This is to certify that the

dissertation entitled

PROCESSOR SCHEDULING IN A DISTRIBUTED-MEMORY COMPUTING ENVIRONMENT

presented by

Stephen W. Turner

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

Lionel M. Ni
Major professor

Date February 28, 1995

Processor Scheduling In A Distributed-Memory Computing Environment

By

Stephen W. Turner

A Dissertation

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science

February 28, 1995

Abstract

Processor Scheduling In A Distributed-Memory Computing Environment

By

Stephen W. Turner

In recent years, the development of large-scale distributed-memory computers has given the user community unprecedented levels of computing power. In order to effectively use the available computing power, processor scheduling algorithms have been developed that allow many users to share distributed computing resources while obtaining the best possible job turnaround time. However, not all existing scheduling techniques take full advantage of available computing power. For example, in hypercubes, a cluster must normally be allocated as an entire subcube, which can result in high internal fragmentation, as well as poor job performance. Although the distributed workstation environment has recently become popular as a choice for a distributed-memory parallel computer, the problem of scheduling specifically for parallel job execution has not been well studied in this environment.

In this thesis, we present cluster allocation and scheduling methods that can be used to improve performance, in the form of system throughput, in two classes of distributed-memory parallel computers: hypercubes, and the *network of workstation*-based *clustered parallel computer* (NoW-based CPC). For hypercubes, we present an improved cluster al-

location technique that reclaims unused processors from 2D mesh allocation, resulting in improved system throughput. For the NoW-based CPC, we present new cluster allocation and processor scheduling techniques that result in improved system throughput without adversely affecting the *job turnaround time* (JTT) for individual users.

To my parents, and my wife, Dawn

Acknowledgments

I wish to thank my thesis advisors, Professors Lionel M. Ni and Betty H.C. Cheng. Without their direction and support, this dissertation would not be possible. They provided me with a great deal of encouragement, a direction for my research, and a great sense of organization. I have learned a great deal from them during my time in graduate school. I also wish to acknowledge my committee members, Professors Anthony Wojcik and David Yen, for their support. As the Chair of our department, Dr. Wojcik has at many times in the past provided assistance. Professors Wojcik and Ni are primarily responsible for my decision to study at Michigan State University. Professors Richard Enbody, Philip McKinley, John Forsyth, and Donald Weinshank all deserve credit for their willingness to talk to me and provide assistance at numerous times in the past.

I also wish to thank the members of the staff of the Computer Science department, particularly Cathy Davison, Laura Mae Higbee, Linda Moore, and Michele Sidel. I cannot count the number of times they have helped me through the intricacies of the MSU bureaucracy. My assistantship supervisor for the past four years, Frank Northrup, also deserves credit for his support, as well as his flexibility with my work schedule.

Many fellow students and colleagues from the department have provided support and preserved my sanity. The lunch crowd, especially Dr. Reid Baldwin, Barbara Birchler, Dr.

David Chesney, Barbara Czerny, Dr. Marie-Pierre Dubuisson, Edgar Kalns, Ron Sass, and Dr. Christian Trefftz, provided many hours of entertaining conversation and camaraderie when it was greatly needed. I regard all of them as good friends. Dr. Maureen Galsterer, whom I have known since I entered graduate school, deserves special acknowledgement for being a great friend during all these years. We have the special bond that comes from sharing a *common enemy.* Patrick Edsall, whom I knew in both undergraduate and graduate school, introduced me to my wife, and he was a good friend in the early days when I knew nobody else at MSU.

I wish to acknowledge the love and support of the many members of my extended family. My father, Professor Walter Turner, has been my role model all my life. He provided the standard to shoot for and the example to follow. My mother, M. Patricia Gioia, and her husband, Professor Anthony Gioia, deserve a great deal of credit for being there, providing support, and giving sage advice when I needed it. Of course, my siblings, Connie, Anne, Roberta, Nancy, Nick, and Jane (in order of seniority) all deserve mention for the influences they have had on my life. I wish to thank my wife's parents, Jacquelyn and David Moore, for welcoming me into their family and giving us so much help.

Foremost, I wish to acknowledge and thank the most important person in my life, my wife Dawn. She deserves the most credit, for putting up with me during all of my ups and downs, for loving me, and for providing support in times of need.

Last, but least, I'd like to thank the Academy, especially all the little people...

TABLE OF CONTENTS

List of Tables

List of Figures

Chapter 1

Introduction and Motivations

In recent years, computing power has been made available to the user community at a larger scale than ever before. The rapid development of processor technology for the scientific workstation and personal computer markets has contributed to the development of *scalable parallel computers* (SPCs). SPCs are a viable platform on which to solve the so-called grand-challenge problems, and such systems are usually characterized by the distribution of memories among a collection of *nodes.*

There are a wide variety of SPC architectures available, although they can be classified into two popular general approaches: (1) *massively parallel computers* (MPCs), including meshes, hypercubes, *multistage interconnection networks* (MINs), etc., and (2) *networks of workstations* (NoWs) (also referred to here as *workstation clusters*). In an MPC, the nodes, each of which consists of a processor, local memory, and other supporting devices, are connected by either a *direct* (point-to-point) or *indirect* (switch-based) network. Examples of direct network MPCs include the hypercube and 2D mesh. Examples of indirect network MPCs include MINs and crossbar networks. Nodes in a NoW are similar to those in an MPC, but the two architectures are different in two primary ways: (1) the network bandwidth

available in a workstation cluster is generally lower than that available in an MPC, and (2) local disks may be easily added to individual workstation nodes, which is almost impossible in an MPC. Workstation clusters can be connected by either *shared-medium* or switch-based networks. Examples of shared-medium networks for workstation clusters include Ethernet and Token Ring. Examples of switch-based networks for workstation clusters include the DEC GIGAswitch and the ATM switch.

MPCs and NoWs are expandable and can achieve a proportional increase in performance without changing their basic architecture. By increasing the capacity of certain hardware components, performance increases can be achieved in computational capacity, memory bandwidth and capacity, internal interconnect (network) bandwidth, and I/O bandwidth and capacity. However, in order to efficiently utilize the increased performance offered by a SPC, an effective *processor scheduling* policy must be used. In a parallel computer, processor scheduling involves ordering the execution of arriving jobs (*job dispatching*), as well as allocating sets of processors, called *clusters*, for the scheduled jobs (*cluster allocation*). For a given application, the optimal cluster size to achieve its lowest execution time is often much lower than the total number of processors available in the parallel computer. Tzen and Ni [1] demonstrated that exceeding the optimal number of processors can lead to performance degradation due to the overhead involved in communication amongst many processors, as well as that of allocation of programs and data, etc. to the processors in a distributed memory machine. Therefore, since the optimal cluster size of many jobs is less than the total number of processors available, it is possible for many different user jobs to execute concurrently in a distributed memory parallel computer.

The MPC and NoW environments may also be characterized as *high-performance computing environments*, in which the primary concern is performance as perceived by the

user. In this kind of environment, it is desirable to maximize system performance without adversely affecting the performance of user jobs.

1.1 Processor Scheduling

Classic work in processor scheduling for distributed memory machines has focused on the static scheduling problem [2]. For this well-known NP-complete problem, specific job characteristics, such as the number of jobs, the cluster size, the execution time of each job, and the memory requirements of each job, can be determined in advance. A static scheduling algorithm considers some or all of these characteristics and determines a schedule that will minimize the total completion time for a set of pre-submitted jobs.

Static scheduling may be suitable for real-time and/or batch scheduling systems, in which overall completion time is the primary concern. However, in a dynamic environment, in which jobs arrive in a stochastic stream over time, it is impossible to know most job characteristics beforehand. For an incoming job, the system must rely on currently observable information to make scheduling decisions. Such information includes the system's current status, as well as a number of characteristics about executing jobs. These characteristics may include a job's cluster size, its degree of parallelism, its execution time, its message-passing behavior, its memory requirements, and its CPU utilization. A scheduler operating in a dynamic environment may use some or all of these observed characteristics to make job dispatching and cluster allocation decisions about incoming jobs. Due to their interaction, the cluster allocation and job dispatching algorithms may also depend on one another. For example, if the first-come, first-served (FCFS) queueing discipline is used by the job dispatcher, a situation may often occur in which there is no suitable cluster for the job being scheduled, yet there are suitable clusters for many jobs waiting in the queue.

There are two primary criteria for measuring the effectiveness of a system's processor scheduling algorithms: *job turnaround time* (JTT) and *system throughput*. JTT measures the time a user's job spends in the system from when it is enqueued to when it finishes its execution, and it is the primary measure of performance from the user's perspective. System throughput measures the number of jobs that are executing in the system within some unit of time. A good system scheduler in a dynamic environment maintains a high system throughput with a low average JTT.

In a parallel system, processor scheduling can be further classified as occurring at either the *system* or *node* level. Figure 1.1 illustrates the process of system-level scheduling. In the figure, solid arrows represent the flow of user jobs among major logical steps of the scheduling process. The job dispatcher first chooses among jobs in a global job queue, sending its choice to the cluster allocation algorithm. The cluster allocation algorithm reserves a cluster of processors for the use of the chosen job. The figure also shows that jobs may not necessarily be given exclusive access to their clusters. That is, one processor belongs to two different clusters, containing threads from both job A and job B.

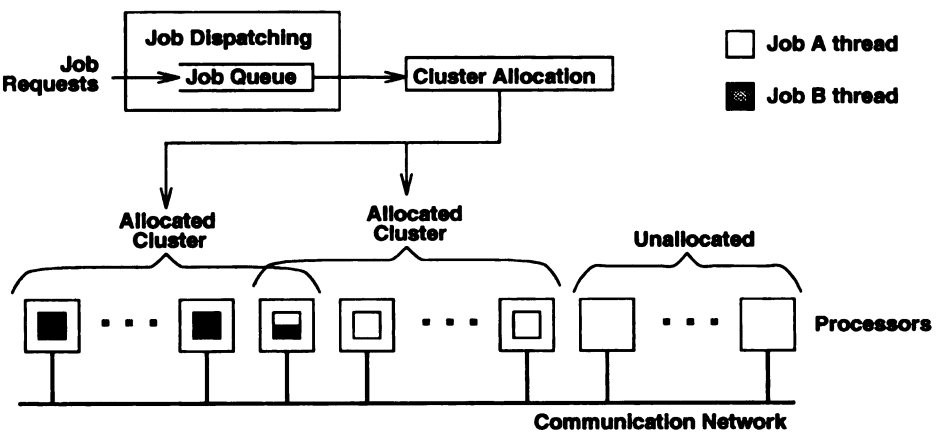

Figure 1.1: System-Level Processor Scheduling.

Figure 1.2 illustrates an example of node-level scheduling, in which three jobs share access to a single processor through some form of processor sharing. The figure illustrates the process of context-switching, which occurs when a job's thread either expires its time quantum or is blocked. This figure presents a simplified model, in which a thread simply returns to the process queue when it is removed from the processor.

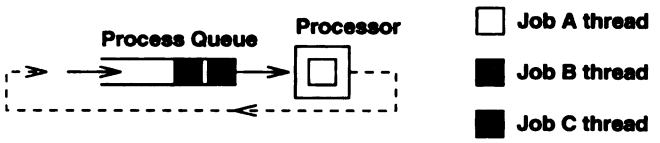

Figure 1.2: Node-Level Processor Scheduling.

The focus of the research presented here is how to design good, practical processor scheduling algorithms that increase performance from the standpoint of both the user and the system. We investigated both job dispatching and cluster allocation in two classes of distributed memory multiprocessors: hypercubes and NoWs. First, we devised a new cluster allocation technique for hypercubes that increases system throughput and decreases observed job turnaround time as compared to existing subcube allocation methods.

Second, we studied cluster allocation techniques for NoWs that take advantage of processor sharing in a high performance computing environment consisting of scientific workstations connected together by a high-speed network. At the node level, our work used existing Unix operating system facilities to develop processor sharing techniques that adapt to the execution of parallel jobs running on individual nodes. At the system level, the job dispatching and cluster allocation methods use information provided by individual workstations to make scheduling decisions that take advantage of idle CPU cycles on individual workstations.

1.2 Thesis Statement and Research Overview

Thesis Statement

The thesis of our research is that existing processor scheduling facilities, in the form of cluster allocation and job dispatching techniques, can be used to improve the performance of parallel jobs in two classes of distributed memory architectures: the hypercube and the workstation cluster viewed as a clustered parallel computer *(CPC). The performance of the parallel jobs improves in terms of both job turnaround time and system throughput. This performance improvement is realised by taking advantage of underutilized CPU resources.*

1.2.1 Research Objectives

Our research can be divided into two primary areas: cluster allocation in hypercubes, and processor scheduling for networks of workstations. Our work with cluster allocation in hypercubes accomplishes two main objectives: increased performance, and practicality. In terms of performance, our cluster allocation method increases overall system utilization and decreases average JTT by eliminating the internal fragmentation that results from subcube-only cluster allocation. Our cluster allocation method, the *auxiliary free list* (AFL) method [3], allows unused processors in a subcube to be reclaimed by the system for use in other jobs. In terms of practicality, the AFL method can be implemented with any existing subcube allocation method on numerous existing hypercube multiprocessors.

Networks of workstations are very popular as a choice for high-performance parallel computation, primarily due to their flexibility, expandability, and low cost relative to massively-parallel processors of similar computational capability. While the individual workstations may be used in the traditional manner as single-user computers, the cluster of workstations may also be used as a compute server. Our research examines the scheduling of resources

in a workstation cluster from this standpoint, and our primary goal is to preserve the best possible JTT available to the user while achieving the secondary goal of improved system throughput.

In order to achieve the goals stated here, we have developed processor scheduling techniques that take advantage of idle time in CPUs. This includes system-level and node-level methods, and we present a study of their impact on system performance. The major components of this research include:

- The development of the *auxiliary free list* (AFL) algorithm for subcube allocation. This cluster allocation technique allows two-dimensional mesh requests of any size to be allocated in a hypercube (limited by the number of processors in the hypercube).

- A simulation study of the AFL method for subcube allocation in hypercubes. We demonstrated that the AFL method, when combined with the well-known Free List subcube allocation method, can increase system utilization and decrease average JTT, relative to using the Free List method alone. Furthermore, the AFL method can be implemented on any existing hypercube as a supplement to any existing subcube allocation method.

- A feasibility study that demonstrated that timeshared execution of parallel jobs in a cluster can be used to improve system throughput, as well as user JTT.

- An experimental cluster allocation facility, *Elmer*, that uses realtime scheduling mechanisms to allow the prioritized, timeshared, scheduling of parallel jobs in a workstation cluster, as demonstrated in the feasibility study.

- Cluster allocation algorithms for use in a network of workstations. These algorithms are used by the Elmer facility to perform cluster allocation at the system level for

NoW-based CPCs. The algorithms are characterized by their use of the current degree of multiprogramming, as well as CPU utilization, at individual nodes in the system.

Throughout our investigations, we have explicitly addressed the practicality of our proposed methods. This is illustrated in three major focuses of our work. First, we made realistic assumptions so that the results of our research may be implemented on existing machines. For example, it is well-known that workstation clusters can be subject to periods of intense activity during the day, yet sit almost idle during the night [4, 5]. We concentrated on making more effective use of the workstation environment during periods of low usage, allowing for users to log in without fear of having their workstation resources consumed by the computing demands of some parallel job. Second, much of our research involved an actual implementation, simplifying the task of parallel programming for users, and providing an example implementation of our research. For example, the AFL method for cluster allocation in hypercubes can be implemented on any existing machine using any known subcube allocation method. Furthermore, our implementation of a scheduling facility for workstation clusters facilitates parallel programming. Currently, one of the most commonly available programming environments for distributed workstation clusters is PVM [6]. While PVM allows users to program heterogeneous computing environments as a single computational resource, it requires them to perform all scheduling and allocation of CPUs in workstation environments. Our facility eliminates this requirement. Third, our research involving workstation clusters is applicable to other architectures, including different workstation environments, as well as other classes of distributed memory multiprocessors.

1.2.2 Research Contributions

The primary contribution of this research is the demonstration that it is possible for system resources to be reclaimed to improve system and user performance in a distributed computing environment. We developed scheduling techniques that improve system throughput and decrease observed user JTT. Parallel applications rarely, if ever, are able to fully utilize the hardware resources over which they have control. However, most distributed parallel programming environments give user jobs exclusive control over the cluster of processors on which they execute. In some cases, such as subcube allocation for hypercubes, the cluster allocation technique may allocate too many processors to a job. In other cases, such as in a workstation cluster environment, parallel jobs contain message-passing overhead that results in idle CPU time on individual processors. Both of these examples illustrate forms of internal system fragmentation, in which certain system resources (the CPUs) are underutilized. Our research presented methods that overcome these limitations by giving other jobs access to these underutilized processors.

Our research involving cluster allocation in hypercubes demonstrated that unneeded CPU resources can be reclaimed by the system without adversely affecting the performance of user jobs while increasing system utilization and throughput, thereby decreasing average JTT. Our research involving scheduling for networks of workstations demonstrated the feasibility of the timesharing of jobs in distributed memory computing environments. Furthermore, we provided cluster allocation techniques that take advantage of timeshared job execution. We showed that improved system throughput can be attained at little or no cost to user JTT in a high-performance computing environment.

1.3 Organization of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 defines the problem of processor scheduling and presents work relevant to our research. A classification of parallel architectures according to a layered architecture of parallel programming models [7] is presented. In addition, work related to our investigations in scheduling hypercubes and workstation clusters, including the general class of distributed memory parallel machines are discussed. Chapter 3 presents the AFL algorithm for cluster allocation in hypercubes, including the motivation for, development of, and complexity analyses of the algorithm. Chapter 4 presents the results of a simulation study that compared the AFL cluster allocation method to the *free-list* subcube allocation method for numerous different workloads in a 1024 processor hypercube. Chapter 5 presents our initial study into the feasibility of timesharing in a *Clustered Parallel Computer* (CPC). The results of this study verified the feasibility of using timeshared scheduling in distributed-memory parallel computers. Chapter 6 describes the scheduling facility that we implemented to accomplish the timeshared scheduling of jobs in a CPC, also presenting the experimental results used to verify the operation of our scheduling facility. Finally, Chapter 7 presents our conclusions and directions for future work.

Chapter 2

Problem Statement and Related Work

Processor scheduling in parallel machines consists of two major components: job dispatching and cluster allocation. The purpose of a job dispatching algorithm is to order the execution of enqueued user jobs. The purpose of the cluster allocation algorithm is to determine a set of processors on which to allocate the next scheduled job.

In a parallel system, the scheduling of work on processors occurs at both the system and node levels. At the system level, the job dispatching algorithm implements some form of queueing discipline for newly-submitted user jobs. At the node-level, the processor sharing algorithm determines the order of execution of jobs on individual processors. The most common form of job dispatching algorithm is to place arriving jobs in a FCFS queue. This method ensures fairness to all jobs, because every job is guaranteed to eventually receive service. However, the disadvantage of FCFS is that a large job waiting in the head of the queue can block access to smaller jobs waiting behind it in the queue, even if the system has the resources to service the smaller jobs. Therefore, the use of an FCFS queue can

result in lower system utilization. The processor sharing algorithm determines the *degree of multiprogramming* at individual nodes, which is the number of jobs that execute on one processor at once. Job dispatching and processor sharing algorithms can be highly dependent on the architecture of a parallel computer. For example, a hypercube may implement no processor sharing algorithm whatsoever, while a distributed workstation cluster might use a variation of the round-robin timesharing technique to implement processor sharing.

Cluster allocation techniques are also highly dependent on the architecture of the parallel computer. Due to the difficulty of connecting hundreds or thousands of processors together, the interconnection network of a machine may be constructed according to a regular geometry, such as a hypercube or 2D mesh. Such geometries impose restrictions on communication among processors, which makes it desirable for programs to execute on clusters of processors that are grouped closely together. For example, a cluster in a 2D mesh consists of a contiguous $X \times Y$ submesh, as opposed to a randomly-picked set of $X \times Y$ processors. The geometry can also affect the number of processors allowed to be allocated to an application. For example, in hypercubes, cluster allocation is usually restricted to subcube allocation, in which a user job must execute on a subcube of 2^k processors (for some $k \geq 0$).

In other systems, cluster allocation techniques are somewhat less complicated. For example, in a bus-based shared memory multiprocessor, the communication medium is a shared bus coupled with a shared memory. In such a system, the limitation on communication exists only in the bandwidth of the bus and memory. Communication occurs at a uniform rate from one processor to another, as well as between any processor and any memory location. Therefore, cluster allocation in a bus-based system is as simple as obtaining a

set of processors, although the total number of processors available may be limited by the bandwidth of the memory and bus.

The wide variety of parallel computer architectures available today implies that there are a wide variety of processor scheduling algorithms. Factors affecting scheduling decisions include the memory access speed, the method of accessing memory, and the type of interconnection network. We present a classification of different parallel architectures that will facilitate the study of the many different methods that have been developed. Ni [7] defined a layered architecture of parallel programming models. The first two levels of this model define the memory architecture and the memory address space and are presented in Figure 2.1.

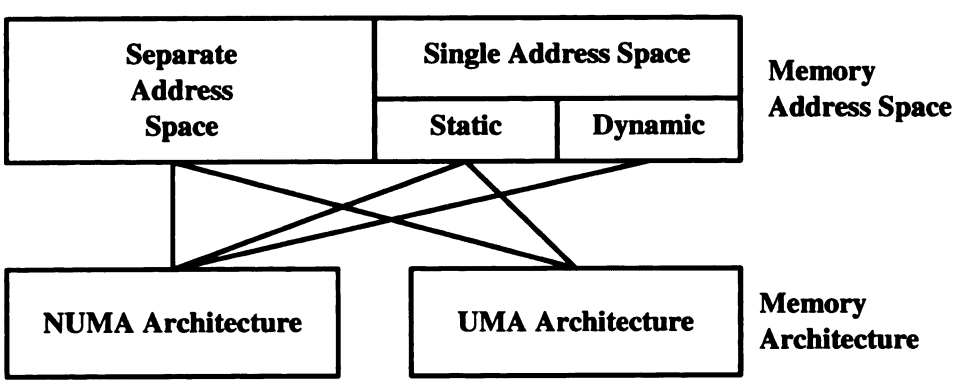

Figure 2.1: Memory Architecture and Memory Address Space Models.

At the first level, the memory architecture defines the method of memory access used by the system hardware. In NUMA (*Non-Uniform Memory Access*) architectures, the speed at which a word in memory is retrieved is dependent on the physical location of the memory word. For example, a processor in a NUMA machine might have access to both local and remote memory; local memory access can be considerably faster than access to remote memory. In UMA (*Uniform Memory Access*) architectures, memory access speed is not

dependent on its physical location. That is, speed of access is the same for all physical locations.

The memory address space, which can be either *single* or *separate*, defines the memory as it is seen by an executing parallel program. In the separate address space model, each sub-process (or *thread*) of a parallel application has its own address space, to which no other thread has direct access. In the single address space model, the program views memory as one large, uninterrupted address space. Each thread of a parallel application may access any location in this address space. This model can be further classified as either static or dynamic binding. In a static binding single address space, there is exactly one mapping of a virtual memory address to a physical location (that is, once the virtual memory address is assigned a physical address, it remains that way for the duration of program execution). In a dynamic binding address space, virtual memory addresses may migrate from one physical address to another.

Memory Access Method	**Address Space**	**Binding Type**
NUMA	single	static/dynamic
	separate	static
UMA	single	static
	separate	static

Table 2.1: Architecture classification according to memory access.

From the model presented in Figure 2.1, we can derive a taxonomy of parallel architectures classified by their memory access methods, as listed in Table 2.1. For separate address space NUMA machines, in particular, the problem of processor scheduling still contains many open issues involving both job dispatching and cluster allocation. Sections 2.1 through 2.3 identify and discuss some of these issues as they pertain to hypercubes and

networks of workstations. Section 2.1 first presents a review of the research in processor scheduling techniques for NUMA architecture multiprocessors, including single and separate address space machines. Section 2.2 then discusses the problem of cluster allocation in hypercube multiprocessors, presenting a review of literature relevant to the research presented in Chapters 3 and 4. Section 2.3 defines the issues of processor scheduling in a workstation cluster and presents a review of literature relevant to the research presented in chapters 5 through 6.

2.1 Processor Scheduling in NUMA Architecture Multiprocessors

Non-Uniform Memory Access multiprocessors are commonly considered as the class of distributed memory separate address space machines. However, numerous examples of shared memory NUMA multiprocessors exist, such as the Kendall Square Research KSR1 [8], the BBN GP1000 and TC2000 [9, 10], and the Cray T3D [7]. In addition, many other distributed memory machines maintain separate address space as their standard programming model, yet support a shared memory environment. Examples include the nCUBE 3 hypercube [11], and MIT's J-machine [12], a 3-D torus.

Most NUMA architecture multiprocessors can be characterized by the method of interconnection among processors, which is normally some structured topology (e.g., hypercube or mesh). Examples of NUMA multiprocessor interconnections include the Multistage Interconnection Network (MIN), the hypercube, the 2D and 3D mesh or torus, the FAT tree [13], and the ring. Examples of the MIN include the BBN GP1000 and TC2000 [9, 10], IBM's RP3 [14] and SP-1 [7], and the NYU Ultracomputer [15]. Examples of the hyper-

cube include the Intel iPSC [16], the nCUBE 2 and 3 [17, 11], and the Thinking Machine Corporation's CM-2.

2.1.1 Single Address Space Machines

Static Binding Machines

In a single address space NUMA machine, all memory references are part of a single global address space. The implementation of this global address space may either be by hardware support for direct remote memory access, as in the case of the Cray-T3D, or in a special network that separates processors from memories, as in the case of the BBN GP1000, BBN TC2000, IBM's RP3, and the NYU Ultracomputer.

The Cray T3D is a massively parallel computer consisting of a 3-dimensional torus that connects nodes together. In this machine, a node consists of two PEs (processor/memory pairs). Currently, no information is available as to processor scheduling techniques on the Cray T3D.

In the case of the BBN GP1000 and TC2000, as well as the IBM RP3 and SP-1, processors are connected together via a *multistage interconnection network* (MIN), in which each stage of the network consists of a line of small crossbar switches. The outermost two stages are connected on one side to processors and on the other to memories (or, alternatively, reconnected to the processors), while the inner stages of the MIN are connected so that a direct link may be established, through switching through the MIN, from one processor to another (or from a processor to a memory module). To our knowledge, there have been no substantive studies on processor scheduling in MINs.

Dynamic Binding Machines

In a single address space dynamic binding NUMA machine, the binding between a virtual address and a physical address is not permanent. This class of machines attempts to take advantage of the principle of locality by moving the physical location of data when necessary. That is, if a processor issues a memory reference, and the data it refers to is not stored locally, then the dynamic binding mechanism moves the data over to the local processor's memory. There is only one known example of this kind of machine, the Kendall Square Research KSR 1 [8].

The KSR 1 is implemented using a fat-tree [13] and is scalable from 8 to 1,088 processors. The lowest level of the tree connects up to 32 processors using a form of token ring. Cluster allocation on the KSR 1 is accomplished hierarchically according to the organization of the system, implying that, on a fully-configured system, a process may allocate from 8 to 1,088 processors, increasing the size of the cluster by adding more leaves of the fat-tree.

2.1.2 Separate Address Space Machines

Separate address space parallel machines are often characterized by the geometry of their interconnection networks. Although many different interconnection network geometries have been proposed, only a few different geometries have been implemented in actual machines. These include the hypercube [18], the k-dimensional mesh, and the multistage interconnection network (MIN) [9, 10]. The research presented in this section primarily considers cluster allocation techniques in meshes and MINs, while Section 2.2 presents research relevant to our investigations in hypercubes.

Cluster allocation in mesh-connected parallel computers is concerned with obtaining 2-dimensional submeshes for allocation to user applications. The first proposed method, by Li and Cheng, is known as the *two-dimensional buddy* (2DB) strategy [19]. This strategy

is a generalization of the buddy strategy developed for subcube allocation in hypercubes. For this strategy to apply, meshes must be square, having side lengths that are a power of two. Additionally, each submesh request must be a square submesh whose side lengths are a power of two. This method suffers from two primary drawbacks: (1) large internal fragmentation results from forcing the allocation of square submeshes with side length the power of two; (2) it is not applicable to general 2-dimensional meshes.

Chuang and Tzeng [20] developed the *frame sliding* strategy to address the drawbacks of the 2DB strategy. In this method, the incoming request dimensions are treated as a *frame* used to search for available space in the mesh. The search begins at the lowest leftmost available processor. When a frame area is unavailable, the frame is slid horizontally or vertically with a stride equal to the width or height, respectively, of the frame. Zhu [21] recognized that the frame sliding method also suffers from external fragmentation, as well as the inability to recognize all available free submeshes in its searching technique. Therefore, he developed the BF and FF (best-fit and first-fit) strategies, which are capable of recognizing any available submesh.

For a $w \times h$ mesh, either of Zhu's BF or FF strategies requires $O(w \cdot h)$ time to allocate a submesh, which can be prohibitive in large meshes. To eliminate this problem, Bhattacharya *et al* [22] developed a method based on computing free submesh regions. For a mesh containing n currently-allocated submeshes, their method requires $O(n^2)$ time to allocate a submesh. They point out that since n is often small, their method usually outperforms Zhu's, unless $n \geq \sqrt{w \cdot h}$.

For job scheduling on meshes, most authors assume FIFO queueing and use some submesh allocation algorithm to perform cluster allocation [19, 20, 21]. However, Bhattacharya *et al* [22] examined lookahead processor scheduling in the context of submesh

allocation. They modified two strategies: BF and WF (worst-fit) to examine the effects of scheduling if the contents of the job queue are considered. In all cases, their lookahead method achieved better performance than the comparable method with no lookahead.

2.2 Cluster Allocation in Hypercubes

Most research on cluster allocation in hypercubes is concerned with obtaining a subcube for allocation to a user application. In a hypercube of degree n ($n \geq 0$), a subcube is a cluster of processors that is, itself, a hypercube of degree k ($0 \leq k \leq n$). Our investigations examined the question of whether processors left over from a subcube allocation can be reclaimed by the system for use by other jobs. The motivation for this work is that the optimal cluster size for a particular job may be much lower than the dimension of subcube required to obtain the required number of processors for the job. Because having too many processors can be detrimental to the performance of a job, we proposed and studied the *auxiliary free list* (AFL) strategy [3], in which 2D meshes are used as clusters in hypercubes. The AFL strategy achieves a more effective utilization of processors in hypercubes. Details of this method are given in s Chapters 3 and 4.

The most well-known subcube allocation methods are the buddy system [23], the gray code method [24], and the free list method [25]. Other variations include the modified buddy strategy [26], the associative memory method [27], and the MPP method [28].

The buddy system was first proposed in 1965 as a memory storage allocation scheme. For a hypercube Q_n, it uses 2^n bits to keep track of the availability of the 2^n nodes. A k-cube ($0 \leq k \leq n$) is located by finding a contiguous sequence of 2^k bits whose addresses start with an integral multiple of 2^k. For a subcube of dimension k, the buddy system can recognize 2^{n-k} subcubes.

The gray code method is a variant of the buddy system. The buddy system searches a list of addresses that are arranged in ascending order from 0 to 2^n in an n-cube. By arranging the addresses as a gray code listing, the gray code method is able to detect twice as many subcubes as the buddy system.

The so-called modified buddy strategy [26] was proposed as an improvement on the buddy system and gray code strategies. Like the gray code method, it works by making a more effective search of the list of processor addresses and their corresponding allocation bits.

The free list method is unique due to its method of search for subcubes. Instead of a list of processor addresses with allocation bits, the free list method only maintains a list of currently available subcubes. When a subcube of dimension k is requested, a free k-cube is either immediately allocated, or the free list is searched in higher dimensions until a larger subcube is found, decomposed (broken into smaller subcubes), and a k-cube allocated.

The free list method is the only practical method that provides complete subcube recognition. That is, for an n-cube and a k-cube request, there are $2^{n-k}\binom{n}{k}$ possible recognizable subcubes, since there are $\binom{n}{k}$ ways to arrange the "don't care" bits of a k-cube, and 2^{n-k} ways to arrange the remaining $n-k$ bits. For a comparable request, the buddy system is able to recognize 2^{n-k} subcubes, and the gray code method can recognize 2^{n-k+1} subcubes.

The associative memory method [27] also provides complete subcube recognition. However, it requires access to an associative memory to determine subcube availability. For a subcube of dimension n, the memory required is 2^n n-bit words. For example, a 10-cube would require an associative memory of 1024 10-bit words. Although it has constant time performance for subcube allocation, the hardware costs of this associative memory would be prohibitive in a large scale hypercube.

Krueger *et al* [29] studied different job scheduling techniques in hypercubes. They argued that job scheduling can have a greater effect on system throughput and job turnaround time than the choice of subcube allocation method. Instead of FCFS scheduling, they proposed and implemented a family of scheduling disciplines, called *scan*, that have a great effect on system performance. Their studies indicated that an effective job scheduling strategy is more important than the cluster allocation strategy in hypercubes.

2.3 Processor Scheduling in Workstation Clusters

A network of workstations (or workstation cluster) can be defined as a set of scientific workstations connected together by some form of network. Normally, workstations are characterized as containing their own disk drives, memory, operating system, keyboard, and monitor. That is, a workstation is an independent computer capable of functioning without the aid of services from other computers. Parallel programming in workstation clusters is typically performed using user-level software systems, such as PVM [6], Linda [30], P4 [31], or MPI [32]. Such software systems enable the user to treat networks of machines as single "virtual machines" by implementing message-passing and other protocols over existing network protocols.

To date, very little research has been published on the subject of scheduling such parallel jobs on clusters of workstations. Workstations are not originally intended for the execution of large-scale parallel jobs, and the use of standard network protocols, coupled with software systems like PVM, Linda, or P4, makes it likely that the message-passing performance of jobs in that environment will be lower than that on a purpose-built MPP. Although research is under way towards building low-overhead message-passing protocols for networked computer systems [33], workstation clusters are still at a disadvantage because they must be

general enough to accommodate the software for which the network protocols were originally built.

Lower message-passing performance can result in lower program performance, as well as significant amounts of idle CPU time. With this assumption, our research examined the effects of allowing multiple jobs to execute on individual workstations. Measurable improvements in system throughput were observed, often at little cost to the performance of individual jobs. Our research focused on scheduling algorithms, at the system and node level, that are optimized towards maximizing the benefit of allowing time shared job execution in a cluster of workstations.

A central issue in research in scheduling for a workstation cluster is how it is perceived as a computational resource. There are two different primary models. A workstation cluster can be treated as a collection of independent computers, each of which is "owned" by its primary user, or it may be treated as a compute server, available for the execution of any job. Under the ownership model, the quality of service to a workstation's owner is the primary concern. If the owner is currently using his/her workstation, then he/she should perceive no change in responses from the system. The compute server model views the set of workstations in much the same way as a massively parallel processor might be viewed, which is as a single computational resource on which to execute parallel jobs.

A number of studies have examined scheduling in workstation clusters . Most of these studies are concerned with the ownership of a workstation by its primary user, and are thus concerned with preserving the owner's quality of service while finding idle cycles with which to execute jobs [4, 5, 34, 35, 36]. Litzkow *et al* [5] developed Condor as a means to exploit idle workstations during non-peak usage times. In developing the scheduling algorithm for Condor, Mutka and Livny [4] addressed the problem of long-term scheduling for jobs that

execute for long periods of time with little need for interaction or interprocess communication. Theimer and Lantz [34] examined the tradeoffs between centralized and decentralized scheduling facilities, concluding that although decentralized facilities are more fault tolerant, centralized facilities are more scalable and perform better. Stumm [35] developed a decentralized facility for automatically scheduling jobs in a distributed, heterogeneous computing environment. Stumm argued that parallel jobs require exclusive access to the clusters allocated to them in order to execute at optimum efficiency.

All of the above workstation scheduling methods use a *conservative* approach to preserving the owner's quality of service, which is to ensure that remote jobs are not executed on previously occupied workstations. In contrast, Krueger and Chawla [36] use *priority resource allocation* in the form of prioritized CPU scheduling, file system access, and memory management. Their argument is that a significant portion of idle workstation cycles go unused by other distributed schedulers simply because the cycles exist on workstations previously occupied. They showed that it is possible for remote and local jobs to coexist through judicious use of priority mechanisms.

One study has examined the workstation cluster in the context of high performance parallel computing. Atallah *et al* [37] examined the problem of using a network of workstations for computationally intensive parallel applications. They presented analytical arguments that co-scheduling (gang scheduling) is necessary to guarantee that sub tasks start concurrently and execute at the same pace, although they presented no conclusions about the performance of gang scheduling in a workstation cluster.

In a study not specifically related to workstation clusters but relevant to our research, Setia *et al* [38] compared the performance of simple round-robin scheduling with that of "run to the end of phase" scheduling, in which a job is suspended when it reaches the end

of a computation/communication phase. They proposed a processor scheduling method in which incoming parallel processes share time on equal-sized clusters. By using analytical and simulation studies, they concluded that allowing processor sharing on individual nodes can be of some benefit to system throughput performance.

Chapter 3

Cluster Allocation in Hypercubes

This chapter discusses our investigations in the area of processor scheduling in hypercube multiprocessors. Specifically, we addressed the problem of cluster allocation [3]. Our research achieved an improvement in both response time and overall system utilization by allowing 2D mesh clusters to be allocated within a hypercube, making obsolete the traditional requirement that only subcubes be allocated. The technique we introduced, known as the auxiliary free list method, can be implemented on existing systems with any known subcube allocation method.

3.1 Problem Definition

The problem of subcube allocation has been studied extensively to maximize processor utilization and minimize system fragmentation in hypercubes. Several strategies have been proposed and implemented for subcube allocation, including the buddy strategy [23], the gray code (GC) strategy [24], the modified buddy strategy [26], the MPP method [28], and the free list strategy [25]. Of these approaches, only the MPP method and the free list strategy have been shown to perform optimally, since they provide perfect subcube

recognition. Additionally, the free-list strategy operates with lower overhead than the MPP method, and is therefore regarded as the best subcube allocation policy.

For hypercube machines, such as the nCUBE-2 and the nCUBE-3, the restriction of allocating subcubes causes low processor utilization. Although the hypercube is a powerful network topology [39], 2D and 3D meshes are more popular application topologies. For example, grid domain decomposition for solving partial differential equations is an application that can easily be implemented on 2D and 3D meshes. In addition, 2D and 3D meshes can be used more efficiently by allocating exactly the number of processors requested. For example, if the optimal number of processors for a task is 600, then the smallest subcube that can be allocated is 1024 processors, resulting in a waste of 424 processors, while a 2D mesh may allocate a 20 $\times$ 30 cluster.

Consider the 4-dimensional cube shown in Figure 3.1, in which one job is allocated a 2 $\times$ 5 mesh, and another job is allocated a 2 $\times$ 3 mesh. With a restriction to subcube allocation, both jobs cannot be simultaneously executed, even though the total number of processors, 16, is sufficient. Without the subcube restriction, both clusters may be allocated in the 4-cube, as shown in Figure 3.1. However, a closer look reveals that communication from node 0100 to node 1010 in the 2×5 cluster may cause link contention with communication between nodes 0110 and 0010 in the 2×3 cluster, if the popular deadlock-free E-cube routing is used [40].

If nothing is assumed about communication among processors within a cluster, then communication may occur between any two processors within any cluster. Thus, the potential for contention can result in *intercluster communication interference*, which is desirable to avoid. Many known processor allocation strategies have been developed to guarantee contention-free cluster allocation, such as the subcube allocation strategies for hyper-

cubes [23, 24, 25, 26, 27, 28], the 2D mesh allocation techniques [19, 20, 21, 22], and the one used in CM-5 [41] (fat tree topology).

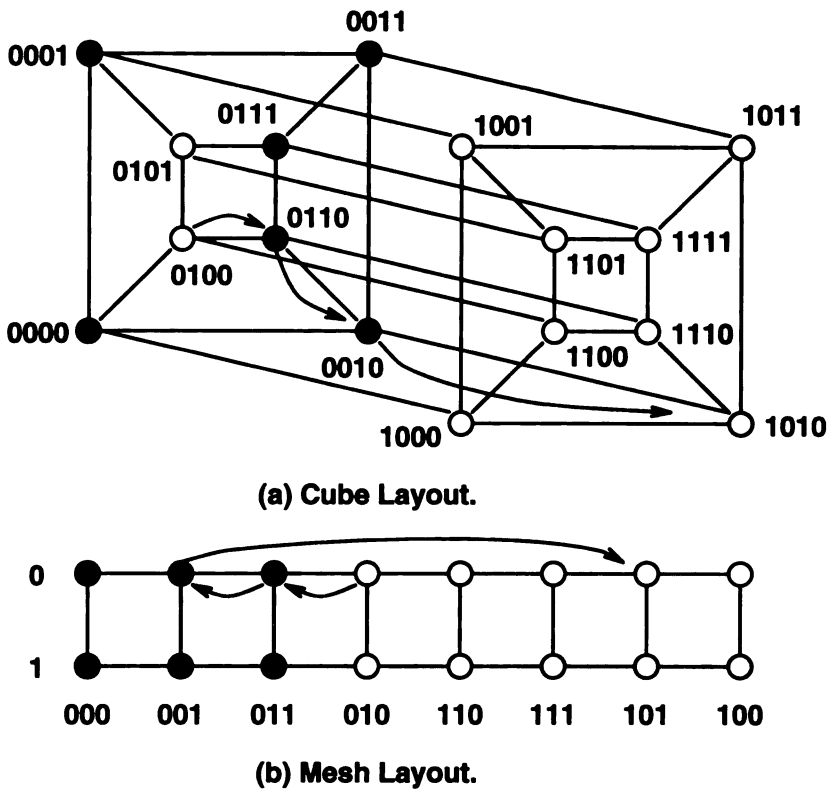

Figure 3.1: Communication interference in an embedded mesh in a subcube.

Non-cubic (NC) cluster allocation in a hypercube is the process of allocating x connected nodes, where $x \neq 2^k$. Kim et al [42, 25] performed preliminary investigations into the possibility of non-cubic cluster allocation. Their method requires that a non-cubic cluster be composed of adjacent cubes. For example, if a cluster request consists of 11 nodes, the free list allocation algorithm is used to find adjacent 3- and 2-cubes (or alternatively, a 3-cube, a 1-cube, and a 0-cube) to form a cluster of 11 processors. In their study of NC allocation, the effects of intercluster communication interference are not explicitly addressed.

It is well-known that the hypercube may embed a k-dimensional mesh with dilation 1 [43]. The *auxiliary free list method* uses dilation 1 embeddings of meshes to address the problems

of low processor utilization and intercluster communication interference. The allocation method is contention-free and uses existing deadlock-free E-cube routing. In addition, it improves performance over the free-list method by decreasing job turnaround time and increasing processor utilization. Our investigations address 2D mesh allocation. For applications not requiring a 2D mesh, the closest 2D mesh is assumed to be allocated.

3.2 Notation

Let Q_n denote an n-dimensional hypercube of 2^n nodes, in which the address of nodes and subcubes are represented by an n-bit string $\alpha = a_{n-1}a_{n-2}\ldots a_0$, $a_i \in \{0,1,*\} = \Sigma$, with bit a_i corresponding to dimension i and '$*$' representing the "don't care" symbol. Σ^k is the set of k-bit address strings whose elements are in Σ. Σ^n can be used to uniquely represent addresses in an n-cube, and operations can be defined on cube addresses as follows:

Definition: Hamming Distance: For two address strings $\alpha = a_{k-1}a_{k-2}\cdots a_0$ and $\beta = b_{k-1}b_{k-2}\cdots b_0$ in Σ^k for some integer k, the Hamming distance is defined as

$$H(\alpha,\beta) = \sum_{i=0}^{k-1} h(a_i,b_i),$$

where $h(a,b) = 1$ if $a,b \in \{0,1\}$ and $a \neq b$, and $h(a,b) = 0$ otherwise [42, 25].

Definition: Binary Reflected Gray Code (BRGC): Let G_n be the n-bit BRGC. G_n is defined recursively as follows:

$$G_1 = \{0,1\}, G_i = \{0G_{i-1}, 1G^*_{i-1}\}$$

For the above definition, G_i^* is the set obtained by reversing the order of G_i. For example, $G_2 = \{00, 01, 11, 10\}$ and $G_2^* = \{10, 11, 01, 00\}$, hence $G_3 = \{000, 001, 011, 010, 110, 111, 101, 100\}$. Furthermore, g_i will refer to the ith gray code in a sequence. For example, in G_3, $g_4 = 110$. A more general definition of gray codes is given in [24].

Definition: Adjacent and Complementary Cubes:

> Given two cubes identified by address strings $\alpha = \alpha_{n-1}\alpha_{n-2}\cdots\alpha_0$ and $\beta = \beta_{n-1}\beta_{n-2}\cdots\beta_0$, they are adjacent if $H(\alpha, \beta) = 1$. The complement of α at j is $\alpha_{\overline{j}} = \alpha_{n-1}\alpha_{n-2}\cdots\alpha_{j+1}\overline{\alpha_j}\alpha_{j-1}\cdots\alpha_0$, where $0 \leq j \leq n-1$ and $\alpha_j, \overline{\alpha_j} \in \{0, 1\}$.

In order to introduce a lemma relevant to our new allocation scheme, a new notation for binary addresses, as well as communication channels, is presented. A node address S is represented by $S = S_h s_c S_\ell$, where S_h is a binary string representing its high order bits (those bits that are higher-order than the communication channel), s_c is the one-bit string in the position of the communication channel, and S_ℓ is a binary string representing the low order bits. A communication channel is represented as $C = C_h \lambda C_\ell$, where $\lambda \in \{\uparrow, \downarrow\}$. Here, $\uparrow$ indicates a bit changing from 0 to 1 and $\downarrow$ indicates the opposite. Thus, $C = C_h \uparrow C_\ell$ indicates a communication channel in which the direction is $0 \rightarrow 1$. For example, consider the following two node addresses that share a communication channel: $N_1 = 0111010$, $N_2 = 0110010$. If communication is flowing from N_1 to N_2, then the channel is represented by $C = 011 \downarrow 010$, with $S_h = 011$ and $S_\ell = 010$ for C. For N_1, $s_c = 1$, and for N_2, $s_c = 0$.

The following lemma states important properties shared between source and destination nodes, as well as a communication channel used by messages sent from the source to the destination.

Lemma 1 *Let $S = S_h s_c S_\ell$ be a source address, and $D = D_h d_c D_\ell$ be a destination address, where s_c and d_c denote the bit used by communication channel C. Without loss of generality, let $C = C_h \uparrow C_\ell$. Then E-cube routing from S to D will use channel C if the following 3 conditions hold: (1) $s_c = 0$ and $d_c = 1$, (2) $S_h = C_h$, and (3) $D_\ell = C_\ell$.*

Proof: (1) Since C is $0 \rightarrow 1$, the channel bit on S must be 0, and the channel bit on D must be 1. (2) E-cube routing examines and (possibly) changes the lowest order bit first, followed by the second lowest order bit, and continues until the highest order bit is examined. Assume that I_j is the intermediate address produced after the jth step of the E-cube algorithm. Also assume that $S_h \neq C_h$ and C is an intermediate channel used by the E-cube routing algorithm. Before the first step, $I_0 = S$ ($I_0 = S_h s_c S_\ell$). If $|S_\ell| = k$, then after k steps of the E-cube algorithm, $I_k = S_h s_c D_\ell$. Since C is the channel used, this condition contradicts the assumption that $S_h \neq C_h$. Therefore, S_h must be the same as C_h. (3) $D_\ell = C_\ell$ is seen from the proof of item (2), in which after k steps of the E-cube algorithm, $I_k = S_h s_c D_\ell$. Therefore, $D_\ell = C_\ell$. □

The following lemma states properties about the addresses of the leftover cubes.

Lemma 2 *Let G_n be the n-bit BRGC, and let H_i be any subset of BRGC gray codes having 2^i elements ($0 \leq i \leq n$). If all gray codes in H_i have the same $n - i$ most significant bits, then the cube address for this set of nodes may be represented by replacing the i least significant bits of any gray code in H_i with '$*$'.*

Proof: The proof of this lemma is obtained by observing the n-bit BRGC (for any $n > 0$).

□

3.3 The AFL Cluster Allocation Method

The primary objective of the auxiliary free list is to supplement existing subcube allocation methods by partitioning the unused nodes into subcubes, which are then stored in the auxiliary free list. Leftover nodes are partitioned into subcubes so that clusters may be guaranteed to be contention-free. In our investigations, the auxiliary free list was used to supplement the free list algorithm [25]. However, the AFL may be used in conjunction with any of the well-known subcube allocation methods. Furthermore, since this study restricts itself to the allocation of 2D meshes in subcubes, a modified version of the free list algorithm, reflecting this restriction, is presented.

The free list algorithm for subcube allocation has been shown to be the most effective algorithm for recognizing available subcubes. It maintains a list of currently available subcubes for each subcube dimension. When an incoming request requires a subcube of dimension k, the free list algorithm can check, in $O(n)$ time (for a hypercube Q_n), whether a k-cube can be obtained. There are three main issues to consider with auxiliary free list allocation: (1) how to partition leftover nodes, (2) allocation using the auxiliary free list, and (3) deallocation using the auxiliary free list.

In the free list algorithm, subcubes exist in one of two states: *allocated* (in-use) or *free* (available for cluster allocation). Available subcubes are located by consulting a list of currently free subcubes, rather than by consulting a list of what is currently allocated. The AFL introduces a third state, in which a subcube is *leftover* from a 2D mesh embedding. Subcubes in this state are maintained in the auxiliary free list, which has the same format as the original free list.

To embed an $X \times Y$ mesh cluster, it is necessary to find an available k-cube ($k = \lceil \log_2 X \rceil + \lceil \log_2 Y \rceil$) for a dilation 1 embedding. This embedding leaves $2^k - XY$ leftover

nodes. The AFL allocation algorithm generates leftover subcubes that comprise these leftover nodes, placing them in the auxiliary free list. Each 2D mesh cluster that generates leftover subcubes is also associated with an ownership list, containing only the leftover subcubes that it generates. This ownership list is maintained so that the original k-cube may be quickly recovered, if possible, when the 2D mesh cluster is deallocated. That is, the cube manager can quickly check whether the cubes in its ownership list are allocated to determine whether they may be concatenated back into the original k-cube. The ownership list is explained further in Section 3.3.3.

3.3.1 Partitioning Leftover Subcubes

The partitioning algorithm (Figure 3.2) is the means by which the AFL algorithm extracts leftover subcubes from a 2D mesh allocation. For a dilation-1 2D mesh embedding, assume that $\ell = \lceil \log_2 X \rceil$ and $m = \lceil \log_2 Y \rceil$ ($k = \ell + m$). If G_i is used to denote the i-bit *binary-reflected grey-code* (BRGC), then the first X and Y codewords of G_ℓ and G_m, respectively, are used to address the embedded mesh in the X and Y dimensions. Let L_x and L_y be the set of unused codewords resulting from the embedding of the X and Y dimensions, respectively ($|L_x| = 2^\ell - X$ and $|L_y| = 2^m - Y$). In addition, let A_x and A_y be the set of codewords used for X and Y dimension embedding, respectively ($|A_x| = X$ and $|A_y| = Y$).

Figure 3.3 illustrates the regions of the subcube (in a mesh layout form) covered by the sets A_x, A_y, L_x, and L_y. In addition, it illustrates two overlapping regions resulting from an embedding. Region (1) represents the area defined by ($|L_x| \times |G_m|$), and region (2) represents the area defined by ($|L_y| \times |G_\ell|$). It is desirable for the algorithm to produce the largest possible leftover subcubes. Hence, there are two cases: (a) region (1) is larger than region (2); (b) region (2) is larger than region (1). The algorithm is essentially identical

for either case, the only difference being the sets of codewords being used in steps (1) and (2) of the partitioning algorithm. Thus, for clarity, Figure 3.2 presents the algorithm as handling only case (a).

Algorithm: *Partitioning.*

Input: a subcube of dimensionality k, an $X \times Y$ mesh request.

Output: A set of leftover subcubes.

Procedure:

(1) For all elements of L_x:
Compute x-component subcube addresses by changing low-order bits of subsets of L_x to $*$.
Replace all L_x addresses with subcube addresses computed from L_x.
Add the concatenation of each x-component with $*^m$ to the AFL.

(2) (a) For all elements of L_y:
Compute y-component subcube addresses by changing low-order bits of subsets of L_y to $*$.
Replace all L_y addresses with subcube addresses computed from L_y.

(b) For all elements of A_x:
Compute x-component subcube addresses by changing low-order bits of subsets of A_x to $*$.
Replace all A_x addresses with subcube addresses computed from A_x.

(c) For all elements of A_x and L_y:
Compute subcube addresses by concatenating members of L_y to members of A_x.
Add each computed address to the AFL.

Figure 3.2: The partitioning algorithm.

The partitioning algorithm takes advantage of the property given in Lemma 2, which enables the algorithm to easily compute cube addresses by using an address from some gray code subset and replacing the i least significant bits with '$*$'. For example, in the 4-bit BRGC, the sequence of 4 codes $\{1010, 1011, 1001, 1000\}$ has "10" as its two most significant bits. The cube address of this sequence is therefore $10**$.

The results of the partitioning algorithm are illustrated in Figure 3.4, in which a 5×5 mesh has been embedded in a 6-cube. The cube is pictured in a mesh-layout form. As the figure illustrates, the leftover subcubes generated by the partitioning algorithm are disjoint,

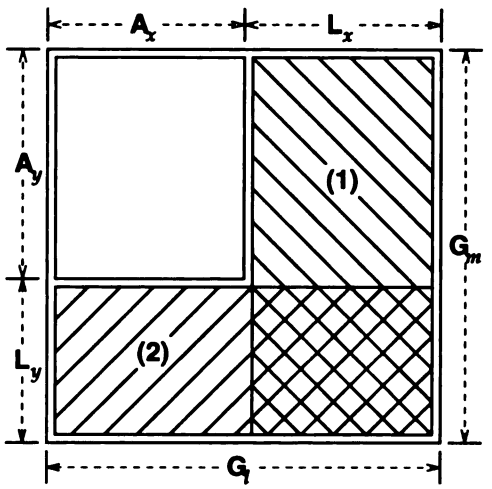

Figure 3.3: Leftover regions from 2D mesh embeddings.

and many pairs of cubes are adjacent. To keep them in this form, they are kept separate from the normal free list, as the free list deallocation algorithm would rearrange this format.

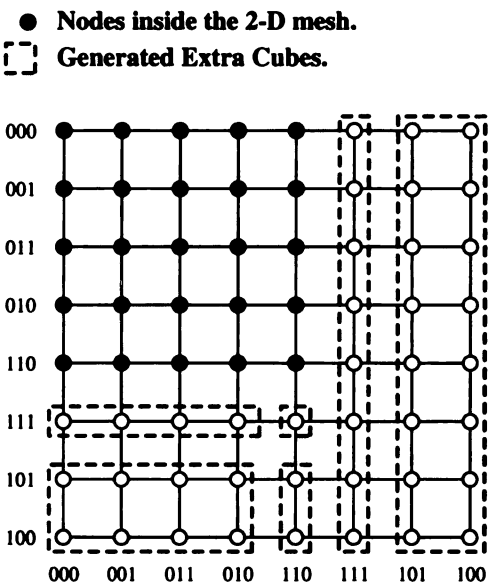

Figure 3.4: Allocation of a 5 × 5 mesh.

Intuitively, it can be seen that using the partitioning algorithm can increase system utilization: the previous example of a 20 × 30 mesh leaves 424 extra nodes that would not be allocated by the original free list algorithm. The partitioning algorithm will generate an 8-cube, a 7-cube, a 5-cube, and a 3-cube as available leftover cubes.

Theorem 1 states that the auxiliary free list algorithm will allow multiple 2D meshes to be allocated within one subcube without communication interference if E-cube routing is used.

Theorem 1 *The leftover cubes created by the 2D mesh partitioning algorithm suffer no communication interference from the 2D mesh cluster, and the 2D mesh cluster suffers no communication interference from the leftover cubes.*

Proof:

First, it is necessary to show that communication within a subcube never leaves that subcube. Assume, without loss of generality, that a k-cube Q_k exists having the address $\alpha_{n-1}\alpha_{n-2}\cdots\alpha_{k+1}\alpha_k X_{k-1}\cdots X_0$, where $\alpha_i \in \{0,1\}$ and $X_j = *$. In addition, assume that S and D are source and destination nodes within Q_k, respectively, and that the E-cube routing algorithm uses a channel, C, outside of Q_k, when routing from S to D. By Lemma 1, $S_h = C_h$. Because it exists within Q_k, the highest-order $n-k$ bits of S are $\alpha_{n-1}\alpha_{n-2}\cdots\alpha_{k+1}\alpha_k$. Since C is not contained within Q_k, it must be true that C differs from S in at least one of these bits. Assume that this bit is bit m ($k \leq m \leq n-1$). According to Lemma 1, $D_\ell = C_\ell$. This condition implies that the highest order $n-k$ bits of D differ in at least one position from those of S. However, since D also exists within Q_k, it must be true that the highest order $n-k$ bits of S and D are identical. Therefore, C cannot exist outside of Q_k if E-cube routing is used. Therefore, the 2D mesh cluster suffers no communication interference from the leftover cubes.

To show that the leftover cubes suffer no communication interference from the 2D mesh cluster, it is only necessary to show that, for any (S, D) pair in the 2D mesh cluster, the E-cube routing algorithm will not use any channels used by nodes within any of the leftover cubes.

Assume that (S_1, D_1) are two nodes inside the 2D mesh cluster and that (S_2, D_2) are two nodes inside a leftover cube generated by the partitioning algorithm. Also assume that C is a channel used by communication from S_1 to D_1, as well as by communication from S_2 to D_2. Then by condition 2 of Lemma 1, it must be true that $S_{1_h} = S_{2_h}$, and by condition 3 of Lemma 1, it must be true that $D_{1_\ell} = D_{2_\ell}$.

Assume that, for a 2D mesh embedding, an address A may be broken into an X component and a Y component, where $A = A_X \bullet A_Y$. For an $X \times Y$ mesh embedding, let $k = \ell + m$, where $\ell = \lceil \log_2 X \rceil$ and $m = \lceil \log_2 Y \rceil$. A_X will be ℓ bits long, and A_Y will be m bits long. Now assume that d_i is an element of either S_x or S_δ used to generate the X component of some leftover cube B, and let $d_i = 2^i$. Then the X component of B will be $B_X = \beta_{\ell-1}\beta_{\ell-2}\cdots\beta_{i+1}\beta_i *_{i-1}\cdots *_0$, where $\beta_p \in \{0,1\}$ and $*_p = *$ $(0 \leq p \leq \ell - 1)$ (recall that '$*$' represents the don't care bit). Similarly, if d_j is an element of S_γ used to generate the Y component of the same leftover cube B, and $d_j = 2^j$, then the Y component of B will be $B_Y = \theta_{m-1}\theta_{m-2}\cdots\theta_{j+1}\theta_j *_{j-1}\cdots *_0$, where $\theta_p \in \{0,1\}$ and $*_p = *$ $(0 \leq p \leq m - 1)$.

The don't care bits in the X and Y components of B define the communication channels within the leftover subcube. Therefore, there are two cases to consider: (1) C is defined by one of the "don't care" bits in the X component of the leftover cube B, and (2) C is defined by one of the "don't care" bits in the Y component of the leftover cube B.

(Case 1): If C is defined by the X component of B, then the $\ell - i$ most significant bits of C define a unique prefix to the address of C. If node S_1 exists within the 2D mesh, it cannot have this prefix to the X component of its address, since all 2^i nodes having this prefix are, by definition, part of a generated leftover cube. Therefore, since $S_{1_h} \neq S_{2_h}$, S_1 cannot use communication channel C in any message it sends.

(Case 2): If C is defined by the Y component of B, then the ℓ most significant bits of C define a unique prefix to the address of C. As in case 1, if node S_1 exists within the 2D mesh, it cannot have this prefix to the X component of its address, since all 2^i nodes having the same most significant ℓ bits as C are by definition part of a generated leftover cube. Therefore, since $S_{1_h} \neq S_{2_h}$, S_1 cannot use communication channel C in any message it sends. □

3.3.2 Allocation of 2D Meshes

The AFL method is intended to supplement any existing subcube allocation algorithm. Therefore, cluster allocation using the AFL algorithm is a simple decision procedure, in which the AFL is checked first for an incoming cluster request. If it cannot satisfy an incoming request, then control is passed to the regular subcube allocation algorithm (modified to make use of the partitioning algorithm). Again, note that a subcube contained in the AFL may also be partitioned to produce a set of leftover subcubes. Assume that the $X \times Y$ request requires a k-cube to be satisfied ($k = \lceil \log_2 X \rceil + \lceil \log_2 Y \rceil$).

The AFL allocation algorithm checks dimensions k through n of the AFL for the existence of a subcube to satisfy the $X \times Y$ mesh request. In the event that a k-cube is not available, but a higher dimensioned subcube is available, the algorithm applies a process identical to the procedure performed by the free list algorithm, in which a higher-dimensioned subcube is iteratively partitioned into lower-dimensioned subcubes until the required k-cube is obtained and used. Figure 3.5 illustrates the AFL allocation algorithm.

When cluster requests arrive, the AFL allocation algorithm is used to first consult the AFL. If no suitable subcube can be found in the auxiliary free list, the modified free

Algorithm: *Auxiliary Free List Allocation.*

Input: An $X \times Y$ mesh request.

Output: Allocated $X \times Y$ mesh (control is passed to the Modified FL algorithm if no k-cube can be found).

Procedure:

(1) If a subcube of dimension k, Q_k, exists in the AFL, then
  (a) Partition the subcube using the Partitioning algorithm
  (b) Mark the $X \times Y$ region of Q_k as allocated, and exit.
  Else, perform step (2).

(2) Find the lowest dimension i, $k+1 \leq i \leq n$, such that a subcube Q_i exists in the AFL.

(3) If no such Q_i exists, then perform the normal subcube allocation algorithm
  (here, the Modified Free List Algorithm).
  Else, decompose the obtained subcube Q_i iteratively as follows:
  (a) While $i > k$, do:
    Obtain a subcube Q_i from the ith dimension AFL.
    Split Q_i into two subcubes of dimension $i-1$, adding them to the AFL of dimension i.
  (b) Perform step (1).

Figure 3.5: The AFL Allocation algorithm.

list algorithm (Figure 3.6) is called. The primary difference between the modified free list algorithm and the original free list algorithm is that the modified free list algorithm, instead of simply allocating a subcube, performs the partitioning algorithm to allocate 2D mesh clusters.

3.3.3 Deallocation of 2D Mesh Requests

The deallocation of a 2D mesh request involves three major steps. First, the region embedding the mesh is partitioned into a set of subcubes in a manner similar to the partitioning algorithm (Figure 3.2). Second, the resulting list of subcubes is *coalesced* with the AFL subcubes produced when the 2D request was allocated. Third, any higher-dimensional subcubes resulting from this coalescing process are then "deallocated" by adding them to the

Algorithm: *Modified Free List Allocation.*

Input: An $X \times Y$ mesh request.

Output: Sends a subcube to the partitioning algorithm or returns nothing if the requested cluster cannot be allocated.

Procedure:

(1) If a subcube of dimension k, Q_k, exists in the FL, then
  (a) Partition the subcube using the Partitioning algorithm
  (b) Mark the $X \times Y$ region of Q_k as allocated, and exit.
  Else, perform step (2).
(2) Find the lowest dimension i, $k+1 \leq i \leq n$, such that a subcube Q_i exists in the FL.
(3) If no such Q_i exists, then Exit (the subcube could not be allocated).
  Else, decompose the obtained subcube Q_i iteratively as follows:
  (a) While $i > k$, do:
    Obtain a subcube Q_i from the ith dimension FL.
    Split Q_i into two subcubes of dimension $i-1$, adding them to the FL of dimension i.
  (b) Perform step (1).

Figure 3.6: The modified free list allocation algorithm.

list of subcubes to deallocate and, under certain conditions, performing the normal subcube deallocation procedure.

The partitioning of the embedded mesh is performed by the *Cluster Partitioning Algorithm* (Figure 3.7). This algorithm is similar to the partitioning algorithm (Figure 3.2), except that it partitions the 2D mesh cluster, instead of the region leftover from the 2D mesh cluster allocation.

Leftover subcubes originally created by the partitioning algorithm are linked together by an ownership list that allows the deallocation procedure to quickly check (using a linked-list traversal) whether these leftover cubes are free or allocated. In the event that all of the leftover cubes are currently free when the 2D mesh is deallocated, the Coalesce algorithm (Figure 3.8) will restore the original k-cube. In this case, the normal subcube deallocation algorithm is performed on the k-cube. Otherwise, coalescing will produce several disjoint subcubes, which may be added to the database of free subcube regions for the normal

subcube allocation algorithm. In any event, the ownership list is destroyed when the 2D mesh is deallocated using the Auxiliary Free List Deallocation algorithm (Figure 3.9). Those owned subcubes that are still allocated will be deallocated using the AFL Deallocation algorithm when they become free. This process will eventually restore the original k-cube when all of the leftover subcubes become free.

Algorithm: *Cluster Partitioning.*

Input: An $X \times Y$ cluster to be deallocated.

Output: A set of subcubes composing the $X \times Y$ cluster.

Procedure:

(1) (a) For all elements of A_x:
Compute x-component subcube addresses by changing low-order bits of subsets of A_x to $*$.
Replace all A_x addresses with subcube addresses computed from A_x.

(b) For all elements of A_y:
Compute y-component subcube addresses by changing low-order bits of subsets of A_y to $*$.
Replace all A_y addresses with subcube addresses computed from A_y.

(c) For all elements of A_x and A_y:
Compute subcube addresses by concatenating members of A_y to members of A_x.
Add each computed address to the AFL.

Figure 3.7: The cluster partitioning algorithm.

3.4 Analysis of the AFL Cluster Allocation Method

This section presents the analysis of the algorithms associated with 2-D mesh allocation and deallocation. A complete analysis of the complexity of the original Free List algorithms is given by Kim et al [25]. For allocation and deallocation in an n-cube, assume that the original cluster request required a k-dimensional subcube. Both the allocation and deallocation of 2D mesh clusters do not add noticeable overhead to the original free list algorithm. AFL allocation adds $O(k^2)$ complexity to the Free List algorithm, which is

Algorithm: *Coalesce.*

Input: A list of subcubes, T, of dimension 0 to $k-1$.

Output: A list of subcubes resulting from coalescing complementary cubes.

Procedure:

(1) For $i = 0$ to $k-1$ **do**
Let c be the number of subcubes of dimension i. If $c \geq 2$, then
For all pairs of previously un-coalesced subcubes of dimension i:
If the two subcubes are complementary, then
(a) Combine them into a subcube of dimension $i+1$
(b) Remove them from T.
(c) Add the new $(i+1)$-subcube to T.

Figure 3.8: The coalesce algorithm.

Algorithm: *Auxiliary Free List Deallocation.*

Input: A deallocating $X \times Y$ cluster C, with ownership list O.

Output: Adds subcube(s) to the AFL or the FL.

Procedure:

(1) Perform the Cluster Partitioning algorithm on C,
placing the subcubes obtained in a temporary list T.
(2) For all subcubes Q_i in O:
(a) Remove Q_i from O ("disown" it).
(b) If Q_i is not allocated, then remove it from the AFL and add it to T.
(3) Perform the Coalesce algorithm on T.
(4) If T contains one subcube of dimension k, then
add that subcube to the FL, and
perform the Free List deallocation algorithm;
Else (T contains multiple subcubes), add all subcubes in T to the FL.

Figure 3.9: The auxiliary free list deallocation algorithm.

$O(n)$. More importantly, AFL deallocation adds no more than $O(k^3)$ complexity to the Free List deallocation algorithm, which is normally at least $O(n^3)$. The existence of the AFL deallocation algorithm makes it often unnecessary to perform the Free List deallocation algorithm, resulting in a greater time savings. Therefore, the methods introduced in this paper do not introduce significant time complexity to the free list algorithms. A more complete analysis of the individual algorithms involved in the AFL method is presented in the following subsections.

3.4.1 Analysis of the Partitioning Algorithm

In step (1) of the partitioning algorithm, the number of elements of L_x is $O(\ell)$ (recall that $\ell = \lceil \log_2 X \rceil$). Each of the three substeps of step (1) takes constant time for each element of L_x. Thus, step (1) is $O(\ell)$. Similarly, steps (2)(a) and (2)(b) are $O(m)$ and $O(\ell)$, respectively. Because ℓ and m can both range from 0 to k, any operation that is $O(\ell)$ or $O(m)$ can be assumed to be $O(k)$. Step (2)(c) requires a two loops to perform operations on two sets, each of which is $O(k)$. Therefore, this step is $O(k^2)$, and the overall complexity of the partitioning algorithm is $O(k^2)$. Practically, ℓ and m are normally much less than k, meaning that the complexity of $O(k^2)$ for this algorithm is an overestimate.

3.4.2 Analysis of the AFL Allocation Algorithm

The auxiliary free list allocation algorithm is almost identical to the Free List algorithm for subcube allocation. The two primary differences are that it (1) examines the AFL and not the FL, and (2) it uses the partitioning algorithm when a 2-D mesh request requires less than a complete subcube. If the complexity due to the partitioning algorithm is ignored, the

complexity of the auxiliary free list algorithm is $O(n)$ (see the analysis of the FL algorithm in [25]).

3.4.3 Analysis of the Modified Free List Algorithm

The only difference between the modified free list algorithm and the original free list algorithm is that the modified algorithm calls the partitioning algorithm when a suitable k-cube is obtained. In [42, 25], the authors showed that the complexity of the free list algorithm is $O(n)$, where n is the dimension of the hypercube multiprocessor. Thus, if the overhead of the partitioning algorithm is not considered, the Modified Free List algorithm is still $O(n)$.

3.4.4 Analysis of the Cluster Partitioning Algorithm

Step 1(a) of the cluster partitioning algorithm (Figure 3.7) performs a loop on all elements of the set A_x, which may have up to k elements. Each sub step of this loop requires a constant time to compute. Therefore, this step is $O(k)$. Step 1(b) is $O(k)$ for the same reason. Since step 1(c) requires a doubly-nested loop on two sets having up to k elements, it is $O(k^2)$, implying that the entire algorithm is $O(k^2)$.

3.4.5 Analysis of the Coalesce Algorithm

This algorithm requires a triple-nested loop. The outer loop considers k dimensions of subcubes. The inner two loops consider all possible pairings of subcubes of some dimension i, where i is the dimension under consideration by the outer loop. It is possible for any particular i dimension to contain k subcubes. Therefore, the inner two loops may require the selection of up to k^2 pairs of subcubes for any given dimension. Each substep ((a) to (c)) requires a constant time operation. Therefore, the entire algorithm can be considered to be

$O(k^3)$. In practice, however, since there are no more than k^2 total cubes in the temporary list, and often many fewer than this, this algorithm does not take significant time to run.

3.4.6 Analysis of the AFL Deallocation Algorithm

The auxiliary free list deallocation algorithm is a decision procedure that determines whether to perform an auxiliary free list deallocation, a free list deallocation, or a 2-D mesh deallocation, which uses the coalesce algorithm. Steps 1 and 3 are simply calls to other algorithms. Step 2 requires a linked-list traversal of the ownership list. Each substep is a constant time operation, and the ownership list itself may contain up to k^2 elements, because the partitioning algorithm creates up to k^2 leftover subcubes. Step (4) takes either constant time (the If condition) or up to k^2 steps (the Else condition). Therefore, the complexity of the AFL deallocation algorithm is $O(k^2)$, if the cluster partitioning algorithm, the coalesce algorithm, and the free list deallocation algorithm are ignored.

It is important to note that both the allocation and deallocation of 2-D mesh clusters do not add significant overhead to the free list algorithm. The auxiliary free list allocation algorithm adds $O(k^2)$ complexity to the free list allocation algorithm, which is $O(n)$. However, more importantly, the auxiliary free list deallocation algorithm adds no more than $O(k^3)$ complexity to the free list deallocation algorithm, which is normally at least $O(n^3)$. The existence of the auxiliary free list deallocation algorithm also makes it often unnecessary to perform the free list deallocation algorithm, resulting in a greater time savings over the course of allocating/deallocating many requests.

A real-world example from our simulations provides insight into the actual overhead incurred from using the auxiliary free list method. Simulations were performed on Sun

SPARCstation 10 computers. On a SPARC 10, a typical job can be allocated and deallocated in approximately 800 microseconds.

Chapter 4

Experimental Results for the AFL Cluster Allocation Method

Extensive simulations have been conducted to compare the performance of the FL method alone with the FL method augmented with the AFL method for allocating user jobs. In our experiments, 2D mesh requests were generated and allocated using the free list and auxiliary free list algorithms on a 10-dimension hypercube (1024 processors). The X and Y dimensions of incoming requests were varied under numerous distributions. Simulation results depend on the workload distribution, including such factors as the mesh size, the mesh geometry (rectangular or square), and the job's service time. With this in mind, many simulations were run, including: square meshes, in which the X dimension was randomly generated on [1,16] and [1,32] uniform distributions; rectangular meshes, in which the X and Y dimensions were randomly generated on [1,16] and [1,32] random distributions; and numerous *interval distributions* [21] for rectangular meshes.

The interval distributions were used to simulate bipartite distributions, in which there are many small requests coupled with many very large requests. Interval distributions are

based on uniform distributions, except that different ranges of values are generated with specific probabilities. The sum of all interval probabilities must equal 1.0. For example, An interval distribution might be $P_{[1,8]} = 0.3$, $P_{[9,24]} = 0.5$, and $P_{[25,32]} = 0.2$, indicating that the probability of a value being generated over the interval [1,8] is 0.3, the probability of a value being generated over the interval [9,24] is 0.5, and the probability of a value being generated over the interval [25,32] is 0.2.

Table 4.1 lists the sets of interval distributions that were simulated for rectangular mesh requests. Here, "small dimension" refers to the interval over which small mesh request dimensions are generated, and "large dimension" refers to the large mesh request dimension intervals. Each bipartite distribution was run with P(small) = A, P(large) = $1 - A$, for $A \in \{0.10, 0.30, 0.50, 0.70, 0.90\}$ (P(small) is the probability for small mesh dimensions, and P(large) is the probability for large mesh dimensions).

	small dimension in [1,8]	**small dimension in [1,16]**
large dimension in:	[9,32]	
	[17,32]	[17,32]
	[20,32]	[20,32]
	[25,32]	[25,32]
	[28,32]	[28,32]
	[30,32]	[30,32]

Table 4.1: Interval distributions simulated.

In these experiments, cluster requests are processed according to a first-come-first-serve (FCFS) queueing strategy, and the overhead of the cluster allocation and deallocation algorithms is ignored. The job interarrival time and service (execution) times are both assumed to have exponential distributions. Service time is generated with a mean of 4.0 seconds, and interarrival time is varied with respect to the service time. Results presented are nor-

malized with respect to the system load (service time / arrival time). For each separate simulation run, all random number streams are initialized with the same unique seed value, to guarantee that each allocation method receives the same input values. The interarrival time, service time, X dimension, and Y dimension, as well as a probability generator used to generate values in the interval distributions, are all independent. The results presented are generated with a 95% confidence interval on the *job turnaround time* (JTT) measure.

The results from an experiment using a constant workload illustrates the potential performance increase of the AFL method over the FL method. In this experiment, 50% of mesh requests were 4×6 and the other 50% were 4×2 meshes. The workload was generated by alternating the 4×6 and 4×2 requests, with the first request being a 4×6 mesh (that is, requests are generated in the sequence $4 \times 6, 4 \times 2, 4 \times 6, 4 \times 2, \cdots$). With this workload, the AFL method is able to allocate a 4×6 and a 4×2 mesh request in one 5-cube, because each 4×6 allocation generates a leftover 3-cube. The FL method, however, requires an entire 5-cube for each 4×6 mesh request, followed by a 3-cube for each 4×2 request. When the first 4×2 request is processed, the smallest available free-list subcube is a 5-cube, which is decomposed into a 4-cube and two 3-cubes. After allocation, the remaining 3 and 4-cubes cannot be used to allocate the incoming 4×6 request. Under very heavy system loads, this condition occurs after every 4×2 mesh allocation, effectively lowering the overall system utilization by 50%.

Figure 4.1 confirms the performance advantage of the AFL method over the FL method when applied to this workload. Here, the AFL method obtained 100% utilization, while the FL method obtained 50% utilization. In addition, the point at which JTT started to increase exponentially was 4.0 for the AFL method, while it was 2.0 for the FL method (the AFL method performed twice as well as the FL method, as expected). Furthermore,

this experiment helped to verify the correctness of the simulator used in the experiments presented here.

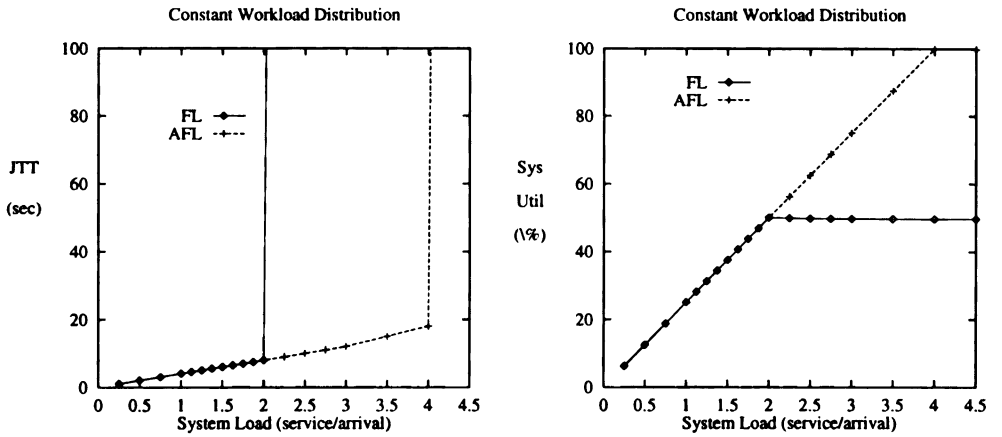

Figure 4.1: JTT and system utilization vs. system load for constant workload.

4.1 Square and Rectangular Requests With Uniform Distributions.

Figures 4.2 and 4.3 show the job turnaround time (JTT) versus the system load, as well as the system utilization versus the system load, for square and rectangular mesh loads in which mesh dimensions were generated in [1,16] uniform distributions. The performance of both methods is approximately the same. This is due to the fact that there are no mesh requests larger than 16 $\times$ 16. Any leftover subcubes generated by the AFL's partitioning algorithm are consequently of dimension at most 6, which results in fewer opportunities for the AFL method to exploit the existing leftover subcubes.

Figures 4.4 and 4.5 give the experimental results for square and rectangular mesh loads in which mesh dimensions were generated in [1,32] uniform distributions. In this case, the AFL method gives slightly better average utilization and JTT values than the FL method alone. Since there is a greater variation in mesh sizes being generated (from 1 $\times$ 1 to

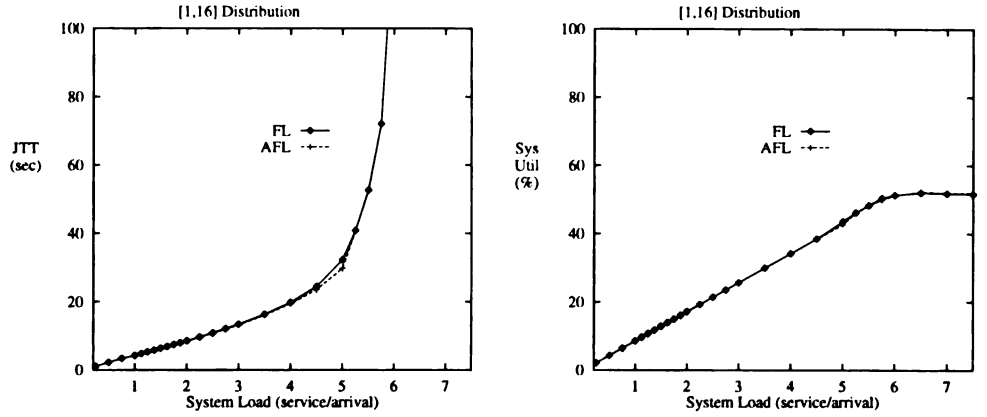

Figure 4.2: JTT and system utilization vs. system load for square mesh requests in a $[1, 16]$ uniform distribution.

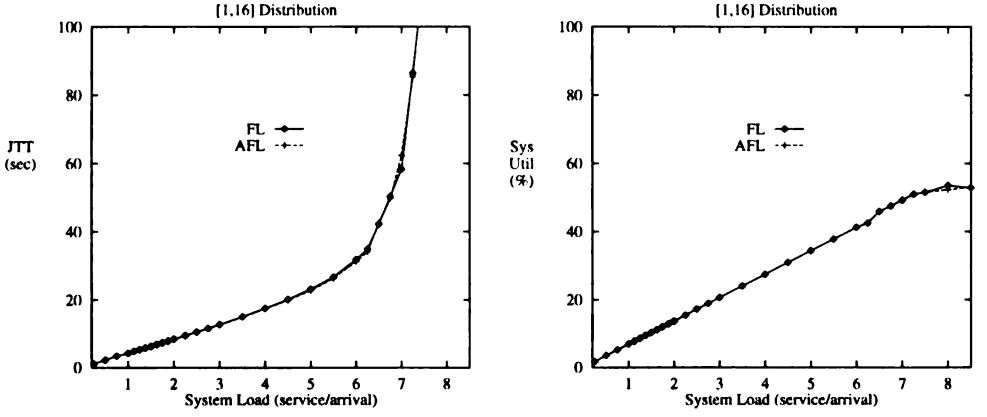

Figure 4.3: JTT and system utilization vs. system load for rectangular mesh requests in a $[1, 16]$ uniform distribution.

32 × 32), there are more instances in which a large mesh request generates many leftover subcubes, allowing the AFL method to satisfy a larger percentage of incoming requests, resulting in the increased performance shown.

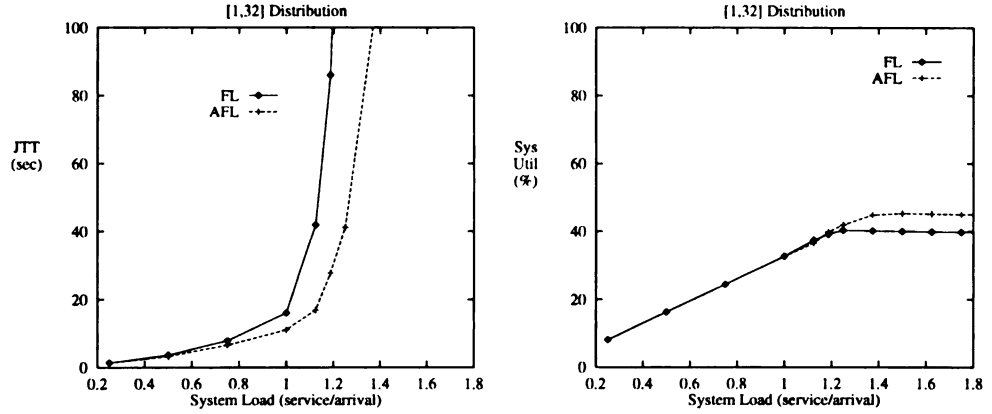

Figure 4.4: JTT and system utilization vs. system load for square mesh requests in a $[1, 32]$ uniform distribution.

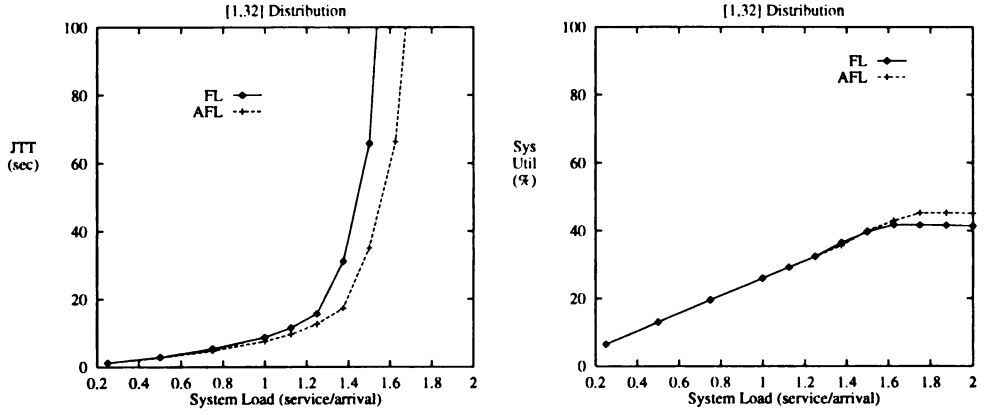

Figure 4.5: JTT and system utilization vs. system load for rectangular mesh requests in a $[1, 32]$ uniform distribution.

4.2 Interval Distribution, Small Dimension in $[1, 8]$.

Figures 4.6 and 4.7 present the JTT and system utilization versus the system load for meshes generated using bipartite interval distributions in which the small mesh dimension is in the [1,8] uniform interval, and the large mesh dimension is in the [9,32] uniform interval. The interval distributions were assigned as follows: $P_{[1,8]} = A$ and $P_{[9,32]} = 1 - A$, for $A \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

We observe that as the percentage, A, of small jobs is increased, the overall JTT improves, yet the system utilization also decreases. This is due to the greater number of small mesh requests being processed as A increases. It results in a larger percentage of small leftover subcubes that cannot be used to allocate any incoming requests. This trend is also observable for all other results in this section (see Figures 4.8 through 4.17).

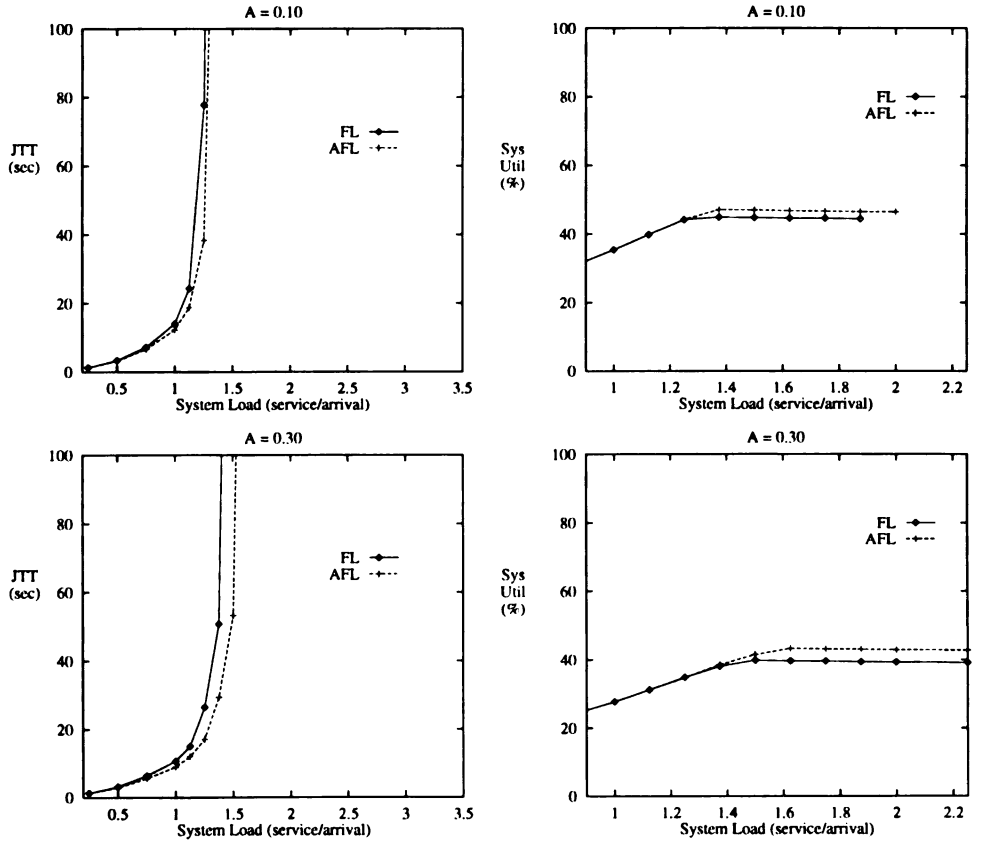

Figure 4.6: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[9,32]} = 1 - A$ interval distributions.

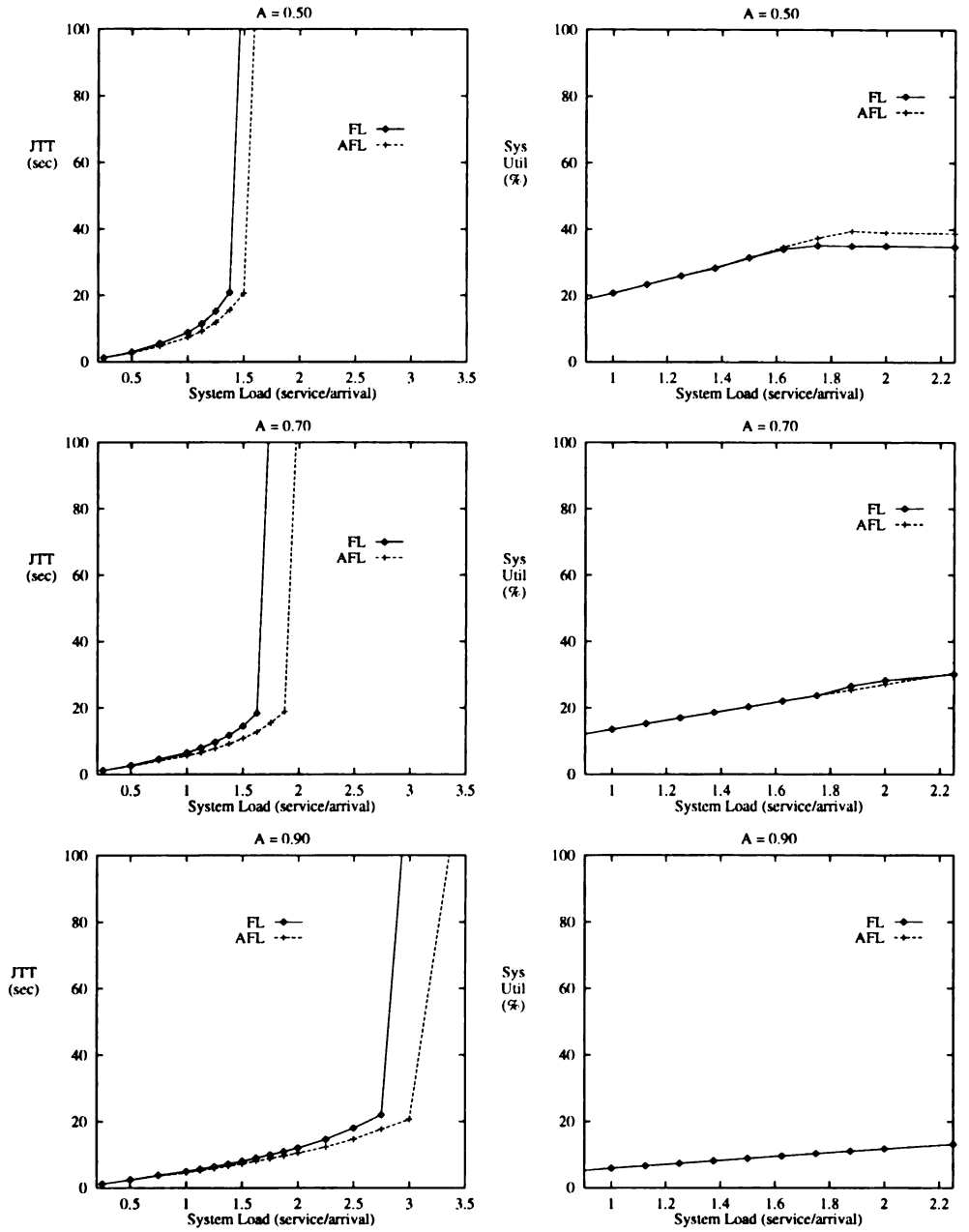

Figure 4.7: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[9,32]} = 1 - A$ interval distributions.

Figures 4.8 and 4.9 present the JTT and system utilization versus the system load for meshes generated using bipartite interval distributions in which the small mesh dimension is in the [1,8] uniform interval, and the large mesh dimension is in the [17,32] uniform interval. The interval distributions were assigned as follows: $P_{[1,8]} = A$ and $P_{[17,32]} = 1 - A$, for $A \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

This set of experiments illustrates the effect of removing some of the "middle" range of mesh sizes in the experiments. As the lower limit on the upper interval increases, the overall system utilization and JTT improves (again, the effect can be seen by observing all figures in this Section). This effect is due to the fact that, from experiment to experiment, a greater percentage of leftover subcubes are being created by larger and larger generated meshes, which results in a greater number of the small mesh requests being satisfied by leftover subcubes.

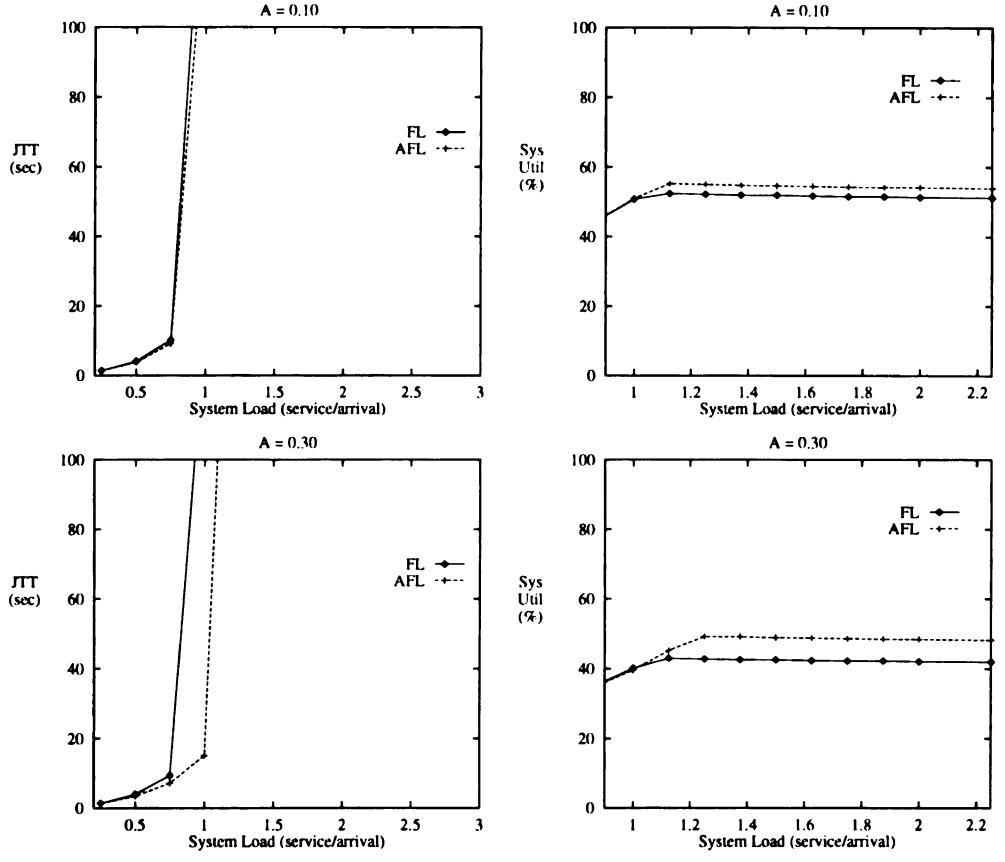

Figure 4.8: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[17,32]} = 1 - A$ interval distributions.

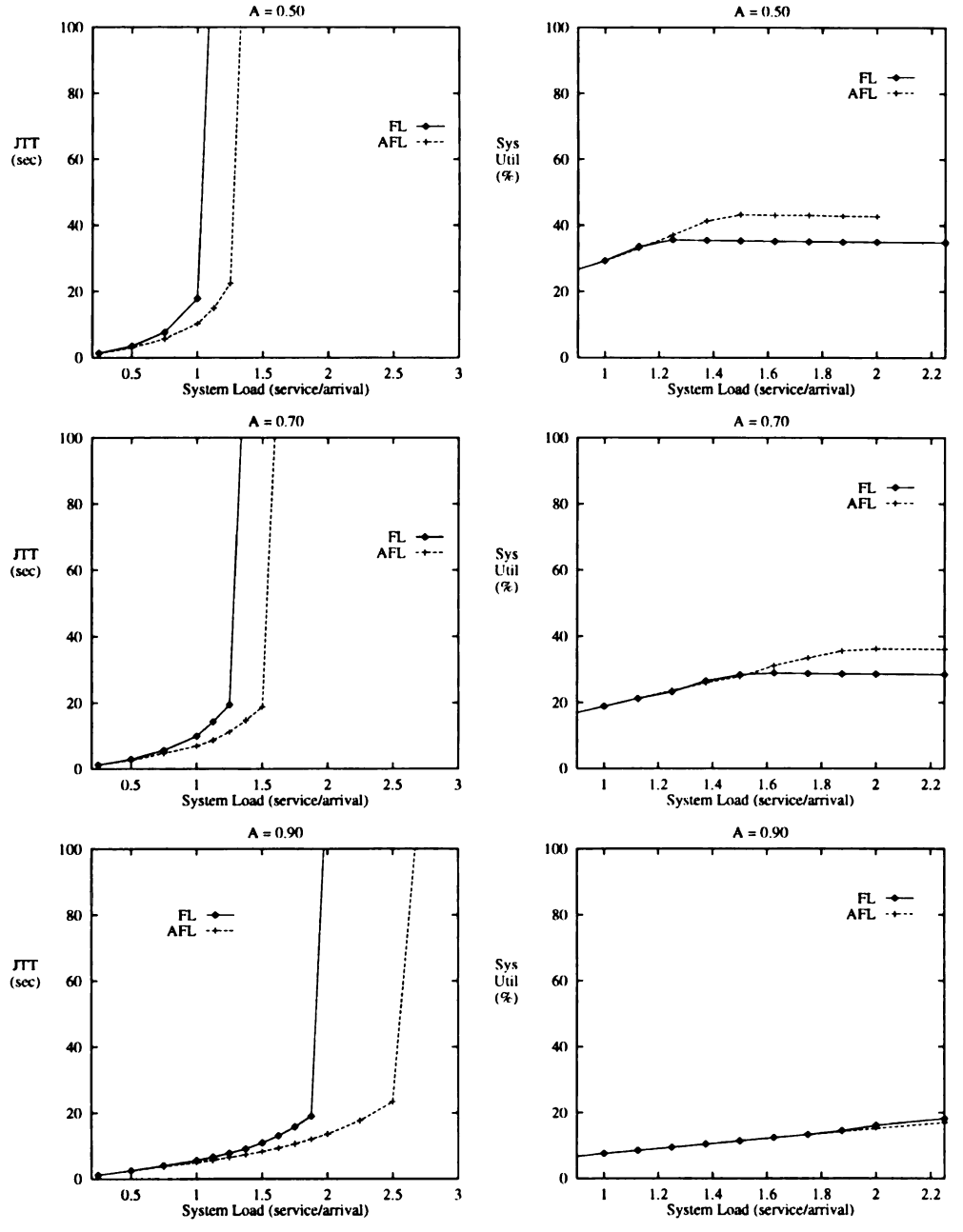

Figure 4.9: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[17,32]} = 1 - A$ interval distributions.

Figures 4.10 and 4.11 present the JTT and system utilization versus the system load for meshes generated using bipartite interval distributions in which the small mesh dimension is in the [1,8] uniform interval, and the large mesh dimension is in the [20,32] uniform interval. The interval distributions were assigned as follows: $P_{[1,8]} = A$ and $P_{[20,32]} = 1 - A$, for $A \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

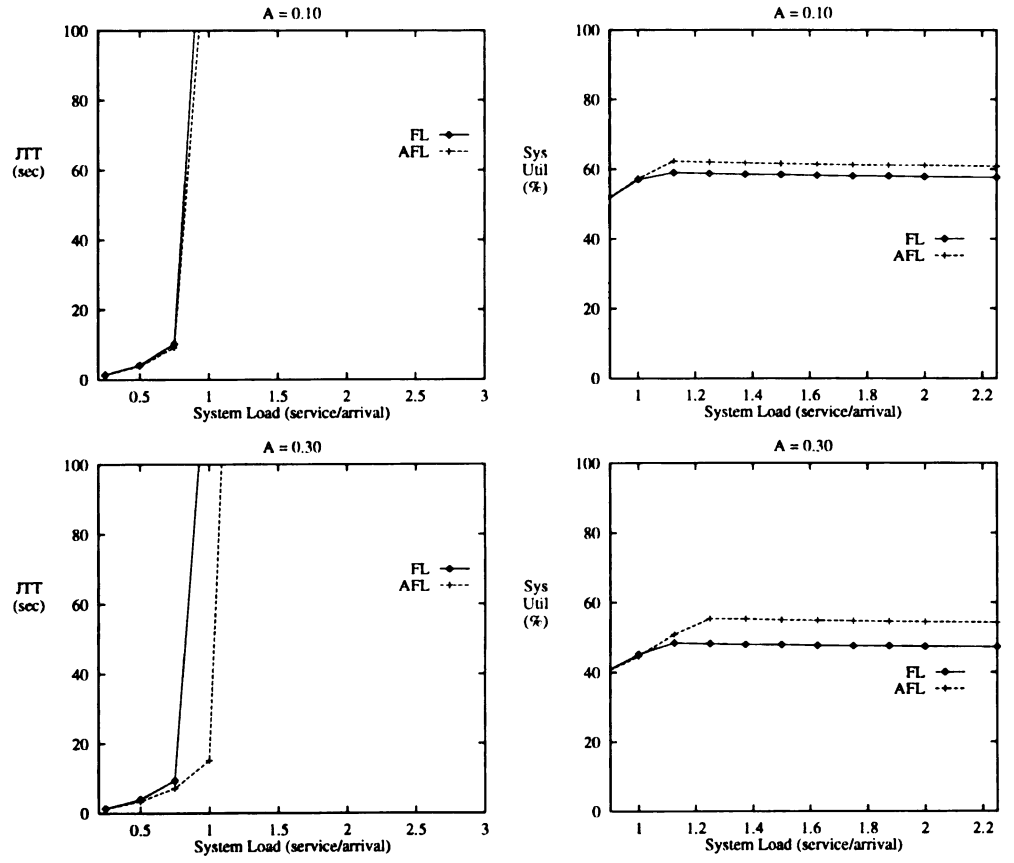

Figure 4.10: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[20,32]} = 1 - A$ interval distributions.

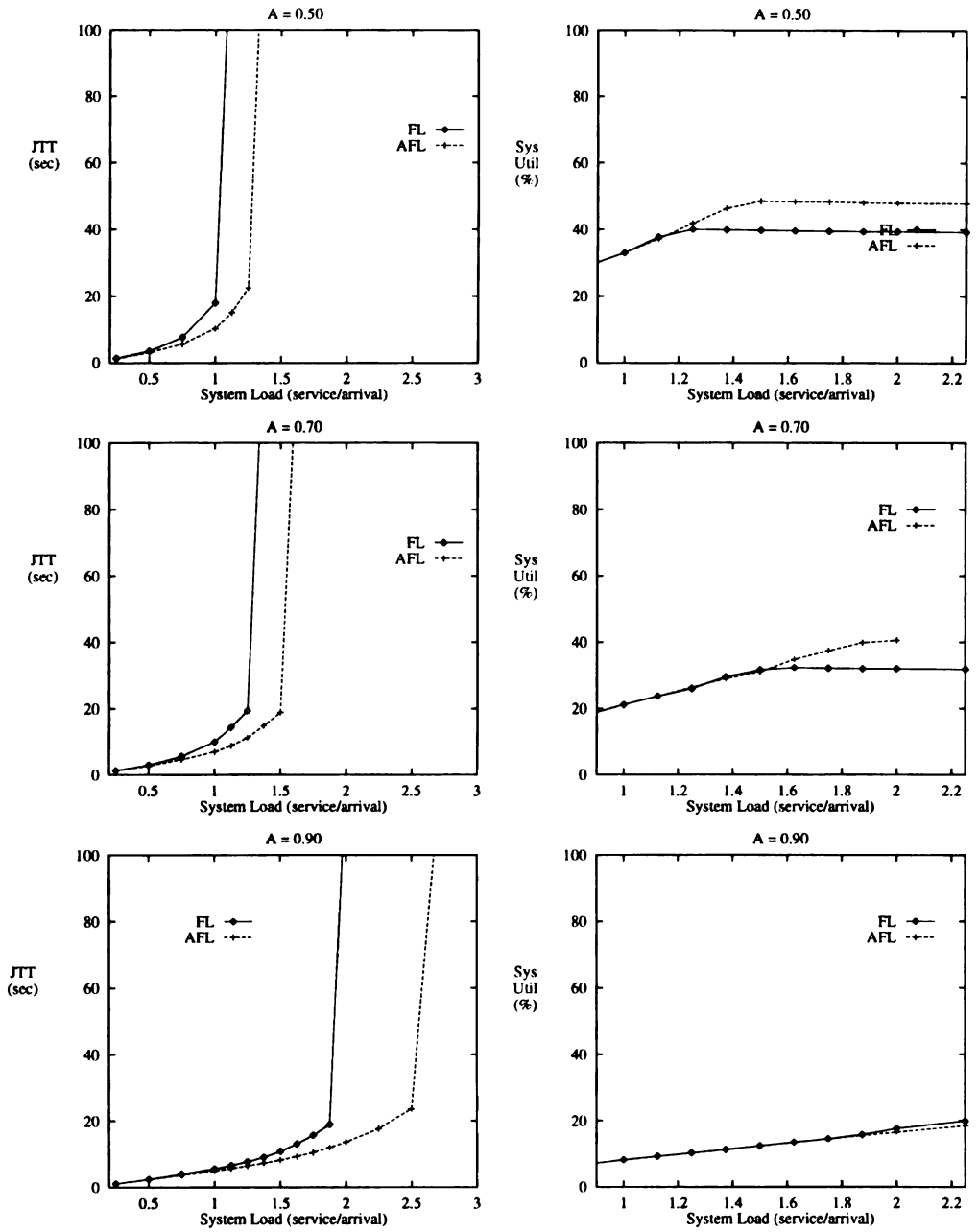

Figure 4.11: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[20,32]} = 1 - A$ interval distributions.

Figures 4.12 and 4.13 present the JTT and system utilization versus the system load for meshes generated using bipartite interval distributions in which the small mesh dimension is in the [1,8] uniform interval, and the large mesh dimension is in the [25,32] uniform interval. The interval distributions were assigned as follows: $P_{[1,8]} = A$ and $P_{[25,32]} = 1 - A$, for $A \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

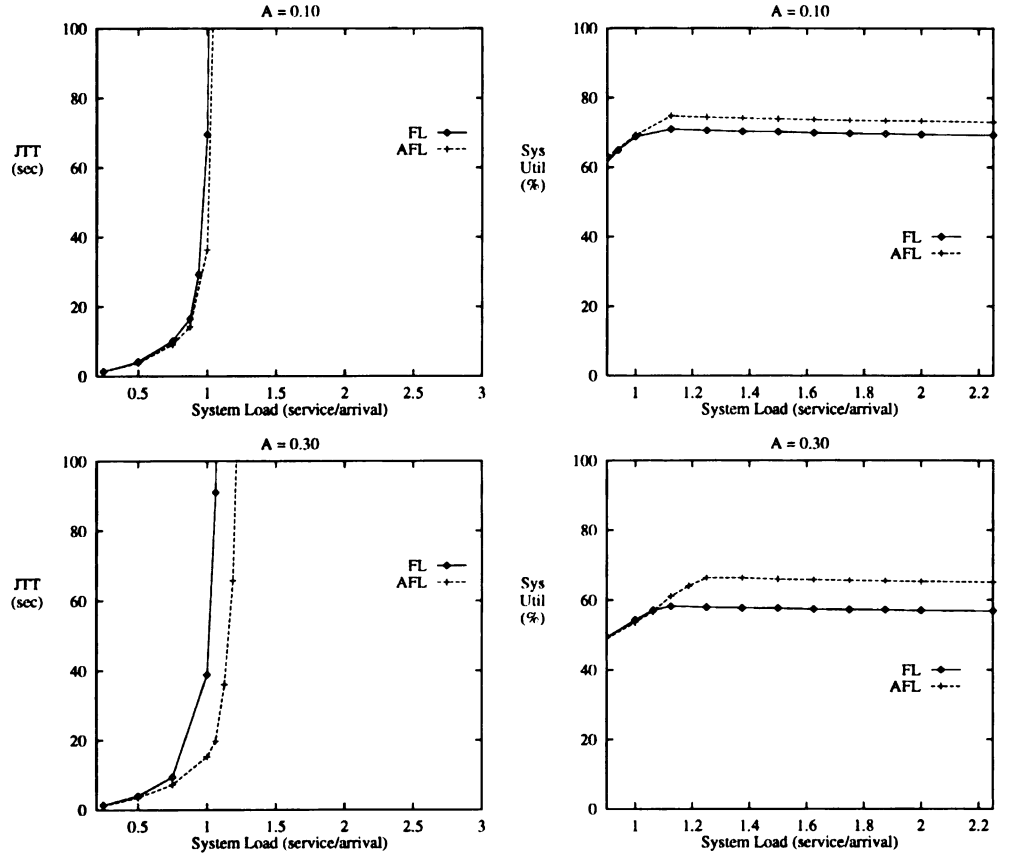

Figure 4.12: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[25,32]} = 1 - A$ interval distributions.

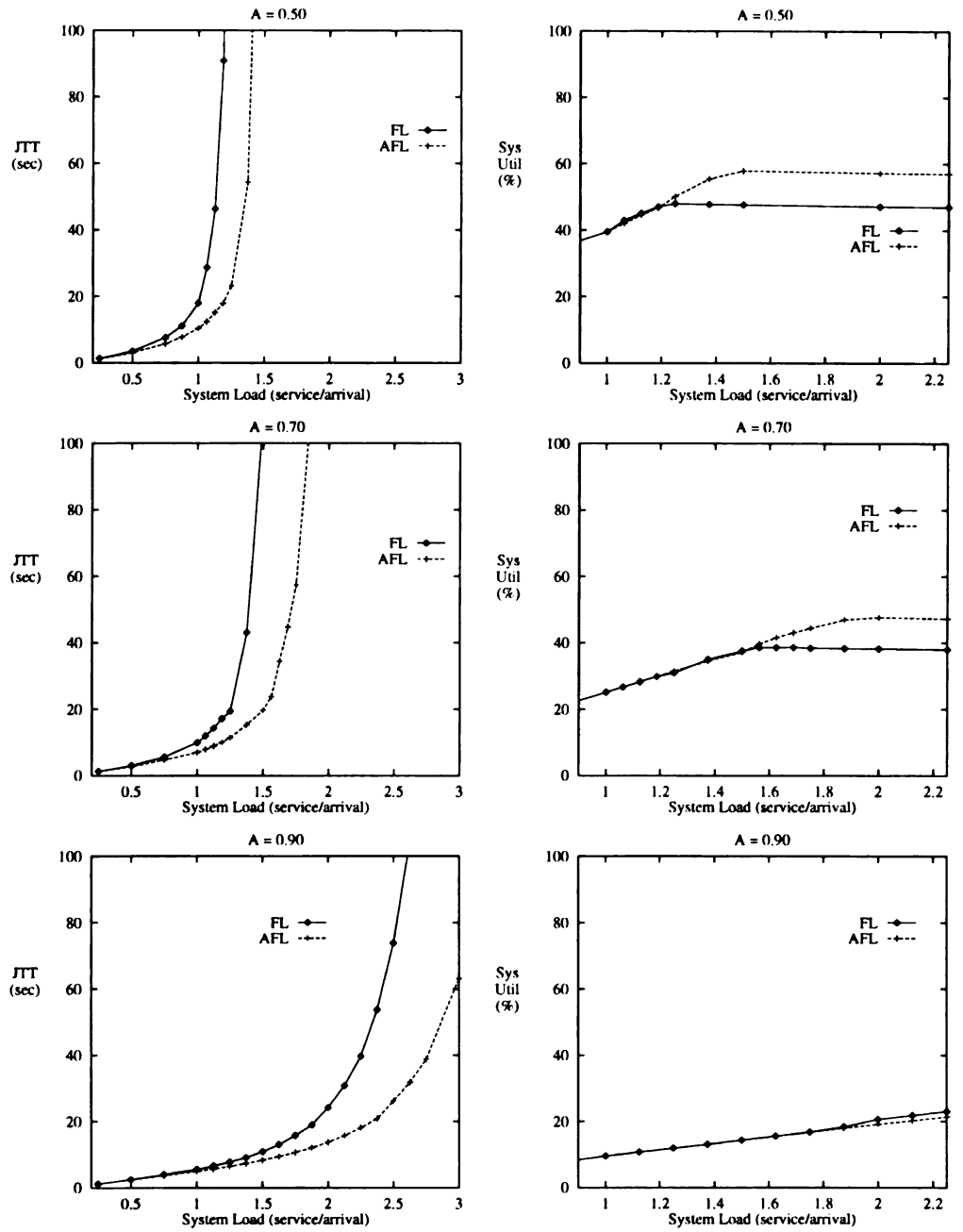

Figure 4.13: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[25,32]} = 1 - A$ interval distributions.

Figures 4.14 and 4.15 present the JTT and system utilization versus the system load for meshes generated using bipartite interval distributions in which the small mesh dimension is in the [1,8] uniform interval, and the large mesh dimension is in the [28,32] uniform interval. The interval distributions were assigned as follows: $P_{[1,8]} = A$ and $P_{[28,32]} = 1 - A$, for $A \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

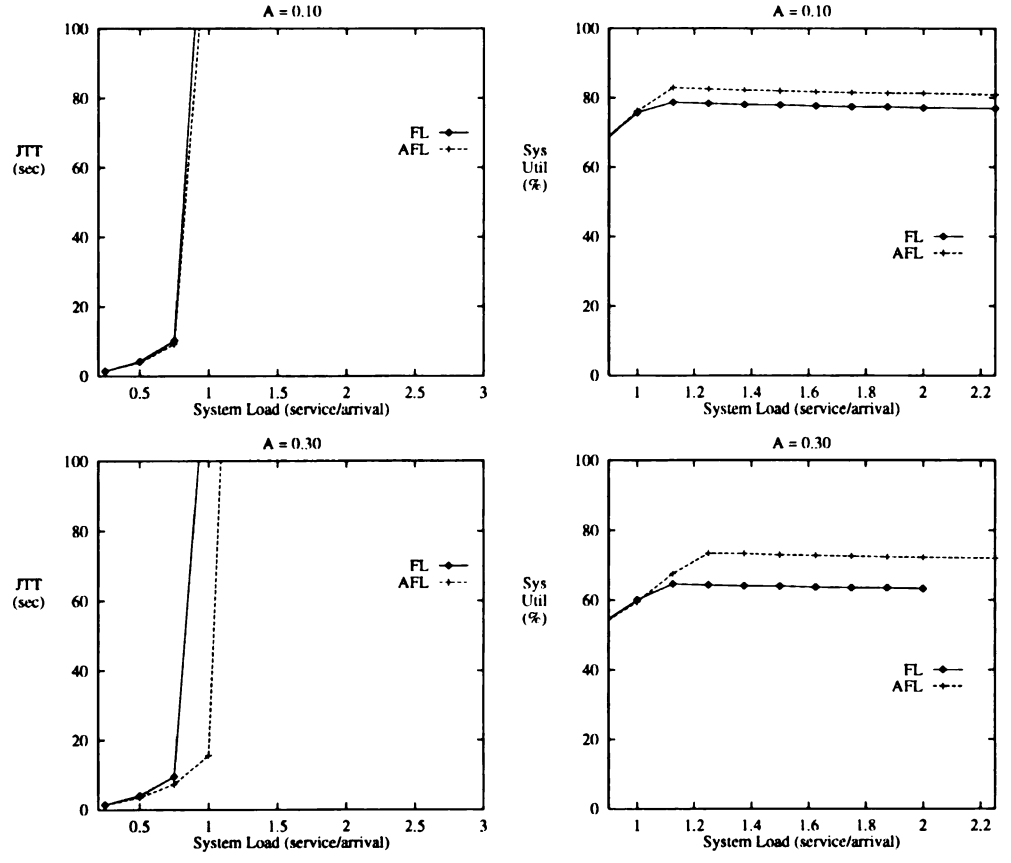

Figure 4.14: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[28,32]} = 1 - A$ interval distributions.

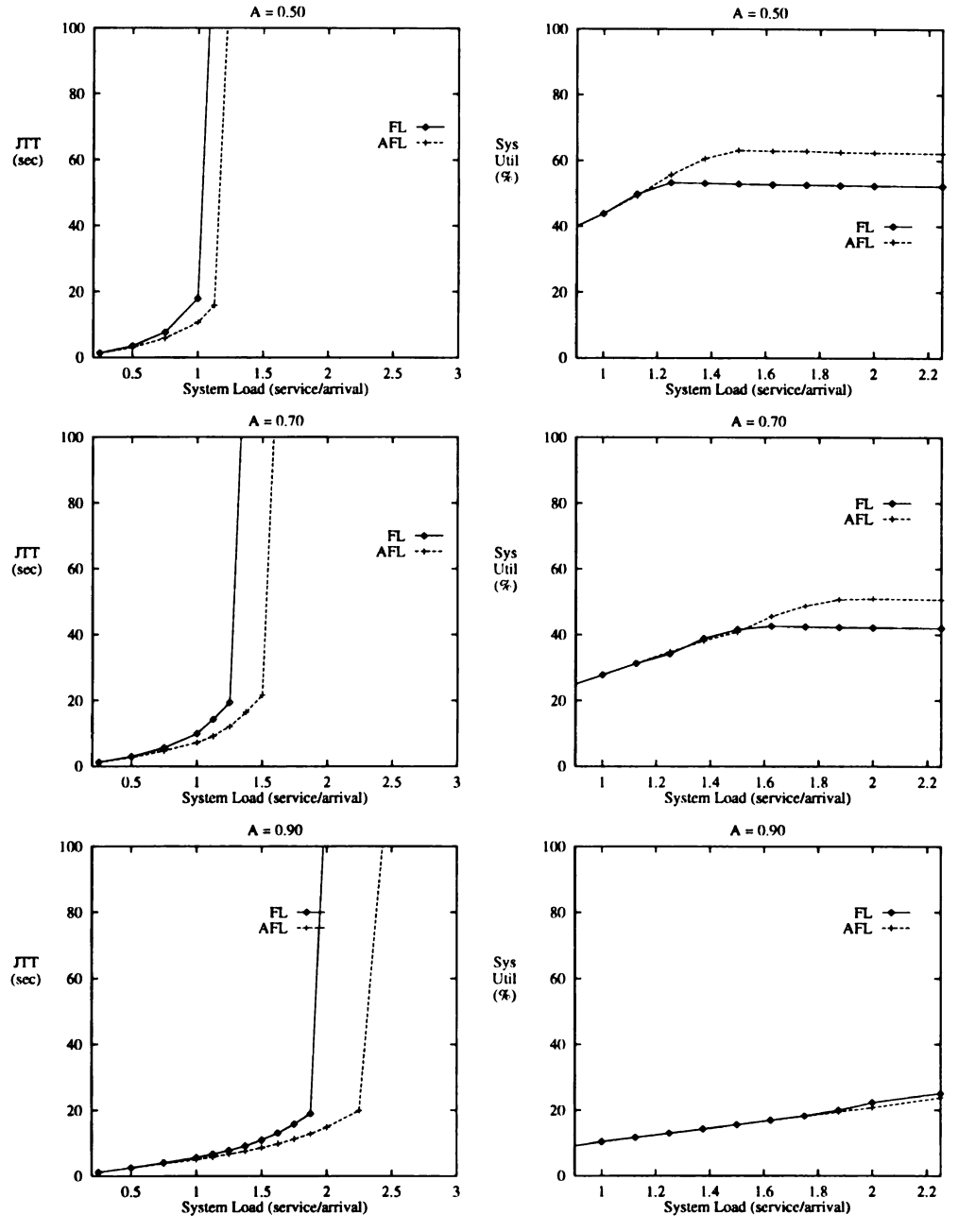

Figure 4.15: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[28,32]} = 1 - A$ interval distributions.

Figures 4.16 and 4.17 present the JTT and system utilization versus the system load for meshes generated using bipartite interval distributions in which the small mesh dimension is in the [1,8] uniform interval, and the large mesh dimension is in the [30,32] uniform interval. The interval distributions were assigned as follows: $P_{[1,8]} = A$ and $P_{[30,32]} = 1 - A$, for $A \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

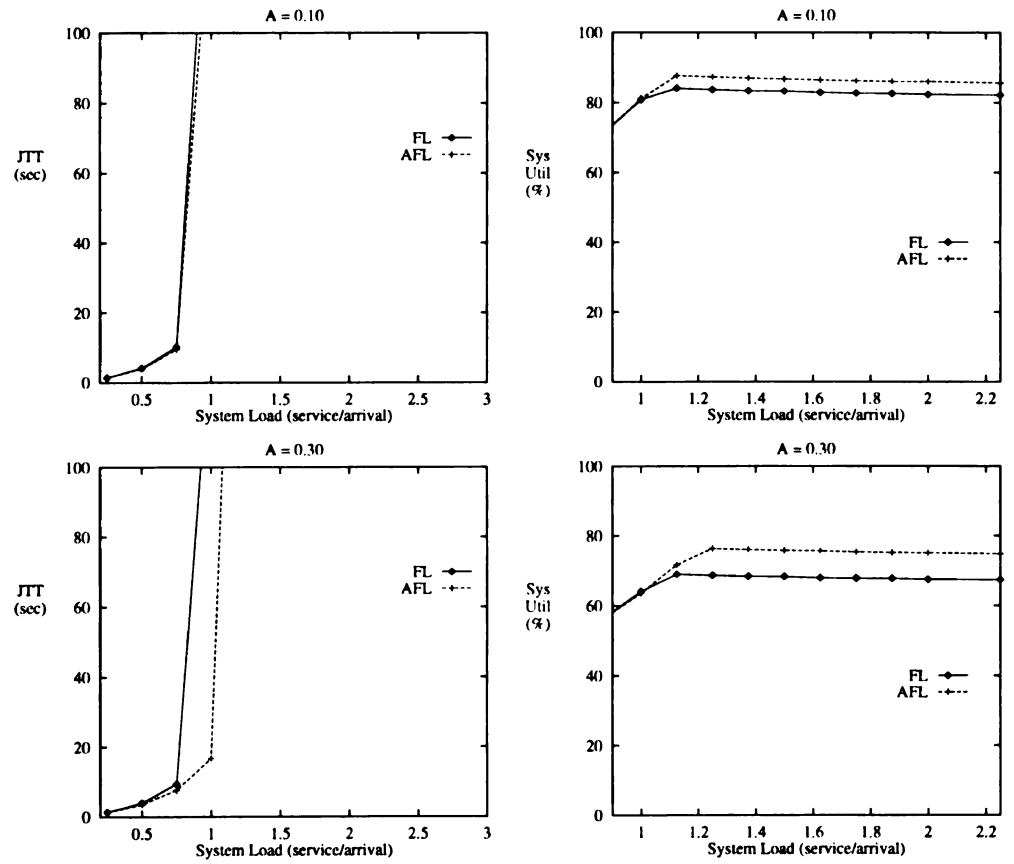

Figure 4.16: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[30,32]} = 1 - A$ interval distributions.

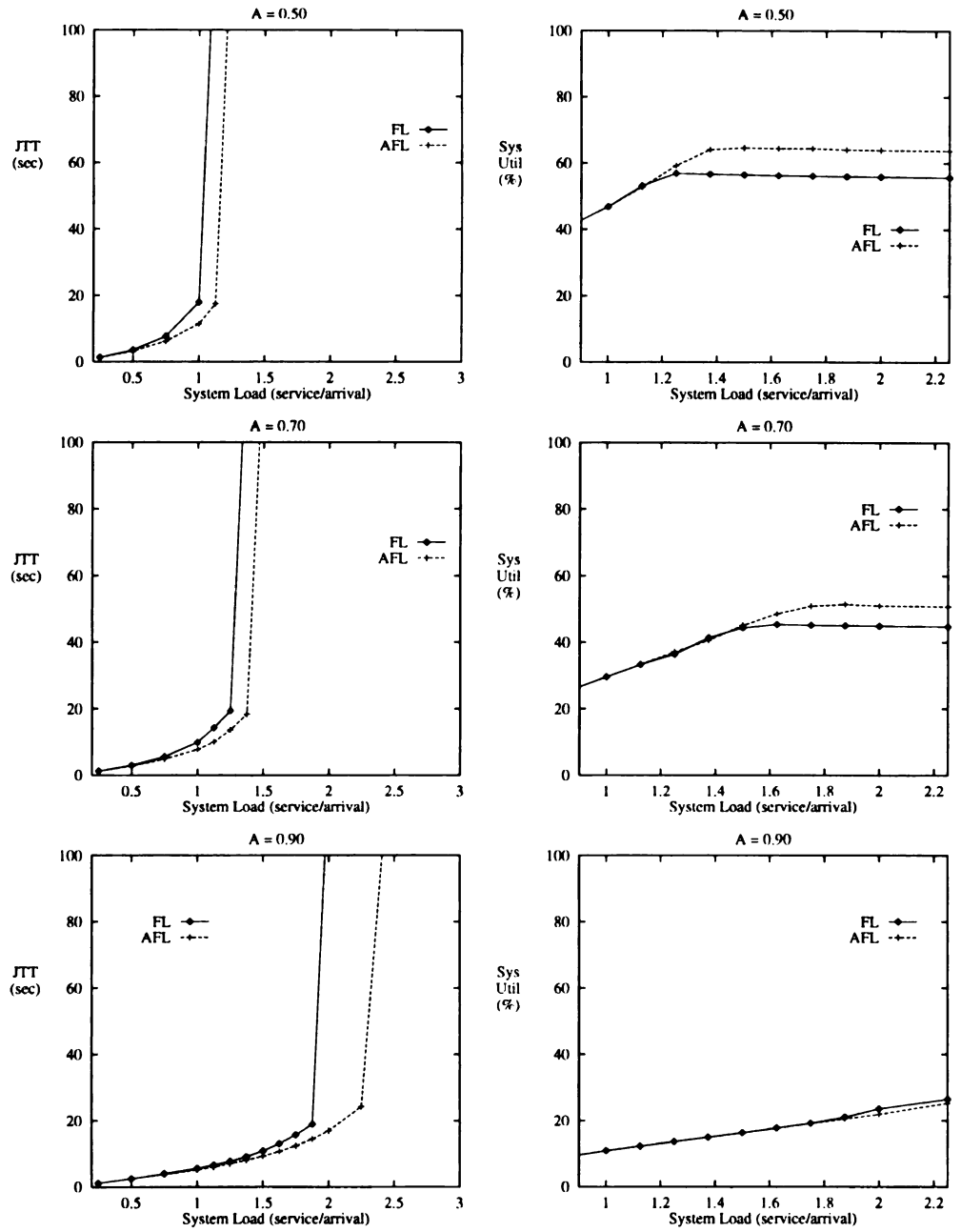

Figure 4.17: JTT and system utilization vs. system load for $P_{[1,8]} = A$, $P_{[30,32]} = 1 - A$ interval distributions.

4.3 Interval Distribution, Small Dimension in $[1, 16]$.

Figures 4.18 and 4.19 present the JTT and system utilization versus the system load for meshes generated using bipartite interval distributions in which the small mesh dimension is in the [1,16] uniform interval, and the large mesh dimension is in the [17,32] uniform interval. The interval distributions were assigned as follows: $P_{[1,16]} = A$ and $P_{[17,32]} = 1 - A$, for $A \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

The conclusions to be drawn from these experiments are similar to those presented in Section 4.2. However, the change in the lower interval from $[1, 8]$ to $[1, 16]$ has the further effect of increasing user JTT, yet improving system utilization. This occurs because, for the lower range of mesh requests, a greater percentage of requests are larger (that is, the $[1, 16]$ interval can create larger mesh requests than the $[1, 8]$ interval), thus using more of the system's resources. However, the JTT is adversely affected because the difference in the lower range of mesh requests results in a lower number of leftover subcubes that are large enough to satisfy incoming requests. This effect is more pronounced as A increases. For example, compare Figures 4.9 and 4.19.

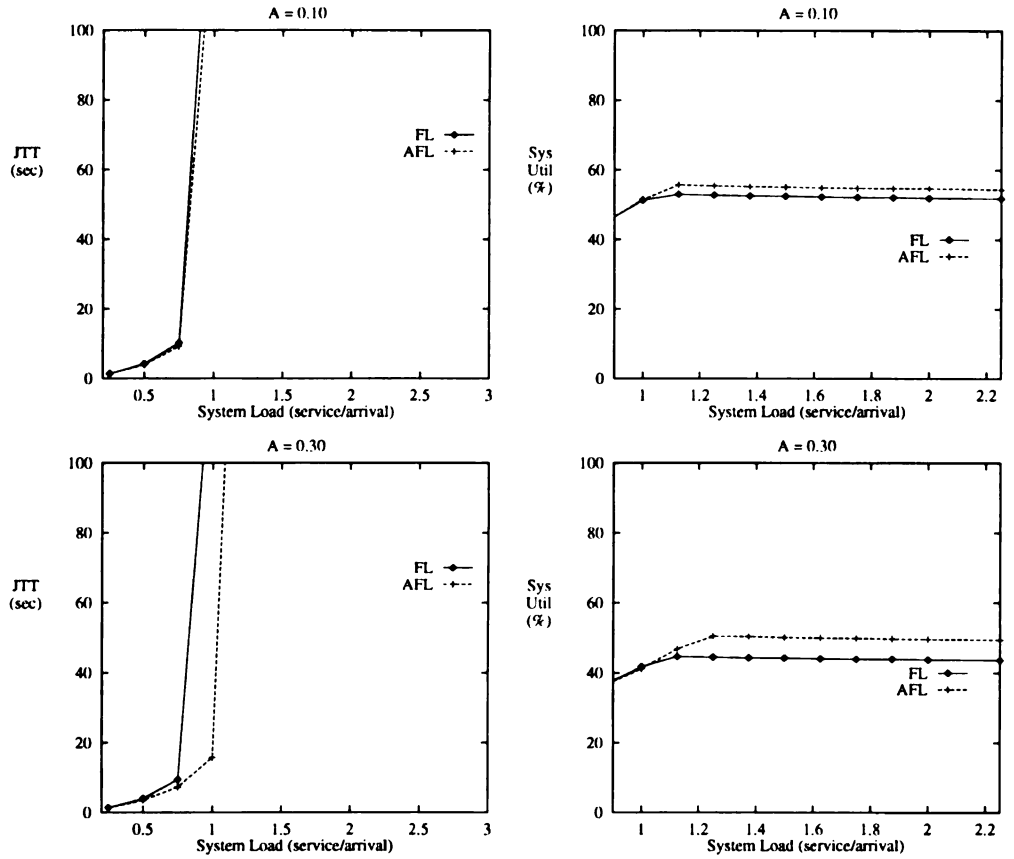

Figure 4.18: JTT and system utilization vs. system load for $P_{[1,16]} = A$, $P_{[17,32]} = 1 - A$ interval distributions.

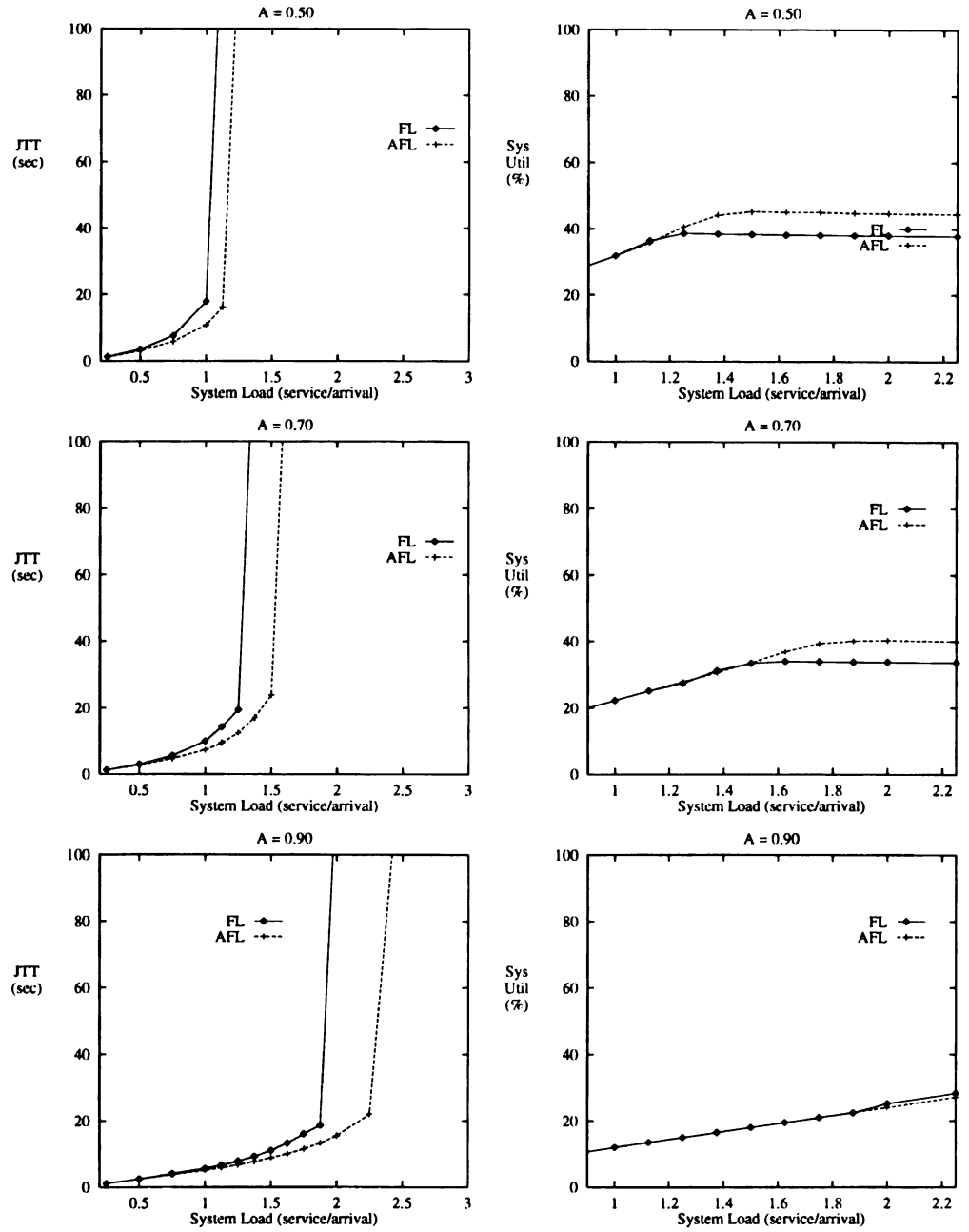

Figure 4.19: JTT and system utilization vs. system load for $P_{[1,16]} = A$, $P_{[17,32]} = 1 - A$ interval distributions.

Figures 4.20 and 4.21 present the JTT and system utilization versus the system load for meshes generated using bipartite interval distributions in which the small mesh dimension is in the [1,16] uniform interval, and the large mesh dimension is in the [20,32] uniform interval. The interval distributions were assigned as follows: $P_{[1,16]} = A$ and $P_{[20,32]} = 1 - A$, for $A \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

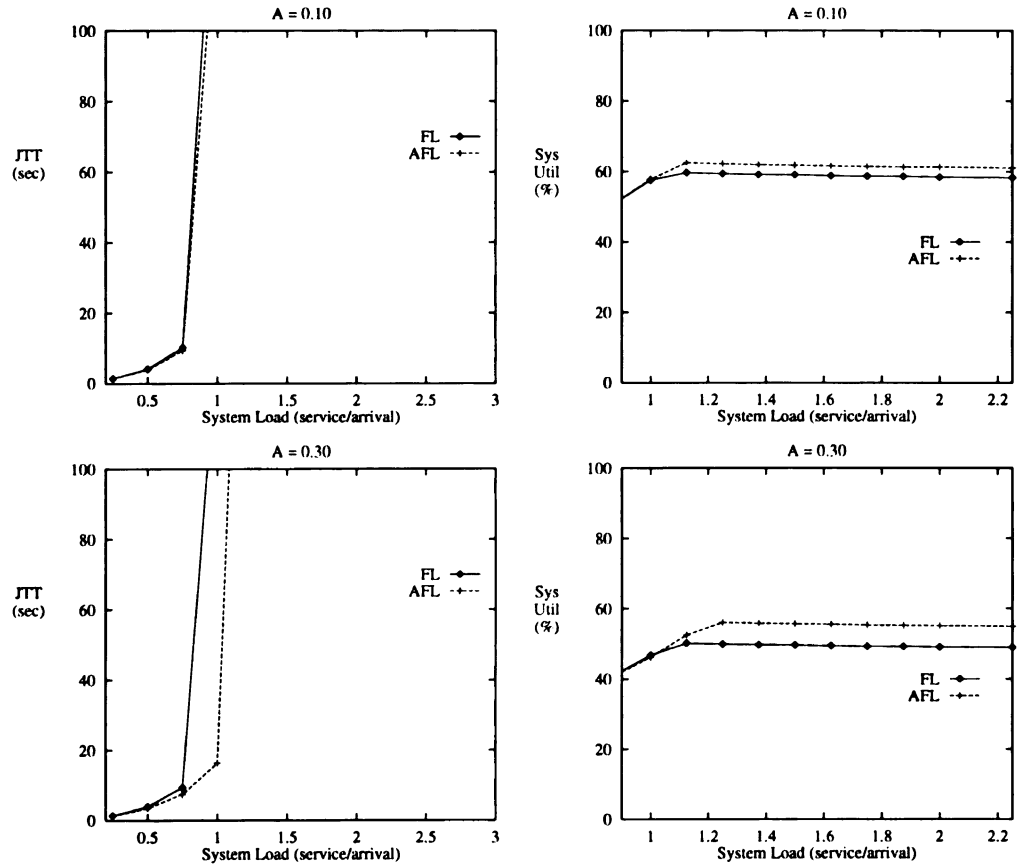

Figure 4.20: JTT and system utilization vs. system load for $P_{[1,16]} = A$, $P_{[20,32]} = 1 - A$ interval distributions.

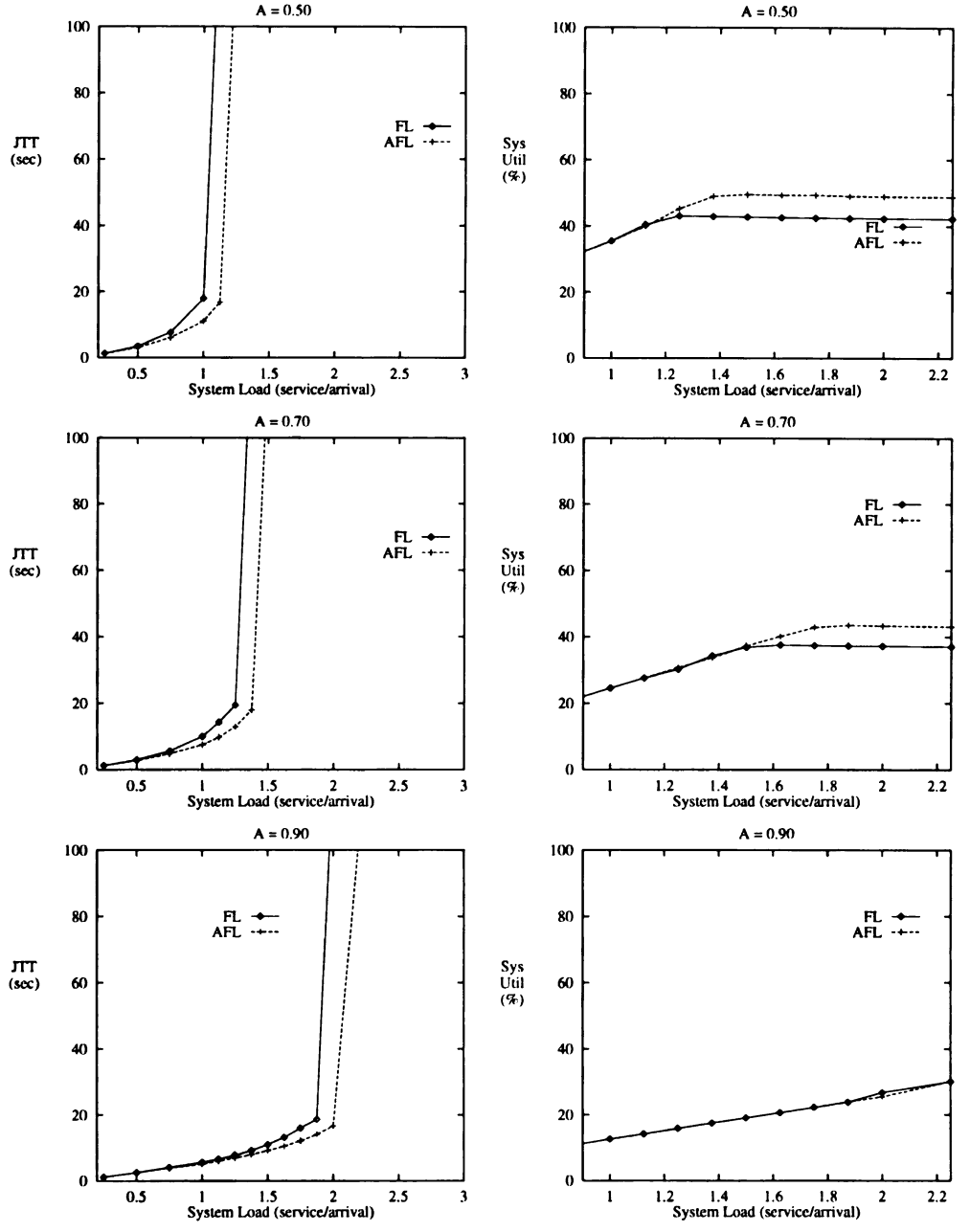

Figure 4.21: JTT and system utilization vs. system load for $P_{[1,16]} = A$, $P_{[20,32]} = 1 - A$ interval distributions.

Figures 4.22 and 4.23 present the JTT and system utilization versus the system load for meshes generated using bipartite interval distributions in which the small mesh dimension is in the [1,16] uniform interval, and the large mesh dimension is in the [25,32] uniform interval. The interval distributions were assigned as follows: $P_{[1,16]} = A$ and $P_{[25,32]} = 1 - A$, for $A \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

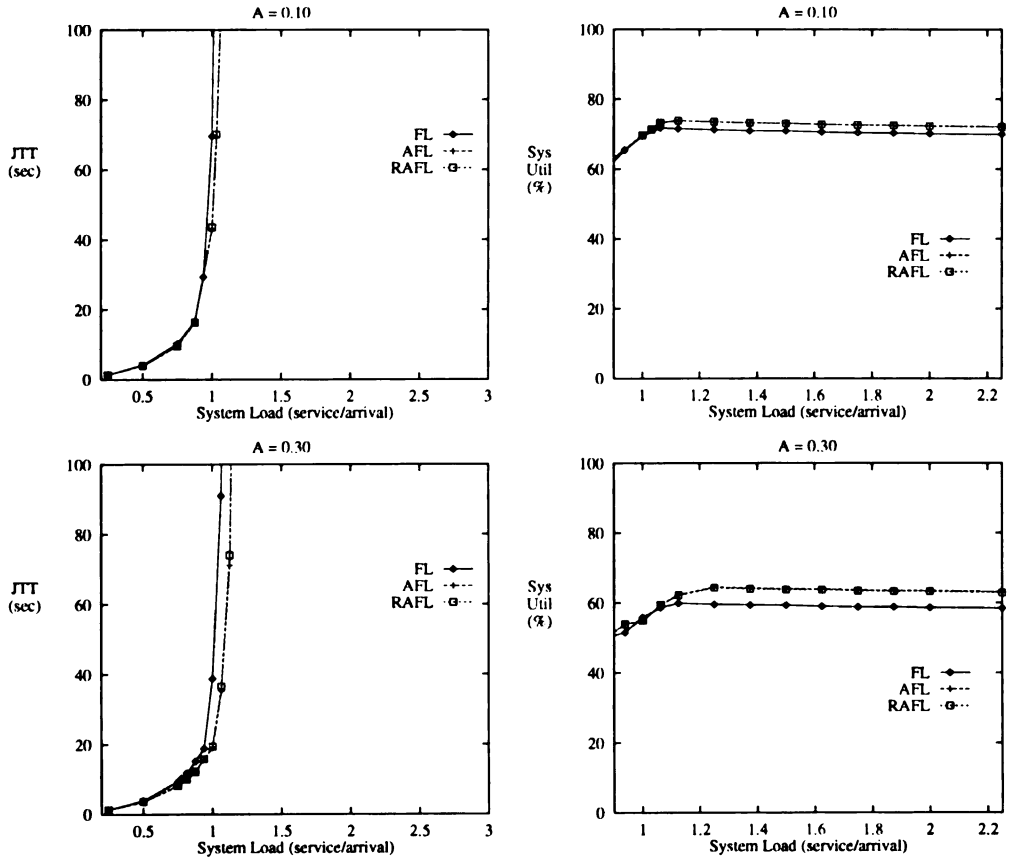

Figure 4.22: JTT and system utilization vs. system load for $P_{[1,16]} = A$, $P_{[25,32]} = 1 - A$ interval distributions.

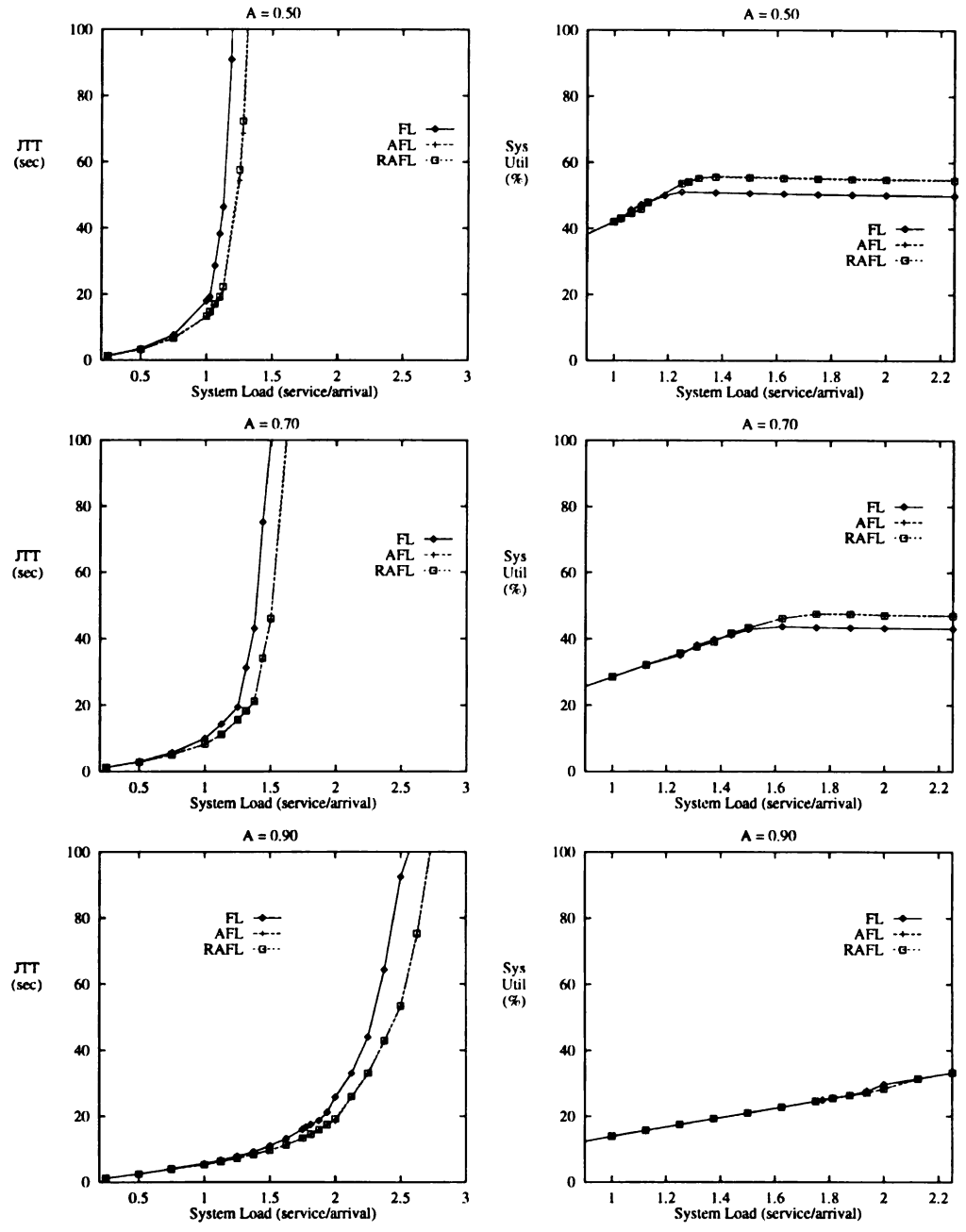

Figure 4.23: JTT and system utilization vs. system load for $P_{[1,16]} = A$, $P_{[25,32]} = 1 - A$ interval distributions.

Figures 4.24 and 4.25 present the JTT and system utilization versus the system load for meshes generated using bipartite interval distributions in which the small mesh dimension is in the [1,16] uniform interval, and the large mesh dimension is in the [28,32] uniform interval. The interval distributions were assigned as follows: $P_{[1,16]} = A$ and $P_{[28,32]} = 1 - A$, for $A \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

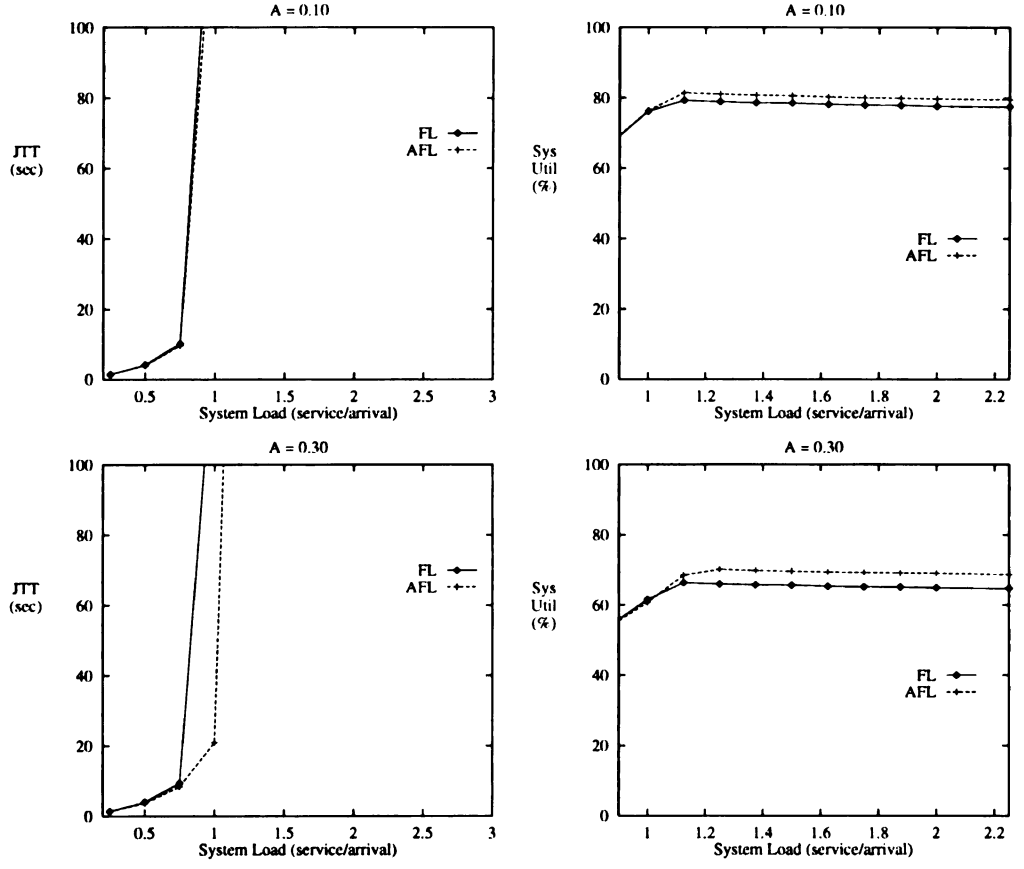

Figure 4.24: JTT and system utilization vs. system load for $P_{[1,16]} = A$, $P_{[28,32]} = 1 - A$ interval distributions.

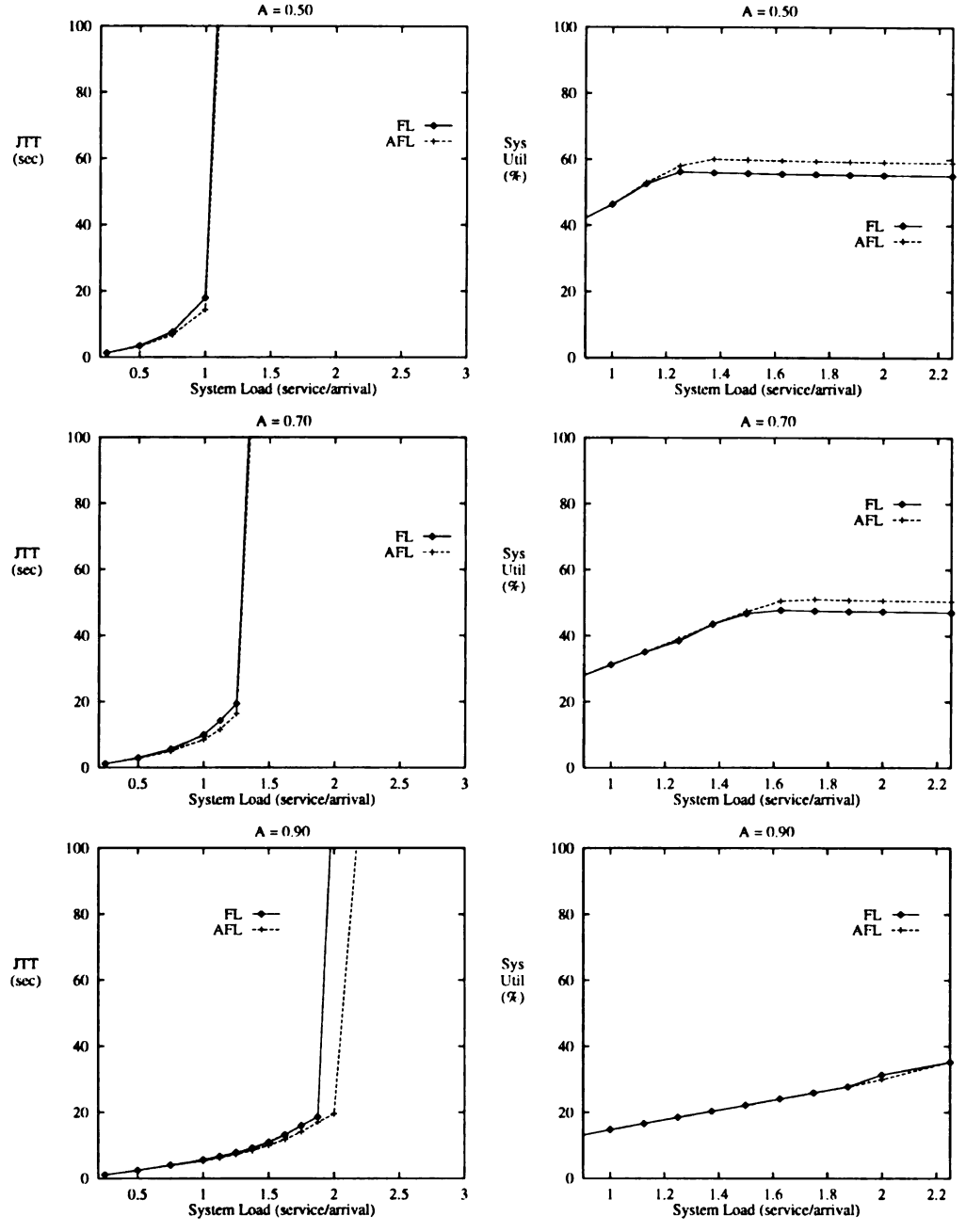

Figure 4.25: JTT and system utilization vs. system load for $P_{[1,16]} = A$, $P_{[28,32]} = 1 - A$ interval distributions.

Figures 4.26 and 4.27 present the JTT and system utilization versus the system load for meshes generated using bipartite interval distributions in which the small mesh dimension is in the [1,16] uniform interval, and the large mesh dimension is in the [30,32] uniform interval. The interval distributions were assigned as follows: $P_{[1,16]} = A$ and $P_{[30,32]} = 1 - A$, for $A \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

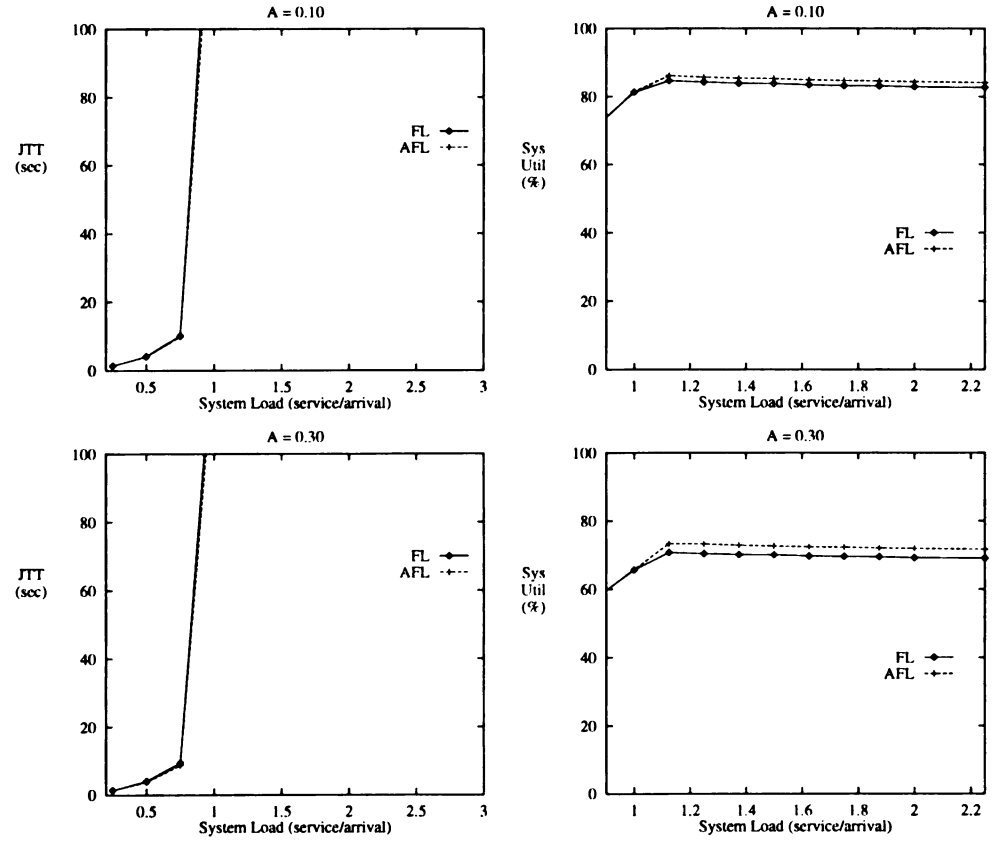

Figure 4.26: JTT and system utilization vs. system load for $P_{[1,16]} = A$, $P_{[30,32]} = 1 - A$ interval distributions.

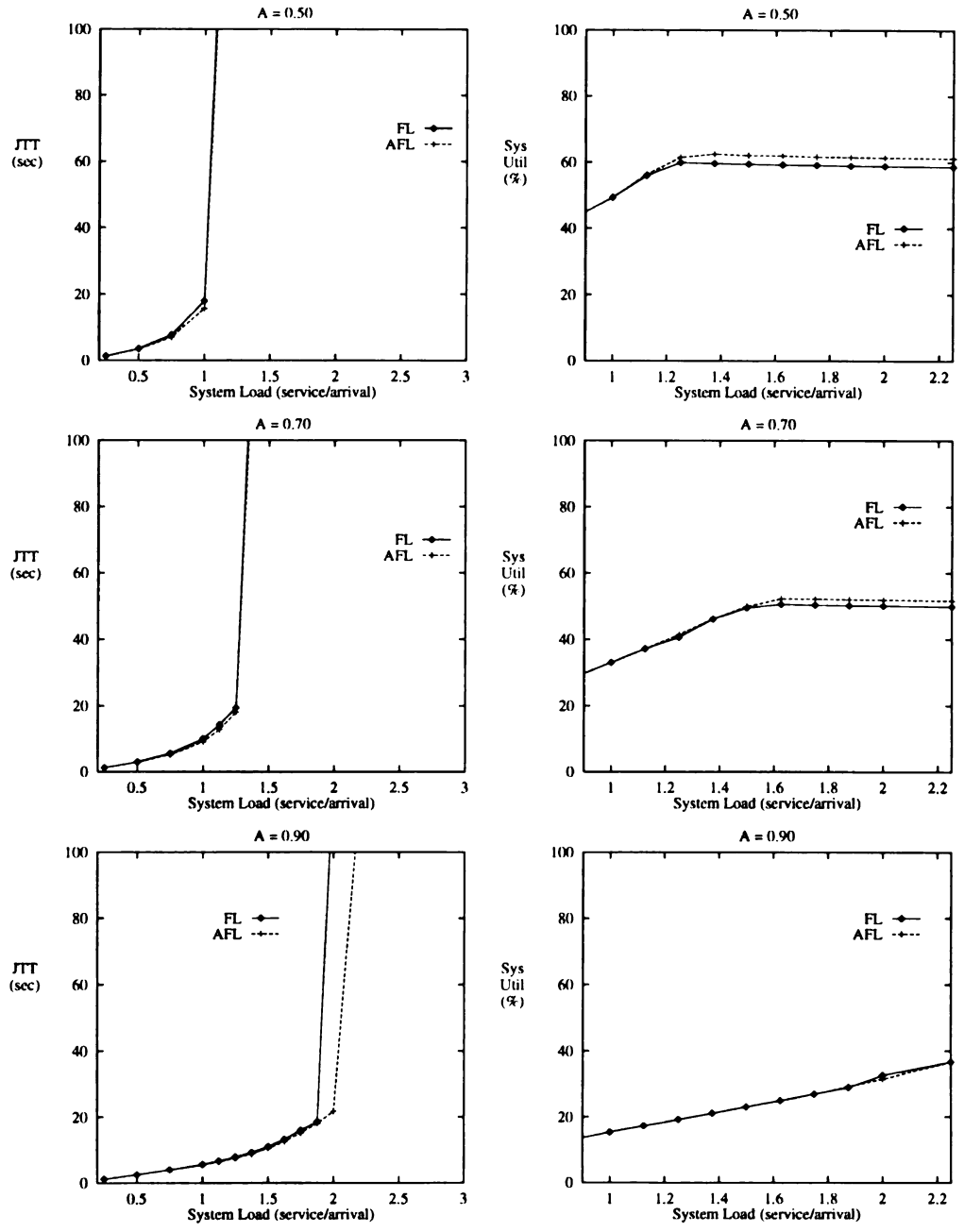

Figure 4.27: JTT and system utilization vs. system load for $P_{[1,16]} = A$, $P_{[30,32]} = 1 - A$ interval distributions.

Chapter 5

Timesharing in Workstation Clusters

This chapter discusses issues involving timeshared processor scheduling and cluster allocation in *Networks of Workstations*. We examined the problem of whether it is possible for high-performance parallel jobs to be scheduled in a timeshared fashion yet still accomplish two primary goals: (1) the JTT of the high-priority job is not affected; (2) system throughput is improved [44].

5.1 Problem Definition

In a NoW considered as a CPC, the primary performance concern is the *job turnaround time* (JTT) as perceived by the individual user. JTT measures the time a user's job spends in the system from the time it is submitted to the time it finishes its execution. Normally, *space sharing*, in which a job is given exclusive access to its cluster of processors, is used to minimize JTT. Another measure of performance is *system throughput*, which measures the

amount of work being performed by the system (that is, the number of processors occupied by jobs) within some unit of time.

The inherent flexibility of the workstation cluster makes it suitable for several divergent types of tasks. Its most common use is as a set of personal workstations for groups of primary interactive users. The use of workstation clusters for remote batch job execution has also been extensively examined [5, 34, 35, 36]. In a workstation-based CPC, JTT can be affected by a number of factors, such as message-passing latency, communication protocols, and the processor scheduling policy, as well as uneven workload distributions. These factors can contribute to sub-linear speedup for many user jobs, as illustrated in Figure 5.1. The difference between ideal and observed speedup often represents a certain amount of idle CPU cycles, which can be exploited through timeshared job execution.

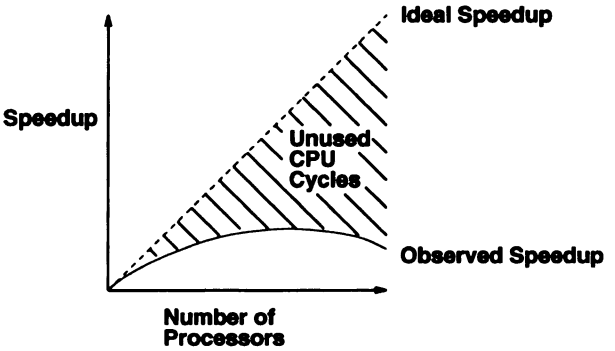

Figure 5.1: Difference Between Ideal and Observed Speedups.

In an ideal CPC environment, timeshared execution of parallel jobs would be achieved by redesigning the operating system to produce a lower overhead, more efficient system. For example, the common Unix operating system contains many daemon processes that reside in memory, creating extra overhead that may interfere with the execution of parallel jobs. An optimized CPC may eliminate many of these processes to achieve higher performance.

Classic work in processor scheduling has focused on static scheduling [2], in which the overall completion time of a set of pre-submitted jobs is the primary concern. For this

well-known NP-complete problem, specific job characteristics, such as the number of jobs, the cluster size, the execution time, and the memory requirements of each job, can be determined in advance. A static scheduling algorithm may consider these characteristics to determine an optimal schedule in a batch or real-time environment.

We consider the environment in which a CPC exists to be dynamic. Jobs arrive in a stochastic stream over time, making it impossible for the system to know an incoming job's characteristics beforehand. Thus, scheduling decisions rely on observable information, including the characteristics of executing jobs, as well as the current state of the system.

In order to ensure fairness, most system-level schedulers use the FCFS queueing discipline. Since the user's perceived performance is the primary concern, a processor scheduling policy in which their job is preempted by a later-arriving job is unacceptable. For the system to maintain high performance for the highest-priority job, that job's execution must not be hindered by later-arriving, lower-priority jobs.

The approach of this study is to use a *first-come, highest-priority* (FCHP) queueing mechanism to schedule the timeshared execution of incoming user jobs. In a workstation cluster, the use of timeshared scheduling can measurably increase system throughput. Furthermore, the use of priority allows us to greatly reduce the penalty paid, by the highest-priority job, for having to share its cluster of processors with lower-priority jobs. In most cases, the high-priority job achieves performance at, or close to, the performance it achieved under space sharing. Our scheduling techniques, implemented on an existing operating system, provide a simple testbed for processor scheduling experiments in NoW-based CPCs. Our study involves the use of existing, unmodified kernel scheduling primitives, with no other changes to the operating system or communication protocols.

5.2 Characterization of Workload

The organization of message-passing activity in a workstation cluster normally is non-structured, in the sense that there may be no benefit to using a particular kind of message-passing pattern. For example, a program in which each process communicates with its four logical nearest neighbors will not necessarily derive a benefit from executing on processors connected by an Ethernet or FDDI crossbar. Therefore, the experiments in our study are concerned not with the patterns of communication, but with the volume of communication that occurs.

For our experiments, we chose two programs that have varying *communication-density* characteristics. The first program is the *Embarrassingly Parallel* (EP) kernel from the NAS parallel benchmark suite [1]. In this program, there is an initialization period in which some communication is used to perform self load-balancing, followed by a long period of computation. Each thread of EP performs some initial computation, and work is allocated on each processor depending on how much each thread was able to finish during the initialization period. At the end of the program's execution, each thread sends its results to the initiating process. Thus, the EP kernel is characterized as having a high average processor utilization with very little interprocess communication.

The second program is an example of a barrier synchronization program, a common parallel programming model in which the execution can be divided into a set of *phases*, each of which consists of computation followed by communication that is used to synchronize processes and/or communicate intermediate results [45]. Since it is impossible to test many

[1]We wish to thank Dr. V. Sunderam of Emory University for providing the code for the NAS Kernels used in our measurements.

different programs, we implemented the barrier program to create an artificial load on processors in order to perform our experiments.

Figure 5.2 presents a space-time diagram of the message-passing activity of the barrier program executing on 6 processors in our testbed. The horizontal lines represent CPU activity for each thread of the parallel process, while the gaps in the horizontal lines represent times during which the CPU is idle due to communication. The angled lines connecting processors 1 through 5 to processor 0 represent messages being sent to/from the barrier. The figure shows that on processor 0 (the barrier processor), there is little idle time, yet on the non-barrier CPUs, there are significant amounts of CPU time that the program does not use.

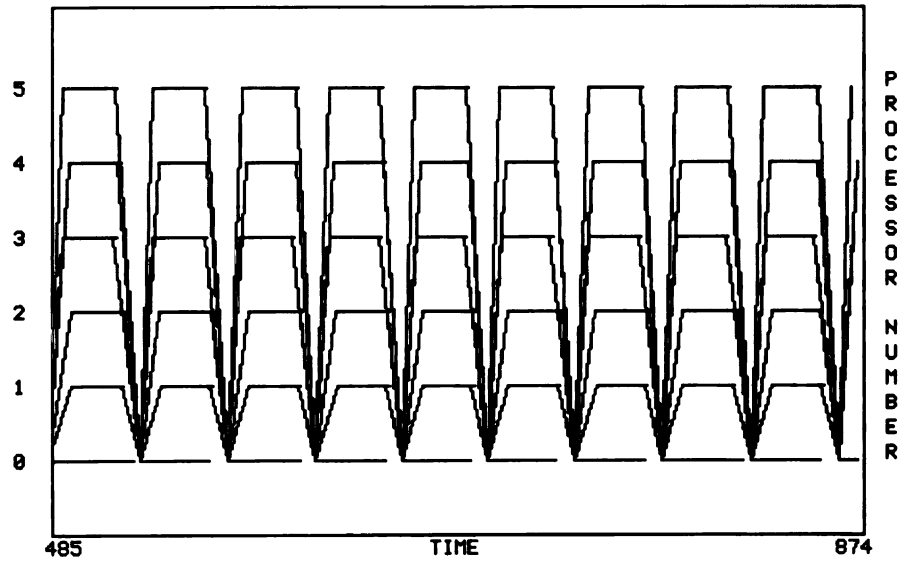

Figure 5.2: Barrier communication characterization.

Our experiments with the barrier program used three versions differing according to the number of phases, the total amount of work per phase, and the amount of data passed at the end of each phase. The parameter ρ determines the number of phases performed. The amount of work per phase, w, determines the total amount of work to be divided among the P processors in each iteration. For example, if $w = 900,000$ and $P = 6$, then each processor

performs 150,000 inner loop iterations per phase. The token length, t, determines the amount of data sent by processes at the end of each computational phase.

Each of our test programs uses a master/slave paradigm, in which the master program spawns P copies of the slave program, which perform the parallel computation (the master program resides on the same processor as one of the slave programs). After spawning the slaves and sending initial data to them, the master program goes to sleep, waiting for a "done" message from the slave program that implements the barrier.

5.2.1 Format of Experiments

Space Sharing Performance

Performance measurements for each program when it had exclusive use of the cluster (that is, under space sharing conditions) were taken to measure each job's maximum possible performance in our environment. The experiments were run under both *preemptive* and *non-preemptive* kernels. In a preemptive kernel, a high-priority job that is ready to run forces any executing low-priority job to be moved off the CPU (that is, preempted). In a non-preemptive kernel, a high-priority job that is ready to run must wait for a low-priority job to either expire its time quantum or block.

The results of these experiments are given in Figures 5.3 and 5.4. The relatively low speedup of EP is due primarily to two factors in our environment: (1) all of our experiments measured the program's total execution time, including the load-balancing overhead built into the EP program; (2) our experiments used the small EP problem size ($N = 2^{22}$, where N defines the total number of outer loop iterations in the EP kernel). For example, the full problem size of EP ($N = 2^{28}$), running on 6 processors, achieved a speedup of nearly 6 in our environment.

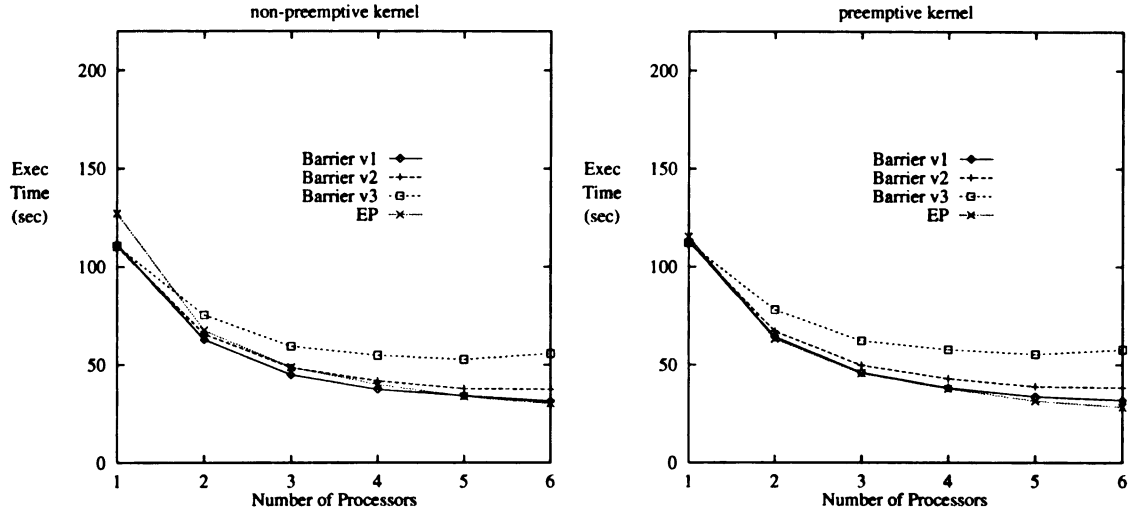

Figure 5.3: Programs running alone, execution time.

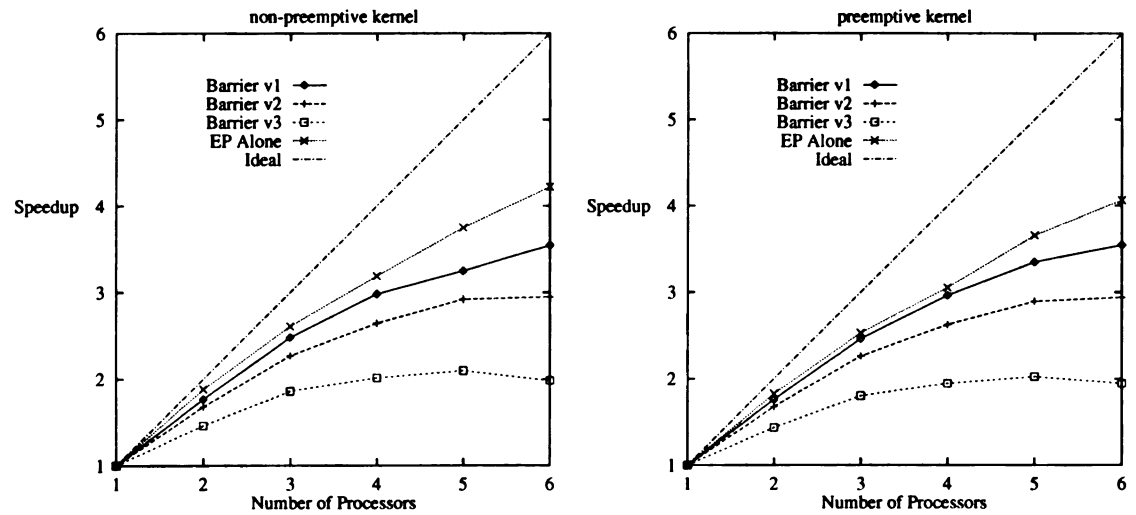

Figure 5.4: Programs running alone, speedup.

The higher overhead of the preemptive kernel resulted in a slightly higher overall execution time (and slightly lower speedup) for each barrier program. This overhead was incurred primarily because the barrier programs' communication overhead resulted in greater overhead imposed by the preemptive kernel. Thus, the difference in execution time between the non-preemptive version and preemptive version grew as the number of processors grew. For example, the 5 processor execution time, on the non-preemptive kernel, for version 2 of Barrier was approximately 37.7 seconds. The same version required approximately 38.3 seconds when running under the preemptive kernel. The opposite effect occurred for the

EP program because of its lack of communication. It ran slightly faster on the preemptive kernel, because it ran with lower context switching activity, as well as low message-passing activity. For example, its 5 processor execution time on the non-preemptive kernel was about 33.9 seconds, while it required about 31.5 seconds on the preemptive kernel.

Table 5.1 gives the parameters used with Barrier to simulate different workloads. The message size passed at the end of each computational phase was kept constant so that the expected program delay would be the same, under space sharing conditions, for each program. The variable parameters were chosen so that the total work performed by each version is approximately the same. In addition, the execution time of each version of Barrier on one processor is approximately the same as that of the EP program. The number of phases and the number of iterations per phase are varied so that the frequency of communication and the duration of each phase changes. Version 1 obtains a speedup close to that of EP. Version 3 simulates a program in which communication begins to dominate the execution of the program once a certain number of processors are used. As Figure 5.3 illustrates, its execution time increases from 5 to 6 processors. The parameters of version 2 were chosen so that its performance would fall between versions 1 and 3.

Version	Phase iterations	Number of Phases	Message Size (bytes)
1	1,350,000	830	800
2	890,000	1250	800
3	445,000	2500	800

Table 5.1: Barrier Parameters used to simulate different workloads.

Timesharing Experimental Methods

For this study, the purpose of using timeshared scheduling was to improve overall system throughput by allowing jobs to share time in the cluster. However, simply allowing an arriving job to begin execution on a previously occupied cluster of processors would adversely affect the performance of the previously executing job. Therefore, our experiments examined the use of priority scheduling to allow incoming jobs to use idle CPU cycles without severely affecting the JTT of existing high-priority jobs.

Figure 5.5 gives an illustrative example of our experimental method. In this figure, EP and a version of barrier are run concurrently (using timeshared scheduling) in a cluster. When the first job to finish terminates, the remaining job is given exclusive control of the cluster (normally, the high-priority job is expected to finish first). The measured execution time of the programs gives an estimate of their JTT if they begin execution at approximately the same time. The figure also illustrates the fact that multiple measurements were taken for each experiment to improve confidence intervals.

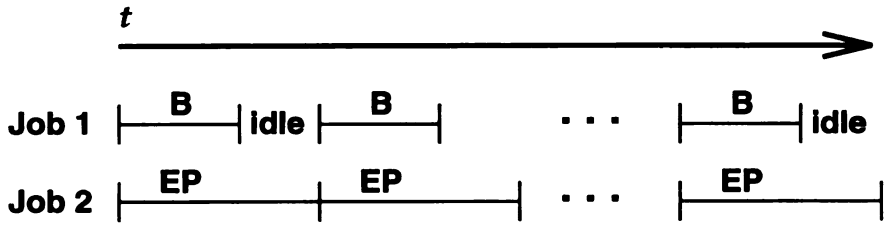

Figure 5.5: Experimental method.

The purpose of this experimental method is to simulate the condition in which the second job to arrive suddenly becomes the high-priority job when the original high-priority job terminates. The first set of experiments, implemented on a non-preemptive kernel, used a system call embedded in the user code that reset the priority of the program at the beginning of execution on each CPU. The high-priority job retained its default priority, while the low-

priority job issued the system call. The second set of experiments were implemented on a preemptive kernel and performed using the facility described in Chapter 6. All timesharing experimental results are reported in Section 6.5.

5.2.2 Testbed

The platform on which we conducted our experiments is a *homogeneous* distributed workstation environment configured for high-performance parallel computing research. It is composed of 6 identical DEC Alpha 400 AXP workstations, connected by a DEC GIGAswitch crossbar network, as well as by a standard Ethernet connection. The limitation of 6 processors in our environment was overcome by using different versions of Barrier to vary its granularity. Each Alpha is configured with 32 MB of RAM and a 424 MB SCSI disk, on which resides the OSF/1 (v. 2.0) operating system. In order to ensure minimal interference from outside factors, all experiments using the GIGAswitch were performed when no other user jobs were running on the Alpha workstations.

All user jobs involved in our experiments are written in C with PVM 3.2 [6]. The programs were implemented so that *direct routing*, in which each thread of the program bypassed the PVM daemon after connections were established, was used whenever possible. Our initial experiments illustrated that not using direct routing in our environment would have a significant adverse affect on program performance. For example, running on the non-preemptive kernel, the space-shared execution time of version 1 of Barrier increased from approximately 31.4 seconds to 34.4 seconds on 6 processors. Furthermore, not using direct routing resulted in considerably more overhead and/or interference attributable to the PVM daemon.

The preemptive kernel referred-to in the text was implemented by configuring the OSF/1 kernel for the realtime (RT) environment. This configuration enabled various realtime system calls to be used. Several scheduling classes were also enabled, allowing the use of the required preemptive scheduling by the operating system. Chapter 6 describes the facility implemented on the RT kernel that was used to perform the experiments that required the use of a preemptive kernel.

5.3 Analytical Model

The purpose of the analytical model presented here is to understand the penalty paid by high-priority jobs when they are run in a timeshared fashion with low-priority jobs. For the purposes of this study, we assumed that the operating system scheduling mechanism employed in the non-preemptive (non-RT) kernel is *round-robin with multilevel feedback* (RRMF) [46]. RRMF maintains a stratum of different job priorities, in which jobs at the same priority level are given timeshared access to the CPU using round-robin (RR) scheduling, while jobs at lower priority levels are not given CPU time unless all jobs at a higher priority level are either blocked or finished. Furthermore, we assumed that once a job occupies the CPU on the non-RT kernel, it may not be preempted by another user-level job. The average context-switch time is assumed to be very small relative to the total execution time of a program, and is therefore ignored in the models presented here. Under these assumptions, a simple model was constructed to represent the execution time of a program in a timeshared workstation cluster.

Consider a barrier program that has exclusive access to its cluster of processors. In this case, the total program execution time, E_B, is determined by several factors: (1) the average execution time of each computational phase, e_B; (2) the average message-passing

delay at each processor, δ; (3) the number of phases executed by a program, ρ; and (4) an initial setup time at the beginning of the program, s. Since no process of a Barrier process may proceed past the barrier before all other processes are finished with a given phase, each process is conceptually identical. Thus, a model of the activity on one CPU is sufficient to characterize the behavior of the program, as well as determine its approximate execution time. For example, Figure 5.6 presents an execution model of Barrier when it maintains exclusive control of its cluster. For the Barrier program, the execution time can then be represented with the equation:

$$E_B = s + \rho(e_B + \delta) \tag{5.1}$$

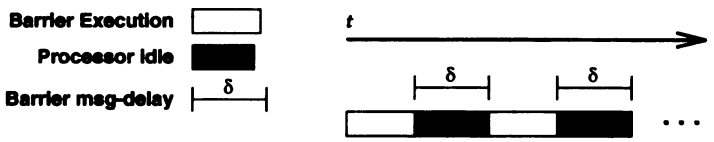

Figure 5.6: Execution pattern of Barrier with exclusive control of cluster.

For our experiments involving the two jobs sharing time on processors, there are two major analytical models to consider: (1) EP running with Barrier; (2) Barrier running with Barrier.

5.3.1 Barrier with EP, Non-preemptive Kernel

Figure 5.7 presents an execution model of barrier and EP running at the same priority. In this model, EP and Barrier execute in round-robin fashion during Barrier's execution phases, as well as during Barrier's startup phase (assume that context switching occurs when Barrier and EP are executing in round-robin fashion, and that no context switching occurs when EP executes during Barrier's message delay). This results in each phase of Barrier being (approximately) doubled in total time, since EP and Barrier are both given

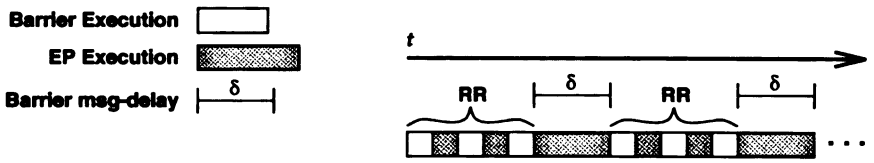

Figure 5.7: Execution pattern of Barrier and EP, equal priority.

equal time quantums during each of Barrier's execution phases. When Barrier is blocked waiting for a message, EP is given exclusive access to the CPU, which results in a random additional delay, d, imposed on Barrier when its message arrives. In this case, $d \leq q_{EP}$, where q_{EP} is the time quantum given to EP while Barrier is blocked (this value cannot be determined precisely). Using this model, the execution time of barrier can be approximated by a simple equation:

$$E_B = 2s + \rho(2e_B + \delta + d) \tag{5.2}$$

At the processor implementing the barrier, the message delay is normally close to 0 (that is, $\delta \ll e_B$). Thus, the overall execution time can be approximated by:

$$\begin{aligned} E_B &= 2s + \rho(2e_B + 0 + d) \\ &\geq 2s + \rho(2e_B) \\ &= 2(s + \rho e_B) \end{aligned} \tag{5.3}$$

Since the progress of the Barrier entire program depends on its barrier process, Equation 5.3 estimates that the overall execution time of Barrier can be expected to approximately double, since the execution time of its barrier process will approximately double.

Figure 5.8 presents a model of a barrier program running with EP executing at a lower priority, in which EP is only given CPU time when Barrier is blocked due to the issuance of a message receive. In this case, the execution time of Barrier is expected to be close to

that available when Barrier maintains exclusive control of its cluster, although one can also expect that the execution time will be somewhat higher due to the random extra delay, d, occurring whenever Barrier's message arrives.

$$E_B = s + \rho(e_B + \delta + d) \tag{5.4}$$

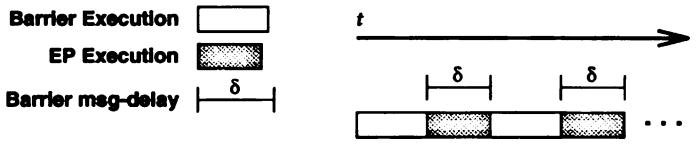

Figure 5.8: Execution pattern of Barrier and EP, EP at low priority.

This model estimates that with EP running at a lower priority, the execution time of Barrier will be affected primarily by the random extra delay, d, without otherwise having its performance penalized. If the value of d is small enough, EP may "steal" CPU cycles on the processors on which Barrier is executing without severe penalty to the JTT of Barrier.

5.3.2 Barrier with Barrier, Non-preemptive Kernel

When considering one version of Barrier executing concurrently with another, an additional message delay is introduced into the model. However, the choice of keeping the message size constant for different versions of Barrier allows us to simplify the analysis, as we can assume that $\delta = \delta_1 \approx \delta_2 \approx \delta_3$ (where δ_i is the delay experienced by version i). Since we are primarily concerned with the execution time of the high-priority process when both processes are running at separate priority levels, we omit an analysis of the two programs running at the same priority. For all analyses in this subsection, we refer to the high-priority version of Barrier as "p1" and the low-priority version of Barrier as "p2" (e.g., the total execution time and phase execution time of p1 are denoted by E_{p1} and e_{p1}, respectively).

The most informative means by which to analyze the performance of these programs under timeshared conditions is to examine the activity at the processor implementing the barrier process of p1. Since each of the other processes of the program depends on the barrier process to resume execution, the barrier process effectively dominates the progress of the program's execution. On the processor implementing the barrier, the program's CPU demand is high, and the message delay is close to zero (refer to the graph of processor 0 in Figure 5.2).

For example, if the barrier processes of both programs are run on the same CPU, then the situation as illustrated by Figure 5.9 can occur. In this case, the message delay for p1 is initially almost zero, but p2 interferes with the execution of p1 by obtaining a quantum of CPU time every time p1 issues a message-receive. For P processors, the processor implementing p1's barrier issues $P-1$ message-receives before issuing a multicast to the $P-1$ other processors. Since the CPU demand by p2 is also high, it will likely obtain a time quantum every time p1 issues a message-receive. Therefore, the execution time of p1 in the cluster can be approximated by:

$$E_{p1} = s + \rho(e_{p1} + (P-1)d) \tag{5.5}$$

where $d = q_{p2}$, the time quantum given to p2 when p1 issues a message-receive.

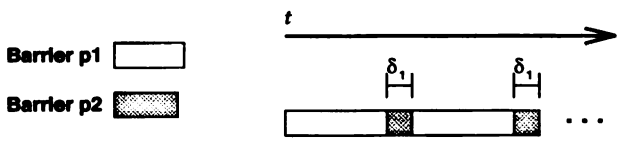

Figure 5.9: Execution pattern of two Barriers with both barrier processes on one CPU.

Figure 5.10 presents the model of execution in which the processes implementing the barrier for each program are on different CPUs. The figure models the activity of the pro-

cessor implementing p1's barrier. In this case, p2 is still able to steal some CPU cycles from p1. However, since the CPU demand of p2 is expected to be lower on non-barrier processors, the likelihood that p2 will require fewer time quantums when p1 reaches the barrier increases. Thus, the execution is expected to be less than that predicted by Equation 5.5 but greater than that obtainable with exclusive access to the cluster. In addition, it is possible for idle CPU time to occur on processors not implementing p1's barrier, as illustrated in the figure.

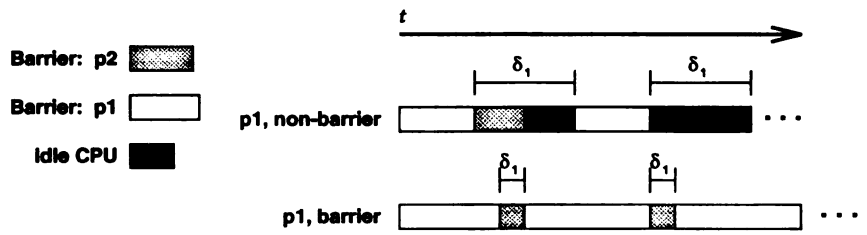

Figure 5.10: Execution pattern of high-priority Barrier with low-priority barrier when barrier-implementing processes are on different CPUs.

5.3.3 Analytical Model With Preemptive Kernel

The prior analyses considered the execution time of the high-priority job if it is scheduled using the RRMF scheduling mechanism. If preemptive scheduling is employed, two fixed-priority scheduling mechanisms can be used, including the *FIFO* and *RR* scheduling mechanisms. The FIFO mechanism is defined recursively as follows:

- when a FIFO process at the head of a process queue becomes ready to run, it preempts any process running at a lower priority. It will run until it blocks or terminates, or until it is preempted by a higher-priority process.

- When a FIFO process blocks, the CPU is given to the next process existing in the FIFO queue of the same priority.

- If no such processes exist, then the CPU is given to the process at the head of the queue at the next highest priority level.

The RR scheduling mechanism is defined similarly, except that processes are given time quantums instead of being allowed to run until they block.

Therefore, when we consider a high-priority Barrier job running using a preemptive scheduling mechanism, with the additional assumption that no other processes exist at the same priority level, then the execution of the Barrier job can be modeled by Figure 5.6, and its expected execution time is represented by Equation 5.1. Thus, on a system that uses preemptive scheduling methods, a high-priority process should be able to achieve its ideal execution time, regardless of whether lower-priority jobs exist in the system.

5.3.4 Discussion of Analysis

In order to simplify our model, and because context switching overhead is assumed to be relatively low when compared to the total execution time, our analysis ignored the effects of context switching. However, in real execution, the extra context switching occurring during the timeshared execution of programs can be expected to add some overhead to the execution times of the programs.

The analyses predicted that an improvement in system throughput should be obtainable if the relative priority of two jobs are adjusted with respect to each other. For example, figure 5.8 shows that under ideal conditions, the low-priority EP program is able to use the CPU while the Barrier program is waiting for message delivery. When two Barrier programs occupy the same cluster, however, the analyses predict that little, if any, throughput improvement may be obtained. Figures 5.9 and 5.10 illustrate that this low predicted

improvement is due primarily to the high CPU demand of the high-priority job's barrier process.

Of further consequence is the random delay introduced when the low-priority job is executing when the high-priority job's message arrives. This delay, which exists only under a non-preemptive kernel, is a function of the time quantum given to the low priority process, and may be as high as one time quantum. The analyses predicted that this delay can have a measurable affect on the JTT of the high-priority job, when the second job is executing at a lower priority. If the random extra delay does not exist (as in the case of the preemptive kernel), then the analysis predicted that the high-priority job will obtain the same performance that is available under pure space sharing conditions.

Chapter 6

Elmer: A Scheduling Facility for a Network of Workstations

In the previous chapter, we discussed the issues involved in employing timeshared scheduling to take advantage of idle CPU cycles in a workstation cluster. Our analyses made two predictions regarding the execution time of the high-priority job: 1) if a non-preemptive kernel is employed, then the high-priority job will obtain a JTT close-to, but not exactly equal to, the JTT that can be obtained under space sharing conditions; 2) if a preemptive kernel is employed, then the high-priority job will obtain a JTT equal to the JTT that can be obtained under space sharing conditions. The analyses also showed that under certain conditions, system throughput can be improved when two jobs are allowed to share time in the system.

The predictions suggested a potential experimental technique, in which the source code of a low-priority job is modified to reset its priority with respect to the high-priority job. However, there are several drawbacks to this technique. First, it requires user involvement in the form of source-code modification, which is unrealistic in a real high-performance

computing environment. Second, there is no automatic means to reset a job's actual priority if its effective priority is changed. That is, if the low-priority job suddenly became the high-priority job, its priority could not be automatically improved, relative to other processes. Third, there is no way to implement automatic cluster allocation. The code implementing the EP and Barrier programs require that the programs perform their own cluster allocation.

Furthermore, the RRMF (round-robin with multilevel feedback) scheduling policy of a non-preemptive kernel can allow a low-priority job to steal a significant number of CPU cycles from a high-priority job. On a CPU that uses RRMF scheduling, a process at a low priority level is not scheduled on the CPU unless all processes at higher priority levels are unable to run. However, a lower-priority process can be given a longer round-robin time quantum than higher-priority processes, and no user-level process can be preempted by another user-level process, regardless of the relative priorities. Therefore, once a low-priority process is scheduled on a CPU, it may continue to execute even when a higher-priority process becomes ready to run. Processes scheduled using the RRMF policy are also not guaranteed to remain at the same priority level throughout their life, resulting in the remote possibility that a job designated as high-priority could end up executing at a lower priority than a job designated as low-priority. The drawbacks of RRMF revealed the need for the use of preemptive scheduling policies to guarantee that the higher-priority job executed whenever it was ready.

The preceding conclusions provided the motivation to create the facility described within this chapter. This facility, *Elmer*, was developed as a processor scheduling facility for a NoW-based CPC. It also serves as a testbed for experimental *job dispatching* and *cluster allocation* methods, in order for us to further study and improve processor scheduling methods within a NoW-based CPC. For job dispatching methods, which schedule the order of

execution of incoming jobs, Elmer's queueing algorithm can be modified to allow for the prioritization of user jobs. Elmer's cluster allocation facility, which allocates processors to incoming jobs, is designed to have its cluster allocation algorithm modified to take advantage of various forms of observable system information, such as individual CPU utilization and the overall degree of timesharing allowed.

The Elmer facility also provides a practical solution to the problems involved with modification of user code to reset priority. Elmer automatically schedules user jobs, and it automatically allocates clusters of processors for them. Minimal user-code modification is required in the form of a library call that registers the user job with the Elmer facility, enabling it to correctly determine the processors on which it can execute. Elmer also automatically sets and resets the priority of executing jobs, depending on when other jobs enter and exit the system.

6.1 Implementation Issues

The Elmer facility operates as follows: a job arrives in the system when a user executes a command that submits his/her job to be executed by the facility. Elmer runs a cluster allocation algorithm to determine whether sufficient resources exist for the job. If insufficient resources exist, then the job is placed in a queue. When the job is scheduled for execution, daemons on each CPU intercept it by resetting its priority with respect to previously executing jobs. The system daemons also monitor the progress of each job, automatically resetting the priority of executing jobs when other jobs enter and/or leave the system. Furthermore, the Elmer facility uses the realtime scheduling facilities of the OSF/1 kernel to implement preemptive process scheduling, ensuring that the highest-priority job can run whenever it is ready.

A central issue in research in scheduling for a NoW is how it is perceived as a computational resource. Two primary models exist. A NoW can be treated as a collection of independent computers, each of which is "owned" by its primary user, or it may be treated as a CPC, available for the execution of any job. Under the ownership model, the quality of service to a workstation's owner is the primary concern. If the owner is currently using his/her workstation, then he/she should perceive no change in responses from the system. The CPC model views the set of workstations in much the same way as a single parallel processor might be viewed, in which the entire system, consisting of many processors, memories, disks, and a network, is viewed as a single computational resource on which to execute parallel jobs.

The CPC model provides the premise under which Elmer operates. In a typical high-performance computing environment, jobs are submitted and scheduled through some form of system-wide scheduling mechanism to ensure fairness and high performance for individual jobs. However, a user operating within the typical workstation cluster environment performs parallel programming by using different types of user-level software, such as PVM [6], p4 [31], or MPI [32]. These systems often require the user to perform job scheduling and cluster allocation tasks, which are normally performed by the operating system in a CPC. Thus, the purpose of Elmer is to provide a scheduling facility with which high-performance jobs can be executed in a workstation cluster. Elmer provides a somewhat simpler interface for user-level parallel programming tools, as well as a centralized queueing point to reduce contention for system resources. Elmer is designed to run on a computer architecture consisting of a set of Unix-based [46] workstations connected by some form of network.

The primary goal of Elmer is to ensure that the highest-priority job receives the highest-possible quality of service, which means that the highest-priority job should always be able

to achieve the shortest possible JTT for the number of processors it uses. Traditional CPC scheduling methods ensure high-quality service by enforcing *space sharing*, in which a job exclusively occupies its cluster of processors. In contrast, Elmer resets the priority of processes on individual CPUs to allow them to coexist. Jobs are assigned their initial priority according to the queueing mechanism implemented within Elmer. As a simple and effective means to guarantee fairness, it currently enforces the *first-come, first-served* (FCFS) queueing discipline. It also extends this concept to include priority, in which the first-to-arrive job receives the highest priority (*first-come, highest-priority*, or FCHP). This concept is similar to the ownership model described above. The highest-priority job executing in a cluster is considered to be the *owner* of the cluster until it terminates. When an owner job terminates, the first job that arrived after the owner job is elevated to owner status, and its priority is updated accordingly.

For timeshared parallel job execution, the primary implementation issue to consider is that of how to maintain a high level of service to the highest-priority process on an individual CPU. Since minimizing JTT is of greatest importance, the highest-priority process must be able to execute whenever it is ready. Ideally, to ensure that lower-priority jobs cannot interfere with the progress of the high-priority job by infringing upon its CPU time, the high-priority job must always be able to preempt lower-priority jobs. Under many circumstances, this would require a modification of the scheduling facilities within the Unix kernel. However, the OSF/1 operating system implements the Posix realtime standards [47], which contain the FIFO scheduling class. This class is recursively defined as follows: a Posix FIFO process running at a given priority level will continue to run until it blocks, voluntarily yields the CPU, or is preempted by a process running at a higher priority level. When a Posix FIFO process becomes ready to run and reaches the head of the queue for its priority

level, it preempts any lower-priority processes occupying the CPU. Thus, our implementation uses realtime scheduling mechanisms to ensure that the highest-priority user job runs when it is ready.

The realtime scheduling facilities also allowed us to implement a form of *self gang-scheduling* for the highest-priority job. At each processor, the highest-priority process preempts all other processes when it is ready to run. By definition, this approach is a form of gang-scheduling, because gang-scheduling requires that all processes of a user job be running (occupying their respective CPUs) at the same time. However, this scheduling policy is somewhat weaker in its requirements than gang-scheduling, for two reasons: 1) none of the lower-priority processes can be guaranteed to receive gang-scheduling, 2) in order for this policy to be effective, all processes of a user job must exist at the same, highest, priority level. Any lower-priority processes will hinder the execution of the program.

The need for all processes of a high-priority job to execute at the highest priority is illustrative of some of the cluster allocation issues that can be studied by using the Elmer facility. First, there are a number of cluster-allocation issues for lower-priority jobs. For example, it may be undesirable for a lower-priority job to occupy a set of processors contained in the clusters of two (or more) separate higher-priority jobs. Second, the CPU usage on individual processors must be considered. Third, our facility currently does not consider memory usage in its scheduling decisions. Fourth, the global *degree of multiprogramming* is a factor in making cluster allocation decisions. Fifth, the facility is currently implemented on a homogeneous network of workstations. The Elmer facility can be used to study issues involving heterogeneous facilities.

The queueing facility used to implement cluster allocation and job dispatching uses a centralized scheduling mechanism because this mechanism has been shown to be simple and

efficient to operate. Furthermore, Theimer and Lantz [34] examined the tradeoffs between centralized and decentralized scheduling facilities, concluding that although decentralized facilities are more fault tolerant, centralized facilities are more scalable and perform better. Section 6.4 will present some measurements that examine the performance of Elmer's centralized facility.

We also considered the issue of robustness in the design of Elmer. In order for Elmer to be robust, it must be able to perform several system tasks without failing, which may include: 1) the cancellation of executing jobs; 2) the ability to monitor system activity, such as CPU usage and/or memory usage; 3) support for PVM and other user-level parallel programming software; 4) the ability to change the system queueing policy.

When an executing job is canceled, the Elmer facility must be able to accordingly update the priority of lower-priority jobs that were sharing that job's cluster. That is, new jobs may not be allowed to preempt previously executing jobs. In monitoring CPU and/or memory usage, the Elmer facility provides itself with information that allows more intelligent cluster allocation and job dispatching decisions to be made. Support for user-level parallel programming software is essential in a NoW-based CPC, because very few user programs are written that use direct message-passing implementations. The ability for Elmer's queueing policy to be changed is important, because it will allow further experiments, involving advanced clustering and scheduling methods, to be conducted.

6.2 Facility Functions

The Elmer facility is composed of two root-level daemon processes, Elmer and *RSD* (ReScheduling Daemon), together with several user-level commands for interfacing with the facility. The Elmer daemon is a centralized queueing facility responsible for all job dispatching and

cluster allocation decisions within a NoW, as well as communication with users. The RSD daemon acts as a slave server that performs actions requested by Elmer. It is responsible for intercepting incoming jobs and rescheduling their priorities, as well as other actions related to the local CPU. Figure 6.1 illustrates the communication between Elmer and RSD in a workstation cluster. Running on one workstation, the Elmer daemon accepts user requests, either queueing or executing jobs as appropriate. When an action local to a workstation is required, Elmer sends a message to the RSD running on that workstation via a TCP connection. The RSD then executes the action, which may include: kill a running job, report CPU activity, reschedule an incoming job, and reschedule running jobs because a higher-priority job has finished execution.

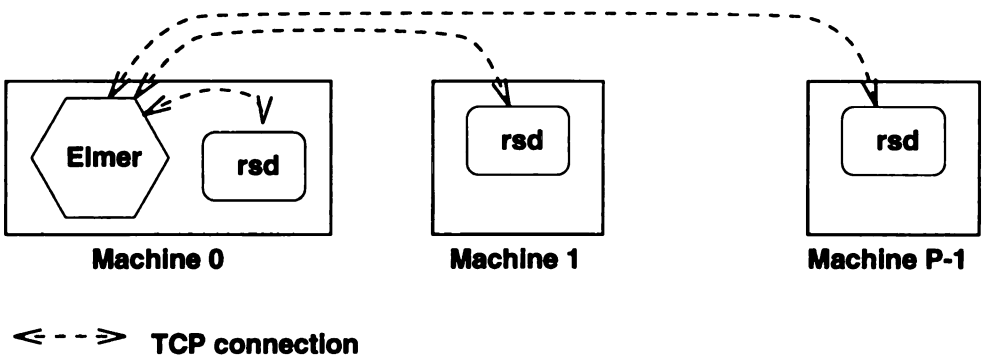

Figure 6.1: Logical view of Elmer operations.

The operation of Elmer with RSD to schedule an incoming job works as follows: the user issues the **esubmit** command to submit their parallel program to be executed. Elmer runs the cluster allocation allocation algorithm, which uses information passed from **esubmit**, together with state information from the system, to determine whether the job can be allocated within the cluster of workstations. If the cluster allocation algorithm allows it, then Elmer sends a message to RSD to inform it that a user job is entering the system. The message includes information such as the job's sequence number, its priority level, and the user identification.

RSD then waits for the incoming job to begin executing, after which it collects system information to determine all PIDs of the newly executing job. When a list of PIDs is obtained, RSD resets the priority and/or scheduling class of the user job to an appropriate value. All user jobs execute at a lower priority level than Elmer or RSD.

Cluster allocation for a user job requires some interaction with the job itself. PVM programs require that the user's job specify a set of hosts on which to execute. Therefore, the Elmer interface for PVM requires a certain amount of user participation in the form of a library call that must be inserted into the user's PVM code. This library call simply reads an environment variable, set by Elmer, to determine the set of hosts on which the program may execute. The library call is implemented for C programs only. It is inserted into the user code as follows:

```
#include "elmer_reg.h"


main()
{
  elmer_register(); /* Run this at the start of program execution */
  ...
}
```

In the course of implementation, it was discovered that the user-level PVM daemon can have a great affect on the operation of the user-level program. That is, if a single user runs two separate PVM-based programs using the Elmer facility, then each program must interact with the same PVM daemon. It was discovered that this interaction greatly impaired the performance of both jobs, especially if realtime scheduling techniques were being used. Currently, there is not a solution for this problem. However, if two distinct

users are running PVM-based programs, then the PVM daemons are also rescheduled with respect to each other and the user-level processes.

The PVM daemon is a *helper* process that facilitates communication among processors. It continues to run after the nominal user job terminates, requiring a special implementation with respect to the priority at which the PVM daemon runs. When a PVM-based user job enters the system, Elmer and RSD also reschedule that user's PVM daemon. When the user job terminates, the PVM daemon is restored to its original priority level and scheduling class. This was required in the implementation for two reasons: 1) if the PVM daemon is not set to a FIFO scheduling class with a higher priority than the user's job, then it adversely affects the performance of the user job; 2) if the PVM daemon is not restored to its original priority and scheduling class when the user job terminates, then it adversely affects the performance of future jobs from the same user.

The operation of Elmer with RSD when a user job terminates works as follows: when Elmer detects that a user job has terminated, it runs the cluster deallocation algorithm, which restores that job's hosts to the list of available processors. It also sends a message to the RSD on each of that job's hosts, informing the RSD that the user job has terminated. RSD uses this information to update the priority of processes that still exist on its CPU.

6.2.1 User-Level Commands

User-level communication with Elmer and RSD is accomplished with a set of commands, including: **esubmit**, **ekill**, **ecancel**, and **equery**.

The **esubmit** command is used for submitting job requests to the Elmer facility. Its format is as follows:

```
esubmit [-p hostfile] [-P nprocs]
```

or

```
esubmit nprocs command
```

In the first command form, **esubmit** is used to submit PVM daemon requests to the system. Users may specify automatic generation of PVM hosts ("**-P nprocs**"), in which they include the number of processors they wish to use, or they may specify a default hostfile, which contains a list of hosts in the PVM-specified format. The second command form is the means by which the user submits a job to the system. The **esubmit** command requires the user to include the number of processors in their request so that Elmer may perform the required cluster allocation.

The **equery** command is used for requesting status information about the cluster. Its format is as follows:

```
equery [-r] [-q]
```

If no options are given to **equery**, then it reports the status of running and queued jobs in the cluster (that is, "**equery**" with no options is equivalent to "**equery -r -q**"). The two options to the command are used to report only running or only enqueued jobs. The "**-r**" option causes Elmer to report only the currently running jobs (that is, the current status of the cluster). The "**-q**" option causes Elmer to report only the enqueued jobs.

The **ecancel** and **ekill** commands are used to cancel queued jobs and to terminate running jobs, respectively. The format of the commands are as follows:

```
ecancel <job sequence number>
```

or

```
ekill <job sequence number>
```

6.2.2 Operation of Elmer

Figure 6.2 presents a logical diagram of the operation of the Elmer daemon. The primary program logic consists of an infinite loop operating on the queueing host. In this loop, the program accepts TCP connections from users executing the various Elmer commands (e.g., `esubmit`). When a connection is established, the main loop first forks a child process, which reads the first value sent, parsing it to determine what action is to be performed. The child then executes the appropriate procedure, while the parent continues to accept further incoming requests using an infinite loop. In the figure, the "Parse Message" action leads to one of the following: a PVM daemon initiation, a parallel job submission, a status query (queues and currently running processes), a queue remove request, or a job kill request.

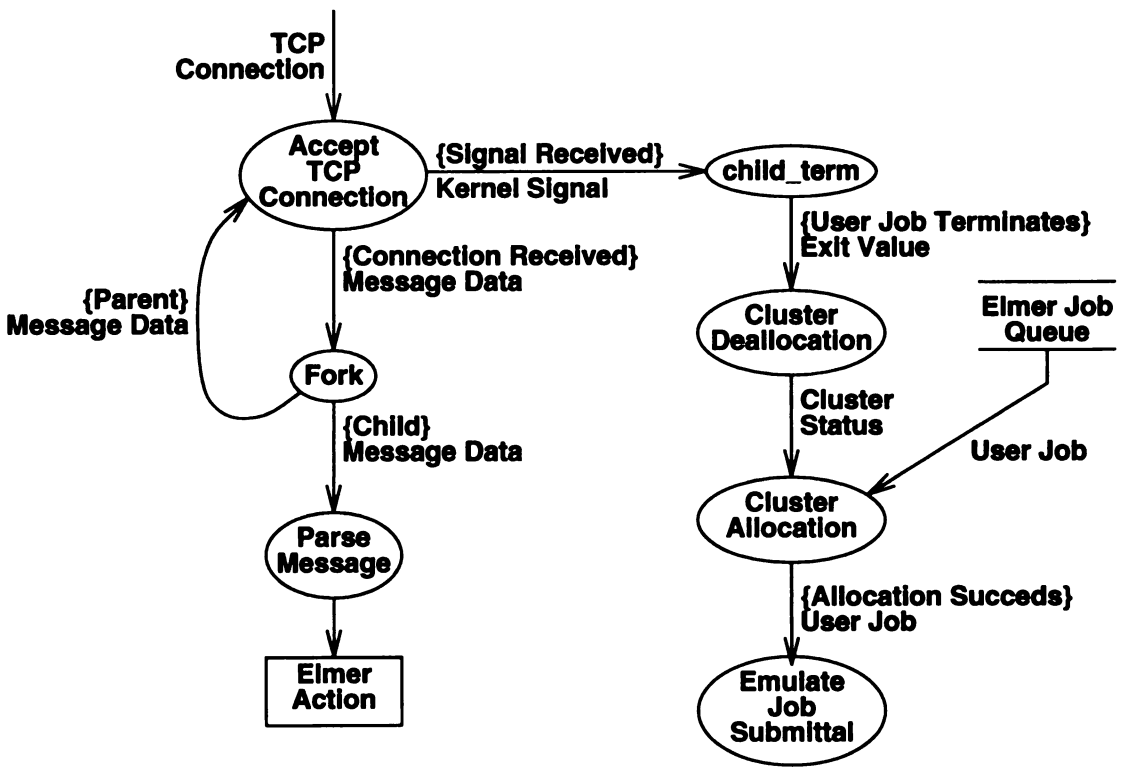

Figure 6.2: Logical view of the Elmer main program.

The process labeled **child_term** is a signal handling facility. When the Elmer daemon receives the `SIGCHILD` signal, which indicates that a child process of Elmer has terminated,

the **child_term** process is executed. Since all user jobs submitted to Elmer are executed by Elmer, it becomes their parent, enabling it to detect their termination. When the **child_term** process detects that a child process has terminated, it first checks to see whether the child process is actually a user job. It accomplishes this by reading the exit value given by the job (this exit value is discussed below in the description of the **elmer_submit** procedure). If child that terminated was a user job, then **child_term** uses the exit value to read a shared memory location that tells it what the job sequence number and former job priority were. These values allow **child_term** to run the cluster deallocation algorithm (represented by the "Cluster Deallocation" process in the figure), which notifies the RSDs from that job's cluster of the job's termination.

After cluster deallocation is performed, **child_term** checks the global queue for waiting user jobs. If any exist, it runs the cluster allocation algorithm to determine whether it may execute one (represented by the "Emulate Job Submittal" process in the figure). If the cluster allocation algorithm is successful, **child_term** forks a child process that runs a procedure (**emulate_elmer_submit**) that emulates the **elmer_submit** procedure by reading the enqueued job's data from a queueing directory and then executing the job.

Operation of the esubmit Command

Figure 6.3 presents a logical diagram of the actions performed when a user job is submitted to the Elmer daemon. Elmer's main program (Figure 6.2) receives a TCP connection with an initially-read action indicating a user job submission. It forks a child process, which parses the input and executes the **elmer_submit** procedure illustrated in Figure 6.3. If the data sent involves a user job submittal, then the procedure reads all relevant data sent by the **esubmit** command and runs the cluster allocation algorithm to determine whether the

job may be executed. This can only happen if: (1) there are no jobs previously enqueued, and (2) the cluster can satisfy the user job request.

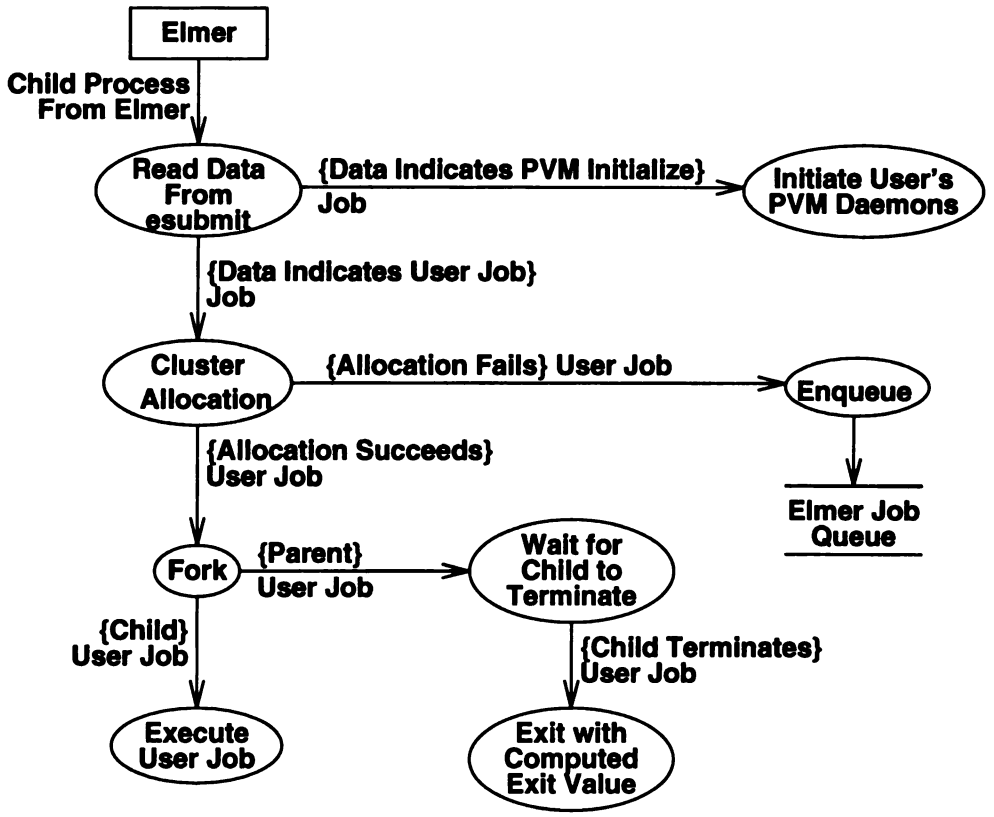

Figure 6.3: Logical view of **elmer_submit**.

If the job cannot currently be run, then the user's job is enqueued by storing the data received from `esubmit` in a file in Elmer's global queueing directory. If the job can be run, then another child process is forked, which executes it. The parent process of the new child waits for the child process to terminate. When the child terminates, the parent continues by computing an exit value, $e = j \bmod N$, where j is the user job's global job sequence number, and N is a prime number that determines the largest number of user jobs allowed in the system at once. It then exits with an exit value of $e + 100$ (100 is added to the exit value because the value of $j \bmod N$ could equal 0 or 1 for some jobs, and there are

Elmer processes that exit with values of 0 or 1, which the **child_term** signal handler should ignore).

The **esubmit** command is also used to initiate user-level PVM daemon processes. When the **esubmit** data is received, **elmer_submit** initially determines whether the job is a PVM initialization or a normal user job submittal. When **elmer_submit** initiates PVM daemons, one of two options are specified by the user: (1) the user supplies the PVM host file, and (2) the Elmer daemon automatically generates the PVM host file. When the user supplies the PVM host file, the location of the host file is simply sent by **esubmit** to the Elmer daemon, which then starts the PVM daemon with the host file passed as a parameter. If automatic host file generation is specified by the user, then the Elmer daemon chooses enough hosts to satisfy the user's request, creating the host file and executing the PVM daemon with that host file as a parameter.

Operation of the equery Command

Figure 6.4 provides a logical view of the actions performed by Elmer for the **equery** command. The procedure illustrated in this figure, **elmer_query**, simply checks the status of user jobs in the system. The **equery** command accepts two parameters, which tell it whether to query running user jobs, queued user jobs, or both (the default). The **elmer_query** procedure simply reads the contents of Elmer queue and/or its cluster and prints them out for the user.

Operation of the ecancel and ekill Commands

The **ecancel** command requires a simple procedure to be performed by the Elmer daemon. When the initial action requesting "cancel" is read via TCP, Elmer calls the **elmer_cancel**

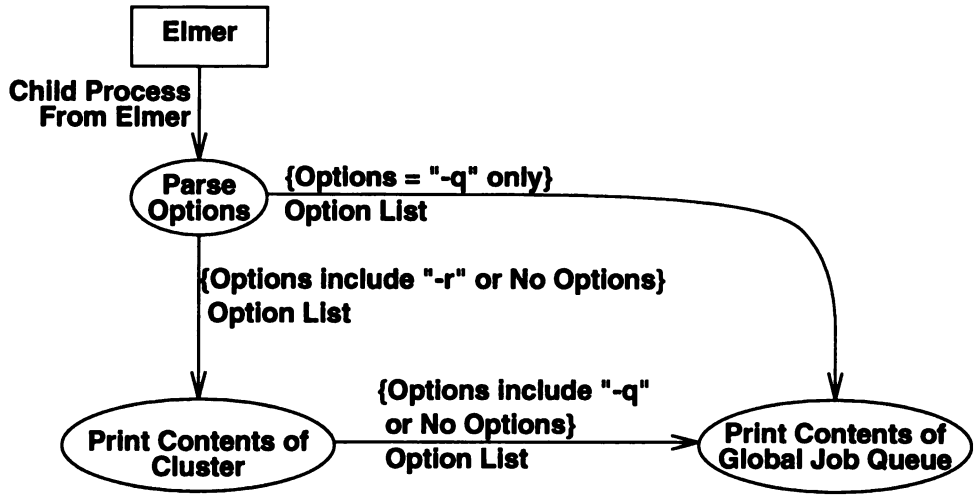

Figure 6.4: Logical view of **elmer_query**.

procedure. This procedure checks the queue for the job that is being canceled. If it does not exist within the queue, then an error message is printed. Otherwise, the job is removed from the queue, and its associated data file is removed.

The `ekill` command is somewhat more complicated, because it requires that the Elmer daemon communicate with every RSD running on hosts that involve the user job. Figure 6.5 illustrates the communication actions that Elmer must perform when a job is killed by its user. When the action is read in by the Elmer daemon, it calls the **elmer_kill** procedure, which reads the sequence number of the job to be killed from the `ekill` command. The **elmer_kill** procedure then examines the cluster data structure. If no active jobs in the cluster match the job sequence number, then an error message is reported back to the user. Otherwise, the host names within this job's cluster are recorded, and the cluster is updated to reflect that the user's job has terminated. The actual updating of the cluster data structures is accomplished by the **child_term** signal handler. When the parents of the former user job terminate, they will compute a special exit value when their child is killed by Elmer. The **child_term** signal handler will read that exit value and proceed as defined in Figure 6.2.

When the hosts involving this user's job are identified, the **elmer_kill** procedure will open a TCP connection to the RSD at each of these hosts. This connection allows it to send relevant job data to the individual RSDs, which actually kill the executing jobs.

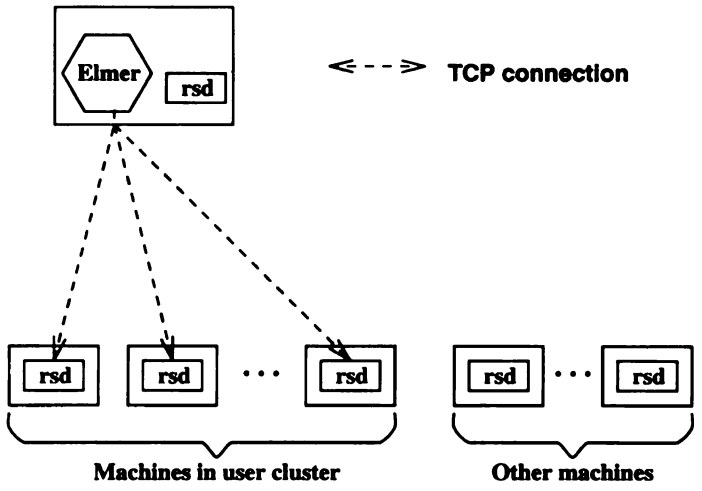

Figure 6.5: Communication involved in **elmer_kill**.

6.2.3 Operation of RSD

Figure 6.6 presents a logical diagram of the operation of RSD. When RSD is initiated, it first spawns a sub-process called **util**, which implements an infinite loop to continually update a region of shared memory (every few seconds) with the current CPU utilization. The primary program logic of RSD consists of a single infinite loop. As in the Elmer daemon, the main loop accepts TCP connections, although it only communicates with the Elmer daemon. When a connection is established, the main loop first forks a child, which reads the first value sent to determine what action is to be performed. The child then executes the appropriate procedure, while the parent continues in its infinite loop to accept further connections from the Elmer daemon. In the figure, "RSD Action" is one of the following: a *job initiation* (reschedule an incoming job), a *job kill* (terminate an executing job), a

CPU report (report current CPU utilization), and a *job termination* (an executing job has finished).

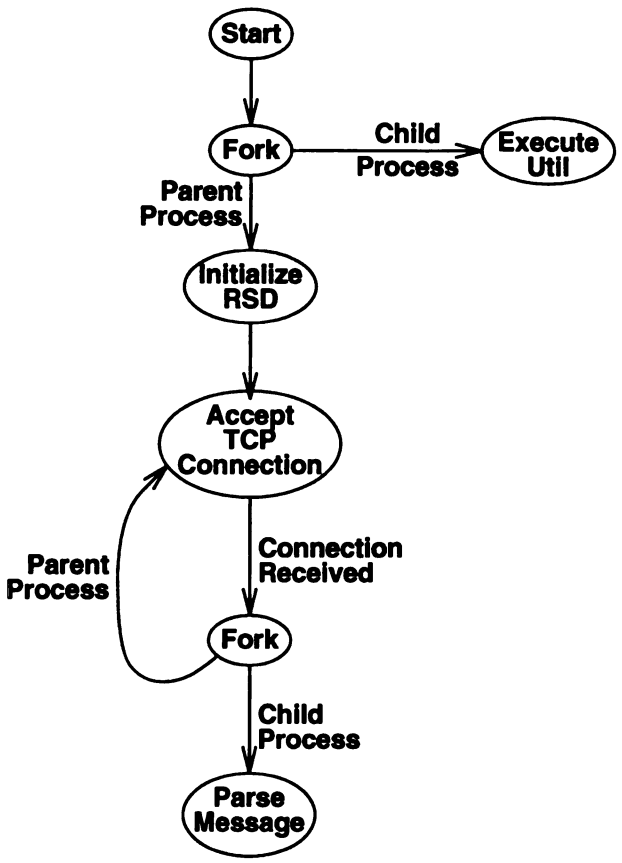

Figure 6.6: Logical view of the RSD main program.

A job initiation action occurs when Elmer allocates the local processor to be part of an incoming job's cluster. When the **elmer_submit** or **child_term** procedures successfully receive a cluster from the cluster allocation algorithm, a child process is forked that executes the request. The parent of that child process then calls a procedure called **notify_alloc**, which opens a TCP connection with the RSD running on each host in the cluster. When the connection is established, **notify_alloc** communicates the job's priority level, its UID, and its global job sequence number to each of the RSDs. The RSD uses this information to

establish the job's priority, issuing system calls to reset the job's priority and/or scheduling policy.

When Elmer's **child_term** procedure detects that one of its jobs has finished execution, it calls the **notify_dealloc** procedure, which performs actions similar to those performed by **notify_alloc**. That is, **notify_dealloc** opens a TCP connection with the RSD running on each host in the former job's cluster. When the connection is established, it sends the job's former priority level, UID, and global job sequence number to each of the RSDs. Each RSD uses this information to update its local scheduling information, which involves the updating of the scheduling policies and/or priorities of jobs currently executing on the local processor. If any lower-priority jobs exist locally when the RSD receives notification of a deallocation, then the priority of those jobs is increased so that they receive more favorable scheduling. This action also serves to enforce the policy that the earliest jobs to arrive are the highest priority. That is, RSD and Elmer ensure that no new jobs can arrive to take the departing job's high-priority spot.

When a user issues the **ekill** command, they must supply Elmer with a job's global sequence number. Elmer interacts with all RSDs in that job's cluster by supplying the job's global sequence number. RSD then terminates the job using the **kill** system call.

When the user issues the **ecancel** command, they must supply Elmer with a job's global sequence number. For the **ecancel** command, however, no interaction with RSD is required. Elmer simply removes the job from its queue.

When the cluster allocation algorithm is executed by Elmer, it may or may not use the current CPU utilization from individual CPUs in the system, depending on its current implementation. When the cluster allocation algorithm requires the CPU utilization from individual processors, it sends a request to the RSD on each processor under consideration.

Each RSD reads a shared memory location (updated every few seconds by the **util** process) and sends a reply back to Elmer.

6.3 Cluster Allocation Algorithms

In a NoW-based CPC in which the allowable *degree of multiprogramming* (number of user jobs allowable on any host at once) is one, the cluster data structure might consist of a one-dimensional array, with each array element representing a host in the NoW. In Elmer, which considers timeshared scheduling, one can think of the cluster as a set of *virtual clusters*, with each virtual cluster representing a priority level and/or one of the degrees of multiprogramming allowed. Figure 6.7 illustrates the structure of the cluster as maintained by Elmer. A virtual cluster is maintained for D degrees of multiprogramming (priority levels range from 0 to $D-1$, with level 0 indicating the highest priority). If P hosts exist within the system, the cluster is essentially a $D \times P$ array of cluster entries. In order to uniquely identify different jobs, each cluster entry consists of a job's global sequence number and a job UID.

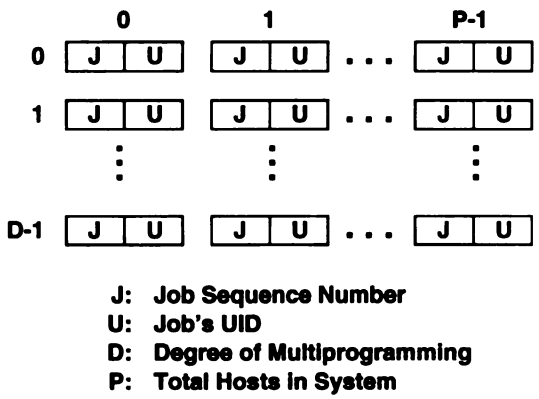

Figure 6.7: Format of the cluster data structure.

Two primary cluster allocation algorithms are being considered for the Elmer facility. The first (most naive) version is illustrated in Figure 6.8. This algorithm relies on no

information from the individual RSDs. Rather, it considers only the current degree of multiprogramming at each host, as well as a global allowable degree of multiprogramming, D, which is determined in advance. It examines the cluster data structure (Figure 6.7) at each priority level. If the values of J and U are -1 for a particular host at a particular priority level, then it is considered *free* at that priority level.

Version 1 of the cluster allocation algorithm examines the cluster at each priority level until it finds a cluster that satisfies the request or determines that no cluster can be allocated. For a job request of n hosts, if there are n or more free hosts at priority level i, then the request may be satisfied at that priority level. Since it is most desirable for a job to have the highest possible priority, the search proceeds from 0 to $D-1$. If the algorithm finds no priority level in which n hosts are available, then it returns null to indicate that no cluster can be allocated. If $D = 1$, then this algorithm is equivalent to allowing only *space shared* scheduling, in which only one user job occupies a processor at any given time (that is, the user job is given exclusive access to its cluster).

Version 1 of the cluster allocation algorithm is conceptually simple, and it has a low time complexity. Step (1) is a simple assignment that takes constant time. The While loop of step (2) considers D levels of priority. Substep **(a)** requires a simple **For** loop that considers up to P hosts. Substep **(b)** is a constant time operation. Thus, step (1) requires $O(DP)$ time. For the **Else** portion of step (2), a loop of up to P hosts is required to select the host list at priority j, marking each of the selected hosts as allocated. Thus, step (2) is $O(P)$. If we consider that D is a constant selected at run time, then the complexity of version 1 is then $O(P)$.

The second version of the cluster allocation algorithm, illustrated in Figure 6.9, is similar to the first version, except that it uses CPU utilization gathered from individual RSDs as

Algorithm: *Cluster Allocation, v1.*

Input: A cluster request consisting of its number of processors, n.

Output: A list of hosts and an integer priority level if the request is granted (NULL if not).

Procedure:

(1) Let $i = 0$ and $j = -1$
  Let D = Degree of multiprogramming
  Let P = total hosts in system.
(2) While $(i < D)$ and $(j = -1)$
  (a) Let c = the number of hosts with priority slot i free.
  (b) If $(c \geq n)$ then $j = i$, the priority level to allocate.
    Else, add 1 to i.
(3) If $(j = -1)$ then return NULL (no allocation).
  Else
    Randomly select n free hosts at priority j.
    Mark each of those hosts as allocated.
    Return the host list and the priority value.

Figure 6.8: The cluster allocation algorithm, version 1.

extra information when making cluster allocation decisions. In this algorithm, D is still determined in advance, but the utilization levels at each host can be used to override the value of D at each host. The primary difference between the two algorithms is that the second version collects the current utilization from every host and uses that information as an additional criteria in determining whether a host is free or unallocatable for a particular priority level.

If the overhead of computing CPU utilization at each host is ignored, the second version of the cluster allocation algorithm has the same time complexity as the first version. This assumption is realistic, because version 2 only requires that a message be sent to each host inquiring about the current CPU utilization. Step 3 (a) only requires an extra **IF** comparison for each host in the loop, as compared to step 2 (a) in version 1.

The second version of the cluster allocation algorithm illustrates a number of issues involved in cluster allocation in this environment. Version 2 can be expected to be more

Algorithm: *Cluster Allocation, v2.*

Input: A cluster request consisting of its number of processors, n.

Output: A list of hosts and an integer priority level if the request is granted (NULL if not).

Procedure:

(1) Let $i = 0$ and $j = -1$
  Let D = Degree of multiprogramming
  Let P = total hosts in system.
  Let t = the maximum CPU utilization threshold $(0 < t < 1)$.

(2) For each host H_i Do
  Collect the current CPU utilization, u_i, at H_i.

(3) While $(i < D)$ and $(j = -1)$
  (a) Let c = the number of hosts with priority slot i free for which $u_i \leq t$.
  (b) If $(c \geq n)$ then $j = i$, the priority level to allocate.
    Else, add 1 to i.

(4) If $(j = -1)$ then return NULL (no allocation).
  Else
    Randomly select n free hosts at priority j.
    Mark each of those hosts as allocated.
    Return the host list and the priority value.

Figure 6.9: The cluster allocation algorithm, version 2.

effective than version 1 because different parallel jobs use varying percentages of the total available CPU time. If a job executing at the highest priority level uses most of the CPU time on all of the hosts in its cluster, then another lower-priority job should not be executed in that cluster, since the lower priority job will receive little or no CPU time until the high-priority job is finished executing. In this case, space sharing is a more effective scheduling policy, because it provides the potential that the newly-arrived job will be scheduled earlier, if another cluster is made available.

The algorithm also illustrates the variable definition of the term "free" as it applies to a host for a particular priority level. In the naive cluster allocation algorithm (version 1), a host is free for a priority level if that level has not been allocated. In version 2, the host is only free if that level has not been allocated and if sufficient CPU resources are available.

It also reveals other issues involving the degree of multiprogramming, D. A workstation with a CPU-bound job is likely to have a maximum local degree of multiprogramming, d, equal to 1. However, if a workstation contains several highly communication-bound jobs, its d value could be greater than the global degree of multiprogramming, D. Version 2 of the cluster algorithm does not take this issue into consideration. That is, on every host, the value of d is always less than or equal to D.

Yet another issue not considered by version 2 is whether to allow a lower-priority cluster to span multiple higher-priority clusters. As it is presented, version 2 allows the creation of a lower-priority cluster that may consist of hosts containing multiple separate, independent, high-priority jobs. Our prior research in [44] suggested that this may be detrimental to both user JTT and system throughput.

6.4 Evaluation of Elmer's Performance

Figure 6.10 illustrates the overhead, in milliseconds, that Elmer uses when a job initiation occurs. The figure presents the measure of the time it takes for Elmer to accept the incoming job, allocate a cluster (using version 1 of the cluster allocation algorithm), and notify all of the RSDs in the cluster of the incoming job. The figure shows that our implementation requires an overhead ranging from a low of about 59 milliseconds (on one processor) to a high of about 103 milliseconds (on all 6 processors). By interpolating the results, it can be observed that the performance shown is sufficient for Elmer to handle clusters with up to 107 processors, if a maximum request rate of 1 per second is allowed. This figure is sufficient for a maximum desired cluster size of 100 processors.

Figure 6.11 illustrates the overhead, in milliseconds, that Elmer uses when a job termination occurs. The figure presents the measure of the time it takes when Elmer performs

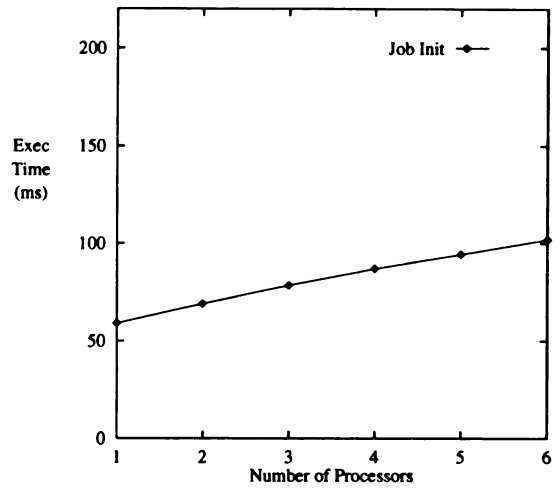

Figure 6.10: Overhead (ms) for job initiation.

several tasks upon detecting a user job termination. The tasks include performing the cluster deallocation algorithm, including updating the cluster status, and notifying the individual RSDs so that they can update their internal data structures. The figure shows that our implementation requires an overhead ranging from a low of about 13 milliseconds on one processor, to a high of about 57 milliseconds on 6 processors. This overhead is much lower than that of job initiation, due mainly to the fact that the job initiation functions of Elmer perform more tasks and require a number of more expensive system calls. Again, the overhead of job termination in Elmer is sufficient for the maximum desired cluster size of 100 processors.

6.5 Experimental Results

This section describes the experiments involving the Elmer facility. It presents a comparison of a set of timesharing experiments run under both realtime (RT) and non-RT operating system kernels. The set of experiments run under the non-RT kernel were originally presented in [44]. These experiments present the performance of timeshared jobs in which

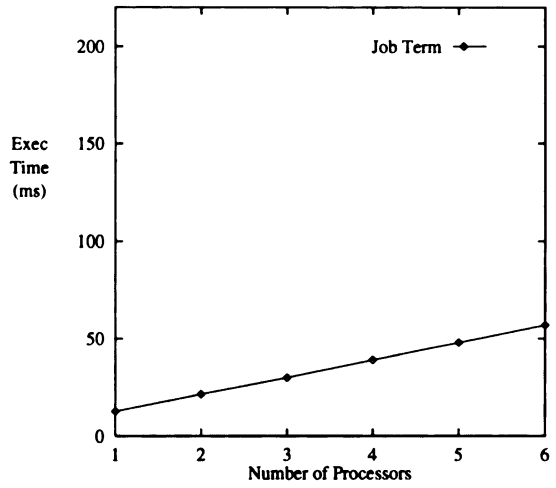

Figure 6.11: Overhead (ms) for job Termination.

non-preemptive scheduling methods were employed. The experiments run under the RT kernel present the performance of timeshared jobs in which preemptive scheduling methods were employed.

For the results presented in this section, all "SS" JTT figures represent the JTT of each job under space-sharing conditions, in which the high-priority job was run first to completion, followed by the low-priority job. All "TS" JTT figures represent the JTT of each job when both were submitted at the same time. JTT is represented as execution time in the figures.

Figure 6.12 presents the experimental format of the first set of experiments, in which Barrier (v1, v2, and v3, respectively) ran as the high-priority job, with EP running as the low priority job.

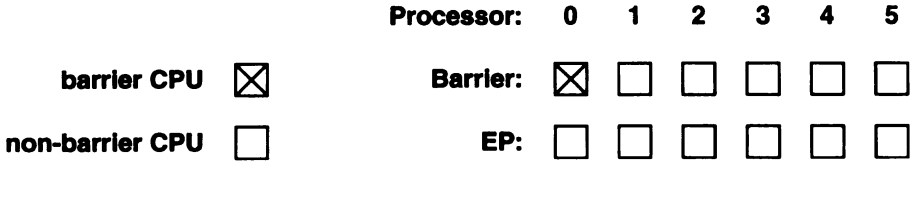

Figure 6.12: Experimental format, Barrier running with EP.

Figures 6.13 through 6.15 present the results of those experiments. In each case, a measurable improvement in overall finishing time was observed. The improvement is illustrated by the fact that the execution time of the low-priority EP job, when run under timesharing conditions, improved with respect to the space sharing case. Part of the observed improvement was likely due to the EP program's self load-balancing feature. The CPU containing the barrier process was given less work by the EP algorithm, allowing Barrier's execution to proceed relatively unhindered on that processor.

Of more interest, however, is the comparison of the non-RT experiments to similar experiments run using the RT kernel. In the non-RT experiments, the JTT of EP improved significantly over the space-sharing case, but the Barrier program was penalized somewhat in its JTT. On 1 or 2 processors, there was little benefit to decreasing EP's priority relative to Barrier, since Barrier incurred little or no communication overhead. Furthermore, in the non-RT case, significant degradation of the performance of the high-priority Barrier job occurred due to the system's scheduling algorithms, which allowed EP to steal CPU cycles from the Barrier job. When communication overhead became more significant (at 4 to 6 processors), the execution time of Barrier, with EP at a lower priority, approached the execution time that it obtained using space-sharing scheduling. The use of the preemptive scheduling facilities, however, allowed the high-priority Barrier program to execute whenever it was ready, achieving an execution time that was essentially identical to its space-sharing execution time for any number of processors.

Figure 6.16 presents the experimental format of the second set of experiments, in which different versions of Barrier were run together in the cluster. In this set of experiments, the barrier process of each job shared the same CPU.

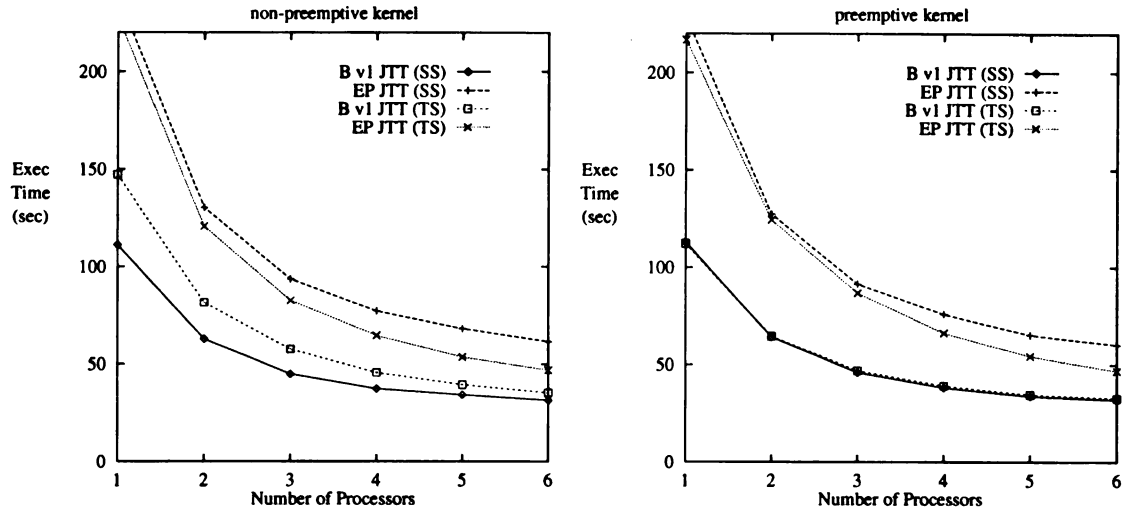

Figure 6.13: Job turnaround time for B (v1) and EP.

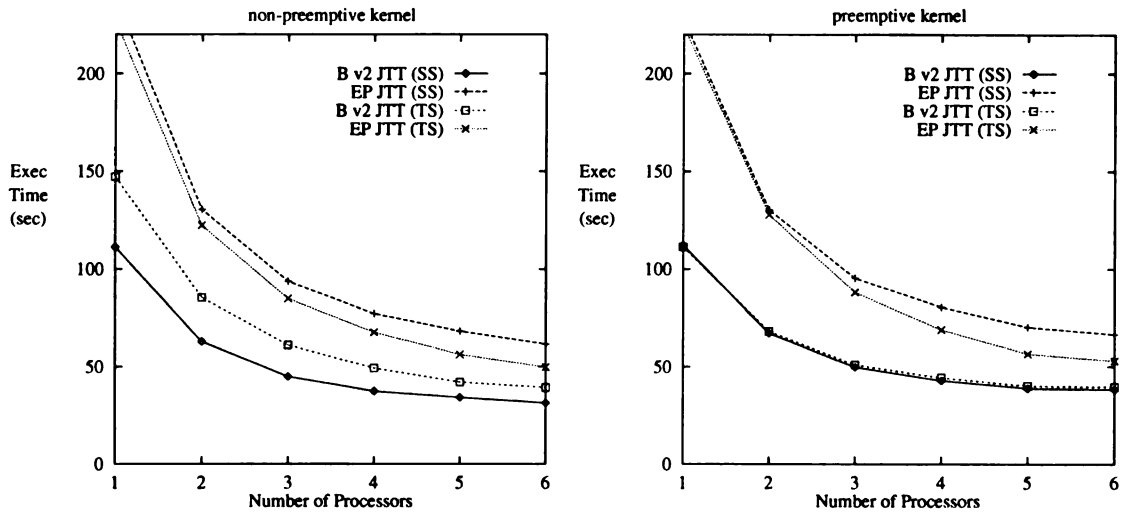

Figure 6.14: Job turnaround time for B (v2) and EP.

Figures 6.17 through 6.19 present the results of those experiments. In each case, the high-priority version of barrier, running on the RT kernel, achieved an execution time identical to its execution time under space-sharing conditions. Running on the non-RT kernel, the high-priority job was again penalized somewhat in its execution time. In this case, the penalty is explained by the fact that the barrier process of both programs shared the same CPU. As illustrated in Figure 5.2, the Barrier program's barrier process (executing on processor 0 in the figure) has a high CPU demand for each program. This high CPU

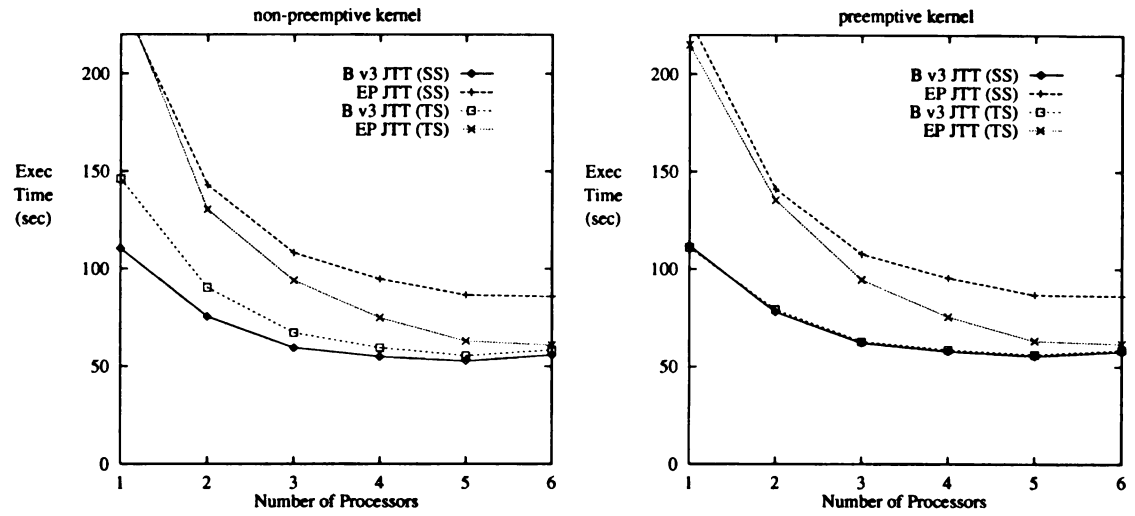

Figure 6.15: Job turnaround time for B (v3) and EP.

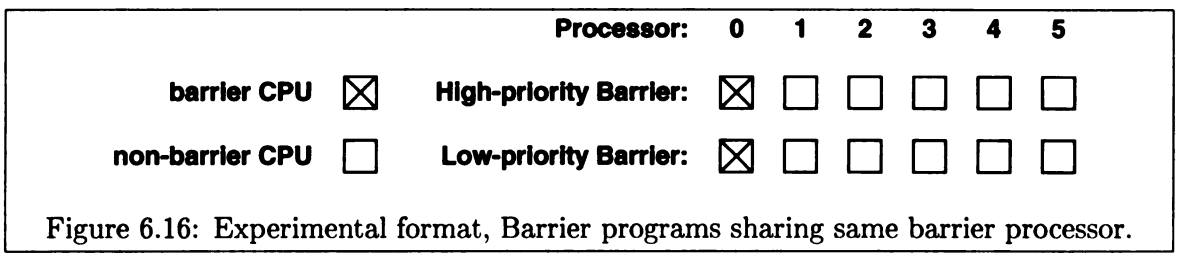

Figure 6.16: Experimental format, Barrier programs sharing same barrier processor.

demand results in contention for CPU resources by both jobs, and the low-priority job is able to steal CPU cycles as a result.

The high CPU demand of each job's barrier process also explains the very low improvement in throughput, as shown by the finishing time of the low-priority job. Since the high-priority job requires most of the cycles on the CPU containing its barrier process, it prevents the low-priority job from obtaining enough of the otherwise idle CPU cycles in the cluster, resulting in the low observed improvement in throughput.

The results of the previous experiments suggested the third set of experiments, as illustrated in Figure 6.20. Here, the barrier process of each job shared its CPU with a non-barrier process of the other job.

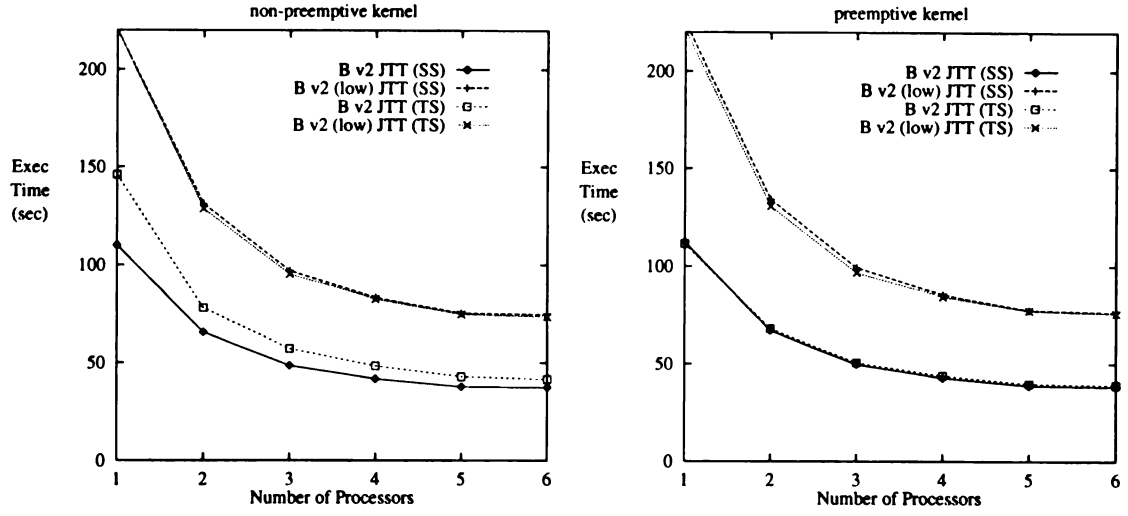

Figure 6.17: Job turnaround time for B (v2) and B (v2), barrier processors same.

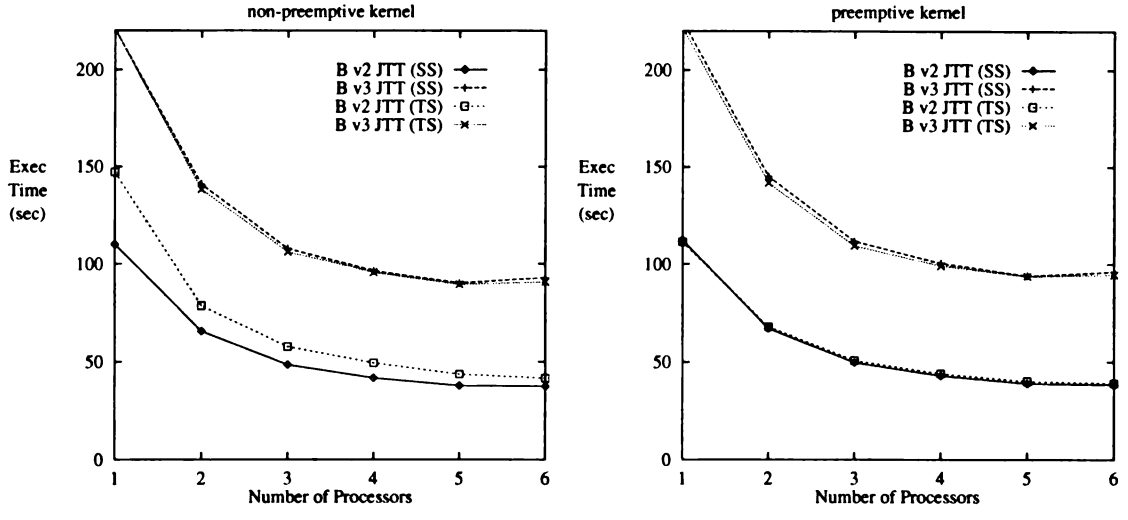

Figure 6.18: Job turnaround time for B (v2) and B (v3), barrier processors same.

This set of experiments further illustrated the effects of the high CPU demand of the barrier process of each job. In this case, although a non-barrier process of the low-priority job occupied the same CPU as the barrier process of the high-priority job, the high CPU demand of the high-priority barrier process still prevented the low-priority job from obtaining significant CPU time. For the non-RT experiments, the high-priority job achieved slightly better performance than the case in which two barrier processes shared the same CPU. However, the RT experiments still achieved the highest performance for the high-priority job, as expected.

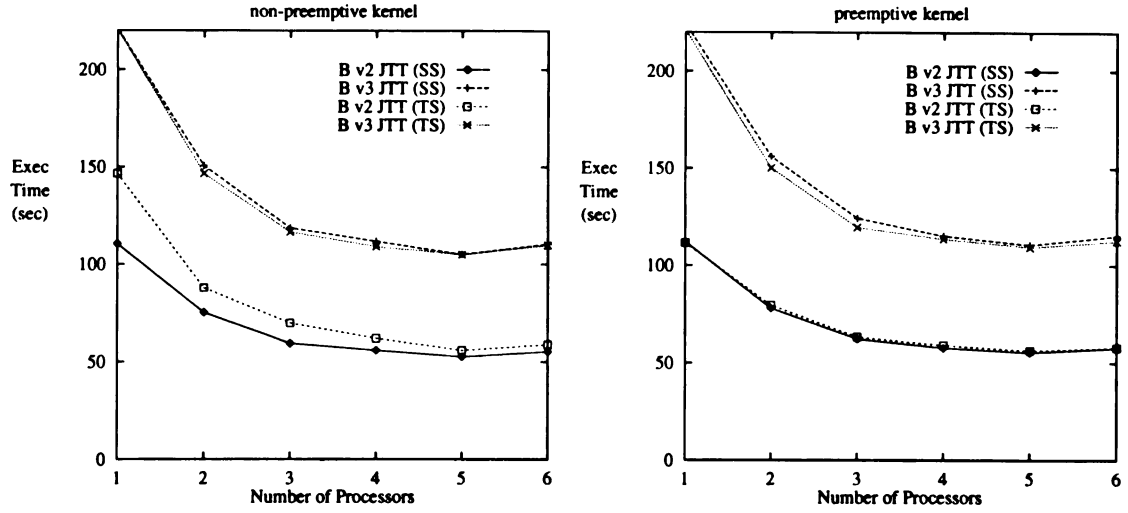

Figure 6.19: Job turnaround time for B (v3) and B (v3), barrier processors same.

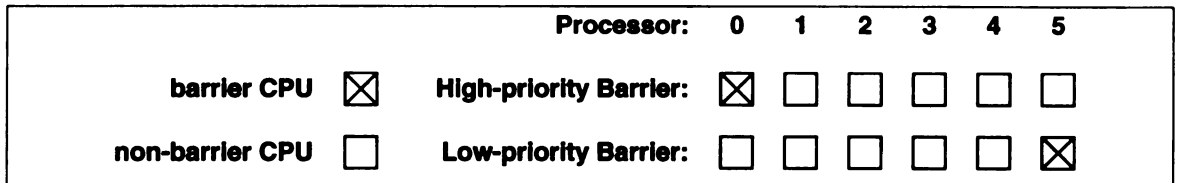

Figure 6.20: Experimental format, Barrier programs, barrier process shares CPU with non-barrier process.

The prior results provided motivation for the examination of more advanced cluster allocation techniques. Up to this point, all experiments had given the same set of processors to both jobs. Figure 6.23 illustrates the fourth variation on the experiments, in which the barrier process of each job was given exclusive access to its CPU. In this experiment, the programs were run on partially intersecting clusters, in which the barrier process of each program had exclusive access to a CPU (hence, the experiment could be run only for cluster sizes of 1 to 5 processors, and for a cluster size of 1, the programs used the same processor).

When each job was given exclusive access to the CPU implementing its barrier process, a significant improvement in overall JTT (hence, throughput) was observed, as illustrated in Figure 6.24. Furthermore, the RT-based experiment achieved the best-possible perfor-

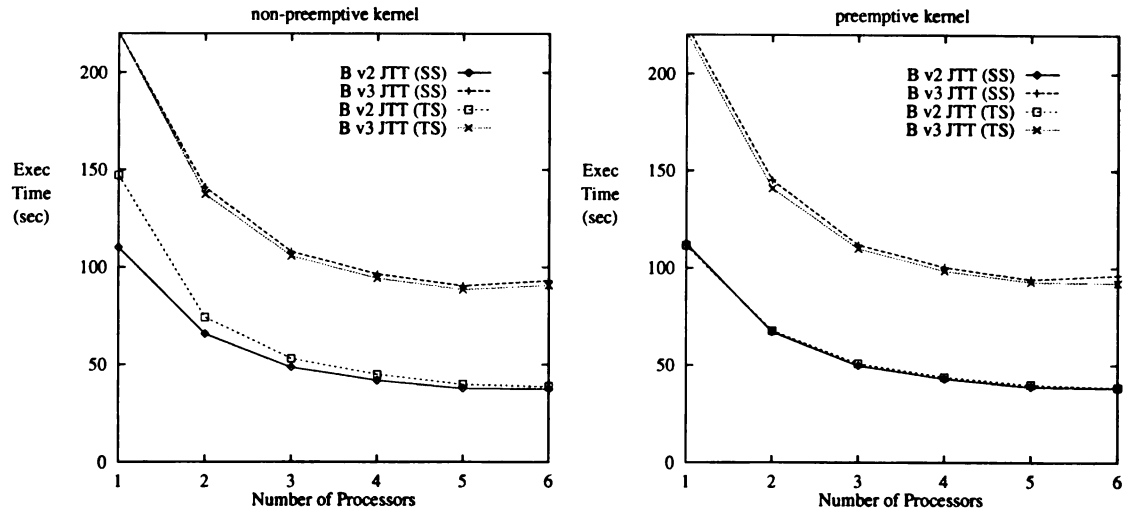

Figure 6.21: Job turnaround time for B (v2) and B (v3), barrier processors different.

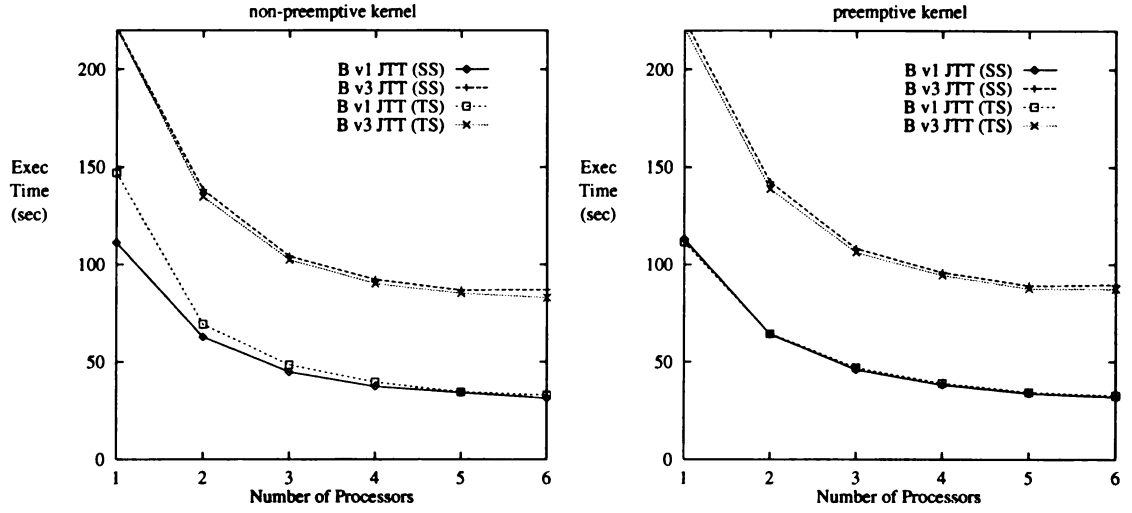

Figure 6.22: Job turnaround time for B (v1) and B (v3), barrier processors different.

mance for the high-priority job, while it allowed the low-priority job to achieve a significant improvement in its overall JTT.

Figure 6.25 illustrates the final variation on the experiments, in which the low-priority job was given a subset cluster of the high-priority job's cluster. In this experiment, the high-priority job's barrier process had exclusive use of its CPU, while the low-priority job's barrier process had to occupy a CPU from the high-priority job's cluster. In each case, the high-priority job ran using 6 processors, while the low-priority job varied from 1 to

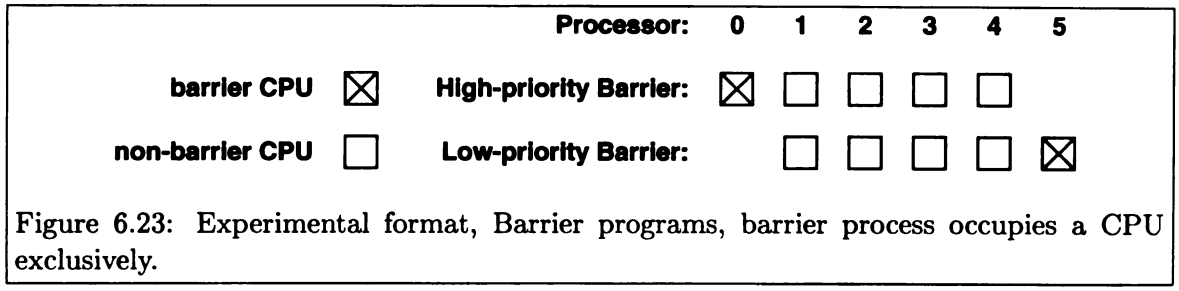

Figure 6.23: Experimental format, Barrier programs, barrier process occupies a CPU exclusively.

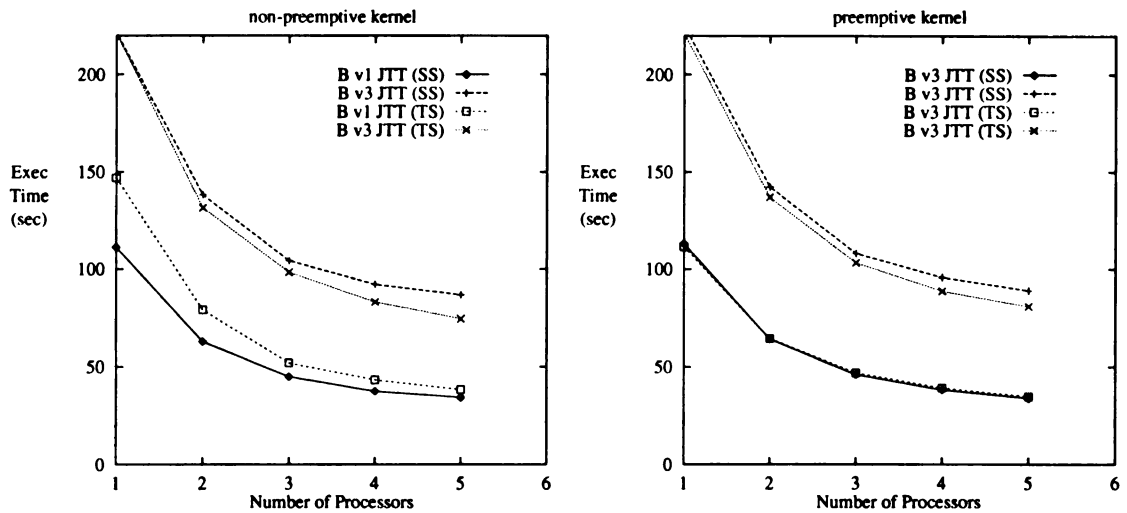

Figure 6.24: Job turnaround time for B (v1) and B (v3), partially intersecting clusters.

5 processors (again, to preserve the exclusivity of the high-priority barrier process, the low-priority job could only use up to 5 processors).

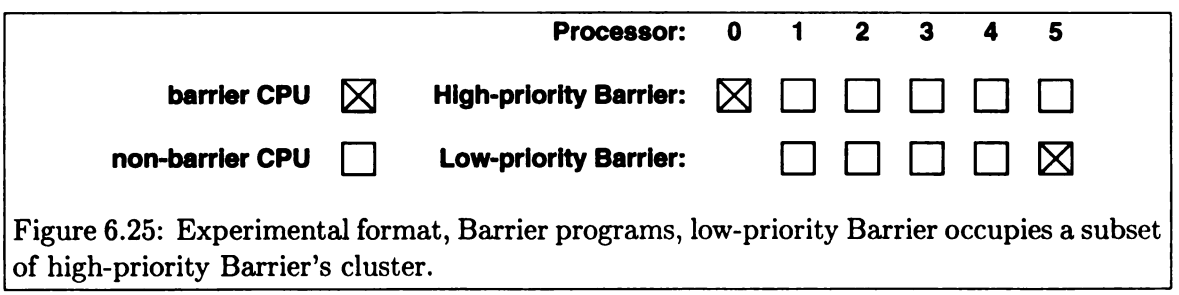

Figure 6.25: Experimental format, Barrier programs, low-priority Barrier occupies a subset of high-priority Barrier's cluster.

Figure 6.26 presents the results of that experiment. In both the non-RT and RT cases, the high-priority job's JTT was unaffected, while the low-priority job achieved an improvement in throughput by finishing ahead of the space-sharing case.

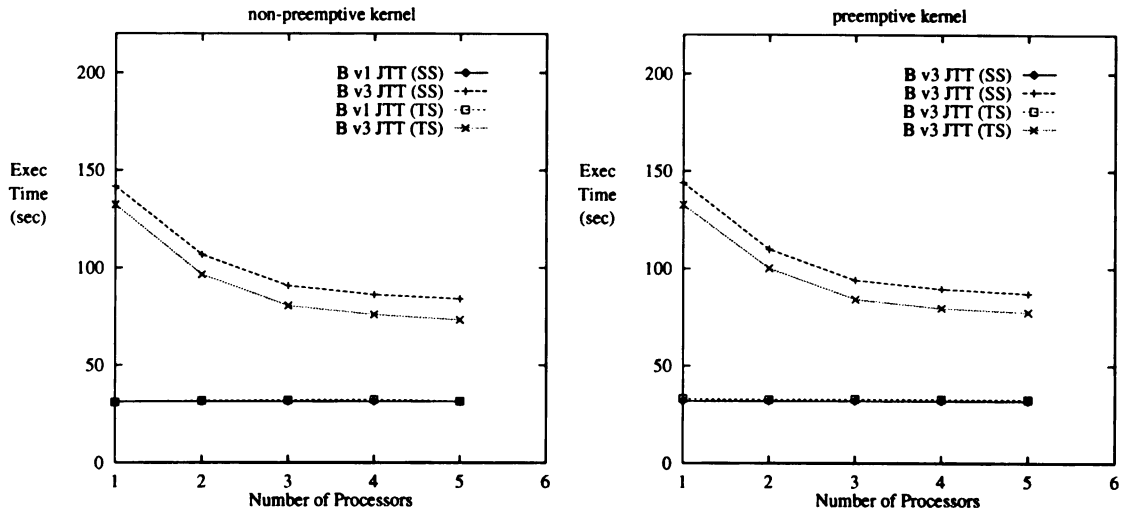

Figure 6.26: Job turnaround time for B (v1 and v3), v3 having a subset cluster.

These experiments have shown that the primary goal of not affecting the high-priority job's JTT can be (and has been) accomplished. They also suggest that more advanced cluster allocation techniques are required to ensure the secondary goal of higher throughput. For example, the experiments presented in Figures 6.17 through 6.22 illustrate that while the high-priority job may be unaffected, the low-priority job may gain nothing if an ineffective cluster allocation is made. Figures 6.24 and 6.26 indicate that, for barrier programs, the CPU on which a barrier process exists should not be scheduled with other jobs.

Chapter 7

Conclusion

This thesis has described advanced processor scheduling methods for two classes of distributed memory parallel computers. The techniques described here have been shown to decrease job turnaround time and increase system throughput. Furthermore, they have been shown to be practical techniques that are easily integrated into existing systems. This chapter presents a brief summary of the two major aspects of our research, followed by our conclusions, as well as directions for future work.

7.1 Hypercubes

There are two primary conclusions that may be reached from the study of the AFL technique for cluster allocation in hypercubes. First, using the AFL with 2-D mesh requests generally increases system throughput and decreases average user JTT. When the AFL method is used in conjunction with the FL algorithm, jobs in allocated clusters suffer no communication interference from other jobs, yet they are not required to be a perfect subcube. The AFL provides an additional benefit to jobs that would typically suffer severe performance degradation due to having too many allocated processors. Second, the geometry of the mul-

tiprocessor architecture is not necessarily a restriction on the cluster allocation method. By using 2-D mesh allocation within subcubes, the AFL method is shown not only to increase system performance, but to provide the potential to increase individual job performance, as well.

The AFL algorithm also has practical benefits: it may be used in conjunction with any existing subcube allocation algorithm. Therefore, it may be implemented on many existing installed systems that may not use the FL method of subcube allocation.

7.2 Networks of Workstations

Our initial feasibility study into timesharing techniques for user jobs in workstation clusters indicated that the prioritized execution of parallel jobs can improve system throughput without significant penalty to the observed user JTT. Our analytical models suggested that user code could be modified to reset a job's priority with respect to other jobs. However, one of the primary drawbacks to this technique was its requirement of user participation to realize an improvement in system throughput. The study also revealed that preemptive scheduling methods would be required so that the execution time of the high-priority job was unaffected.

The feasibility study provided the motivation for the creation of the Elmer clustering and scheduling facility. Elmer has allowed us to study experimental clustering and scheduling methods in NoW-based CPCs. Furthermore, it provides a simpler user interface for the prioritized execution of high-performance jobs. The Elmer facility is uniquely specialized to take advantage of the timeshared execution of parallel jobs. The cluster allocation algorithms used within Elmer were also developed for this purpose.

The performance measurements obtained from the use of the Elmer facility allowed us to verify that our cluster allocation algorithms provide a higher system throughput than pure space sharing techniques.

7.3 Contributions of this Thesis

In a high-performance computing environment in which standard processor scheduling techniques are used, parallel applications are rarely able to fully utilize the hardware resources over which they have control. Standard scheduling techniques can result in two forms of under-utilization of system resources. First, inefficient cluster allocation can result in unused processors within a cluster. Second, space sharing of parallel jobs can result in underutilization of CPUs due to the communication latency present in clusters of workstations. The research presented in this thesis addresses and solves these problems using unique, yet practical, methods.

Our research in hypercube cluster allocation has demonstrated a means to more effectively allocate clusters of processors in the hypercube architecture. Unneeded CPU resources, in the form of over-allocated clusters of processors, can contribute to decreased performance of individual parallel jobs due to communication overhead. Our cluster allocation technique, the auxiliary free list method, demonstrated improved performance over the free list subcube allocation method in two respects. First, it improved system throughput and decreased average user JTT. Second, it enabled specific user jobs to utilize only the resources necessary to achieve their optimum performance, eliminating the over-allocation of processors that can result in decreased performance for the specific job. Furthermore, our research has demonstrated that it is possible to allocate clusters of a specific size in hypercubes, eliminating the requirement of using subcube allocation.

Our research in scheduling for workstation clusters has demonstrated a means by which processor sharing can be employed to improve system throughput and individual CPU utilization without penalty to the JTT of user jobs. We first showed that timesharing of parallel jobs can be employed to improve system throughput. We then demonstrated a means by which preemptive scheduling can be employed so that the execution time of higher-priority jobs are not affected by the presence of lower-priority jobs. We have also developed a practical facility that uses realtime scheduling mechanisms to implement this prioritized scheduling technique. Our implementation also demonstrates that the use of preemptive scheduling techniques can be accomplished with little or no modifications to the operating system.

7.4 Future Work

There are a number of issues yet to be addressed, specifically involving the allocation of clusters in a NoW-based CPC. Our research has presented a means by which processor sharing may take place in a CPC. However, our experimental facility has limited the scope of our experiments. First, the number of machines is limited to 6, which prevented us from examining issues involving cluster allocation when a larger number of processors is available. Furthermore, we were unable to examine issues concerning the scalability of the Elmer facility. Second, our experiments did not examine the impact of memory usage on the allowable degree of multiprogramming at each node. Third, we presented two possible cluster allocation algorithms for NoW-based CPCs. Our preliminary results of experiments testing the throughput obtainable with these cluster allocation algorithms suggests that further investigations into cluster allocation, as well as job scheduling, are merited.

Bibliography

Bibliography

[1] T.-H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 87–98, January 1993.

[2] J. Mario J. Gonzalez, "Deterministic processor scheduling," *ACM Computing Surveys*, vol. 9, pp. 173–204, September 1977.

[3] S. W. Turner, L. M. Ni, and B. H. C. Cheng, "Contention-free 2d-mesh cluster allocation in hypercubes," *IEEE Transactions on Computers*, To appear.

[4] M. W. Mutka and M. Livny, "Scheduling remote processing capacity in a workstation-processor bank network," in *Proceedings of the 7th International Conference on Distributed Computing Systems*, (Berlin, West Germany), pp. 2–9, September 1987.

[5] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, (San Jose, California), pp. 104–111, June 1988.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3 user's guide and reference manual," Tech. Rep. ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1993.

[7] L. M. Ni, *Parallel Computer Architecture and Programming.* East Lansing, MI: Unpublished Manuscript, 1993.

[8] Kendall Square Research, *KSR1 Technical Summary*, 1993.

[9] BBN Advanced Computers Inc., Cambridge, Massachusetts, *Inside the GP1000*, 1989.

[10] BBN Advanced Computers Inc., Cambridge, Massachusetts, *Inside the TC2000 Computer*, 1990.

[11] B. Duzett and R. Buck, "An overview of the nCUBE 3 supercomputer," in *Proceedings of the Frontiers '92*, pp. 458–464, IEEE, 1992.

[12] W. J. Dally, J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler, "The message-driven processor: A multicomputer processing node with efficient mechanisms," *IEEE Micro*, pp. 23–39, 1992.

[13] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, pp. 892–901, October 1985.

[14] G. F. Pfister *et al.*, "An introduction to the IBM research parallel processor prototype (RP3)," in *Experimental Parallel Computing Architectures* (J. J. Dongarra, Ed.), pp. 123 - 140, Elsevier Science Publishers B.V., Amsterdam, 1987.

[15] A. Gottlieb, "An overview of the NYU ultracomputer project," in *Experimental Parallel Computing Architectures* (J. Dongarra, Ed.), pp. 25 - 95, North Holland, 1987.

[16] R. Arlauskas, "iPSC/2 System: A second generation hypercube," in *Proceedings of the Third Hypercube Conference*, (Pasadena, CA), pp. 38–42, Association for Computing Machinery, Jan. 1988.

[17] nCUBE Corporation, *nCUBE 6400 Programmer's Guide.* Beaverton, Oregon, 1990.

[18] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM*, vol. 28, pp. 22–33, Jan. 1985.

[19] K. Li and K.-H. Cheng, "A two-dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system," *Journal of Parallel and Distributed Computing*, vol. 12, pp. 79–83, 1991.

[20] P.-J. Chuang and N.-F. Tzeng, "An efficient submesh allocation strategy for mesh computer systems," in *Proceedings of the 11th International Conference on Distributed Computing Systems*, (Arlington, Texas), pp. 256–263, 1991.

[21] Y. Zhu, "Efficient processor allocation strategies for mesh-connected parallel computers," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 328–337, 1992.

[22] S. Bhattacharya, L. M. Ni, and W.-T. Tsai, "Lookahead processor allocation in mesh-connected massively parallel computers," Tech. Rep. 93-25, Computer Science Department, University of Minnesota, April 1993.

[23] K. C. Knowlton, "A fast storage allocator," *Communications of the ACM*, vol. 8, pp. 623–625, October 1965.

[24] M.-S. Chen and K. G. Shin, "Processor allocation in an n-cube multiprocessor using gray codes," *IEEE Transactions on Computers*, vol. C-36, pp. 1396–1407, December 1987.

[25] J. Kim, C. R. Das, and W. Lin, "A top-down processor allocation scheme for hypercube computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 1, pp. 20–30, 1991.

[26] A. Al-Dhelaan and B. Bose, "A new strategy for processor allocation in an n-cube multiprocessor," in *Proceedings of the International Phoenix Conference on Computers and Communications*, March 1989.

[27] C.-H. Huang, T.-L. Huang, and J.-Y. Juang, "On processor allocation in hypercube multiprocessors," in *Proceedings of COMPSAC '89: Computer Software and Applications Conference*, (Orlando, Florida), pp. 16–23, IEEE Computer Society, September 1989.

[28] S. Dutt and J. P. Hayes, "On allocating subcubes in a hypercube multiprocessor," in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, (Pasadena, California), pp. 801–810, ACM, January 1988.

[29] P. Krueger, T.-H. Lai, and V. A. Radiya, "Processor allocation vs. job scheduling on hypercube computers," in *Proceedings of the 11th International Conference on Distributed Computing Systems*, (Arlington, Texas), pp. 394–401, 1991.

[30] N. Carriero and D. Gelertner, "Linda in context," *Communications of the ACM*, vol. 32, pp. 444–458, April 1989.

[31] R. Butler and E. Lusk, "User's guide to the P4 programming system," Tech. Rep. ANL/MCS-92/17, Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, Illinois 60439-4844, 1992.

[32] D. W. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," Tech. Rep. ORNL/TM-12512, Oak Ridge National Laboratory, Oak Ridge, TN, October 1993.

[33] S. White, A. Alund, and V. S. Sunderam, "The NAS parallel benchmarks on virtual parallel machines," tech. rep., Department of Mathematics and Computer Science, Emory University, Atlanta, Georgia, November 1993.

[34] M. M. Theimer and K. A. Lantz, "Finding idle machines in a workstation-based distributed system," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, (San Jose, California), pp. 112–122, 1988.

[35] M. Stumm, "The design and implementation of a decentralized scheduling facility for a workstation cluster," in *Proceedings of the Second IEEE Conference on Computer Workstations*, (Santa Clara, California), pp. 12–22, March 1988.

[36] P. Krueger and R. Chawla, "The stealth distributed scheduler," in *Proceedings of the 11th International Conference on Distributed Computing Systems*, (Arlington, Texas), pp. 336–343, 1991.

[37] M. J. Atallah, C. Lock, and D. C. Marinescu, "Co-scheduling compute-intensive tasks on a network of workstations: Model and algorithms," in *Proceedings of the 11th International Conference on Distributed Computing Systems*, (Arlington, Texas), pp. 344–352, 1991.

[38] S. K. Setia, M. S. Squillante, and S. K. Tripathi, "Processor scheduling on multiprogrammed, distributed memory parallel computers," in *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, (Santa Clara, California), pp. 158–170, 1993.

[39] Y. Saad and M. H. Schultz, "Topological Properties of Hypercubes," *IEEE Transactions on Computers*, vol. C-37, pp. 867–872, July 1988.

[40] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer*, vol. 26, pp. 62–76, Feb. 1993.

[41] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak, "The network architecture of the connection machine CM-5," in *Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, (San Diego, California), pp. 272–285, May 1992.