Parallel Job Scheduling on Heterogeneous Networks of Multiprocessor Workstations

by

Gerard Lynch

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Parallel Job Scheduling on Heterogeneous Networks of Multiprocessor Workstations

Gerard Lynch

Master of Science

Graduate Department of Computer Science

University of Toronto

1999

Parallel job scheduling has been heavily studied on various architectures, from vector supercomputers to networks of workstations. A common assumption of this previous work has been that all processors in the system have equal processing power. We investigate parallel job scheduling on *heterogeneous* networks of multiprocessor nodes, where the machines in the network can vary in *processor speed* and *node size* (number of processors per node).

Two dynamic scheduling policies and two adaptive scheduling policies are simulated in this framework. We provide some evidence that a network's *coefficient of variation* (standard deviation over mean) of processor speed or node size is a good predictor of job mean response time for these algorithms. As well, processor-speed heterogneity is found to have a greater effect on job mean response time than node-size heterogeneity. In some cases, the adaptive algorithms can outperform the more flexible dynamic algorithms due to job imbalances created by dynamically reassigning processors.

Acknowledgements

I would like to thank my supervisor, Prof. G. Scott Graham, for his advice and encouragement over the past two years. Thesis drafts were reviewed by both Scott and Prof. Ken Sevcik; their comments helped to improve the quality of the thesis. As well, the "support" provided to me by the good folks at Corel Corporation cannot be underestimated, as without it, my simulation experiments would likely have proved infeasible. Finally, fruitful discussions with my officemates Andrew, Jenn, Kal and Nataša helped distract me from useful work and prolong my stay in the groves of academe.

All lies and jest; still, a man hears what he wants to hear and disregards the rest.

Simon & Garfunkel, The Boxer

Contents

Chapter 1

Introduction

Traditionally, parallel processing has been performed on large, expensive supercomputers. However, steady growth in processing power and network bandwidth have resulted in an alternative: networks of workstations (NOWs). In theory as powerful as large supercomputing platforms, but at a lower cost, NOWs have been called "the poor man's supercomputer". Much time and effort have been devoted to researching parallel job scheduling in networked environments, and progress has been made with approaches such as load balancing, communication libraries to facilitate parallel communication, and different scheduling techniques.

A common assumption of research in this area has been that all machines in the network are equal in capability. This ideal situation is not always achievable in the real world, since many networks are not created all at once. More often, a string of purchases at different times will create a network that is *heterogeneous*, as the machines are not all identical. The flexibility and low cost of assembling such a "poorer man's supercomputer" is attractive, so it is likely that such systems will continue to exist and be used for high-performance computing.

Figure 1.1 illustrates where heterogeneous computing fits into the spectrum of high-performance computing platforms. The possible architectures range from traditional

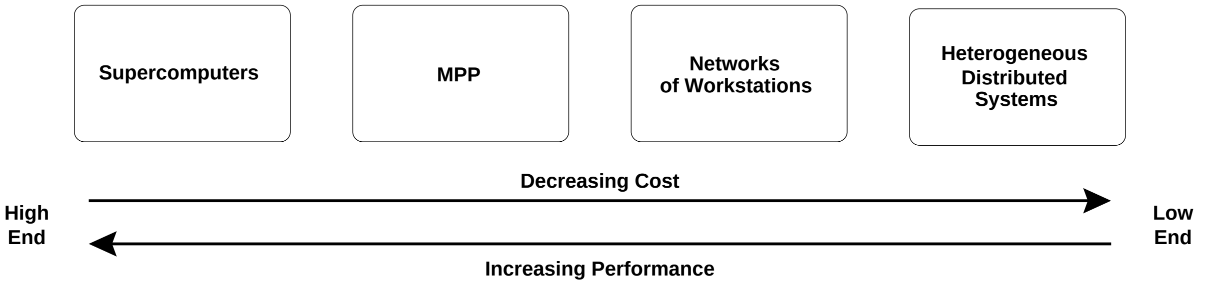


Figure 1.1: *The cost versus benefit for various high performance computing architectures. Adapted from [DRSS99].*

vector supercomputers, to massively parallel processors (MPPs) with varying degrees of memory coherence and processor coupling, down to workstations on a LAN, and finally to geographically distributed sites connected through the Internet. The type of system we describe above, and consider in this thesis, fits into the "Networks of Workstations" category, although it is farther to the right than the left of this section.

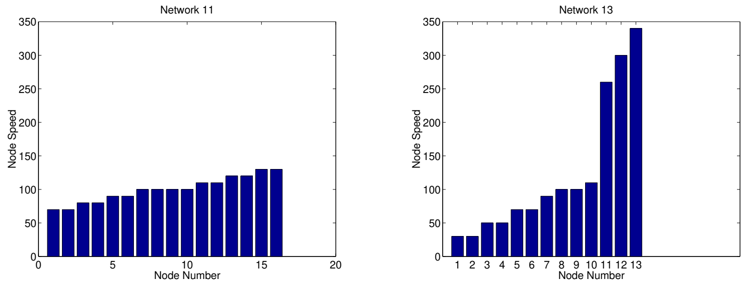
In much of the computer science literature, the term *heterogeneous computing* refers to computing with loosely-coupled, geographically distributed *meta-systems* such as those on the far right of Figure 1.1. The issues that arise in this area are legion, and constitute an interesting body of systems research; however, in our research, we will assume a network of workstations of similar architecture, connected by a fast network. This allows us to ignore the *mechanisms* involved in getting jobs to run, and concentrate on *policy* issues for selecting where and when they should run.

We have restricted the problem domain by limiting ourselves to networks of workstations, but by allowing the workstations to differ from one another, we are still confronted by a dizzying array of possible areas of heterogeneity. The processors in the network could have different clock speeds, memory sizes, network cards, cache sizes, disks, or many other things that could cause them to behave differently. Separating out how each of these factors affect job performance alone or in combination is a large task.

As a result, we restrict the problem further, and focus our investigation on only two dimensions of heterogeneity: processor speed and node size. It is obvious that a heterogeneous network can vary in processor speed. We hypothesize that future networks will also vary in the number of processors at each workstation. Small-scale shared-memory multiprocessors (SMPs) have already been seen in the high-performance computing field, and impressive performance results have been reported when factors such as cache memories and differential communication latencies are taken into account [Woo96]. It is a reasonable assumption that future heterogeneous networks will contain a mix of uniprocessor and multiprocessor nodes; in fact, the term "Clumps" (CLUsters of MultiProcessors) has been proposed for such networks [Lum98]. Our model of this type of network is explained in greater detail in Chapter 3, where we introduce notation as well.

In this thesis, we attempt to determine how scheduling performance is affected by varying the degree of heterogeneity in the network. The problem is approached through simulation of scheduling algorithms on various networks that vary in our chosen dimensions of heterogeneity: processor speed and node size. As mentioned above, these are not the only aspects of a system that could be heterogeneous. However, we believe that speed and size are the two most prominent attributes of a network of multiprocessors, and have the added virtue of being easier to model than other attributes.

We intend to compare networks that differ in heterogeneity. To our knowledge, this has not been done before, so we must decide on some measure that will effectively differentiate between networks that have different degrees of heterogeneity. Although no formal method exists, we can see that there must be some intuitive sense of what constitutes a situation of greater or lesser heterogeneity, where "lesser" heterogeneity is closer to being homogeneous than "greater" heterogeneity. Looking at Figure 1.2, we can see that the network in 1.2(b) is more speed-heterogeneous than the network in 1.2(a). We believe that this "intuition" of the degree of heterogeneity really comes down to an estimate of the distribution of the attribute in question (here, processor speed), and so



(a) Less heterogneneous (b) More heterogeneous

Figure 1.2: *Speed-heterogeneous networks*

	Non-Spanning	**Spanning**
Adaptive	Never-Span (NS)	Span-Sometimes (SS)
Dynamic	Dynamic Never-Span (DNS)	Dynamic Span-Sometimes (DSS)

Table 1.1: *The algorithms studied in this thesis, divided by category.*

our decision has been to use the coefficient of variation or CV (standard deviation over mean) of the dimension in question as a numeric measure of heterogeneity. Since our work is primarily experimental rather than theoretical, we do not spend time developing this concept, but simply use it as a tool to characterize the possible networks in our model.

By increasing the CV of either node size or processor speed, we are able to create heterogeneous networks in either dimension. We use networks created in this way to examine the performance of four scheduling algorithms. The algorithms are explained in detail in Chapter 4, but we will briefly describe them here. They are subdivided along two axes, dynamic/adaptive and spanning/non-spanning, as illustrated in Table 1.1. There are two adaptive algorithms, Never-Span (NS) and Span-Sometimes (SS), of which NS is non-spanning and SS is spanning, and two dynamic algorithms, Dynamic Never-Span (DNS) and Dynamic Span-Sometimes (DSS), of which DNS is non-spanning and DSS is spanning.

The classification of algorithms as adaptive or dynamic is part of a larger division into space-partitioning and time-slicing algorithms. Space-partitioning policies divide the available processors among the competing jobs and time-slicing algorithms rotate the processors among the competing jobs. Space-partitioning algorithms can be classified as fixed, variable, adaptive or dynamic. Fixed partitioning uses a number of permanent subdivisions of the available processors. Variable partitioning assigns the number of processors requested by a job when they become available. Fixed partitioning has been shown to be insufficiently flexible for most workloads, and variable partitioning is interesting only in restricted architectures. As a result, we consider only adaptive and dynamic algorithms in our study.

Dynamic algorithms allow running jobs to have their processor allocation changed at any time (jobs are *malleable*), while adaptive algorithms can set any processor allocation for a job when it begins to run, but this allocation is fixed for the lifetime of the job (jobs are *moldable*). Dynamic scheduling algorithms provide the most flexibility to the scheduler, and thus are preferred when the architecture can support them. Adaptive schemes are usually most appropriate when the pre-emption and data re-partitioning of a dynamic technique is expensive, and for this reason have been considered more often in distributed environments such as NOWs. Our use of multiprocessor workstations, however, allows pre-emption and re-allocation of processors belonging to a workstation to be done cheaply, and so we consider algorithms that are dynamic in this way. These are not fully dynamic, as jobs cannot be assigned any processor allocation, so we use the term *semi-dynamic* to describe them.

Spanning refers to whether or not a job is allowed to execute on more than one workstation; the spanning algorithms allow a job to execute on more than one workstation at a time, while the non-spanning algorithms restrict each job to running on a single workstation. Although it seems intuitive that a job would benefit from using more than one workstation at a time, this is not a foregone conclusion. We allow multiprocessor workstations, so a certain amount of parallelism can sometimes be exploited by a job even without spanning. Communication between threads on the same workstation is much cheaper than between threads on different workstations, so a much larger communication overhead is incurred by a spanning job. As well, the fact that our workstations may be of different speeds and node sizes can affect a spanning job's performance, as it is not obvious that adding a very slow processor to a job that already has a number of fast processors will actually improve performance.

All of our algorithms attempt to divide the processing power of the system somewhat evenly between the competing jobs, as this has been shown in previous work to be a good strategy when other characteristics of the jobs are not known. In a homogeneous system, this is accomplished by giving each job a roughly equal number of processors. In our heterogeneous system, this is not directly applicable, as the processors have different speeds. We therefore introduce the concept of the *capacity* of a set of processors, and define it as the sum of the speeds of the processors in the set. We then attempt to divide the available capacity of the system equally between the running jobs.

We evaluate the algorithms through simulation, allowing us to easily vary the architecture of the network. This decision makes it necessary to model some parameters of the workload, the most important of which are: job inter-arrival time, job parallelism, job runtime, and communication cost. The type of workload in which we are primarily interested consists mainly of traditional high-performance parallel applications, such as scientific computing and engineering work. This is not to say that the job mix is entirely large parallel jobs, but that we are interested in the workloads of sites that perform these kinds of computations. In fact, as is explained further in Section 2.2, workloads of this type often have a substantial fraction of sequential jobs.

We represent job parallelism (number of threads per job) and runtime with a model developed by Feitelson and Jette [FJ97]. This model is based on several studies of job traces from supercomputing sites, and is in our opinion consistent with most published empirical studies of parallel workloads.

We have designed our own model of communication based on data from the SPLASH-2 parallel benchmark suite [WOT+95]. Although we would prefer to have used data from job traces, this type of information was not available in any of the workload studies of which we are aware. Our model relates the amount of communication performed by a job to the job parallelism with a logarithmic function, and assumes that communication demand is equally distributed both across all threads and throughout the lifetime of the job. Admittedly, this is a simple model that does not capture the wide variation in communication patterns that would be present in a true parallel workload. Its value for our purposes will be in simplicity. A more complex model would have to account for different types of communication such as barriers, nearest-neighbours or global exchange, and the distribution and frequency of occurrence of each type. Without reliable workload data from job traces, such a complex model would be as likely to be misleading as helpful.

We do not attempt to model the network itself explicitly, as we feel that this is a level of complexity not warranted in a study of this type. Modern switched networks can provide point-to-point communication latencies that are relatively constant for different levels of network loads, allowing us to concentrate on the amount of communication performed by a job, instead of the way in which the communication is performed.

Parallel workload modelling is discussed in more detail in Section 2.2, and Feitelson's model and our communication model are covered in Chapter 3.

We intend the primary contribution of the thesis to be an increased understanding of the ways in which heterogeneity (in the restricted sense described above) affects known approaches to parallel job scheduling. We are less interested in finding a very good parallel job scheduling algorithm for a heterogeneous environment than we are in determining why aspects of known approaches do or do not perform well in our model. Of course, practical and useful algorithms are a desired offshoot of this work, and we would hope that further investigation would eventually produce practical algorithms suitable for heterogeneous systems. Our purpose in doing this research is to provide some guidelines for future designers of such algorithms.

1.0.1 Heterogeneity Background

The word *heterogeneity* has been heavily used in computer science, to the point where it can mean many different things, in different contexts, to different people. While all of these meanings come down to the fact that not all of the constituent parts making up a system are identical, they often describe quite different fields of research. Some areas often described as heterogeneous computing that we do *not* consider here include:

- **heterogeneous task placement**

  This is the problem of matching segments of a parallel program described by a directed acyclic graph to minimize the execution time, as in work by Watson et al. [WAS+96].

- **network heterogeneity**

  This is the idea of optimizing execution by utilizing different network protocols for different message types and sizes, as in work by Kim and Lilja [KL97].

- **meta-computing**

  This assumes machines of different architectures, where a program binary is not guaranteed to run identically or at all at a different place in the network, as in work by Berman et al. [SBWS99].

- **wide-area computing**

  This involves using the Internet or some other loosely-coupled system to perform computation, such as the Legion system [GW97].

A taxonomy of heterogeneous systems has been suggested by Eshaghian [Esh96], and is shown in Figure 1.3.

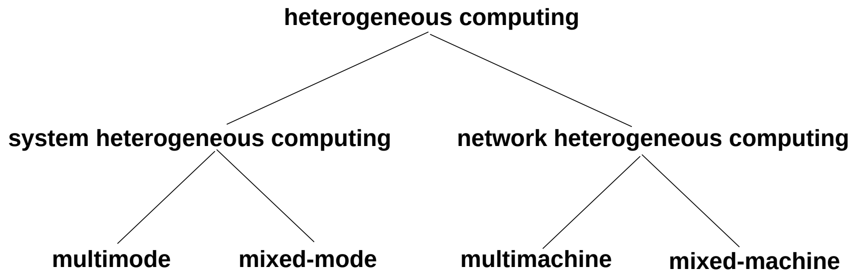


Figure 1.3: *A taxonomy of heterogeneous computing. Adapted from [Esh96] .*

- **System Heterogeneous Computing (SHC)**

  System heterogeneous computing describes systems where a single autonomous multiprocessor can operate in both single-instruction-multiple-data (SIMD) and multiple-instruction-multiple-data (MIMD) modes. A *multi-mode* system can operate in both modes simultaneously, while a *mixed-mode* system will switch back and forth between SIMD and MIMD.

- **Network Heterogeneous Computing (NHC)**

  Network heterogeneous computing utilizes a collection of autonomous machines to perform parallel or distributed computing. *Multi-machine* networks are a collection of identical machines. In *mixed-machine* computing, the computers in the network may be different. Multi-machine is a special case of mixed-machine.

Our work falls into the category of mixed-machine NHC, since our model is a collection of autonomous machines. Our feeling is that future heterogeneous systems are more likely to be collections of workstations connected by a fast network.

Although we would like to present a breakdown of scheduling approaches in heterogeneous systems similar to ours, we are unable to, as we know of almost no work applicable to ours. We feel that this provides sufficient motivation for our work, especially since we believe that this type of heterogeneous network is likely to become more commonly used for high-performance computing.

1.1 Thesis Organization

The remainder of the thesis is organized in the following way.

In Chapter 2, an overview of research in various relevant areas is presented. We discuss parallel job scheduling and classify different approaches and models used in the field. An overview of workload characterization is given to motivate the workload assumptions we make in our own experiments.

In Chapter 3, we describe the system model in which our experiments are performed, and present the workload model we use for generating simulated jobs.

In Chapter 4, the scheduling policies we use are defined, and we present an explanation of our experiments and their implementation through simulation.

In Chapter 5, the results of our simulation experiments are presented graphically and discussed.

In Chapter 6, we present our conclusions and discuss avenues for future work.

Chapter 2

Background

In this chapter we present some definitions, terminology, and previous work that is relevant to our research.

First we present background material in the area of parallel job scheduling. This field is impossible to cover completely, so we concentrate on areas closely related to our work, especially space-partitioning algorithms and networks of workstations. Some terms and definitions are provided to guide the reader, and we also introduce some basic policies through a discussion of uniprocessor scheduling.

As well, we discuss previous work in the area of parallel workload characterization. Since our study is carried out through simulation, we are interested in motivating the distributions of the parameters used in our experiments. Many previous studies have used somewhat arbitrary choices for their system and job parameters. We try to avoid this by using data based on studies of actual job traces whenever possible.

2.1 Scheduling

2.1.1 Definitions

A wide variation exists in the terminology used by various authors, in scheduling, and in the computer science literature in general. In this thesis the following definitions will be used.

A *processor* is an entity that performs some computation at a given speed. A *node* is a collection of processors of the same speed. A *thread* is a program that is executed by a processor, usually consisting of an address space, although on a shared memory architecture the address space could be shared with other threads. A *job* consists of one or more threads, and corresponds to an execution of an *application* on a given input. Note that different executions of the same application will produce different jobs.

Scheduling is the act of deciding which threads should be assigned to which processors at any given time.

Speedup and Efficiency

One useful thing to know about a job is how efficiently it is likely to use any extra processors it is allocated. For this purpose we define *speedup* and *efficiency*. Let T_i^j to be the time required to execute job j on i processors. For clarity, we will omit the superscripted j in the following discussion, but bear in mind that these application characteristics are different for every job. The speedup of an application running on p processors is:

$$S_p = \frac{T_1}{T_p}$$

Technically, this is the *relative speedup* , as opposed to the *absolute speedup*, where T_1 is the execution time of the best known sequential algorithm. However, in most of the literature, "speedup" refers to relative speedup.

A speedup that increases unit-linearly with p is desirable, and in theory is the maximum attainable. In most cases, this is unrealistic, as not all work can be parallelized, and overheads such as communication medium contention and congestion will grow with the number of processors. Some *super-linear speedup* has been reported, but this is generally due to the increase in cache memory or some other resource as extra processors are added. A related measure is *efficiency*, defined as

$$efficiency = \frac{S_p}{p}$$

A job with "good" speedup (close to linear) will tend to have efficiency close to 1.

A well-known bound on the speedup is the fraction α of sequential work that must be performed, due to Amdahl [Amd67]. If the total work is 1, then $1 - \alpha$ is the amount of work that can be perfectly parallelized, and the speedup is bounded by:

$$S_p \leq \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \leq \frac{1}{\alpha}$$

regardless of the number of processors. So if 10% of the work must be performed sequentially, the speedup can be no larger than 10.

Gustafson [Gus88] has argued, however, that since having more processors allows a larger problem to solved, it is generally the total execution time of the application that stays constant as processors are added. The sequential execution time is relatively constant regardless of the number of processors, so this results in a smaller α as p grows. This gives a formulation for *scaled speedup*

$$Scaled\ S_p = \alpha' + (1 - \alpha')p$$

where α' is the fraction of sequential work on a parallel machine of p processors. Scaled speedup is bounded only by p.

Metrics

Many different metrics are available with which to evaluate a scheduling algorithm. We define some metrics here that we use later in the thesis.

A common metric is the *mean response time* (MRT). For n jobs, where job $j_i (i = 1...n)$ arrives in the system at time s_i and leaves the system at time t_i, the mean response time is defined as

$$mean\ response\ time = \frac{\sum_{i=1}^{n}(t_i - s_i)}{n}$$

Another standard metric is the *makespan*. Informally, this is how long it takes to finish all n jobs. Formally, assuming we started at time 0, this can be defined as the finish time of the last job:

$$makespan = \max_{1 \le i \le n} t_i$$

The point at which the system saturates is known as the level of *sustainable throughput*. This is the job arrival rate at which the expected mean response time becomes infinite, since jobs are arriving faster than they can be serviced.

Power is mean response time divided by throughput [Kle79]. At low load, throughput is the same as the arrival rate, and at high load, the mean response time will tend to infinity, so power will tend to zero at both low and high arrival rates. The maximum point reached by the power curve is the arrival rate at which the tradeoff between mean response time and throughput provides the most overall benefit.

2.1.2 Uniprocessor Results

To understand the development of multiprocessor scheduling algorithms, we will first consider some well-known results concerning single processor scheduling. A further discussion can be found in [Dei90].

Scheduling policies for a single processor can be classified by how much information is available to the scheduler. If the service demands of all jobs are known explicitly, then Shortest Job First (SJF), running the jobs in increasing order of service demand, is the best policy for minimizing mean response time, since the short jobs will run first and finish quickly. If pre-emption of running jobs is allowed and is free, then Shortest Remaining Processing Time First (SRPT) will minimize the mean response time.

Priority scheduling assigns a priority to each job, and runs the job with the highest priority, with jobs of equal priority being scheduled in FCFS order. This can allow queues of jobs with different resource requirements to execute at different priorities, e.g., short interactive jobs at high priority and long batch jobs at low priority.

Since it is often difficult to tell what type of job one is dealing with, many uniprocessor schedulers use a multi-level feedback scheduling policy. A high priority is assigned to new jobs, but as a job acquires processing time, its priority is gradually lowered. Studies of typical workloads indicate that the variance in the distribution of job service times is high. There tend to be many short jobs, and a few very long jobs, implying that newly arrived jobs are more likely to be short jobs. Due to their high priority, they are run soon after they arrive, until they reveal themselves as long non-interactive jobs (by continuing to run and not blocking on I/O), and cause their priority to drop. This is an approximation of SRPT, as the high-priority jobs have a shorter expected remaining processing time.

If the coefficient of variation of job runtime is known, then the optimal scheduler can be chosen. If the CV is less than 1, then FCFS is optimal, as a job with some acquired processing time is expected to finish before a newly arrived job. If the CV is greater than 1, then a feedback policy that approximates SRPT will perform best.

If no information about the job runtimes is available, then Round-Robin (RR) is a good compromise policy, as it gives all jobs equal service. However, the quantum length must be selected carefully, as a long quantum will degenerate to first-come-first-served scheduling, and a short quantum will have excessive context switch overhead.

2.1.3 Classification Schemes for On-line Scheduling

Scheduling algorithms can be subdivided in a number of ways. In this section we briefly discuss the differences between algorithms that are single-level or two-level algorithms, thread-oriented or job-oriented, and dynamic or static.

Single-Level vs. Two-Level

A well-known division in classifying scheduling disciplines is the difference between *single-level* and *two-level* scheduling. Single-level schedulers allocate processors to jobs, and also control the allocation of specific threads of the job to specific processors. In a two-level scheduling policy, the scheduler allocates a set of processors to a job, but then the scheduling of threads is done separately, perhaps by the application itself, lower-level operating system code, or the use of a runtime library. The second, lower level of scheduling allows job-specific optimizations to be made, and avoids the expensive overhead of managing every thread in the system.

Although the two levels of scheduling seem quite different, most scheduling algorithms could be used at either level in different situations [Fei95].

Thread-Oriented vs. Job-Oriented

Closely related to single-level and two-level scheduling is the difference between *thread-oriented* and *job-oriented* dispatching. A thread-oriented discipline does not distinguish between threads based on the job to which they belong. A FCFS global queue of threads is a simple example. A job-oriented discipline, on the other hand, schedules jobs rather than threads. Although a single-level scheduler is not necessarily thread-oriented or job-oriented, the top level of a two-level scheduler is job-oriented. In general, a thread-oriented scheduler can achieve good performance if the threads of jobs in the workload tend to perform unrelated work, and thus do not synchronize often. However, it has long been known that if threads of a job communicate often, a form of thrashing can occur due to excessive context switching [Ous82], so some form of co-ordination between the threads of a job is required in this case.

Early multiprogrammed multiprocessor scheduling research concentrated on comparisons between thread and job-oriented policies through simple extensions of well-known uniprocessor algorithms [MEB88]. For workloads that showed correlation between job demand and number of processes (as is often the case, see Section 2.2), thread-oriented policies such as RR (between threads) and FCFS resulted in higher mean response times than job-oriented policies such as SJF, SRPT and Smallest Number of Processes First (SNPF). Since thread-oriented policies favour jobs with more threads, they were also favouring jobs with greater service demands, causing the mean response time to increase. This is different from uniprocessor scheduling, where RR is known to be insensitive to job demand distributions. However, for workloads with low variability in job parallelism and high variability in job demand, RR still performed as well as the job-oriented policies, confirming the uniprocessor result.

Leutenegger and Vernon [LV90] attempted to counteract this effect with the job-oriented policy RRJob, which time-slices jobs, and during a job's time quantum, time-slices the threads of the job. They compared RRJob and an equipartition policy (see Section 2.1.4) with the thread-oriented policies studied by Majumdar et al. [MEB88] and found that the job-oriented policies gave lower mean response times under most workload assumptions.

Dynamic vs. Static

Scheduling algorithms can be classified by whether a job's processor allocation is fixed by the application, set at runtime, or changes as the application executes. There are some differences in the terminology used by different authors for this distinction, especially in earlier work. Where appropriate, we will note the differences for clarification, but otherwise the following taxonomy due to Feitelson [Fei95] will be used.

- **rigid** the number of desired processors is set by the application before running. The job will not run on less, and cannot utilize more. These types of jobs, as well as algorithms assuming these types of jobs, are often known as *static*.

- **moldable** the number of processors can be set at the beginning of the job, but cannot be changed during execution. This is often implemented as a parameter to the application. Moldable jobs are sometimes called *adaptive* jobs.

- **evolving** the application goes through a number of phases of varying parallelism, controlled by and/or communicated to the scheduler by the job as it executes.

- **malleable** the application can have its processor allocation changed at any time in response to a request by the scheduler. Malleable jobs are not a generally supported programming model, since often significant programming effort and execution overhead must be expended to change a job's processor allocation during execution. In a distributed memory environment especially, the costs of data repartitioning can overwhelm the scheduling benefits realized by malleable job support. The programming model known as "workpile of chores" is often assumed in studies of malleable jobs. In this model, a global queue of work is maintained, and any number of worker threads pick up work from the queue. This allows threads to be started and stopped at (almost) any time. Malleable jobs can also be "faked" through a technique known as *folding*, introduced by Zahorjan and McCann [MZ94]. Threads of a job are multiplexed onto fewer processors than there are threads. When the number of processors does not divide the number of threads, the extra processors are "rotated" to give equal average efficiency.

The type of job assumed by the scheduler determines what kind of *preemption* the scheduler can perform. The various degrees of pre-emption have received different names as well. For example, Parsons [Par97] uses this taxonomy:

- **run-to-completion (RTC)** all jobs run until they are finished.
- **simple preemption** jobs can be stopped so that their processors can be made available to other jobs, but they must be restarted on the same set of processors.
- **migratable preemption** jobs can be stopped and restarted on any set of processors, but the number of processors cannot change.
- **malleable preemption** jobs can be stopped and restarted on any number of processors. As the name suggests, the jobs must be malleable.

Most schedulers currently used in production parallel environments use some form of run-to-completion discipline [FRS+97], with fixed partitions (see Section 2.1.4) Although it seems intuitive that the flexibility allowed by a preemptive discipline would give better performance, RTC can be attractive for other reasons. The time when a job will finish is often quite predictable. A popular scheduling algorithm of this type is EASY [SCZL96], which guarantees that jobs will never be delayed by jobs submitted after them. However, research has demonstrated the desirability of more adaptable disciplines and job models [PS97].

2.1.4 Space Partitioning

Algorithms that divide the available processors among the competing jobs without explicitly time-slicing processors among jobs are known as *space-sharing* or *space partitioning.*

They can be classified similarly to the type of job model that they assume (see Section 2.1.3), into fixed, variable, adaptive and dynamic partitioning. As mentioned in Chapter 1, the algorithms that we study in our work are adaptive and dynamic partitioning algorithms; thus our discussion of these two areas is longer than that for fixed and variable partitioning.

Fixed Partitioning

In fixed partitioning, the system is permanently subdivided into partitions of fixed size. These are generally set by the system administrator, and require reconfiguration and perhaps downtime to be changed. Often machines will have a small partition for development and testing, and reserve a number of larger partitions for production work. The partitions can be associated with different sizes of jobs through job queues. Internal fragmentation can be a problem with this scheme if jobs are able to utilize a number of processors that does not correspond to an existing partition size.

The utility of dividing the available processors into equal-sized partitions has been studied with analytic models [Sev89, MEB91, ST93, SST93], compared with adaptive policies [Sev89, MEB91, GST91, RSD+94, ST93], compared with timeslicing policies [SST93], and compared with both adaptive and dynamic policies [NSS93b, NSS93a, NSS97]. The general conclusion is that fixed partitioning can perform well if the workload is well-understood and jobs tend to have low variation in parallelism and service times. In many cases, a particular size of fixed partitioning outperforms all other policies for a particular system load and workload, but will perform poorly in other situations. This has led to research into adaptive policies, which attempt to find an appropriate partition size given the current load and perhaps other system and application information.

Variable Partitioning

Variable partitioning refers to partitioning the processors in response to requests from applications for certain numbers of processors. The problem is uninteresting in terms of scheduling unless there are special architectural problems that make it difficult to find a partition of a given size. For example, hypercube architectures generally require that partition sizes are multiples of powers of two, which has given rise to a large body of literature on finding and uniting sub-cubes of various sizes. Mesh architectures can have requirements for rectangular allocations or thread adjacency, which motivated the use of schemes such as the buddy system.

If a given partition size cannot be allocated, the job must be queued. This leads to some questions about the order in which to run the jobs in the queue. If job runtime is correlated with job size, Smallest Job First can perform well [Fei95], but this is not always the case.

Further discussion of variable partitioning and the associated literature can be found in the survey by Feitelson [Fei95].

Adaptive Partitioning

Adaptive partitioning assumes that the jobs submitted to the system are moldable (see Section 2.1.3), and thus will execute on any number of processors assigned by the scheduling discipline. Adaptive policies are useful in environments where the benefit of changing a job's processor allocation during execution can be offset by the overhead of doing so. Distributed memory environments are an example, since the overhead of redistributing data structures can be high.

Work on adaptive partitioning algorithms has concentrated on reducing the sizes of processor allocations under high load conditions to improve mean response times, while running applications on the partition size that gives the greatest execution efficiency.

Sevcik [Sev89] compares adaptive policies that utilize information derived from a job's parallelism profile to decide on the appropriate processor allocation. The policy called A+&mM gave lower mean response times using system load and the average, minimum, maximum and variance in parallelism, beating other rules using only one or two of these characteristics.

Ghosal et al. [GST91] introduce a concept known as the "processor working set" (pws). They use a measure called "efficacy" $\eta(p)$, defined as

$$\eta(p) = \frac{S_p}{C(p)}$$

where the cost function $C(p)$ is defined as

$$C(p) = \frac{p}{S_p}$$

and thus

$$\eta(p) = \frac{{S_p}^2}{p}$$

The pws is the minimum value of p which maximizes $\eta(p)$. It is shown that for the cost function used, pws is the same as the number of processors associated with the knee of the execution time-efficiency profile [EZL89]. Some adaptive policies utilizing pws were investigated. Among them, the best power was observed with a simple strategy of allocating a job's pws when possible. If no jobs in the queue can be allocated their pws, the free processors are allocated to the first job in the queue. This has the effect of allocating smaller partition sizes at high loads.

Majumdar et al. [MEB91] investigate a scheme where the number of processors per job changed with the number of jobs in the system. They found substantial improvements over a fixed partitioning scheme when using as few as three different sizes of processor allocation.

Rosti et al. [RSD+94] compare a range of adaptive policies that differed in the speed with which they would change the target partition size in reaction to a burst of arrivals or a long period of no arrivals. This family of adaptive policies was found to be more robust across workloads and system loads than both an equipartition policy and a non-work-conserving policy that attempted to leave some free processors for use by future arrivals. The most robust policy, known as AP4, increases the target partition size only when there are two free partitions of the previous size, and decreases it whenever the queue length increases.

Adaptive Static Partitioning (ASP), introduced by Setia and Tripathi [ST93], attempts to decrease processor partition sizes under heavy load. If there are free processors, an arriving job is allocated the minimum of the number of available processors and the job's maximum parallelism. If no processors are free, the job is queued. On job departures, the newly freed processors are divided evenly between the jobs in the queue in FCFS order, subject to no job exceeding its maximum parallelism. ASP was found to outperform two variations of the *pws* policy [GST91] and a fixed partitioning policy.

A variation on ASP is studied by Naik et al. [NSS93b], where the allocation size is also limited by a system-wide minimum. This policy performed nearly as well as fixed policies at particular loads, but was more consistent across different configurations. A dynamic policy allowing limited pre-emption was also studied, but it tended to perform poorly at high load. Further study by the same authors [NSS93a] showed that the dynamic policy handled transient bursts of arrivals much better than the fixed policies.

Chiang et al. [CMV94] compare adaptive policies that use application characteristics such as average parallelism and *pws* with various versions of ASP. They find that for RTC policies to perform well, it is crucial to have a maximum allocation limit as well as reduce allocation under high load. The best results were found with the policy ASP-max+, a version of ASP with a maximum allocation limit that decreases as load increases. This policy consistently gave lower mean response times for workloads with high variability, beating rules using average parallelism and *pws* information. The intuitive explanation presented for this behaviour is the fact that these characteristics, although useful in determining the optimal execution point for jobs in isolation, are poor predictors of job demand. Long jobs can still acquire a large number of processors and hold up short jobs.

Anastasiadis and Sevcik [AS97] study various adaptive policies, using execution time functions based on measurements performed by Wu [Wu93], and comparing the difference $T(p) - T(p+1)$, for $p \geq 1$ for all waiting jobs to see which jobs would benefit most from receiving another processor. They were able to surpass even the performance of dynamic equipartition at very high load levels (although not at lower load levels). The improvement was mainly caused by improved queue ordering due to the application knowledge.

Although adaptive policies will adapt to changing system conditions such as load, they do so at a slower rate than dynamic policies. One proposed solution is leaving some processors in reserve for future arrivals to allow more flexibility in allocation decisions. Policies like this are called *non-work-conserving* if they leave idle processors when there are jobs in the queue. Smirni et al. [SRS+95] study policies that attempted to retain some memory of previous load conditions, hoping that past load would accurately predict current load. They found that saving processors could have some benefit when any of the following conditions held: the workload has poor speedup (in this case there tends to be less benefit in assigning the idle processors to currently running jobs), the system load is high, the system size is large, or job arrivals tend to be bursty. Although this result seems at odds with results that say equipartition can perform well, it does makes sense, since saving processors for future use allows fewer processors to be allocated to the running jobs, and thus will tend to decrease their allocation sizes at higher loads.

Further study of non-work-conserving policies was carried out by Rosti et al. [RSSD95]. They compared a version of ASP-MAX and a processor-saving policy. Their results tended to confirm that non-work-conserving policies perform best when irregularities such as arrival bursts, multi-class workloads and processor failures exist in the system, as well as when the workload has poor speedup. Further results reported by Rosti et al. [RSD+98] indicate that these policies can also perform better when the workload has irregular execution times, and when the system size is increased.

Dynamic Partitioning

Dynamic partitioning policies allow the scheduler to change the processor allocation of a job after it has been started. Varying degrees of flexibility can be used here, corresponding to the levels of preemption discussed in Section 2.1.3. The cost of a preemption or migration assumed for a particular system usually influences the type of dynamic policy considered. It is easy to see that a tightly coupled shared memory machine will have less overhead for thread migration than a distributed network of workstations.

A heavily studied form of dynamic partitioning is *equipartitioning* (EQ), where all jobs in the system are given an equal share of processing power. True equipartitioning is only a theoretical concept, as it requires perfect processor sharing, i.e., no context switch overhead, cache refreshing time or other problems. The approximation most often used first calculates a target allocation t (number of processors over number of jobs), and assigns t processors to each job in increasing order of maximum parallelism. If the maximum parallelism of a job (M_j) is less than t, the job is assigned M_j processors, and t is recomputed using the remaining jobs and processors. Non-integer values of t may be possible if the study is theoretical; otherwise some further decisions may have to be made to assign fractional parts of processors. Equipartition based algorithms have been shown to perform well in many studies [TG89, ZM90, CMV94, AS97] although their frequent repartitioning may result in high overheads. For this reason EQ is often used as an idealized baseline policy in studies of adaptive policies that attempt to approach its performance while limiting overhead costs. Since equipartitioning gives (roughly) equal service to all jobs, it is insensitive to the job service time distribution, and can be looked upon as a multiprocessor analogue to round-robin scheduling. If more application or workload knowledge is available, it is possible to improve upon EQ [Sev94, PS95, PS96, AS97].

The earliest work on equipartitioning is by Tucker and Gupta [TG89], under the name *Process Control*. Their work focuses on the practical aspects of structuring applications to create and destroy threads to keep the number of threads per job equal to the number of processors per job. Jobs were able to suspend and create threads so that the number of threads always matched the number of processors, reducing context-switch overhead and resource contention.

Leutenegger and Vernon [LV90] study the dynamic partitioning scheme in comparison with some simple policies such as FCFS, Shortest Number of Processes First, RR, and RRJob (described in Section 2.1.3). They find that dynamic partitioning (processors are split evenly, with no job receiving more than its maximum parallelism) and RRJob perform better than the other policies. Both give a roughly equal share of processing power to all jobs, whereas the other policies tended to favour one class of jobs at the expense of another.

Crovella et al. [CDD+91] study timeslicing, coscheduling, and space partitioning in a NUMA environment. They find that giving applications exclusive use of a smaller number of processors is more efficient than uncoordinated timesharing or coscheduling. Even with multiple threads of the same application timesliced on a partition, applications did not experience as large of a slowdown as expected.

Zahorjan and McCann [ZM90] compare an adaptive policy that used execution time functions (similar to Anastasiadis and Sevcik [AS97], as described above in Section 2.1.4) and a dynamic policy. The dynamic policy is actually closer to a variable partitioning policy, as the job model is evolving (see Section 2.1.3), allowing jobs to request and release processors according to their parallelism demands. The dynamic policy gave lower mean response times, except when context switch overhead was very high. In later work, McCann et al. [MVZ93] compare a version of the above dynamic policy with RRJob and equipartitioning. They find that the dynamic policy performs best with respect to mean response time, fairness, and response to short, interactive jobs. This is mainly due to the fact that the dynamic policy is able to use processors more efficiently than the other policies, since applications will give up any processors they cannot use.

A policy called *Folding* is introduced by McCann and Zahorjan [MZ94]. The policy is intended for mesh-connected parallel computers, so much work is done to preserve adjacency relationships between threads when changing job partition sizes. On job arrivals, the largest existing partition is split in two, the existing job's threads are "folded" onto half of their previous allocation, and the other half is given to the new job. Processors are rotated between differently sized partitions to give equal average service rates to all jobs. This folding policy is compared with a version of equipartitioning that attempts to allocate partitions with at least one power-of-two dimension, and performs mass reallocations on every job arrival and departure. Although folding tends to perform better when the number of jobs in the system is a power of two (rotations are minimized), it outperforms equipartitioning in an analytic model, a Markovian birth-death model, and a simulation.

Padhye and Dowdy [PD96] compare adaptive and dynamic policies. An adaptive algorithm (RA) based on those studied by Rosti et al. [RSD+94] is compared with a dynamic EQ policy and a folding policy similar to that studied by McCann and Zahorjan [MZ94] (described above). Using a synthetic workload running on an Intel Paragon, they found that folding outperformed EQ. Folding also gave a better MRT than the RA policy when the workload had poor speedup and variable demand, but not if the workload had good speedup and less variability in demand.

The impact of repartitioning overheads on dynamic equipartitioning policies is investigated by Islam et al. [IPS96]. They implement a malleable job system on NOW and an IBM SP-2, both distributed memory systems where repartitioning could be expected to be expensive. Measurements of this system are then used to create an analytic model that includes the repartitioning overhead of equipartitioning. To simulate an adaptive policy, a static partitioning policy is simulated for different partition sizes and load levels, and the best partition size for each load level is used when comparing against the EQ policy. Even given these overheads, the dynamic policy tended to outperform the static, except when the workload was dominated by small, short jobs, for which the overhead of frequent repartitioning was not warranted. EQ's relative performance benefit increased with higher system loads and arrival burstiness.

The study by Naik et al. [NSS97] compares a fixed partitioning policy, an adaptive policy and a dynamic policy. The policies are based on those developed by Naik et al. [NSS93b], and discussed in Section 2.1.4. Jobs are classified as small, medium or large. The static policy divides the processors among the job classes and then assigns equal-sized partitions to jobs according to their class. The adaptive policy also divides the processors into 3 pools (for large, medium and small jobs) classes, but within the pools the partition size grows and shrinks with system load. The dynamic policy will reconfigure large and medium-sized jobs to different processor allocations when processors are needed or when there are enough free processors available. The adaptive policies tended to perform as well as the fixed policies, and provided more flexibility, as they performed well for a wider range of workloads and arrival rates. The dynamic policy outperformed the others w.r.t. MRT for all loads. The overhead of application reconfiguration is explicitly modeled, and does not change the qualitative results until it is increased by a factor of 4.

2.1.5 Time Slicing

Partitioning along the axis of time rather than space is known as time-slicing. Generally, there are few algorithms that perform any kind of time-slicing in a distributed system such we are considering in this thesis. The ability to time-slice is desirable, however, as mean response time can be greatly reduced by the ability to pre-empt running jobs so newly arrived short jobs can finish quickly. This has inspired some researchers to

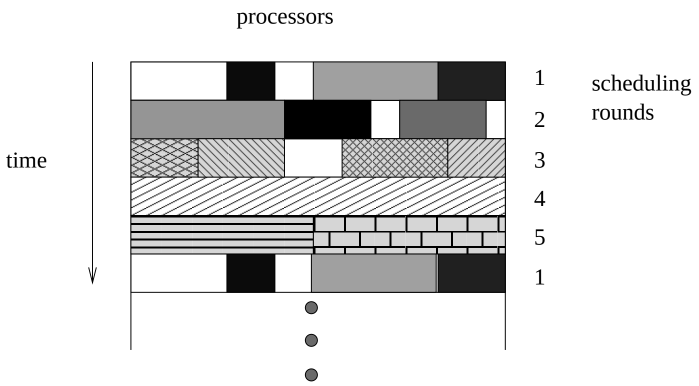


Figure 2.1: *Co-scheduling with 5 rows: scheduling rounds continue through, giving each row a quantum, and then repeat. Each pattern in the figure is a different job, white space is packing loss. New jobs form new rows if they don't fit into existing ones, and departing jobs will cause rows to be collapsed together if possible.*

implement forms of time-slicing in NOW environments [DAC96, ADCM98, SPWC98].

Time-slicing techniques can be distinguished by whether they use a global queue for the entire system, or local queues on each processor. Local queues are more common in distributed memory machines, because of the communication overhead involved in maintaining a shared global queue. A major issue when using local queues is how to balance the load between processors, since there is no global co-ordination. This can be performed by intelligent initial placement, or by migrating threads from heavily-loaded processors to lightly-loaded ones. A global queue is simple to implement if shared memory is available, and does not have any load balancing issues. It can become a central bottleneck, however, so is more likely to be seen in smaller scale machines.

Co-ordinated timeslicing refers to job-oriented time-slicing policies. There are two main types of co-ordinated time slicing, gang scheduling and static quantum-based scheduling.

Gang Scheduling Early work by Ousterhout [Ous82] established that if threads of a job communicate often, performance could be undermined due to the large number of context switches caused by threads blocking when sending messages to threads that were not scheduled. He proposed *co-scheduling*, which attempted to schedule threads of the same jobs together whenever possible. A matrix algorithm was used, which maintained rows of jobs which were scheduled together, as in Figure 2.1.

Although the term *gang scheduling* is often used interchangeably with co-scheduling, it is claimed by Feitelson at least, that gang scheduling means that threads of a job are always scheduled together, as opposed to co-scheduling, which will schedule threads together when possible [Fei96]. Gang scheduling is something of a cross between space-slicing and time-slicing. All threads have a processor each, so the processors are partitioned by job, but jobs are also time-sliced with other jobs. This reduces the thrashing due to excessive context-switching described by Ousterhout, since all threads in the job are scheduled at the same time. The time-slicing of jobs allows mean response time to be kept low, as short jobs will get a chance to run quickly.

Achieving gang scheduling can be difficult unless special hardware exists to perform simultaneous context switching across all of the processors. In a tightly coupled MPP, this is easier to accomplish than in a distributed system. However, some have implemented mechanisms for gang scheduling across a network of workstations. These approaches are the most applicable to our work, and are described in Section 2.1.6.

Static Quantum-Based Policies Parsons and Sevcik [PS95] study what they call *static quantum-based* algorithms. The policies investigated are based on the multilevel feedback queue scheduling used for uniprocessors. They generalize the well-known adaptive space partitioning disciplines ASP [ST93] and PWS [GST91] to feedback preemptive disciplines, FB-PWS and FB-ASP. Every quantum, the jobs with the least acquired processing time are selected to run. In workloads with high CV of job demand these jobs are expected to finish first. The partition sizes allocated decrease as load on the system increases. The feedback versions of the policies performed much better, according to mean response time, than the RTC versions for all choices of workload and offered load, with more pronounced tendencies as the CV of job demand rose. In the case of CV = 70, the difference was two orders of magnitude. The FB-PWS also policy came close to, and in some cases surpassed an idealized equipartition policy with no overhead penalties.

Chiang and Vernon [CV96] follow up on this work using different workload models. The results of Parsons and Sevcik [PS95] are corroborated for workloads with a large number of inefficient jobs, but it is found that for workloads where large jobs tend to have at least moderate efficiencies, EQ still outperforms the feedback policies. A new policy called EQS-PWS is presented that combines the approaches of the *pws*-based policy and EQ. Instead of allocating $\min(t, M_j)$ processors, $\min(t, pws)$ processors are allocated. This policy performs as well or better than the FB-PWS and EQ for all of the workloads examined.

2.1.6 Networks of Workstations (NOWs)

Networks of commodity workstations have a much lower cost per processor than large parallel computers. For this reason, they have been touted as the parallel computing platform of the future [ACP95]. There is certainly evidence that parallel computing is feasible on this type of platform [CADAD+97]. Much research [ADCM98, DAC96, ADC97, WZ97, KL97, LS93, ADV+95, SPWC98, DZ97, AS97, AFKT98] has been devoted to determining how to schedule parallel applications effectively in a NOW environment.

In theory a NOW could be as powerful as a large, tightly coupled parallel machine, but there are some problems with the architecture that tend to limit performance. The communication medium is generally orders of magnitude slower. Gang scheduling is difficult to support directly, as hardware for co-ordinated context switching does not exist. Although implementations of distributed shared memory [ACD+96, SBIS98] allow shared-memory applications to execute on a NOW, they are still generally out-performed by applications optimized for message-passing. In particular, recent improvements in processor speed have not been matched by improvements in network latencies, limiting the speedups achievable [PBS97].

Sobalvarro et al. [SPWC98] study coscheduling on workstation clusters. They use a mechanism known as *dynamic co-scheduling* (first studied by Sobalvarro et al. [SW95]) to achieve strong co-scheduling behaviour. Threads that are not running when a message arrives are awakened and run. Transitive message-passing relationships can tend to co-schedule groups of communicating processes without explicit identification of the communicating groups. This approach has been successfully implemented in both the Solaris and Windows NT operating systems [BC97].

A related idea is *implicit co-scheduling*, studied by Dusseau et al. [DAC96, ADCM98]. "Two-phase spin blocking" is used. Instead of blocking when waiting for a message, the process will spin for a certain amount of time before blocking, in hopes of receiving the message reply and avoiding the context switch overhead. The length of the spin period is dependent on the amount and type of messages received and sent by a process. Processes that are communicating often will increase their spin time. They give results showing this mechanism to be competitive with explicit coscheduling; a maximum slowdown of 30% is observed.

Parsons and Sevcik [PS97] have done work implementing a range of scheduling disciplines on a 16-node NOW. The scheduler is implemented on top of the commercial load-balancing system LSF. They show that despite high preemption and reconfiguration overheads (over a minute for preemption), adaptive and preemptive disciplines outperform run-to-completion static scheduling for both MRT and makespan.

2.1.7 Non-Uniform Memory Access (NUMA) Machines

Somewhere in between networks of workstations and shared memory machines lies the NUMA model. Machines of this type have globally shared memory, but the memory is organized hierarchically, so that access times to different areas of memory are not always equal. Examples of this type of machine include Hector [VSLW91], developed at the

University of Toronto, and MIT's Alewife [ABC+99].

In a NUMA environment, as in our heterogeneous model, not all processors are created equal. Thus in addition to the classic *allocation* problem (how many processors to allocate), we are also faced with the *placement* problem (which processors to allocate).

Work by Zhou and Brecht [ZB91, Bre97] established that placing threads close to their data and to other threads in the job is crucial to performance. They used a construct called a "processor pool" to group processors for allocations. Unsurprisingly, the best performance was found with pool sizes that matched the underlying architecture; in this case a hierarchical design with four "stations" of four processors each. This restricted jobs to executing within a single station, making all data accesses local. As well, they found that a "Worst-Fit" policy of placing jobs in the least-loaded pool performed best, as jobs then had the most available room to expand.

2.1.8 Scheduling Research Lessons

We have presented a large amount of material related to parallel job scheduling in this section. From this mass of information, we can distill a few principles that we will use to guide us in our design of algorithms for a heterogeneous environment.

1. Among the dynamic disciplines, equipartitioning performs well when workload knowledge such as demand or speedup is not available.

2. Among adaptive disciplines, the best results have been found with disciplines that reduce processor allocations under high loads.

3. Some form of co-ordinated scheduling among job threads is necessary to allow forward progress.

4. Any knowledge of the job's service demand, even approximate, can improve scheduling through better queue ordering.

These are fairly basic guidelines that have come out of the research done over the past few years. Given more specific workload or system knowledge, it is possible to improve on these suggestions. However, since we are experimenting with a relatively new model, we will start from these assumptions and attempt to find out where they may break down in a heterogeneous system.

2.2 Workload Characterization

Over the past ten years, parallel job scheduling research has been hampered by a chronic lack of data concerning the nature of workloads from parallel computing sites. Many studies have used workloads consisting of a small number of parallel programs obtained from benchmark suites or industry, or synthetically generated workloads with somewhat arbitrary distributions. However, the situation has improved in recent years, and a clearer picture of workload distributions has begun to emerge. The major job parameters of interest in our study are inter-arrival time, runtime, degree of parallelism, and communication demand. We use the words "short" and "long" when contrasting job runtimes, and "small" and "large" when describing job parallelism.

The least well understood of these parameters is communication overhead. None of the existing studies have attempted to characterize the communication patterns of general parallel workloads, although much research exists on the problem of modeling and optimizing the communication of individual parallel programs. We have therefore used our own model of communication demand, based on data from the SPLASH-2 parallel benchmark suite [WOT+95], and explained in more detail in Chapter 4. The other parameters are discussed below.

2.2.1 Job Inter-arrival Time

The most popular choice for modeling job arrivals has been a Poisson process. Numerous studies have used this distribution [MEB88, LV90, ZM90, GST91, CMV94, PS95, CV96, Fei96, Dow97], although some have experimented with the hyper-exponential distribution [RSD+98, CV96, IPS96].

An exponential distribution is known to have a CV of 1. However, studies of workload traces from parallel computing sites have tended to find inter-arrival CV's considerably larger than 1. In the first widely cited in-depth study of a parallel job trace, Feitelson [FN95] reports that inter-arrival times from NASA Ames had a relatively high CV, which indicates that a hyper-exponential distribution may be more correct. The arrival rate varied throughout the day: the daytime CV was 3.56, the night-time was 2.11, and the weekend was 2.83.

The NASA Ames workload was compared with one from the San Diego Supercomputer Center (SDSC) by Windisch et al. [WLM+96a]. They conclude that the inter-arrival time is the most regular of the workload parameters studied, and could be modeled by "simpler probabilistic functions". Based on comments by the same authors in [LMW98], one would assume that this means an exponential distribution. The distributions at both machines were quite similar. Due to some limitations in the trace data, the actual data studied were job start times, rather than job arrival times. For interactive jobs, these times are the same. However, for batch jobs, the scheduling policy can affect how soon the submitted job actually runs. Nevertheless, the trends were similar: the daytime CV was 3.0, the night-time was 5.8, and the weekend was 3.13. (These numbers are from the technical report version of the paper [WLM+96b]).

In studying a workload from the Cornell Theory Center, Jann et al. [JPF+97] use a method that fits a Hyper Erlang distribution to the workload data. This distribution is a generalization of the hyper-exponential and Erlang distributions, with two branches and a number of stages in each branch. They found that the inter-arrival times were too dispersive to be captured by an exponential distribution, and a hyper-exponential distribution was required to fit the data.

Daily Cycle Some studies [FN95, WLM+96a] have found significant fluctuations in the arrival rate according to the time of day. When users go home at night, the arrival rate drops off sharply, and many of the large queued jobs are able to finish.

2.2.2 Job Runtime

Job runtimes have usually been modeled with a hyper-exponential distribution [PS95] and workload studies tend to support this practice. Reported CV's from workload traces are generally in the range of 2-10, although numbers as high as 70 have been reported [CMV94, FN95, WLM+96a, JPF+97].

A more interesting and less well-understood question is whether job runtimes are correlated with job sizes. A weak positive correlation (i.e., larger jobs run longer) has been reported by several authors [FN95, WLM+96a, Hot96, Fei96, LMW98], but few researchers have modeled this. One slight exception is a study of data from the Cornell Theory Center (CTC), where Hotovy [Hot96] reports a two-phase distribution of job duration. Jobs using from 2-16 processors had decreasing runtimes associated with more processors, but jobs using more than 16 processors tended to use more resources as their parallelism increased.

Many studies [FN95, SSG95, Hot96, SGS96, WMKS96] have also noted the large numbers of small, short jobs – often over half the total number of jobs. As well, they have all found that these small jobs consume a small fraction of the total resources, e.g., node-seconds or CPU cycles.

Downey [Dow97] proposes a model where the sequential lifetime L (runtime of a job on a single processor) is used to calculate the speedup of a job, and thus the runtime on n processors. Since L is generally not known, T (the wall-clock time multiplied by the cluster size) is used as an estimator. T will generally over-estimate L, since most programs have sub-linear speedup. Based on trace data from the Cornell Theory Center, Downey models T as linear in log space, or “uniform-log”. The cumulative distribution function is of the form

$$cdf_T(t) = Pr\{T \leq t\} = \beta_0 + \beta_1 \ln t$$

where β_0, β_1 are intercept and slope of the line in log space.

Jann et al. [JPF$^+$97] reported that their fitting procedure for total CPU time was unable to fit a hyper-exponential distribution to the runtimes for a workload at the Cornell Theory Center. A Hyper Erlang distribution of higher order was required in most cases to fit the data.

Lo et al. [LMW98] attempt to determine what effect the use of different synthetic models has compared to real job traces with respect to simulations evaluating scheduling disciplines. They compare two “naive” synthetic models, two synthetic models derived from analysis of job traces (due to Feitelson [Fei96] and Downey [Dow97]), and four workload traces. Although the relative ranking of all scheduling algorithms was not affected by the particular model used, some other aspects of the models were found to be important, in particular the proportion of power-of-two job sizes and the correlation of runtime with job size. System utilization tended to rise as the number of power-of-two job sizes increased. Real workload traces were found in most cases to have a weak positive correlation between job size and runtime, with the correlation coefficient running between 0.13 and 0.35. Algorithms that performed well on positively correlated workloads tended to perform poorly on negatively correlated workloads, and vice-versa.

2.2.3 Parallelism

The job size parameter has received the most widely varying treatment in the literature. This is due not only to the lack of information concerning the true distribution, but as well to the difficulty of modeling the data that does exist. The reported data in workload studies are difficult to characterize succinctly. They have a high discrete component, and tend to vary considerably from site to site.

Feitelson's early study [FN95] found a roughly uniform distribution of parallel jobs (those using more than one processor). The number of single-node jobs was much higher than any of the other sizes. The trace data was from a 128-node hypercube architecture (an Intel iPSC/860), so all of the job sizes are powers of two. There is a large number of 32-node jobs, for which no explanation is posited.

Later studies [SSG95, WLM+96a, Hot96, SGS96, Fei96, Dow97] have tended to find that the workload contains a high number of power-of-two sized jobs, although they are not required by the architecture in any of these cases. There has been speculation that the organization of the queueing system has encouraged the submission of jobs sized by powers of two [WLM+96a, Dow97]. However, in at least one study [Hot96], there does not appear to be any encouragement to have jobs of these sizes. The submittal queues are organized by estimated runtime, not number of processors, and the machine studied is an IBM SP2, which has no performance degradation for irregularly-sized jobs.

Feitelson [Fei96] studies 6 traces from different machines, some of which had been used in other studies already. Again, small jobs were more common than large jobs. Powers of two were common in all of the workloads. Other commonly occurring sizes, termed "interesting sizes", were squares, multiples of 10, and the full system size. This tended to create strong discrete components in the distribution. The author models this by using an harmonic distribution, which is adjusted by hand to account for more small jobs and interesting sizes.

2.2.4 Workload Summary

Some points emerge from the above discussion as being generally accepted, or consistent across studies. These are the main points to which we have attempted to be faithful when selecting input distributions for our simulations.

1. Arrival processes have higher variability than the traditional Poisson process, and are probably better modelled with a hyper-exponential distribution.

2. Job runtimes are highly variable, with many short jobs that require relatively few resources, and a few long jobs that use a large portion of the total resources.

3. Job runtimes show a slight positive correlation with job parallelism.

4. Job sizes tend to be multiples of powers-of-two, even when they are not required by the architecture. Other common sizes usually have some numerical or architectural significance, such as 1, squares, multiples of ten, and the full machine size.

Discussion of the workload model which we have used in our experiments is contained in Chapter 3.

Chapter 3

System and Workload Model

3.1 System Model

The system under consideration is a network of multiprocessor nodes. Processors within a node are the same speed, but processors in different nodes may have different speeds. Nodes may be of different sizes, i.e., they may have different numbers of processors (see Figure 3.1).

More formally, the network consists of a set S of nodes, $S_i, i = 1, \ldots, |S|$. Each node S_i has $|S_i|$ processors, each of speed $s(S_i)$.

A subset of the processors belonging to a node is called a *node partition*. The processors of a node partition can be dedicated to a single job j, in which case a partition N_i^j denotes a set of processors from node S_i, belonging to job j.

We define *capacity* of a set G of processors g_i, $i = 1, \ldots, |G|$ to be the sum of the speeds of the processors in the set, and use $\kappa(G)$ to denote this quantity:

$$\kappa(G) = \sum_{i=1}^{|G|} s(g_i)$$

The capacity describes the computational power of a set of processors. It can be looked at as an analogue to the number of processors in a homogeneous system.

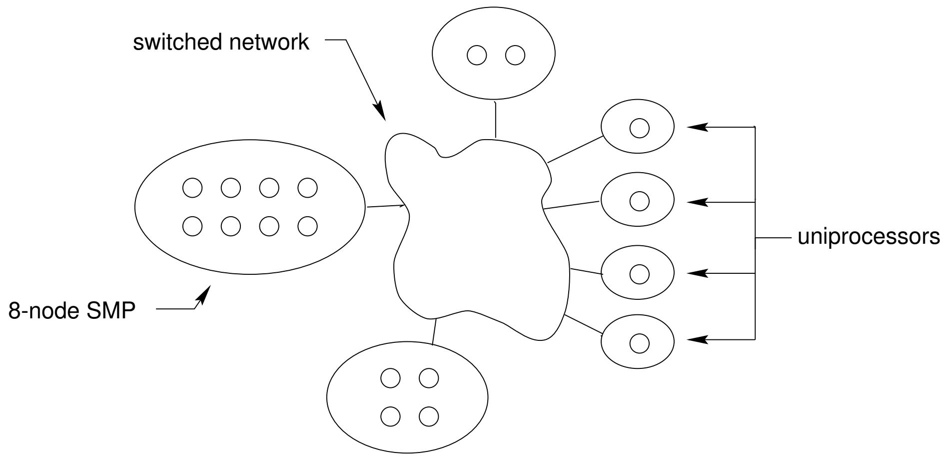


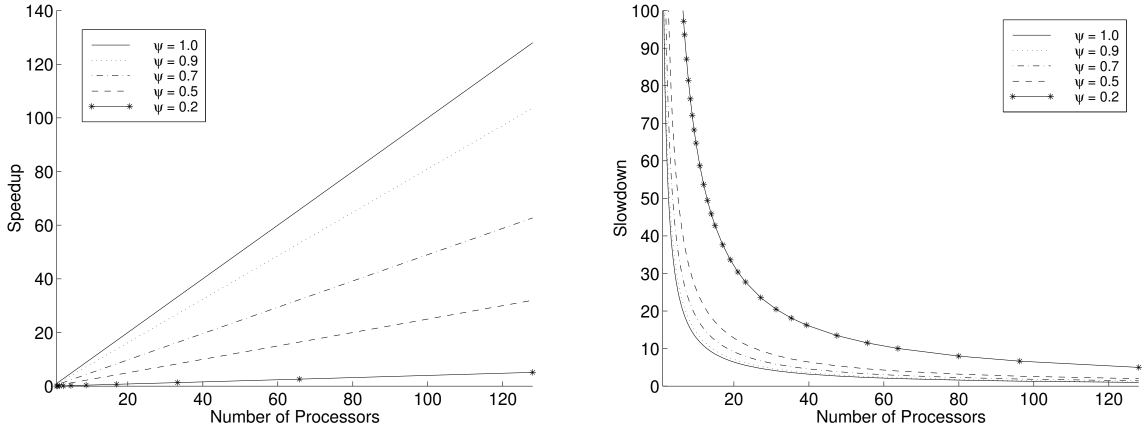
Figure 3.1: *Heterogeneous network of multiprocessors*

We assume the nodes are connected by a fast network, so that communication between any two nodes has the same cost. Network contention and congestion are not modelled. We will assume that communication between processors in the same node (local communication) is much less expensive than communication between processors in different nodes (network communication), so that local communication effectively has zero cost. Further, memory is shared inside the node, but is not shared between nodes (at least at the hardware level.) The model is thus closer to that of a network of workstations than, say, a NUMA machine.

As far as we know, this model has not been studied in the literature before. Previous work in scheduling on networks of workstations has always assumed that nodes are single processors, each of equal power. The assumptions concerning equal communication cost between nodes are common in simulation studies in NOW environments.

3.2 Job Model

In the terminology of Chapter 2, a distinction is made between *moldable* jobs that cannot change their processor allocation after starting, and *malleable* jobs that can change their processor allocation at any time. Since memory is shared at a node, jobs in our model



(a) Speedup curves for different degradations due to multiplexing of job threads.

(b) Slowdown curves for different degradations due to multiplexing of job threads.

Figure 3.2: *Effect of different values of ψ.*

may change their processor allocation within a node at any time, but are unable to change their node allocation. These jobs are malleable in a restricted sense.

Malleable jobs are not a generally supported programming model, so we will achieve malleability by folding threads when necessary. Due to context switch overhead and other factors, there is a performance cost to this option. We assume that if t threads are running on k processors of speed s, where $k < t$, then the rate of service of the threads can be modeled as

$$\frac{k}{t} s\psi$$

where ψ is a constant. A value of $\psi = 0.91$ has been used in previous work [GST91, Par97]. Speedup curves for different values of ψ are shown in Figure 3.2(a). The curves are not completely accurate, as they represent the drop from the perfect speedup that the job will have at its maximum allocation. Perhaps more accurate is a graph of slowdown for different ψ, shown in Figure 3.2(b). (Slowdown is defined as the factor by which execution increases when the number of processors is decreased.)

We need to assume that it is possible for local node schedulers to create independent partitions on a single node. Although general operating systems do not have this capability, it is not difficult to implement using a number of queues.

The minimum parallelism of all jobs is 1, as all threads of a job could be multiplexed onto a single processor. The maximum parallelism is set at runtime as the number of threads requested by the job.

We assume the jobs are programmed in single-program-multiple-data (SPMD) style and all threads will have identical service demands. The threads synchronize "frequently". (This is imprecise, but assume it is often enough that the progress of the job will be limited by the slowest thread.) The finish time of a job is then the finish time of its most slowly progressing thread. This load imbalance will not be modelled explicitly, i.e., no processors belonging to the job will be freed until all of the threads belonging to the job have finished.

3.3 Workload Model

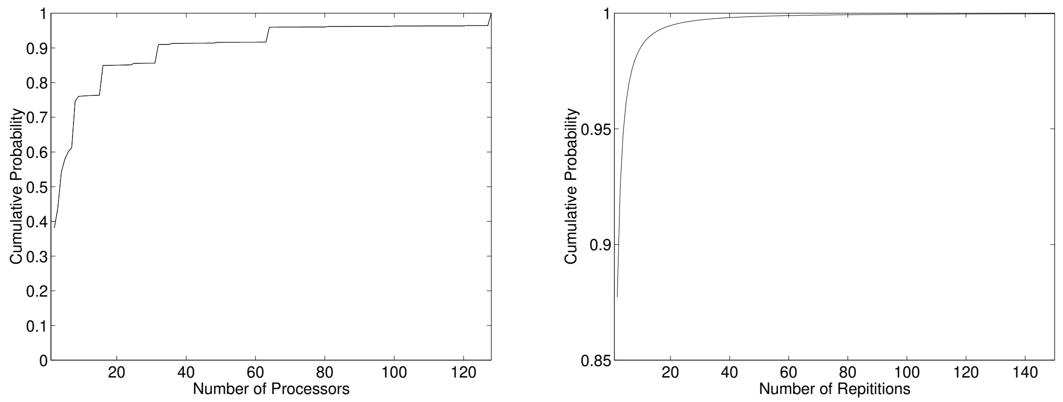
As discussed in Section 2.2, parallel workload modelling has advanced considerably in the past few years, and studies of workload traces from supercomputing centres have allowed some general conclusions to be made about parallel workloads. Our workload assumptions have been guided by these points whenever possible.

We have chosen to model the arrival process with a hyper-exponential distribution with CV of 8. The method of Sevcik et al. [SLTZ77] is used to create a distribution with the required CV. Code is included in Appendix C.

In the interests of modelling job runtimes and job sizes, we have selected a parallel workload generator based on that used by Feitelson and Jette [FJ97], and implemented based on code supplied by Feitelson from his Parallel Workloads Archive [1]. The adapted code used in our experiments is listed in Appendix B.

Feitelson's workload uses an harmonic distribution of order 2.5 for the job sizes, which

[1] http://www.cs.huji.ac.il/labs/parallel/workload



(a) Cumulative distribution function of the job size distribution used in Feitelson's distribution.

(b) Cumulative distribution function of the job repetition distribution used in Feitelson's distribution. Note that the y-axis scale starts at 0.85.

Figure 3.3: *Repetition and size distributions.*

is adjusted by hand to add extra weight for small jobs and interesting sizes such as powers-of-two, squares, multiples of ten, and the full machine size. The cumulative distribution function of the job size distribution is shown in Figure 3.3(a). The job runtime is chosen from a three-stage hyper-exponential distribution, with branches of mean 50, 900, and 20000. The probability of choosing a particular branch is based on the job size, to give a slight positive correlation between job size and runtime. A scatterplot of 100,000 jobs, with runtime plotted against job size, is shown in Figure 3.4. The predominance of the interesting sizes can be clearly seen. For the sample of jobs shown, the correlation coefficient between job size and job runtime is 0.187.

Feitelson's distribution also includes a distribution for the number of times that a job is repeated. This is modeled as an harmonic distribution of order 1.5, for which the cumulative distribution function is shown in Figure 3.3(b). This has the effect of skewing the average job demand by the number of times that a job is repeated, since these job repetitions are not new arrivals - it is as if there is one arrival with a *total* demand equal to the demand multiplied by the number of repetitions. This *total* demand must be used when calculating arrival rates that correspond to different system load factors.

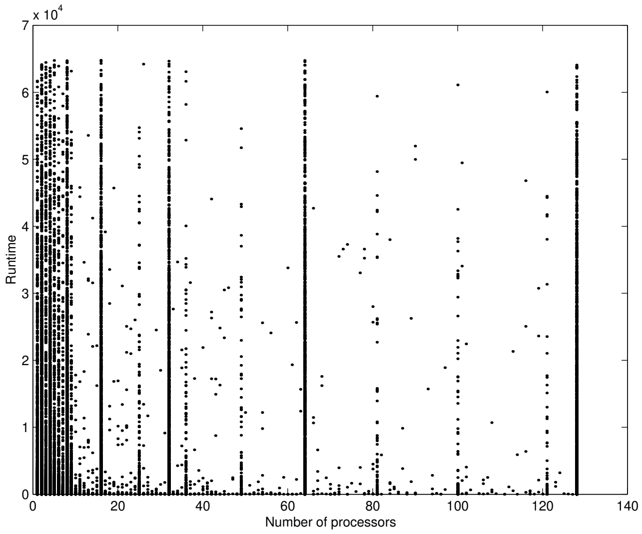


Figure 3.4: *Scatterplot of 100,000 jobs, showing runtime plotted against job size.*

3.4 Communication Model

As discussed in Section 2.2, there is little existing work on the statistical distribution of communication behaviour in parallel programs. For this reason, we have developed a simple model based on data from the SPLASH-2 parallel benchmarks [WOT+95].

A graph characterizing the ratio of communication to computation in the SPLASH-2 benchmark suite is presented in Figure 3.5. (This graph was generated from the online database of SPLASH-2 data[2].) The SPLASH-2 suite consists of shared memory applications; however this gross approximation of communication requirements is useful. Given a distributed shared memory implementation [ACD+96, SBIS98], it is even possible to implement these applications in a model such as ours, although the added overhead of maintaining the DSM environment would make our assumptions optimistic.

Discounting the local traffic, the average number of bytes communicated per instruction for each size of processor is given in Table 3.1. If we graph this table with the number of processors on a logarithmic scale as in Figure 3.6(a), then we see a roughly

[2] http://www-flash.stanford.edu

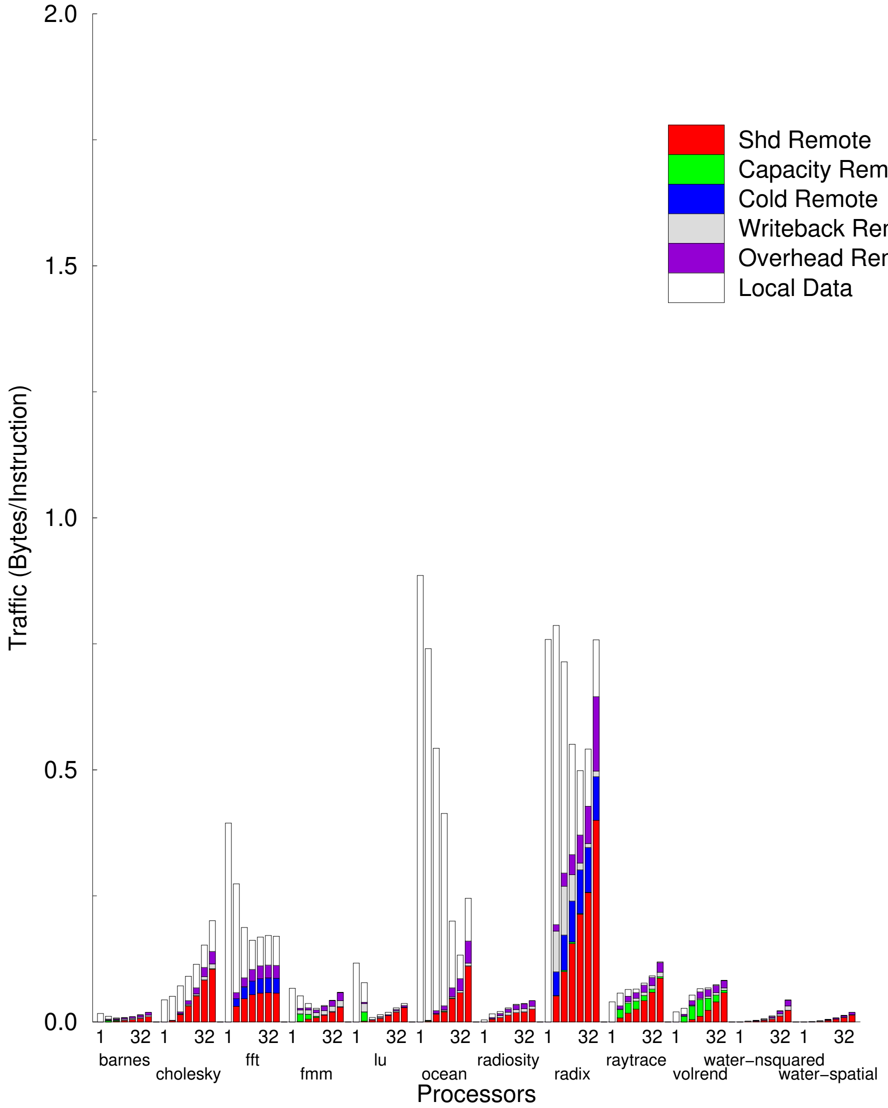


Figure 3.5: *Communication/computation ratio in the SPLASH-2 suite (generated using the online database at the SPLASH-2 website, http://www-flash.stanford.edu).*

Number of processors	Bytes sent per instruction
1	0
2	0.3899
4	0.5815
8	0.7094
16	0.8661
32	1.0493
64	1.4733

Table 3.1: *Average number of bytes sent per instruction in the SPLASH-2 benchmarks.*

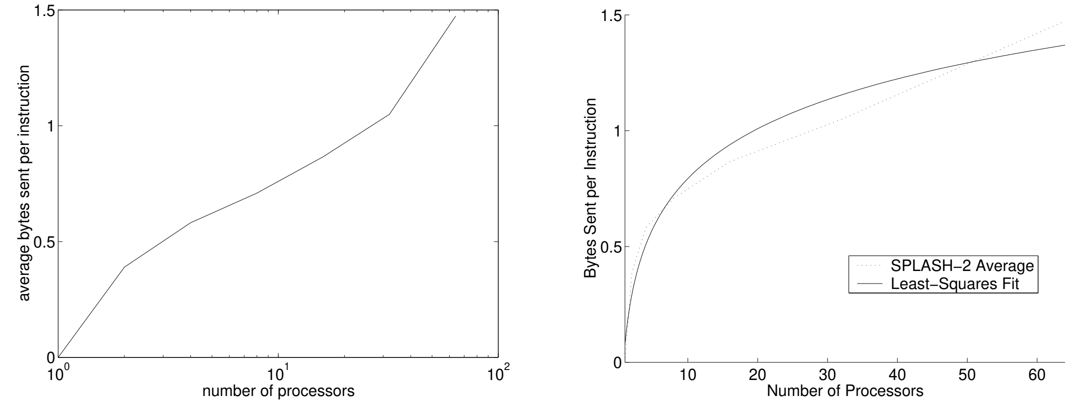
linear relationship in log space. Performing a least squares fit in log space, we get the approximate curve in Figure 3.6(b). The fit is better for smaller numbers of processors, but it does capture the basic trend: Applications using larger numbers of processors are likely to have greater communication needs. The actual curve plotted with the solid line in Figure 3.6(b) is

$$D_{comm}(x) = 0.7146(\log_{10} x) + 0.0789$$

We can use $D_{comm}(x)$ to calculate an appropriate time penalty in the simulation for jobs that must perform network communication, as opposed to local communication. We assume, for simplicity, that communication is uniformly distributed among all threads of a parallel job. This allows us to determine the amount of network communication required by a particular thread t by using the number of threads in the same job that are not in the same node as t.

Admittedly, this is a simple model that does not capture the wide variation in communication patterns that would be present in a true parallel workload. Its value for our purposes will be in simplicity. A more complex model would have to model different types of communication such as barriers, nearest-neighbours or global exchange, and the distribution and frequency of occurrence of each type. Without reliable workload data from job traces, such a complex model would be as likely to be misleading as helpful.

The unit of communication in this model is bytes per instruction. This gives a ratio



(a) Average number of bytes sent per instruction in SPLASH-2 benchmarks. (Note that number of processors is graphed on a logarithmic scale.)

(b) Least squares fit in log-space to the SPLASH-2 average communication needs.

Figure 3.6: *Derivation of the communication distribution.*

of the amount of communication to the amount of computation, but unfortunately does not include how to convert the communication into a time penalty expressed in units of computation. As a first approximation, we use a ratio of $c = 2$ for communication to computation. Using Feitelson's workload model as described in Section 3.3, this gives a relationship like that in Figure 3.7. The data in the figure assume that each thread is on a separate node, and so must perform network communication with all other threads in the job. This could be reduced by placing some threads in the same node, so the figure represents the worst case communication time. Different communication workloads can be modelled by varying the ratio c, as well by varying the shape of the curve in Figure 3.6(b).

For some insight into this model of communication, we consider a simple system with 8 uniprocessor nodes of the same speed and fractional thread distribution. Figure 3.8 shows the response time of a single job with 8 threads for various values of c and different numbers of processors. If the communication cost is much higher relative to the computation cost, we see that even using the maximum number of processors does not greatly improve the response time over just executing the job sequentially on one processor with

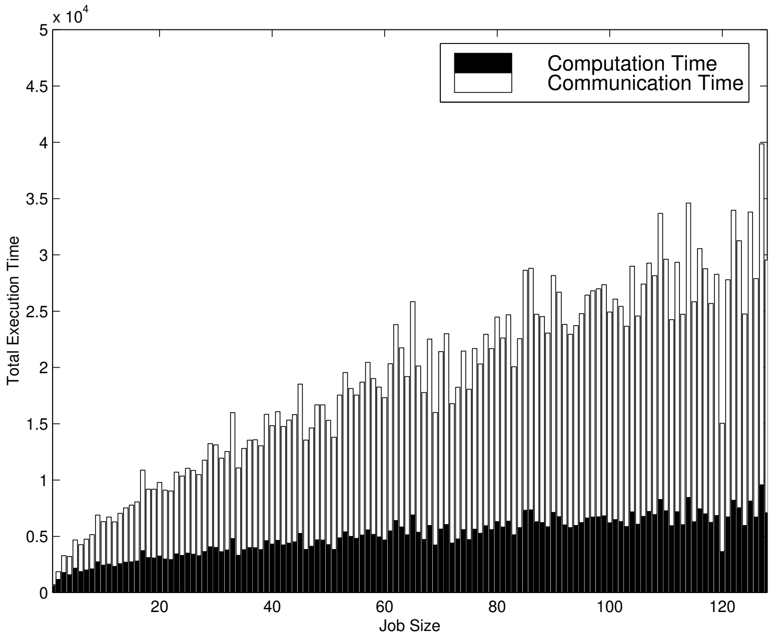


Figure 3.7: *Average computation and communication time for 10,000,000 jobs with a communication to computation ratio of 2, assuming each thread of the job is on a separate node. Note the correlations between size and computation as well as communication.*

no communication costs. However, for a smaller ratio, some improvement is always found as processors are added.

3.5 Summary of Models

We have motivated and developed models for the system, jobs, and workload under consideration.

The system consists of a network of multiprocessor nodes, each of which can be a different speed and contain a different number of processors. Jobs are malleable in a restricted sense; their processor allocations within a node can be changed at any time, but their allocation of nodes cannot change once the job has started. Jobs are composed of threads, the number of which is requested by the job and stays constant throughout the job. This requires that malleability be achieved by folding threads when there are more threads than processors; a performance penalty relative to the ratio of threads to

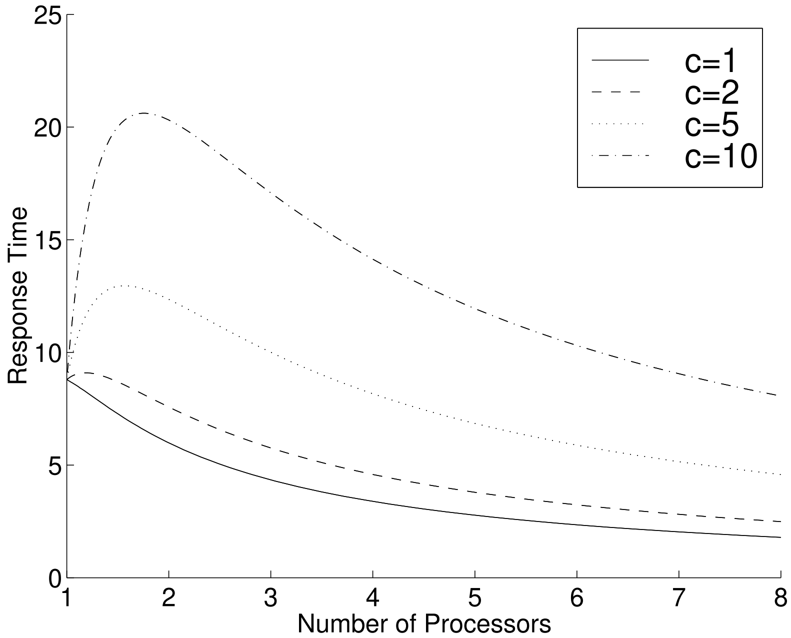


Figure 3.8: *Response time of a job with 8 threads for different values of c as processors are added. Work per thread is 1, so the fastest possible response time is 1.*

processors is applied when folding occurs.

The workload parameters are modelled as follows: Job inter-arrival times are hyper-exponentially distributed with a CV of 8, job sizes are from a harmonic distribution of order 2.5 that has been hand-adjusted to add weight at common sizes, job runtimes are from a three-stage hyper-exponential distribution and have a slight positive correlation with the job size, and job repetitions are from a harmonic distribution of order 1.5.

The communication model is developed based on data from the SPLASH-2 parallel benchmark suite. Communication demand is related logarithmically to the job size, and we define a parameter c that is the ratio between units of communication and units of work. This ratio is set as $c = 2$ for our experiments.

Chapter 4

Scheduling Policies and Experiments

In this chapter we define the policies that we use to investigate scheduling in heterogeneous environments. As well, we describe the experimental environment and the simulation assumptions and decisions that we have made.

4.1 Two Levels of Allocation

Scheduling in a heterogeneous environment raises an additional problem not encountered in traditional scheduling problems. Most algorithms previously studied give all processors the same weight (although there are location-dependent considerations in variable partitioning). Thus the problem is: *How many* processors should a given job have at any one time? In the model we consider, processors may differ in speed and node size, and so the problem becomes not only *How many?*, but *Which ones?* as well.

An additional problem in our model is thread placement. After we have solved *Which ones?*, we need to decide how many threads should be assigned to each node. We would like to minimize the completion time of each job. Since each thread has an equal demand, this is accomplished if all threads progress at the same rate. If allocation of fractional threads were allowed, then an equal rate of progress is achieved by assigning to each processor a number of threads proportional to the processor's speed over the capacity of all of the job's processors. Define the number of threads of job j assigned to a processor p as $e(j,p)$. If a job j has t_j threads to be assigned to a set of processors $G = \{g_i | i = 1...|G|\}$, we assign to processor g_i a number of threads equal to

$$e(j, g_i) = t_j(s(g_i)/\kappa(G))$$

With integer constraints, we cannot achieve this exactly. We approximate this as closely as possible by using the following algorithm:

1. Assign to each partition N_i^j a number of threads equal to the floor of the sum of the perfect thread allocation for all of the processors in N_i^j:

$$\lfloor \sum_{p \in N_i^j} e(j,p) \rfloor$$

2. Each partition's thread allocation now differs from the ideal by less than 1 thread, so there must be fewer unassigned threads than nodes. Now we assign the leftover threads one at a time to the partitions that have the greatest difference between their actual and their ideal allocation.

3. Any partitions that have fewer assigned threads than processors have their sizes reduced so there is one processor per thread. Any unused processors are returned to the pool of free processors.

4.2 The Policies

4.2.1 Adaptive Policies

Never-Span (NS) Never-Span attempts to maximize locality benefits by always running all threads of a job in the same node. Newly arrived jobs are run on the node whose set of free processors will give the fastest service rate. If there are no free processors, the job is queued. On job departures, the newly freed processors are allocated one at a time to the jobs in the queue, not going over the maximum parallelism for any job. So, if a job using 5 processors ends, and there are two jobs in the queue, the first job will receive 3 processors and the second job will receive 2.

Span-Sometimes (SS) Span-Sometimes attempts to utilize some of the supposed benefits of allowing jobs to span nodes and decrease their level of timeslicing.

Define a target job capacity κ_{target}:

$$\kappa_{target} = \frac{\sum_{i=1}^{|S|} \kappa(S_i)}{number\ of\ jobs}$$

where the number of jobs includes all jobs in the system, including running and queued jobs. Algorithms using this definition of the number of jobs have tended to outperform those that used just the number of running jobs [AS97].

To control the degree to which jobs will tend to span nodes, we define a *spanning factor*, denoted σ, which controls the attractiveness of choosing a processor in another node. This allows us to formulate an alternative definition of capacity, which subtracts some amount to compensate for the increased overhead due to communication.

$$\kappa_{span}(j) = \sum_{i=1}^{G} g_i(sp(G_i) - \sigma(\sum_{j=1, i \neq j}^{G} g_j))$$

This basically calculates the used capacity of the job, with something subtracted at each node that is relative to the amount of capacity at other nodes (and thus the amount of communication the job will have to perform at that node).

The size of the spanning factor, between 0 and 1, allows the policy to range between uncontrolled node-spanning when $\sigma = 0$ and no node spanning when $\sigma = 1$.

Span-Sometimes assigns one processor at a time to a newly arrived job, until the capacity of the job is as close to κ_{target} as possible. For each free processor p, the benefit of adding this processor to the job's allocation is calculated in the following way:

$$benefit = |\kappa_{target} - \kappa_{span}(job\ without\ p)| - |\kappa_{target} - \kappa_{span}(job\ with\ p)|$$

The free processor for which *benefit* is the greatest is selected and added to the job. The process is continued until *benefit* is negative for all free processors, the maximum number of processors have been assigned, or there are no more free processors. On job departures, jobs in the queue are treated as new arrivals in FCFS order.

4.2.2 Dynamic Policies

Dynamic Never-Span (DNS) On new arrivals, find the node for which equally partitioning all processors among the jobs in the node (plus the new job) will bring the average capacity over all jobs closest to κ_{target}.

If all jobs are running on a single processor, then no further partitioning is possible, and the job is queued.

On job departures, the same process is followed for jobs in the queue, as if they were new arrivals. If there are no jobs in the queue, the newly freed processors are divided as evenly as possible among the remaining jobs in the node.

Dynamic Span-Sometimes (DSS) We extend the Span-Sometimes policy by allowing jobs to steal processors from currently running jobs. Instead of only considering free processors, processors belonging to running jobs are also considered. If job j_{thief} is a new arrival, and processor p belongs to job j_{victim}, then the benefit of taking p away from j_{victim} and giving it to j_{thief} is calculated as:

$$\begin{aligned} benefit &= (|\kappa_{target} - \kappa_{span}(j_{thief}\ without\ p)| - |\kappa_{target} - \kappa_{span}(j_{thief}\ with\ p)|) \\ &+ \\ &\quad (|\kappa_{target} - \kappa_{span}(j_{victim}\ with\ p)| - |\kappa_{target} - \kappa_{span}(j_{victim}\ without\ p)|) \end{aligned}$$

This amounts to adding up the benefit for the victim and the thief. Should the benefit of removing p from j_{victim} be negative (thus in fact a detriment), and be larger in absolute magnitude than the benefit gained by j_{thief}, then the overall benefit will be negative, and p will not be re-assigned. Free processors compete with assigned processors, using the definition of benefit for free processors from the adaptive Span-Sometimes algorithm. Processors that are the only processor in a node belonging to a job are not considered. As in the adaptive Span-Sometimes policy, processors continue to be assigned to a new job until the benefit is negative for all processors, the job cannot utilize any more processors, or there are no more processors that are eligible to be stolen from running jobs.

4.3 The Experiments

The polices above were compared using a discrete-event simulation, consisting of about 15,000 lines of C++. Our initial decision was to write the simulation from scratch to allow the greater flexibility and avoid the learning curve of a simulation package. However, in retrospect, the use of a simulation language such as CSIM might have shortened our development time.

Job inter-arrival times were generated using a hyper-exponential distribution with a CV of 8, and 7 different means corresponding to system load factors between 0 and 1. Job work, parallelism, and repetitions were generated using the distributions of Feitelson, described in Chapter 3. Simulations were run until 200000 jobs had completed, except if the queue length increased to greater than 150000 jobs, in which case the run was terminated early. This was done to cope with rare situations in which very long queue lengths overwhelmed the memory of the simulation machine. The results for the first 500 jobs of each run were not counted, to allow a warmup period for the scheduling policies. Runs were replicated 50 times with 50 different random number seeds. Common random numbers were used for each sequence of runs, and all job parameters were calculated on job arrival, ensuring that each policy and network was evaluated with the same job arrivals. In most cases, this gave 95% confidence intervals with a half-width of less than 5% of the mean. However, some intervals are larger, especially at higher loads, due to the high variability inherent in the workload. We note that previous users of this workload [FJ97] have reported difficulty achieving tight confidence intervals as well.

Chapter 5

Results

In this section we compare the algorithms from the previous chapter on a variety of networks with different degrees and types of heterogeneity. We constrain all of the networks to have the same total capacity of 12800, so that comparisons of response time between networks can be made meaningfully. A network with no communication costs is used as the baseline comparison.

We begin by comparing non-heterogeneous, i.e. homogeneous, networks that fit into our model, to see how a few, fast processors compare against a larger number of slow processors. The algorithms are then evaluated on networks that vary in processor speed, followed by networks varying in node size. As well, we compare some random samples of networks with the same CV of speed or size, to validate our use of the CV as a basis for describing heterogeneity. To reduce the number of figures in this chapter, all networks are shown in bar graphs in Appendix A.

Our results, especially at lower loads, tend to have 95% confidence intervals with half-widths less than 5% of the mean. These intervals are shown on all graphs of mean response time. In general, where we describe similarities and differences in mean response time, these characteristics are defined as overlapping and non-overlapping confidence intervals respectively.

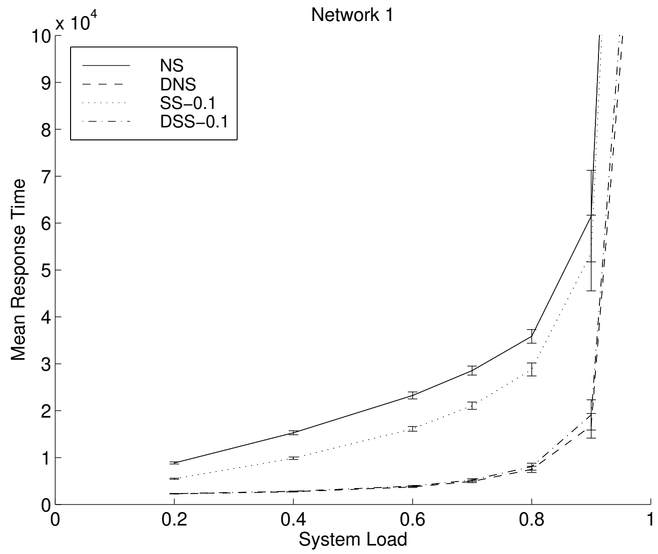


Figure 5.1: *Mean response times in the homogeneous Network 1*

5.1 The Homogeneous Case

We first compare our algorithms using a basic network, which we denote hereafter as Network 1. This network consists of one 128-processor node of speed 100. Since all processors are of the same speed, and all nodes are of the same size, this is a condition of zero heterogeneity. Network 1 was chosen as it has neither communication costs nor overhead for switching threads between processors. This allows dynamic algorithms to make full use of the network with no added cost. Performance on Network 1 can be used as a benchmark ideal situation against which to compare performance in other networks.

Mean response times for the four algorithms in Network 1 are shown in Figure 5.1. The two dynamic algorithms, DNS and DSS, perform much better than the adaptive algorithms NS and SS. This is not surprising, as both of the dynamic algorithms reduce to variants of equipartitioning in this network, and it is well known that equipartitioning can perform well if overheads are low and no other workload information is available. The two dynamic algorithms are nearly identical in performance. Between the adaptive algorithms, SS performs somewhat better than NS. This is because NS will assign as many free processors as possible to a new arrival, while SS will only assign a capacity up to κ_{target}, making a long wait in the queue more likely with NS.

To investigate the effect of adding communication costs on performance, we next compare the algorithms using various configurations of homogeneous networks with 16 nodes. These networks are numbered 35–40, and their configurations are shown in Table 5.1. Each network has 16 nodes and a total capacity of 12800, giving each node a capacity of 800 to be divided among its processors. This causes the total number of processors to vary from network to network, although the total capacity is constant.

Network	Node Size	Processor Speed	Number of Processors
35	1	800	16
36	2	400	32
37	4	200	64
38	8	100	128
39	10	80	160
40	16	50	256

Table 5.1: *Node sizes and processor speeds of the 6 homogeneous networks. All networks have 16 nodes and a total capacity of 12800.*

We graph the performance of the four algorithms on these networks in Figures 5.2 and 5.3. All four algorithms saturate faster on the networks with fewer processors. At higher load, there are not enough processors to sustain throughput when large, long jobs come along. In Figure 5.2 we see that at low loads, the NS algorithms show little difference across networks, while the performance of the Span-Sometimes algorithms (Figure 5.3) improves as the networks have fewer processors. Although these networks have fewer processors, the ones they have are faster. Thus if there are only a few jobs in the system, the spanning algorithms are able to utilize more of the faster processors and allow jobs to complete sooner, without holding up many jobs in the queue. This effect is more noticeable with the adaptive spanning algorithm SS than it is with DSS. The dynamic spanning algorithm shows the same basic trend at high load, but at medium load it is the in-between networks, 37 and 38, that perform the best.

Results Summary After simulation of our algorithms on a number of homogeneous networks, we find that at low loads, the spanning algorithms perform better on networks with fewer, faster processors, but the non-spanning algorithms show little difference across networks. However, at high loads, all four algorithms degrade much faster on the networks with fewer processors than those with a larger number of slower processors.

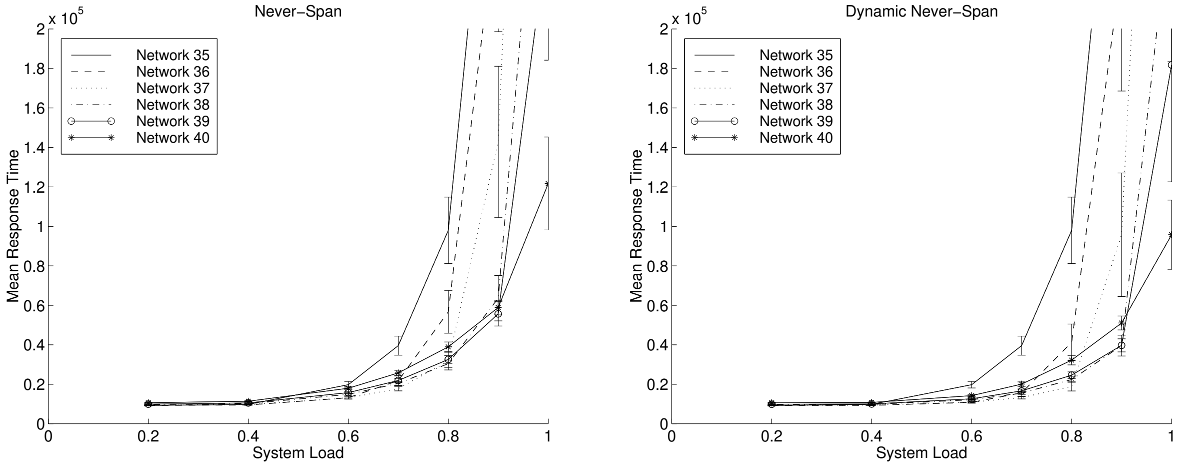
5.2 Speed Heterogeneity

The first dimension of heterogeneity we investigate is that of processor speed. If we hold the node size fixed, and vary the speed of the different nodes in the network, what happens to the relative ranking of the algorithms? How do the algorithms themselves vary as the speed-heterogeneity is increased?

As a measure of heterogeneity, we use the CV of processor speed. The node size is fixed at 8, and processor speed is varied to get three networks, numbered 11, 12, and 13, which have low, medium and high speed-heterogeneity respectively. Graphs of the processor speeds are shown in Figure A.1. The three networks have node-speed CV's of approximately 0.19, 0.43 and 0.86, respectively.

Algorithm performance on Networks 11, 12, and 13 is shown in Figures 5.4 and 5.5. Interestingly, NS actually performs better in these speed-heterogeneous networks than in the homogeneous case of Network 1. This is likely due to the fact that NS is a greedy algorithm, which will assign as many free processors as possible to an arriving job. In Network 1, this allowed jobs to monopolize large numbers of processors without releasing them. In the multi-node case, NS will limit jobs to the size of a node, leaving more processors free for other jobs.

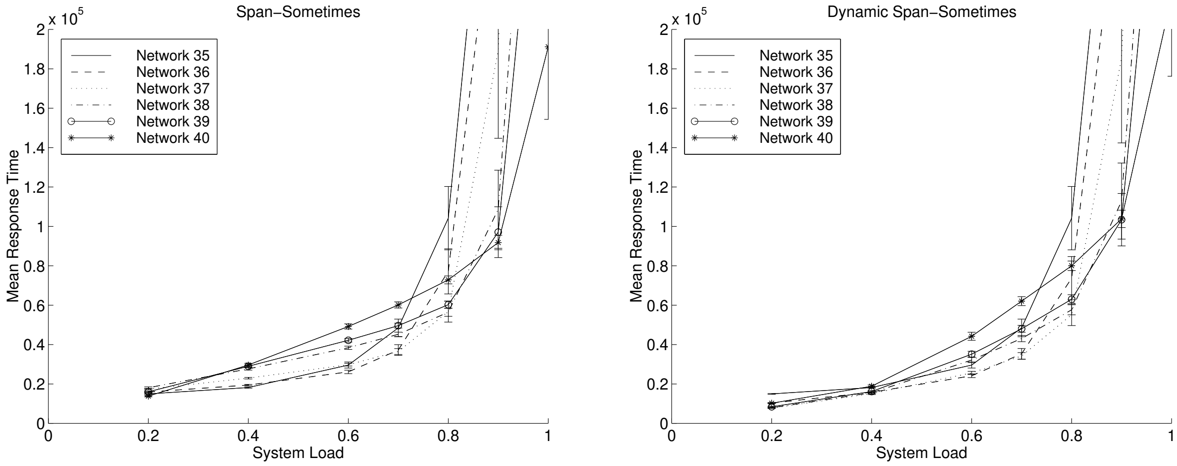
The other three algorithms, DNS, SS, and DSS show performance degradations for



(a) Never-Span

(b) Dynamic Never-Span

Figure 5.2: *Mean response time of the Never-Span algorithms on the homogeneous Networks 35–40.*



(a) Span-Sometimes

(b) Dynamic Span-Sometimes

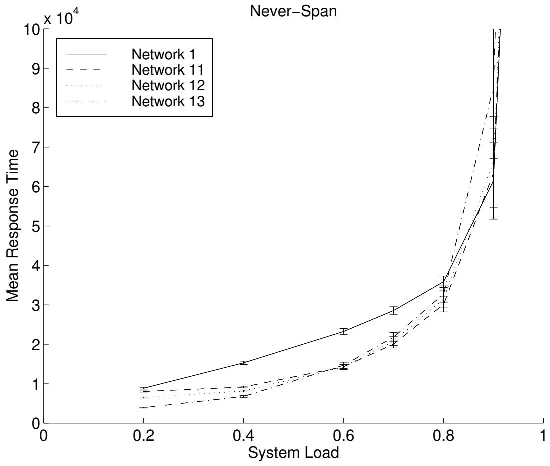
Figure 5.3: *Mean response time of the Span-Sometimes algorithms on the homogeneous Networks 35–40.*

the speed-heterogeneous networks compared to Network 1. In the case of NS and DNS at low loads, the more speed-heterogeneous the network, the better the mean response time. At medium to high loads, this situation is reversed: higher mean response times are observed as the speed-heterogeneity of the network increases. At low loads, there tend to be so few jobs in the system that most jobs end up using the faster processors in the higher end of the network. As load increases, more and more jobs are forced to execute on the slower processors, increasing mean response time.

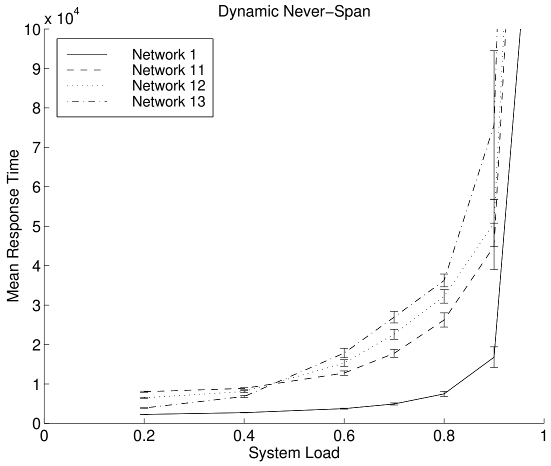
We do not observe this crossover effect with SS and DSS, shown in Figure 5.5. For all loads, the mean response time is the same (statistically speaking) or greater as we increase the speed-heterogeneity.

The next question is how the performance of the algorithms relative to each other is affected by the increasing speed heterogeneity. Figures 5.6 and 5.7 compare the performance of the algorithms on Networks 1, 11, 12 and 13. (Figure 5.6(a) is identical to Figure 5.1, but is shown again for ease of comparison.) In the homogeneous case, the split in performance was between static and dynamic policies. However, in the speed-heterogeneous networks, NS goes from being the worst policy to one of the best policies, while DSS, which had good performance in Network 1, has relatively poor performance in Networks 11–13. This makes the split occur between policies that are node-spanning or not, while in Network 1 it was between policies that were dynamic or not. The magnitude of the difference in performance between node-spanning and non-node-spanning algorithms increases with the degree of speed-heterogeneity. One somewhat inexplicable result is the fact that as the networks become more speed-heterogeneous, NS moves from slightly underperforming DNS in Network 11 to outperforming it in Network 13.

The dynamic spanning algorithm, DSS, outperforms its adaptive version, SS, for low and medium loads, but at high loads their performance becomes indistinguishable. The gap between the two algorithms at low to medium load increases with the degree of speed heterogeneity. This is because the performance of SS degrades faster with increasing

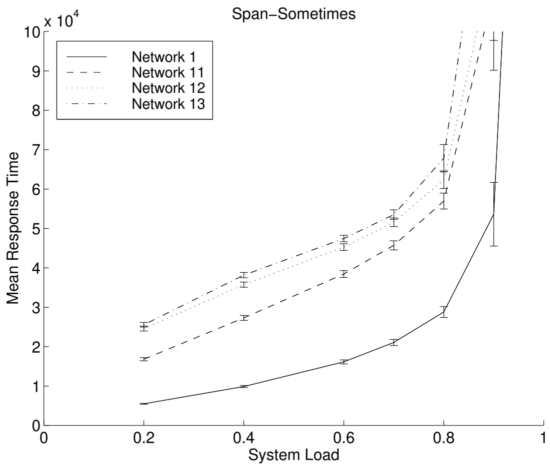


(a) Never-Span

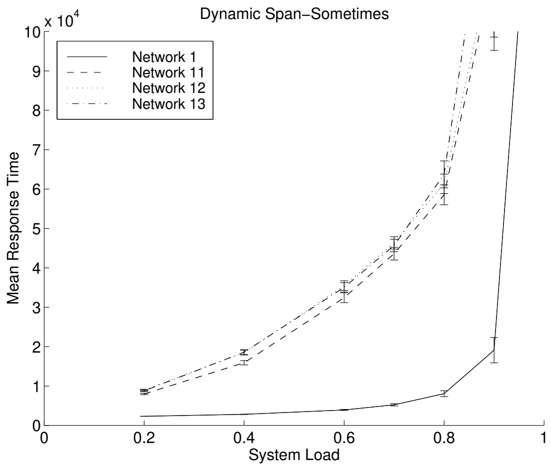


(b) Dynamic Never-Span

Figure 5.4: *Mean response time of the Never-Span algorithms on Networks 1,11,12, and 13.*



(a) Span-Sometimes



(b) Dynamic Span-Sometimes

Figure 5.5: *Mean response time of the Span-Sometimes algorithms on Networks 1,11,12, and 13.*

speed heterogeneity than does DSS, as can be seen by comparing Figures 5.5(a) and (b).

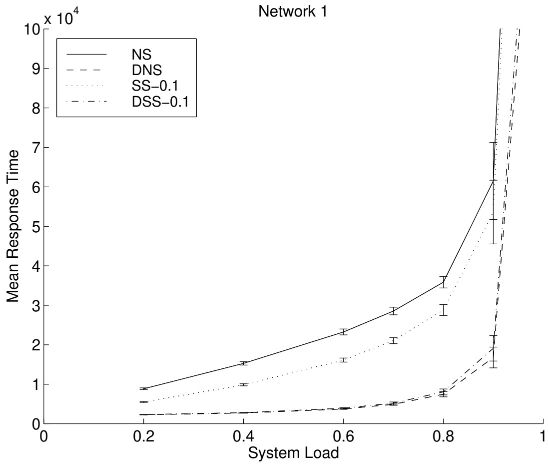
Results Summary Our investigation of speed-heterogeneous networks shows that the non-spanning algorithms exhibit a crossover effect, where they perform better on more speed-heterogeneous networks at low loads, but worse on more speed-heterogeneous networks at high loads. The node-spanning algorithms do not show this effect, but instead have worsening performance at all loads as the speed-heterogeneity increased. As well, the non-spanning algorithms match or outperform the spanning algorithms at all loads.

5.3 Node-Size Heterogeneity

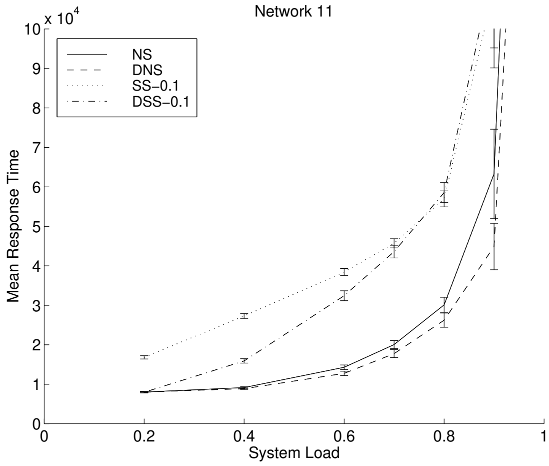
The next dimension of heterogeneity that we investigate is the number of processors per node. Similarly to the previous section, we create three networks where all nodes are speed 100, but the size of the nodes are varied. These three networks are numbered 14–16, and are shown in Figure A.2. They exhibit low, medium and high degrees of size-heterogeneity, with node-speed CV's of approximately 0.48, 0.85 and 1.21, respectively.

The results of comparing the performance of the four algorithms on Networks 1 and 14–16 are shown in Figures 5.8 and 5.9. As in the speed-heterogeneous networks, NS performs better in the size-heterogeneous case than the homogeneous case due to the limitation of node size. There is also a slight tendency at low load for NS and DNS to perform better on more size-heterogeneous networks. This also parallels the speed-heterogeneous case, and is presumably due to more jobs utilizing the larger nodes when the system is lightly loaded.

However, unlike the speed-heterogeneous case, at medium load there is almost no difference in relative performance for the algorithms across different degrees of size-heterogeneity. The exception to this rule is DSS, which has equal performance at low load for all three networks, but shows decreasing performance as the size-heterogeneity increases at medium and high loads. The fact that three of four algorithms show almost

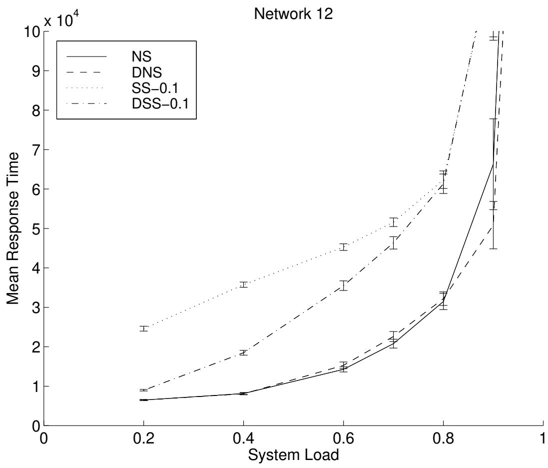


(a) Network 1

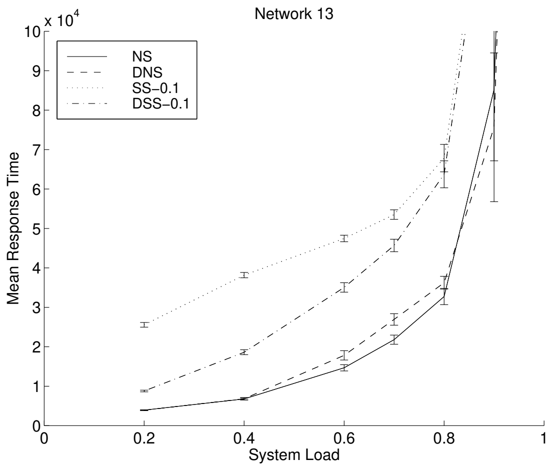


(b) Network 11

Figure 5.6: *Mean response time of all four algorithms on Networks 1 and 11. Note that in a), it is DNS and DSS that have the best performance, but in b) it is NS and DNS.*

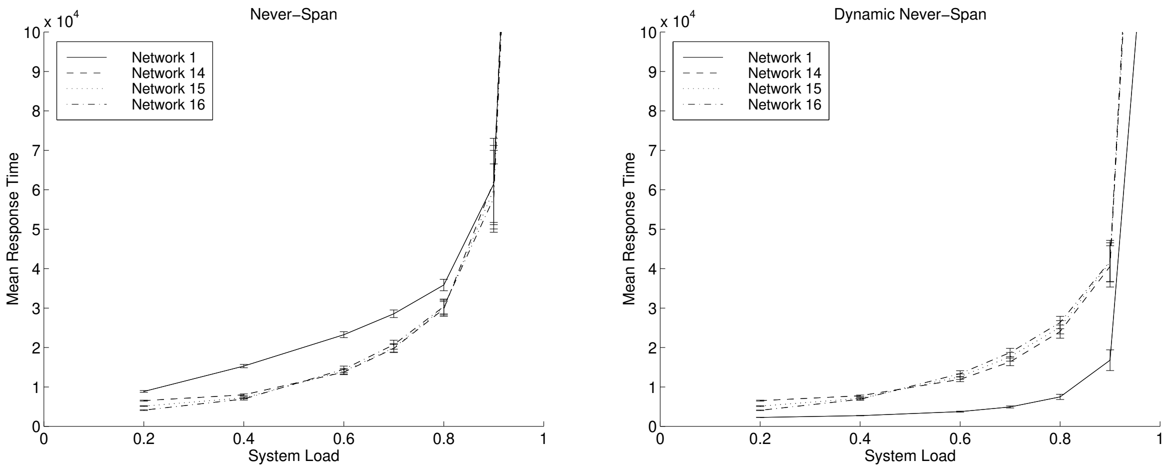


(a) Network 12



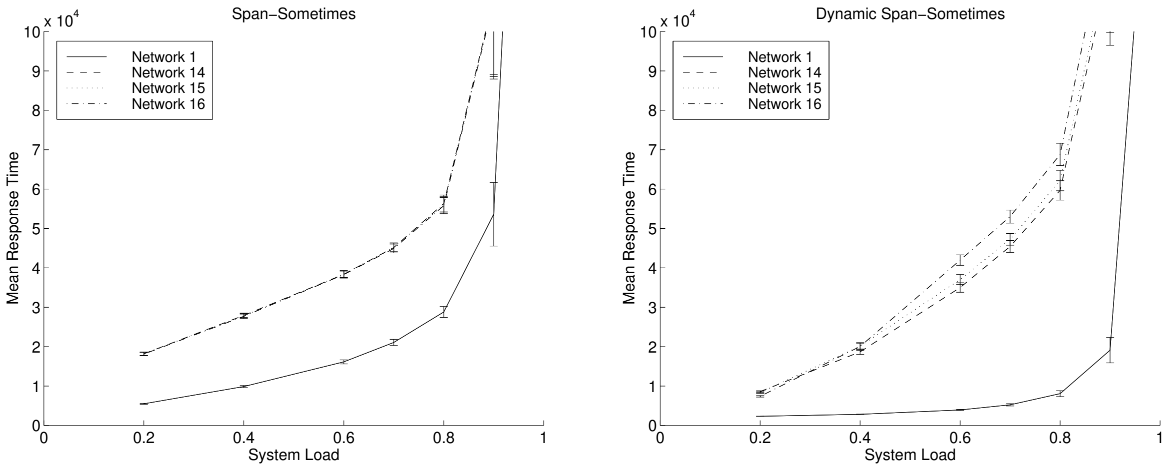
(b) Network 13

Figure 5.7: *Mean response time of all four algorithms on Networks 12 and 13.*



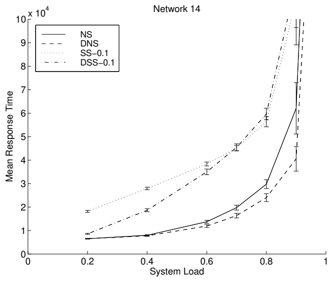
(a) Never-Span (b) Dynamic Never-Span

Figure 5.8: *Mean response time of the Never-Span algorithms on Networks 1,14,15, and 16.*

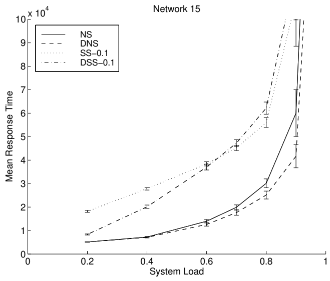


(a) Span-Sometimes (b) Dynamic Span-Sometimes

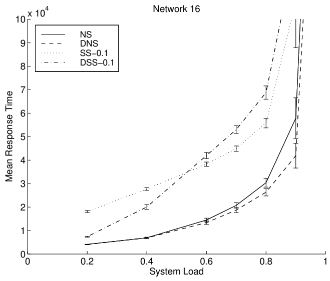
Figure 5.9: *Mean response time of the Span-Sometimes algorithms on Networks 1,14,15, and 16.*



(a) Network 14



(b) Network 15



(c) Network 16

Figure 5.10: *Relative performance of all four algorithms on Networks 1,14,15, and 16.*

no difference in mean response time is interesting, as the degree of node-size heterogeneity is high, especially in Network 16, which has a CV of around 1.2, and a range of 32. Note especially the case of SS in Figure 5.9(a), where performance is identical on all three networks, despite this high heterogeneity.

Next we turn to the relative performance of the algorithms on the different size-heterogeneous networks. The ordering of the algorithms is shown for the three networks 14,15,16 in Figure 5.10. At low loads, the algorithm ordering is the same as for the speed-heterogeneous networks, i.e., NS and DNS are best, followed by DSS and then SS. However, while in the speed-heterogeneous networks, DSS and SS have similar performance at high loads, in the size-heterogeneous networks SS actually outperforms DSS at high loads. The magnitude of this difference increases as the networks become more heterogeneous. This is an interesting result, as we would expect a dynamic algorithm to outperform its adaptive counterpart at all loads.

Results Summary We determine from experimenting with size-heterogeneous networks that there is little or no statistically significant differences in performance across the networks for three of the four algorithms. The fourth algorithm, DSS, shows worsening performance as the networks become more size-heterogeneous, resulting in DSS being outperformed by its adaptive counterpart SS at medium to high loads.

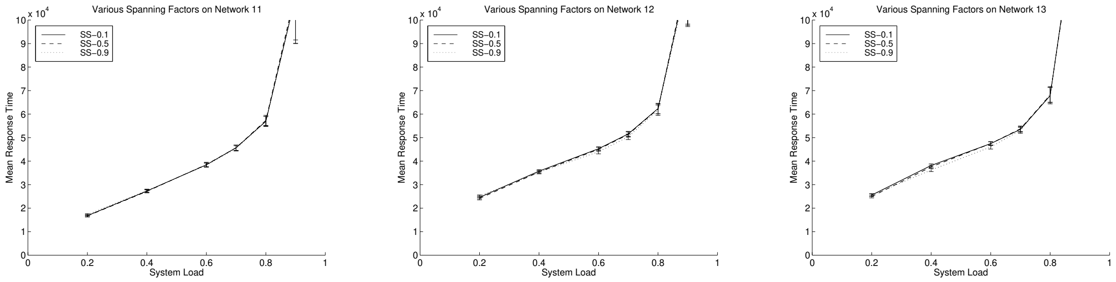
5.4 Effect of the Spanning Factor σ

The two Span-Sometimes algorithms use a spanning factor (denoted σ) that controls the attractiveness of choosing processors in other nodes. We compare the effect of this parameter on the mean response time for the values of $\sigma = 0.1$, 0.5 and 0.9. Recall that when $\sigma = 0$ no spanning will occur, and when $\sigma = 1$ processors will be chosen with no consideration of the node they are in.

Our results show that the spanning factor makes little difference to the mean response time for both SS and DSS. Results are presented in Figures 5.11, 5.12, 5.13 and 5.14. For these reasons we show only the results for $\sigma = 0.1$ in the other sections of this chapter.

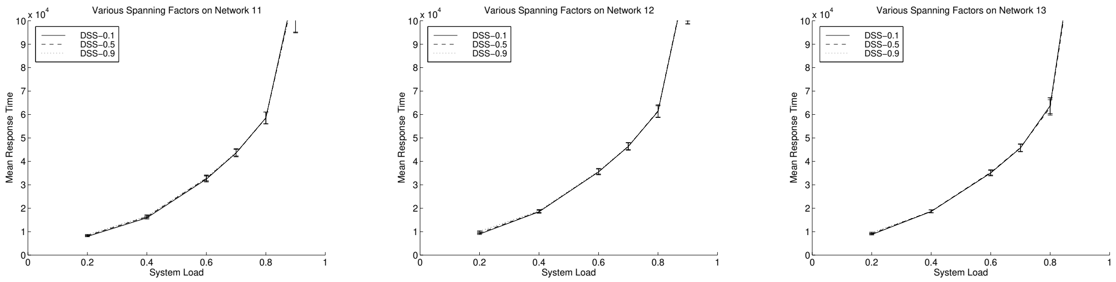
The only result that differs in this section is in Figure 5.13(a) where the performance of SS-0.9 on Network 14 is somewhat worse than SS-0.5 and SS-0.1. This effect is statistically significant, but we have not performed a deeper analysis to determine the reason for its occurrence.

Although varying the value of σ does not affect the mean response time of the algorithms, it does affect the average partition size. We illustrate this with the SS algorithm on our six networks in Figures 5.15 and 5.16, but the results for DSS are similar. The speed-heterogeneous networks 11–13 behave as expected: partition sizes are larger for larger values of σ. However, the speed-heterogeneous networks show the opposite trend: partition sizes are smaller with larger values of σ. This result may be caused by the method SS and DSS use to achieve their target capacity. Processors are added to a new job until the job is as close as possible to the target capacity. In a size-heterogeneous network, all processors are of the same speed, and thus have the same value when calculating capacity. The spanning factor changes this by changing the value of processors outside the current node. A low value of σ allows the algorithm to come closer to the target capacity by using a few low-valued processors, whereas a higher value of σ makes all processors appear equal and thus the algorithm is less likely to add more partitions to come closer to the target capacity.



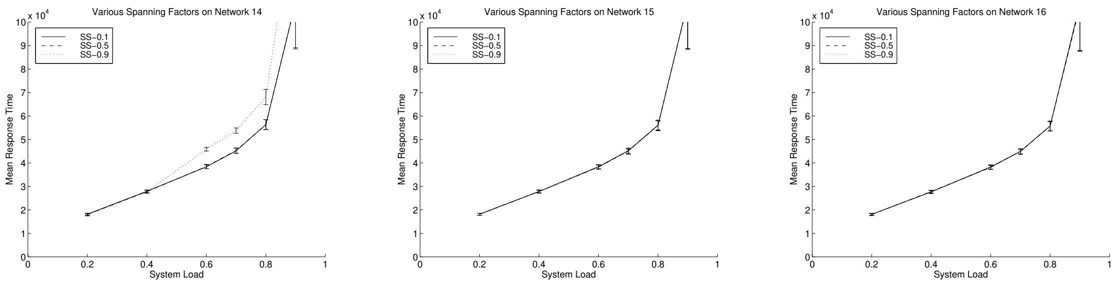
(a) Network 11 (b) Network 12 (c) Network 13

Figure 5.11: *Performance of SS for different values of* σ *on size-heterogeneous networks 11,12, and 13.*



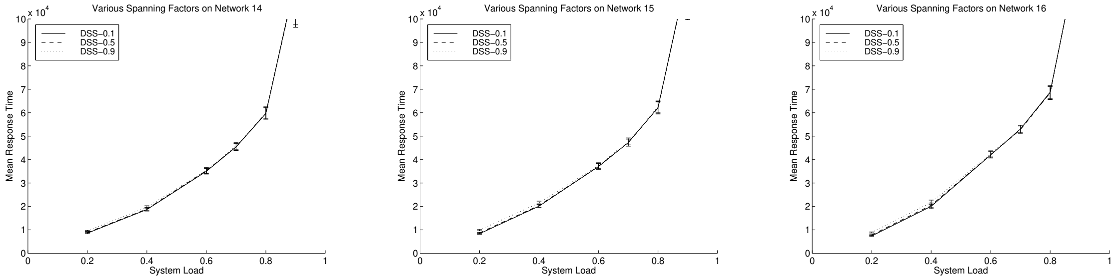
(a) Network 11 (b) Network 12 (c) Network 13

Figure 5.12: *Performance of DSS for different values of* σ *on size-heterogeneous networks 11,12, and 13.*



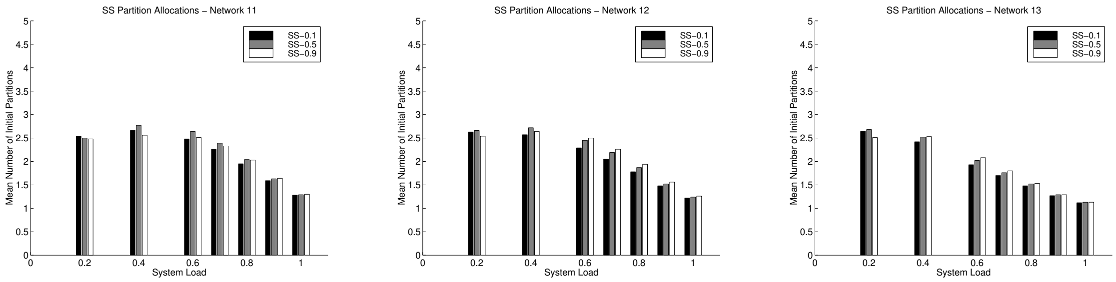
(a) Network 14 (b) Network 15 (c) Network 16

Figure 5.13: *Performance of SS for different values of* σ *on 14,15, and 16.*



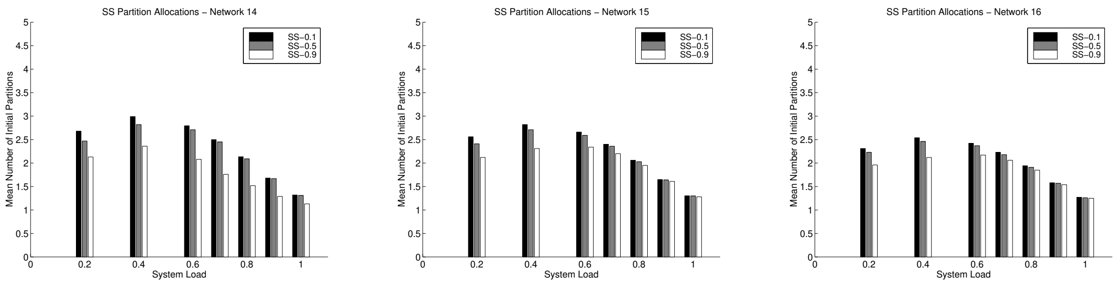
(a) Network 14 (b) Network 15 (c) Network 16

Figure 5.14: *Performance of DSS for different values of σ on 14,15, and 16.*



(a) Network 11 (b) Network 12 (c) Network 13

Figure 5.15: *Mean number of partitions per job on the speed-heterogeneous networks 11–13 for the SS algorithm.*



(a) Network 14 (b) Network 15 (c) Network 16

Figure 5.16: *Mean number of partitions per job on the size-heterogeneous networks 14–16 for the SS algorithm.*

Results Summary Simulation of different values of the spanning factor show that it has no effect on scheduling performance. Although it does affect the mean partition sizes allocated to jobs, this happens in unexpected ways: a high spanning factor results in a larger mean partition size on the speed-heterogeneous networks, but in a smaller mean partition size on the size-heterogeneous networks.

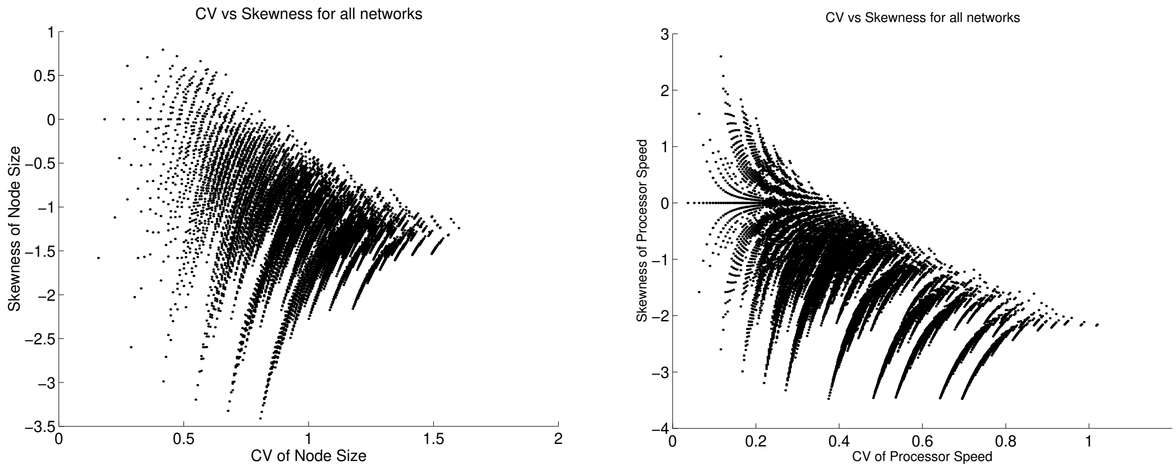
5.5 CV As a Performance Predictor

In the experiments of the previous sections, we investigated performance on networks that have different CV's of speed and size. This involves an implicit assumption that the CV is a useful measure for differentiating between networks. However, it is possible for two networks with the same CV to have different characteristics, and thus perhaps different performance.

To investigate this, we randomly selected networks with the same CV and capacity, and used them to compare our algorithms' performance. Since there are an infinite number of networks having any given CV and capacity, we reduced the search space by considering only a limited number of possible speeds and sizes, and constraining all networks to have exactly 16 nodes. We hypothesize that higher moments could describe different types of network configurations, so we calculated the skewness[1] for all networks as well. The distribution of possible speed- and size-heterogeneous networks is shown in Figure 5.17, with the CV of node size or speed graphed against skewness. The constraints of 16 nodes and a total capacity of 12800 tend to create patterns in the distribution.

To compare otherwise different networks of equal CV, we randomly selected networks from vertical cross-sections of these graphs. Figures A.3–A.8 show the resulting networks. The relative range of possible node sizes (1–32) was much larger than that for processor

[1]Using the formula: $\frac{\sum_{i=0}^{n}(\hat{\mu}-x_i)^3}{(n-1)\hat{\sigma}^{3/2}}$, where $\hat{\mu}$ is the sample mean, $\hat{\sigma}$ is the sample variance, and n is the sample size.



(a) Size-heterogeneous networks (b) Speed-heterogeneous networks

Figure 5.17: *Skewness versus CV for the two spaces of possible heterogeneous networks.*

speeds (60–300). This allowed the size-heterogeneous networks to be more heterogeneous (maximum CV $\approx$ 1.6) than the speed-heterogeneous networks, which had a maximum CV $\approx$ 1.0.

5.5.1 Size Heterogeneity

We randomly selected nine networks with CV $\approx$ 1.2 from those shown in Figure 5.17(a), three with low skew, three with medium skew and three with high skew, numbered 17–25. As hypothesized, the skew seems to provide a general characterization of the distribution of network sizes. This is illustrated in Figures A.3, A.4 and A.5. A lower skew indicates that there is more capacity contained in the larger nodes. Note that we are not advocating the use of higher moments such as the skew to precisely describe the characteristics of heterogeneous networks. This is merely a convenient method to create classes of heterogeneous networks with similar looking distributions.

Performance results on these 9 networks for the Never-Span algorithms (NS and DNS) are presented in Figure 5.18. They show almost no difference in performance across the networks.

Results for the two Span-Sometimes algorithms are shown in Figure 5.19. The adaptive algorithm SS shows similar results to the NS algorithms: there is almost no variation among the networks. The dynamic version, DSS, does show a small difference: Networks 23-25 perform better at low and medium loads than the other networks. These networks have higher (less negative) skewness, and thus a higher proportion of larger nodes. However, the difference is not statistically significant. We mention it here only because it is the only case where the results are less than identical.

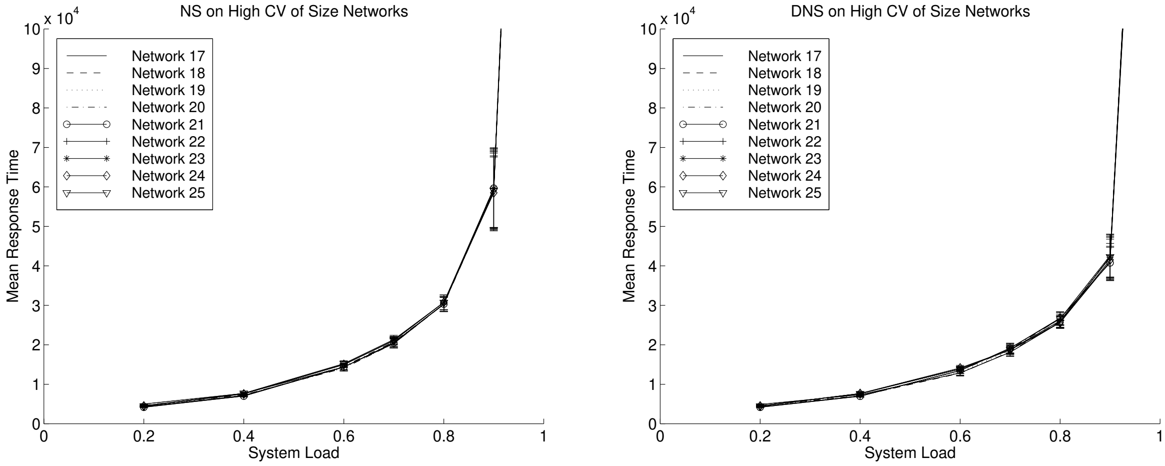
5.5.2 Speed Heterogeneity

As in the previous section, we begin by selecting 9 networks with processor-speed CV of 0.6 from the space of possible 16-node networks having a total capacity of 12800, depicted in in Figure 5.17(b).

Three of these networks have low skewness, three medium skewness, and three high skewness. They are numbered Networks 26-34 and are shown in Figures A.6, A.7 and A.8.

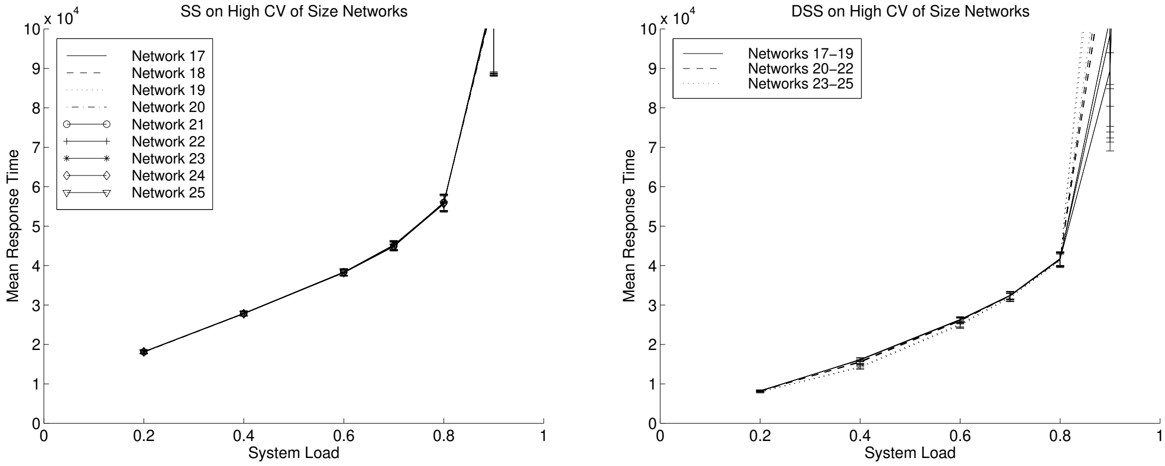
Results of the four algorithms on Networks 26–34 are presented in Figures 5.20 and 5.21. As in the size-heterogeneous case, there is almost no difference in performance for any of the algorithms, validating our use of the CV as a basis for network comparison.

Results Summary Simulation on samples of speed-heterogeneous and size-heterogeneous networks indicate that the CV is a good predictor of scheduling performance for our algorithms. Although far from conclusive proof, we feel this offers some support for the comparisons made in our study.



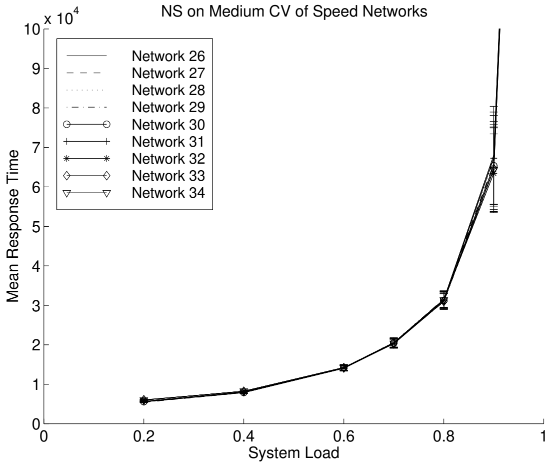
(a) Never-Span (b) Dynamic Never-Span

Figure 5.18: *Performance of NS and DNS on networks with high CV of node size and varying skewness.*

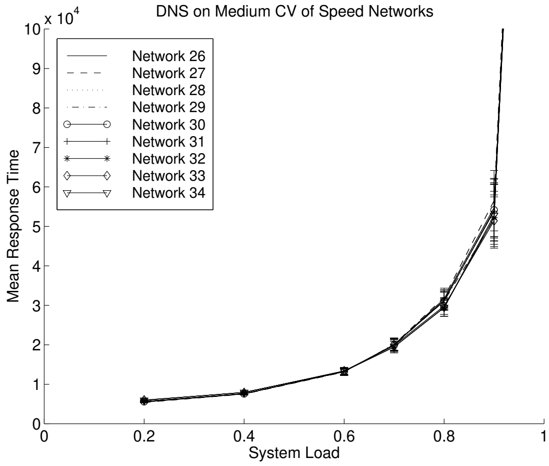


(a) Span-Sometimes (b) Dynamic Span-Sometimes

Figure 5.19: *Performance of SS and DSS on networks with high CV of node size and varying skewness.*

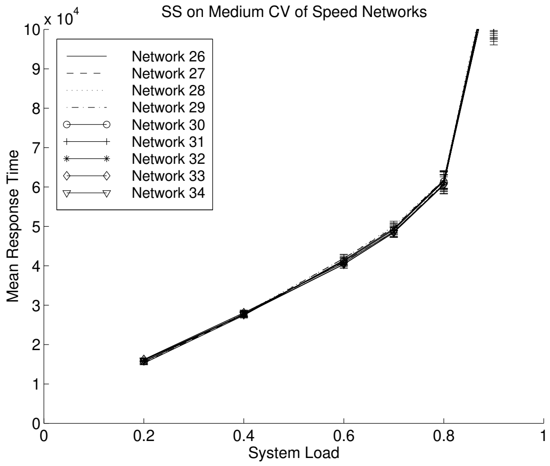


(a) Never-Span

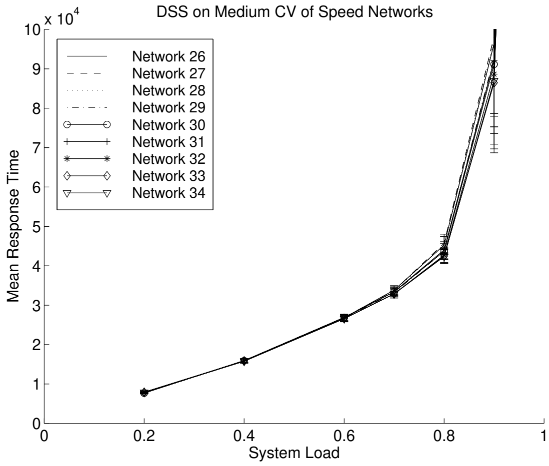


(b) Dynamic Never-Span

Figure 5.20: *Performance of NS and DNS on networks with medium CV of node speed and varying skewness.*



(a) Span-Sometimes



(b) Dynamic Span-Sometimes

Figure 5.21: *Performance of SS and DSS on networks with medium CV of node speed and varying skewness.*

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Initially, we compared various homogeneous networks having the same overall capacity and number of nodes, but varying in the number of processors per node, and thus the total number of processors. Since the total network capacity was limited (to 12800), networks with a larger number of processors had slower processors. We found that at low loads, the spanning algorithms benefited from the faster processors, and so performed better on the networks with fewer, but faster, processors. As load increased, both the adaptive and the dynamic algorithms saturated much faster on the networks with fewer processors.

This leads us to believe that it may be beneficial to construct a heterogeneous network, using a few fast processors and a large number of slow processors. Networks containing only fast or only slow processors both performed well, but at different loads, suggesting that a heterogeneous network made up of some of each kind might be able to perform well at both low and high load. Older, slow processors are often scrapped in favour of new ones, but this argument might present some reasons for keeping them in a network, if only to alleviate the pressure at high loads.

Some previous work has had similar results. Sauer and Chandy [SC81] performed a queueing analysis of a two-processor system versus a one-processor system, where the dual processors have half the processing speed of the single processor. They found that a job with a long service time interfered less in the two-processor case, as throughput could still be maintained on the other processor. Classical queueing theory results due to Kleinrock [Kle76] indicate that as servers are added (with total system capacity remaining the same), mean waiting time decreases, but mean processing time increases (where response time is the sum of waiting and processing times). Although our result seems counter-intuitive in light of classical results, our intuition is that it is likely due to the specifics of the workload in question, which has large numbers of small, short jobs, and a few very long, very large jobs. With a large network, short jobs are almost always able to run, albeit at a slower rate. In a small network, it is easy for a large, long job to tie up all of the processors in the network for a long period of time, forcing many short jobs to wait. The tradeoff is that the extra waiting time due to queueing is not made up by the faster processing speed for many jobs, causing the overall response time to increase.

When we held the node size fixed, and varied the processor speed in the network, we found that at low load, the non-spanning algorithms performed better on the more speed-heterogeneous networks, but at medium to high loads, the trend was reversed, with better performance on the less speed-heterogeneous networks. The spanning algorithms did not have this crossover tendency, but instead had worsening performance at all loads as the speed-heterogeneity increased.

We then investigated node-size heterogeneity by fixing the processor speed and varying the number of processors per node. The non-spanning algorithms showed a very slight tendency to perform better on more heterogeneous networks at low load, but worse at higher loads. This is similar to the results for speed-heterogeneous networks, although the magnitude of the effect is less pronounced. At medium load, there is little or no statistically significant difference for all algorithms except DSS, which had higher mean response times as the node-size heterogeneity increases. The performance of DSS is actually worse than that of SS on the more size-heterogeneous networks.

The preceding two paragraphs provide evidence that the increased speed or size available in the high end of a heterogeneous network is beneficial at low loads, but at higher loads, the increased use of the slower or smaller end of the network becomes detrimental.

To support our use of the CV as a basis for comparing network heterogeneity, we sampled from the space of heterogeneous networks of a given CV with 16 nodes and a capacity of 12800. We found that all of the algorithms perform nearly identically on networks with similar CV's, even when the actual distribution is different, as indicated by the skewness. This result was observed for both speed and node-size heterogeneity, providing some evidence that the CV is a good predictor of scheduling policy performance for these algorithms in our model.

We note that for all networks studied, the simple non-spanning algorithms NS and DNS outperformed the more complex spanning algorithms SS and DSS. This was somewhat unexpected, as our original hypothesis was that node-spanning would be beneficial in some cases. However, this does not show that spanning is detrimental, only that our approach to spanning does not work as well as desired. As well, our communication model incurred fairly high overheads for communicating jobs. In the Future Work section below, we discuss how further study with different communication models and various overheads could shed more light in this area.

The better performance of the non-spanning algorithms recalls the results of Brecht in a NUMA environment [Bre97], where the best performance was found by restricting jobs to pools that matched the underlying architecture. This effectively amounted to using a non-spanning algorithm, although in a shared-memory environment rather than a message-passing one.

Our observations indicate that processor-speed heterogeneity has a greater effect on scheduling than node-size heterogeneity. This may be due to some limitations in our model, since we do not model bottlenecks at the network interface or memory bus level. Leaving that aside, we find that trends existing in situations of low heterogeneity are often magnified by increased heterogeneity (as measured by the CV of speed or node size). We conclude that heterogeneity cannot be ignored when designing scheduling algorithms.

As well, the most dynamic and flexible algorithm, Dynamic Span-Sometimes, failed to perform as well as expected. At higher loads, DSS was as bad or worse than its adaptive counterpart SS, for both size-heterogeneous and speed-heterogeneous networks. This unexpected result indicates that the careful balancing of threads performed by SS may be more important than the pre-emptive flexibility given by DSS. At higher loads, when DSS steals processors from running jobs and causes them to be unbalanced, this may offset the benefit gained from starting some jobs earlier. This result is interesting, as the dynamic algorithm is outperformed by the adaptive algorithm for a different reason than usual. Normally, one would expect a dynamic algorithm to outperform a similar adaptive algorithm whenever the costs of pre-emption are not too high. In this case, the pre-emption performed is cheap, but it is the upsetting of the balancing scheme of threads to processors that is the cause of the dynamic policy's performance degradation.

6.2 Future Work

The field of heterogeneous scheduling remains largely unexplored. We have investigated only two of the possible dimensions of heterogeneity, and only in isolation. In this section we list some areas and approaches that would be reasonable directions to extend our work.

A truly heterogeneous network would probably vary in more than the single dimensions investigated here. It would be interesting to see if any of our results hold on networks that vary both in processor speed and node size. For example, we noted a small effect on networks that vary only in node size: does this change when the network also varies in processor speed? Our approach in this thesis involved enumerating many of the possible networks composed of certain sizes or speeds, and then randomly choosing a subset with which to experiment. This method is probably computationally infeasible if the network is heterogeneous in two or more dimensions. Therefore, we would likely have better success with randomly generating networks until we have a sufficiently large subset within a certain tolerance of the desired point.

Our work provides some evidence that the CV is a good predictor of scheduling performance, at least for the networks, algorithms, and workloads considered. However, more extensive work is required to prove or disprove this hypothesis. Consideration of more types of scheduling algorithms and a wider range of possible networks would make this result much more useful to future system designers. It would also be interesting to see if other measures of variation, say inter-quartile range, or simpler ones such as minimum and maximum provide any kind of scheduling predictive power.

As well, experimentation with networks that are heterogeneous in multiple dimensions would also require the use of more than one CV to describe the heterogeneity of the network, an approach that does seem unlikely to scale as more dimensions of heterogeneity are added. It is difficult to believe that an n-tuple of CVs would effectively describe scheduling performance in these networks. Future work would be needed to determine if it is possible to compare networks in any meaningful way when the number of variables becomes larger.

The results of our experiments with homogeneous networks suggest that a network with many slow processors can outperform a network with a smaller number of fast processors, especially at high loads. This suggests the use of heterogeneous networks composed of a small number of fast processors and a group of slow processors. Hopefully the fast processors could be used at low loads, while at high loads, small, short jobs could be executed on the slow processors. Useful future work could include attempts to quantify this somehow, giving some guidelines as to how many slow processors are needed before a difference in performance is seen.

We compared two node-spanning algorithms with two non-spanning algorithms, and in all cases except the single-node Network 1, the spanning algorithms were outperformed by non-spanning algorithms. More experimentation is required to determine if this is a property of all spanning algorithms in our model, or if it is just that our chosen algorithms are particularly bad. Some of our observations indicated that thread balancing is more important than pre-emption, especially at higher loads. Since our dynamic spanning algorithm, DNS, made no attempt to re-balance the threads, or to pre-empt in a manner that minimized imbalance, we speculate that a dynamic algorithm that did do so might perform better.

The most weakly supported part of our model is the communication overhead. Although the paucity of data in this area is partly to blame, we could increase the validity of our results by attempting our experiments with different communication models. One possible approach is to design parameterized models of different communication patterns, e.g. nearest-neighbour, barrier synchronization. Experiments could then be run using different proportions of these models in the job mix. One further limitation due to the communication model is that the amount of communication is constant, given a job's runtime and number of processors. This almost certainly does not reflect the true situation, so any future model must include some variability in the amount of communication for a job.

We have restricted ourselves to space-partitioning algorithms, as they are the traditional approach in loosely-coupled systems such as a network of workstations. However, in the same manner that we adopted algorithms that are somewhat dynamic, we might also attempt some algorithms that do some time-slicing between jobs at the level of a single node. If a job is running on a single node, then it could be useful to timeslice it with other jobs on the same node.

Our use of a performance penalty for multiplexing a number of threads onto a fewer number of processors created speedup curves for the applications that are monotonically increasing with the number of processors. This is more an artifact of our job model than a deliberate choice for a speedup model - nevertheless, speedup curves for real applications tend to have more convex shapes. Speedup will increase up to a point, and then level off or decline thereafter. Use of these types of speedup curves in our simulations would most likely affect our results, making this another area for future work in heterogeneous scheduling. Our current model allows small changes in processor allocations to be made without severe consequnces in response time, but more complex speedup curves would probably neccesitate scheduling algorithms that attempted to find the optimal execution point for a job in terms of the number of processors allocated.

Bibliography

[ABC+99] A. Agarwal, R. Bianchini, D. Chaiken, F. T. Chong, K. L. Johnson, D. Kranz, J. D. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):430–444, March 1999.

[ACD+96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, February 1996.

[ACP95] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[ADC97] Andrea C. Arpaci-Dusseau and David E. Culler. Extending proportional-share scheduling to a network of workstations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, June 1997.

[ADCM98] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan M. Mainwaring. Scheduling with implicit information in distributed systems. In *Proceedings of 1998 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, pages 233–243, 1998.

[ADV+95] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of ACM SIGMETRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, May 1995.

[AFKT98] Olaf Arndt, Bernd Freisleben, Thilo Kielmann, and Frank Thilo. Scheduling Parallel Applications in Networks of Mixed Uniprocessor/Multiprocessor Workstations. In *Proc. ISCA 11th International Conference on Parallel and Distributed Computing Systems (PDCS-98)*, September 1998. to appear.

[Amd67] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS conference proceedings, Spring Joint Computing Conference*, volume 30, pages 483–485, 1967.

[AS97] Stergios V. Anastasiadis and Kenneth C. Sevcik. Parallel application scheduling on networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):109–124, June 1997.

[BC97] Matt Buchanan and Andrew A. Chien. Coordinated thread scheduling for workstation clusters under Windows NT. In *The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, pages 47–53, Berkeley, CA, USA, August 1997. USENIX.

[Bre97] T. B. Brecht. An Experimental Evaluation of Processor Pool-Based Scheduling for Shared-Memory NUMA Multiprocessors. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS '97 Workshop*, volume 1291 of *Lecture Notes in Computer Science*, pages 139–165, New York, NY, USA, 1997. Springer-Verlag Inc.

[CADAD+97] David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Richard Martin, Chad Yoshikawa, and Frederick Wong. Parallel computing on the Berkeley NOW. In *Proceedings of the 9th Joint Symposium on Parallel Processing*, 1997.

[CDD+91] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on multiprocessors. In *Proc. 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, December 1991.

[CMV94] Su-Hui Chiang, Rajesh K. Mansharamani, and Mary K. Vernon. Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies. In *Proceedings of the 1994 ACM SIGMETRICS Conference*, pages 33–44, February 1994.

[CV96] S.-H. Chiang and M. Vernon. Dynamic vs. static quantum-based parallel processor allocation. *Lecture Notes in Computer Science*, 1162:200–223, 1996.

[DAC96] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, 1996.

[Dei90] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley, Boston, 1990.

[Dow97] Allen B. Downey. Using queue time predictions for processor allocation. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 35–57. Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[DRSS99] L.W. Dowdy, E. Rosti, G. Serazzi, and E. Smirni. Scheduling issues in high-performance computing. *Performance Evaluation Review , (Special Issue on Parallel Scheduling)*, March 1999. to appear.

[DZ97] Xing Du and Xiaodong Zhang. Coordinating parallel processes on networks of workstations. *Journal of Parallel and Distributed Computing*, 46(2):125–135, 1 November 1997.

[Esh96] Mary Mehrnoosh Eshaghian. An introduction to heterogeneous computing. In Mary M. Eshaghian, editor, *Heterogeneous Computing*, chapter 1, pages 1–15. Artech House Publishing, 1996.

[EZL89] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408–23, March 1989.

[Fei95] Dror G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Research report rc 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, February 1995. revised August 1997.

[Fei96] Dror G. Feitelson. Packing Schemes for Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'96 Workshop Proceedings*, volume 1162 of *Lecture Notes in Computer Science*, pages 89–110, Honolulu, Hawaii, April 1996. Springer.

[FJ97] Dror G. Feitelson and Morris A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS '97 Workshop*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261, New York, NY, USA, 1997. Springer-Verlag Inc.

[FN95] Dror G. Feitelson and Bill Nitzberg. Job characteristics of a production parallel scientific workload on the NASA ames iPSC/860. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949 of *Lecture Notes in Computer Science*, pages 337–360. Springer-Verlag, 1995.

[FRS+97] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS '97 Workshop*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34, New York, NY, USA, 1997. Springer-Verlag Inc.

[GST91] Dipak Ghosal, Giuseppe Serazzi, and Satish K. Tripathi. The Processor Working Set and Its Use in Scheduling Multiprocessor Systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

[Gus88] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[GW97] Andrew S. Grimshaw and William A. Wulf. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[Hot96] Steven Hotovy. Workload evolution on the Cornell Theory Center IBM SP2. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 27–40. Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[IPS96] N. Islam, A. Prodromidis, and M. S. Squillante. Dynamic partitioning in different distributed-memory environments. *Lecture Notes in Computer Science*, 1162:244–270, 1996.

[JPF+97] Joefon Jann, Pratap Pattnaik, Hubertus Franke, Fang Wang, Joseph Skovira, and Joseph Riodan. Modeling of workload in MPPs. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 95–116. Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[KL97] JunSeong Kim and David J. Lilja. Utilizing heterogeneous networks in distributed parallel computing systems. In *International Conference on High-Performance Distributed Computing*, pages 336–345, August 1997.

[Kle76] L. Kleinrock. *Queueing Systems, Volume 2: Computer Applications*. John Wiley & Sons, New York, NY, 1976.

[Kle79] L. Kleinrock. Power and deterministic rules of thumb for probabilistic problems in computer communications. *Conference Record of the International Conference on Communications (ICC); IEEE; Boston, MA*, pages 43.1.1–43.1.10, 1979.

[LMW98] V. Lo, J. Mache, and K. Windisch. A comparative study of real workload traces and synthetic workload models for parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1998. Lect. Notes Comput. Sci. vol. .

[LS93] Scott T. Leutenegger and Xian-He Sun. Distributed Computing Feasibility in a Non-Dedicated Homogenous Distributed System. In *Proceedings of Supercomputing '93*, pages 143–152, 1993.

[Lum98] Steven Lumetta. *Design and Evaluation of Multi-Protocol Communication on a Cluster of SMP's*. PhD thesis, University of California at Berkeley, September 1998.

[LV90] Scott T. Leutenegger and Mary K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the 1990 ACM SIGMETRICS Conference*, pages 226–36, May 1990.

[MEB88] S. Majumdar, D. L. Eager, and R. B Bunt. Scheduling in multiprogrammed parallel systems. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 104–113, May 1988.

[MEB91] S. Majumdar, D. Eager, and R. Bunt. Characterisation of Programs for Scheduling in Multiprogrammed Parallel Systems. *Performance Evaluation*, 13(2):109–130, October 1991.

[MVZ93] C. McCann, R. Vaswani, and John Zahorjan. A Dynamic Processor Allocation Policy for Multi-programmed Shared-Memory Multiprocessors. In *ACM Transactions on Computer Systems*, volume 11, pages 146–178, May 1993.

[MZ94] Cathy McCann and John Zahorjan. Processor Allocation Policies for Message-Passing Parallel Computers. In *Proceedings of the 1994 ACM SIGMETRICS Conference*, pages 19–32, February 1994.

[NSS93a] Vijay K. Naik, Sanjeev K. Setia, and Mark S. Squillante. Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments. In *Proceedings of Supercomputing '93*, pages 824–833, November 1993.

[NSS93b] Vjay K. Naik, Sanjeev K. Setia, and Mark S. Squillante. Scheduling of large scientific applications on distributed memory multiprocessor systems. In Linda R. Petzold Richard F. Sincovec, David E. Keyes, Michael R. Leuze and Daniel A. Reed, editors, *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 913–922, Norfolk, VI, March 1993. SIAM Press.

[NSS97] Vijay K. Naik, Sanjeev K. Setia, and Mark S. Squillante. Processor allocation in multiprogrammed distributed-memory parallel computer systems. *Journal of Parallel and Distributed Computing*, 46(1):28–47, October 1997.

[Ous82] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[Par97] Eric Parsons. *Using Knowledge of Job Characteristics in Multiprogrammed Multiprocessor Scheduling*. PhD thesis, University of Toronto, March 1997. Also appears as Technical Report CSRI-361.

[PBS97] E. W. Parsons, M. Brorsson, and K. C. Sevcik. Predicting the performance of distributed virtual shared-memory applications. *IBM Systems Journal*, 36(4):527, 1997.

[PD96] J. D. Padhye and L. W. Dowdy. Dynamic versus adaptive processor allocation policies for message passing parallel computers: An empirical comparison. In *Job Scheduling Strategies for Parallel Processing – IPPS '96 Workshop*, volume 1162 of *Lecture Notes in Computer Science*, New York, NY, USA, 1996. Springer-Verlag Inc.

[PS95] E. W. Parsons and K. C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. *Lecture Notes in Computer Science*, 949:127–145, 1995.

[PS96] Eric W. Parsons and Kenneth C. Sevcik. Benefits of Speedup Knowledge in Memory-Constrained Multiprocessor Scheduling . *Performance Evaluation*, 27 & 28:253–272, October 1996. (ftp) University of Toronto Technical Report CSRI-344.

[PS97] E. W. Parsons and K. C. Sevcik. Implementing multiprocessor scheduling disciplines. In *Job Scheduling Strategies for Parallel Processing – IPPS '97 Workshop*, volume 1291 of *Lecture Notes in Computer Science*, New York, NY, USA, 1997. Springer-Verlag Inc.

[RSD+94] E. Rosti, E. Smirni, L. Dowdy, G. Serazzi, and B. Carlson. Robust Partioning Schemes of Multiprocessor Systems. *Performance Evaluation*, 19((2-3)):141–165, March 1994.

[RSD+98] E. Rosti, E. Smirni, L.W. Dowdy, G. Serazzi, and K.C. Sevcik. Processor saving scheduling policies for multiprocessor systems. *IEEE Transactions on Computers*, 47(2):178–189, February 1998.

[RSSD95] E. Rosti, E. Smirni, G. Serazzi, and L. W. Dowdy. Analysis of non-work-conserving processor partitioning policies. In *Job Scheduling Strategies for Parallel Processing – IPPS '95 Workshop*, volume 949 of *Lecture Notes in Computer Science*, pages 165–181, New York, NY, USA, 1995. Springer-Verlag Inc.

[SBIS98] Rudrajit Samanta, Angelos Bilas, Liviu Iftode, and Jaswinder Pal Singh. Home-based SVM protocols for SMP clusters: Design and performance. In *Fourth International Symposium on High Performance Computer Architecture*, February 1998.

[SBWS99] Alan Su, Francine Berman, Richard Wolski, and Michelle Mills Strout. Using AppLeS to schedule simple SARA on the computational grid. *The International Journal of High Performance Computing Applications*, 13(3):253, Fall 1999.

[SC81] C. H. Sauer and K. M. Chandy. *Computer Systems Performance Modelling*. Prentice Hall, 1981.

[SCZL96] Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. The EASY - LoadLeveler API Project. In *Job Scheduling Strategies for Parallel Processing – IPPS'96 Workshop Proceedings*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47, New York, NY, USA, 1996. Springer-Verlag Inc.

[Sev89] Kenneth C. Sevcik. Characterizations of Parallelism in Applications and their Use in Scheduling. In *Proceedings of the 1989 ACM SIGMETRICS and PERFORMANCE Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1989.

[Sev94] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation North Holland*, 19, 2-3:107–140, 1994.

[SGS96] Jaspal Subhlok, Thomas Gross, and Takashi Suzuoka. Impact of job mix on optimizations for space sharing schedulers. In ACM, editor, *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. ACM Press and IEEE Computer Society Press.

[SLTZ77] K. C. Sevcik, A. I. Levy, S. K. Tripathi, and J. L. Zahorjan. Improving approximations for aggregated queuing network subsystems. *Computer Performance*, pages 1–22, 1977.

[SPWC98] Patrick G. Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A Chien. Dynamic Coscheduling on Workstation Clusters. In *Proceedings of the IPPS '98 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 231–256, New York, NY, USA, March 1998. Springer-Verlag Inc.

[SRS+95] E. Smirni, E. Rosti, G. Serazzi, L. W. Dowdy, and K. C. Sevcik. Performance gains from leaving idle processors in multiprocessor systems. In *Proceedings of the 24th International Conference on Parallel Processing*, pages III:203–210, Oconomowoc, WI, August 1995.

[SSG95] T. Suzuoka, J. Subhlok, and T. Gross. Evaluating job scheduling techniques for highly parallel computers. Computer Science CMU-CS-95-149, Department of Computer Science, Carnegie Mellon University, 1995.

[SST93] Sanjeev K. Setia, Mark S. Squillante, and Satish K. Tripathi. Processor scheduling on multiprogrammed, distributed memory parallel computers. In Blaine D. Gaither, editor, *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, volume 21-1 of *Performance Evaluation Review*, pages 158–170, New York, NY, USA, May 1993. ACM Press.

[ST93] S. Setia and S. Tripathi. A Comparative Analysis of Static Processor Partitioning Policies for Parallel Computers. In *International Workshop on Modeling and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 283–286, January 1993.

[SW95] Patrick G. Sobalvarro and William E. Weihl. Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 63–75, April 1995.

[TG89] Andy Tucker and Anoop Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. *Operating Systems Review*, 23(5):159–66, 1989.

[VSLW91] Zvonko G. Vranesic, Michael Stumm, David M. Lewis, and Ron White. Hector: A hierarchically structured shared-memory multiprocessor. *Computer*, 24(1):72–79, January 1991.

[WAS+96] Daniel W. Watson, John K. Antonio, Howard Jay Siegel, Rohit Gupta, and Mikhail J. Atallah. Static matching of ordered program segments to dedicated machines in a heterogeneous computing environment. In *Proceedings of 5th Heterogeneous Computing Workshop (HCW '96)*, pages 24–37, April 1996. sponsor: IEEE Computer Society.

[WLM+96a] K. Windisch, V. Lo, R. Moore, D. Feitelson, and B. Nitzberg. A comparison of workload traces from two production parallel machines. In *6th Symp. Frontiers Massively Parallel Comput.*, pages 319–326, Oct 1996.

[WLM+96b] K. Windisch, V. Lo, R. Moore, D. Feitelson, and B. Nitzberg. A comparison of workload traces from two production parallel machines. Technical Report CIS-TR-96-06, Department of Computer and Information Science, University of Oregon, April 1996.

[WMKS96] Michael Wan, Reagan Moore, George Kremenek, and Ken Steube. A batch scheduler for the Intel Paragon with a non-contiguous node allocation algorithm. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 48–64. Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[Woo96] Paul R. Woodward. Perspective on supercomputing: Three decades of change. *Computer*, 29(10):99–111, October 1996.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Shingh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual*

International Symposium on Computer Architecture, pages 24–36, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA.

[Wu93] C.S. Wu. Processor Scheduling in multiprogrammed shared memory NUMA multiprocessors. Master's thesis, University of Toronto, 1993. Technical Report CSRI-341, Department of Computer Science.

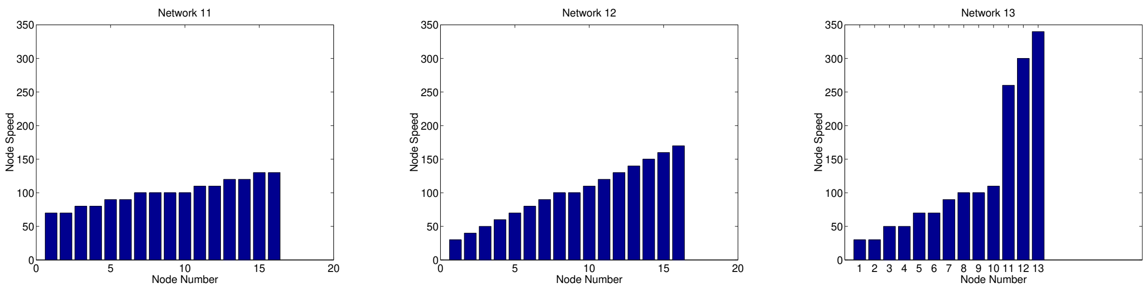
[WZ97] Jon B. Weissman and Xin Zhao. Run-time Support for Scheduling Parallel Applications in Heterogeneous NOWs. In *Sixth International Symposium on High Performance Distributed Computing (HPDC-6)*, August 1997.

[ZB91] Songnian Zhou and Timothy Brecht. Processor pool-based scheduling for large-scale NUMA multiprocessors. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 133–142, May 91.

[ZM90] J Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. *Performance Evaluation Review*, 18(1):214–225, May 1990. Proc. 1990 ACM Sigmetrics Conf. on Meas. and Mod. of Comp. Syst.

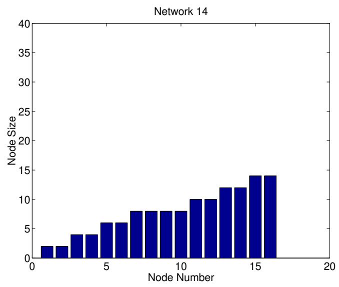
Appendix A

Network Configurations

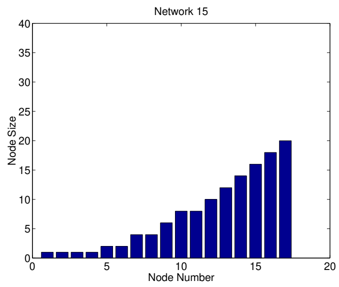


(a) Network 11 (b) Network 12 (c) Network 13

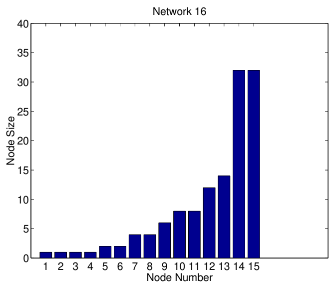
Figure A.1: *Speed-heterogeneous networks*



(a) Network 14

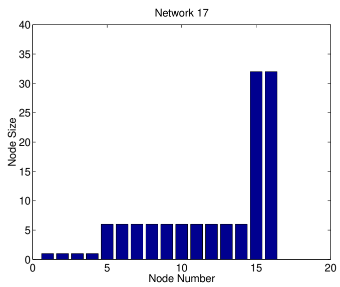


(b) Network 15

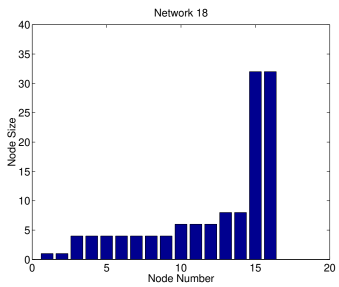


(c) Network 16

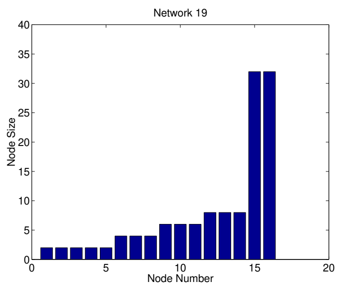
Figure A.2: *Size-heterogeneous networks*



(a) Network 17

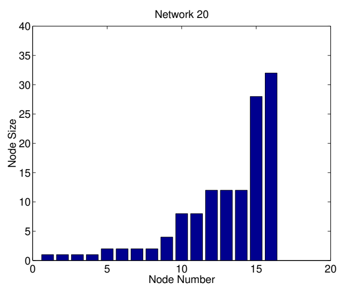


(b) Network 18

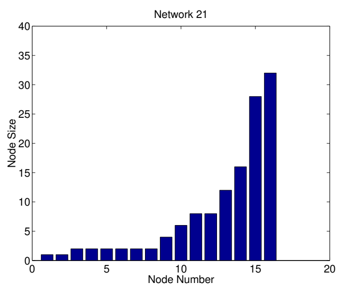


(c) Network 19

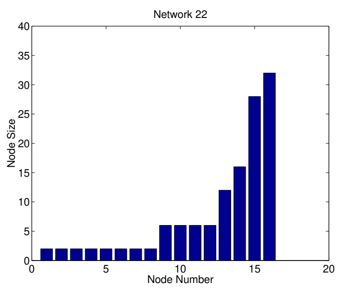
Figure A.3: *Size-heterogeneous networks with skewness* ≈ -2.0



(a) Network 20

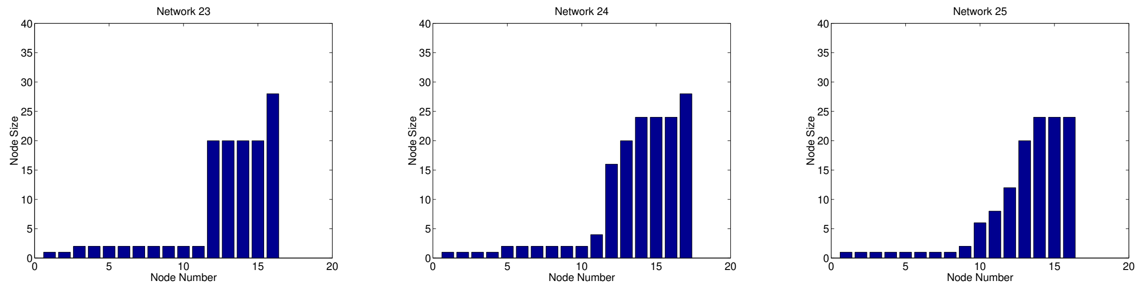


(b) Network 21



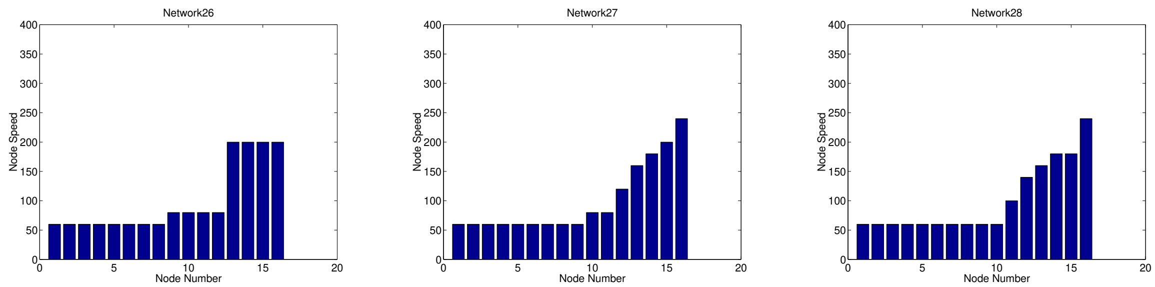
(c) Network 22

Figure A.4: *Size-heterogeneous networks with skewness* ≈ -1.5



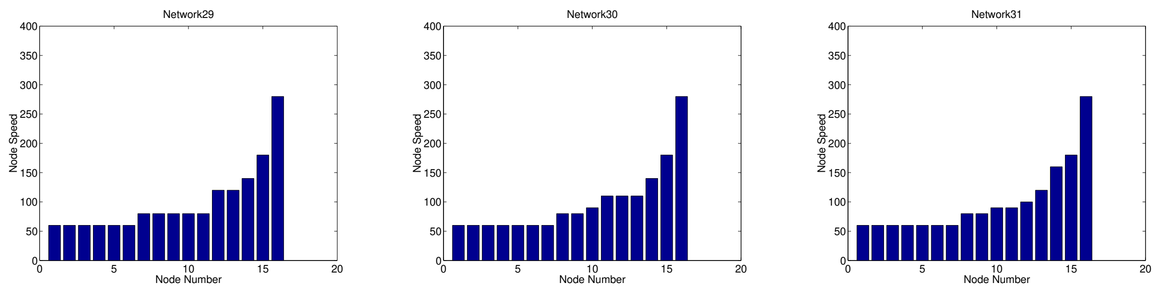
(a) Network 23 (b) Network 24 (c) Network 25

Figure A.5: *Size-heterogeneous networks with skewness* ≈ -0.9



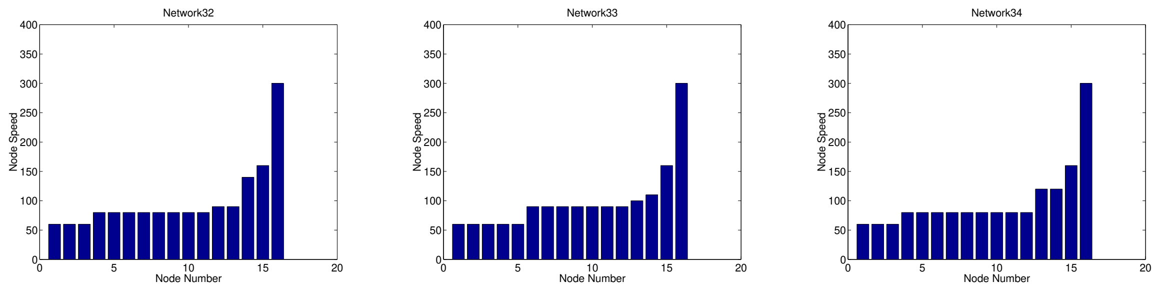
(a) Network 26 (b) Network 27 (c) Network 28

Figure A.6: *Speed-heterogeneous networks with skewness* ≈ -1.0



(a) Network 29 (b) Network 30 (c) Network 31

Figure A.7: *Speed-heterogeneous networks with skewness* ≈ -1.9



(a) Network 32 (b) Network 33 (c) Network 34

Figure A.8: *Speed-heterogeneous networks with skewness* ≈ -2.5

Appendix B

Workload Model Code

The code used to generate the workload:

```
/*
 * generate workload based on statistical model
 *
 * for each job, decide the size (number of nodes used), the runtime,
 * and the repetition count (how many times it will be run).
 * the output is one line per job, with these 3 values.
 *
 * the size is based on a complex discrete distribution.
 * the runtime is hyperexponential, based on the size.
 * the repetition is generalized Zipf (harmonic)
 *
 * the arrival process is not modeled, and should be part of the
 * simulation that uses this workload as input.
 */

#include <math.h>
#include <stdio.h>
#include        <assert.h>
#include        "utils.h"
#include        "randlib/randlib.h"
#include        "feitelson/wlfeitelson96.h"

#ifdef __cplusplus
extern "C" {
#endif


#define MAXINT 0x7fffffff

#define LEN 300000 /* jobs generated */

#define TIME_LIM 64800 /* longest job allowed - 18 hr */
#define FACTOR 9 /* about MAX_TIME / log(TIME_LIM) */
```

```
#define MAX_SIZE 128 /* machine size */
#define MAX_REP 1000 /* far upper bound on number of repetitions */
#define MAX_TIME 100 /* for distribution of runtimes */

int     gkMaxSize;              /* user-set max size ( <= MAX_SIZE) */
float size_dist[ MAX_SIZE ]; /* cummulative distribution of sizes */
int  time_dist[ MAX_TIME ]; /* cummulative distribution of runtimes */
float rep_dist[ MAX_REP ]; /* cummulative distribution of repetitions */

/* flag to indicate that intialization has been done */
static  Bool_T  bInitialized  = FALSE;


#ifdef MAKE_PLOTS
char filename[ 20 ];

int size_hist[ MAX_SIZE ];
float   time_hist[ MAX_SIZE ];
int rep_hist[ MAX_REP ];

GenJob_T jobs[ LEN ];

int head[ MAX_SIZE ];
#endif


/* InitJobGeneration
 *   sets up the cumulative distributions needed to generate the jobs.
 */
void InitJobGeneration(int nMaxJobSize)
{
  int      i, j;
  float    sum;

  /* no point in doing this twice */
  if (bInitialized)
    return;

  assert(nMaxJobSize > 0 && nMaxJobSize <= MAX_SIZE);
  gkMaxSize = nMaxJobSize;

  /* first create a semi-harmonic distribution */
  size_dist[0] = 1;
  for (i=1 ; i<gkMaxSize ; i++) {
    size_dist[i] = (float)1/(float)sqrt((double)i);
  }
```

```
/* add weight at powers of 2 */
for (i=1 ; i<=gkMaxSize ; i<<=1) {
  size_dist[i-1] += 35 + 1.5*i;
}

/* add weight at squares */
for (i=2, j=i*i ; j<=gkMaxSize ; i++, j=i*i) {
  size_dist[j-1] += 5;
}

/* add weight at multiples of 10 */
for (i=10 ; i<=gkMaxSize ; i+=10) {
  size_dist[j-1] += 5;
}

/* reduce weight at 1,2,4 */
size_dist[0] /= 4;
size_dist[1] /= 4;
size_dist[3] /= 3;
/* add weight at 3,5-7 */
size_dist[2] += 5;
size_dist[4] += 7;
size_dist[5] += 5;
size_dist[6] += 3;

/* impose a harmonic envelope to emphasize small sizes  */
/* for non-special sizes, this becomes harmonic of order 1.5 */
for (i=1 ; i<gkMaxSize ; i++) {
  /* size_dist[i] /= (float)sqrt((double)i); semi-harmonic - no good */
  size_dist[i] /= (float)i;
  /* size_dist[i] /= (float)exp(i); exponential decreasing - no good*/
}

/* normalize */
sum = 0;
for (i=0 ; i<gkMaxSize ; i++) {
  sum += size_dist[i];
}
for (i=0 ; i<gkMaxSize ; i++) {
  size_dist[i] /= sum;
}

/* accumulate */
for (i=1 ; i<gkMaxSize ; i++) {
  size_dist[i] += size_dist[i-1];
}
```

```
  /* create harmonic repetition distribution of order 2.5 */
  sum = 0;
  for (i=0 ; i<MAX_REP ; i++) {
    rep_dist[i] = ((float)1)/(float)pow((double)(i+1),(double)2.5);
    sum += rep_dist[i];
  }

  /* normalize */
  for (i=0 ; i<MAX_REP ; i++) {
    rep_dist[i] /= sum;
  }

  /* accumulate */
  for (i=1 ; i<MAX_REP ; i++) {
    rep_dist[i] += rep_dist[i-1];
  }

  bInitialized = TRUE;
}



/* GenerateJob
 *   - get the next job in this sequence.
 *   - the random number generator used is in "randlib".
 *   - the specific random number stream used depends on the seed being
 *     set by setall()
 *   - this code is lifted from the original Feitelson code in GenRunFile
 *      and code was added to generate a hyperexponential inter-arrival time
 */
GenJob_T GenerateNextJob()
{
  GenJob_T   genjob;
  int        i, size, rep;
  float      p, runtime, m;

  /* check that initialization has been done */
  if (!bInitialized)
  {
    assert(FALSE);
    return genjob;
  }

  /* choose size from distribution */
  p = ((float)ignlgi())/MAXINT;
  i = 0;
```

```
  while (p > size_dist[i])
    i++;
  size = i;

  /* assign runtime */

  /*
   * in the improved model, runtimes are from a three-stage
   * hyperexponential distribution, and the probability of
   * choosing each branch depends on the size. in addition,
   * there is an upper bound on the runtime.
   */
  while (1) {
    p = ((float)ignlgi())/MAXINT;
    if (p < (0.90 - 0.65*( sqrt(((double)size)/gkMaxSize) )))
      m = 50;
    else if (p < (0.97 - 0.37*( sqrt(((double)size)/gkMaxSize) )))
      m = 900;
    else
      m = 20000;
    if ((size == 2) || (size == 4) || (size == 8) || (size == 16) ||
(size == 32) || (size == 64) || (size == 128))
      m *= 2;
    runtime = -m * (float)log( ((double)ignlgi()) / MAXINT );
    if (runtime < TIME_LIM)
      break;
  }

  /* assign repetition */
  p = ((float)ignlgi())/MAXINT;
  i = 0;
  while (p > rep_dist[i])
    i++;
  rep = i;

  /* assign the job fields */
  genjob.rtime = runtime;
  genjob.rep   = rep + 1;
  genjob.size  = size + 1;

  return genjob;
}



#ifdef __cplusplus
```

```
}
#endif
```

Appendix C

Arrival Process Code

The code used to generate the arrival process:

```
// Workload::SetArrivalParams
//  set the mean and the coefficient of variation for the
//  hyperexponential arrival process and calculate some constants.
void Workload::SetArrivalParams(double dMean, double dCV)
{
  dArrivalMean = dMean;

  // CV^2 is actually required by the formulae
  dArrivalCVsq   = dCV * dCV;

  // these parameters are from the paper "Improving Approximations
  // of Subsystems" by Sevcik et al., 1977, Computer Performance
  // North-Holland, ed. K. M. Chandy and M. Reiser.
  a = 0.5 * (1 - sqrt( (dArrivalCVsq - 1.0) / (dArrivalCVsq + 1.0)));
  assert(a > 0.0 && a <= 1.0);
  r = dMean / (2.0 * a);
  s = dMean / (2.0 * (1.0-a));

  // set flag that this has been done - just for a sanity check
  bArrivalInitialized = TRUE;
}



// Workload::GenerateArrivalTime
//   Inter-arrival times are hyper-exponentially distributed
//   with a varying mean and Cv set at program startup.
//   See the comments in Workload::SetArrivalParams for the
//   paper reference.
Time_T Workload::GenerateArrivalTime()
{
```

```
  assert(bArrivalInitialized);

  // generate uniform variate.
  double dU = GenerateUniformTime(0.0, 1.0);

  // flip a random coin
  double dCoin = GenerateUniformTime(0.0, 1.0);
  Time_T time;

  // 2-stage hyperexponential - choose one branch with
  // probability a and the other with prob. (1-a)
  if (dCoin <= a)
  {
    time = (-r) * log(dU);
  }
  else
  {
    time = (-s) * log(dU);
  }

  return time;
}
```