

1991

Analysis of a coordination framework for mapping coarse-grain applications to distributed systems

Linda Ruth Schaefer
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds



Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

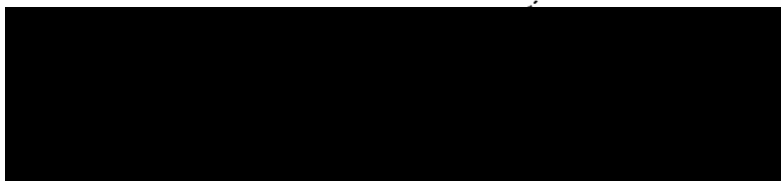
Schaefer, Linda Ruth, "Analysis of a coordination framework for mapping coarse-grain applications to distributed systems" (1991). *Dissertations and Theses*. Paper 4270.
<https://doi.org/10.15760/etd.6154>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

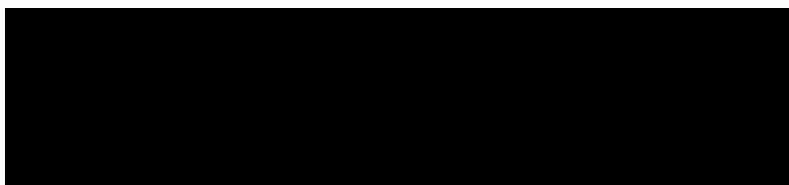
AN ABSTRACT OF THE THESIS OF Linda Ruth Schaefer for the Master of Science in Electrical and Computer Engineering presented May 31,1991.

Title: Analysis of a Coordination Framework for Mapping Coarse-Grain Applications to Distributed Systems.

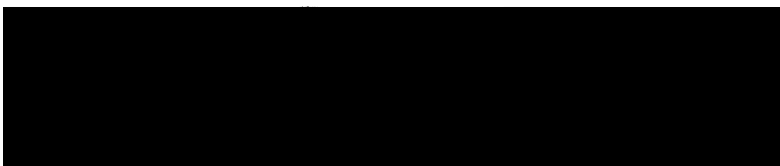
APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:



W. Robert Daasch, Chair



Michael A. Driscoll



Laszlo Csanky

A paradigm is presented for the parallelization of coarse-grain engineering and scientific applications. The coordination framework provides structure and an organizational strategy for a parallel solution in a distributed environment. Three categories of primitives which define the coordination framework are presented: structural, transformational, and operational. The prototype of the paradigm presented in this thesis is the first step towards a programming development tool. This tool will allow non-specialist programmers to parallelize existing sequential solutions through the distribution, synchronization, and collection of tasks. The distributed con-

trol, multidimensional pipeline characteristics of the paradigm provide advantages which include load balancing through the use of self-directed workers, a simplified communication scheme ideally suited for infrequent task interaction, a simple programmer interface, and the ability of the programmer to use already existing code. Results for the parallelization of SPICE3C1 in a distributed system of fifteen SUN 3 workstations with one fileserver demonstrate linear speedup with slopes ranging from 0.7 to 0.9. A high-level abstraction of the system is presented in the form of a closed, single class, queueing network model. Using the Mean Value Analysis solution technique from queueing network theory, an expression for total execution time is obtained and is shown to be consistent with the well known Amdahl's Law. Our expression is in fact a refinement of Amdahl's Law which realistically captures the limitations of the system. We show that the portion of time spent executing serial code which cannot be enhanced by parallelization is a function of N , the number of workers in the system. Experiments reveal the critical nature of the communication scheme and the synchronization of the paradigm. Investigation of the synchronization center indicates that as N increases, visitations to the center increase and degrade system performance. Experimental data provides the information needed to characterize the impact of visitations on the performance of the system. This characterization provides a mechanism for optimizing the speedup of an application. It is shown that the model replicates the system as well as predicts speedup over an extended range of processors, task count, and task size.

ANALYSIS OF A COORDINATION FRAMEWORK FOR MAPPING
COARSE-GRAIN APPLICATIONS TO DISTRIBUTED SYSTEMS

by

LINDA RUTH SCHAEFER

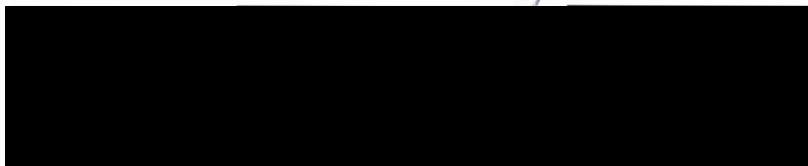
A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

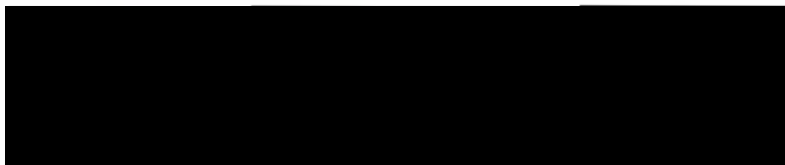
Portland State University
1991

TO THE OFFICE OF GRADUATE STUDIES:

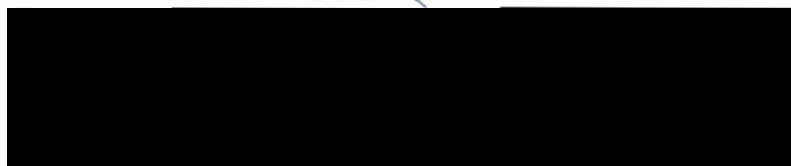
The members of the Committee approve the thesis of Linda Ruth Schaefer presented May 31, 1991.



W. Robert Daasch, Chair



Michael A. Driscoll



Laszlo Csanky

APPROVED:



Rolf Schaumann, Chair, Department of Electrical Engineering



C. William Savery, Vice Provost for Graduate Studies and Research

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my heartfelt thanks to Rob Daasch, my advisor and friend. Rob provided guidance, encouragement, and unwavering intellectual and emotional support, that directly contributed to my success. His patience and wisdom guided me through this endeavor and his confidence in my abilities provided me with confidence in myself.

I wish also to thank Mike Driscoll for his friendship and intellectual support. His enthusiasm for this work lifted my spirits and his ideas and suggestions contributed greatly to this thesis.

I would also like to thank my colleagues at the university for their contribution to my academic growth and the faculty and staff of the Electrical Engineering department for their encouragement and support in ways too numerous to mention.

Finally, I would like to thank my many friends and my mother, Marguerite Schaefer, whose encouragement and understanding provided for my emotional wellbeing throughout my schooling.

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGEMENTS	iii
LIST OF TABLES	vii
LIST OF FIGURES	ix
CHAPTER	
I INTRODUCTION.....	1
Parallelism and Distributed Computing	1
Parallel Paradigms	5
A Coordination Framework.....	6
Thesis Overview.....	8
II THE PARADIGM.....	11
Primitives	11
Structural Primitives	
Transformational Primitives	
Operational Primitives	
Characteristics of the Paradigm.....	15
Multidimensional Pipeline	
Distributed Control	
Implementation.....	20
System Configuration	
Control Programs	
Jobpools	
Workers	
Communication Topology	
Locks	
Load Balancing	
Job Partitioning and Granularity	

	Summary.	31
III	APPLICATIONS	33
	Characteristics	33
	Selected Examples	36
	Fault Simulation Database Searching	
	Design Centering: A Detailed Example.	40
	The Monte Worker The SPICE Worker The Pass/Fail Worker The Update Worker	
	Summary.	43
IV	STATISTICS	44
	Introduction	44
	Computer Systems Analysis	45
	Queueing Theory versus Queueing Network Theory Queueing Network Models The Physical System	
	Queueing Network Models	47
	The Single Class Queueing Network Network Analysis The Complex Model The Simple Model Performance Metrics	
	Explanation of Experiments.	61
	Instrumentation Experimental Environment	
	Experimental Basis.	65
	Input2 Input6	
	Experimental Results	66
	Summary.	95

V	CONCLUSION.....	99
	Limitations and Future Work.....	101
	REFERENCES	103
	APPENDIX.....	105

LIST OF TABLES

TABLE	PAGE
I Queueing Network Performance Parameters.	48
II Network Parameter Relationships	51
III Queueing Network Measurements.	62
IV Average Total Work and Communication Time 500 Tasks, 6 Workers, Input6	72
V Average Total Work and Communication Time 100 Tasks, 6 Workers, Input6	72
VI Average Total Work and Communication Time 50 Tasks, 6 Workers, Input6	72
VII Average Total Work and Communication Time 500 Tasks, 12 Workers, Input2	73
VIII Average Total Work and Communication Time 100 Tasks, 12 Workers, Input2	74
IX Average Total Work and Communication Time 50 Tasks, 12 Workers, Input2	74
X Average Total Work and Communication Time 100 Tasks, 9 Workers, Input6	76
XI Average Total Work and Communication Time 100 Tasks, 13 Workers, Input6	76
XII Average Total Work and Communication Time 100 Tasks, 14 Workers, Input6	77

XIII	Average SPICE Times (in seconds)	78
XIV	Service Center Averages Applied to Simple Model from Experiment Input2-100 Workers	83
XV	Equations for V_{c_lock} Corresponding to Figure 18	86
XVI	Equations for V_{c_lock} Corresponding to Figure 19	87
XVII	Slope Values for V_{c_lock} Within Linear Range for Varying Numbers of Tasks	89
XVIII	Equations for V_{c_lock} Corresponding to Figure 24	93
XIX	Equations for V_{c_lock} Corresponding to Figure 25	94

LIST OF FIGURES

FIGURE	PAGE
1. Message passing architecture.	3
2. Structure of the parallelized program.	8
3. Workers obtaining and placing tasks in jobpools.	10
4. Graphical representation of a jobpool and transformations by workers. ...	14
5. Job flow pipeline for the fault simulation application	38
6. Job flow pipeline for the design centering application.	43
7. Queueing and delay service centers.	47
8. The Complex model.	53
9. The Simple model.	54
10. Gantt chart of idle and work time.	56
11. Cycle time for experimental data input2	68
12. Cycle time for experimental data input6	68
13. Experimental speedup within linear range.	69
14. Experimental speedup for all data	69
15. Experimental data versus best fit line.	82
16. Family of curves within linear range for input6.	85
17. Family of curves within linear range for input2.	85
18. Theoretical extrapolation versus experimental linear speedup for input6 .	86
19. Theoretical extrapolation versus experimental linear speedup for input2 .	87
20. Experimental versus piecewise linear best fit for input6-100 jobs.	91

21.	Theoretical breakdown extrapolation versus experimental speedup for input6-100 jobs	91
22.	Experimental versus piecewise linear best fit for input2-50 jobs	92
23.	Theoretical breakdown extrapolation versus experimental speedup for input2-50 jobs	92
24.	Theoretical breakdown extrapolation versus experimental speedup for input6.	93
25.	Theoretical breakdown extrapolation versus experimental speedup for input2	94
26.	Simple model cycle time versus Complex model cycle time.	96
27.	Simple model speedup versus Complex model speedup.	96

CHAPTER I

INTRODUCTION

PARALLELISM AND DISTRIBUTED COMPUTING

It is generally recognized within the scientific and engineering communities that many problems in these fields would benefit from parallel processing. Many of the solution techniques are computationally intensive and require hours of processing time on a single machine. Decomposition of these problems into smaller subproblems and simultaneously processing them on multiple processors can greatly speed up the solution process. There is no lack of candidates for concurrent scientific programming. Nor is there lack of opinion as to which approach is the preferred method of choice. The following statements were reported as having been heard at various times around the Cornell National Supercomputing Facility in Ithaca, New York [1].

"Most parallel algorithms can be expressed in producer-consumer terms."
- a computer scientist.

"Almost all scientific codes are parallelizable at the DO loop level."
- a physicist.

"Master-slave is the most intuitive parallel programming model."
- an engineer.

"The pool of tasks is likely to be the basic paradigm..."
- high energy physics (sic)

"To really incorporate natural processes into your program, you need to use domain decomposition."
- a meteorologist.

Fortunately, many computationally intensive problems exhibit potential for concurrency. But decomposition or partitioning of a problem is a significant task. The communication overhead of the system, coordination of subtasks once the problem is partitioned, and minimization

of serial processing to exploit parallelism and decrease computing time must be taken into consideration. In addition, paradigms for mapping these problems to an existing parallel environment are lacking.

From the computation scientist's point of view, two important aspects of parallel computing include turnaround time and implementation complexity [2]. One of the objectives of parallel processing is to reduce the total "wall-clock" or turnaround time required for a process. The time it takes to get results out of a particular computer configuration for a specific computation is critical to the user. In addition, the amount of effort the user must go to in order to utilize the computer system says something about the complexity of the problems to be undertaken. The question of what class of problems can be run efficiently with respect to turnaround arises. Other concerns, such as how much effort must be expended in learning a new operating system, new programming languages, or concurrency control functions used to implement a specific parallel paradigm all go into the decision making process as to whether or not to attempt the parallelization of an application. The user will want to have some knowledge of what is to be gained prior to rewriting application code or interfacing old code to a new system for parallel execution.

The benefits of parallelizing an application can take the form of greater accuracy and detail of results, more problems solved in a shorter amount of time, and larger and more complex problems solved when uniprocessor computation time for these solutions would be prohibitive. In some respects, parallel processing may even offer a return to the more simplistic approaches to problem solving over complicated and hard to understand heuristics designed to cut back on computation time.

The particular system on which to run the parallel solution must also be decided. Many architectures exist for the realization of parallelism. Some of these architectures are special-purpose while others are more general. We will focus our attention on the Multiple Instruction Multiple Data (MIMD) architecture. The general principle behind the MIMD architecture is

based on the idea that given p processors operating on p independent but similar tasks of an application, performance improvements ideally will be linearly proportional to the number of processors [2]. This is rarely observed in practice. The way in which speedup scales with the addition of processors is bound to be different for each system configuration. Many factors must be recognized as affecting the speedup including system synchronization and communication topology.

The MIMD subclass of network processors and distributed computing is the environment of interest for this thesis. Processors in the distributed environment have their own local memory and execute their own programs. Each processor has its own control unit and central processing unit. Configurations for this environment can consist of quite diverse machines and are often comprised of heterogeneous, general purpose computers. The communication topology is shown in Figure 1. It is a loosely coupled network of processors where interprocessor communication is implemented via a message passing protocol.

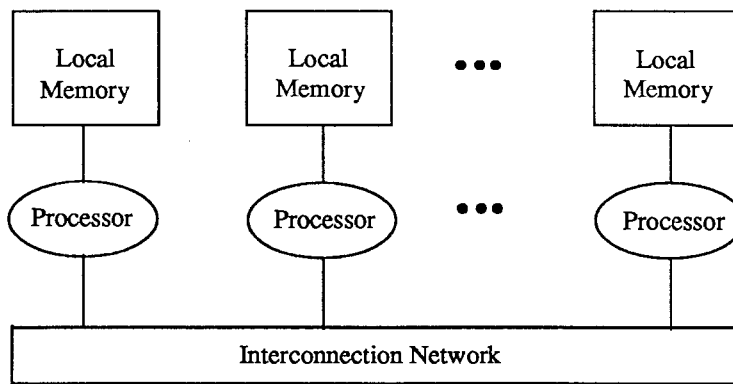


Figure 1. Message passing architecture [3].

The individual workstation is becoming a more common item in the industrial, university, and commercial work place. By connecting these workstations together over a local area network this computing power can be harnessed and used as a viable approach to solving computationally intensive problems. The distributed network of workstations is a promising environment and offers many challenges as a parallel processing platform.

One particular challenge is accessing global memory. For information to be global in a distributed memory system there must be some mechanism available for accessing other processors' local memory since no common physical address space exists. The message passing protocol is one solution to this problem. Other solutions include replication of memory in multiple processors and I/O capabilities for accessing memory. Regardless of what approach is used, it is guaranteed that contention for global memory will result in additional communication overhead, extra data movement, or both.

Applications of a distributed parallel computer have been explored in a variety of ways. Three of the main focuses are:

- The design of distributed processor algorithms specifically targeted to distributed parallel computers. Strictly speaking this approach is intended for a dedicated network of processing elements such as an Intel iPSC and not a general purpose workstation network [4].
- The development of monitors that allow the sharing of unused computer cycles for cooperating processes in a local area network. Such a monitor provides a mechanism for determining 'lightly' loaded machines but rarely provides a means for coordinating a parallel task [5, 6].
- The specification of new coordination languages and program development tools that allow programmers to distribute and synchronize tasks and then collect the results of the completed computation [7, 8].

Our paradigm falls into the third category and is targeted for scientists and engineers who do not have the inclination to become proficient parallel programmers but have a desire to take advantage of parallelism in their computer problems. To do this we must develop a suitable technique to control and coordinate the actions of individual nodes so that they can contribute to the solution of complex problems. This technique takes the form of a problem solving tool that is matched to the distributed processing environment.

PARALLEL PARADIGMS

A paradigm provides the framework or structure within which the programmer can work to arrive at a solution. It is a problem solving strategy which allows for the structuring of dissimilar problems in such a manner that they can be solved by similar means. A programming paradigm is a high-level methodology which serves as a specification for communication patterns used to reference data [9].

Carriero and Gelernter distinguish between three basic paradigms for providing parallelism on general purpose asynchronous parallel machines: *result*, *agenda*, and *specialist* [10]. Each of these paradigms lends itself to an identifiable group of problems and represents a distinct way of thinking about parallelism. The boundaries between the three sometimes become blurred and elements of each are often used to arrive at a final solution. Result parallelism focuses on a data structure which yields the final result. It is used most effectively in problems which require a series of values where each value represents a piece of the whole solution. Here, each worker constructs an individual piece of the finished product. Agenda parallelism focuses on an agenda of tasks which many workers apply themselves to simultaneously. It is a versatile approach that adapts easily to many different problems. Here, a program executes in the same way regardless of the number of workers. Specialist parallelism assigns a worker to perform a specific task. This approach focuses on the make-up of the workers and is well suited to a pipeline where multiple transformations on identical tasks are required. Chapter III will expand on the characteristics of these three paradigms and discuss the suitability of each for implementation using our paradigm.

In developing these three basic paradigms, Carriero refers to the term *coordination* as the process of "building programs by gluing together active pieces". The active pieces are the tasks or threads of the problem and the glue is what holds them together. We can think of this glue as the ingredient that addresses the need for communication and synchronization.

There are several coordination languages that have been developed for solving medium or coarse-grain problems in a distributed environment. These methodologies provide a language to facilitate the organization and structure of the parallel program which is written using a computing language. Three such languages are Linda, FrameWorks, and Paralex.

Linda [10] is a memory model and the C-Linda compiler supports the combined language environment of the computing language of C and the coordination language of Linda. Memory is defined as a tuple space consisting of a collection of process and data tuples. Process tuples execute simultaneously and exchange data by generating, reading, and consuming data tuples. Linda supports all three types of parallelism discussed above.

FrameWorks [7] is a coordination language which uses remote procedure calls to implement message passing communications. Here, the programmer writes sequential C procedures and encapsulates them in templates. Templates are the coordination language which defines how each process will interface with others within an application. Framework is a specialized system which provides a vehicle for mapping agenda and specialist parallelism to the distributed system.

Paralex [8] is a coordination language designed to facilitate the mapping of coarse-grain data flow problems to a distributed system. Nodes and links define a Paralex program where nodes are provided to identify computations and links to indicate flow. Paralex uses the ISIS [11] toolkit to support universal data representation and remote communication. Paralex addresses the specialist parallelism required in data flow computations where nodes are assigned a specific kind of work.

A COORDINATION FRAMEWORK

In this thesis we will develop, implement and evaluate a prototype of the paradigm designed to map applications exhibiting coarse-grain parallelism to the distributed network. In a loosely coupled system, communication costs are relatively high due to greater communication

frequency and limited bandwidth. The user can accommodate this drawback by choosing problems that have a coarse computational granularity and infrequent task interactions. Coarse-grain parallelism entails a group of tasks working collectively towards a solution with a relatively small amount of task interaction.

For the paradigm to be useful to the general user, the distributed processing environment would ideally include:

- A simple programmer interface. This would facilitate conversion of the uniprocessor application to the parallel environment.
- A simplified communication scheme ideally suited for infrequent task interaction. This results in less overhead which is important in a message-passing environment where bandwidth is low and communication is expensive. Furthermore, programming is easier than with difficult communication protocols.
- The ability to use already existing code. This includes sequential application code as well as standard system calls which facilitate portability. These applications will be identified with specialist workers.
- A flexible approach to synchronization. This is needed to handle applications where an equitable distribution of work is difficult to arrange in advance. This will be accomplished using agenda parallelism.

We will view this paradigm as a *coordinating framework* which provides structure and an organizational strategy for resolving the decomposed problem into a viable distributed parallel program. A three level structure that illustrates our implementation is shown in Figure 2. At the bottom level is the original sequential program unencumbered by knowledge of a parallel platform on which to run. The intermediate level is the interface between the parallel and sequential implementation. This provides an easier entry into parallel programming in the form of a 'wrapper' around the original program. The vehicle used to deliver the parallelization is our coordination framework which sits on top of the interface.

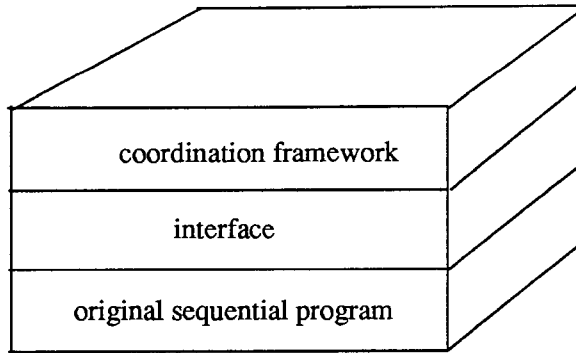


Figure 2. Structure of the parallelized program.

Central to the organizational strategy of our coordination framework are the concepts of the *jobpool* and *worker*. Similar tasks which can be run in parallel are found in jobpools and are placed there when it is time for execution. Any number of workers can be assigned to a jobpool and are assigned to processors at the discretion of the user. This leads to flexibility when gaining or losing processors on a network. Workers are self-scheduled and obtain work when it becomes available in the jobpool. Figure 3 provides a pictorial representation of workers retrieving tasks from their assigned jobpools.

THESIS OVERVIEW

In this chapter we presented the general computing environment for the parallelization of coarse-grain applications. Our paradigm provides the basic framework within which the programmer can work to fashion a solution to a problem. It provides the structure and organizational strategy for mapping the application to the distributed environment. Several parallel paradigms were presented as problem solving tools for scientific applications and three coordination languages were discussed in accordance with these concepts of parallelism.

In Chapter II our paradigm is defined and a nomenclature is developed that will be used to discuss the prototype in detail. This chapter presents the coordination framework and discusses the organizational primitives of this framework as well as issues of synchronization and communication.

Problems which lend themselves to our paradigm are addressed in Chapter III. Several problems are presented along with possible implementation strategies. The methodology for the decomposition process will be explained by example. This will entail the complete mapping of an application called Design Centering to the paradigm.

Experimental results are presented in Chapter IV. A model is developed to describe the topology of the physical system and is used as a template for the instrumentation of the experiment. Graphs and tables detailing the experimental results are provided. With the help of queueing network theory we present a simple expression for the behavior of the system. Using this model we predict behavior of the system for large numbers of workers and determine how accurate our model is from the experimental results.

Chapter V concludes the thesis with an evaluation of the research. A discussion on the limitations as well as the strengths of the paradigm is included. Comparisons to related work in this field are undertaken. Future work based on the conclusions drawn from the experimental results is considered.

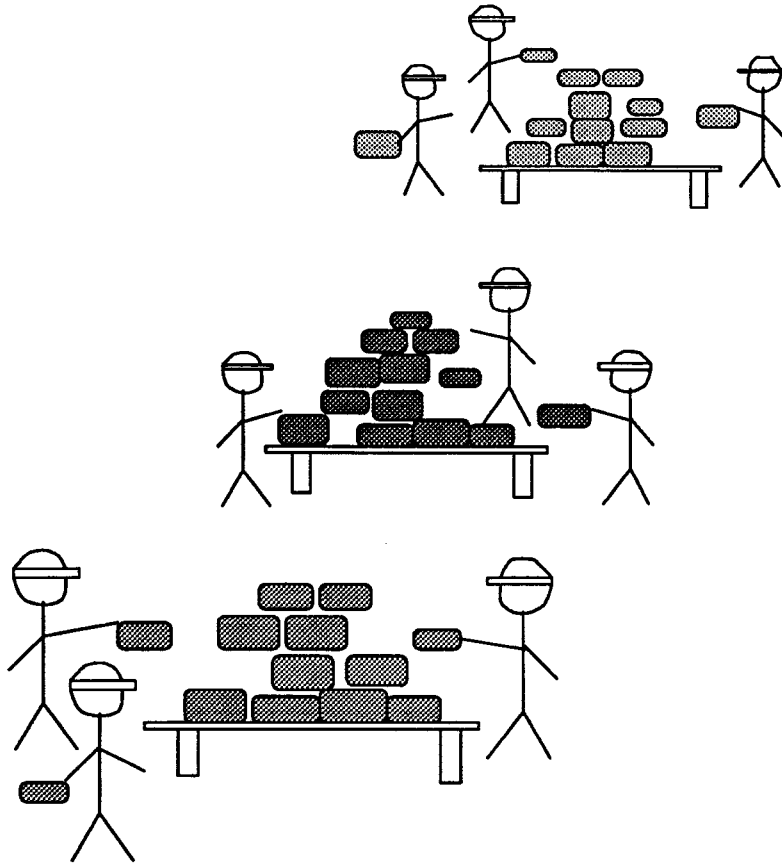


Figure 3. Workers obtaining and placing tasks in jobpools.

CHAPTER II

THE PARADIGM

This chapter discusses the specifics of the coordination framework exhibited in our prototype which we have developed and implemented. Primitives are introduced and a nomenclature is developed to facilitate the discussion of the abstractions of the paradigm. Characteristics of the paradigm are addressed and the details of the implementation are presented. Topics of load balancing, job partitioning and task granularity are considered with respect to our paradigm.

PRIMITIVES

To support parallelization of applications in the distributed environment we developed a set of primitives to define the coordination framework used for synchronization and organization of the parallel problem. The primitives fall into three categories: structural, transformational, and operational.

Structural Primitives

Structural primitives include the jobpool and worker. A jobpool is a resource shared by several concurrent processes. More specifically, it is a data structure designed to manage a collection of independent tasks which are to be operated on in the same manner. It is assigned a unique name at the time of its creation which is used by all processes accessing the jobpool. Jobpools facilitate agenda parallelism by providing the organizational structure for a list of tasks to be performed. We have identified four characteristics of a jobpool which are as follows:

1. A jobpool contains independent but similar jobs.

2. Tasks residing in the jobpool require the same type of worker.
3. A jobpool is a source for a worker.
4. A jobpool is a destination for a worker.

Work is carried out on a specific jobpool by a collection of workers, identical in function, and designed strictly to service the jobpools. Workers take the form of specialist parallelism in their function and are the computational engines for the paradigm. Upon initialization, workers are permanently assigned to a source jobpool and a destination jobpool. Characteristics of a worker include:

1. A worker facilitates the transformation between jobpools.
2. A worker is data driven.

Transformational Primitives

There are several types of transformations that a worker can perform between a source and a destination jobpool. In this chapter it will be convenient to refer to specific workers by their transformational attributes. Prior to describing these transformations, the concepts of *consuming* and *spawning* work must be defined.

The consuming of a task begins when a worker removes it from a source jobpool and is finished when processing of the task is complete. The spawning of tasks occurs when a worker consumes work and generates one or more tasks, placing these in the assigned destination jobpool. There may be any number of intermediate destinations for a particular task as dictated by the assigned worker. Regardless of the number of intermediate destinations an iteration is the same: the task is removed from a source jobpool and operated on (consumed) and placed in a destination jobpool (spawned). It will be seen in the special case that a worker will not necessarily spawn a job after consuming a task.

The first of the transformations to be discussed is the 1-to-many transformation. This entails a worker consuming a single task, operating on the task, and then spawning multiple

new tasks. The well known divide and conquer paradigm uses this transformation to divide up an original problem into smaller subproblems. Here we are using the same approach but also taking advantage of the fact that given sufficient parallelism the subproblems can be solved simultaneously.

A second type of transformation is a many-to-1. A worker consumes two or more jobs from its assigned source jobpool and deposits only one job in the destination jobpool. This transformation would be typical of the recombining or accumulating of data from processes working on smaller subproblems in an effort to produce a single task.

A third transformation is the 1-to-1. Here, a worker consumes a single job from the source jobpool and places a single job in the destination jobpool. In the special case of initialization this destination jobpool may be null. A typical example of the use of this transformation might be in the execution of a pipeline stage. Each data object entering the pipeline stage is operated on in an identical manner and the output from the pipeline reflects these changes.

Other possibilities include a many-to-many worker and a conditional worker. The conditional worker allows for a question to be asked with respect to a task and provides different directions of flow depending on the decision. Any of the branches of the conditional worker can be of the form one or many. Figure 4 introduces a graphical representation for a jobpool and the various transformational workers which will later be used to construct a pictorial representation of an application pipeline. In this picture we show an example of a conditional worker with an input branch specifying multiple task consumption and two output branches for single task spawning.

In our paradigm, the workers are data driven or self-directed. Using this approach there is no need for the traditional master-slave communication protocol. Workers check the jobpool periodically for any new work that has arrived and consume work upon availability. Workers who are dedicated to 1-to-1 transformations do not require knowledge about the number of jobs being processed through the jobpool. Workers who perform 1-to-many, many-to-1, or many-

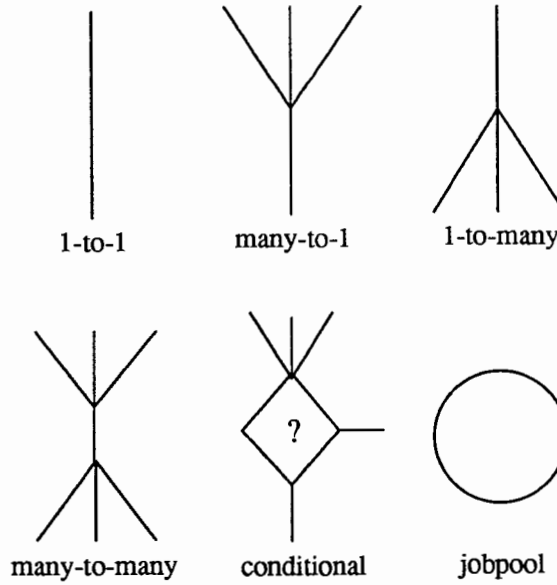


Figure 4. Graphical representation of a jobpool and transformations by workers.

to-many transformations require additional information relative to the number of tasks to be consumed or spawned.

The jobpool structure does not care, nor does it have any knowledge of, how many workers are assigned to it. Theoretically, it does not care how much work is processed through it. From a practical point of view there will always be a limit to how much work a pool can contain at an instant of time with respect to system limitations.

Operational Primitives

To facilitate transformations on jobpools we have created operational primitives for obtaining work from the jobpool and monitoring within the jobpool. When more than one process attempts to consume a task, problems can occur. To sequentialize the manner in which jobs are obtained by the workers, which in turn will maintain integrity of the jobpool, arrangements are made to guarantee that no more than one process has control of the jobpool at any one time.

The *lock* and *unlock* primitives address the need for maintaining the integrity of the jobpool. A worker obtains control of the lock, using the lock primitive, in order to consume work

from the source jobpool. After a task has been removed from the source jobpool the worker releases the lock, using the unlock primitive, and the jobpool is available for other workers to control. There are two primitives available for the consuming of a task. The first is the *deliver_task* primitive which allows for a single task to be consumed. The second is the *deliver_jobpool* which allows for multiple tasks from a jobpool to be consumed. Moving the task to the appropriate destination jobpool is done using the *place_task* primitive.

In addition to the basic primitives already discussed, a primitive for monitoring the jobpool has been supplied. This primitive enables the user to write to a data structure for the sake of internal monitoring of the processing. The user may be interested in developing a trail so that the resident time a task spends in a destination jobpool can be determined, as well as which worker consumed it, and how long the operations on the task took prior to the completion of the transformation. This primitive makes this possible.

In an effort to simplify the transition to a distributed environment for non-specialist programmers, these primitives have been designed to mask the semantics of the operations required and to look like simple procedure calls to the user. There will be more discussion of the above primitives under the topic of implementation.

CHARACTERISTICS OF THE PARADIGM

The general description of the paradigm we have developed falls under the category of a distributed control, multidimensional pipeline. That is, the paradigm has attributes that can be described as distributed with respect to control or communication, multidimensional in structure, and pipelined in functionality.

Multidimensional Pipeline

The pipelining of processes requires that each process perform a specific function for the duration of its existence. Information is passed through the pipeline from one process, or stage, to the next as if moving through an assembly line. A process operates on each input data object

as it enters the stage and passes the results to the next stage in the pipeline upon completion.

In his paper, King differentiates between *pipelined* and *concurrent execution* [4]. For concurrent execution, multiple processors execute multiple independent tasks at the same time. For pipelined execution, each processor takes on the role of a stage in a pipeline and operates on data as it enters the stage and passes output data on to the next processor. In our paradigm, we exploit concurrent execution in a pipelined fashion. Using King's definitions, our parallel execution style falls into both categories of execution. We allow for the running of multiple independent processes on multiple workstations thus allowing for different processors to execute different functions simultaneously. But we go beyond the classification of concurrent execution in parallel computation when we structure the pipelining of the execution process using jobpools.

Jobpools with their respective workers represent the pipeline stages dedicated to various functions in the solution to an application. Here, workers are assigned a specific function to be performed on each data object that enters the jobpool. When operation on a task is complete, the task is moved to the next stage in the pipeline which we have defined as the destination jobpool. Concurrency in computation is exploited in two ways: by the overlapping of computation between pipeline stages and by the parallel computation of multiple tasks in a single jobpool. Using Carriero's concepts, the collection of jobpools is the vehicle for implementing a list of tasks found in agenda parallelism while the workers provide a computational form of specialist parallelism by their assigned expertise.

Distributed Control

Pipeline control is often centralized to facilitate the movement of data from one pipeline stage to the next stage in lock-step fashion. Here, a system-wide clock signal is used to control the system and serves two purposes. The first purpose is that of a sequence reference. The transitions of the clock define instances in which the state of the system may change. The second purpose of the global clock signal is that of a time reference. The period, or interval, between

clock transitions determines the length of computation time allowed for a particular state of the system. In a centrally controlled pipeline, the stage which has the longest processing time controls the clock period for the entire pipeline. This is an explicit dependency between processes. An example of a centrally controlled system as described above can be seen in a systolic array implementation of matrix multiply [12].

In a distributed control system, the clocking discipline is internal to the individual process. Transition instances are determined in a manner such that each process has an internal clocking mechanism which regulates its own input and output phase. The period of this internal clock allows for the individual computation time of the process. A process which operates on its own independent clock is self-timed. With no global signal to synchronize operations between processors, a self-timed paradigm such as ours requires that parallel computations be data driven.

No two workers assigned to the same jobpool will necessarily be at the same point in processing at any given time. Work in a jobpool is not done in lock-step fashion nor is there any strict requirement that all work be completed in a particular jobpool before additional work can be made available to the next jobpool. Here, no global clocking mechanism dictates at what time a task is moved to the next stage. In fact, in a distributed system, it is actually hoped that the pattern of communication describing the moving of work from one stage randomizes over time. The overlapping of communication and computation leads to a more efficient use of the distributed system and less time spent blocking at the input phase of individual processes. If shared resources are accessed at random, on average, the throughput of the system will be greater due to less time spent spinning, or synchronizing, per processor and more time spent computing.

As pointed out by Duba [13], if job partitioning and job allocation are not done correctly, an increase in the number of processors may actually result in the decrease of total throughput. For his implementation, this decrease can be directly related to the amount of messages that are

exchanged between processors. Duba uses a client-server communication topology to run a distributed fault simulator for VLSI circuits on a loosely-coupled network of general purpose computers. Here the client process acts as a host, sending and receiving work to the server processes. The server processes perform simulations and send results to the client to collect. The bottleneck would occur in the communication protocol at the client processor when large numbers of server processes require servicing. In addition, a hand-shaking scheme factors into the communication overhead. It is precisely this bottleneck which we are attempting to minimize with our paradigm.

A distributed control approach to pipelining allows for a variability of time for each stage. In designing the control of the paradigm to be distributed we are able to surmount the limitations that so often accompany the pipeline solution. The largest limitation being that of the global timing mechanism employed for propagating data through the pipe. The concurrent processing seen by the overlapping of pipeline stage computation allows for multiple independent threads to propagate through the pipe. This in turn allows for fast moving threads of the decomposed problem to proceed rapidly through the pipe resulting in scheduling efficiency since idle workers receive work as soon as it becomes available.

The characteristics of the paradigm also allow for multiple iterations, as well as multiple algorithm threads, of a problem to co-exist in the same pipeline stage. Let us say, for example, that a problem has several different algorithms to use towards a solution and can branch and choose one of these algorithms at some designated point in the pipeline process. Now, let's imagine that a 1-to-many worker spawns a number of red tasks and places them in the destination jobpool. Simultaneously, another 1-to-many worker spawns a number of blue tasks and places them in the same jobpool. Workers who consume red tasks are required to operate on these tasks using the red algorithm, while workers who consume blue tasks use the blue algorithm. If the work to be done on the tasks is the same regardless of the color of the deck, then that stage of the pipeline ignores the distinguishing colors. At the point where a new algorithm is to begin,

a new seed task would be color coded, reflecting the algorithm which was used to generate it. By tagging the new seed task with the color of the algorithm, the resulting tasks will again be color coded and may co-exist in the job pools. In effect this would be multiple algorithms of a solution to a problem being concurrently processed. Along the same lines, iteration threads could be color coded as they progress through a cyclic pipeline.

The pipeline that has been discussed above is of a generic, multidimensional nature and can be used to implement a variety of topologies. The most obvious structure would be that of the single-dimension, linear pipeline. Systolic arrays that employ a two-dimensional square mesh would also be a superset of the form discussed here. Systolic arrays consist of a network of modular processing units with a regular structure and local interconnects. They also feature an important property of pipelinability. Here, a stage of the pipeline is uniquely assigned to a processor and a global clock is required to synchronize the point at which data is moved from one stage of the pipeline to the next. Since timing for the systolic array is centralized at both input and output phases of each processors' clock, it is not necessarily a good structure to map to this paradigm. The close synchronization required in a systolic array is oriented more towards direct connections found in hardware designed for these specific applications rather than general networks. These centralized message systems are not particularly suited to distributed control, message passing facilities.

A topology more similar in execution to the paradigm is the wavefront array, a subset of the systolic array [14]. Wavefront arrays differ most from systolic arrays in their timing requirements and how they are driven. In a wavefront array, timing requirements in the systolic array are replaced by the requirement for correct sequencing in the wavefront array [12]. In both array architectures, data transfer between processors is done through a simple handshaking scheme. This communication occurs at the mutual convenience of the processors but the communication is not control driven as in the systolic array. Computation occurs when new data is available for a processor and the processor is ready for it. Thus, computation is self-timed. Data

can be passed without concern for a global clocking mechanism and there is no requirement to hold back faster processors in order to accommodate slower ones as is required in the systolic array. This is similar to what we have described in our paradigm to the extent that computation is data driven and self-timed. We have lowered the overhead of the handshaking scheme to pass work to a processor. Data objects are moved to the neighboring jobpool by the operating system and will be consumed for additional computation by the next process at its convenience. Processes may acquire available new tasks to work on by simply checking the jobpool. Acquisition of this new task, once found in the jobpool, only requires the operating system.

Properties of our distributed multidimensional pipeline can be summarized as follows:

- modularity - the paradigm allows for flexible structure as well as standardized units in the form of jobpools for easy construction
- simplified distributed communications
- high degree of pipelining
- dynamic asynchronous data flow computing - computations are data driven and self-timed
- extendibility of computing structure - by the addition of more jobpools in both width and depth
- flexibility and ease of adding or deleting processors

IMPLEMENTATION

System Configuration

The system configuration for our implementation consists of a local area network of SUN workstations running SUNOS 4.0.3. The network consists of a jobpool file server and twenty-two diskless workstations comprised of 3/110s and 3/50s. All of the engineering workstations are equipped with Motorola 68881 math coprocessor chips. System calls used in the implementation are Berkeley 4.3bsd UNIX commands.

The shared file system contains the source and destination jobpools in the form of directories. Tasks to be processed by workers take the form of files. A parallel solution to a problem using our paradigm will entail the use of a directory tree. A single file server manages the structure to ensure that any changes made to directories functioning as jobpools are made serially. A separate file server provides swap space for the diskless workstations. Attention is focused on the directory file server for the jobpool subtree in our instrumentation.

Executables that are to be initialized as workers are located in the top directory of the structure. Other information found in this top level directory includes control programs for various aspects of initialization and shutdown, files used to collect data for run time statistics, and one or more tasks required to seed the computation at the very beginning of the parallelization process.

Control Programs

Control programs for various aspects of the parallel processing environment are provided in our implementation. These include programs to initialize the parallel process, shutdown a portion or all of the process, and collect statistics and generate graphs from data supplied by the worker programs. Some of these programs are in the form of C-shell scripts while others are written in C.

A control program initializes all subdirectories required by the parallel application. Subdirectories consist of a directory to hold all data collected, a directory to manage messages generated during the shutdown sequence, and all directories used for managing and processing work, i.e. jobpools. At the time of initialization, data structures, in the form of log files, are created in each jobpool. These files can be written to via the monitoring primitives, mentioned earlier, which are available to the worker programs.

The control program sees that all worker programs are initiated on specific processors and are assigned to their respective source and destination jobpools via the rsh remote shell command. At this point multiple copies of a worker may be assigned to as many processors as the

user deems necessary. Additionally, workers requiring information with respect to the consuming and spawning of multiple tasks are passed this information upon initialization. Processing begins when the appropriate jobpool, i.e. directory, is seeded with the initial work, by the control program.

Jobpools

The control program creates the jobpools and establishes the structure of the jobpool system. Each jobpool is created as a unique subdirectory where the structure of the jobpool system is defined by the user. Included in our implementation of the directory tree structure are intermediate directories referred to as workpools. Each jobpool has a workpool associated with it. A workpool utilizes the same type of data structure as a jobpool but its use is somewhat different. It functions as a temporary work space where processing on a task takes place prior to moving the task to the next destination jobpool. Unlike the jobpool, a workpool is used by workers already assigned work. With this approach, the jobpool is used only to manage work that is available, while all processing occurs in the corresponding workpool. The structure may also be used for any temporary files required by the worker.

Since a jobpool is a subdirectory, each jobpool must be labeled with a unique name. For our implementation, the unique name assigned to the jobpool is a directory name. At the time of initialization the user must know the number of source and destination pools, including intermediate workpools, that will be required. Unique names are assigned to each jobpool which worker programs will use when accessing tasks.

Workers

In our paradigm, worker programs take the form of specialist parallelism in their function and are the processes which operate on the tasks found in the jobpools. In addition, workers move tasks between jobpools. It is possible for a worker to get single tasks from a jobpool, take exclusive control of a jobpool to get all or part of the tasks, place either single or multiple tasks

in a jobpool, and die gracefully. With the assistance of the monitoring primitives, the worker can write status information to the log files.

Recall that in a distributed control system, the clocking discipline is internal to the individual process. The input phase of a worker's clock enables a pseudo-blocked receive of input at the point of request for work. The receive is considered to be pseudo-blocked due to the low-level synchronization mechanism of the lock which maintains integrity of the jobpool. Here, when the lock can not be obtained, the worker backs off and then tries again. A blocked receive would use a queue to guarantee processes access to the jobpool in some ordered fashion. Workers consume a task found in their assigned jobpool, when they are idle and input is available. If there is no work to consume in a jobpool, a spin mechanism is employed. This low-level synchronization technique inhibits the input phase of the individual worker's clock until there is work available. In addition, the locking mechanism facilitates the blocking of the receive until a single worker has been given control of the jobpool.

When the worker reaches the output phase of its internal clock, data is exported by a non-blocking send. A task is moved to the next stage in the pipeline when work on it is complete. The fulfillment of this send request hinges solely on the operating system. The clock phase of other workers has no impact on when this communication takes place. There is no explicit dependency on other processes as there is in a centralized implementation.

Recall that a worker consumes a task by removing a file from its source jobpool, operates on the task in a workpool, and then places the results in the destination jobpool. Tasks are moved from one location to another by the use of the system call *rename*. Any file location on the UNIX system is described by a path which starts at the root directory of the tree structure and continues down through the subdirectories to the location of the file. The renaming of a file requires changing this path to reflect the new subdirectory name where the file is to be placed. This is seen at the user level as a file being moved from one directory to another. Between an initial source pool and a final destination pool there may be any number of interim moves to

facilitate processing of the task. The interim processing would be done using workpool data structures.

In a workpool there is no contention for tasks because tasks are assigned to specific workers upon removal from the original source pool. Each worker only has knowledge of its assigned task once it has been moved to the workpool. A worker never checks a workpool to see if others are using this shared resource. It is this intermediate location which allows for operations on the task, with no possibility of interference from other workers, and allows for the moving of the task from this pool to the next destination pool to be done strictly by the worker to which it is assigned.

A control structure is imposed on the paradigm in the form of naming conventions. Naming conventions for tasks are imbedded in worker programs. Care must be taken to insure unique file names for all tasks. Process id, job number, and machine name are but a few of the components available to assist in this requirement. Since workers dedicated to 1-to-many and many-to-1 transformations can generate new tasks, an ascending number scheme is imbedded in these workers. In our implementation, all tasks destined for a specific jobpool are given the same base name and then made unique by attaching a number as an extension. In addition, each time a file is placed in a destination jobpool, identifying information is appended to the name as an extension. This information takes the form of the host name of the processor where the worker is executing. Since the file name was unique when it was originally generated, it remains unique throughout processing. A nice side effect to this is the ability of the user to visually trace the progress of a task through the jobpools and identify which workers on which workstations are obtaining which tasks. This can be facilitated by the use of the log files made available in each jobpool. These log files remain behind in their respective directories at the completion of execution. Other techniques can be envisioned for the separation of control information and job data.

Communication Topology

The communication topology of the paradigm is based on the network file system (NFS). A request to move a file results in message packets sent to and received from the file server over the Ethernet connection. The Ethernet connection is a shared resource used by all workstations, including other workstations outside the SUN environment. Since both the Ethernet and the directory jobpools are shared resources, the possibility for contention exists. The system command used by a worker to move a file is an atomic action but this in itself does not guard against contention for a file. Problems may arise when several processes look into a jobpool and see the same file simultaneously. Only one process can actually move the file, but more than one may think they have done so. A locking mechanism is provided to ensure that there is no contention for the same file.

Locks

The simplest form of lock provides exclusive access by the process holding that lock to a data object or objects, in this case the data object is a directory [15]. By providing exclusive access, we ensure protection of the files contained in the directories. The locking mechanism we have employed uses a separate file lock associated with each jobpool. To gain access to the jobpool, a worker must obtain exclusive access to the lock file. This lock file is created the first time a jobpool is accessed and remains in the jobpool for the duration of the processing. Prior to locking a directory, a test must be made of the status of the lock file. If the lock is free then an attempt is made by the worker to set the lock. If the attempt is successful then the resource may be exclusively accessed by that worker. If the lock has been set prior to the interrogation then the test will notify the worker that some other process has already obtained exclusive use of the resource. The setting of the lock is not attempted unless the resource is free. Many processes may test simultaneously and see the resource open for access but only one process will obtain the lock and earn the right to exclusive access. Processes which have failed to obtain the lock continue the cycle of checking the status and then attempting to lock. The atomic

action of setting the lock is guaranteed by the operating system call to the SystemV function *lockf()*.

Once access to the jobpool has been obtained, a worker checks for tasks to consume. If a task exists, then the file is removed from the jobpool and the lock is released. Now, the process of setting the lock, consuming work and releasing the lock is an atomic action which is required to ensure integrity of the jobpool.

There are two different types of locks available in our implementation: a lock file and a directory lock. The lock file is used when a jobpool is accessed to obtain a single job. Here the directory is locked long enough for a worker to obtain a single job and then the lock is released. A worker dedicated to 1-to-1 transformations between pools would expect to find this type of locking mechanism in the source jobpool. A directory lock is used when a jobpool is accessed exclusively and released only when all tasks have been consumed from the directory. A worker dedicated to many-to-1 transformations will need exclusive access to a jobpool in order to accumulate all expected tasks. This worker will require a directory lock to facilitate this transformation. Since not all tasks will arrive at the source jobpool at the same time, the lock is acquired forever and released only when the worker completes all processing which has been defined by some user supplied parameter. In either case, if the directory is locked, no other process will have access to it.

The function of the lock file and the directory lock could be extended to handle the locking of a jobpool for different colored tasks. A lock for each color would be required to maintain integrity of the jobpool, as discussed earlier. In the case of directory locks, one worker for each color could lock the directory and obtain multiple tasks.

Depending on the implementation, when placing a new task in a jobpool the status of the lock may be ignored. Since we are using the file system to implement jobpools, the placing of a task into a jobpool is accomplished by moving a file from one directory to another. This action is guaranteed to be atomic by the operating system so there is no need for a lock.

As we noted before, in our implementation, there is no contention on the intermediate data structure we are using called a workpool. Each worker has already been preassigned the task prior to its placement in the workpool and knows nothing about the other processes using this temporary work space. In addition, each task has a unique name, so there is no need for a worker to acquire a lock prior to the processing of its task.

By creating a data structure where interim processing can take place without fear of contention, the shared resource of the jobpool is not tied up while a worker operates on a specific task. But exclusive access to the jobpool presents the potential bottleneck in this paradigm. It is to the paradigm's advantage to move the majority of the processing to a workpool where there will be no locking mechanism required. This isolates the potential bottleneck at the jobpool level where the only objective is to obtain work. This in turn minimizes the time a worker will spend stuck at the bottleneck and maximizes the amount of time a worker will spend working versus time spent in the communication portion of the process.

Load Balancing

In Cvetanovic's [16] work, some of the issues to be considered prior to mapping an application to a parallel processing environment include:

- the amount of parallelism inherent in the problem
- job partitioning or the decomposition of the problem into jobs
- the grain size of the jobs
- load balancing which includes:
 - allocation of jobs to processors
 - scheduling of jobs on the processors
- communication overhead

In a distributed system, load balancing and communication overhead are two major concerns. Included in load balancing are the two issues of assignment of workers to processors and

scheduling of jobs among workers. The goal in scheduling is to distribute the work evenly so that all processors keep working as long as there is work to be done. The larger the fraction of time processes spend working, the sooner the job gets done. Two options for scheduling that the user has available are static and dynamic allocation of tasks to processors [17]. This is opposed to processor management policies done at run time by the operating system which are broadly classified as preemptive and nonpreemptive scheduling [18]. We will look at the user's options, recognizing that once the processor obtains the tasks there are other internal scheduling policies occurring.

In static scheduling, all work is assigned to processes in advance of execution. This predetermined scheduling produces a static load at run time. With this approach, it is possible for some processes to finish their work and stand idle while other processes continue. This problem compounds if there are several processes running on the same processor. Here, it is conceivable that with a poor static assignment some processors might stand idle at a time when others are overloaded. This type of scheduling is only appropriate where the problem is partitioned well and program behavior is very well understood.

A more equitable distribution of work among processors might be realized by using dynamic scheduling. In dynamic scheduling, work is assigned to a processor when it becomes idle. There is the added overhead that the processor must go out and find work, but there is less likelihood of idle processors versus overloaded processors occurring.

In our paradigm, at the time of jobpool initialization, workers are statically assigned to processors and jobpools and there is no migration of processes. Thus, there is no physical movement of executables or rerouting of communication paths for workers at any time. Workers execute on individual workstations and multiple workers may execute on a single workstation. The distribution of workers to jobpools and workers to processors is entirely up to the user. With the assignment of workers to jobpools, dynamic mapping of tasks to workers will take place. Workers look in their assigned jobpools for work and are only idle when no work is

available. Multiple workers executing on a single workstation will be dynamically scheduled through the operating system scheduler.

No effort has been made to automate the identification of idle processors for the assignment of workers to workstations. The network file system (NFS) provides a remote procedure call environment which could be used to predetermine the state of each processor thereby ruling out any processor with work already assigned to it. Along these same lines, our implementation makes no effort to reassign worker programs in the event of other users requiring the use of an assigned processor once computation has begun.

The user may have an intuitive feel for assigning workers to processors for a specific application. With a pipelined configuration, one example might be to assign one worker from each pipeline stage to the same processor. As work propagates through the pipeline, some workers may finish their work while others are just starting. This could result in less contention for CPU time on any one processor. In addition, multiple identical workers assigned to the same jobpools may be less likely candidates for residing on the same processor.

Job Partitioning and Granularity

The key to developing efficient parallel algorithms for a distributed environment is to maximize concurrency and minimize communication. In order to minimize communication and to assist in the parallelization of applications, the user must have intimate knowledge of the control flow of their application so that partitioning of the application is done to accommodate efficient use of the system. This is even more critical in a distributed system due to the manner in which communication is accomplished.

Most applications lend themselves to one of two methods for partitioning computation: data-parallel and function-parallel [19]. Function-parallel computation deals with the partitioning of a program into subtasks which can then be executed in parallel. This particular type of partitioning is appropriate for programs which perform many different operations on the same data. This is consistent with Carriero's definition of agenda parallelism where an application is

partitioned in an agenda of tasks. Data-parallel computation involves the division of data among multiple processors. This approach to computation is appropriate for applications that perform the same operations repeatedly on a large amount of data. Data-parallel computation is ideally suited for either result or specialist parallelism.

Implied in the issue of partitioning is the concept of granularity. Granularity is often defined at three basic levels: fine, medium, and coarse [14]. Fine granularity is found in vector processors where the same operation is applied to multiple data simultaneously. Medium granularity, is defined at the operational level where independent instructions can occur simultaneously. Finally, coarse-grain is defined at the task level where multiple tasks can be executed concurrently. The choice of granularity depends on the number and type of processors available in the system configuration and on communication overhead. The grain size of a task given to a processor should take into consideration the possibility of a communication bottleneck.

There are two dimensions of granularity that must be investigated in our paradigm. In this thesis we will refer to the amount of time spent processing during the interval between synchronization occurrences as task size. Here we are interested in the total amount of time a task requires from the time it is consumed to the time results are placed in the destination jobpool. The second dimension of granularity relates to the total number of tasks processed by an individual worker for a given application. We refer to this dimension of granularity as task count.

King suggests that since communication is expensive compared to computation, it is important that the granularity of the tasks relative to the number of stages in the pipeline be sufficient to create a situation where there is a large ratio of computation to communication [4]. This addresses the second dimension of granularity which relates to the number of tasks found at a specific pipeline stage or the number of tasks processed in a jobpool.

Cvetanovic points out that there are several tradeoffs to consider when manipulating the grain size of a particular subproblem for execution [16]. In her paper, she demonstrates the

interaction and quantifies the influence of problem partitioning, allocation, and granularity on shared memory, multi-processor systems. As an example, the simple case of decreasing the grain size of computations on processors may result in more rapid completion of the computation. But on the flip side, this may result in more frequent communication which in turn slows a processor down. We will see that this phenomena occurred in our experiments with respect to both task size and task count. On the other hand, increasing the granularity for some algorithms may tend to reduce the communication overhead but it may well reduce potential concurrency. Here the risk is starvation by workers and a poorly distributed workload.

In the decentralized multidimensional pipeline defined in this thesis, communication has a "hands-off" approach where packets of data are deposited and retrieved independently. This approach does not support an intimate communication topology as required at the subroutine or function level within a program, for example. Nor does it support a data-parallel or fine grain size computation. However, it provides an improved method for information exchange at the task level, where coarse-grain parallelism is used for function-parallel computation.

SUMMARY

The primary purpose of this chapter was to present a detailed description of our paradigm. The characteristics of the paradigm as well as the structural, operational, and transformational primitives upon which the paradigm is based were addressed in detail.

The organizational framework is made up of two structural primitives: jobpools and workers. Jobpools contain independent but similar tasks. They are implemented through the use of directories. Files residing in each directory are seen as tasks for that jobpool.

Workers are assigned to a source jobpool and a destination jobpool and are implemented as processes assigned to individual workstations. Global memory for the processes is implemented using the file system. Workers are self-directed in their quest for work and facilitate transformations between jobpools.

The coordination of these transformations is accomplished by specialist workers who consume or spawn work. Methods for the consuming and spawning of tasks entail transformations such as 1-to-many, many-to-1, and 1-to-1. Operational primitives facilitate the distributed control of our paradigm.

The communication methodology of the self-directed worker is at the heart of the distributed control mechanism. Coupling this with a multidimensional pipeline leads to the flexible overlapping of communication and computation and allows for a variability of time for each pipeline stage. The multidimensional pipeline is implemented through the use of a directory tree.

The distributed control topology of our paradigm is argued to be a viable alternative to other recognized communication protocols such as the client-server topology. Our simplified approach requires no handshaking between processes which keeps message passing to a minimum. Handshaking adds to the complexity of the client-server topology and results in additional communication overhead. We propose that the overlapping of communication and computation leads to less potential for a bottleneck at the shared resource and a more efficient use of the distributed system.

In the next chapter we look at the characteristics of applications which lend themselves to the use of our paradigm. Several examples of problems found in the literature will be provided. In addition we will show how the coordination framework of the paradigm is used to map a specific application to the distributed system.

CHAPTER III

APPLICATIONS

In this chapter we will discuss the characteristics of problems which map well to our paradigm. The concepts of result, agenda, and specialist parallelism, and function-parallel partitioning are again addressed in the determination of applications best suited to our paradigm. An example of a distributed application from the literature is presented and an explanation of an implementation of the problem using our paradigm is imparted. Next, a brief discussion of the decomposition and implementation of a database search using our coordination framework is presented. Finally, the Design Centering application which we implemented for this thesis is discussed in detail. The application is decomposed into jobpools representing pipeline stages and the transformations provided by specialist workers for job flow between jobpools is developed.

CHARACTERISTICS

To use our paradigm, there is a need for some sense of the complexity of a worker versus the tasks it obtains. Ideally, a worker looks to its assigned jobpool and anonymously selects a task, creates output, and places this output in a destination jobpool. The refining of work by file name to aid workers in selecting particular tasks leads to a finer granularity. For the paradigm to be most effective workers should be able to select files by directory not by file name. As a practical matter, the less a worker has to look for a specific file, the more effective the paradigm.

Problems that lend themselves to coarse-grain, function-parallel partitioning will benefit most from the use of our paradigm. There are two levels of decomposition required for map-

ping these applications. The first level of decomposition is based on the pipeline and the second level is the contents of an individual pipeline stage. Decomposition into sequential stages in order to exploit the pipeline characteristic of the paradigm is necessary. Each of these sequential stages is made up of a collection of independent jobs or tasks which can be run in parallel. Tasks within a pipeline stage need to be self-contained. Communication overhead requires that these tasks be computationally independent guaranteeing infrequent task interactions. As discussed previously, the granularity of the tasks should ideally accommodate a high computation to communication ratio for the paradigm to be effective.

Agenda parallelism tends towards medium and coarse-grain problems and readily lends itself to our paradigm by focusing on an agenda of tasks which many workers apply themselves to simultaneously [10]. The pipeline characteristic of our paradigm is the vehicle for implementing the list of tasks found in agenda parallelism and the specialist approach to our workers makes it feasible to implement specialist and result parallelism using an agenda approach. An application which has an agenda of tasks could be implemented using separate jobpools for each item on the agenda. Ensembles of specialists workers would be assigned to individual jobpools to complete the tasks.

Agenda parallelism covers a variety of applications such as parametric sensitivity analysis, optimization, and computer aided design. Parametric sensitivity analysis [20] is used in many fields as diverse as economics to aerospace and involves trial runs using different input parameters to test a model's sensitivity. Optimization problems such as fault simulation [21, 13] and design centering [22] often entail Monte Carlo simulations [23] which generate a large number of statistically representative trials suitable for computing simultaneously. Computer graphics and imaging techniques such as ray tracing are also possible candidates for our paradigm [24]. In addition, database management [25] where files need to be built, joined, sorted and searched could be handled as well.

Recall, from Chapter I, that result parallelism focuses on the data structure which yields

the final results of a solution [10]. Each element is a separate process that leaves behind a single datum in the data structure as its contribution to the final result. This occurs upon completion of the process. Here, every data object is permanently associated with some process and processes communicate by referencing each other as elements of the data structure. It is used most effectively in problems which require a series of values where each value represents a piece of the whole solution and results are often stored in tables, vectors, or matrices. Applications using result parallelism often create large numbers of processes with relatively little computation being done by each process. This manifests itself in a granularity which is too fine to allow for efficient execution on a distributed system. To correct this inefficiency, one solution would be the transformation of the result parallel solution into an agenda parallel solution through the grouping of elements into larger data objects. Now a collection of worker processes can compute sub-blocks of the result. This provides more work per communication event and uses the distributed system more efficiently.

Specialist parallelism tends towards problems which have network-type solutions [10]. Typically, these applications exhibit the finer granularity found in data-parallel partitioning. Specialist parallelism assigns a worker to perform a specific task and is suitable for pipelining multiple transformations on identical tasks. Neural network emulators, graphing algorithms, and circuit simulations are all specialist parallelism candidates. Here, the system has well known communication paths between nodes and individual nodes have self-governing computations. Often a single process is mapped to a single node in the network and each process is only responsible for updating the state of its assigned node. Here, processes communicate by exchanging messages since no data objects are shared among processors. This approach is used in data analysis where data can be thought of as broken down into pieces and then organized into a hierarchical graph that represents the conceptual hierarchy of the problem. Carriero calls this form of software architecture a process trellis [10]. Intrinsic to this parallelism is the need for multiple transformations on a given set of data. One approach to implementing specialist parallelism with our paradigm is to exchange messages through distributed data structures.

Transformation of a specialist parallel solution to an agenda parallel solution may result in a more efficient implementation when using our paradigm. One approach is to transform node states into tasks which require updating and assign workers to do the updating process. Now, processes are no longer mapped to nodes so when nodes are added the problem grows in task size not processor size.

SELECTED EXAMPLES

Fault Simulation

Circuit faults caused by local disturbances has led to the development of distributed yield simulators for VLSI [13,21]. Building on previous work, Walker uses a yield simulator called VLASIC (VLSI Layout Simulation for Integrated Circuits) to develop the distributed version called DVLASIC [21]. This version is based on a master process which controls many daemon processes assigned to evaluate multiple Monte Carlo samples in parallel. A fabrication process description and the IC layout geometry is required for each defect analysis and is stored in a database. Walker's implementation uses a replicated database on each daemon process where the master initializes and distributes the process description and the daemons initialize the layout geometry. Daemons request work, perform analysis and return results to the master process. Defect analysis generates data structures containing circuit fault information that must be collected by the master worker.

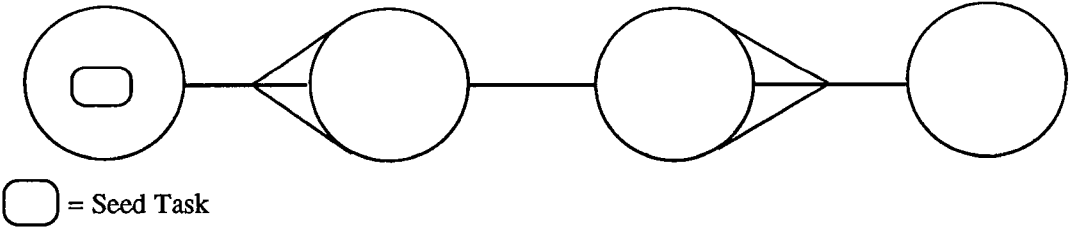
In Walker's implementation, Remote Procedure Calls (RPC) are used for the interface between master and daemon and a handshaking protocol is required for communication between the two. The master is the potential bottleneck for DVLASIC when many daemons request service from the master. It is this bottleneck we are trying to address with the distributed control of our paradigm.

Fault simulation provides an example for our paradigm of initialization. Workers initially require process technology information prior to fault analysis. This information could be con-

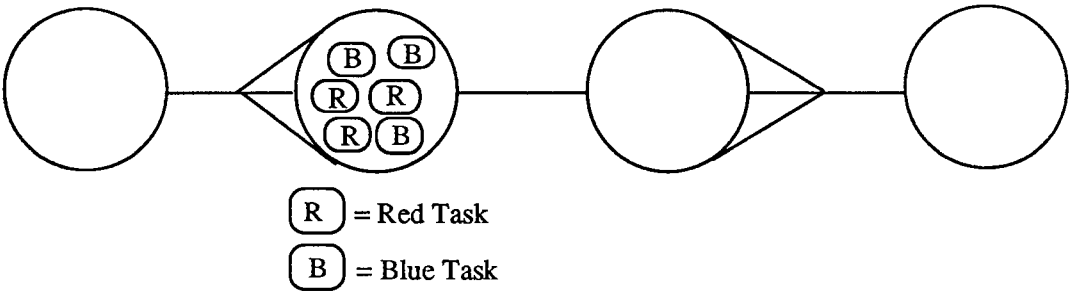
tained in particular colored tasks, say red, where there is one red task for each worker. Workers could be constructed to recognize red tasks as initialization information and blue tasks as work. After a worker consumes a red task from the jobpool it is then free to consume any number of blue fault simulation tasks.

One example of the implementation of this application would consist of four jobpools and their respective workers. The first jobpool requires 1-to-many workers, the second requires 1-to-1 workers, and the third requires many-to-1 workers. The fourth jobpool will contain the solution set. Although a graphical interface is not included as part of the paradigm implementation, Figures 5a through 5d represent a graphical depiction of the jobpools used for the fault simulation application. Using the graphical representations introduced in Figure 4 we construct a four stage pipeline using jobpools and transformations that take place between each jobpool.

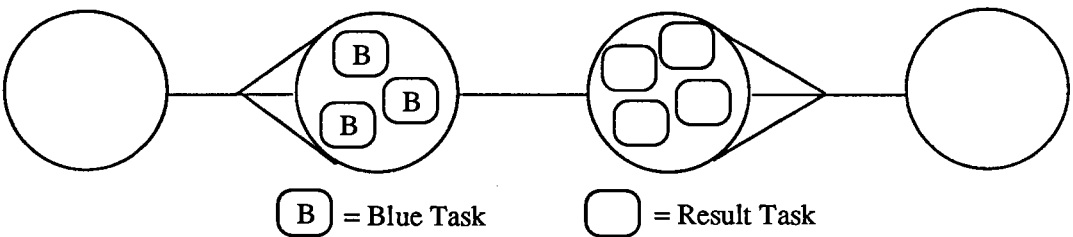
In Figure 5a an individual task containing process technology, defect statistics and simulation control information is deposited in the first jobpool. The 1-to-many worker which consumes this task will replicate and spawn all process technology information in the form of one red task per worker. These red tasks are placed in the second jobpool which is the source jobpool for the 1-to-1 workers. In addition, the 1-to-many worker would handle the Monte Carlo spawning of blue tasks to the second jobpool. Figure 5b shows red and blue tasks waiting to be processed. Multiple 1-to-1 workers, assigned to the second jobpool, first consume a red task and initialize their database. Next, each 1-to-1 worker consumes blue fault simulation tasks and spawns results which are placed in the third jobpool as depicted in Figure 5c. The first many-to-1 worker to obtain a task from the third jobpool will lock the directory and accumulate all results. We see in Figure 5d that final composite results will be deposited in the fourth jobpool.



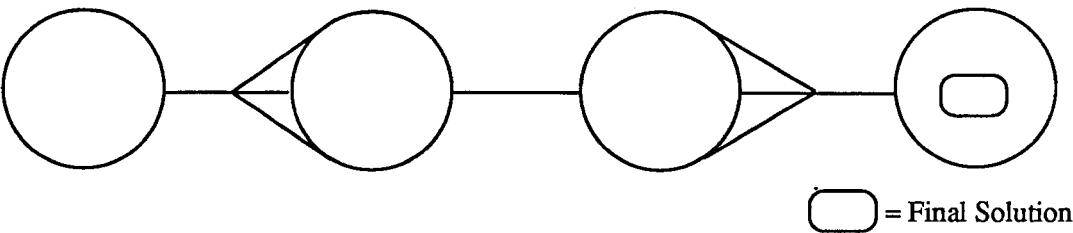
(a)



(b)



(c)



(d)

Figure 5. Job flow pipeline for the fault simulation application.

Database Searching

A completely different type of problem could entail the searching of a large relational database. Waltz proposes one such problem which requires the searching of a database made up of medical patient records [25]. Each record consists of features such as age, sex, patient history, diagnosis, test results and so on. To find diagnostic hypotheses for a new patient, the first step is one of initialization so that workers receive all relevant features of the new patient. Next, workers do comparisons and compute a numerical measure of closeness to each feature and sum the total for each record in the database. Patients whose total scores are closest to that of the new patient are selected from the database for analysis. This comparison function is just one example of what could be performed for an exhaustive database search. What is important is how we would coordinate and organize the strategy for manipulating the database and not the specific function being performed on the database.

In our paradigm, we could implement this search using four jobpools with a 1-to-many worker assigned to the first jobpool in the pipeline. Initialization would be performed, as in the fault simulation example, using two colors to differentiate between work and initialization tasks. After initialization tasks are generated, the 1-to-many worker would be dedicated to reading the database and spawning individual records as tasks for the second jobpool in the pipeline. Multiple 1-to-1 workers would keep track of the records which meet the comparison requirements and place information pertaining to records which match the search criteria in the third jobpool. A many-to-1 worker would specialize in collecting, compiling, and perhaps filtering the match information. Final composite results would be placed in the fourth jobpool.

In many cases it may be necessary to implement some extra synchronization to control the amount of data in global memory. This could take the form of an upper and lower limit. The upper limit avoids flooding the global memory and the lower limit guards against worker starvation.

DESIGN CENTERING: A DETAILED EXAMPLE

The application we have chosen to implement using the distributed control, multidimensional pipeline is a circuit optimization problem called design centering [22]. Design centering is a method to improve circuit performance by adjusting nominal values to improve yield. Component variation in electrical circuits can lead to variation in overall circuit performance. Often this results in a low yield of circuits which meet design specifications. This fraction of acceptable circuits defines the manufacturing yield. Design centering is one means of maximizing the yield by adjusting the nominal values of the parameters keeping tolerances fixed.

Most design centering methods are based on the random exploratory character of the Monte Carlo analysis [26, 22]. The yield maximization problem is very hard to solve because it involves the evaluation of an integral in multidimensional space. Monte Carlo analysis for yield estimation allows us to avoid the evaluation of the integral. Random sampling of the multiparameter component space simulates the manufacturing process and dispenses with the need for derivatives and/or linear search methods that can become cumbersome and complicated.

From the probability density function of the current nominal point, sample points are randomly selected using Monte Carlo analysis and concurrent circuit simulations are done on all sample points using the SPICE analysis package. No rewriting of SPICE was undertaken to accommodate this concurrency. Parallelism is exploited through the concurrent execution of multiple SPICE tasks running on multiple processors. Results of the simulations are then collected and used to determine the yield estimate for the current nominal point and the direction in which the nominal point should be moved in an effort to maximize yield. Yield is defined as the ratio of the number of points that meet specifications to the total number of points sampled. A new position for the nominal point is determined and the procedure is repeated until an acceptable yield has been obtained. Many methods have been proposed for adjusting the nominal point [27, 22]. The Center of Gravity method was chosen for its ease of implementation and its convergence properties [28].

Intense computational time spent evaluating each circuit simulation provides the incentive to parallelize this application. In addition, using a distributed processing system allows for many more sample points to be considered than when estimating circuit yield within the timeframe used by a uniprocessor. With the addition of more sample points the accuracy of the solution is expected to increase.

The flow of the design centering problem can be modeled by a cyclic pipeline using our paradigm primitives. This pipeline terminates when an optimum yield has been met as specified by the user. Decomposition of the problem into individual jobpools of tasks is as follows:

1. Monte Carlo generation of SPICE jobs.
2. Execution of SPICE jobs.
3. Analysis of output from SPICE jobs for pass or fail criteria.
4. Updating of the nominal point to describe the next circuit to be input to the Monte Carlo process for the next iteration.

Decomposition of the problem is followed by the assignment of workers to individual jobpools representing pipeline stages. Recall that workers are assigned a source jobpool from which to obtain work and a destination jobpool where the task is deposited upon completion of the computation.

The Monte Worker

Monte Carlo workers are dedicated to a 1-to-many transformation required for the spawning of many SPICE tasks from a single input specification. Workers get an input file from the assigned source jobpool and spawn multiple SPICE jobs. These jobs are placed into the destination jobpool which activates the next set of workers.

The number and size of the SPICE tasks spawned will play a role in determining how many workers will be producing tasks. Parallelization of the Monte Carlo sampling can be

facilitated by multiple input tasks being placed in the source jobpool.

The SPICE Worker

Each SPICE worker performs a 1-to-1 transformation which entails a SPICE circuit analysis on each task obtained from the source jobpool. Output from the execution is concatenated onto the original task and placed into the destination jobpool. SPICE workers continue with 1-to-1 transformations until all work is exhausted.

The Pass/Fail Worker

The pass/fail workers also perform a 1-to-1 transformation. Jobs are consumed from the source jobpool and analyzed for pass or fail by criteria provided by the user. Results of this pass or fail are concatenated onto each task and the result is deposited in the destination jobpool. Pass/Fail workers continue processing jobs from the source jobpool until the work is exhausted.

The Update Worker

An update worker performs a many-to-1 transformation requiring the source jobpool to be locked for the duration of the calculations. This exclusion of other workers from the source jobpool is performed by the first worker to obtain a task from the jobpool. The lock is not released after the task is consumed but instead remains locked for the update worker to obtain all tasks.

It would also be possible to have several update algorithms working concurrently. Here the idea of different colored tasks would come into play and a many-to-1 worker for each color would lock the jobpool. Each colored directory lock would be released after the appropriate worker finished consuming all expected work for the assigned color.

In our implementation the update algorithm performed by the workers is referred to as the Center of Gravity (COG) method [28]. Here, a center of gravity is computed for every component that is to be updated. Accumulated component values for both pass and fail circuits are compiled and are divided by the respective number of pass and fail circuits. The nominal point

for each component moves in the direction along the line from the center of gravity for the fail points towards the center of gravity of the pass points.

If the number of pass circuits meets the yield criteria of the user then processing halts, otherwise another iteration is required. For the iterative process, a single task is produced from the new nominal points generated by the COG method. This task is placed in the source jobpool for the Monte Carlo workers to begin the next design centering iteration.

Figure 6 represents a graphical depiction of the flow control of the design centering application. Again, we have used the graphical representations introduced in Figure 4 to construct the four stage pipeline. This figure introduces the use of a conditional worker.

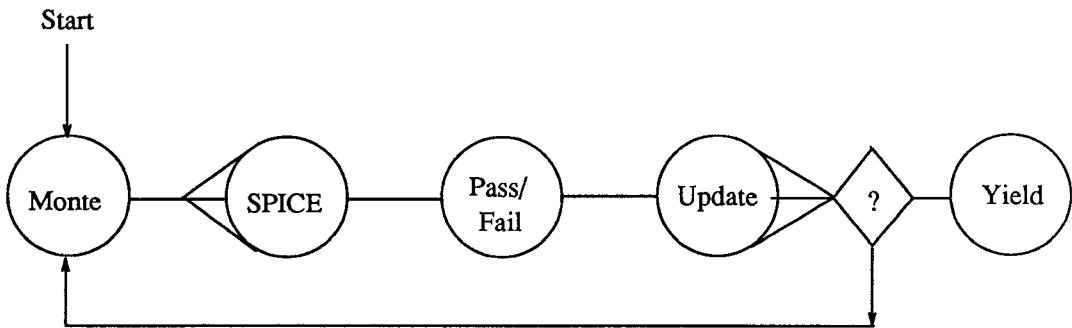


Figure 6. Job flow pipeline for the design centering application.

SUMMARY

In this chapter we have addressed the question of what types of applications can benefit from this paradigm. Two examples were briefly discussed and suggestions as to how they would be implemented were presented. The design centering application was introduced as a representative example for the types of problems which will benefit from this paradigm. Decomposition of the design centering problem is a working example of how an engineering application maps to a distributed system using the coordination framework developed in Chapter II. In the next chapter we introduce the actual experiments which were developed from the design centering application to study the organization and coordination of the paradigm.

CHAPTER IV

STATISTICS

INTRODUCTION

In this chapter queueing network theory is used to develop two models that serve as analysis tools for determining system behavior during the experimental process. A model to describe the topology of the physical system is developed and is used as a template for the instrumentation of the experiments. We will refer to this model as the *Complex Model*. A second model which is simpler in form is introduced and a closed form expression for this model is derived and used to predict theoretical behavior of the system for large numbers of workers. This model will be referred to as the *Simple Model*. Experimental basis as well as instrumentation and the experimental environment are discussed in detail. Performance measures calculated from experimental data are presented and analysis of our implementation is discussed. Overall system behavior as well as interesting performance measures are presented in the form of graphs and tables.

To obtain information about a general stage in the distributed control, multidimensional pipeline we chose the SPICE stage of the design centering application, introduced in Chapter III, for our data collection. Instrumentation was applied to the second jobpool of this pipeline, i.e. the SPICE stage. Workers obtain a task from the source jobpool, pass the task to the SPICE analysis package for processing, and place the results in the destination jobpool. The fact that we are interested in exploiting already coded work makes this stage especially representative of what we are trying to accomplish with this paradigm.

COMPUTER SYSTEMS ANALYSIS

Queueing Theory versus Queueing Network Theory

To demonstrate that parallelism has effectively been exploited, performance measures must be resolved. A tool is required in order to analyze these performance measures. There are two theories from which to choose when it comes to analyzing computer systems: queueing theory [29] and queueing network theory [30].

Queueing theory is based on the study of random events and the analysis of the probabilistic, stochastic parameters of a computer system. These random processes are used to model queues and are described by differential equations. Much of queueing theory revolves around the modeling of a computer system as a single service center made up of a *queue* where transactions wait to be serviced by a *server*. Due to the probabilistic nature of the service center, the characteristics are quite complex. This requires sophisticated mathematics for the analysis of queueing theory models which often results in detailed performance measures based on distributions.

In queueing network theory, directly measured values are used to replace the stochastic parameters in queueing theory. Equations have been derived that relate quantities measurable in the network and are used to characterize the performance of the system. Queueing network theory is based on the modeling of computer systems using a network of queues. The network of queues is comprised of multiple service centers and customers which represent transactions requiring service. The characteristics of the service center used in queueing network theory are quite simple and performance measures are in the form of averages.

Queueing network theory has, in recent years, become the tool of choice for the analysis of computer systems because of its simplified nature of network description, parameterization and evaluation. In addition, in the context of computer modeling, computer network models have been shown to be accurate within a level required for a wide variety of design and analysis

applications. For these very reasons we have chosen to use it as our tool for the analysis of our implementation.

Queueing Network Models

There are three major uses of queueing network models in studying the performance of a system: performance calculation, consistency-checking, and performance prediction [31]. Performance calculation utilizes directly measured values and equations relating these values to compute quantities not measured. Consistency-checking is a method of validation. Comparing the model performance to the actual measured performance of the system is the approach used to identify inconsistencies in the model and its accuracy in calculating performance measures. Performance prediction uses a validated model to estimate performance measures for the system where measured data is unavailable.

The first use of the Simple model will be for validation. This is done to determine the accuracy of the model in calculating performance measures. The validation approach we will use for the model starts with experiments that apply a typical workload to the system. During the experiment, performance quantities are measured. Next, measured parameters are applied to the model and compared to actual system performance. If the validation phase is successful, we will employ the model for predicting future behavior.

The Physical System

The first step of queueing network theory is to define a model of the physical system which will facilitate the data collection for performance measures [30]. The basic component in queueing network theory is the service center. There are two types of service centers: queueing and delay. Figure 7 illustrates both centers. The queueing service center is made of a queue where workers wait to receive service from the server. Requests for system resources which require sequential servicing are represented by *queueing service centers*. On our local area network, the directory file server must queue all worker requests for NFS transactions. As an

example, any request to move a file, open a file, close a file, or search a directory would have to enter a queue along with other similar requests and wait to be serviced. Any communication with the file server can be thought of as time spent in a queueing service center. This could be further broken down into time spent waiting in the queue and time spent being serviced.

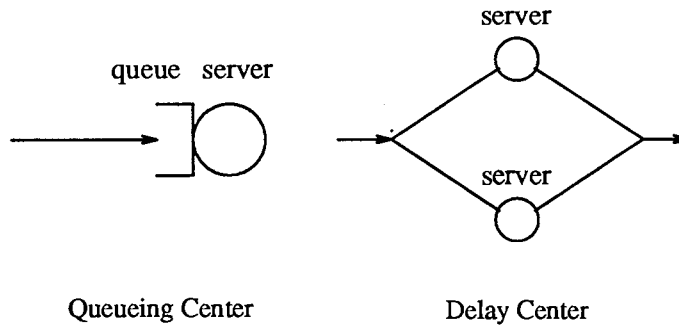


Figure 7. Queueing and delay service centers [30].

Delay service centers suggest concurrent activity where workers are allocated their own server. It is assumed that any computation, or work, done locally at the workstation requires no competition for service, provided only one worker is assigned to a workstation. Processing at delay service centers, across multiple workstations, can be viewed as parallel processing since there is no contention for the resources. The time spent at a delay service center is exactly the amount of time spent processing a customer's service demand. For our purposes, a customer in queueing network theory is the same as a worker in our implementation. The delay center in Figure 7 reflects this multiple server, concurrent processing attribute.

QUEUEING NETWORK MODELS

The Single Class Queueing Network

Using the two basic components described above, multiple, or separable networks can be combined to model an entire system. The network can be developed by connecting the components in the same configuration as the system. We have used this approach to formulate two closed, single class, queueing network models [30] for our physical system: the Complex model

and the Simple model. Single class models view customers as indistinguishable from one another. In our implementation, the workers are identical except for the tasks they consume and tasks are so similar as to be inconsequential. Service demands at each service center represent the average customer, or worker, in the system. Since workers are sufficiently similar, service demands can be viewed as belonging to a single set. The fact that the model is closed specifies that the number of workers in the system is bounded, or fixed, and no external arrivals or departures from the system are allowed. Queueing network parameters which will be referred to throughout this thesis are defined in Table I.

TABLE I
QUEUEING NETWORK PERFORMANCE PARAMETERS [30]

service center parameters	D_k	avg service demand at center k
	V_k	avg number of visits to center k
	U_k	avg utilization of center k
	R_k	avg residence time at center k
	S_k	avg service requirement per visit at center k
	I_k	avg number of idle workers at center k
	Q_k	avg number of workers at center k
cycle parameters	X	cycle throughput
	I	number of workers idle in all queues
	C	cycle time
	N	number of workers active in the system

Network Analysis

The analysis of a closed system begins with the fact that the number of workers is known. There are also several laws that are central to the parameters used to evaluate queueing network models. One of these is *Little's Law* [30]. Little's law states that the average number of requests in a system, Q , is equal to the product of the throughput, X , of the system, and the average residence time, R , of the request in the system. This can be stated as

$$Q = XR$$

Another law of importance is the *Forced Flow Law*. This law states that throughputs or flows throughout the system must be proportional to one another. The throughput of center k is stated in Eq.(1) as:

$$X_k(N) = V_k(N)X \quad (1)$$

In addition, we will assume the system satisfies the *flow balance property*. This states that arrivals equal departures at every center in the system.

The technique used to solve a closed queueing network model is called *mean value analysis* (MVA) and is outlined in detail in Lazowska's book on queueing network theory [30]. All useful relationships between the parameters used in the MVA solution technique are listed in Table II. Three equations from this table form the backbone for this analysis. Two of these equations are direct applications of Little's law. Eq.(2) is a direct application of Little's law applied to a closed queueing network as a whole:

$$X(N) = \frac{N}{\sum_{k=0}^K R_k(N)} \quad (2)$$

where $X(N)$ is throughput as seen at the cycle level with N workers active in the network and $R_k(N)$ is the residence, or response, time for service center k . For a queueing center, such as that shown in Figure 7, throughput is the rate at which the resource is satisfying requests and residence time is the average length of time a request spends at the center during a single visit. The average number of requests at a queue includes both requests waiting in the queue and those being serviced. Applying Little's law to the service centers individually yields Eq.(3):

$$Q_k(N) = X(N)R_k(N) \quad (3)$$

where Q_k is the average number of requests at center k . Individual service center residence times, R_k , are defined in Eq.(4) where $Q_k(N-1)$ is the average number of requests in the queue upon the arrival of a new worker. The average service demand at center k is defined as D_k . For closed, separable networks we can say that any time a worker arrives at a queueing service center we are guaranteed that the worker was not in the queue at the time of the arrival.

Therefore, the average queue length, upon an arrival of a new worker, can be no more than the average number in the queue with $N-1$ workers in the system [30]. This is reflected in the following equation.

$$R_k(N) = \begin{cases} D_k & (\text{delay centers}) \\ D_k [1.0 + Q_k(N-1)] & (\text{queueing centers}) \end{cases} \quad (4)$$

The MVA algorithm we are using is based on the iterative application of Eqs.(2)-(4). These equations compute residence times, system throughput, and average queue lengths for N customers in the system given the average queue lengths for $N-1$ customers in the system. This iterative process is based on the knowledge that the queues are empty for the first iteration and that D_k , average service demand for any center k , is known.

We apply a modified MVA disk subsystem model to our shared resource of the directory file server. This disk subsystem is composed of a queue and a disk. Workers wait in the queue prior to obtaining access to the disk. Once workers obtain access to the disk there are four basic components which make up the *effective service demands* at the disk: seek, latency, transfer, and contention. Contention is defined as the time spent waiting for access to the channel by a request associated with the disk. In other words, average effective service demand at center k can be expressed as

$$D_k = V_k \left[seek_k + latency_k + transfer_k + contention_k \right]$$

where V_k is the average number of visits to center k .

To compute the MVA solution exactly for a disk subsystem it is suggested that the effective service demand be computed iteratively using an estimate for contention. This value is then used in the MVA algorithm, as discussed above. The mechanism for including the effect of the queue in the disk subsystem where workers wait for access to the disk is provided by the MVA solution. We will see presently that our accumulated measurements for each queueing service center combine to represent the total effective service demand for the file server includ-

ing contention. Weighted averages calculated from all experiments were used for effective service demand times instead of iteratively computing these values. We recognize that there will be some double counting of time for workers stalled because of contention versus workers waiting in the queue. This will result in a pessimistic weighted average used for the MVA estimate of demand service time. But comparison of theoretical values to experimental results will show that this does not significantly affect the behavior of the model.

TABLE II
NETWORK PARAMETER RELATIONSHIPS [30]

$X_k(N)$	$= V_k(N)X$	(1)
$X(N)$	$= \frac{N}{\sum_{k=0}^K R_k(N)}$	(2)
$Q_k(N)$	$= X(N)R_k(N)$	(3)
$R_k(N)$	$= \begin{cases} D_k & \text{(delay)} \\ D_k [1.0 + Q_k(N-1)] & \text{(queueing)} \end{cases}$	(4)
$Q_k(0)$	$= 0$	(5)
$U_k(N)$	$= X(N)D_k(N)$	(6)
$I_k(N)$	$= Q_k(N) - U_k(N)$	(7)
$D_k(N)$	$= S_k V_k(N)$	(8)

The Complex Model

Separable queueing networks used to define a larger system, such as those seen in Figure 8, allow us to characterize the complex system behavior taking place during an experiment. To develop the topology of the system specifically for our instrumentation we assume (falsely) that the service centers are separable. Figure 8 of the Complex model characterizes the behavior of a worker, with respect to processing requirements, for the SPICE stage of the pipeline we have chosen to characterize. The behavior of the system will be captured implicitly in the measurement data used to parameterize individual service centers. Using Figure 8, we can define a *cycle* as starting and ending just prior to the delay service center labeled "obtain lock". Workers begin a new cycle each time the process of obtaining the lock file in the source jobpool is undertaken. Later it will be seen that a basic unit of performance measurement is one cycle time.

Identification of individual service centers as seen in the Complex model plays an important role in determining where primary and secondary bottlenecks exist during the data analysis. By defining work, which is processed at delay centers in parallel, and communication, which must be handled sequentially at queueing centers, we are able to differentiate between the critical attributes of the system. This makes the Complex model useful in defining a topology of the physical system and necessary for determining how instrumentation must be distributed throughout the experiment. This, in turn, allows us to microscopically study the behavior of the system at the service center level. Unfortunately, the characteristics of the model cannot be represented as a mathematical closed form expression. Therefore we will define a second and simpler model for analyzing the behavior of the system while using the Complex model to describe the topology of the physical system as well as a template for our instrumentation.

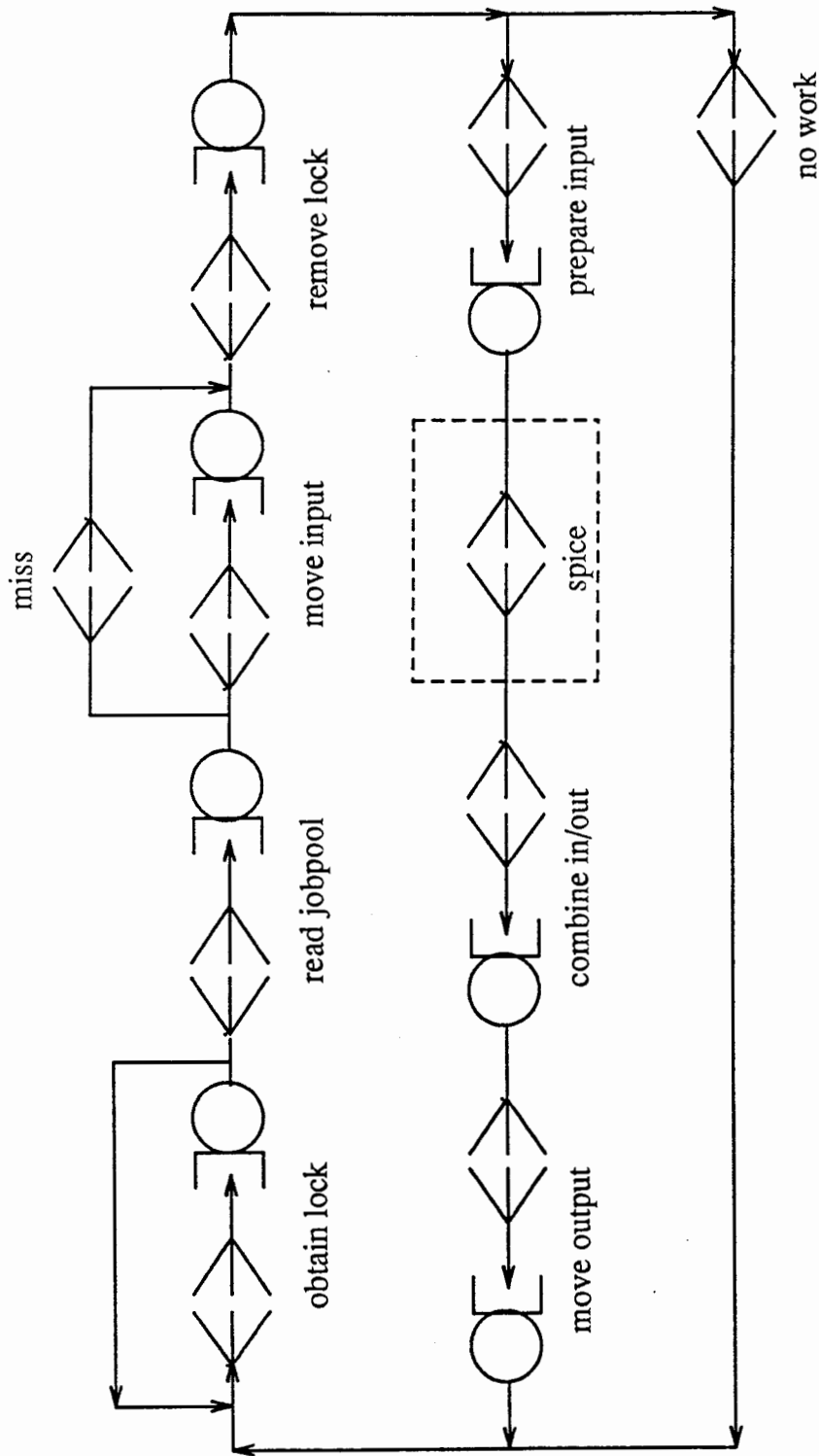


Figure 8. The Complex model.

The Simple Model

In our implementation, the physical system is made up of only one large queue for all NFS transactions. Delay service centers in the Complex model accurately reflect parallel processing within the system but the individual queueing service centers do not. In the Complex model the queueing effect is only factored in at the individual service center. In an effort to model all queueing service centers as a single file server, we collapse the entire Complex model into the easily evaluated Simple model, shown in Figure 9. Later in our discussion we will compare the Simple and Complex models and see how well each reflects the behavior of the physical system.

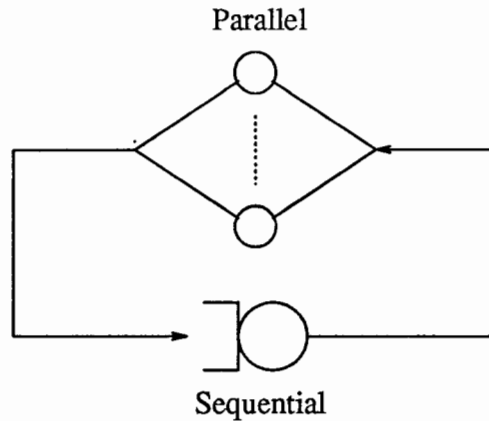


Figure 9. The Simple model.

The Simple model is also a closed, single class, queueing network model. But because of its simplicity, applying the MVA analysis technique to this network yields a mathematical closed form expression. We begin by deriving an expression for total cycle time. For this derivation we define the following:

- T = total time interval
- W = total amount of work accomplished in seconds
- n = number of total cycles completed
- N = number of workers in the system
- $N*T$ = total CPU time
- $I(t)$ = number of workers idle at time t

From the topology of the instrumentation, as defined by the Complex model, the sum of the

service demand for all delay centers is represented by d and defined by

$$d = \sum_{k \in \text{delay}} D_k(N) = \sum_{k \in \text{delay}} S_k V_k(N) \quad (9)$$

Since all delay centers represent work done locally at the workstation, where there is no contention or queueing, we can represent all work done in the system as taking place at multiple delay centers executing in parallel. Summing the average service demand times for all delay centers in the Complex model gives us a single value for average delay or work for a single cycle. The value q represents the sum of the average effective service demand for all queueing service centers in the Complex model and is defined by

$$q = \sum_{k \in \text{queueing}} D_k(N) = \sum_{k \in \text{queueing}} S_k V_k(N) \quad (10)$$

This provides a single cycle value for the average effective service demand for the disk, representing the directory file server, as shown in the queueing service center in the Simple model. This effective service demand includes the disk contention component discussed earlier. The MVA solution evaluates the effects of the queue. In Eqs.(9) and (10), S_k and $V_k(N)$ are average service time and average number of visits to center k respectively. We will see from our experiments that the average number of visits to center k is a function of the number of workers for particular centers and a primary value for predicting experimental results.

In order to derive an equation for total time for the Simple model we will refer to Figure 10. Figure 10 is a Gantt chart describing the general behavior of the system. It shows that given N workers, some amount of total work, W , gets done within time T where the rectangles represent work done by a specific worker. The amount of work done from zero to time T is expressed as

$$W = \int_{t=0}^{t=T} N - I(t) dt$$

$$W = \int_{t=0}^{t=T} N dt - \int_{t=0}^{t=T} I(t) dt$$

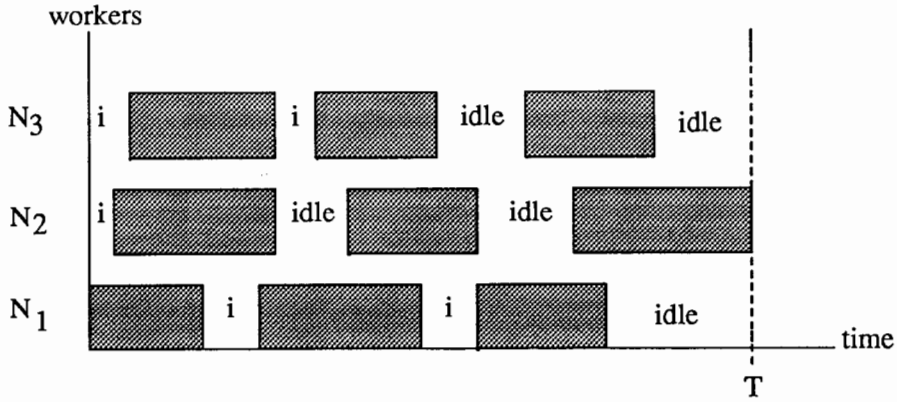


Figure 10. Gantt chart of idle and work time.

$$W - NT = - \int_{t=0}^{t=T} I(t) dt$$

Now let $I(t)$ become some average value of I such that

$$I(t) = \bar{I}$$

and replacing $I(t)$ with \bar{I} in the equation yields

$$W - NT = -\bar{I}T$$

and simplifying

$$T = \frac{W}{N - \bar{I}}$$

where W must represent all service demand time accumulated for both queueing and delay centers. In the case of the Simple model

$$W = n * \text{average time through a cycle}$$

or

$$W = n (d + q)$$

where n is the number of cycles, where 1 task is completed per cycle. The equation for total time becomes

$$T = \frac{n (d + q)}{N - \bar{I}} \quad (11)$$

For a multitasking operating system, the numerator of Eq.(11) includes contention components due to a common resource. The numerator is an estimate of how long it takes to complete n tasks, where 1 task is completed per cycle. The term \bar{I} , found in the denominator, can be identified as an estimate of the effects of worker starvation and synchronization. Synchronization blocks workers from obtaining access to the disk thus rendering them idle.

Now that we have an expression for total time we can derive the closed form expression for the Simple model using the equations from Table II provided for us by queueing network theory. Table II lists the equations of interest. For the next set of equations we will continue to refer to the number of workers active in the system as N . In the following development the dependence on N , as expressed in the equations of Table II, is understood and will be retained only for clarity. We begin with no workers in the system and all queues empty. Simply stated

$$Q_k(0) = 0 \quad (5)$$

For this derivation, the values of q and d are defined in Eqs.(9) and (10) respectively. From Eq.(4), average residence time for the single file server queueing service center can be represented by

$$R = (1.0 + Q(N-1)) q \quad (12)$$

For clarity we define

$$\alpha \equiv \frac{\text{delay time}}{\text{queueing time}} = \frac{\sum_{k \in \text{delay}} S_k V_k(N)}{\sum_{k \in \text{queueing}} S_k V_k(N)} = \frac{d}{q} \quad (13)$$

Using Eq.(2) we will express the Simple model throughput as

$$X(N) = \frac{N}{\sum_{k=0}^N R_k} = \frac{N}{d + R} = \frac{N}{d + (1.0 + Q(N-1)) q} \quad (14)$$

Utilization for the queue service center is represented by Eq.(6)

$$U = q * X(N) = \frac{N}{\alpha + 1.0 + Q(N-1)} \quad (15)$$

Substituting Eq.(12) and Eq.(14) into Eq.(3) yields the average number of workers at the

queueing service center

$$Q = RX(N) = \frac{N(1.0 + Q(N-1))}{\alpha + 1.0 + Q(N-1)} \quad \text{for } N \geq 1 \quad (16)$$

which includes both idle workers and those being serviced. Average number of workers idle at the queueing center is defined in terms of utilization and queue length by Eq.(7) where

$$\bar{I} = Q - U = \frac{N Q(N-1)}{\alpha + 1.0 + Q(N-1)} \quad (17)$$

Substituting Eq.(17) into Eq.(11) yields

$$T = \frac{n(d+q)}{N - \bar{I}} = \frac{nq(\alpha+1)}{N - \frac{NQ(N-1)}{\alpha + 1.0 + Q(N-1)}}$$

which after simplification becomes

$$T = \frac{nq(\alpha + 1.0 + Q(N-1))}{N} \quad (18)$$

To express T(N) in terms of known, measured values of q and d , we must find an expression for Q in terms of α . For $N \geq 1$ the general form for Q is (see Appendix for proof)

$$Q = \frac{N\alpha^N + (N-\alpha) \sum_{j=0}^{N-1} \frac{N!}{j!} \alpha^j}{\sum_{j=0}^N \frac{N!}{j!} \alpha^j} \quad \text{for } N \geq 1 \quad (19)$$

Equivalently, total time can be expressed as

$$T = nq * \left[1 + \frac{\alpha^N}{N! \sum_{j=0}^{N-1} \frac{\alpha^j}{j!}} \right]$$

Our closed form expression of the MVA solution for a single cycle of the Simple model then becomes

$$\text{cycle time} = q * \left[1 + \frac{\alpha^N}{N! \sum_{j=0}^{N-1} \frac{\alpha^j}{j!}} \right] \quad (20)$$

Turning to the well known Amdahl's Law [32] the execution time for N processors can be expressed in terms of the fraction

$$\alpha \equiv \frac{T_p}{T_s}$$

where T_p is the time the program spent executing code that can be enhanced by parallelization and T_s is the time the program spends executing serial code that cannot be parallelized. The execution time for a parallelized program is expressed as

$$T_N = T_s * \left[1 + \frac{\alpha}{N} \right] \quad (21)$$

where T_N is the total execution time when N workers are assigned to the problem. Our Eq.(20) for cycle time is consistent with the expression for T_N in Eq.(21). Our serial factor, nq , is equivalent to Amdahl's total serial time, T_s . Representing the body of Eq.(20) as

$$\left[1 + \frac{f(\alpha, N)}{N} \right] \text{ where } f(\alpha, N) = \frac{\alpha^N}{(N-1)! \sum_{j=0}^{N-1} \frac{\alpha^j}{j!}}$$

is a refinement of

$$\left[1 + \frac{\alpha}{N} \right]$$

in Eq.(21). From Eq.(13) we see in our expression for cycle time that α is now a function of the number of processors.

Performance Metrics

Now that we have defined the model we look next at the actual experimental performance metrics. Using Figure 8 for the representation of the model used to describe our instrumentation we define the *effective cycle time* for an experiment. This can be thought of as the average completion rate of tasks. More precisely, the effective cycle time, $C(T', t, N, A)$, of application A , requiring t tasks to be processed using N workers is defined as

$$\text{cycle time} \equiv \frac{T'}{t * N}$$

The total amount of time accumulated for all workers for a specific experiment is defined as T' , where

$$T' = \sum_{e=1}^N T_e$$

and T_e is the total run time for the executable assigned to processor e . From this definition, T , the time required for N workers to finish all tasks can be expressed as

$$\frac{T'}{N} = T$$

and

$$\frac{T}{t} = \frac{1}{\text{throughput}}$$

If a task is completed every y seconds, the effective cycle time = y . This value could also be thought of as the inverse of throughput.

In order to determine whether the parallelized solution of an application is really worthwhile, some measure of performance must be defined. Speed is the definitive value when discussing the performance of a program. For our experiments, speed is quantitatively measured in the form of effective cycle time. Comparing the speed of the parallel solution to the speed of the sequential solution is referred to as *speedup*. The speedup of application A , for t tasks, using N processors is defined as

$$S(t, N, A) \equiv \frac{C(T', t, 1, A)}{C(T', t, N, A)}$$

As discussed earlier, speedup is typically less than N due to the overhead of communication as additional processors are used in the parallelizing of an application. In our calculations for speedup, uniprocessor effective cycle times were established per input deck. We saw no need to reevaluate the uniprocessor cycle time for a particular input deck because this value does not vary with respect to number of tasks. For each SPICE simulation these performance measures were used for all speedup analyses.

EXPLANATION OF EXPERIMENTS

Instrumentation

The cornerstone to our experimental instrumentation is the UNIX system call *gettimeofday* used in obtaining time intervals within the code. Total execution time for a worker was broken down into two categories: communication and work. These two categories reflect the various delay and queueing service centers depicted in Figure 7. Time collected at queueing service centers was directly related to NFS processing requests and all file server requests were viewed as queueing service center requests. Examples of these types of requests are: opening a file, reading a directory, moving a file, and closing a file. All computations fall into the category of delay center time intervals.

Calls to *gettimeofday* were distributed in an effort to minimize the impact on measured code. Accumulated service times for each queueing and delay service center shown in Figure 8 were collected. One important objective was to account for all executable run time. For each SPICE worker executable, a total run time was obtained and checked against the accumulation of total time for all service centers within the same executable. Differences for the two values of less than 3% were consistent across all experiments.

Measured parameters obtained from our instrumentation are listed in Table III. We will refer to the total number of visits to each service center as V_{Tk} . The total accumulated time spent at each service center is D_{Tk} . We remind the reader that effective service time for a queueing service center includes the component of contention for the file server. Total effective service time was obtained on a per worker basis for each experiment. From these aggregate values we can calculate service center parameters defined on a per-cycle basis for any individual service center k .

TABLE III
QUEUEING NETWORK MEASUREMENTS

system measures	T	total execution time
	N	worker population
	D_{Tk}	total effective service time at center k
	V_{Tk}	total number of visits to center k
	n	total number of completed cycles.
	t	total number of tasks obtained
	m	total number of attempts for the lock

By using the two measured parameters mentioned above, S_k can be calculated by

$$S_k = \frac{D_{Tk}(N)}{V_{Tk}(N)}$$

where S_k is the average service requirement per visit at service center k . The average number of visits to center k as defined by $V_k(N)$ is calculated from

$$V_k(N) = \frac{V_{Tk}(N)}{n}$$

where n is defined as the total number of completed cycles. System level parameters are aggregate values accumulated over an entire experiment. Cycle parameters are aggregate values accumulated over a single cycle. A cycle-level interaction would be defined with reference to a single cycle. By obtaining direct measurements of total effective service demand time and total visits relative to each center, we can compute solutions to all other performance measures, listed in Table II, through the parameters of V_k and S_k . These two values are related by

$$D_k(N) = S_k V_k(N) \quad (8)$$

where $D_k(N)$ signifies the effective service demand at center k with N workers active in the system.

There were some difficulties in separating delay time from queueing time in the code. An example of this is the actual call to SPICE. Timestamping was done prior to the call to SPICE and directly upon the return from SPICE. The distinction of work versus communication intervals inside the analysis package was not available. The time spent loading the SPICE binary

into memory was not a value that was accessible. Consequently, the entire time spent using the SPICE analysis package had to be accumulated in a single delay service center. Other problems entailed atomic system calls which could not be broken down into their work and communication components. All system calls were handled as communication or queueing service center requests.

In addition, it was observed during data analysis that isolated queueing service centers showed unexplained deviations from expected behavior. The source of these deviations is unknown and can only be explained as systematic failure in the measuring process. Although these deviations exist, we will see that the single server, multiple delay center model of Figure 9 is a resilient model for the overall behavior of the experiment in spite of these system failures and proves to be insensitive to them.

Experimental Environment

In an effort to isolate the test system, logins were inhibited to all workstations connected to the Sun Local Area Network. Nineteen processors were available, seven 3/110s and twelve 3/50s. During the course of the experiments it was determined that there was no significant difference in the computation time between the two types of processors. But in an effort to maintain some sort of homogeneity, all 3/50 processors were used exclusively for experiments requiring less than twelve processors. Only experiments that required more than twelve processors used 3/110s. A SPARC workstation was used as the directory file server and a dedicated 3/50 was used as the monitor for running the experiments.

Traffic on the Ethernet was not curtailed for other systems connected to it and probably played a role in some of the irregularities seen in the data. Due to the fact that the network is never dedicated strictly to testing we were at no time working under isolated, ideal conditions. Sendmail, backups and calendar updates took place when required. Every effort was made to work around large jobs that were scheduled on a regular basis and for which the scheduled starting times and duration were known.

Prior to assigning workers to processors, a Monte Carlo worker generated a specified number of tasks from a given input deck and placed all tasks in the assigned jobpool. Using this approach, the source jobpool for the SPICE worker contained all tasks to be completed at the time of initialization. Initialization of the SPICE worker took place via the *rsh* command which was implemented by a shell script cycling through a list of available processors. SPICE workers were assigned one worker to a processor to realize maximize speedup. Assigning one worker to a processor ensures a dedicated processor to an individual worker. More than one worker to a processor would have complicated the data analysis by introducing the variable of timeslicing between executables on a single workstation. During the course of the experimental explanation worker and processor are used synonymously.

Spice workers obtained tasks from their assigned source jobpool, initiated the SPICE analysis package, and attached the results from the SPICE analysis to the original task before moving it to the destination jobpool. All workers were killed after all tasks were accounted for in the destination jobpool. A routine was used to check the number of files in the destination jobpool every y seconds. The value of y was chosen as large in the beginning of the experiment. This guaranteed that the file server spent minimal time implementing the request of the kill routine to count the number of tasks in the destination jobpool so as not to impact the experiment. To keep idle time to a minimum, the time y was made smaller as the experiment progressed in order to kill the workers as soon as possible after all tasks were completed. Time y was made smaller as the experiment progressed in order to kill workers as soon as possible after all tasks were completed. This was in an effort to keep idle time to a minimum.

Due to the inaccuracy of the process used to kill the workers, information accumulated in our data may have been obscured. Ideally, the amount of time a worker spends looking for work in an empty jobpool would be accumulated as idle time. In our implementation, obtaining the lock, reading an empty directory, and unlocking the lock file is seen as accumulated time at the respective service centers. Idle time is not accumulated as a separate interval of time.

Since workers were not killed until all tasks were accounted for in the destination jobpool, there was a lag time imbedded in every experiment where all workers were idle. A shell script was used to cycle through a list of the appropriate process identification numbers and workers were killed via the *rsh* command. This approach interjects a fairly constant amount of time between each kill command. This means that workers were not all killed at the same time and some workers accumulated more idle time than others. With this approach, the amount of time a worker spends idle is a random value dependent on several variables. The order of the processor identification numbers in the list, length of time the last task took to complete, and the quality of the fine tuning of the kill routine all contributed to the amount of time a worker spent idle.

EXPERIMENTAL BASIS

For this thesis, experimental cycle time, theoretical cycle time, experimental speedup, and theoretical speedup are reported. Two different circuits were used and experiments were run using 500, 100, and 50 jobs for each circuit. Time limitations on the system precluded an exhaustive set of experiments for one through fifteen processors for each input deck. The number of processors varied across the input decks but data was obtained for the same number of processors for 50,100, and 500 tasks for a given input deck. The two experimental circuits will be referred to as input2 and input6.

Input2

This circuit analyzed an enhancement mode, MOS bootstrapped inverter made up of four NMOS transistors. An MOS capacitor was used to facilitate the bootstrapping. Monte Carlo sampling was done with respect to the model used for the MOS transistors. The specific parameter sampled was the threshold voltage, V_{TO} , using a Gaussian distribution and a relative tolerance of ten percent. A transient analysis was performed on this circuit for 16ns. Deck size, prior to combining the SPICE results with the original deck, was 1346 bytes. After attaching the SPICE results to the original deck, the size of each task was a file of approximately 10 Kbytes.

Input6

This circuit compared two MOS multiplexers: one using transmission gates and one using gated inverters. The circuit was made up of five PMOS transistors and five NMOS transistors. Sampling was done with respect to the models used for the transistors. Specific parameters sampled were the threshold voltages, V_{TO} , for both PMOS and NMOS models. A Gaussian distribution and a relative tolerance of ten percent was assumed for each. A transient analysis was performed on this circuit for 30ns. Deck size prior to combining the SPICE results with the original deck was 2131 bytes. After attaching the SPICE results to the original deck, the size of each task was approximately 36 Kbytes.

The two circuits were chosen because of their difference in SPICE batch processing time. Circuit input2 takes 30 seconds to process as a batch SPICE job. Circuit input6 runs for a factor of four times as long, taking 120 seconds to complete. To emphasize the expected performance improvement, input6 would take roughly 16 hours of computation for 500 tasks when run on a single processor and input2 would take approximately 4 hours. We will see that both these circuits benefit greatly from the parallelization process.

EXPERIMENTAL RESULTS

The nature of the decomposition of an application into computationally independent stages leads to the conclusion that the bottleneck in our paradigm exists at the locking or synchronization mechanism, required for exclusive access to the jobpool. In this section we present graphs and tables that will explain the behavior of the system and focus attention on the bottleneck of the jobpool. Graphs will be used to present the global picture of the experiments and how the system performed overall. Tables summarize the instrumentation of the experiments for a more narrow focus on what is happening at the worker and service center level.

Data collected for both experiment input2 and input6, for various task counts, is plotted in Figure 11 and Figure 12 in the form of cycle time per number of workers. Figure 11 presents

data accumulated for the fastest running SPICE deck which we refer to as input2. Figure 12 presents data accumulated for the longest running SPICE deck, input6. Both graphs exhibit the same shape curves for the decrease in cycle time for all experiments but performance improvement is greater and more consistent for the larger granularity task count of 500. Better performance improvement overall is exhibited in the longer running experiment of input6. The least improvement, along with an obvious decrease in performance, is exhibited in the shortest executing experiment of input2, and the smallest task count. The behavior of the cycle time for small task counts of 50, for both experiments, presents questions about performance degradation. We will attempt to discover what variables are involved that impact smaller grain tasks, both size and count, in such a manner that they respond less favorably to parallelization than larger grain tasks.

A more graphic representation of what is actually occurring is displayed in the plots of speedup versus number of workers, shown in Figure 13 and Figure 14. The reference lines on both graphs have slopes of 1.0 and 0.5 corresponding to 100% linear speedup and 50% linear speedup respectively. In Figure 13, we observe that speedup is linear across all experiments. Experiment input6-500 tasks, with a uniprocessor cycle time of 120 seconds, shows the greatest consistent performance improvement. This is followed closely by the input2 experiment of 500 tasks and that of input6 with 100 tasks. In addition, this graph reflects a consistency across each experiment that emphasizes the relationship of task count to speedup; the smaller the number of tasks the less speedup is gained. In fact, there are indications that performance may be flattening out for the experiments of smaller granularity in Figure 13. Regardless of the input, it is evident that the 500 task experiment out-performs the 100 which in turn out performs the 50 task experiment.

Figure 14 is the same data as seen in Figure 13 but with additional data included for some experiments. Although linearity has been established, it is important to recognize that not all the experiments continue in this fashion. The 500 task experiments contain no additional data

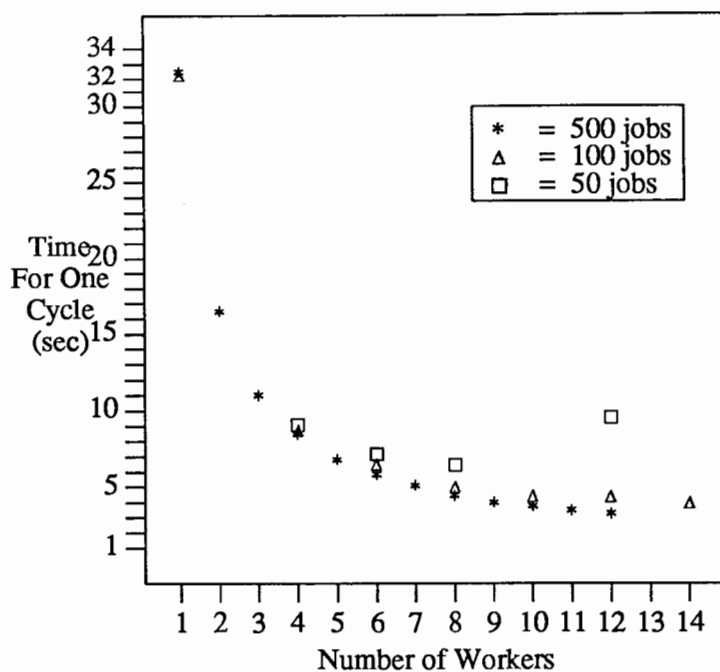


Figure 11. Cycle time for experimental data input2.

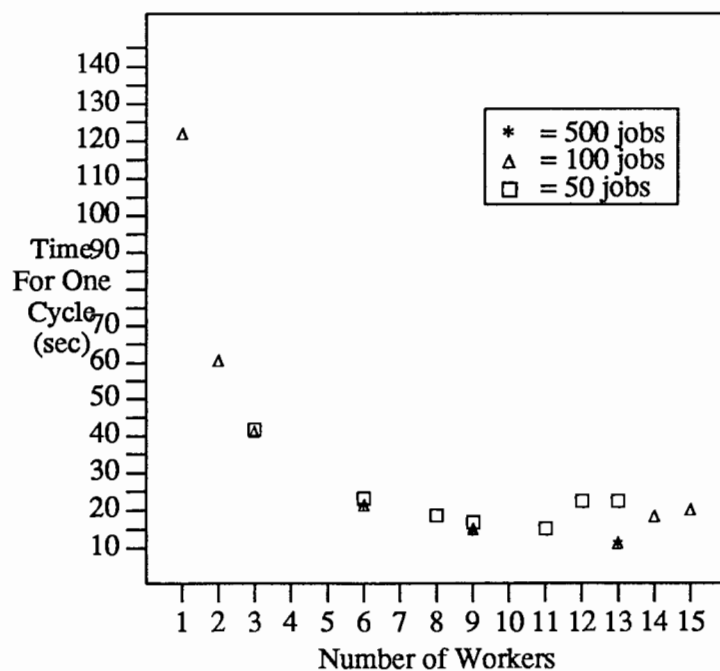


Figure 12. Cycle time for experimental data input6.

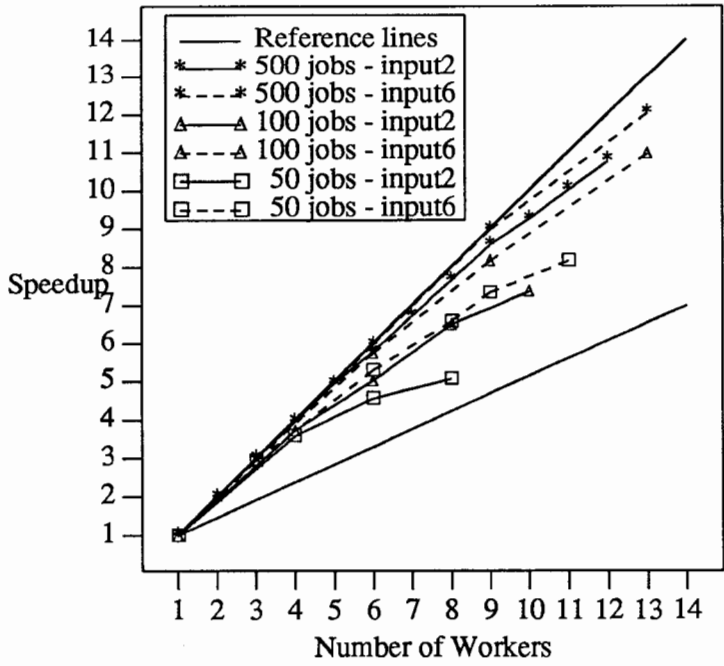


Figure 13. Experimental speedup within linear range.

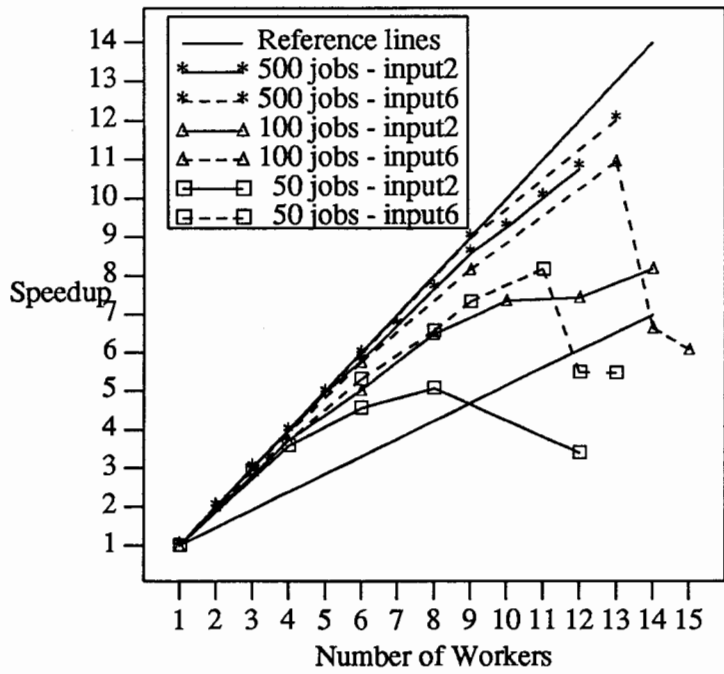


Figure 14. Experimental speedup for all data.

and we see no inclination for either of them to flatten out. In the experiment for input2-100 jobs, speedup exhibits a leveling off and for input2-50 jobs the rolloff is more pronounced. For input6, both sets of data for 100 tasks and 50 tasks show a dramatic breakdown in performance. It should be noted that none of the points on these graphs are superlinear.

After viewing these four graphs, it is apparent that there is a direct impact on performance related to both task size, as represented by uniprocessor cycle time, and number of tasks. The longer run time circuit consistently shows better performance than the shorter run time circuit and larger numbers of tasks perform better than the smaller task counts. This leads to the conclusion that the smaller the uniprocessor task cycle time, plus the smaller the task granularity, the less speedup will be gained. Not unexpectedly, given this view, input2 with 50 tasks shows the least performance improvement overall.

In addition, as explained earlier, there are startup costs incurred with the addition of each new worker. With the shortness of the task cycle time and a granularity of 50 jobs there is less total time over which to amortize the cost of additional worker startup. The longer run time of input6, with a task cycle time four times greater than for experiment input2, has more total time over which to amortize the startup cost for additional workers for any task count. Workers executing tasks which complete in a short amount of time encounter more collisions in their attempt for access to the jobpool. As we increase the number of workers this problem will exaggerate. Recall that workers are started via the *rsh* system command and that there is a measurable and uniform interval of time between the initialization of workers. It is our theory that the greater the number of tasks for each worker to complete, the more accumulated time there is for the desynchronization of the workers from the uniform pattern in which they were initialized. This results in fewer collisions over time and greater availability of the jobpool. The time between requests for the lock among workers approaches a more random distribution over a longer run time. This suggests that desynchronizing the workers at startup time, particularly when there are fewer and shorter tasks to complete, may result in improved throughput.

Further investigation of our data will lead to a strong interest in the value of V_{c_lock} which is defined to be the average number of attempts for the lock per successful lock of the jobpool.

Next we will introduce three sets of tables which will help to support our theories and increase our understanding of what is occurring at the processor level. Tables IV through XII present total average work time and total average communication for each processor participating in typical experiments. Recall that the time spent in the SPICE analysis package is modeled strictly as a delay center. Therefore the accumulated average for SPICE shows up in its entirety under the category of work. Our first approach will be to hold the number of workers constant and vary the number of tasks for a given input. For this we have chosen to look at two different slices of the job spectrum to focus on the deviation of speedup early in the process with six processors and then later with twelve processors. The inverse of this is to hold the number of tasks constant and vary the number of workers. The tabulated data reflects all experimental points. Workers are listed in the tables in the order in which they were initialized during the experiments.

Table IV, V, and VI present total average work time and total average communication time for six processors assigned to the jobpool containing tasks for experimental circuit input6. The tables contain information for 500, 100 and 50 tasks respectively. Using this approach we can compare the behavior of the experiments through investigation of work and communication times as task count decreases.

Tables VII, VIII, and IX look at the same type of information only for experimental circuit input2 using twelve workers. Tables X, XI, and XII again present information in the form of total average work and total average communication but now the number of tasks is fixed and the number of workers is varied. Using this approach we can compare the behavior of the experiments through the investigation of accumulated work and communication times as workers increase. Finally, analysis of the model will complete the discussion of what role communication and work each play in the behavior of the system.

TABLE IV

AVERAGE TOTAL WORK AND COMMUNICATION TIME
500 TASKS, 6 WORKERS, INPUT6

worker	work in sec.	%diff	comm in sec.	%diff	tasks
Lady	119	0.8	1.8	-0.0	84
Goofy	121	-0.8	1.6	5.6	83
Huey	120	0.0	1.6	5.6	83
Louie	119	0.8	1.9	-5.6	84
Scrooge	120	0.0	2.0	-11.1	83
Dumbo	119	0.8	1.6	5.6	83
work: mean = 120 variance = 0.6 stddev = 0.8					
comm: mean = 1.8 variance = 0.02 stddev = 0.1					

TABLE V

AVERAGE TOTAL WORK AND COMMUNICATION TIME
100 TASKS , 6 WORKERS, INPUT6

worker	work in sec.	%diff	comm in sec.	%diff	tasks
Lady	120	0.8	2.4	0.0	17
Goofy	120	0.8	2.6	-8.3	17
Huey	119	1.6	2.2	8.3	17
Louie	127	-5.0	3.3	-37.5	16
Scrooge	119	1.6	2.9	-20.8	17
Dumbo	119	1.6	1.1	54.1	16
work: mean = 121 variance = 9.9 stddev = 3.1					
comm: mean = 2.4 variance = 0.6 stddev = 0.8					

TABLE VI

AVERAGE WORK AND COMMUNICATION TIME
50 TASKS, 6 WORKERS, INPUT6

worker	work in sec.	%diff	comm in sec.	%diff	tasks
Lady	121	0.0	4.2	-40.0	9
Goofy	120	0.8	3.8	-26.7	9
Huey	119	1.6	1.3	56.7	8
Louie	124	-2.5	3.9	-30.0	8
Scrooge	119	1.6	2.3	23.3	8
Dumbo	120	0.8	2.4	20.0	8
work: mean = 121 variance = 3.5 stddev = 1.9					
comm: mean = 3.0 variance = 1.3 stddev = 1.2					

As the number of tasks decreases from 500 to 100, as seen in Tables IV and V, the mean communication time increases from 1.8 seconds to 2.4 seconds. At the 50 task level, we see even more increase in communication mean over the 500 and 100 task experiments. Across the three tables we see no change in the average total work mean.

Tables VII, VIII, and IX present a more dramatic example of the increase in communication as the number of tasks decreases. The model will substantiate that communication is the dominant factor in the degradation of speedup. Less dramatic, but of importance also, is the increase in average total work time as the task count decreases. Specifically, for 50 tasks there is a 19% increase in the average total work mean over the 500 and 100 task experiments. Looking at average total communication time for 50 tasks we see an increase of 263% over the 100 task experiment. Average total communication is approximately 7 times more when 50 tasks are processed than when 500 task are processed. This increase reflects the dramatic fall off in speedup from 11, to 7, to 3, for 500, 100, and 50 tasks respectively, as seen in Figure 14.

TABLE VII
AVERAGE TOTAL WORK AND COMMUNICATION TIME
500 TASKS, 12 WORKERS, INPUT2

worker	work in sec.	%diff	comm in sec.	%diff	tasks
Dumbo	33	-3.1	2.0	9.1	42
Goofy	32	0.0	2.0	9.1	43
Tramp	32	0.0	2.3	-4.5	43
Huey	32	0.0	2.2	0.0	43
Louie	31	3.1	2.3	-4.5	43
Peterpan	34	-6.3	2.6	-18.2	40
Tinkerbell	31	3.1	2.0	9.1	42
Mickey	32	0.0	2.5	-13.6	42
Dewey	32	0.0	1.6	27.3	41
Sleezy	32	0.0	2.5	-13.6	41
Scrooge	32	0.0	2.3	-4.5	40
Lady	32	0.0	2.2	0.0	40
work: mean = 32 variance = 0.6 stddev = 0.8					
comm: mean = 2.2 variance = 0.08 stddev = 0.3					

TABLE VIII

AVERAGE TOTAL WORK AND COMMUNICATION TIME
100 TASKS, 12 WORKERS, INPUT2

worker	work in sec.	%diff	comm in sec.	%diff	tasks
Huey	32	0.0	4.4	-2.3	10
Louie	32	0.0	4.2	2.3	10
Scrooge	32	0.0	3.8	11.6	9
Dumbo	32	0.0	4.5	-4.7	9
Geppetto	32	0.0	5.8	-34.9	9
Goofy	32	0.0	1.8	58.1	8
Mickey	32	0.0	4.2	2.3	8
Dewey	32	0.0	5.7	-32.6	8
Lady	32	0.0	5.2	-21.0	8
Peterpan	32	0.0	3.1	28.0	7
Sleezy	33	-3.1	4.9	-14.0	7
Tinkerbell	32	0.0	4.3	0.0	7
work: mean = 32 variance = 0.1 stddev = 0.3 comm: mean = 4.3 variance = 1.2 stddev = 1.1					

TABLE IX

AVERAGE TOTAL WORK AND COMMUNICATION TIME
50 TASKS, 12 WORKERS, INPUT2

worker	work in sec.	%diff	comm in sec.	%diff	tasks
Lady	36	5.3	14.1	9.6	5
Scrooge	39	-2.6	13.5	13.5	7
Peterpan	37	2.6	14.6	6.4	5
Sleezy	35	7.9	14.3	8.3	4
Huey	36	5.3	17.4	-11.5	5
Louie	37	2.6	13.0	16.7	4
Goofy	38	0.0	17.3	-10.9	4
Dewey	38	0.0	16.1	-3.2	4
Dumbo	40	-5.3	15.7	-0.6	4
Geppetto	39	-2.6	19.3	-23.7	3
Mickey	40	-5.3	14.9	4.5	2
Tramp	38	0.0	17.5	-12.2	3
work: mean = 38 variance = 2.6 stddev = 1.6 comm: mean = 15.6 variance = 3.7 stddev = 1.9					

Up to this point, there has been no discussion as to how the increase in the number of workers affects behavior of these experiments. Tables X, XI, XII, are introduced here for a closer look at total average work time and total average communication time for individual processors as number of workers increases. This data is again focusing on experiment input6 and has the fixed task granularity of 100 jobs. The reader may refer back to Table V, in addition to the three tables presented here, to include the data presented for six workers for 100 jobs in the discussion. The important issue presented in this collection of four tables is the fact that as the number of workers grows, the mean work values increase and the mean communication values increase dramatically.

Referring to Tables V, X, XI, and XII we see a dramatic increase in communication times as the number of workers increases. The mean communication time for nine workers increases by approximately 46% over the six processor experiment. Increasing the number of processors to thirteen increases communication by 54% over nine workers and 125% over the data compiled for six workers. When the number of workers is increased to fourteen the average total communication mean grows to a factor of 4.5 times greater than that for thirteen workers. In addition, the average total work mean for fourteen workers is significantly higher but nowhere near as large as the communication increases. It is this increase in average communication as workers are added, plus the less obvious increase in average work time, which causes speedup to increase from 5, to 7, to 10.5, and then decrease to 6, for 6, 9, 13, and 14 workers respectively.

TABLE X

AVERAGE TOTAL WORK AND COMMUNICATION TIME
100 TASKS, 9 WORKERS, INPUT6

worker	work in sec.	%diff	comm in sec.	%diff	tasks
Lady	120	0.0	1.5	57.1	11
Goofy	119	0.8	4.1	-17.1	12
Huey	119	0.8	2.4	31.5	11
Louie	120	0.0	3.7	-5.7	11
Scrooge	127	-5.5	4.3	-22.8	11
Dumbo	119	0.8	3.6	-2.8	11
Peterpan	120	0.0	4.1	-17.1	11
Sleezy	119	0.8	4.6	-31.5	11
Tinkerbell	118	1.6	3.6	-2.8	11
work: mean = 120 variance = 7.1 stddev = 2.7 comm: mean = 3.5 variance = 1.0 stddev = 1.0					

TABLE XI

AVERAGE TOTAL WORK AND COMMUNICATION TIME
100 TASKS, 13 WORKERS, INPUT6

worker	work in sec.	%diff	comme. in sec.	%diff	tasks
Pinocchio	121	0.0	5.8	-7.4	8
Lady	119	1.6	5.4	0.0	8
Goofy	121	0.0	5.9	-9.3	8
Huey	119	1.6	6.9	-27.8	8
Louie	129	-6.6	3.4	37.0	7
Scrooge	120	0.8	6.3	-16.7	8
Dumbo	123	-1.6	6.3	-16.7	8
Peterpan	123	-1.6	6.4	-18.5	8
Sleezy	121	0.0	6.0	-11.1	8
Tinkerbell	119	1.6	6.2	-14.8	8
Tramp	121	0.0	3.4	37.0	7
Dewey	120	0.8	4.8	11.1	7
Mickey	119	1.6	3.2	40.7	7
work: mean = 121 variance = 7.5 stddev = 2.7 comm: mean = 5.4 variance = 1.6 stddev = 1.3					

TABLE XII
AVERAGE TOTAL WORK AND COMMUNICATION TIME
100 TASKS, 14 WORKERS, INPUT6

worker	work in sec.	%diff	comm in sec.	%diff	tasks
Lady	125	0.0	23.5	5.24	10
Scrooge	125	0.0	20.9	15.7	8
Peterpan	126	-0.8	14.9	39.9	6
Huey	124	0.8	23.2	6.5	7
Louie	128	-2.4	22.6	8.9	6
Goofy	126	-0.8	27.1	-9.3	6
Dewey	126	-0.8	29.9	-20.6	6
Dumbo	125	0.0	20.8	16.1	7
Geppetto	127	-1.6	39.0	-57.3	6
Mickey	126	-0.8	28.8	-16.1	7
Tramp	125	0.0	24.7	0.4	6
Tinkerbell	129	-3.2	26.1	-5.2	10
Doc	121	3.2	25.0	-0.8	7
Dopey	118	5.6	20.4	17.7	8
work: mean = 125 variance = 7.6 stddev = 2.8					
comm: mean = 24.8 variance = 31.4 stddev = 5.6					

Now that increases in average total communication and average total work have been identified, we focus our attention on determining exactly where these increases are occurring. This requires investigation at the service center level. In the previous set of tables, increases in average total work and a dramatic increase in average communication were directly associated with the increase in the number of workers. We first turn our attention to the increase in total average work. Recall that all of the time spent in the SPICE analysis package is accumulated in a single delay service center. Therefore, the majority of the time accumulated for total average work is due to the SPICE analysis package. Average SPICE time is measured at the SPICE delay center as seen in Figure 8 of the Complex model. Table XIII demonstrates how the average SPICE time behaves as the number of workers increases.

In the experiment run with 50 tasks, the average SPICE time increases slowly up through eight processors and then shows a large increase with twelve processors. This increase in SPICE time coincides with the breakdown in performance seen in Figure 14. In addition, this phenomena is observed for the experiment with input6-100 tasks. Here the average SPICE time

jumps from 121 seconds to 125 seconds with an increase from thirteen to fourteen processors. This is directly reflected in the speedup graph of Figure 14 where performance takes a nosedive. Further decay in performance is reflected with fifteen workers, where time increases to 128 seconds. This phenomena was observed in each of the experiments that display dramatic rolloff in performance. Notice also in Table XIII, that an experiment that does not show a dramatic breakdown in performance, such as input2-100 tasks, does not reflect significant change in the average SPICE time. This correlates to the gradual leveling off, seen in this speedup graph of Figure 14.

TABLE XIII
AVERAGE SPICE TIMES (IN SECONDS)

input6-100 tasks		input2-50 tasks		input2-100 tasks	
workers	SPICE	workers	SPICE	workers	SPICE
1	119	1	31	1	31
4	-	4	32	4	31
6	120	6	33	6	33
8	-	8	33	8	32
9	120	9	-	9	-
10	-	10	-	10	32
12	-	12	38	12	32
13	121	13	-	13	-
14	125	14	-	14	34
15	128	15	-	15	-

We believe that the increase of average SPICE times is another consequence of the breakdown in communication and we offer two possible reasons for this behavior. Computation for the SPICE package takes place at the individual workstation and input and output is handled by the directory file server. One possible explanation is that the increase is a function of the communication occurring within the SPICE analysis package which we cannot instrument. Recall that a second file server was used to handle swap space for the diskless workstations. It is this file server which provides the SPICE binary for each worker. It is possible that the SPICE binary is taking longer to load and that this behavior is a microcosm of the behavior we are seeing at the directory file server. The main point to be made here is that these increases in work

are still small compared to the increases in communication that are occurring.

To determine where communication increases are occurring, we turn our attention to the the individual queueing service centers, as represented in Figure 8 for the Complex model. Here, large increases in communication were observed as the number of workers grew. When fourteen workers were used to process input6-100 jobs, increases of approximately 95% over uniprocessor averages were observed at all but one queueing service center. The locking queueing service center showed an increase in average time of 66% over the uniprocessor average. For experiment input2-100 tasks, when four workers were used, we observed increases of 17% over uniprocessor averages up to 87% which was observed at the locking mechanism. An increase to fourteen workers increased all queueing center values to above 80% with the lock service center average climbing to 97% over uniprocessor averages.

These increases in communication indicate that either the jobpool file server is starting to bog down or the Ethernet is swamping. It is our belief that the problem resides in the file server. With a task granularity of 500, it was impossible to collect information for more than thirteen workers for experiment input6. The same can be said for experiment input2 and twelve workers. System behavior that was observed when these runs were attempted was dramatic. The SPARC workstation that was used as the jobpool file server accumulated large amounts of time at the lock daemon and response on the workstation came to a grinding halt. It appeared that so much time was spent responding to requests for the lock that actual work ceased.

On SUN O/S it costs approximately 3.4ms of compute time to send a 256 byte unreliable datagram packet (UDP). The Ethernet is used only 0.1ms or less of the total time. For 97% of the send time the Ethernet is idle [11]. The Ethernet is a common resource for both our jobpool file server and the file server handling swap space for the diskless workstations. If the Ethernet were swamped we would see failure at both servers. In our observations we only saw failure at the directory file server. Given that the System V call to *lockf()* uses UDP as its message passing protocol and that UDP is considered unreliable, it is possible that messages requesting the lock

or responses from *lockf()* are getting lost [33].

At this point it should be noted that the tabulated total average communication values include some inconsistencies. As an example, we see in Table V that Louie has a smaller total average communication value than all other workers in this experiment. This inconsistency can also be seen in Table VI for Huey. As mentioned previously, over the course of the experiments it was observed that isolated queueing service centers showed unexplained deviations from expected behavior. Earlier, we explained these deviations as systematic failure in the measuring process. Here, then, is an example of this anomaly. It is important to note, that although these deviations exist, we will show that the single server, multiple delay center model is resilient in spite of these system failures and appears to be insensitive to them. We mention these idiosyncrasies as a matter of completeness.

We have seen in the speedup graph of Figure 14 that some experiments show an obvious breakdown in performance, some indicate a leveling off, and two look like they will continue linearly as workers are added. It is now time to employ the Simple model in our evaluation of the behavior of these experiments. First we will examine the Simple model and its behavior using only the data from the speedup graph of Figure 13. This data falls within what we will refer to as the *linear range*. No data points that even hint of a leveling off or a breakdown are included in this graph. Theoretical analysis will be done to project behavior for a large number of workers for which we were unable to collect data. The theoretical analysis will be compared to the experimental data and we will see how well the model validates the experimental behavior. Next we will examine the Simple model using all experimental data points as seen in Figure 14. Theoretical behavior will again be presented for a large number of workers. This two step approach validates the model within the linear range and then validates it outside this range where breakdowns in speedup occur.

Examining the model reveals that there are two parameters affiliated with each service center. The first component is the service time for service center and the second is the number

of visits to that center. We have already examined the instrumentation tabulated for service center values for average accumulated time. Now we will look at the parameter of visitations. Earlier, the parameter V_k was defined in Table II as the visit count of resource k , or the average number of visits that a worker makes to a service center. Referring to previously defined Eqs.(20),(13),(9), and (10) for the Simple model we see how V_k impacts communication and work values.

$$cycle\ time = q * \left[1 + \frac{\alpha^N}{N! \sum_{j=0}^{N-1} \frac{\alpha^j}{j!}} \right] \quad (20)$$

where

$$\alpha = \frac{d}{q} \quad (13)$$

and the sum of the service times for delay and queueing centers is

$$d = \sum_{k \in delay} D_k(N) = \sum_{k \in delay} S_k V_k(N) \quad (9)$$

$$q = \sum_{k \in queueing} D_k(N) = \sum_{k \in queueing} S_k V_k(N) \quad (10)$$

In our instrumentation, we tabulated the number of total attempts for the lock, and the number of successful attempts for the lock for each worker. From these two numbers we will calculate the average number of visits to the queue and delay service centers in Figure 8 labeled "obtain lock". We will refer to the average visitation for the queueing, or communication, service center as V_{c_lock} and to the delay, or work, service center as V_{w_lock} where

$$V_{c_lock} = V_{w_lock} = \frac{\text{number of total attempts for lock}}{\text{number of successful attempts for lock}}$$

Since the visitation value is the same for both the delay and queueing center we may alternatively refer to these two values as c_lock . From this we can graphically depict what is occurring at the synchronization center as the number of workers increase.

The average service center visitations for c_lock have been plotted on Figure 15 as discrete points for increasing number of workers. A best fit line over the range of one to fourteen workers, has been calculated and is shown here as a solid line. We see that V_{c_lock} is a linear function for this particular experiment and is represented by the line

$$V_{c_lock} = 0.76 + (0.211 * N).$$

where N represents the number of workers in the system.

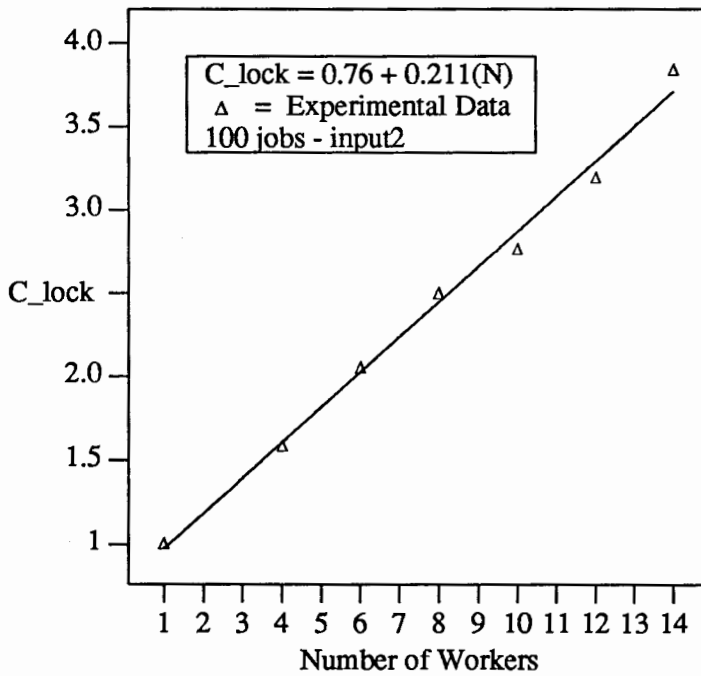


Figure 15. Experimental data versus best fit line.

Recall that experiment input2 with 100 tasks indicated a gradual leveling off. Including the discrete points of V_{c_lock} for twelve and fourteen processors changed the slope of the line less than 5% in this particular case so the points were included for completeness. In this experiment we observe that the values for V_{c_lock} are linear beyond the points where speedup levels off. Following the determination of V_{c_lock} , this equation was then used in the Simple model to represent average number of visits to both the queueing lock service center and the delay lock service center.

Table XIV is an example of data from experiment input2-100 tasks, that was used to parameterize the Simple model. Input to the computer calculations for both the Simple and Complex models take the form shown in this table. As discussed previously, the average service times for individual service centers, S_k , are calculated by taking accumulated totals for each center and dividing by the total number of accumulated visits to a specific service center. This table represents average service times calculated from accumulated totals for data collected within the linear range as represented in Figure 13 for the experiment input2-100 workers. Here we see average service time, in seconds, for each service center, as well as the variable representing the average number of visitations to each center. In this table V_{c_lock} is the linear extrapolation of the behavior at the synchronization service center.

TABLE XIV
SERVICE CENTER AVERAGES APPLIED TO SIMPLE MODEL
FROM EXPERIMENT INPUT2 - 100 WORKERS

Service Center	Work Service Times in sec.	Visits	Comm. Service Times in sec.	Visits
obtain lock	0.000000	V_{c_lock}	1.029548	V_{c_lock}
read jobpool	0.000000	Q	0.157300	Q
move input	0.000343	Q	0.118600	Q
miss	0.000000	1-Q	-	-
remove lock	0.001359	Q	0.225601	Q
prepare input	0.000971	Q	0.000371	Q
spice	32.179886	Q	-	-
combine in/out	0.000000	Q	0.781514	Q
move output	0.004057	Q	0.334371	Q
no work	0.000000	1-Q	-	-
Sum	32.186616		2.647305	
Sum Total = 34.833921 Total Time = 35.76 %Error = 2.67%				

For our analysis of the experiment we have chosen to denote the model value $V_k(N)$ with variables of either V_{c_lock} , Q or 1-Q. We have chosen to set the value for Q to the optimistic value of 1.0. This says that service centers, which are visited if work is found in the jobpool, have an average number of visits during a single cycle of 1.0, or

$$V_k(N) = 1.0 .$$

Centers which handle the branches in Figure 8 relating to an empty jobpool, where no work is found, have an average visitation value per cycle of zero, or

$$V_k(N) = 0.$$

By these definitions, we are guaranteeing that work was always found in the jobpool during the experiment. We will see that although not completely accurate, this value is a good approximation for the analysis of system behavior.

For each experiment plotted on Figure 13, model values were calculated. Using the Simple model and the assumption that c_lock is linear we have plotted theoretical cycle time for large numbers of workers. Here we assume that as the number of workers increases, all service times are constant and only visitations relating to c_lock change. Figure 16 shows the results of the Simple model using input data from experiment input6 for 50, 100, and 500 tasks. This graph suggests a slight increase in cycle time as the model extrapolates beyond the experimental data values. Unfortunately, this is not a particularly useful graph since it is hard to differentiate what is occurring as number of workers increases. To obtain a more clear picture of the behavior of the model, we have plotted this same theoretical extrapolation in Figure 18 in the form of speedup. For comparison, the experimental speedup from Figure 13 is included.

Figure 17 represents the theoretical cycle time for experiment input2 for 50, 100, and 500 tasks. The degradation of speedup in Figure 13 is depicted here in the gradual increase of cycle time as extrapolated over thirty-five workers. In Figure 19 we see this theoretical information plotted in the form of speedup with the experimental speedup included on the graph for clarification. In both theoretical speedup graphs of Figure 18 and Figure 19, we see that the model reflects the experiments optimistically.

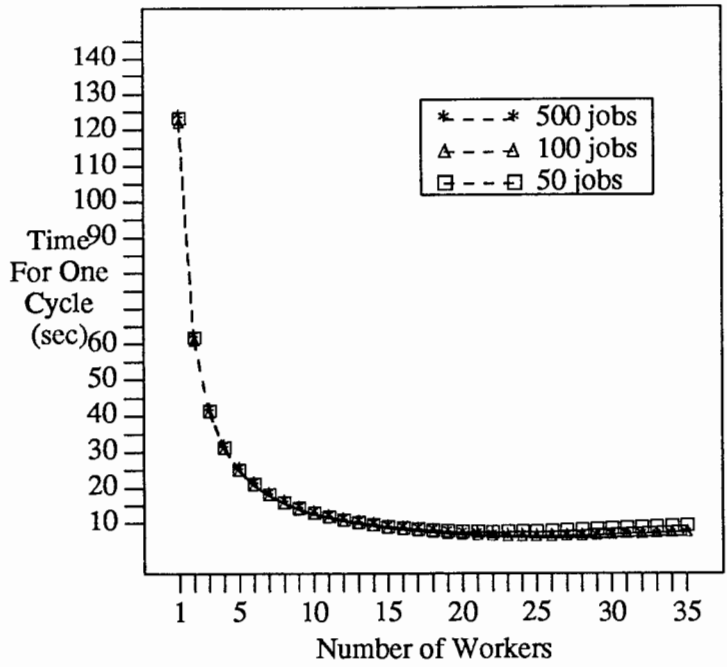


Figure 16. Family of curves within linear range for input6.

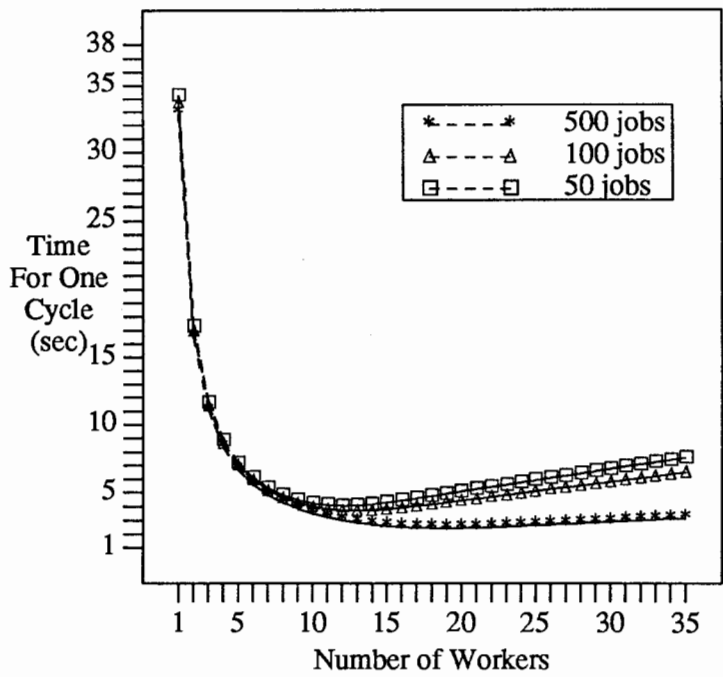


Figure 17. Family of curves within linear range for input2.

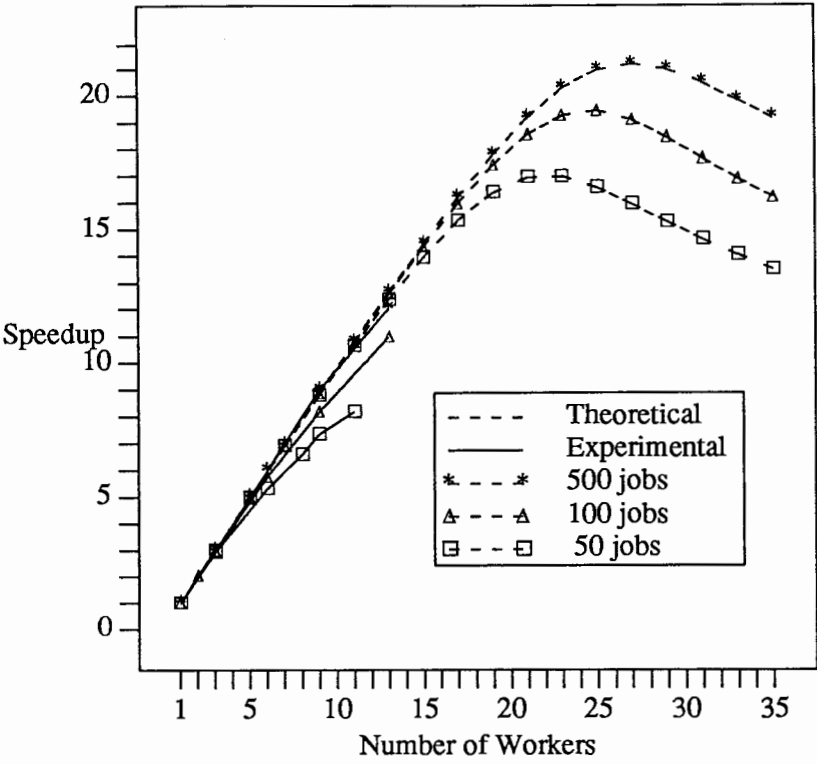


Figure 18. Theoretical extrapolation versus experimental linear speedup for input6.

TABLE XV

EQUATIONS FOR V_{c_lock} CORRESPONDING TO FIGURE 18

jobs	V_{c_lock}	N workers
500	$V_{c_lock} = 1.03 + 0.123(N)$	1-35
100	$V_{c_lock} = 0.55 + 0.236(N)$	1-35
50	$V_{c_lock} = 0.59 + 0.249(N)$	1-35

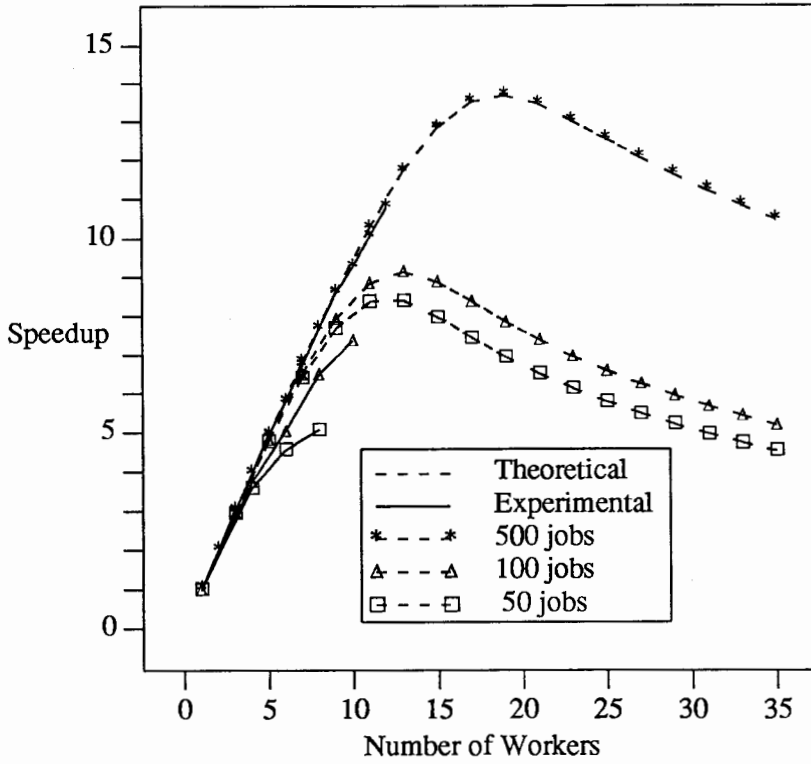


Figure 19. Theoretical extrapolation versus experimental linear speedup for input2.

TABLE XVI

EQUATIONS FOR V_{c_lock} CORRESPONDING TO FIGURE 19

jobs	V_{c_lock}	N workers
500	$V_{c_lock} = 0.87 + 0.106(N)$	1-35
100	$V_{c_lock} = 0.76 + 0.213(N)$	1-35
50	$V_{c_lock} = 0.60 + 0.249(N)$	1-35

It is comforting to note that as expected, each theoretical speedup curve for both sets of experiments, input2 and input6, reach a peak performance and then gradually decay as the number of workers continues to increase. Intuitively, this is what we would expect as the value of c_lock continues to increase with the number of workers. Common sense tells us that as workers increase, there will be more contention for the lock reflected in more time spent trying to obtain the lock. This, in turn, means less time is spent doing actual work. We see better speedup performance for greater number of workers as we increase the number of tasks from 50 to 500 and a smaller slope for V_{c_lock} as the tasks increase.

We can explain the decay process of the speedup graphs for any given theoretical extrapolation using the model. Recall that the number of visitations for the lock queueing service center is defined by V_{c_lock} where all other queueing servers have visitations set to 1.0. This reduces the summation of all queueing servers, except one, to a constant value. The time associated with the lock queueing service does not reduce to a constant because of V_{c_lock} . Recall from Eq.(20) that cycle time for the Simple model is

$$cycle\ time = q * \left[1 + \frac{\alpha^N}{N! \sum_{j=0}^{N-1} \frac{\alpha^j}{j!}} \right] \quad (20)$$

For large numbers of workers (N)

$$\frac{\alpha^N}{N! \sum_{j=0}^{N-1} \frac{\alpha^j}{j!}}$$

goes to zero which reduces cycle time to

$$cycle\ time = q .$$

From Eq.(10) the summation of all queueing centers reduces to

$$q = constant + (S_{c_lock} * V_{c_lock})$$

where V_{c_lock} has been shown to be linear. So

$$q = cycle\ time = A + BN \approx BN \text{ for } N \gg A$$

where

$$A = \text{constant terms}$$

$$B = S_{c_lock} * (\text{slope of } V_{clock})$$

We see that q changes by B for the addition of each new processor. This change is reflected in the decay of speedup at a steady rate of $1/q$. Speedup reduces to the equation:

$$\text{Speedup} = \frac{\text{uniprocessor cycle time}}{q}.$$

In an effort to gain more understanding of what role c_lock plays in the behavior of the system, Table XVII was compiled. The most interesting aspect of this table is how comparable the slopes are for the same number of tasks regardless of the experimental cycle time for each experiment. This, coupled with the decrease in the slope as the number of tasks increases, lends support to the theory mentioned earlier which suggests that c_lock is larger for fewer tasks and that the more tasks run in an experiment, the more time there is for the workers to desynchronize from their startup pattern. This results in fewer collisions in an attempt to obtain the lock for larger numbers of tasks.

TABLE XVII

SLOPE VALUES FOR V_{c_lock} WITHIN LINEAR RANGE
FOR VARYING NUMBERS OF TASKS

	50	100	500
input2	0.249	0.211	0.106
input6	0.249	0.236	0.123
%diff	0.0%	10.6%	13.8%

Let us now look at c_lock for those experiments which result in a dramatic breakdown of speedup as the number of workers increases. Figure 14 contains three such experiments from which to chose. We turn our attention specifically to experiment input6-100 tasks. Speedup for this experiment continues to be linear out to thirteen processors. With the addition of one more processor, speedup drops dramatically and then continues to drop with the addition of the

fifteenth processor. In Figure 20 we plot all experimental values for c_lock , as number of workers increases, and then calculate a piecewise linear best fit for these points. C_lock is linear up to thirteen processors and then increases sharply reflecting the simultaneous breakdown in speedup seen in the aforementioned plot. Using the accumulated values for all data shown in Figure 14, and the piecewise linear equations for c_lock , we next plot the theoretical speedup for this experiment. Figure 21 indicates that the model does reflect the experimental results, given the piecewise linear behavior of c_lock .

Figure 22 is another example of the breakdown of c_lock . This experiment is input2 with 50 tasks. In Figure 14 we see that this particular experiment breaks down when the number of workers is increased from eight to twelve. We see this same breakdown reflected in c_lock . Here again, we have experimental values of c_lock plotted on the same graph with the calculated piecewise linear best fit for these data points. In Figure 23 this change in c_lock is reflected in the plot of the theoretical model. The model depicts a clear representation of the behavior of the actual experiment.

Figure 24 and Figure 25 now reflect theoretical speedup versus experimental speedup for all data accumulated for all experiments, including break points outside the linear range of speedup. For experiment input6-100 jobs in Figure 24, the recalibration of S_k for all centers changed the predicted speedup in the linear range by less than 10%. Similar results were noted for other experiments.

In these graphs we still continue to see the behavior of improved speedup as the number of tasks increases. Even more obvious now is the behavior related to c_lock . The steeper the slope of c_lock , the steeper the decay of speedup past the break point. In Figure 25, the 100 jobs experiment does not show an obvious breakdown for the experimental data. The average effective service demand times for the queueing service centers provided for the theoretical calculations of the Simple model obviously reflect enough of an increase in contention for the file server that the theoretical model predicts a gradual rolloff would appear likely.

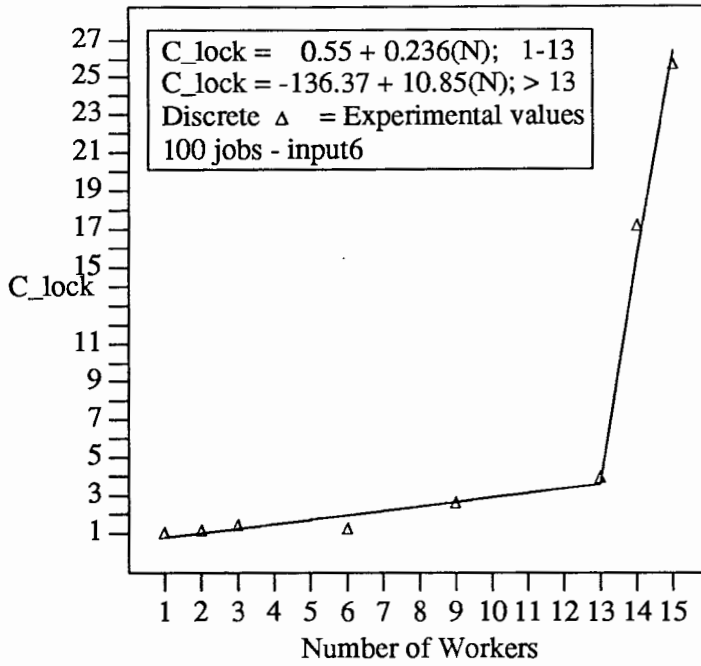


Figure 20. Experimental versus piecewise linear best fit for input6 - 100 jobs.

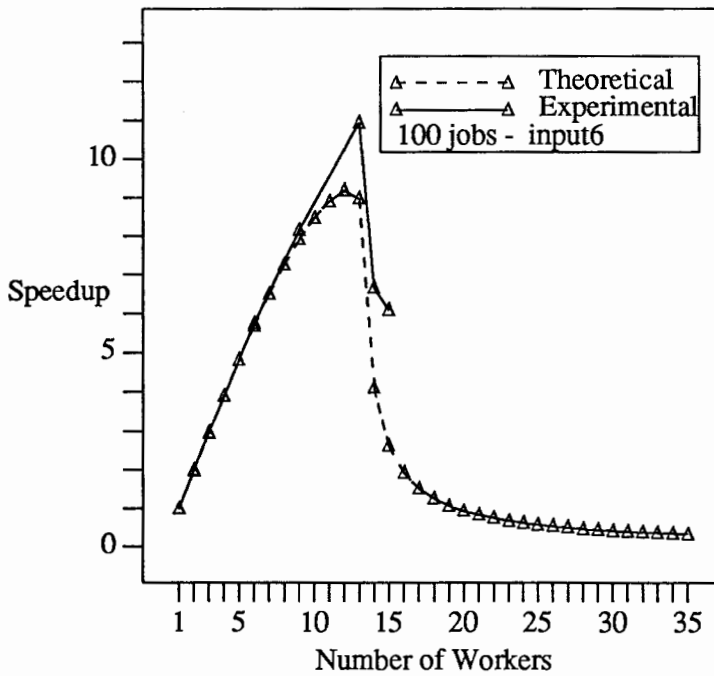


Figure 21. Theoretical breakdown extrapolation versus experimental speedup for input6 - 100 jobs.

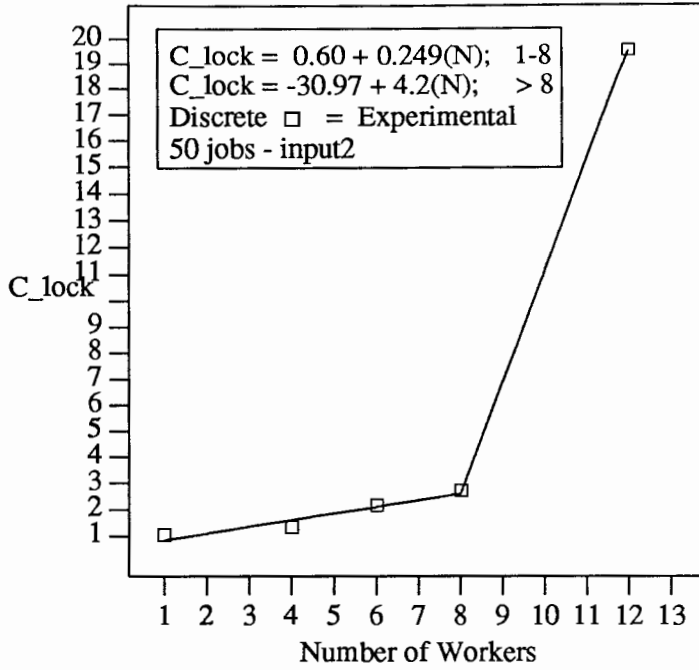


Figure 22. Experimental versus piecewise linear best fit for input2 - 50 jobs.

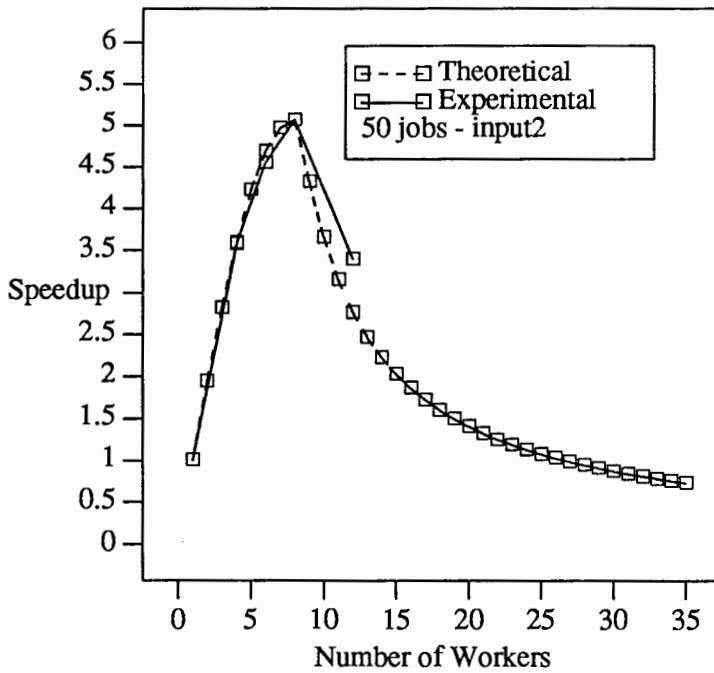


Figure 23. Theoretical breakdown extrapolation versus experimental speedup for input2 - 50 jobs.

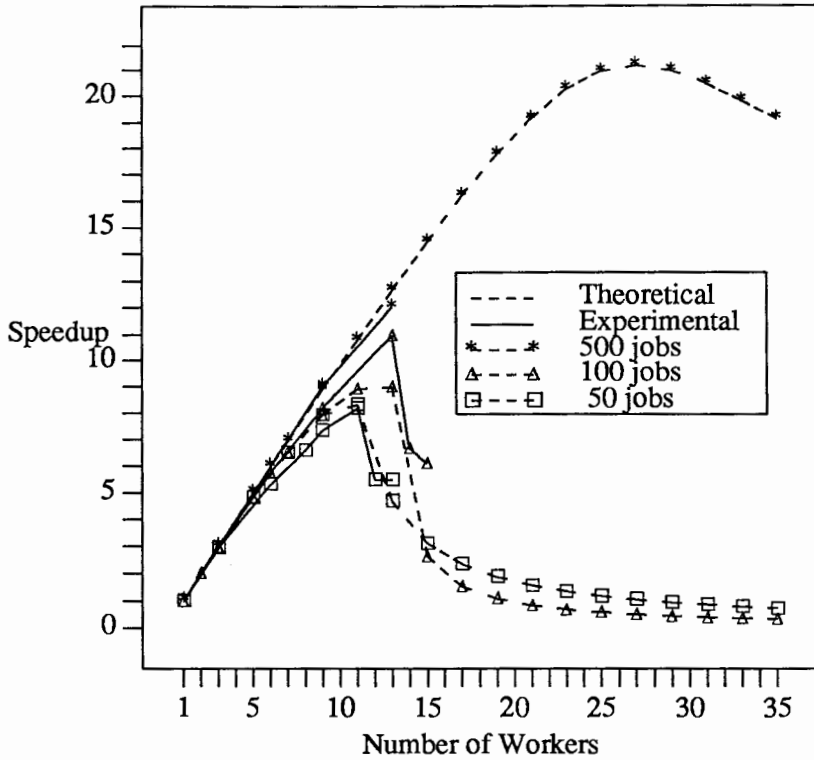


Figure 24. Theoretical breakdown extrapolation versus experimental speedup for input6.

TABLE XVIII

EQUATIONS FOR V_{c_lock} CORRESPONDING TO FIGURE 24

jobs	V_{c_lock}	N workers
500	$V_{c_lock} = 1.03 + 0.123(N)$	1-35
100	$V_{c_lock} = 0.55 + 0.236(N)$	1-13
	$V_{c_lock} = -137.16 + 10.85(N)$	13-35
50	$V_{c_lock} = 0.59 + 0.249(N)$	1-11
	$V_{c_lock} = -56.16 + 5.53(N)$	11-35

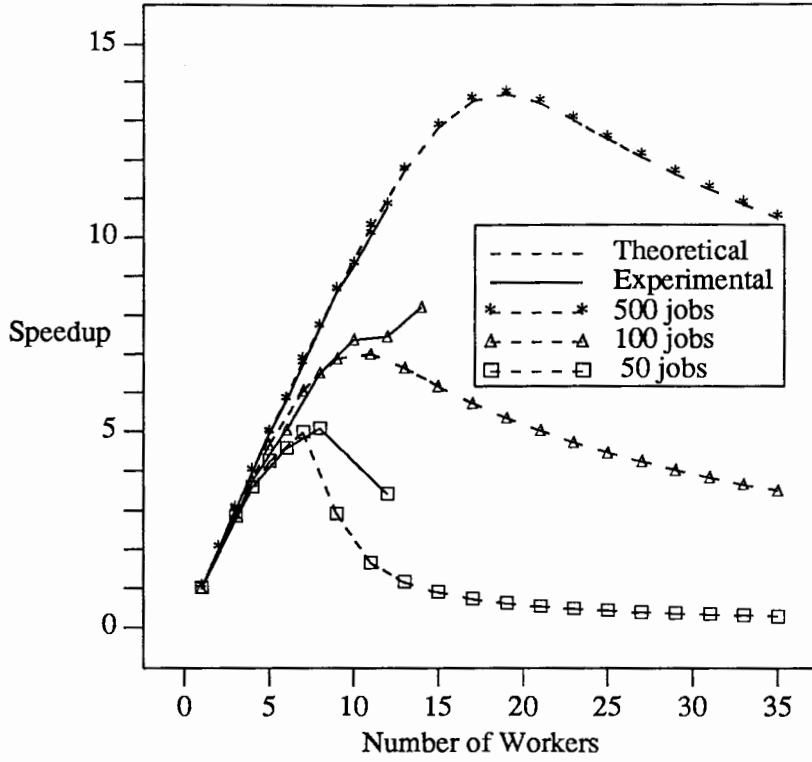


Figure 25. Theoretical breakdown extrapolation versus experimental speedup for input2.

TABLE XIX

EQUATIONS FOR V_{c_lock} CORRESPONDING TO FIGURE 25

jobs	V_{c_lock}	N workers
500	$V_{c_lock} = 0.87 + 0.106(N)$	1-35
100	$V_{c_lock} = 0.76 + 0.213(N)$	1-35
50	$V_{c_lock} = 0.60 + 0.249(N)$	1-8
	$V_{c_lock} = -9.24 + 1.44(N)$	8-35

SUMMARY

In this chapter, queueing network theory was used to develop two models that serve as analysis tools for determining system behavior during the experimental process. The Complex model was used to describe the topology of the physical system and served as a template for the instrumentation process of the experiments. The Simple model was used to replicate, as well as predict, the behavior of the system. The most obvious difference between these two models is the characterization of the system with respect to the synchronization mechanism. The Complex model treats each queueing service center as independent and mutually exclusive and the Simple model provides a single queueing service center for modeling the jobpool file server.

Figures 26 and 27 compare the results of the Complex model with the Simple model. Evaluation of the Complex model is done using the iterative process outlined in Lazowska's MVA solution technique assuming (falsely) that service centers are separable for this model [30]. The Complex model analysis of cycle time is a more optimistic representation than the Simple model. This results in a more optimistic graph for speedup. From Figure 26 it is difficult to tell which model represents the experimental data more closely. But from Figure 27 it seems indisputable that the experiment is more realistically modeled by the Simple model. The two models are similar for smaller numbers of workers but diverge as the number of workers increase. This seems reasonable given the difference in how the jobpool file server queueing factor is handled. The Complex model definitely describes the behavior of the system but does not do a particularly good job of modeling the queueing factor as the number of workers increases. The Simple model appears to represent the actual behavior of the queueing factor as defined by the jobpool file server.

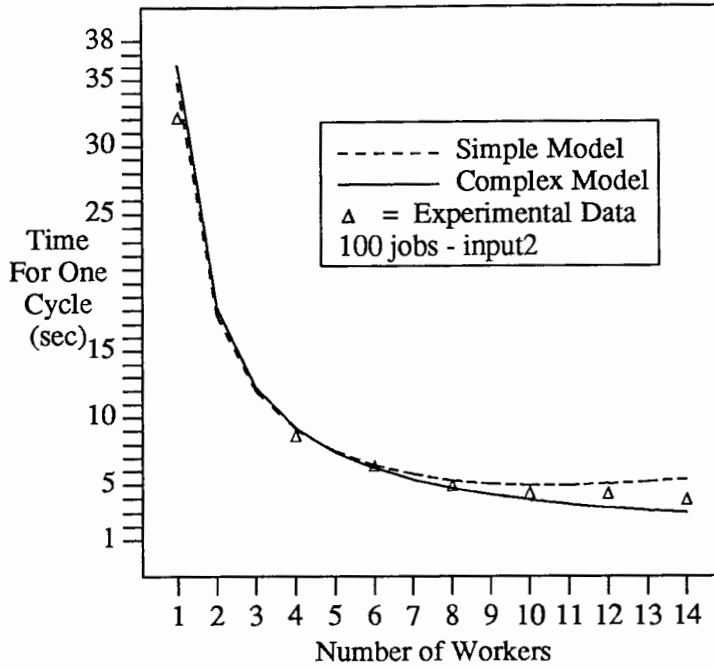


Figure 26. Simple model cycle time versus Complex model cycle time.

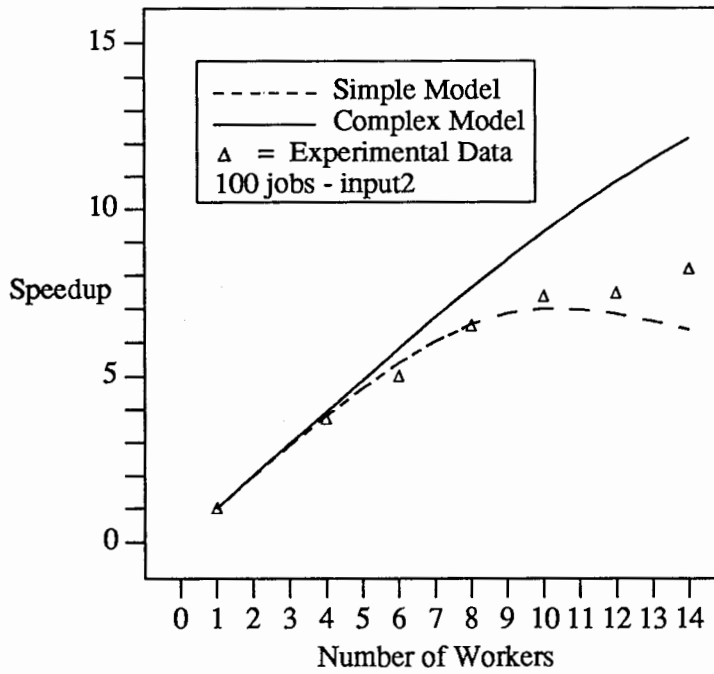


Figure 27. Simple model speedup versus Complex model speedup.

The Simple model has proven to be a resilient model. We were able to isolate and identify the essential characteristics of the system and to represent these and only these in the model. We obtained measurements to support our beliefs about the primary effects on performance and through the techniques provided by Mean Value Analysis were able to accurately replicate the behavior of the system. The key to the Simple model's resilience is in the simplicity of the essential characteristics and the fact that we are using average values to analyze the system. Observations that we have noted throughout this chapter can be summarized as follows:

- Performance of the system is directly influenced by both task size and the number of tasks in a jobpool. Larger tasks and larger numbers of tasks perform best.
- As the number of workers increases, average effective service demand time for the Simple model queueing service center increases. Contention for the file server is reflected in large increases in visitations resulting in a large increase in total service demand time.
- As the number of workers increases, average effective service demand times for delay service centers reflect only small increases.
- Experimental discontinuities observed at the synchronization center in the form of V_{c_lock} have been shown to reflect discontinuities in experimental speedup.
- Using piecewise linear equations for V_{c_lock} in the evaluation of the Simple model provides for an accurate representation of the system behavior before and after breakdown.
- The slope of V_{c_lock} appears to relate to the number of tasks and not the task size. This supports the theory that the more tasks in the jobpool, the more time workers have to desynchronize from their startup pattern, resulting in fewer collisions at the shared resource.

- A dramatic change in slope for V_{c_lock} reflects a breakdown in performance.
- Using the linear range V_{c_lock} curve, a change of a factor of 3 in the average service time, S_k , at the synchronization center shows less than a 10% variation in speedup for the Simple model.

CHAPTER V

CONCLUSION

In this thesis we have developed a paradigm for mapping a large class of computationally intensive problems to a distributed environment. The distributed control, multidimensional pipeline characteristics of the paradigm provide advantages which include load balancing through the use of self-directed workers, a simplified communication scheme ideally suited for infrequent task interaction, a simple programmer interface, and the ability of the programmer to use already existing code.

The paradigm is built on a three level structure of original existing sequential code, an interface, and a coordination framework. The coordination framework of the paradigm is based on structural primitives in the form of jobpools and workers, transformational primitives providing job flow between jobpools, and operational primitives which facilitate the synchronization mechanism required for the jobpool.

The complete mapping of the Design Centering application shown in Chapter III provides information as to the decomposition of an application into jobpools representing pipeline stages and specialist workers dedicated to computations and transformations required by the application for the job flow between jobpools. Experiments were run on a network of SUN 3/50s and data collected from the instrumentation applied to the SPICE stage of the Design Centering application was presented.

Experiments indicate that significant speedup can be obtained by parallelizing native sequential code using this paradigm. Specifically, using 13 workers, experiment input6 for 500 jobs ran approximately 12 times faster when processed in parallel than on a uniprocessor. This translates into a wall-clock turnaround time of approximately 1 hour and 20 minutes as

compared to the uniprocessor turnaround time of 16 hours and 40 minutes. For experiment input6 for 50 tasks using 11 workers the execution was 8 times faster over a uniprocessor. The 50 jobs were completed in 13 minutes compared to a 1 hour and 40 minute turnaround time when processing on a single machine. Performance speedup for experiment input2 for 500 jobs using 12 workers was shown to be a factor of 11. This experiment took 4 hours and 27 minutes to complete on a single machine and 24 minutes using our distributed system paradigm. Even at the low end of the performance spectrum we see an improvement in turnaround time of 6 minutes to complete 50 tasks for input2 using 8 workers as compared to a uniprocessor time of 27 minutes. This is a speedup factor of 4.5. These improvements in turnaround time would please any computational engineer.

To facilitate analysis of the paradigm we developed the Simple model based on queueing network theory. We have seen that the Simple model faithfully replicates the behavior of the system as well as predicts speedup in the linear range and is resilient in the face of a multitude of factors which contribute to the overall behavior of the system. In addition, the model reproduces the system bottleneck shown by experiment to be located at the synchronization center.

Other similar schemes provide comparable approaches to distributed computation but terminate their discussion of speedup prior to saturation of their respective bottlenecks [34, 8, 7]. In the DVLASIC approach to distributed fault simulation [21] and the traveling salesman implementation using workstation clusters [34] the client-server paradigm produces a potential bottleneck when servers request service from the client. FrameWorks7 provides a contractor process which hires and fires employees dynamically at execution time depending on the workload. Each of the schemes we have mentioned in this thesis contains a bottleneck directly related to the addition of processors. Regardless of the implementation, as the number of processors increases, the number of messages in the system increases. No reported mechanism for predicting the behavior of these systems with the addition of processors has been noted. An analysis tool is needed that will add a modicum of intelligence to the assignment of additional

processors so that breakdown in speedup can be avoided.

In our implementation, processors may be added or deleted with ease and at the programmer's control. But we have seen in our experiments that a dramatic breakdown in performance can result with the addition of processors. Queueing network theory provides us with a mechanism for obtaining the optimum speedup given some known parameters within the system.

LIMITATIONS AND FUTURE WORK

A large class of distributed applications where synchronization, scheduling and communication is much more restricted cannot be handled efficiently using our paradigm. In addition, dedicated point to point communication is not directly provided. Applications recognizing strict constraints are more easily parallelized using specialized languages and architectures rather than a general purpose paradigm.

Currently, work is in progress to make the paradigm more accessible to the amateur parallel programmer. This includes a Graphical User Interface (GUI) which will allow the user to represent the flow of the application and its communication patterns via a pictorial representation. Emphasis is on high-level, external control and efforts are in progress to make the interface layer of the three level structure in Figure 3 as small as possible. Ideally, all the code needed to distribute the application across the system would be provided for the user. We believe that the fewer changes to software for cooperation with the controlling facility the better.

No features currently are available to handle failure of a processor during run time. Fault tolerance in a distributed system most often takes the form of a workstation being rebooted by an unsuspecting owner. Node failure can be catastrophic to the application which contains a many-to-1 worker who has no contingency plan for an incomplete set of tasks.

This work has characterized the synchronization mechanism of a distributed computing

paradigm by examining the visitations to the synchronization center. Our expression for total execution time is a refinement of the well known Amdahl's Law which realistically captures the limitations of the system. We show that the portion of time spent executing serial code which cannot be enhanced by parallelization is a function of N , the number of workers in the system. Experiments reveal the critical nature of the communication scheme and the synchronization of the paradigm. Investigation of the synchronization center indicates that as N increases, visitations to the center increase and degrade system performance. Experimental data provides the information needed to characterize the impact of visitations on the performance of the system. This characterization provides a mechanism for optimizing the speedup of an application.

Although work remains to be done, it is anticipated that this thesis provides the foundation for calibration of the synchronization mechanism through a suite of prototypes. The result of the calibration can then be used by the amateur parallel programmer to guide decisions about when and how many workers are to be assigned to specific jobpools. The calibration of the system requires a simple model which retains valuable information. Keeping the model simple helps to alleviate the dependence of the model on the prototype experiments. Ideally, the user would be able to graphically depict the job flow of an application using a GUI, provide some specifications about the application and receive back information from the GUI in the form of optimum number of workers assigned to jobpools and perhaps number of file servers required.

REFERENCES

- [1] Donna Bergmark, "Parallel Programming Languages for Scientists," in *Conpar 88*, ed. K. D. Reinartz, University Press, Cambridge, England, 1989.
- [2] James J. Hack, "On The Promise of General-Purpose Parallel Computing," *Parallel Computing*, vol. 10, pp. 261-275, 1989.
- [3] Leonard Kleinrock, "Distributed Systems," *Communications of the ACM*, vol. 28(11), pp. 1200-1213, 1985.
- [4] Chung-Ta King, Wen-Hwa Chou, and Lionel M. Ni, "Pipelined Data-Parallel Algorithms: Part I-Concept and Modeling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1(4), pp. 470-485, October 1990.
- [5] Michael J. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," *Proceedings of the 8th International Conference on Distributed Computing*, San Jose, CA, June 1988.
- [6] Jason Gait, "A Distributed Process Manager for an Engineering Network Computer," *Journal of Parallel and Distributed Computing*, vol. 4, pp. 423-437, 1987.
- [7] Ajit Singh, Jonathan Schaeffer, and Mark Green, "A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2(1), pp. 52-67, 1991.
- [8] Ozalp Babaoglu, Lorenzo Alvisi, Alessandro Amoroso, and Renzo Davoli, *Paralex: An Environment for Parallel Programming in Distributed Systems*, Technical Report UB-LCS-91-01, Department of Mathematics, University of Bologna, , Bologna, Italy, February, 1991.
- [9] Philip A. Nelson and Lawrence Snyder, "Programming Paradigms for Nonshared Memory Parallel Computers," in *The Characteristics of Parallel Algorithms*, ed. Robert J. Douglass, The MIT Press, Cambridge, Massachusetts, 1987.
- [10] Nicholas Carriero and David Gelernter, *How To Write Parallel Programs: A First Course*, The MIT Press, Cambridge, Massachusetts, 1990.
- [11] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood, *The ISIS System Manual, Version 2.1*, p. 73, September 1990.
- [12] S. Y. Kung, S. C. Lo, S. N. Jean, and J. N. Hwang, "Wavefront Array Processors - Concept to Implementation," *Computer*, vol. 20(7) , pp. 18-33, 1987.
- [13] Patrick A. Duba, Rabindra K. Roy, Jacob A. Abraham, and William Rogers, "Fault Simulation In A Distributed Environment," *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 686-691, 1988.
- [14] E. V. Krishnamurthy, *Parallel Processing: Principles and Practice*, Addison-Wesley Publishing Company, Singapore, 1989.
- [15] K. H. Bennett, "Mechanisms for Distributed Control," in *Distributed Computing*, ed. Fred B. Chambers, Academic Press Inc., Orlando, Florida, 1984.
- [16] Zarka Cvetanovic, "The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems," *IEEE Transactions on Computers*, vol. C-36(4), pp. 421-432, 1987.
- [17] Kai Hwang and Douglas DeGroot, *Parallel Processing for Supercomputers and Artificial Intelligence*, McGraw-Hill, Inc., New York, New York, 1989.

- [18] George S. Almasi and Allan Gottlieb, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1989.
- [19] A. Osterhaug, *Guide to Parallel Programming On Sequent Computer Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [20] K. H. Leung and R. Spence, "Multiparameter Large-Change Sensitivity Analysis and Systematic Exploration," *IEEE Transactions on Circuits and Systems*, vol. CAS-22(10), pp. 796-804, 1975.
- [21] D.M.H. Walker and Daniel S. Nydick, "DVLASIC: Catastrophic Fault Yield Simulation in a Distributed Processing Environment," *IEEE Transactions on Computer-Aided Design*, vol. 9(6), pp. 655-664, 1990.
- [22] R. Spence and Randeep Singh Soin, *Tolerance Design of Electronic Circuits*, Addison-Wesley Publishing Company, Inc., Wokingham, England, 1988.
- [23] V. C. Bhavsar and J. R. Isaac, "Design and Analysis of Parallel Monte Carlo Algorithms," *SAIM Journal Sci Stat. Comput.*, vol. 8(1), pp. 73 - 95, January 1987.
- [24] J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.
- [25] David L. Waltz, "Applications of the Connection Machine," *Computer*, vol. 20(1), pp. 85-97, 1987.
- [26] P. W. Becker, "Finding the Better of Two Similar Designs by Monte Carlo Techniques," *IEEE Transactions on Reliability*, vol. R-23(4), pp. 242-246, 1974.
- [27] E. Wehrhahn and R. Spence, "The Performance of Some Design Centering Methods," *Proc. IEEE International Symposium on Circuits and Systems*, vol. 3, pp. 1424-1438, Montreal, 1984.
- [28] R. S. Soin and R. Spence, "Statistical Exploration Approach to Design Centering," *Proc. IEE Pt. G.*, vol. 127(6), pp. 260-269, 1980.
- [29] Leonard Kleinrock, *Queueing Systems - Volume II: Computer Applications*, John Wiley & Sons, New York, New York, 1976.
- [30] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik, *Quantitative System Performance*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.
- [31] Peter J. Denning and Jeffrey P. Buzen, "The Operational Analysis of Queueing Network Models," *Computing Surveys*, vol. 10(3), pp. 225-261, 1978.
- [32] John L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31(5), pp. 532-533, 1988.
- [33] SUN Microsystems, Inc., *System and Network Administration Manual*, p. 363, Mountain View, CA., March 1990.
- [34] J. N. Magee and S. C. Cheung, "Parallel Algorithm Design for Workstation Clusters," *Software-Practice and Experience*, vol. 21(3), pp. 235-250, 1991.

APPENDIX

From queueing network theory the iterative expression for $Q(N)$ is given in Eq.(16) as

$$Q(N) = RX(N) = \frac{N(1.0 + Q(N-1))}{\alpha + 1.0 + Q(N-1)} \quad \text{for } N \geq 1 \quad (16)$$

A closed form of this equation is shown in Eq.(19)

$$Q(N) = \frac{N\alpha^N + (N - \alpha) \sum_{j=0}^{N-1} \frac{N!}{j!} \alpha^j}{\sum_{j=0}^N \frac{N!}{j!} \alpha^j} \quad \text{for } N \geq 1 \quad (19)$$

Our conjecture is the following:

$$Q = \frac{N(1.0 + Q(N-1))}{\alpha + 1.0 + Q(N-1)} = \frac{N\alpha^N + (N - \alpha) \sum_{j=0}^{N-1} \frac{N!}{j!} \alpha^j}{\sum_{j=0}^N \frac{N!}{j!} \alpha^j} \quad \text{for } N \geq 1$$

and we will verify this is the case using proof by induction.

By induction, show that Eq.(19) is true for $Q(N)$ where $N = 1$ as the base step. Using Eq.(5)

$$Q(0) = 0 \quad (5)$$

our expression for $Q(1)$ as expressed in Eq.(16) shows

$$Q(1) = \frac{1 + Q(0)}{1 + \alpha + Q(0)} = \frac{1 + 0}{1 + \alpha + 0} = \frac{1}{1 + \alpha} \quad (16)$$

By conjecture Eq.(19) for $Q(1)$ becomes

$$Q(1) = \frac{(1\alpha) + (1 - \alpha)1}{1 + \alpha} = \frac{1}{1 + \alpha} \quad (19)$$

which verifies that the base step is true.

Now we assume that the conjecture is true for $N = P$. Then

$$Q(P) = \frac{P \alpha^P + (P - \alpha) \sum_{j=0}^{P-1} \frac{P!}{j!} \alpha^j}{\sum_{j=0}^P \frac{P!}{j!} \alpha^j} \quad (21)$$

From Eq.(16) the definition for $Q(P+1)$ is

$$Q(P+1) = \frac{(P+1)(1+Q(P))}{\alpha+1+Q(P)} \quad (22)$$

Substituting Eq.(21) into Eq.(22) yields

$$Q(P+1) = \frac{(P+1) \left[1 + \frac{P \alpha^P + (P - \alpha) \sum_{j=0}^{P-1} \left[\frac{P!}{j!} \alpha^j \right]}{\sum_{j=0}^P \left[\frac{P!}{j!} \alpha^j \right]} \right]}{\alpha + 1 + \frac{P \alpha^P + (P - \alpha) \sum_{j=0}^{P-1} \left[\frac{P!}{j!} \alpha^j \right]}{\sum_{j=0}^P \left[\frac{P!}{j!} \alpha^j \right]}}$$

Multiplying by $\frac{\sum_{j=0}^P \left[\frac{P!}{j!} \alpha^j \right]}{\sum_{j=0}^P \left[\frac{P!}{j!} \alpha^j \right]}$ and simplifying yields

$$Q(P+1) = \frac{(P+1) \left[\sum_{j=0}^P \left[\frac{P!}{j!} \alpha^j \right] + P \alpha^P + (P - \alpha) \sum_{j=0}^{P-1} \left[\frac{P!}{j!} \alpha^j \right] \right]}{(\alpha + 1) \sum_{j=0}^P \left[\frac{P!}{j!} \alpha^j \right] + P \alpha^P + (P - \alpha) \sum_{j=0}^{P-1} \left[\frac{P!}{j!} \alpha^j \right]} = \frac{A(P+1)}{B(P+1)} \quad (23)$$

To simplify the numerator we first distribute $(P+1)$

$$A(P+1) = (P+1)P \alpha^P + \sum_{j=0}^P \left[\frac{(P+1)!}{j!} \alpha^j \right] + (P - \alpha) \sum_{j=0}^{P-1} \left[\frac{(P+1)!}{j!} \alpha^j \right]$$

Next we add $\frac{(P+1)!}{P!}\alpha^P$ inside the rightmost summation and then subtract it outside the summation to get

$$A(P+1) = (P+1)P\alpha^P - (P-\alpha)\left[\frac{(P+1)!}{P!}\alpha^P\right] + \sum_{j=0}^P\left[\frac{(P+1)!}{j!}\alpha^j\right] + (P-\alpha)\sum_{j=0}^P\left[\frac{(P+1)!}{j!}\alpha^j\right]$$

Combining the summations and simplifying yields

$$A(P+1) = (P+1)P\alpha^P - (P-\alpha)(P+1)\alpha^P + (P+1-\alpha)\sum_{j=0}^P\left[\frac{(P+1)!}{j!}\alpha^j\right]$$

Further simplification yields

$$A(P+1) = (P+1)P\alpha^P - (P+1)P\alpha^P + (P+1)\alpha^{P+1} + (P+1-\alpha)\sum_{j=0}^P\left[\frac{(P+1)!}{j!}\alpha^j\right]$$

which becomes

$$A(P+1) = (P+1)\alpha^{P+1} + (P+1-\alpha)\sum_{j=0}^P\left[\frac{(P+1)!}{j!}\alpha^j\right] \quad (24)$$

Now we simplify the denominator

$$B(P+1) = (\alpha+1)\sum_{j=0}^P\left[\frac{P!}{j!}\alpha^j\right] + P\alpha^P + (P-\alpha)\sum_{j=0}^{P-1}\left[\frac{P!}{j!}\alpha^j\right]$$

by adding $\frac{P!}{(P+1)!}\alpha^{P+1}$ to the leftmost summation and then subtracting it outside the summation. This results in

$$B(P+1) = -(\alpha+1)\frac{P!}{(P+1)!}\alpha^{P+1} + (\alpha+1)\sum_{j=0}^{P+1}\left[\frac{P!}{j!}\alpha^j\right] + P\alpha^P + (P-\alpha)\sum_{j=0}^{P-1}\left[\frac{P!}{j!}\alpha^j\right]$$

Adding and subtracting both $\frac{P!}{P!}\alpha^P$ and $\frac{P!}{(P+1)!}\alpha^{P+1}$ inside and outside the right summation yields

$$\begin{aligned} B(P+1) = & -(\alpha+1)\left[\frac{P!}{(P+1)!}\alpha^{P+1}\right] + (\alpha+1)\sum_{j=0}^{P+1}\left[\frac{P!}{j!}\alpha^j\right] + P\alpha^P \\ & - (P-\alpha)\left[\frac{P!}{P!}\alpha^P + \frac{P!}{(P+1)!}\alpha^{P+1}\right] + (P-\alpha)\sum_{j=0}^{P+1}\left[\frac{P!}{j!}\alpha^j\right] \end{aligned}$$

Combining the summations yields

$$B(P+1) = -(\alpha+1)\frac{\alpha^{P+1}}{P+1} - (P-\alpha)\left[\alpha^P + \frac{\alpha^{P+1}}{P+1}\right] + P\alpha^P + (\alpha+1+P-\alpha)\sum_{j=0}^{P+1}\left[\frac{P!}{j!}\alpha^j\right]$$

Simplifying and rearranging terms yields

$$B(P+1) = -\frac{\alpha^{P+2}}{P+1} - \frac{\alpha^{P+1}}{P+1} - P\alpha^P - \frac{P\alpha^{P+1}}{P+1} + \alpha^{P+1} + \frac{\alpha^{P+2}}{P+1} + P\alpha^P + \sum_{j=0}^{P+1}\left[\frac{(P+1)!}{j!}\alpha^j\right]$$

which when simplified again looks like

$$B(P+1) = \frac{P\alpha^{P+1} - \alpha^{P+1}}{P+1} + \alpha^{P+1} + \sum_{j=0}^{P+1}\left[\frac{(P+1)!}{j!}\alpha^j\right]$$

The final reduction of the denominator yields

$$B(P+1) = \sum_{j=0}^{P+1}\left[\frac{(P+1)!}{j!}\alpha^j\right] \quad (25)$$

Substituting Eqs.(24) and (25) into Eq.(23) gives us

$$Q(P+1) = \frac{(P+1)\alpha^{P+1} + (P+1-\alpha)\sum_{j=0}^P\left[\frac{(P+1)!}{j!}\alpha^j\right]}{\sum_{j=0}^{P+1}\left[\frac{(P+1)!}{j!}\alpha^j\right]}$$

which is exactly the same result given in Eq.(19) which satisfies the conjecture of Eq.(19).