [24] gave upper and lower bounds for the mean execution of a general task graph under the assumption that the times for tasks at the same level are i.i.d. His bounds are applicable for any task graph provided the mean and variance of the task execution times at each level are known. Using a well known result in order statistics (See Eq. 4.3) which bounds the mean of maximum of several i.i.d. random variables given their mean and variance, Robinson derived

$$E(T_G) \leq \sum_{j=1}^L \left[\mu_j + \frac{m_j-1}{(2m_j-1)^{1/2}} \sigma_j \right] \quad (5.1)$$

where T_G is a random variable denoting the execution time for the general task graph G , m_j is the total number of tasks at level j , μ_j and σ_j are the mean and standard deviation, respectively, of the execution time of a task at level j , and L is the number of levels.

Eq. (5.1) can be interpreted as follows. In a general task graph, tasks at a particular level cannot start execution until their respective predecessors in the previous level have completed execution. With the restriction that the tasks at a level start execution only after *all* the tasks in the previous level have completed execution, an upper bound on total execution time for the task graph can be obtained. Loosely speaking, Robinson's upper bound is the mean execution time of a modified task graph where all tasks at a level synchronize at the end of execution. In the next section we use Robinson's approach to bound the mean execution time for algorithms with

partitioning structures, and we present other methods that provide much tighter bounds.

Mohan [3] classified parallel algorithms based on task graph structures. He used a hybrid simulation tool, PEP, to study these structures. PEP accepts distribution information and an arbitrary task graph structure and determines the mean execution time of the structure. Mohan studied the partitioning structure in particular but did not give any analytical results for the case when the task times are non-deterministic.

5.4. Analysis of Partitioning Structures

We present five methods for bounding and approximating the mean execution time of a partitioning structure. The first uses Robinson's approach for specific distributions. A second bound is based on an expression analogous to (4.3) for dependent variables. We then provide two approximations for the mean execution time based on the number of paths from the initial task to the final task in the partitioning structure. Our last approximation is an iterative technique that takes advantage of the recursive nature of the partitioning structure. All the methods draw on results from extreme order statistics.

5.4.1. Bounds for Specific Distributions In deriving (5.1) Robinson used the distribution independent order statistics bound (4.3) which depends only on the mean and variance of the task execution times. With information about the nature of the task time distributions it is possible to improve on (4.3). For example, if task times at a particular level are i.i.d. with an exponential distribution we can use (4.4) to derive

$$E(T_G) \leq \sum_{j=1}^L \left[\frac{\log n_i + \gamma}{\lambda_i} \right] \quad (5.2)$$

where n_i is the number of tasks at level i , λ_i is the parameter for the execution time distribution of tasks at level i , and L is the number of levels. If the task times are i.i.d. with a uniform distri-

bution we can use (4.5) to get

$$E(T_G) \leq \sum_{j=1}^L \left[b_i - \frac{(b_i - a_i)}{n_i} \right] \quad (5.3)$$

where b_i and a_i are the upper and lower limits of the task time distribution at level i . If the task times are i.i.d. with a normal distribution we can use (4.6) to obtain

$$E(T_G) \leq \sum_{j=1}^L \left[\mu_i + \sigma_i \left[(2 \log n_i)^{1/2} - \frac{\log \log n_i + \log 4\pi}{2(2 \log n_i)^{1/2}} + \frac{\gamma}{(2 \log n_i)^{1/2}} \right] \right] \quad (5.4)$$

where μ_i and σ_i are the mean and standard deviations of the task time distributions at level i . Equations (5.2), (5.3), and (5.4) will not be strict bounds since the order statistics bounds these are based on are not exact but become asymptotically correct for large values of n . However, simulation results show that for all practical purposes these can be used as good upper bounds for values of n as low as 5 to 10.

This approach of using distribution specific order statistics bounds is not applicable for all distributions since some distributions do not have tractable expressions for extreme values. An example is the beta distribution.

5.4.2. Bounds for Dependent Task Times

Frequently, the assumption that the tasks at a level are independent is violated in real programs. For example, in a quicksort algorithm the execution times for the two successor work tasks of a divide task will be negatively correlated since more work for one task would result in less work (a smaller subarray to be sorted) for the other task. Our simulation of a parallel quicksort algorithm showed that tasks at a particular level that were adjacent to each other, in the sense that they were successors of a task in the preceding level, were highly correlated. This correlation fell off as the distance between the tasks increased. Under such circumstances we can use an expression analogous to Eq. (4.3) (see pages 78-79 [41]) that states

$$E[\max (X_i \mid 1 \leq i \leq m)] \leq \mu + (m-1)^{1/2} \sigma \quad (5.5)$$

to give

$$E(T_G) \leq \sum_{j=1}^L \left[\mu_j + (n_j-1)^{1/2} \sigma_j \right] \quad (5.6)$$

where n_j is the number of tasks at level j , L is the number of levels, and μ_j and σ_j are the mean and standard deviation, respectively, of task execution times at level j . This bound gives a higher estimate for the mean execution time than would Eq. (5.1).

5.4.3. Independent Paths Approximation

Both of the above methods, as well as Robinson's bound, apply to general task graphs and do not take advantage of the regularity of the partitioning structure. A partitioning structure has a single initial task, a single final task, and b^s different paths from the initial to the final task, where b is the branching factor and s is the number of stages. Under the assumption that the execution times of tasks at a level are i.i.d., the execution times for all paths are identically distributed random variables with mean and variance given by

$$\begin{aligned} \mu_{path} &= \sum_{i=1}^{2s+1} E[e_i] \\ \sigma_{path}^2 &= \sum_{i=1}^{2s+1} \text{variance}(e_i) \end{aligned}$$

where e_i is the random variable denoting the execution time of a task at level i . If the paths are independent of each other, the execution time of the partitioning algorithm is the maximum of b^s random variables with the given mean and variance. Using (4.3) we get

$$E(T_G) \leq \mu_{path} + \frac{k-1}{(2k-1)^{1/2}} \sigma_{path} \quad (5.7)$$

where $k = b^s$. The assumption that the paths are independent of each other is clearly false since each path shares two or more tasks with every other path. Paths which share a large number of tasks will have highly correlated execution times. Nevertheless, simulation results to be

presented in Section 3.2 show that this can be a significant improvement over Robinson's bound.

5.4.4. Independent Paths Normal Approximation

The independent paths approximation does not make any assumptions about the nature of the distribution of the path execution times. If the task execution times on a path were i.i.d. random variables, the distribution for the execution time for a path (ignoring synchronization delays) would tend to a normal distribution as the number of levels gets large. This leads us to consider approximating the execution time for the entire task graph as the maximum of b^s i.i.d. random variables which are normally distributed. Using Eq. (5.4), we can approximate the expected value of the partitioning algorithm by

$$E(T_G) \approx \mu_{path} + \sigma_{path} \left[(2\log k)^{1/2} - \frac{\log \log k + \log 4\pi}{2(2\log k)^{1/2}} + \frac{\gamma}{(2\log k)^{1/2}} \right] \quad (5.8)$$

where k is the number of different paths.

This approximation will be poor if the number of tasks along a path is small or if the execution time for the path is dominated by a single task. In either case the normal distribution assumption will be invalidated. Nevertheless this approximation is quite accurate as is shown by comparison with simulation results later.

5.4.5. Type I Iterative Approximation

The execution time for an i -stage algorithm can be written as

$$T_i = T_{divide,i} + \max (T_{i-1}, \dots, T_{i-1}) + T_{merge,i}$$

where T_i is a random variable denoting the execution time for an i stage structure and $T_{divide,i}$ and $T_{merge,i}$ are the execution times for the divide and merge tasks from the appropriate levels. This recursion equation suggests an iterative solution to the computation of $E[T_G]$. Let T_0 be the execution time for a work task. The number of terms in the maximum operation is the branching factor of G . If we further assume that the three terms in the expression are independent random

variables we can determine the mean and variance of T_i , if we can compute the mean and variance of $\max(T_{i-1}, \dots, T_{i-1})$.

The expression for T_1 involves taking the maximum of b random variables representing work phase task execution times. If the maximum of several i.i.d. random variables tends to a distribution asymptotically it has to be one of three types of distributions, usually referred to as Type I, Type II, or Type III extremal distributions [42]. Extreme values from distributions with an exponential tail behavior tend to the Type I or Gumbel distribution, those from distributions with a polynomial tail behavior tend to the Type II distribution, and those from bounded distributions tend to the Type III distribution. The exponential and normal distributions are examples of distributions whose maximum values tend to the Type I distribution.

The Type I distribution has the properties that the maximum of n i.i.d. variables from a Type I distribution will remain Type I, and further the distribution of the maximum has the same shape as the distribution of the i.i.d. random variables but is shifted to the right. In particular

$$\mu_n = \mu + \frac{\log n}{\alpha}$$

$$\sigma_n = \sigma$$

$$\alpha^2 = \frac{\pi^2}{6 \sigma^2}$$

where μ and σ are the mean and variance, respectively, of the Type I distribution, μ_n and σ_n are the mean and variance of the extreme value distribution for n variables and α is a shape parameter.

Under the assumption that the b random variables $\{T_0, \dots, T_0\}$ are i.i.d. with distributions whose maximum values tend to a Type I distribution, it is generally a good approximation to say that $\max(T_0, \dots, T_0)$ has such a distribution, particularly as b gets large. If we further assume that both the divide and merge phase task execution times are random variables drawn from either bounded distributions or distributions with exponential tail behavior, then T_1 will also have a

Type I distribution. As long as the $\{T_1\}$ are also independent, we can use similar arguments to compute T_2 , and so forth. The above discussion leads to the following algorithm for iteratively computing an approximation to the mean execution time of an s -stage partitioning algorithm.

begin

$$E[T_0] = E[T_{work}]$$

$$\sigma_0 = \sigma_{work}$$

for $i = 1$ until s do

begin

$$E[T_i] = E[T_{divide_i}] + E[T_{i-1}] + \frac{\sqrt{6} \log b \sigma_{i-1}}{\pi} + E[T_{merge_i}]$$

$$\sigma_i^2 = \sigma_{divide_i}^2 + \sigma_{i-1}^2 + \sigma_{merge_i}^2$$

end

end

The addition of variances follows from the assumption that task times are independent of their predecessor tasks.

The Type I iterative method makes use of the fact that there is a simple relation between the means and variances of a Type I distribution and its extreme value distribution. Unfortunately, such convenient relations are not available for Type II and Type III distributions. In particular the variance of the extreme value of a Type II distribution increases as the number of terms in the maximum operation is increased. The experimental results to be presented in the next section suggest that the iterative method may be useful even when the $\{T_i\}$ are not independent and do not have a Type I distribution.

5.5. Performance Comparison

We evaluated the methods presented above by comparison with simulation results and with Robinson's bound. We first present distribution-driven simulation results for the exponential, uniform, and beta distributions. Results predicted by each of the methods are compared against simulation values to determine the accuracy of the methods. We then present a comparison of three of these methods with simulation results for a quicksort algorithm running on the Rice Parallel Processing Testbed (RPPT). The quicksort algorithm violates the independence assumptions on which all of the methods are based. Nevertheless, as will be seen, all of the methods are at least moderately successful in predicting mean execution times, and the independent paths normal approximation and iterative methods are highly accurate.

5.5.1. Stochastic, Independent Task Execution Times

Results from simulation and analysis were obtained for three different distributions, namely, the exponential, uniform, and beta distributions. In each case the three task types, divide, work, and merge, were assumed to have the same type of distribution and the parameters were chosen such that the mean executions times would differ by an order of magnitude.

Fig. 5.2(a) and 5.2(b) are graphs of the mean execution time as a function of the number of stages for the exponential case. Results obtained from Robinson's bound, the independent paths (IP) method, the independent paths normal approximation (IPN), the Type I iterative approximation, and the simulation are shown. The exponential distribution has a well known extreme value behavior, and the curve labeled R + Dist. in the graph is for results predicted by Robinson's bound when modified by extreme value formula for the exponential distribution. Fig. 5.2(a) gives results for the case where the divide and merge tasks are exponentially distributed with $\lambda = 10$ and the work task is exponential with $\lambda = 1$. Fig. 5.2(b) gives results for the case where all three task types are exponentially distributed with $\lambda = 1$.

(a) Divide = $\exp(10)$, work = $\exp(1)$, merge = $\exp(10)$

(b) Divide = $\exp(1)$, work = $\exp(1)$, merge = $\exp(1)$

Figure 5.2: Comparison of bounds and estimates for exponential distribution

(a) Divide = $\text{unif}(0,1)$, work = $\text{unif}(0,1)$, merge = $\text{unif}(0,1)$

(b) Divide = $\text{beta}(0,1)$, work = $\text{beta}(0,1)$, merge = $\text{beta}(0,1)$

Figure 5.3: Comparison of bounds and estimates for uniform and beta distributions

The following observations can be made from the results. Most methods predict the mean execution time accurately for up to a four-stage structure. Beyond four stages, Robinson's bound diverges rapidly from simulation values. The IP method is better than Robinson's bound but it also diverges. Additional information about the distribution improves Robinson's bound (R + Dist.) considerably. The independent paths normal approximation (IPN) is accurate to within 30 percent in Fig. 5.2(a) and is accurate to within 5 percent in Fig. 5.2(b). This is due to the fact the choice of parameters in Fig. 5.2(a) results in a poor normal approximation to the path execution time since the path time is dominated by a single task, the work task. The Type I iterative method is the most accurate of all and is within 5 percent of the simulation results in both figures.

Figs. 5.3(a) and 5.3(b) give similar results for the uniform and beta distributions, respectively. In both cases parameters for all task times were chosen to be identical. The Type I iterative method is not applicable for either distribution. No results are given for R+Dist. in Fig. 5.3(b) since formulae for extreme value behavior of the beta distribution are unavailable. Both these graphs reinforce our earlier observations. The IPN method again proves to be highly accurate in both cases, predicting the simulation values to within 5 percent.

If a strict (asymptotic) upper bound is required and the task time distributions are known, the method of choice is Robinson's method as modified by distribution information. However, some distributions do not have an asymptotic extreme value distribution, and Robinson's original method must be used. The IPN method requires only the means and variances of task execution times and provides a good approximation even when the normal distribution assumption for path execution times cannot be justified. The Type I iterative approximation is highly accurate but is limited to work phase task execution time distributions whose extreme values converge to the Gumbel or Type I asymptotic distribution.

5.5.2. Program-Driven Task Execution Times

Simulation results were obtained for the quicksort algorithm running on the Rice Parallel Processing Testbed (RPPT) [2]. The RPPT is a software simulation tool that facilitates the performance evaluation of parallel programs on parallel architectures. Parallel programs are analyzed for timing information at the assembly language level. The program then drives an architectural model to provide accurate statistics about resource usage and execution times.

The architecture used for the RPPT simulations described in this section is a single bus, shared memory multiprocessor with enough processors so that ready tasks did not have to wait. During the simulation, statistics on individual task execution times were collected for a task at each level. Communication times were assumed to be negligible.

The partitioning quicksort algorithm works as follows. The divide tasks partition the input array using a median element and start further divide or work tasks depending on the level. Each work task sorts its input array using quicksort. The merge tasks are trivial and simply terminate after informing the next level merge tasks. The number of stages of the algorithm was varied from 1 to 8 and the algorithm was run with sets of random integers to be sorted.

Fig. 5.4 shows the execution times for the quicksort algorithm along with the predictions by Robinson's method, the IPN method, and the iterative method. The tasks at a particular level are not independent and tasks close together in the task graph are highly correlated. Thus the independence assumptions on which all the methods are based fail. However, it is possible to correct for dependence among tasks at a level for Robinson's method using Eq. (5.5). The plot in Fig. 5.4 reflects this. Results for the iterative method are presented even though the distribution types are not known and the independence assumptions are violated. Nevertheless, the iterative method gives results that have accuracy comparable to the IPN method.

Figure 5.4: Quicksort of 8192 integers (RPPT simulation)

Figure 5.5: Mergesort of 8192 integers (RPPT simulation and IPN analysis)

The results show that Robinson's method when corrected for dependencies still bounds the simulation values from above. The IPN and iterative methods are close together and consistently underestimate the mean execution time. This is to be expected since the quicksort algorithm violates the independence assumptions.

5.6. Analysis of a Parallel Mergesort Algorithm

The IPN approximation is a useful method for numerically estimating program execution times. It can also be used as a basis for finding an analytic expression for the expected execution times of parallel programs with a partitioning structure. In this section, we use it to derive an expression for the mean execution time of a parallel mergesort algorithm. Predictions from the analysis are compared against simulation results from a mergesort algorithm running on the RPPT.

The mergesort algorithm is an example of a partitioning algorithm which has no or trivial divide tasks. Each work task sorts a subarray of size $\frac{N}{k}$ where N is the number of elements to be sorted, and k is the number of work tasks. Each merge task accepts two sorted subarrays of equal size from its predecessor tasks, merges them into a single sorted array, and passes it to its successor. Task execution times at a particular level are i.i.d. since all tasks at a level do the same type of work but on different sections of data.

Let the number of work tasks be k , a power of two, and let N be the number of elements to be sorted. Assuming that the time to mergesort an array of size N has mean $a_s N \log(N)$ and standard deviation $b_s N^{1/2}$, and the time to merge two subarrays of size N has mean $2a_m N$ and standard deviation b_m [43], we get

$$\mu_{path} = a_s \frac{N}{k} \log\left(\frac{N}{k}\right) + a_m N \left(1 - \frac{1}{k}\right)$$

and

$$\sigma_{path}^2 = b_s^2 \frac{N}{k} + b_m^2 \log(k)$$

where μ_{path} and σ_{path}^2 are the mean and variance, respectively, of the execution time for a path from a work task to the final merge task. Using the independent paths approximation we obtain

$$E[mergesort] = \mu_{path} + \sigma_{path} \left[(2 \log k)^{1/2} - \frac{\log \log k + \log 4 \pi}{2(2 \log k)^{1/2}} + \frac{\gamma}{(2 \log k)^{1/2}} \right]$$

Fig. 5.5 shows the results from the IPN analysis and from an RPPT simulation of the mergesort algorithm. The coefficients a_m , b_m , a_s , and b_s , were estimated from task time measurements made during the simulation. The analysis and simulation agree to within five percent.

5.7. Conclusion

Partitioning algorithms with non-deterministic task execution times show sublinear speedup due to the synchronization requirements inherent in the partitioning structure. It is possible to use results from order statistics and extreme value theory to predict the average execution times of these algorithms.

Two of the methods we have presented, which are based on Robinson's approach, can be used to bound the mean execution time of a general task graph. The three other methods, the independent paths approximation, the independent paths normal approximation, and the iterative approximation, can be used to approximate the mean execution of a parallel program with a partitioning structure. The IP approximation is empirically shown to be slightly better than Robinson's bound and of a general applicability since it does not require complete information about the probability distribution of task execution times. The IPN approximation gives excellent results and is again of a general applicability as has been shown by our analysis of a parallel mergesort algorithm. The iterative method is very accurate, but is based on task time distributions whose extreme values tend to the Type I asymptotic distribution. However, simulation results show that the iterative method gives good results even when the distributions are

unknown and independence assumptions are violated.

CHAPTER 6

Pipeline Algorithms

6.1. Introduction

A pipeline structure consists of ordered stages through which data passes. Typically, each stage does a nearly identical amount of computation on each data item. Pipeline structures can be found in the execution units of most mainframes and some microprocessors, arithmetic logic units, and memory systems. Pipelining is applicable in the design of parallel algorithms as well, where processes receive data from neighbors, compute on the data, and pass it on to other neighbors. In this chapter, we analyze asynchronous pipeline structures and derive some bounds on their average execution times when the execution times at each stage are non-deterministic.

Pipelining can be synchronous or asynchronous. In a synchronous system each stage or process either finishes the computation in a fixed time or waits for all stages to finish before starting the computation on the next data item. A global synchronization mechanism such as a clock usually controls such systems. Hardware pipelines are usually of this type since they are easier to design and less expensive than systems that employ asynchronous pipelining.

Asynchronous pipeline structures are usually found in parallel algorithms and specialized hardware such as wavefront array processor systems. No global clock is available in asynchronous systems and local synchronization takes place as data moves between neighboring stages. Asynchronous systems usually have some amount of data buffering to improve performance. Asynchronous pipelining when applied to parallel algorithms at a process level has also been called macropipelining [44]. An example of such an algorithm is a pipelined compiler where different processes are responsible for lexical analysis, syntax analysis, semantic analysis, code

generation, and optimization.

The computation time at a stage, especially in software pipelines is seldom fixed. This could be due to queueing delays, contention for resources such as memory or data-dependent computation times, among other factors. Such variations are often best characterized by probability distributions. Thus it is important to be able to account for non-deterministic execution times to develop realistic performance models of pipeline structures.

Although it is possible to design complex pipeline structures, the ones found in practice are usually very simple structures such as the linear pipeline or the two-dimensional array. We analyze the linear pipeline and the two-dimensional array and describe some methods for bounding the mean execution times for these structures with non-deterministic stage execution times. The next section presents the architectural model and states the assumptions on stage execution times on which our analysis is based. In Section 3, we describe some related previous work in this area. In Section 4, we present bounds on the mean execution times of linear pipeline structures and evaluate the applicability and accuracy of these bounds. We deal with two-dimensional pipeline structures in Section 5. We present bounds on the mean execution times of wavefront array processor (WAP) structures in this section.

6.2. Model and Assumptions

As in the case of partitioning algorithms, our architectural model is a simple multiprocessor with n identical processors operating asynchronously and in parallel. We will refer to the computation performed at each stage for each data item as a task. Each processor can execute a single task at a time and a task once started runs to completion without interruption. Our assumptions are as follows.

- 1) Communication and contention costs are either negligible or are incorporated into the task execution times.
- 2) The execution time of a task is a random variable, with known probability distribution or at least known mean and variance.
- 3) The execution times for all tasks are independent of each other and identically distributed (i.i.d).
- 4) There is no contention for processors, *i.e.*, if a task is ready to execute it will always find a free processor.
- 5) There are infinite buffers at each stage, *i.e.*, stages do not have to wait for free buffers before transferring data to their neighbors.

The first three assumptions are the same as those given in Section 4.2 when dealing with multi-phase structures. Most pipeline structures are designed so that each stage does a nearly identical amount of work since imbalances in the amount of work per stage result in bottlenecks and consequently poor performance. Therefore it is natural to assume that the task times are identically distributed. However, task times might not be independent of each other in real systems. The assumption of infinite buffering is the price we pay for obtaining a solution.

6.3. Previous Work

Mohan [3] has classified parallel algorithms based on the structure of their dependency graphs and used a hybrid simulation tool for the study of these structures. One particular structure studied by Mohan is the linear pipeline structure. The main thrust of his work in this area has been to study the effect of physical parallelism or the number of processors available when the tasks in the pipeline have execution times that are fixed, uniformly distributed, and beta distributed. However, no analytical results or bounds are given by Mohan for non-deterministic

task execution times as all of his results are based on simulations. Robinson [24] has given a bound for the mean execution times of parallel computations with dependency graphs that have tasks with identically distributed execution times at each level. This bound is applicable to the linear and the two-dimensional pipeline and is used for comparison with the bounds presented in this report. Melhem [45] gives a bound for the execution time for a wavefront array processor implementing matrix multiplication and discusses a method for obtaining bounds for general algorithms. The execution time for a stage in this case is assumed to be one of two fixed values. Kung [46] *et al.* provide timing analysis of general (cyclic or acyclic), decision-free asynchronous architectures and show that the results can be used to synthesize optimal special purpose hardware implementations of both general dataflow arrays and regular wavefront arrays. The execution times for the cells or stages in this case are assumed to be constant.

6.4. Linear Pipeline Structures

Consider the linear unidirectional asynchronous pipeline with four stages shown in Fig. 6.1. Each stage completes computation on its current data item, outputs the data to its right neighbor, and waits for more data from its left neighbor. Under the assumption that there is sufficient buffering at the input of each stage to receive data, the dependency graph (DG) for this pipeline is as shown in Fig. 6.2. Each node in the DG represents a task and the links represent dependencies between tasks. Each vertical chain of nodes in the DG represents tasks at a particular stage, and each diagonal chain of nodes to the right represents a single data item moving through the pipeline. The DG also shows distinct fill and flush stages at the start and end of the computation.

The execution time for the DG shown in Fig 6.2. is the time from the start of the initial task, the task with no predecessors, to the end of the final task, the task with no successors. If the execution times for the tasks are fixed it is relatively easy to determine the execution time for

Figure 6.1: Four-stage linear pipeline

Figure 6.2: Task graph for a four-stage linear pipeline

the DG. If the task times are random variables, the final execution time will also be a random variable. We present three bounds on the mean execution time of the DG.

6.4.1. Deterministic Time Lower Bound

We first prove a simple lower bound on the mean execution times of general task graphs and then use it to bound the mean execution time of the linear pipeline DG. This lower bound is essentially the lower bound presented by Robinson.

Theorem: Given a task graph G , let T_G be a random variable representing total execution time. Let $c_1, c_2, \dots, c_m$ be all the chains from initial to final tasks in G . Let t_i be a random variable representing the execution time of task i . Then

$$E(T_G) \geq \max_{1 \leq i \leq m} \left[\sum_{T_j \in c_i} E[t_j] \right] \quad (6.1)$$

Proof: The execution time for the task graph is the execution time of the longest chain. Thus

$$T_G = \max_{1 \leq i \leq m} \left[\sum_{T_j \in c_i} t_j \right] \quad (6.2)$$

Taking expectations

$$E[T_G] = E \left[\max_{1 \leq i \leq m} \left[\sum_{T_j \in c_i} t_j \right] \right] \quad (6.3)$$

since for any set of random variables $\{x_i\}$

$$E[\max\{x_i\}] \geq \max\{E[x_i]\} \quad (6.4)$$

we get

$$E \left[\max_{1 \leq i \leq m} \left[\sum_{T_j \in c_i} t_j \right] \right] \geq \max_{1 \leq i \leq m} \left[\sum_{T_j \in c_i} E[t_j] \right] \quad (6.5)$$

Our result follows from Eqs. (6.3) and (6.5).

The above bound states the intuitive result that the execution time for a task graph with non-deterministic execution times is bounded from below by the execution time for the task graph with task times replaced by constants equal to the mean of the corresponding times. In other words, any variance in any task execution time can only increase the mean execution time.

Using the above result we can obtain a lower bound on the mean execution time of the pipeline structure by replacing all task execution times with constants equal to the mean task execution time.

$$E[T_G] \geq (d + s - 1) \mu \quad (6.6)$$

where d is the number of data items sent through the pipeline during the entire computation, s is the number of stages in the pipeline, and μ is the mean task execution time.

This bound will be close if the variance of task execution times is low. As we will show later, it turns out to be a surprisingly good bound for a range of task time distributions and parameters.

6.4.2. Disconnected Chains Lower Bound

For any dependency graph, removing dependency links cannot increase the mean execution time. This can be seen as follows. Tasks for which dependencies have been removed might execute earlier implying that the overall execution time could be decreased or not changed from the case when the dependencies existed. Thus the execution time for the graph with some dependencies removed will be a lower bound on the execution time of the graph with the dependencies in place. We will use this result to obtain a lower bound on the mean execution time of the pipeline structure.

Consider the DG in Fig. 6.3, which is the DG for the linear pipeline structure with dependencies across stages removed. This DG corresponds to the case where stages do not wait for data from their neighbors. The execution time for this DG can be determined by noting that the

Figure 6.3: Modified task graph for linear pipeline

execution time for each chain is the sum of d i.i.d. random variables, and the execution time of the entire DG is the maximum of s such sums. It is reasonable to assume that the execution time for each chain would be normally distributed with the mean and variance being the sum of d identical means and variances, respectively. The maximum of several independent normally distributed random variables tends to the Type I asymptotic distribution. Thus an asymptotic lower bound on the mean execution time of a linear unidirectional pipeline is given by

$$T = d\mu + (d\sigma^2)^{1/2} \left\{ (2\log s)^{1/2} - \frac{(\log \log s + \log 4\pi)}{2(2\log s)^{1/2}} + \frac{\gamma}{(2\log s)^{1/2}} \right\} + (s-1)\mu \quad (6.7)$$

where σ is the standard deviation of execution time distribution for a single task. The factor $(s-1)\mu$ in the bound above is to account for the fact that all columns in the DG do not start execution at the same time. This bound will be good if the number of buffers is large and d the number of data items through the pipeline is large. Under these circumstances stages will not have to wait for data and the execution for a single column will closely approximate the normal distribution.

6.4.3. Independent Paths Approximation

We now derive an approximation to the mean execution time based on the number of different paths from the initial task to the final task. There are $n = C(d+s-2, s)$ different paths from the start task to the end task in the original DG. Each path has $d+s-1$ tasks. The execution time of the DG would be the maximum of the execution times of all these different paths. However, the paths are not independent as each path shares two or more tasks with every other path. If we assume that the paths are independent and further assume that the execution time for each path is normally distributed the execution time for the entire DG can be approximated as follows

$$T = n\mu + (n\sigma^2)^{1/2} \left\{ (2\log n)^{1/2} - \frac{(\log \log n + \log 4\pi)}{2(2\log n)^{1/2}} + \frac{\gamma}{(2\log n)^{1/2}} \right\} \quad (6.8)$$

where n is the number of different paths from start to finish.

The number of different paths explodes combinatorially for large d and s . Nevertheless, simulation results show that this approximation is a reasonable one and is better than an upper bound based on Robinson's approach.

6.4.4. Robinson's Upper Bound

In any DG adding extra dependency links can only increase the execution time. This is because a task with extra links could potentially be delayed more than it would be without the extra links. Thus the execution time for a DG modified by additional links will be an upper bound on the execution time of the original DG. Consider a modified DG where no task at a level i may start until all tasks at level $i-1$ have completed. This is equivalent to the DG for a synchronous pipeline structure. This is the method proposed by Robinson [24] for placing bounds on the mean execution times of general DGs under the assumption that execution times for all tasks at a level are i.i.d. random variables. Thus an upper bound on the mean execution time for a pipeline by the Robinson method is given by

$$T = (d + s - 1) \cdot \max(s \text{ i.i.d. r.v.s}) \quad (6.9)$$

The maximum operation in the above expression would depend on the distribution of the task execution times. Even if the distribution were not known, it is sufficient to know the mean and variance to place a bound on the mean execution time.

6.4.5. Evaluation of Bounds

We now evaluate the bounds presented above using stochastic simulation driven by independent task execution times. Figures 6.4, 6.5, and 6.6 present the simulation results for a linear pipeline. Results for three probability distributions, uniform, exponential, and beta, with two different lengths of the pipeline, 8 and 32 stages, were obtained. In each case the results predicted by the various bounds, Robinson's upper bound, independent paths, lower limit (maximum of disconnected chains), and deterministic lower limit are plotted along with simulation

a) 8 stage linear pipeline

b) 32 stage linear pipeline

Figure 6.4: Comparison of bounds for linear pipeline, uniform distribution

a) 8 stage linear pipeline

b) 32 stage linear pipeline

Figure 6.5: Comparison of bounds for linear pipeline, exponential distribution

a) 8 stage linear pipeline

b) 32 stage linear pipeline

Figure 6.6: Comparison of bounds for linear pipeline, beta distribution

Figure 6.7: 3x3 wavefront array processor

results for various values of d (2, 8, 32, 128, and 512).

As can be seen from the graphs most methods perform about equally in predicting the execution time when d is low (2, 8) with errors ranging from 5 to 20 percent. For large values of d Robinson's method is seen to be poor in comparison with most other methods. The average error for this method is 35 percent. This is to be expected since Robinson's method is meant for general DGs and does not make use of the regularity of the pipeline structure DG.

For large values of d some observations can be made. The execution time predicted by the independent paths method is usually higher than the simulation value. This is due to the fact that we assume that the various paths from start to finish in the DG are independent which results in accounting for some of the computational tasks several times. The average error for this method is 12 percent and the variance is 5 percent.

The lower limit based on the maximum of several disconnected chains is consistently below the simulation results. This method is almost as good as the independent paths method, with an average error of 12 percent and a variance of 9 percent. Surprisingly, the simple lower limit based on a fixed execution time equal to the mean of the execution time is not a bad predictor. As expected it predicts a value below the simulation results and also below the other lower limit method. The average error for this method is 19 percent.

When the distribution is one where the values are close to the mean as in the case of a beta distribution, most methods except Robinson's method are good predictors. The low variance of the distribution in this case causes smaller changes in the result from the deterministic execution time and hence most methods perform about equally. For distributions with larger variances such as the exponential and the uniform, the results from these methods are more varied.

6.5. Two-Dimensional Pipeline Structures

Consider the two-dimensional processor array shown in Fig. 6.7. Each cell waits for data from its top and left neighbors, computes on the data, and sends the new data to its right and bottom neighbors. It is assumed that cells on the left border do not have to wait for input data from the left, and that cells on the top border do not have to wait for data from the top. Likewise, the cells on the right border and the bottom border do not wait to output data. Each cell has a finite number of buffers to hold input data items. There is no global synchronization mechanism and only local synchronization takes place when data is transferred to neighboring cells. Computation and data move in "waves" from the top left corner to the bottom right corner, and the waves of computation may bend as they pass through the array. Such structures are called Wavefront Array Processors (WAP) [45].

The dependency graph for a 3x3 WAP under the assumption that there are sufficient buffers is shown in Fig. 6.8. The DG is very regular and similar to the DG for a linear pipeline except that it is in three dimensions. We present three bounds on the mean execution time of this DG. The techniques we use to derive these bounds are similar to those used in bounding the mean execution times of linear pipeline structures.

6.5.1. Deterministic Time Lower Bound

From Eq. (6.1) we know that a lower bound on the mean execution time of a DG can be obtained by replacing the each task execution time with a constant value equal to the mean of the task execution time. Thus a lower bound on the mean execution time of the two-dimensional pipeline structure is given by

$$E[T_G] \geq \mu (d + x + y - 2) \quad (6.10)$$

where μ is the mean execution time for each task, d is the number of data items pumped through the array, and x and y are the dimensions of the array.

Figure 6.8: Task graph for a 3x3 wavefront array processor

As in the case of the linear pipeline structures, this bound will be close if the variance of the task times is low. Simulation results show this to be a very reasonable bound for a range of distributions and parameters.

6.5.2. Disconnected Chains Lower Bound

As in the case of the DG for the linear pipeline, if the dependency links that model the waiting of cells for data from neighbors are removed, the modified DG can be analyzed to obtain a lower bound on the mean execution time of the original DG. The modified DG will have xy independent chains of tasks. The execution time for each chain is simply the sum of d i.i.d. random variables and can be assumed to be normally distributed for large enough d . The execution time for the entire modified DG is the maximum of the execution times for xy chains and will have the type I asymptotic distribution with the mean given by

$$E[T_G] = xy(d\mu) + (xyd\sigma^2)^{1/2} \left\{ (2\log xy)^{1/2} - \frac{(\log \log xy + \log 4\pi)}{2(2\log xy)^{1/2}} + \frac{\gamma}{(2\log xy)^{1/2}} \right\} + \mu(x+y-3) \quad (6.11)$$

where the last term in the equation accounts for the fact that all chains do not start at the same time. The last term uses a lower bound estimate of the time difference between the start of the first chain and the start of the last chain and hence Eq. (6.11) will still remain a lower bound. Simulation results show that this bound is an excellent lower bound.

6.5.3. Longest Wavefront Upper Bound

We now use a property of the wavefront array processors to compute an upper bound on the mean execution time. In a WAP the wavefronts of computation never intersect but can be irregularly shaped. An upper bound on the execution time can be achieved by imposing the restriction that the wavefronts be regular, *i.e.*, that all the cells that participate in a wavefront complete computation on the current data item before any of them can start work on the next

wavefront. Since extra synchronization requirements are being added, the execution time cannot decrease. Under such circumstances the wavefront that takes the longest time to execute on the average determines the execution time. Without loss of generality let us assume that for a x by y array, x is the larger dimension. The longest wavefront then is x cells long and thus the time to compute one wavefront is the maximum of x i.i.d. random variables with the task time distribution. The total number of wavefronts through the array is given by $(d+x+y-2)$. Thus an upper bound on the mean execution time is given by

$$E[T_G] \leq (d+x+y-2)E[\max(x \text{ i.i.d. r.v.s})] \quad (6.12)$$

The expected value of the maximum of several variables can be asymptotically approximated for certain common distributions such as the exponential, the uniform, and the normal distribution. Even in the absence of information about the nature of the task time distribution or in the absence of analytical expressions for a given distribution, we can bound this value provided we know the mean and variance of the task time distribution.

Simulation results show that this bound, which makes use of the regularity of the task graph structure, is better than the general bound based on Robinson' approach.

6.5.4. Robinson's Upper Bound

Robinson's method is applicable for obtaining an upper bound on the mean execution time of the DG for the two-dimensional pipeline structure. Each level in the DG (other than during the fill and flush periods) has xy tasks which are i.i.d. Thus an upper bound is given by

$$E[T_G] \leq (d+x+y-2) E[\max(xy \text{ i.i.d. r.v.s})] \quad (6.13)$$

As before, the expected value of the maximum can be bounded based on information about the task time distribution. Simulation results show that this bound, which does not make use of the regularity of the DG, is a loose upper bound.

6.5.5. Evaluation of Bounds

We now evaluate the bounds presented above using stochastic simulation driven by independent task execution times. Fig. 6.9 presents the results for a 4x4 WAP for three different distributions: exponential, uniform, and beta. In each case the results predicted by the four methods, Robinson's upper bound, longest wavefront upper bound, lower bound based on disconnected chains, and lower bound based on constant execution time, are plotted along with the simulation results for different value of d , the number of data items through the pipeline. We also obtained simulation results for two different cases: single buffer at the input, and infinite buffers at the input were obtained.

Robinson's bound is seen to be poor for large values of d as can be expected. In all cases the simulation results for a single buffer case were found to very close to and slightly below the results predicted by the longest wavefront method. Also, the simulation results for the infinite buffer case were found to be larger than but close to the results predicted by the disconnected chains lower bound. Surprisingly again, the lower bound based on fixed execution time was quite close to the other lower bound.

6.6. Conclusion

Asynchronous pipeline structures can be found in parallel algorithms as well as special purpose hardware such as wavefront array processors. We have presented some methods for bounding the performance of two structures, the linear pipeline and the two-dimensional WAP, under the assumption that the computation times for each stage or cell are non-deterministic with some known probability distribution.

We have presented simulation results and used Robinson's upper bound as a comparison for these methods. It was observed from the simulation results for the linear pipeline that for

a) Exponential (1.0) distribution

b) Uniform (0,1), distribution

Figure 6.9: Comparison of bounds for 4x4 wavefront array processor

c) Beta(0,1,3,3), distribution

Figure 6.9: Comparison of bounds for a 4x4 wavefront array processor

distributions with low variance and for computations that involve small numbers of data items, most methods perform about the same. For a large number of data items the methods presented are reasonably accurate with about an average error of 12 percent. Surprisingly, simple methods based on fixed execution times also give reasonable results with an accuracy of about 20 percent on the average. Simulation results for the two-dimensional WAP show that the longest wave-front upper bound and the lower bound based on disconnected chains are good bounds for the infinite buffer WAPs.

The method presented here can be readily extended for pipeline structures of larger dimensions. However, most pipelines structures encountered in practice are often of the simple linear and two-dimensional type. One aspect of pipeline structures we have ignored is that of data buffering. Finite buffering introduces complications into the model and would require restrictions on the task time distributions to obtain a solution.

CHAPTER 7

Conclusion

We have endeavored to answer the following questions in this thesis. How does non-determinism effect the performance of parallel algorithms? Is it possible to predict this effect for parallel algorithm in an implementation independent manner? How does one validate such predictions?

Our approach to the problem has been to select three frequently encountered regular task graph structures, the multiphase, partitioning, and pipeline structures. We analyzed these structures using results from order statistics and extreme value theory and derived bounds and approximations for the mean execution times. Validation of our predictions was carried out using distribution-driven simulation and program-driven simulation.

7.1. Results Obtained

Non-determinacies in the execution times of tasks in a parallel computation result in synchronization delays where one task has to await the completion of other tasks. In the case of a task structure that is regular it is possible to quantify the performance degradation due to non-determinism under certain assumptions about the task execution times and the number of processors available to execute the tasks. Our primary assumptions have been that 1) execution times of tasks at a particular level in the task graph are i.i.d. random variables with known probability distributions or known means and variances, and 2) there are enough processors so that tasks that are ready to run do not have to wait for free processors.

In the case of the multiphase structure we showed that non-deterministic task times result in sublinear speedup and derived approximations to the speedup that are asymptotically correct for specific task time distributions, namely the exponential, uniform, and normal distributions. In the absence of information about the nature of the task time distribution we derived distribution-free bounds that depend only knowledge of the mean and variance of the task times. In addition, we derived bounds on the mean execution time for the case when the number of processors is smaller than the number of tasks. Our results showed that it is not possible to improve performance under a static scheduling policy by partitioning the parallel phase into more tasks than there are processors. Our results showed that a static policy of assigning tasks to processors. On the other hand, assigning tasks to processors dynamically at run time results in near linear speedup when the number of tasks in the parallel phase is much larger than the number of processors. We also derived the optimum number of subtasks in the parallel phase of the multiphase structure that would minimize the bounds on the execution time under dynamic scheduling when there is constant scheduling overhead.

For the partitioning structure we derived bounds on the mean execution time for three different task time distributions, the exponential, uniform, and normal distributions. We also derived distribution-free bounds under the assumption that the tasks at a level are dependent. We then took advantage of the regularity of the partitioning structure and used the number of different paths in the structure to derive an upper bound on the mean execution time. A variation of this approach resulted in the independent paths normal approximation that proves to be highly accurate and robust even in the presence of dependencies among the task times. This approximation was used to derive an analytical expression for the mean execution time of a parallel mergesort algorithm. We then derived an iterative approximation for the mean execution time for task time distributions whose extreme values converge to the type I extremal distribution.

We dealt with two pipeline structures, the linear pipeline, and the two-dimensional array. For the linear pipeline we derived lower and upper bounds and approximations on the mean execution time. For the two-dimensional array we derived bounds using methods similar to those used for the linear pipeline. In addition, we derived an upper bound on the mean execution time by placing a restriction on the regularity of the wavefronts in the two-dimensional array.

Other contributions of this thesis include the development of significant components of a program-driven simulation tool, the Rice Parallel Processing Testbed. These components include Concurrent C, simulation primitives, statistics collection routines, and a tracer for parallel programs.

Some general observations can be made about our analysis of these structures. As expected, methods which are based only the means and variances of the task time distributions tend to be less accurate than those that utilize information about the nature of the task time distribution. Also, bounds and approximations from methods that do not take advantage of the regularity of the task graph structures are less accurate than the bounds and approximations from methods that do. A good example illustrating the above observations is the upper bound based on Robinson's approach. The bound, which is meant for general task graphs and depends only on the means and variances of task execution times, is seen to be poor in comparison with every other bound or approximation for the three algorithm classes studied.

Approximations based on the number of different paths in a task graph structure are seen to be good, especially under the assumption that the execution times for the paths are normally distributed. This approach is appropriate for any balanced regular graph where the execution times for all the paths are identically distributed (but not necessarily independent) and the paths are composed of several tasks.

7.2. Discussion

Our analysis of regular task graph structures was based on an MIMD architectural model. However, our results would be valid under other models that consisted of multiple computing elements. For example, in some SIMD systems such as the Connection Machine, it is possible to set up synchronization in such a way that processors start executing simultaneously and wait until all of them finish. An iterative use of this will result in a multiphase-like structure and all our derivations for the case where there are sufficient processors will be valid. In short, whenever multiple computing elements are used in a manner so as to create one of the three task graph structures, our analysis will be applicable. Moreover, the tasks themselves can be as small as single arithmetic or logic operations and need not necessarily be large program segments.

The methods we used for deriving bounds are applicable for other regular task graph structures as well. The notion of multiple independent paths and approximations to the path execution time distribution will be particularly useful for balanced regular structures where all paths essentially have the same length and the task execution times are nearly the same.

The fact that this and other approximations are robust in the presence of large dependencies between adjacent paths seems surprising at first. However, the independence assumptions on which we based our results are stronger than required for extreme value theorems to be valid. In fact the extreme value theorems we used for specific distributions hold under certain types of dependencies as well [41]. In particular, as long as the coefficient of correlation between two different random variables decreases at a rate faster than the logarithm of the distance between them, the extreme value theorems still remain valid. We have not been able to exploit this fact in our analysis, but it may be possible to use this fact for other regular structures.

Strict validation requires implementation and measurement on a real system. Real systems are, however, not readily available and are relatively inflexible. We believe that program-driven

simulation tools such as the Rice Parallel Processing Testbed are relatively inexpensive in terms of computation, extremely flexible in the manner in which they can be configured to test different architectures and algorithms, and accurate enough for our purposes.

7.3. Future Research

Synchronization and communication costs are believed to be the major factors effecting the performance of parallel algorithms. We have dealt with the former but have ignored the latter, assuming that they could be incorporated into the task execution time. A better approach would be to explicitly model communication costs for specific algorithms.

Contention between tasks for shared resources such as memory is another area we have not touched upon. As in the case of communication costs we assumed that these could be incorporated into the task execution times. One possible extension of our work is the use of simple queueing models to derive approximations for contention costs and then incorporate them into task execution times.

The number of processors available is one of the most distinguishing features of a multiprocessor. In our desire to derive implementation independent results, we have always assumed that there are enough processors (except in the case of the multiphase structure). A useful extension of our work would be to derive approximations and bounds on performance measures for other structures as a function of the number of processors.

In analyzing the multiphase structure we have assumed that dividing the parallel phase work into subtasks would result in a linear decomposition; *i.e.*, the mean and variance of the subtasks would be related to the mean and variance of the parallel phase in a linear fashion. While this is generally true for parallel computations that work on independent portions of a large data set, other types of computations might have interesting non-linear decomposition functions. It

should be possible to extend our work in a straightforward manner to account for such decompositions.

References

1. E.G. Coffman, Jr., "Introduction to Deterministic Scheduling Theory," in *Computer and Job-Shop Scheduling Theory*, ed. E.G. Coffman, Jr., John Wiley, New York (1976).
2. R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed," *1988 ACM Sigmetrics Conference*, (May 1988). to appear
3. J. Mohan, "Performance of Parallel Programs: Model and Analyses," Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University (July 1984).
4. E. F. Gehringer, D. P. Siewiorek, and Z. Segall, *Parallel Processing: The Cm* Experience*, Digital Press (1987).
5. H. T. Kung, "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors," pp. 153-200 in *Algorithms and Complexity: New Directions and Recent Results*, ed. J. F. Traub, Academic Press, New York (1976).
6. G. Baudet, "The design and analysis of algorithms for asynchronous multiprocessors," Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University (April 1978).
7. M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York (1987).
8. P. A. Nelson and L. Snyder, "Programming Paradigms for Nonshared Memory Parallel Computers," pp. 3-20 in *The Characteristics of Parallel Algorithms*, ed. Robert J. Douglas, MIT Press (1987).
9. J. Deminet, "Experience with Multiprocessor Algorithms," *IEEE Transactions on Computers* C-31(4) pp. 278-288 (April 1982).
10. E. Felten, S. Karlin, and S.W. Otto, "The Traveling Salesman Problem on a Hypercubic, MIMD Computer," *Proc. 1985 International Conference on Parallel Processing*, pp. 6-10 (August 1985).
11. A.K. Jones and P. Schwarz, "Experience Using Multiprocessor Systems-A Status Report," *Computing Surveys* 12(2)(June 1980).
12. B. Lint and T. Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," *IEEE Trans. Software Engineering* SE-7(2) pp. 174-188 (March 1981).

13. S. P. Levitan, "Parallel Algorithms and Architectures: A Programmer's Perspective," Ph.D. dissertation, Department of Computer and Information Sciences, University of Massachusetts (May 1984).
14. R.H. Barlow, "Performance Measures for Parallel Algorithms," in *Parallel Processing Systems: An Advanced Course*, ed. D.J. Evans, Cambridge University Press, Cambridge (1982).
15. L. H. Jamieson, "Characterizing Parallel Algorithms," pp. 65-100 in *The Characteristics of Parallel Algorithms*, ed. R. J. Douglas, MIT Press (1987).
16. M. A. Marsan, G. Balbo, and G. Conte, *Performance Models of Multiprocessor Systems*, MIT Press (1986).
17. P. Heidelberger and K.S. Trivedi, "Queueing Network Models for Parallel Processing with Asynchronous Tasks," *IEEE Trans. Computers* C-31(11) pp. 1099-1108 (November 1982).
18. P. Heidelberger and K.S. Trivedi, "Analytic Queueing Models for Programs with Internal Concurrency," *IEEE Trans. Computers* C-32(1) pp. 73-82 (January 1983).
19. A. Thomasian and P. Bay, "Queueing Network Models for Parallel Processing of Task Systems," *Proc. 1983 International Conference on Parallel Processing*, pp. 421-428 (August 1983).
20. D. Towsley, K.M. Chandy, and J.C. Browne, "Models for Parallel Processing Within Programs: Application to CPU:I/O and I/O:I/O Overlap," *CACM* 21(10) pp. 821-830 (October 1978).
21. D. Vrsalovic, D.P. Siewiorek, Z.Z. Segall, and E.F. Gehringer, "Performance Prediction for Multiprocessor Systems," *Proc. 1984 International Conference on Parallel Processing*, pp. 139-146 (August 1984).
22. D. Vrsalovic, E.F. Gehringer, Z.Z. Segall, and D.P. Siewiorek, "The Influence of Parallel Decomposition Strategies on the Performance of Multiprocessor Systems," *Proc. Twelfth Annual International Symposium on Computer Architecture*, pp. 396-405 (June 1985).
23. M. Dubois and F. A. Briggs, "Performance of Synchronized Iterative Processes in Multiprocessor Systems," *IEEE Trans. Software Engineering* SE-8(4) pp. 419-431 (July 1982).
24. J. T. Robinson, "Some Analysis Techniques for Asynchronous Multiprocessor Algorithms," *IEEE Trans. Software Engineering* SE-5 pp. 24-31 (January 1979).
25. B. W. Weide, "Analytical Models to Explain the Anomalous Behavior of Parallel Programs," *Proc. 1981 Int. Conf. on Parallel Processing*, pp. 183-187 (August 1981).

26. C P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *Proc. 1984 Int. Conf. on Parallel Processing*, pp. 236-240 (August 1984).
27. M. Dubois, F.A. Briggs, I. Patil, and M. Balakrishnan, "Trace-driven Simulations of Parallel and Distributed Algorithms in Multiprocessors," *Proc. 1986 International Conference on Parallel Processing*, pp. 909-916 (August 1986).
28. K. So, F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister, "PSIMUL: A System for Parallel Simulation of the Execution of Parallel Programs," Research Report RC 11674, IBM T.J. Watson Research Center, Yorktown Heights, New York (January 1986).
29. T.S. Axelrod, P.F. Dubois, and P.G. Eltgroth, "A Simulator for MIMD Performance Prediction -- Application to the S-1 MkIIa Multiprocessor," *Proc. 1983 International Conference on Parallel Processing*, pp. 350-358 (August 1983).
30. E. Williams and F. Bobrowicz, "Speedup Predictions for Large Scientific Parallel Programs on Cray X-MP-Like Architectures," *Proc. 1985 International Conference on Parallel Processing*, pp. 541-543 (August 1985).
31. M. J. Flynn, C. L. Mitchell, and J. M. Mulder, "And Now a Case for More Complex Instruction Sets," *Computer* 20(9) pp. 71-83 (September 1987).
32. S. Madala, "Concurrent C User's Manual," Technical Report TR 8701, Dept. of Electrical and Computer Engineering, Rice University (January 1987, revised October 1987).
33. R. G. Covington and J. R. Jump, "CSIM User's Guide," Technical Report TR 8501, Dept. of Electrical and Computer Engineering, Rice University, Houston, Texas (April 1985, revised February 1986).
34. R. G. Covington and M. Ingram, "SIMTOOL: A User Interface for RPPT," Technical Report TR 8713, Dept. of Electrical and Computer Engineering, Rice University, Houston, Texas (October 1987).
35. S. Madala and B. Hsi, "TRAPP: A Trace Tool For RPPT Applications," Technical Report TR 8714, Dept. of Electrical and Computer Engineering, Rice University, Houston, Texas (October 7 1987).
36. P. J. Weinberger, "Cheap Dynamic Instruction Counting," *Bell Systems Technical J.* 63(8) pp. 1815-1826 (October 1984).
37. P. B. Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, New Jersey (1977).
38. H. Dietz and D. Klappholz, "Refined C: A Sequential Language for Parallel Programming," *Proc. 1985 International Conference on Parallel Processing*, pp. 442-449 (August 1985).

1985).

39. H. Dietz and D. Klappholz, "Refining a Conventional Language for Race-Free Specification of Parallel Algorithms," *Proc. 1984 International Conference on Parallel Processing*, pp. 380-382 (August 1984).
40. J.T. Kuehn and H.J. Siegel, "Extensions to the C Programming Language for SIMD/MIMD Parallelism," *Proc. 1985 International Conference on Parallel Processing*, pp. 232-235 (August 1985).
41. H. A. David, *Order Statistics*, John Wiley & Sons Inc., New York (1981).
42. A. H-S. Ang and W. H. Tang, *Probability Concepts in Engineering Planning and Design Volume II*, Rainbow Bridge Book Co. (1984).
43. D.E. Knuth, *The Art of Computer Programming Vol.3*, Addison-Wesley, Reading, MA (1973).
44. K. Hwang and F. A. Briggs, *Computer Architecture And Parallel Processing*, McGraw-Hill, New York (1984).
45. R. Melhem, "Irregular Wavefronts in Data-driven, Data-dependent Computations," pp. 303-312 in *Systolic Arrays*, ed. Roddy Urquhart, Adam Hilger (1987).
46. S. Y. Kung, P. S. Lewis, and S. C. Lo, "Performance Analysis and Optimization of VLSI Dataflow Arrays," *Journal of Parallel and Distributed Computing* 4 pp. 592-618 (December 1987).