(Journal of) Performance Evaluation, vol. 19 (1994), pp. 107-140.

Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems

Kenneth C. Sevcik

Technical Report CSRI-282
March 1993

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems

K.C. Sevcik
Computer Systems Research Institute
University of Toronto

March, 1993

ABSTRACT

When large-scale multiprocessors for parallel processing are subjected to heavy diverse workloads of applications, it will be necessary to schedule them in a multiprogrammed fashion in order to use the system resources effectively and keep response times low. Information about the characteristics of individual applications can be used to achieve effective scheduling. We identify some useful parameters for characterizing applications. Prior work on multiprocessor scheduling and examination of several simplified special cases reveal some guiding principles for scheduling. The synthesis of full scheduling algorithms based on the principles is explored. We address directly the case of scheduling a batch of applications with deterministically known characteristics, but we also consider extensions to handle uncertainty in service times, priorities, and an arrival stream of applications.

1. Introduction and Context

1.1 Platforms for Large-Scale Parallel Processing

In the past few years, the availability and usage of multiprocessors have grown rapidly. Some types of multiprocessors provide extremely high theoretical peak execution rates with great cost effectiveness. It is widely accepted that the substantial gains in computational power required to solve problems in climate modelling, genetic code analysis, cosmological simulation, etc. can only come from exploiting parallelism.

Several forms of multiprocessors are commercially available. The most mature form of multiprocessors is bus-based shared-memory multiprocessors (such as those produced by Encore and Sequent). Cache-consistency can be provided in hardware since all memory requests use the bus. However, the architecture is not scalable since the bus saturates when the number of processors becomes large. Bus-based shared-memory multiprocessors have been very successful in providing high throughput by executing independent jobs or transactions in parallel.

A distinctly different form of multiprocessor is that in which all the processors execute a single instruction stream (as in the Maspar and Thinking Machine's CM-1 and CM-2 systems). This type of multiprocessor can exploit data parallelism, but not control parallelism.

Another significant class of multiprocessor (or multicomputer) architectures is distributed memory systems, in which processor/memory pairs are linked by an interconnection network. These systems can be scalable since all requests need not use a single bus. In some such systems, applications programmers explicitly specify the exchange of information and synchronization through passing messages (Intel and Ncube systems, and Thinking Machines CM-5). Others provide the programmer with an image of a global shared memory, even though non-uniform memory access (NUMA) times result from the fact that some memory requests travel further than others through the interconnection network (TeraComputer, Kendall Square KSR1).

Clusters of workstations can also be used for parallel processing. Many organizations have large numbers of workstations that have low overall utilization. Using load-sharing software, the latent power of the workstation clusters can be used for large parallel computations.

Successful parallel processing systems will attract heavy workloads. They will operate in a multiprogrammed fashion, with several parallel applications being processed concurrently. If a number of applications are available for execution, overall processor utilization will be kept higher (and response times lower) by allocating fewer processors to each of several applications rather than all the processors to a single application.

1.2 Processor Allocation Problem for Parallel Processing

Scheduling algorithms for parallel processing systems determine (1) how many and which applications to activate, and (2) how many processors to allocate to each one. The decisions may be based on information provided or acquired about the applications available for execution. Usage of memory, communication links, and other resources may also influence the activation and processor allocation decisions.

Parallel applications can be written so that they execute on a number of processors that is determined only when the application is started, and the number may even be changed during execution. Under *two-level scheduling*, the scheduling algorithm allocates processors among active applications, while threads of the application are assigned to processors and dispatched by application level software [TG89, MVZ91]. In addition to permitting flexible multiprogramming, other advantages come from making applications independent of the precise number of processors available for execution. The application can execute even when one or more processors fail, and it can execute without modification on various systems in which the numbers of processors differ.

1.3 Scope of the Paper

The main goal of this paper is to identify effective algorithms for application scheduling and processor allocation in parallel processing systems. The primary performance objective will be to keep average response times low, possibly taking priorities into account. Knowledge of the workload and some characteristics of individual applications can be used for improved scheduling decisions. Specific useful application characteristics will be identified, and methods for determining their values will be considered.

The scheduling algorithms developed in this paper are intended for systems that support production workloads dominated by large-scale parallel applications. The algorithms are directly applicable in uniform memory access time (UMA) systems. With some variations, they are also applicable in NUMA shared memory multiprocessors, message-passing multicomputers, and workstation clusters. Because the types of applications under consideration are compute intensive, the algorithms will dedicate processors to individual applications.

The remainder of the paper is organized as follows. Section 2 reviews material relating to the specification and classification of scheduling problems. Section 3 presents scheduling algorithms that do not require any prior knowledge of characteristics of individual applications. In section 4, the types of application characteristics that might be used in scheduling are surveyed, and methods for estimating their values through measurement and experimentation are discussed. Section 5 contains a survey of prior work on scheduling algorithms that use information about individual application characteristics. Through the analysis of several special cases, some insights about scheduling principles are derived. In section 6, a specific algorithmic basis for scheduling algorithms is described, and specialized to use various amounts of information about application characteristics. Finally, section 7 contains concluding remarks that indicate how the basic scheduling algorithm can be made more widely applicable.

2. Multiprocessor Scheduling Issues

2.1 Classification of Scheduling Problems

Scheduling models may be distinguished in several ways [CD73] as is suggested in Figure 2.1. This classification allows us to identify the scheduling models to be addressed in this paper.

Objective:				
minimizing average response time	*vs.*	minimizing makespan		
Flexibility:				
non-preemptive scheduling	*vs.*	preemptive scheduling		
Arrival Process:				
batch of applications	*vs.*	Poisson arrival stream		
Service Demand Knowledge:				
none	*vs.*	stochastic	*vs.*	deterministic
Structure:				
fork/join	*vs.*	multiphase	*vs.*	task graph
Priority:				
equal priorities	*vs.*	distinct priorities		

Figure 2.1. Classification of some scheduling models.

Objective – We will focus on minimizing average response time (or turnaround time or time in system). This is the most important measure when the workload is generated by a number of independent customers submitting work at various times. An alternative objective, treated extensively in the scheduling literature, is the minimization of total schedule length required to complete a set of jobs (i.e., minimizing *makespan*).

Flexibility – Scheduling decisions, once made, may be either irrevocable, or changeable dynamically. In the latter case, there may be a cost or penalty for changing the decision. For example, in sequencing jobs on a single processor, *preemption* may or may not be allowed. If preemption is allowed, the preemption overhead either may be considered negligible or may be taken into account as a loss of some processor time. In cases where more than one unit of a resource is being scheduled (e.g., units of memory or processors of a multiprocessor), the amount of a resource allocated to a job may be increased or decreased during its execution. We shall consider a full range of possibilities for processor allocation from purely *static* allocation to fully *dynamic* allocation as will be described in section 2.2.

Structure – The structure of a computation determines its ability to make use of parallel processors. One model that has been investigated (known and the *fork/join* model [TRS90]) presumes that the computation can be divided among the available processors such that the service times of the portions assigned to each processor can be represented as samples drawn from a single distribution (most often the negative exponential). Another model (known and the *multiphase* model) assumes that the computation progresses through a series of phases, and the amount of parallelism available differs in successive phases (and can be described by a *parallelism profile* [Se89]). A third commonly used model is a *task graph*, which represents the components of the overall computation and ordering constraints on their execution.

Service Demand Knowledge – Some information about the service demand of each application may be available. The knowledge may be precise (e.g., exact service time) or probabilistic (e.g., expected service time or service time distribution). If all customers have statistically identical service times, the resource needs of applications cannot be distinguished before execution is started.

Priority – Customers may be associated with different priority classes according to their inherent importance and/or to application characteristics.

Arrival Process – The workload to be scheduled may consist of either a batch of applications, all available for execution at the beginning of the scheduling interval, or an ongoing stream of arrivals. We shall focus primarily on scheduling a batch of applications, and comment briefly on allowing for a (Poisson) stream of arriving applications.

2.2 Static and Dynamic Scheduling Algorithms

Processor scheduling algorithms differ in the frequency with which scheduling decisions are made and revisited [MEB91]. A *static* scheduling algorithm assigns a certain number of processors to an application at the time it is activated, and those processors are held until the application is completed. At the other extreme, a *dynamic* scheduling algorithm permits the allocation of processors among applications to be changed at any time. In between static and dynamic, there are variations that allow adjustments to the allocation of processors only when specific events occur. Some such events are:

i) completion of an application, in order to use the processors released either to start a new application or to augment the number of processors assigned to other active applications;

ii) arrival of an application, in order to allow the immediate start of the arriving application, possibly preempting processors from other active applications;

iii) a phase change indicated by some active application, either increasing or decreasing the number of processors currently needed by the application.

Static scheduling is simple and has low overhead. However, it may leave processors idle when work is available, or assign them to applications that cannot use them effectively at some points in their execution. Overall, static scheduling is appropriate when applications generally have low variability in their parallelism profiles and enough information is available to determine how many processors an application can use effectively.

Dynamic scheduling (in each of its degrees) permits adaptation to changing circumstances. When an application completes, the processors can be reassigned in the most effective way, and new arrivals can be started immediately, which is desirable particularly when the variability in service required by applications is high. If applications have distinct phases with different processor requirements, dynamic scheduling can increase and decrease allocations accordingly, in order to use the processors well. The major disadvantage of dynamic scheduling is the overhead that is incurred in making more frequent scheduling decisions and in carrying out the preemption and resumption of threads of computations. When processors are deallocated from an application, care must be taken that no critical resources (such as locks) are held by the process that is suspended [ZM90]. Also, more information about application structure is needed for effective dynamic scheduling than for static scheduling.

2.3 Speedup and Slowdown Properties of Applications

Several studies have identified a variety of types of information that can be known or estimated or observed about the execution of individual applications to allow improved scheduling. In addition to the types of information that are required in a single processor sequencing context (such as total service time required and priority), additional information is relevant in the multiprocessor context. For dynamic scheduling, it is useful to know the full *parallelism profile* of the application [K87, Se89], which indicates how many processors could be used by an application at all points in its execution. The parallelism profile can be measured in off-line experiments, as was done by Kumar [K87] and others, or may be estimated by the user from knowledge of the application structure. Values of other higher level descriptive parameters either can be estimated directly, or can be calculated from knowledge of the parallelism profile. For static scheduling, it is sufficient to know the parallelism distribution, $\pi_j(p)$, p=1, 2, 3, $\cdots$, where $\pi_j(p)$ is the fraction of its execution time that application j would be using p

processors if an unlimited number were available to it. Much of this information can be partially conveyed more concisely in parameters such as the fraction of sequential work [A67], f, the minimum, maximum and average of parallelism attained [Se89], m, M, and A, respectively, and values indicating the variability of the parallelism distribution [Se89, MEB91], V and $\omega(A)$.

If the execution time of an application can be measured or estimated for various numbers of assigned processors, then other characterizations are possible. Let $T_j(p)$, $p=1, \cdots, P$, be the *execution time function* where $T_j(p)$ is the execution time of application j assuming a static allocation of p processors. In reality, $T_j(p)$ may depend on how the other $P - p$ processors of the system are being used by other applications (through contention in the processor interconnection network or for shared I/O devices, for example). From the execution time function, two other representations of the effectiveness of parallel execution of the application can be derived [EZL89]:

$$\textit{Speedup} \quad - \quad S_j(p) = \frac{T_j(1)}{T_j(p)}$$

$$\textit{Efficiency} \quad - \quad E_j(p) = \frac{S_j(p)}{p}$$

All these parameters are summarized in figure 2.2.

$\pi(p)$	–	fraction of time that p processors are used with an unlimited number available
A	–	average parallelism ($= \sum_{i=m}^{M} i\pi(i)$)
V	–	variance in parallelism ($= \sum_{i=m}^{M} (i-A)^2\, \pi(i) = \sum_{i=m}^{M} i^2\pi(i) - A^2$)
f	–	fraction of sequential code ($= \pi(1)/A$)
m	–	minimum parallelism ($= \min_{i\geq 1}[i \mid \pi(i) > 0]$)
M	–	maximum parallelism ($= \max_{i\geq 1}[i \mid \pi(i) > 0]$)
r	–	parallelism range ($= M - m$)
$\omega(A)$	–	parallelism variability ($= A \,/\, S(A)$)
$T(p)$	–	execution time on p processors
$S(p)$	–	speedup with p processors ($= T(1) \,/\, T(p)$)
$T(p)$	–	efficiency with p processors ($= S(1) \,/\, p$)

Figure 2.2. Summary of some application characterization parameters.

If each application requests a specific number of processors, then the simplest static scheduling algorithm assigns to each application the number of processors it requests, and activates an application only when enough processors are available. The efficiency of the overall system then depends on applications making good use of the processors they request.

Three elementary static processor allocation algorithms based on parameters from figure 2.2 are:

min (M, P) – assign as many processors as the application can ever use, (up to a maximum of the number of processors in the system);

min (A, P) – assign an application the average number of processors it can use (up to a maximum of the number of processors in the system);

min (m, P) – assign an application only as many processors as it can always keep busy (which is typically just one).

The first of these, *min (M, P)*, allows each individual application to complete as quickly as possible once started, but may leave many processors idle. At the other extreme, *min (m, P)* avoids idleness of allocated processors, but doesn't allow applications to reduce their execution time through fully exploiting parallelism. The intermediate strategy above, *min (A, P)*, achieves some speedup while still using the allocated processors reasonably efficiently.

Eager, Zahorjan, and Lazowska [EZL89] proposed a static allocation algorithm based on minimizing the ratio $T_j(p)/E_j(p)$. If the value of p that minimizes this ratio is not known, the authors recommend the allocation of A_j processors, since it can be proven that A_j differs from the value that minimizes the ratio by no more than a factor of two. Allocating A_j processors to application j guarantees that at least half the possible speedup is attained ($S(A_j) \geq S(P)/2$), while efficiency is kept above 50%. A related policy based on the concept of a *processor working set* maximizes the *power* performance measure, again by allocating the number of processors that corresponds to the knee of the execution time versus efficiency curve [GST91].

If the range of parallelism, $r_j = M_j - m_j$, is known along with A_j for each application, and also the overall load on the system, then an better processor allocation algorithm can be applied [Se89]. When system load is light, the better algorithm would allocate closer to M_j processors to an application, rather than just A_j. Conversely, under heavy load, fewer processors, say m_j, would be allocated rather than A_j.

Using estimates of a few basic parameters of each application, it is possible to establish bounds (subject to certain assumptions) on the speedup function for the application. Amdahl's Law [A67] as applied to multiprocessors provides an upper bound on speedup in terms of the fraction of the computation that must be executed with no parallelism, f:

$$S(p) \leq \frac{1}{f + \frac{(1-f)}{p}} = \frac{p}{1 + (p-1)f}$$

Both upper and lower bounds on speedup can also be expressed in terms of the average parallelism in the application, A [EZL89]:

$$\frac{p\,A}{p + A - 1} \leq S(p) \leq \min(p, A)$$

This upper bound is composed of two parts. When $p < A$, speedup is constrained by the number of processors in the system (the *hardware bound*). When $A \leq p$, speedup is constrained by the amount of parallelism inherent in the application (the *software bound*). Combining the two sets of bounds while making their forms more consistent yields:

$$\frac{p}{1 + (p-1)/A} \leq S(p) \leq \min\left\{ \frac{p}{1 + (p-1)f},\ A \right\}$$

Figure 2.3 illustrates these bounds for a specific case. Note that the lower bound is based on the assumption that the total amount of work in an application does not increase with the number of processors on which it is executed. In actuality, the overhead (additional work incurred) of parallel processing will cause the attained speedup to drop below the "lower bound" for a sufficiently large number of processors.

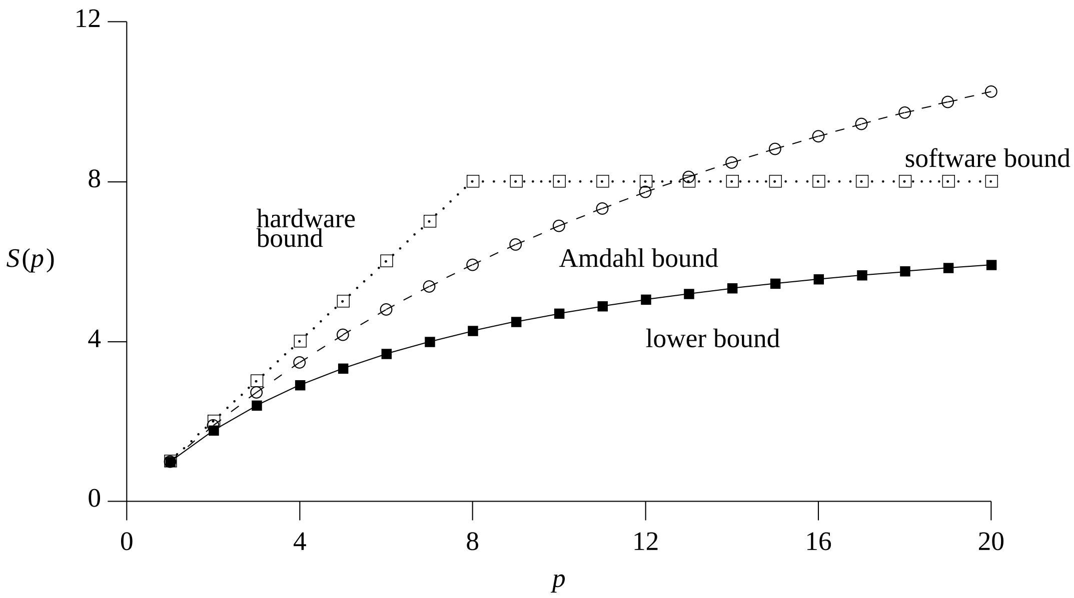

Figure 2.3. Some bounds on speedup.

If other basic characteristics in addition to A and f can be estimated, such as m and M, then tighter bounds on the speedup function can be obtained. The lower the range in parallelism $(M - m)$ and the variance in parallelism, the better the speedup will be. Majumdar, Eager and Bunt have identified four different types of application parallelism patterns that represent extremes of speedup functions [MEB91]. They are distinguished by a parameter $\omega(A) = \frac{T(A)}{T(\infty)} = \frac{A}{S(A)}$, which lies in the range [1, 2] with smaller values corresponding to lower variability in parallelism and better speedup.

The best possible structure of an application is to have it require exactly A processors throughout its execution, leading to $\omega(A) = 1$, and $S(p) = \min(p, A)$. The worst possible structure is where the application requires only one processor for a portion of its execution, but many processors for the remainder, leading to $\omega(A) = 2 - 1/A$ and $S(p) = \frac{pA}{p + A - 1}$. The other two cases relate to the minimization of the ratio of $T_j(p)/E_j(p)$, and hence maximizing the *power* of the system [GST91].

2.4 Slowdown Factors

When executing applications on parallel processors, a major concern is the "speedup factor" achievable through parallel processing. A simple, but usually optimistic estimate of the speedup attained when using P processors instead of one is P. However, there are several "slowdown factors" that significantly counteract the anticipated speedup [MS88]. Some of these are:

(1) Starvation

Due to the internal structure of an application or due to the difficulty in dividing work evenly across the available processors, there may be points in an application's execution where all available processors may not be kept busy.

(2) Inflation
The total work required to execute an application on multiple processors generally exceeds the work required to execute it on one processor, because some work is required to initiate the computation on each processor, and other work may have to be done redundantly on multiple processors.

(3) Communication
The pieces of an application executing on different processors must communicate in order to synchronize and exchange information, such as the status of locks on shared resources.

(4) Congestion
Queueing delays may result when the threads of an application executing on different processors contend for shared resources such as shared data structures, interconnection network components, and possibly even I/O devices.

Figure 2.4 shows the effect of these slowdown factors on the speedup and efficiency curves. In each case, the upper and lower bounds based on assuming no overhead are shown. The curves marked "no overhead" illustrate the actual speedup and efficiency curves of a particular computational structure, ignoring the overhead of parallel execution. The curves marked "with overhead" show that the slowdown factors listed above cause the speedup and efficiency of the particular computational structure to drop below the no overhead lower bounds. Note that the effect of the overhead increases with the number of processors.

2.5 A Case for Assigning One Processor Per Application

Despite a great deal of effort having been devoted parallelizing applications, a number of studies have indicated that, under a surprisingly broad range of circumstances, optimal processor allocation in a multiprogrammed multiprocessor system calls for allocating just one processor per application [Sr91, MN91, MN92, SST92]. We assume in the following that the number of applications available for execution is at least P, and that, for each application, $E_j(p) < 1$ whenever $p > 1$.

One argument supporting this point of view consists of the following sequence of claims:

(1) The demand for efficient large-scale computation is insatiable. To attain sufficiently accurate solutions to many of the major problems in computational science (such as the "grand challenge" problems) unlimited amounts of computation are required.

(2) Due to cost factors and economies of scale, the best systems for large-scale computing will continue to be shared facilities available to tens or hundreds of scientists.

(3) Good effective shared multiprocessor systems for scientific computation will be heavily loaded since computational scientists can identify problems to solve faster than the fastest systems can solve them.

(4) Incurring overhead due to parallel programming cuts down the throughput capacity of the system, bringing it down close to the offered load, and hence leading to large queueing times.

Together, these four claims lead to the conclusion that just a single processor should be allocated per application in a shared system under reasonably heavy load in order to keep average response time low (assuming that at least P applications are available for execution and that the memory size is adequate). Those who have experienced the difficulty of achieving speedup in practice may find the desirability of allocating one processor per application quite intuitive.

Even at moderate load, applications should be allocated many processors only if (1) the computation can be partitioned almost evenly across the processors, and (2) the total amount of computation is not significantly increased by doing so.

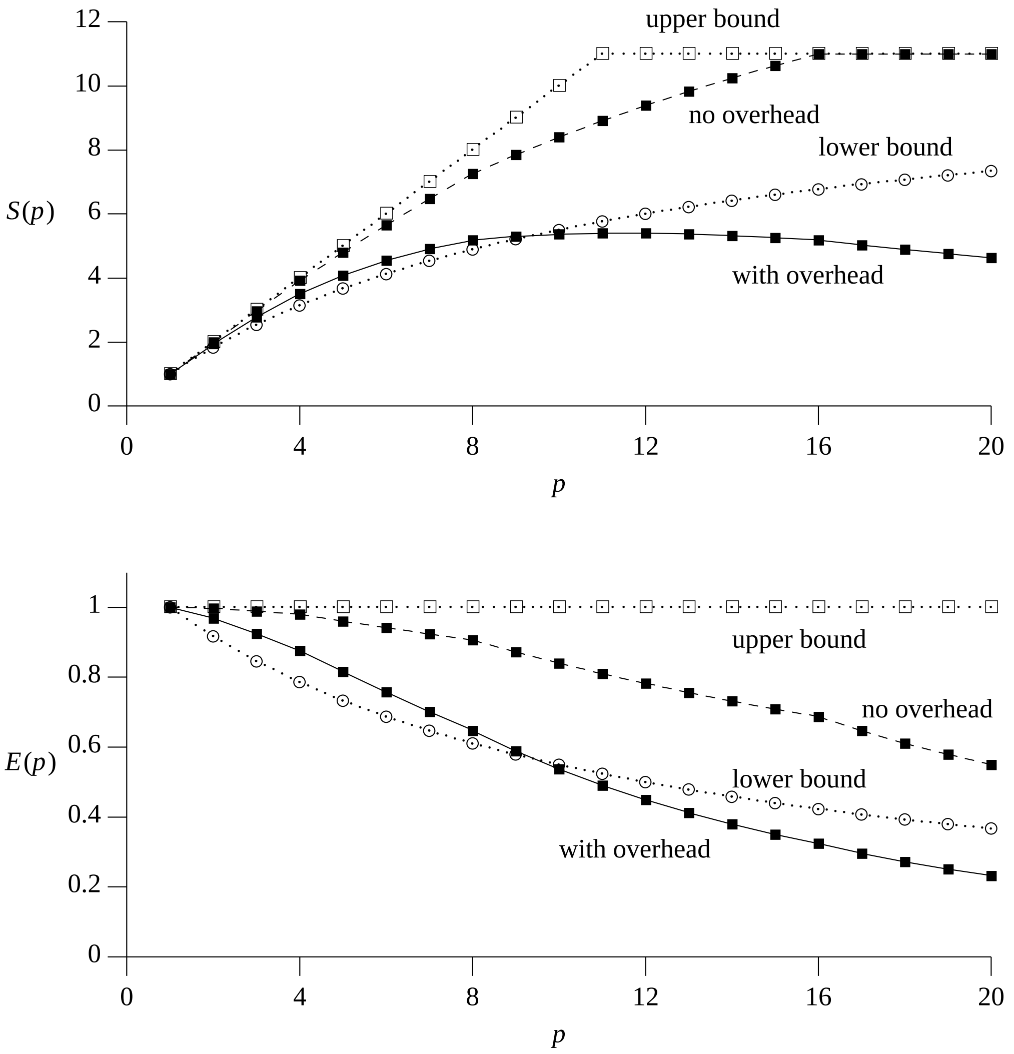

Figure 2.4. The effect of slowdown factors on the speedup and efficiency curves.

Assuming that not all parallel processing systems will be under heavy load all the time, in the remainder of the paper, we will continue to consider scheduling algorithms that allocate more than one processor per application.

3. Scheduling Without Prior Knowledge of Application Characteristics

3.1 Single Processor Scheduling Rules

If jobs are statistically indistinguishable initially, then the scheduling strategy that minimizes average time in system depends on the distribution of processing times. With distributions where the expected remaining processing time decreases monotonically as service is acquired (*increasing failure rate* distributions [BP65], such as the uniform distribution), first-come first-served (FCFS) minimizes the average time in system. For distributions where the expected remaining processing time actually grows with the amount of attained service (*decreasing failure rate* distributions, such as the hyperexponential), the strategy that minimizes average time in system when preemption is free is preemptive processor sharing (PPS), which calls for sharing the processor's attention evenly among the set of jobs that have the least acquired service (since those jobs are tied for having the least expected remaining processing times). More generally, for distributions that have a *coefficient of variation* (the ratio of standard deviation to mean) less than one, FCFS is appropriate, while for distributions with coefficients of variation greater than one, it is not. If even the distribution of service times is unknown, then round-robin (RR) scheduling, or processor sharing (PS) if preemption is free, is a compromise strategy that generally leads to relatively low average time in system. (Note that PPS behaves like PS if all jobs are initially available.)

Some workloads include a significant number of jobs that require very little service to complete, but it is not known in advance which jobs those are. If preemption is inexpensive, then it is desirable to use a scheduling algorithm that has the property that each job gets some small amount of service soon after it becomes available for execution in order to identify and quickly complete short jobs. The algorithms PS and PPS (and RR, if the quantum is small) have this *immediate start* property, whereas FCFS does not.

3.2 Multiprocessor Scheduling Rules

In early work on the scheduling of multiprogrammed parallel systems, Ousterhout proposed *Co-scheduling*, where groups of cooperating processes (or application threads) are all assigned processors at the same time so that they may execute and exchange messages without being delayed [O82]. In order to share the multiprocessor among several groups of processes, a group of cooperating processes would execute on the processors for a time-slice, then a preemption would take place on each processor simultaneously, and a different set of cooperating processes would be scheduled.

Gupta, Tucker, and Urushibara [GTU91] have proposed a scheduling algorithm called process control with processor partitioning (or simply *Process Control*) based on the principle of having each application control the number of logical processors it uses, and then partitioning the available physical processors among the applications available for execution based on priorities and fairness. Studies by Gupta, et al. showed Process Control to be better than Co-scheduling with respect to maximizing processor utilization [GTS91, GTU91].

Zahorjan and McCann proposed a dynamic scheduling policy that attempts to give at least one processor to each application, and allows applications to request and release processors reflecting their changing need for parallelism at various points in their execution [ZM90]. They show using simulation that their algorithm is better than either Co-scheduling or running each application to completion as long as the processor preemption overhead is not too large. In a later study, McCann, Vaswani, and Zahorjan used experimentation on a UMA shared memory system (Sequent) to demonstrate that their dynamic scheduling algorithm, based on space-sharing and coordinated processor preemption, also outperforms Round Robin Job (RRjob) and Equipartition [MVZ91]. RRjob shares the total processing power equally among applications using a quantum-driven scheme [LV90], and Equipartition space shares the processors equally among the applications. Equipartition resembles Process Control, but handles preemption of processors less aggressively and pays less attention to maintaining cache affinity.

Crovella, et al. used an experimental approach to compare scheduling algorithms on a NUMA multiprocessor (BBN Butterfly) [CDDLM91]. They compared the use of time-slicing on a single global process queue with both Co-scheduling and Process Control. They concluded that Process Control was the most effective, in part due to the high cost of cache context loss with Co-scheduling.

Some experimental studies have been carried out in the context of distributed memory rather than shared memory systems. Qualitatively, the scheduling problem is the same as in shared memory systems, although the costs of communication and processor preemption are significantly different. Dowdy and his co-workers have studied several problems involving the partitioning of a set of processors among two or more applications [LDP89, PD89, DCDP90, CD92, RSDSC92]. These studies involve processor scheduling policies ranging from fixed partitions, to static allocation at application initiation, to fully dynamic allocations. Some simple analytic models are developed, and their predictions are compared to experimental results obtained on Transputer and Intel hypercube systems. It was found that, despite the repartitioning overhead, dynamic policies proved best in a variety of circumstances.

Nelson, Towsley and Tantawi have studied a family of multiprocessor scheduling models with which they evaluated the tradeoff between the benefits of dividing a computation into parts to be executed in parallel and the costs of synchronizing the parts of the computation [NTT88]. Nicol examined the problem of splitting a computation specified as a task graph across two processors [Ni89]. A study of processor allocation and scheduling issues in multiprocessors composed of non-identical processors has been carried out by Menasce, et al. [MDT92].

From the studies discussed above and others [Li90, LG87, ST91], some conclusions that emerge by general consensus are:

(1). space-sharing is preferable to time-sharing, yielding benefits analogous to those of round-robin in single processor scheduling, while avoiding the overhead of frequent preemptions;

(2). in UMA systems, the benefits of the adaptability of dynamic scheduling outweigh the cost of the overhead incurred; and

(3). preemption must be done with care to avoid preempting a process holding a critical resource, such as a lock.

Even if characteristics of individual applications are not known prior to execution, the character of the distribution of service demands for the overall workload may be known. If the coefficient of variation of the service demand distribution is below one, then scheduling algorithms such as FCFS or Run-To-Completion are best. If the coefficient of variation is above one, then priority should be given to the applications with the least attained service. Applications can usually be assigned at least one processor immediately, providing the analogue of the *immediate start* property of RR or PS, which keeps long applications from blocking short ones. Equipartition can be used among applications that share highest priority. Finally, if the service demand distribution has a coefficient of variation close to one, then Equipartition among all the applications is appropriate.

3.3 Scheduling Algorithms in Use

The scheduling algorithms used for scheduling in most computer systems draw on several of the ideas above. Jobs are dispatched based on dynamic priorities. Initially, priorities are assigned based on importance of prompt response (interactive transactions get higher priorities) and expected size (jobs expected to be large get lower priorities). Since the overhead of preemption is not negligible, a quantum-driven scheme is used to switch the processor's attention among jobs of the highest current priority. As jobs receive more service without completing, their priority level decreases. In the case of current UMA shared memory multiprocessor systems, it is typically the case that all processors take work from a common queue, and there is no special treatment of threads that are parts of the same application.

4. Workload Characterization in Parallel Processing Systems

A significant problem in testing and evaluating alternative scheduling strategies for parallel processing systems is the paucity of workload characterization studies in multiprogrammed multiprocessor systems. There are few production systems of this type supporting intense workloads. Currently, most such systems are largely experimental in nature. Computer scientists are just beginning to provide good software support for facilitating the use of such systems for large scale computations, and computational scientists are just learning to fully exploit them.

4.1 Characterizations of Parallel Applications

The execution time function, $T_j(p)$, $p=1, \cdots, P$, for application j depends on the detailed aspects of its execution, such as how evenly its work can be partitioned across the available processors and how much interprocessor communication and synchronization is required. While execution time functions derived by experimentation are generally not smooth, it is useful for the purpose of scheduling (and other reasons) to approximate the execution time curve using only a few parameters.

Several such approximations have been defined and used. Dowdy proposed an application characterization, called the *execution signature*, of the form:

$$\mu_j(p) = \frac{p}{C_{j1}\, p + C_{j2}}$$

where $\mu_j(p)$ is the execution rate of application j when it is assigned p processors and C_{j1} and C_{j2} are two constants that characterize application j [D88]. Taking $1/\mu_j(1) = C_{j1} + C_{j2}$ as the execution time on a single processor, we find that

$$S_j(p) = \frac{C_{j1} + C_{j2}}{C_{j1} + C_{j2}/p}$$

and

$$T_j(p) = C_{j1} + \frac{C_{j2}}{p}$$

Salmon used a different form of execution time model [Sa90]:

$$T_{step} = T_{work} + T_{comm} + T_{imbal} + T_{wait}$$

Here, T_{step} is the time required to carry out one iteration of a parallel N-body simulation, and the four terms on the right-hand side represent respectively (1) the average computation per processor, (2) the average data transfer time per processor, (3) the difference between the maximum computation plus data transfer time across all processors and the average, and (4) the time delay due to synchronization between processors to exchange information.

Ghosal, Serrazi, and Tripathi have investigated the execution of several applications on a 16 processor transputer system, and have shown how execution time curves can be split into portions due to computation time and to communication time [GST91].

We introduce another scheduling model that takes into account some of the dominant influences on speedup as a function of the number of processors assigned, while suppressing the low-level variations. The model accounts for:

(1) the essential computational work of the application,

(2) the imbalance with which the essential work is distributed across the available processors,

(3) the overhead per processor to achieve parallel processing,

(4) other overheads, such as those due to interprocessor communication, that may grow in proportion to the number of processors assigned to the application.

Execution time is approximated by:

$$T_j(p) = \phi_j(p)\, \frac{W_j}{p} + \alpha_j + \beta_j\, p$$

where the four parameters correspond to the four factors listed above. The factor W_j/p represents the ideal division of application j's essential work across p processors. The function $\phi_j(p)$ represents the degree to which the work is *not* evenly spread across the p processors, thus causing starvation on some processors. The amount by which the work per processor is increased by the overhead of parallel processing is represented by parameter α_j. Finally, the communication and congestion delays, which grow with the number of processors, are represented by the term $\beta_j\, p$. Note that Dowdy's formula above is a special case of this one where $W_j = C_{j2}$, $\phi_j(p) = 1$, $\alpha_j = C_{j1}$, and $\beta_j = 0$. The value of $\phi_j(p)$ is the ratio of the *maximum* work assigned to any of the p processors to the *average* work per processor. Hence, $\phi_j(p) \geq 1$ for $p > 1$, but $\phi_j(1) = 1$. Observing that $\phi_j(p)$ is insensitive to p for reasonably large p, we will sometimes represent $\phi_j(p)$ by the constant ϕ_j to gain mathematical tractability. Figure 4.1 illustrates the effects of the various parameters on the shape of the execution time curve by showing several special cases.

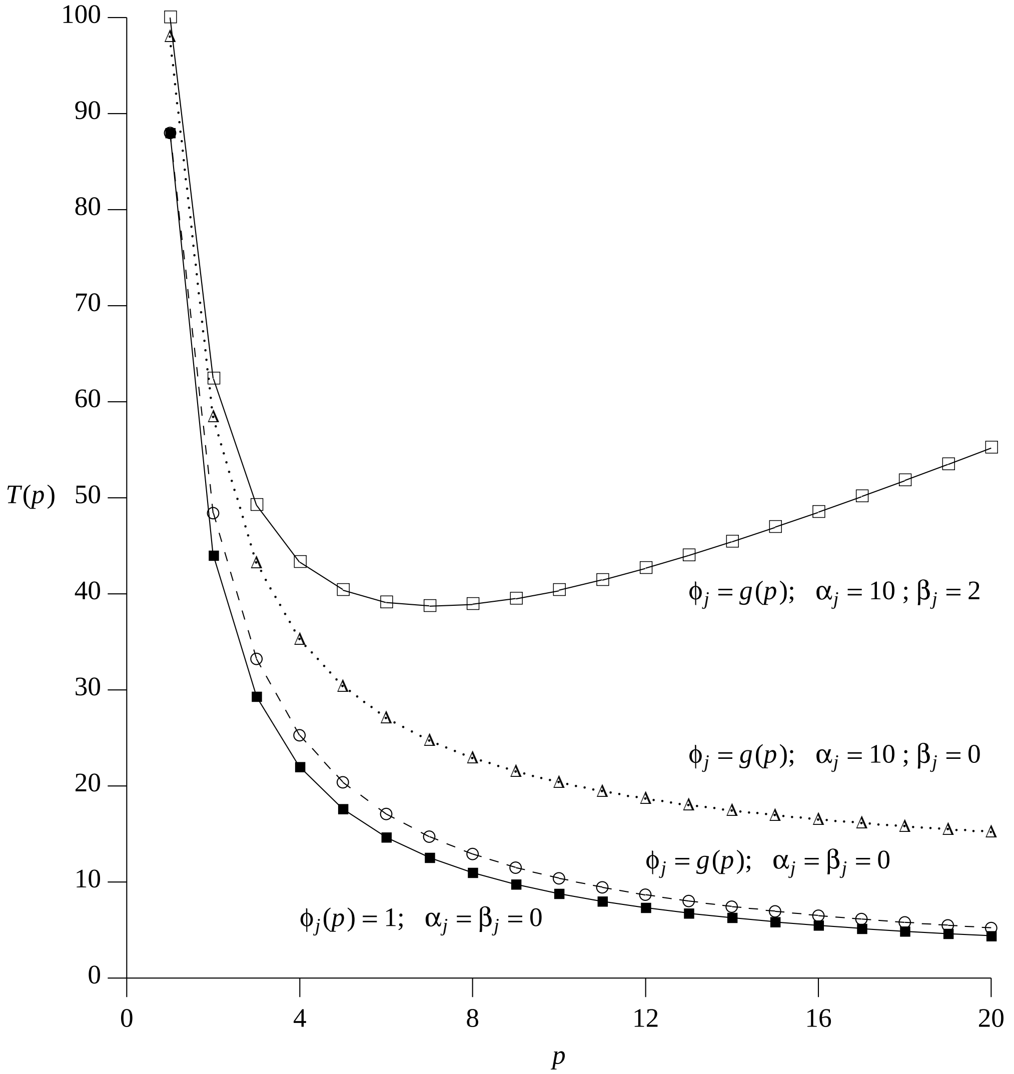

Figure 4.1. The influence of the parameters ϕ, α, and β in the function $T_j(p) = \phi_j(p)\,\dfrac{W_j}{p} + \alpha_j + \beta_j\,p$ (where $g(p)$ is given by $g(p) = 1.2 - .2/p$).

4.2 Obtaining Parameter Value Estimates

In computer systems that support scientific computation, the workloads are generally composed of a variety of applications. They differ in their execution times, priorities, and parallelism properties, as well as their frequencies of execution. In some situations, it is possible to estimate the service times and parallelism properties (probabilistically, if not precisely) before execution begins. As in the case of single processor scheduling, such information about each application permits more effective scheduling.

There are several potential sources for information about a particular application. First, the user could be required to indicate upper bounds on resource requirements, and estimates of other application characteristics. Appropriate scheduling policies can motivate the user to attempt to give accurate information, but a great deal of effort may be required to do so. Second, particular applications that are executed repeatedly could be recognized by the system, and information on their execution characteristics could be accumulated.

In the multiprocessor contexts we are considering, applications are sufficiently large and few that information about the resource requirements of each application can be more precisely and reliably known than in other situations. Taking existing supercomputing centers (most of which are based on vector supercomputers currently) as models, it seems possible to obtain reasonably accurate advance knowledge of the execution characteristics of specific applications since such workloads are dominated by the repeated execution (with different input sets) of a small number of prominent applications. Information about past runs of such applications can be used in scheduling future runs.

Through experimentation and measurement, empirical execution time functions can be determined by repeated execution of an application with different numbers of processors. The measured execution time functions incorporate the effects of insufficient parallelism in parts of the application due to communication, synchronization, etc.

Measured speedup functions are generally concave in shape, but they typically include irregularities of varying degrees. The irregularities complicate the problem of identifying a good schedule and make the problem combinatorial in nature. Unless the number of processors is very small, it is useful to approximate the execution time function of each application by a simple curve that is specified by just a few parameters. The approximation of execution times introduced in section 4.1 is helpful in this regard. It specifies the execution time function based on the parameters W_j, ϕ_j, α_j, and β_j. Scheduling can be done based on these approximations. The schedules that result, while not optimal, are usually quite good relative to other scheduling algorithms. Local modifications can be made to improve the schedule where particular applications happen to be assigned a number of processors corresponding to relative low points in their speedup functions.

For example, in figure 4.2 the actual speedup function of an application is approximated by a speedup function with parameters ϕ_j, α_j and β_j chosen to yield a good fit. Note that at $p = 7$ processors, the actual value of the speedup function of the application lies below the approximation, while at $p = 8$, it lies above. In this situation, if a processor allocation based on the approximation were to assign seven processors to this application, then it might be beneficial to reallocate one processor from some other application to this one in order to give it a better local operating point.

5. Scheduling Algorithms That Use Application Specific Knowledge

5.1. Prespecified Number of Processors Per Application

One form of the multiprocessor scheduling problem studied previously involves scheduling jobs that each require a fixed number of processors for a specified service time. Each application is characterized by (p_j, t_j), where p_j is the (exact) number of processors required by the application, and t_j is the amount of time the application holds the p_j processors to complete its execution. Note that t_j includes the effects of insufficient parallelism to keep all p_j processors busy and any delays due to communication or synchronization among the portions of the computation executing on each processor.

If preemption is not allowed, this problem corresponds to arranging rectangles in two space (although the processors assigned to an application need not be consecutively numbered) such that either average or maximum response time is minimized for a set of N applications, as is illustrated in figure 5.1. Since special cases of this

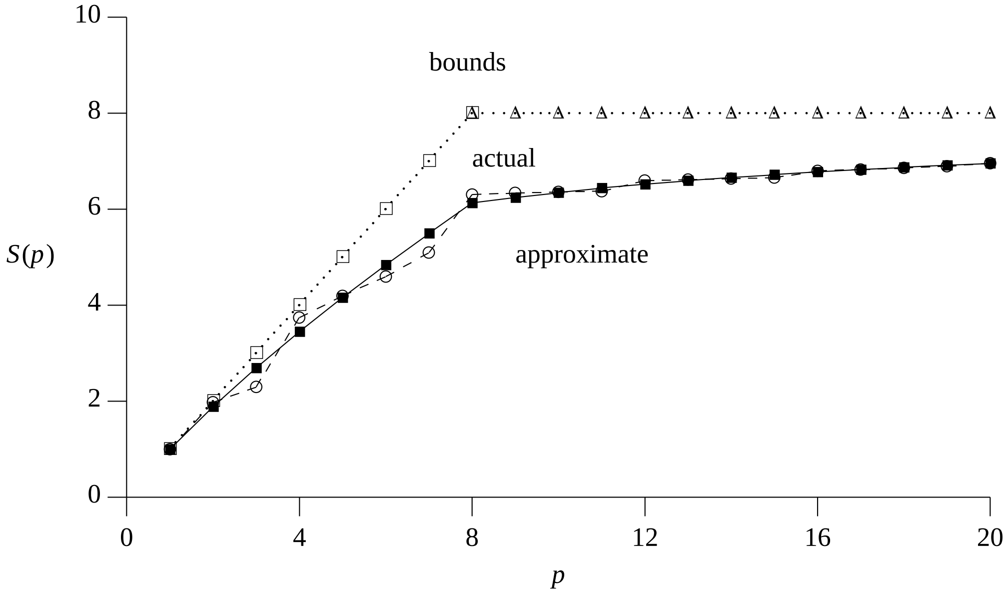

Figure 4.2. An actual speedup function and its approximation.

problem reduce to bin-packing, there is no known efficient scheduling algorithm that minimizes the average completion times of the jobs [GG75].

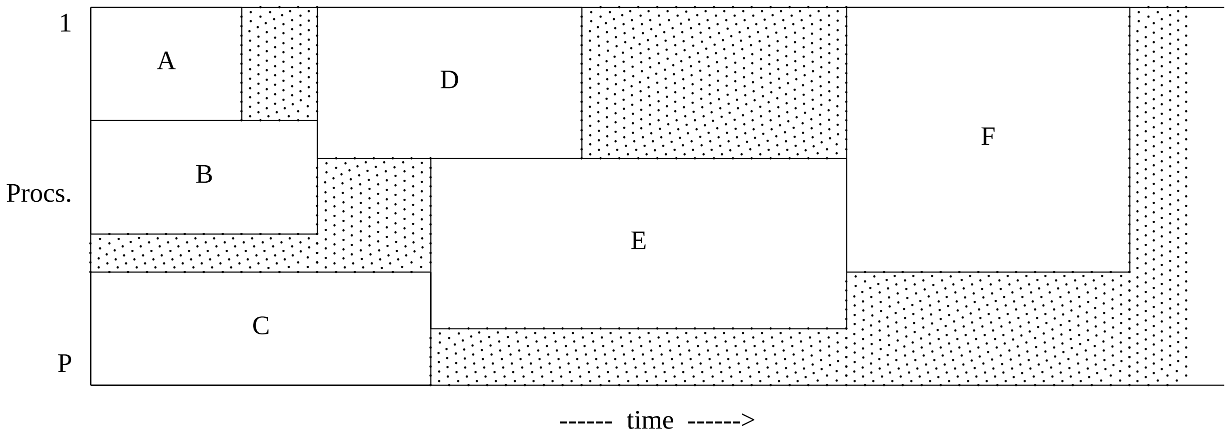

Figure 5.1. Scheduling applications with specified processor requirements.

One useful heuristic in this case is to use a "shelf" based scheduling rule where applications are ordered by non-increasing service time, then grouped onto shelves such that the total number of processors required by applications on any shelf is at most P. The time of the completion of the last application (makespan) is then the sum of the service times of the longest application on each shelf [TWY92, TWPY92].

In order to minimize average (rather than maximum) completion time, different heuristics are needed. One approach is to group applications onto shelves just as before, but to pack the shelves in order of non-decreasing rather than non-increasing service times, so that the applications with shorter service times are completed before those with longer service times. A better heuristic involves activating applications in Least Work First (LWF) order where work is given by $W_j = p_j\, t_j$. In general, schedules generated by these heuristics can be improved by some local rearrangements, but the potential reduction in the average time in system is limited.

5.2. Scheduling Based on Service Time Estimates

A fundamental single processor scheduling problem involves ordering the execution of a set of jobs, each requiring a specified amount of service. If service times are known exactly, then average time in system is minimized by always running the job with the shortest (remaining) processing time (SPT or SRPT). If only the expected service time of each job is known and preemption is disallowed, then running jobs in order of shortest expected processing time first (SEPT) is optimal [CMM67]. If free preemption is allowed, then running jobs in order of shortest expected processing time minimizes the average time in system only if the service time distribution has the property of monotonically non-increasing expected remaining service times, or increasing failure rate [BP65]. If the distribution does not have that property, then a more complex scheduling rule that involves preemption is required in general to minimize average time in system [Se74]. The principle behind the rule is that of minimizing the ratio of resource investment (e.g., expected processor time consumed) to expected payoff (e.g., probability of completing the job with that investment).

In some circumstances, due to inherent priorities, delays to some jobs may be more costly than those to others. If the scheduling objective is generalized to be the minimization of *weighted* average response time where weight c_j is associated with job j, many of the scheduling problems where rules based on shortest (expected) processing time are optimal are then solved by the corresponding rule that minimizes the *ratio* of (expected remaining) processing time to the weight. For example, when service times, t_j, are known, average weighted time in system is minimized by running the jobs in order of increasing (or non-decreasing) t_j / c_j ratio [CMM67].

In scheduling parallel applications on multiprocessors, scheduling algorithms based on total work and number of tasks (threads) have been proposed and evaluated. Majumdar, Eager and Bunt found that such application characteristics can be useful in scheduling. In their workload specification, each application is specified by a total amount of work (cumulative demand) and a number of processes (threads), among which the work is evenly divided. They define the rules Smallest Number of Processes First (SNPF) and Smallest Cumulative Demand First (SCDF), each in both non-preemptive and preemptive forms [MEB88]. Simulation studies showed that SCDF and, to a lesser degree, SNPF perform very well relative to FCFS and RR. The preemptive version of SCDF did particularly well, especially when the variability in work per application was high.

Leuteneggar and Vernon also used simulation to evaluate several scheduling algorithms with a workload definition similar to that of Majumdar, Eager, and Bunt [LV90]. In addition to FCFS, Co-scheduling, SNPF, and preemptive SCDF, they evaluate a dynamic algorithm similar to the one proposed by Zahorjan and McCann [ZM90] and the rules RRjob and RRprocess that use a quantum-driven scheme to share processing power equally among the jobs or processes (threads) respectively. They conclude that RRJob and Process Control are superior to Co-scheduling, RRProcess, and SNPF. Their simulation results show that the performance of preemptive SCDF is consistently better than that of the other algorithms, but they express reservations about the feasibility of implementing it.

In this paper, we will use Least Work First (LWF) and Least Remaining Work First (LRWF) to describe SCDF and preemptive SCDF respectively. The results of these studies confirm the value of using application characteristics in scheduling when they can be obtained. In particular, initiating applications in the order of least (remaining) work first (LWF and LRWF) seems to lead to lower average response times than do other policies. Under certain assumptions, the optimality of LWF can be demonstrated analytically. Assume that all applications attain perfect speedup so that:

$$T_j\,(p) = W_j/p, \quad p = 1, 2, \ldots, P$$

In the absence of arrivals, the scheduling strategy that minimizes average time in system is then Least Work First (LWF) with all P processors being given to each application in turn. More formally, we have:

> *Claim*: Given a set of N parallel applications requiring work $W_1, \cdots, W_N$ respectively, if $T_j\,(p) = W_j/p$ for p=1, $\cdots$,P, then average time in system is minimized by allocating all P processors to the application with the least (remaining) work.

The validity of the claim can be demonstrated by an interchange argument. Consider an arbitrary schedule that assigns the P processors to the N applications such that a total of W_j processor-time units are assigned to application j for j=1, ..., N. Let the applications be indexed such that $W_i \le W_{i+1}$. If there is any pair of applications i and i+1 in a non-LWF schedule such that W_{i+1} gets any processor time at all before W_i is completed, then the schedule can be improved as follows. Take all the time in the schedule assigned to either i or i+1 or left idle before the completion of i and i+1, and reallocate the time such that the earliest W_i units are given to i and the next W_{i+1} units are given to i+1. This exchange will cause i to finish earlier than in the original schedule, by at least as much as i+1 finishes later than originally. Only LWF schedules cannot have their average response times reduced by such an interchange.

If there is Poisson stream of arrivals of new applications and preemption overhead is negligible, then Least Remaining Work First (LRWF) leads to minimum average response time. Under this rule, if a new application arrives and its total work is less than the residual work of the application currently in service, then all P processors are preempted and reassigned to the new arrival.

If preemption is disallowed and only the expected amount of work is known for each application rather than the exact amounts, then Least Expected Work First leads to minimal average response time, either with no arrivals or with Poisson arrivals. When preemption is allowed, average response time can be minimized only if the distribution of service demands is known.

5.3 Analysis of Some Special Cases

In order to gain some insight about the general problem of executing N applications on P processors when overhead is not negligible, we examine some simpler special cases. The insights gained help in identifying principles and guidelines for developing good scheduling algorithms.

(i) One application on P processors

When a single application is executed on a multiprocessor with P processors, the overhead of parallel processing can make it inadvisable to use all P processors. Letting the execution time function be approximated by the function introduced in section 4.1,

$$T_j\,(p) = \phi_j\,(p)\,\frac{W_j}{p} + \alpha_j + \beta_j\,p,$$

it is possible to establish an upper bound on the number of processors that should be assigned to an application.

Unless $\beta_j = 0$, it is clear that, for a sufficiently large value of p, execution time actually increases rather than decreases with the allocation of an additional processor. Hence, for application j, there is a value $p_j^{\max}$ that is the largest number of processors that should ever be allocated to application j, even in the absence of any competing applications. For large p, we may approximate $\phi_j(p)$ by ϕ_j. Treating the execution time function as continuous and setting its derivative with respect to p equal to zero, we find that

$$p_j^{\max} = \sqrt{\frac{\phi_j\,W_j}{\beta_j}}$$

Thus, if $p_j^{\max}$ is sufficiently large that $\phi_j(p)$ is insensitive to p in that range, then no more than $p_j^{\max}$ processors should ever be allocated to application j, as any additional ones would only lengthen its execution time.

(ii) Two applications on two processors ($N = 2$; $P = 2$)

Let applications A and B be characterized by basic work W_A and W_B respectively and speedup values on two processors of $S_A(2)$ and $S_B(2)$, respectively. Assume that

$$T_A(2) \;=\; \frac{W_A}{S_A(2)} \;\le\; T_B(2) \;=\; \frac{W_B}{S_B(2)}$$

In this case, if the two applications are each assigned both processors in turn, then average response time is lower by executing A before B, consistent with the SPT rule.

Next, consider the condition under which executing A then B with two processors each is better than executing them together with one processor each. The first alternative is preferable to the second if and only if:

$$2\,T_A\,(2) + T_B\,(2) < T_A\,(1) + T_B\,(1)$$

Replacing $T_j\,(2)$ by $T_j\,(1)\,/\,S_j\,(2)$, this leads to:

$$T_A\,(1)\left[\frac{2}{S_A\,(2)} - 1\right] < T_B\,(1)\left[1 - \frac{1}{S_B\,(2)}\right]$$

If $S_A(2) = S_B(2)$ (that is, the two applications speedup equally well on two processors), then the inequality above shows that A should be assigned both processors only if the speedup of the two applications is large enough to satisfy

$$S\,(2) \quad > \quad 2 - \frac{T_B(1)}{T_A(1) + T_B(1)} \quad = \quad 1 + \frac{T_A(1)}{T_A(1) + T_B(1)}$$

Since $S_A(2) = S_B(2)$ and $T_B(2) \ge T_A(2)$ by previous assumption, then $T_B(1) \ge T_A(1)$. Thus the bigger $T_B(1)$ is relative to $T_A(1)$, the less speedup with two processors is required to make parallel processing within each application desirable. If $T_A(1) = T_B(1)$, then the speedup with two processors has to exceed 3/2. If $T_A(1) < <T_B(1)$, then speedup with two processors only has to be a little larger than one.

If the applications take the same time to execute on a single processor ($T_A(1) = T_B(1)$) but differ in their speedups on two processors, then A should be assigned both processors if and only if $S_A(2)$ and $S_B(2)$ are big enough to satisfy $\frac{2}{S_A(2)} + \frac{1}{S_B(2)} < 2$.

If only estimates or expected values of W_A and W_B are available, the thresholds for selecting among the three scheduling strategies remain the same.

(iii). N applications on two processors ($P = 2$)

If there are N applications to execute on two processors, an optimal schedule can be determined only by considering all possibilities combinatorially (at least implicitly). Using the LWF heuristic, we assume that two applications with the least work should be executed first. It remains to determine whether to execute these two sequentially, assigning both processors to each application in turn, or together with one processor each.

One case of special interest is where the first two of the N applications to be scheduled are identical (indicated by I). It is desirable to use parallelism for executing the first two applications by assigning both processors to each application in turn if and only if

$$T_I\,(2) + 2T_I\,(2) + (N-2)\,2T_I\,(2) \quad < \quad 2T_I\,(1) + (N-2)\,T_I\,(1)$$

The two expressions in this inequality represent the delays to all applications during the execution of the first two applications for the two scheduling alternatives. The last term on each side of the inequality accounts for the delay imposed on the remaining $N-2$ applications while executing the first two. Replacing $T_I(2)$ by $T_I(1) / S_I(2)$ and simplifying yields

$$S_I(2) > 2 - \frac{1}{N} \quad \text{or} \quad N < \frac{1}{2 - S_I(2)}.$$

Thus, as the number of applications to be executed grows, the speedup with two processors must be very close to two to justify parallelism in executing the first two applications.

The average response time for the batch of N identical applications is minimized by the following strategy. If n is the number of applications remaining to be completed, then as long as $n > \frac{1}{2-S_I(2)}$, assign one processor to each of two applications at a time; thereafter, assign both processors to each remaining application in turn.

If the first two applications to be executed are not identical, parallel processing is more likely to be beneficial. Assume that the first two applications to be scheduled have basic service times satisfying $T_A(1) < T_B(1)$. Then, if they are each assigned two processors in turn, the delay to the other applications will be $T_A(2)+T_B(2)$. If they are each assigned one processor and executed simultaneously, however, it is more difficult to determine what delay to the remaining $N-2$ applications is caused by the execution of the first two. The delay is at least $T_A(1)$ and at most $T_B(1)$, with $(T_A(1)+T_B(1))/2$ being a good estimate particularly when N is large (since approximately the same number of applications would subsequently execute on each of the two processors in that case).

Based on these bounds, we may derive conditions under which the first two applications should be executed one after another with p_j=2. Taking the middle estimate for the delay to the other $N-2$ applications, application A should be assigned both processors initially if and only if

$$N < \frac{\dfrac{1}{S_B(2)}}{\dfrac{R}{S_A(2)} + \dfrac{1}{S_B(2)} - \dfrac{R+1}{2}}$$

where $R = T_A(1) / T_B(1)$. In the special case where $S_A(2) = S_B(2) = S(2)$, the condition reduces to requiring that speedup be large enough to satisfy:

$$S(2) > 2 \left[1 - \frac{1}{N(R+1)} \right]$$

For large N (of about five or more), this threshold is quite accurate. For smaller N, the delay to the other $N-2$ applications depends on their particular characteristics and how they are eventually scheduled. Using the high and low delay estimates above, we may conclude that A and B should be executed serially with two processors each if

$$R > \frac{\dfrac{1}{S_B(2)}}{1 - \dfrac{1}{S_A(2)}} \quad \text{or} \quad N < \frac{\dfrac{1}{S_B(2)} - (R-1)}{\dfrac{R}{S_A(2)} + \dfrac{1}{S_B(2)} - R}$$

and they should be run together with one processor each if

$$R > S_A(2) \left[1 - \frac{1}{S_B(2)} \right] \quad \text{and} \quad N > \frac{\dfrac{1}{S_B(2)} + (R-1)}{\dfrac{R}{S_A(2)} + \dfrac{1}{S_B(2)} - 1}$$

If neither of these conditions holds, then the choice of the better approach is not clear since it depends on the other

$N-2$ applications, but neither can be very bad relative to the other.

If, in addition, the speedups of the two applications with two processors are the same, then A and B should be run serially if

$$R > \frac{1}{S(2)-1} \quad \text{or} \quad S(2) > \frac{R+1-\frac{1}{N}}{R+\frac{1-R}{N}}$$

and together if

$$R > S(2)-1 \quad \text{and} \quad S(2) < \frac{R+1-\frac{1}{N}}{1-\frac{1-R}{N}}$$

Again, if neither of the conditions holds, then there is little difference in the results of the two scheduling alternatives.

Another special case of interest is where the first two applications are identical in basic work ($T_A(1) = T_B(1)$, which means that $R = 1$), but differ in speedups ($S_A(2) \neq S_B(2)$). In this case, the three delay estimates become the same, and the condition for running each of the two applications in turn with $p_j = 2$ each becomes:

$$N < \frac{\frac{1}{S_B(2)}}{\frac{1}{S_A(2)} + \frac{1}{S_B(2)} - 1}$$

Note that when both $S_A(2)=S_B(2)$ and $T_A(1)=T_B(1)$, the first two applications are identical, and all the conditions above reduce to

$$N < \frac{1}{2-S(2)} \quad \text{or} \quad S(2) > 2-\frac{1}{N}$$

which was obtained earlier for the case of two identical applications.

(iv). N identical applications on P processors (where P is a power of two)

Assume that N identical applications are to be scheduled on P (a power of two) processors, and that only processor allocations that are powers of two are permitted. Consider an initial batch of b applications to be executed where b is a power of two that satisfies $b \le \min(N,P)$. Compare the following two alternative ways of executing the first b applications:

- all at once, with P/b processors each, or
- in two waves of $b/2$ at a time, with $2P/b$ processors each.

Then the total time in system for all N applications incurred during the execution of the first b applications under the two scenarios are respectively

$$N\ T_I(P/b) \quad \text{and} \quad \frac{b}{2} T_I(2P/b) + (N - \frac{b}{2})\ 2\ T_I(2P/b)$$

where $T_I(.)$ is the execution time function of at least the first b applications. Therefore, initiating $b/2$ rather than b applications initially is preferable if and only if

$$N\ T_I(P/b) > (2N - \frac{b}{2})\ T_I(2P/b)$$

or

$$\frac{T_I(P/b)}{T_I(2P/b)} \;=\; \frac{S_I(2P/b)}{S_I(P/b)} \;>\; 2 \;-\; \frac{b}{2N}$$

Thus, the ratio $\frac{S_I(2P/b)}{S_I(P/b)}$ must be quite close to two as N grows in order to justify using more internal parallelism by allocating $2P/b$ processors to $b/2$ applications rather than P/b processors to each of b applications.

From this inequality, we can identify b^{best}, the (approximate) best number of applications to activate initially with P/b^{best} processors each. As b ranges from two to P, the right side decreases from $2 - \frac{1}{N}$ down to $2 - \frac{P/2}{N}$, while the left side increases from a value significantly lower than 2 up to a value that is typically close to 2. If the speedup function of the identical applications is smooth then the value of b at which the left and right sides are equal indicates the best value of $b/2$ to use. In particular, b^{best} is the largest power of two that satisfies

$$b^{best} \;<\; 2N\left[2 - \frac{S_I(2P/b^{best})}{S_I(P/b^{best})}\right]$$

and $p^{best} = P/b^{best}$ processors should be allocated to each one.

Using the execution time function introduced in section 4.1,

$$T_j\,(p) = \frac{\phi_j\, W_j}{p} + \alpha_j + \beta_j\, p$$

we may derive

$$\frac{S_I(2p)}{S_I(p)} \;=\; 2 \;-\; \frac{\alpha_I + 3\beta_I\, p}{\dfrac{\phi_I\, W_I}{p} + \alpha_I + 2\beta_I\, p}$$

This, in turn, leads to p^{best} being approximated by

$$p^{best} \;\approx\; \frac{P}{2N}\left[\frac{\dfrac{\phi_I W_I}{p^{best}} + \alpha_I + 2\beta_I p^{best}}{\alpha_I + 3\beta_I p^{best}}\right]$$

Numerical solution of a cubic equation yields an approximation of p^{best}, the preferred allocation.

(v). Two distinct applications on P processors

In this section, we consider how to allocate P processors between two distinct applications, A and B, for each of which the parameters characterizing the execution time functions, W_j, ϕ_j, α_j, and β_j are known. Specifically, we seek fractions of processors f_A and $f_B = 1 - f_A$ to allocate to applications A and B respectively in order to minimize the total time in system,

$$R_{total} \;=\; T_A(f_A\, P) + T_B(f_B\, P).$$

While $f_A\, P$ and $f_B\, P$ must be integers, for sufficiently large values of P, we may disregard this limitation on the choice of f_A. We assume that $P < p_A^{\max} + p_B^{\max}$ so that the allocation problem is non-trivial. We also assume that R_{total} for the best choice of f_A satisfies

$$R_{total} \;<\; T_A(P) + T_B(P) + \min\,(T_A(P), T_B(P)\,)$$

since otherwise it would be better to run one application first then the other.

Since $T_j\,(p) = \dfrac{\phi_j\, W_j}{p} + \alpha_j + \beta_j\, p$, we have:

$$\begin{aligned}
R_{total} &= T_A(f_A\, P) + T_B(f_B\, P) \\
&= \left(\frac{\phi_A\, W_A}{f_A\, P} + \alpha_A + \beta_A\, f_A\, P\right) + \left(\frac{\phi_B\, W_B}{f_B\, P} + \alpha_B + \beta_B\, f_B\, P\right) \\
&= \left(\frac{\phi_A\, W_A}{f_A\, P} + \frac{\phi_B\, W_B}{(1-f_A)\, P}\right) + \left(\alpha_A + \alpha_B\right) + \left(\beta_A\, f_A\, P + \beta_B\,(1-f_A)\, P\right)
\end{aligned}$$

To identify the value of f_A that minimizes R_{total}, $f_A^{\,*}$, we take the derivative with respect to f_A:

$$\frac{dR_{total}}{df_A} = \left[\frac{\phi_B W_B}{(1-f_A)^2} - \frac{\phi_A W_A}{(f_A)^2}\right]\frac{1}{P} + (\beta_A - \beta_B)\, P$$

Thus, $f_A^{\,*}$ satisfies

$$\frac{\phi_A W_A}{(f_A^{\,*})^2\, P^2} - \beta_A = \frac{\phi_B W_B}{(1-f_A^{\,*})^2 P^2} - \beta_B$$

As f_A goes from 0 to 1, the lefthand side of the equality decreases monotonically from ∞ to $\dfrac{\phi_A W_A}{P^2} - \beta_A$, while the right hand side increases from $\dfrac{\phi_B W_B}{P^2} - \beta_B$ to ∞ monotonically. The equality above can be transformed into a fourth order polynomial in $f_A^{\,*}$, that has a unique real root between zero and one, which is the value of $f_A^{\,*}$. The well-behaved character of the two sides of the equation above for f_A between 0 and 1 means that the value of $f_A^{\,*}$ can be determined by a simple iteration, which is found empirically to converge very rapidly. A generalization of the iterative algorithm capable of allocating processors among several applications is given in figure 5.2.

In the special case of $\beta_A = \beta_B$, the value of $f_A^{\,*}$ can be calculated explicitly from

$$\frac{\phi_A W_A}{f_A^{\,*2}} = \frac{\phi_B W_B}{(1-f_A^{\,*})^2}$$

resulting in

$$f_A^{\,*} = \frac{\sqrt{\phi_A\, W_A}}{\sqrt{\phi_A\, W_A} + \sqrt{\phi_B\, W_B}}$$

Thus, assuming that $\phi_A \approx \phi_B$, the optimal share of the processors to give to each application is proportional to $\sqrt{W_j}$. Brecht has previously determined that processor allocation should be proportional to the square root of basic work, and is evaluating this approach relative to some variants [B92].

Generalizing to the case where b applications are designated for activation, this same relationship,

$$\frac{\phi_i\, W_i}{f_i^{\,2}\, P^2} - \beta_i = \frac{\phi_j\, W_j}{f_j^{\,2}\, P^2} - \beta_j \ ,$$

should hold for all pairs of applications among the b. If β_j values are the same for all applications, then the values of $f_j^{\,*}$ should be proportional to $\sqrt{\phi_j W_j}$. If the β_j values are distinct, then applications with relatively high values for β_j should have relatively fewer processors allocated.

Treating the number of processors as large so that the integer constraints can be ignored, estimates of the appropriate fraction of the processors to assign to each of the k applications can be derived from the iterative algorithm in figure 5.2. Since processor allocations must be integral, the quantities $f_j\, P$ obtained at the conclusion of the iteration must be adjusted to nearby integers such that $\sum_{j=1}^{k} f_j\, P = P$.

Input:

$k,\ P,\ \varepsilon$

$\phi_j,\ W_j,\ \beta_j,$ for $j=1, \cdots, k$

Output:

f_j^{opt} , for $j=1, \cdots, k$

Initialize:

$$c_j \ = \ \frac{\phi_j\ W_j}{P^2}$$

$$f_j \ = \ \frac{1}{k} \ \text{for} j=1, \cdots, k$$

$$\gamma \ = \ \sum_{j=1}^{k} \left(\frac{c_j}{f_j^{\,2}} - \beta_j \right)$$

Repeat:

$$f_j \ = \ \sqrt{\frac{c_j}{\gamma + \beta_j}}$$

$$\gamma \ = \ \gamma \sum_{j=1}^{k} f_j$$

Until:

For a small tolerance, ε,

$$\max_j \left(\frac{c_j}{f_j^{\,2}} - \beta_j \right) \ - \ \min_j \left(\frac{c_j}{f_j^{\,2}} - \beta_j \right) \ < \ \varepsilon$$

Finalize:

$f_j^{opt} \ = \ f_j$, for $j=1, \cdots, k$

Figure 5.2. An iterative algorithm for dividing a large number of processors among several applications.

(vi). Processor Allocation with a Large Backlog of Work

If b processors are to be activated initially and there is a very large backlog of unstarted applications, then the objective in allocating processors to the first b applications changes. The goal is to minimize the expected delay imposed on backlogged applications by minimizing the average time processors are occupied by the first b applications:

$$\text{minimize} \quad \sum_{j=1}^{b} p_j\ T_j(p_j)$$

Even when the execution time functions are known, there is no general direct way of determining the optimal allocation of processors for this case. However, when the execution time functions (or the approximations to them) are smooth, a hill-climbing approach that involves successively improving the allocation from an arbitrary starting point by exchanging processors between pairs of processors usually leads to a good allocation.

5.4 Comparison of Approaches

Table 5.1 presents two examples comparing the performance of the following four algorithms:

equi – Each application is assigned P/b processors.

tmin – Processors are allocated to minimize the average completion time $T_{avg} = \frac{1}{b} \sum_{j=1}^{b} T_j(p_j)$

(according to the algorithm of section 5.3(v)).

omin – Processors are allocated to minimize the average processor occupancy, $B_{avg} = \frac{1}{P} \sum_{j=1}^{b} p_j T_j(p_j)$.

(according to the algorithm of section 5.3(vi)).

prop – Processors are allocated in proportion to the basic work of the application.

For each example, the table shows the characteristics of the applications to be executed, the processor allocation by each algorithm, the resulting completion times of each application, and the performance measures, T_{avg}, the average completion time, and B_{avg}, the average processor occupancy. Consistent with the analysis, *tmin* minimizes the average completion time, while *omin* minimizes the average processor occupancy. Note that, in these two examples, *tmin* also leads to nearly minimal average processor occupancy, whereas *omin* causes average completion times to be much higher than what is possible.

6. Synthesizing Complete Scheduling Algorithms

Observations based on the examples investigated in section 5.3 can be used in combination to create complete scheduling algorithms. In general, when the scheduling algorithm is invoked, some processors will be available while others will be assigned to applications in progress. There may also be other applications either partially executed or unstarted that await the assignment of processors.

As described in section 2.2, scheduling decisions may be made when applications complete, when they arrive, when they change phases, or at other points, such as quantum expirations. At each such decision point, it is appropriate to (1) determine the best allocation of processors among applications (both unstarted and partially completed) on the basis of their residual work, and (2) increase and decrease allocations to individual applications to approach the best allocation wherever the benefits of reallocation outweigh the total impact of the reallocation overhead. Some experimental studies have shown that, with proper system support, the preemption and reallocation overhead can be quite low (a few milliseconds) [MVZ91, SL90]. The overhead costs are different for allocating a processor, and for preempting a processor from an unfinished application. Also, when a suspended computation is resumed, there is a cost of lost cache context, even if it is resumed on the processor from which it was suspended. Finally, in distributed memory or NUMA shared memory environments, the relationship between the processor on which a computation is executed and the location of the data it accesses is critical. If an application is assigned additional processors, but they do not have efficient access to the data they need, then either the cost of the remote accesses to required data or the cost of replicating or moving the data must be included in the cost of adjusting the processor allocation.

6.1 Scheduling Without Knowledge of Application Characteristics

If applications all appear statistically identical before their execution starts, equipartition is a suitable basic algorithm. If it is possible to determine how well each application uses its allocated processors, then processors can be shifted from applications that use them poorly to those that use them well. The reallocation overhead will govern the frequency at which the processor allocations should be changed in response to changing behavior of the applications. If not all applications are active, then processors freed by completing applications should be allocated to unstarted applications in order to try to give each application some initial service and complete the small ones

j	W_j	ϕ_j	α_j	β_j
1	10000.00	1.10	1000.00	4.00
2	250000.00	1.20	1000.00	2.00
	equi	omin	tmin	prop
p_1	50	33	16	4
p_2	50	67	84	96
t_1	1420.000	1465.333	1751.500	3766.000
t_2	7100.000	5611.612	4739.429	4317.000
	equi	omin	tmin	prop
T_{avg}	4260.00	3538.47	3245.46	4041.50
B_{avg}	4260.00	4243.34	4261.36	4294.96

j	W_j	ϕ_j	α_j	β_j
1	10000.00	1.20	1000.00	1.00
2	40000.00	1.20	1000.00	2.00
3	100000.00	1.20	1000.00	3.00
4	400000.00	1.20	1000.00	4.00
5	1000000.00	1.20	1000.00	5.00
	equi	omin	tmin	prop
p_1	20	44	5	1
p_2	20	22	9	3
p_3	20	14	14	6
p_4	20	11	28	26
p_5	20	9	44	64
t_1	1620.000	1316.727	3405.000	13001.000
t_2	3440.000	3225.818	6351.333	17006.000
t_3	7060.000	9613.429	9613.429	21018.000
t_4	25080.000	44680.364	18254.857	19565.538
t_5	61100.000	134378.333	28492.727	20070.000
	equi	omin	tmin	prop
T_{avg}	19660.00	38642.93	13223.47	18132.11
B_{avg}	19660.00	19643.81	19735.91	19833.11

Table 5.1. A comparison of four allocation policies.

quickly.

If the distribution of work done by the applications is known from measurements or accounting data, then processor allocation approaches superior to Equipartition may be found. If the variability in the work distribution is low, then batches of applications should be run to completion, with new arrivals not being activated immediately. The appropriate size of the batches depends on the general speedup characteristics of the workload. The better the speedup, the smaller the batches should be, with more processors being allocated to each active application.

If the distribution of work required has a high coefficient of variation (say, two or more), then average response time is kept low by providing service to recent arrivals as much as possible in order to locate and complete the large proportion of small applications. Equipartition processor allocation is appropriate among applications activated at the same time, but new arrivals should be activated immediately if the cost of preemption and reallocation is not too high since since a large proportion will finish quickly. If speedup characteristics of the even the small applications are good and overhead costs are low, then applications with the least attained service at the time should have all the processors divided among them.

6.2 Scheduling With Knowledge of Application Characteristics

As part of a scheduling algorithm, it is necessary to detect when too many applications are being activated. A sufficient condition for this is derived below. Assume that, given N applications to execute, we order them so that $W_1 \leq \cdots \leq W_b \leq \cdots \leq W_N$, and plan to activate applications 1 through b initially.

As seen in section 5.4(v), if b applications are activated, then a good approximation (assuming the β_j 's are nearly the same) of how processors should be allocated is given by

$$p_j = \frac{\sqrt{W_j}}{\sum_{i=1}^{b} \sqrt{W_i}} P, \quad j=1, \cdots, b$$

Consider a reallocation of the $p_1 + p_b$ processors such that application 1 is finished before application b begins. The result is an alternative plan that activates only $b-1$ applications at the start. The new plan does not distribute processors in the best way among the $b-1$ applications, as suggested by the square root allocation. However, if the new plan can be shown to lead to a lower average response time than the original plan, then that is sufficient evidence to conclude that the b^{th} application should not be activated initially.

In the revised plan, the initial processor allocations to applications 2 to $b-1$ are unchanged, but the remaining $p_1 + p_b$ processors are first made available to application 1 (which uses $\min(p_1^{\max}, p_1 + p_b)$ of them) to complete. Then application b is given all of them to execute. (To avoid the assumption that the β_j 's are nearly the same, the tentative allocation among the b applications could be done with the *tmin* algorithm instead, but that would be more computationally costly due to the iteration involved).

The condition under which the new plan leads to expected lower cost is:

$$T_1(p_1) - T_1(p_1+p_b) \; > \; (T_{tot} - T_b(p_b)) + \frac{N-b}{P}\left[p_1(T_{tot}-T_1(p_1)) + p_b(T_{tot}-T_b(p_b))\right]$$

where $T_{tot} = T_1(p_1 + p_b) + T_b(p_1 + p_b)$. This expression is derived in appendix 1. Intuitively, the left side is the reduction in application 1's completion time, while the right side is the increase in completion times of applications b and b+1 through N.

6.3 A Processor Allocation Algorithm Structure

A set of N applications are to be scheduled on P processors, and their work requirements and parameters defining approximations to their individual execution time functions are known to be:

$$[W_j, \phi_j, \alpha_j, \beta_j \mid j=1, \cdots, N]$$

If some applications are partially executed already then interpret W_j as the remaining work to be done. If some processors are already allocated and preemption is not allowed, then interpret P as the number of processors available for scheduling. If new arrivals of applications are anticipated, then include the expected number of them in a busy period in determining the impact of immediate scheduling decisions.

A candidate three-step algorithm for determining how to allocate processors is outlined below.

1. Index the applications according to their (remaining) work requirement, so that $W_1 \le \cdots \le W_b \le \cdots \le W_N$. Applying the Least Work First heuristic introduced in section 5.2, allocate processors only to applications with indices in the range $(1,b)$ for some value of b between 1 and N.

2. Determine the appropriate value for n_{active}, the number of applications to be activated initially, as follows:

   (a). n_{active} should be no less than $n_{\min}$, where $n_{\min}$ is the smallest value of m that satisfies

   $$\sum_{j=1}^{m} p_j^{\max} \ge P$$

   Otherwise, either some processors are left idle or at least one application is being delayed because it is using too many processors.

   (b). n_{active} should not exceed $n_{\max}$, the number of applications that would be activated if all N applications were identical to the first (smallest) one. Applying the result of section 5.3(iv), we find that $n_{\max}$ should satisfy

   $$n_{\max} \ \approx \ 2N\left[2-\frac{S_1(2P/n_{\max})}{S_1(P/n_{\max})}\right]$$

   The value of $n_{\max}$ can be calculated by numerical solution of a cubic equation or by iteration.

   (c). Determine where n_{active} should lie between $n_{\min}$ and $n_{\max}$ using the strategy described in section 6.2 and derived in the appendix. Start with n_{active} initialized to $n_{\min}$, and consider the effect of activating one more application, $b = n_{active} + 1$. Determine tentative processor allocations, $\{p_j\}$, according to the square root allocation algorithm (or the *tmin* algorithm, if the β_j 's differ significantly). With $T_{tot} = T_1(p_1 + p_b) + T_b(p_1 + p_b)$, test the condition:

   $$T_1(p_1) - T_1(p_1{+}p_b) \ < \ (T_{tot} - T_b(p_b)) + \frac{N-b}{P}\left[p_1(T_{tot}-T_1(p_1)) + p_b(T_{tot}-T_b(p_b))\right]$$

   If the condition is true, then increment n_{active} to be b, and, unless n_{active} has reached $n_{\max}$, consider activating yet another application. If the condition is not satisfied, accept n_{active} as the final number of applications to be activated.

3. Knowing n_{active}, the number of applications to be activated, calculate processor allocations based on the *tmin* algorithm of figure 5.2, and also on the *omin* algorithm, which minimizes average processor occupancy as described in section 5.3(vi). The objective function to be minimized is (with $b = n_{active}$):

$$D \ = \ \sum_{j=1}^{b} T_j(p_j) + \frac{(N-b)}{P} \sum_{j=1}^{b} p_j T_j(p_j)$$

where the first term is the sum of the completion times of the applications activated initially, and the second term is an estimate of the average delay in completion of the other $N-b$ applications. If $N{=}b$, then the *tmin* allocation is best, and if N is sufficiently large, the *omin* allocation is best. For intermediate values of N, start with the *tmin* allocation, and shift processors one by one between pairs of applications approaching the *omin* allocations until no further shift decreases average completion time of the set of applications.

Let $\{p_j^t\}$ and $\{p_j^o\}$, $j{=}1, \cdots ,b$ be the processor allocations determined by the *tmin* and *omin* algorithms, respectively. From any given allocation, choose source and destination applications for the movement of one processor as being the applications with the greatest relative excess and greatest relative deficiency of processors respectively relative to the $\{p_j^o\}$ allocation. Set s and d to determine the source and destination applications by

$$\max_{1 \le j \le b} \left(\frac{p_j - p_j^o}{p_j^o} \right) = \frac{p_s - p_s^o}{p_s^o} \quad \text{and} \quad \min_{1 \le j \le b} \left(\frac{p_j - p_j^o}{p_j^o} \right) = \frac{p_d - p_d^o}{p_d^o}$$

The change in D that results from reallocating one processor from s to d is

$$\Delta D_{s,d} = \left(T_s(p_s - 1) - T_s(p_s) \right) + \left(T_d(p_d + 1) - T_d(p_d) \right) + \frac{N-b}{P} \left[\left((p_s - 1) T_s(p_s - 1) - p_s T_s(p_s) \right) + \left((p_d + 1) T_d(p_d + 1) - p_d T_d(p_d) \right) \right]$$

Starting with the *tmin* allocation, repeatedly improve the current allocation by shifting one processor, continuing as long as $\Delta D_{s,d} < 0$ so that overall cost is reduced. Accept the current allocation as final when $\Delta D_{s,d} \ge 0$.

7. Concluding Remarks

We have discussed issues relating to the scheduling of parallel applications on multiprogrammed multiprocessor systems. We have restricted our attention to workloads composed of computationally intensive applications where it is appropriate to focus on the problem of processor allocation and to dedicate each processor to one application, rather than time-sharing processors among applications.

Single processor sequencing results provide direction for designing scheduling rules for multiprocessor contexts. If the scheduling objective is to minimize average time in system for applications, then the Least Work First (LWF) heuristic, derived from the Shortest Processing Time (SPT) first rule, is a useful basis on which to build. When only expected rather than exact total processing time requirements are known and processor allocation is done statically, then Least Expected Work First (LEWF) is generally still the best basis from which to work. Under dynamic scheduling, if the variability in work per application is high, preemption is required to keep average response time low by completing applications with small work requirements quickly. If priorities of applications differ, then weighted average response time can be kept low by activating applications in increasing order of the ratio of expected work to loss rate constant (LEWF/c).

For heavily loaded systems, it is necessary to maximize the efficiency of use of all the processors. At the extreme, this involves allocating very few (possibly just one) processor per application. When there are few applications awaiting execution, processors may be used at lower efficiency in order to complete some of the applications quickly.

We have focussed primarily on scheduling a batch of applications in the absence of arrivals. If the arrival times and execution characteristics of future arrivals are known exactly or probabilistically, determining the best schedule becomes extremely difficult. The additional information makes better scheduling feasible, but only at the cost of using a computationally expensive combinatorial algorithm. If it is assumed only that future arrivals are Poisson (i.e., with exponentially distributed inter-arrival times), then processors should not be held in reserve for future arrivals. The memoryless property of the exponential interarrival time distribution means that precise arrival times or resource requirements cannot be anticipated. However, the overall load intensity sustained by a Poisson arrival stream can be taken into account. Allocating processors to applications that do not use them at full efficiency will cause delays not only to applications already available for execution, but also to other applications anticipated to arrive before the backlog of work is exhausted.

Memory space, a critical resource in many large-scale scientific applications, was not accounted for directly in the algorithms we have discussed (although memory management delays appear implicitly in execution time function estimates). In workloads including memory-intensive applications, processor allocation algorithms can be adjusted to activate fewer applications at a time, allocating more processors to each one, in order to avoid an overcommitment of memory.

Performance is affected by communication and synchronization costs that result from the interactions of threads of an application. Also, congestion for network and memory access may cause delays, particularly in distributed memory or NUMA shared-memory systems. If execution time functions are measured, the effects of communication, barrier synchronization and memory access congestion will be reflected as starvation of processors for some portion of time. Of course, a performance model that is intended to allow investigation of variations that involve changing the amount of communication among threads, or the number of barrier synchronizations, must be sufficiently detailed to include explicit representation of these effects.

In bus based systems, with low degrees of parallelism, only the number of processors assigned to each application matters. However, for scalable NUMA architectures, the particular set of processors allocated is quite significant. Such systems typically have natural allocation units consisting of sets of processors that can communicate efficiently. In such systems, processors should be allocated in the natural groups in order to exploit the efficient communication [ZB91].

A further complication in NUMA shared memory multiprocessors is that memory allocation and processor allocation must be coordinated in order to keep memory access times low on average [ML91]. In such systems, even if the overhead to preempt and reallocate a processor may be low, doing so might substantially increase the average memory access times for some processors, thus reducing their effectiveness. Appropriate memory reallocation can be done to match the processor reallocation, but the required overhead would typically be quite high.

The number of large-scale parallel systems for supporting production workloads of computational science applications is growing. More knowledge of and experience with these systems is needed to determine the extent to which the characteristics of individual applications can be practically and reliably predicted. This paper has presented guidelines for developing processor allocation algorithms capable of using such information to improve the performance of parallel systems.

Acknowledgements

Development of this material has benefited significantly from numerous discussions with and comments from participants in the scheduling discussion group at the University of Toronto, most notably T. Abdelrahman, T. Brecht, B. Carlson, K. Harzallah, H. Li, N. Srikantiah, M. Stumm, S. Zhou, and D. Zilio. Discussions with and suggestions from R. Bunt, L. Dowdy, D. Eager, E. Lazowska, R. Nelson, G. Serrazi, S. Tripathi, and J. Zahorjan were also very helpful. Finally, comments from the co-editors and an anonymous referee contributed greatly toward improved organization and presentation of the material.

References

[A67] G.M. Amdahl, Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities, *Proc. AFIPS Conf. 30*, (1967), 483-85.

[B92] T. Brecht, Processor Scheduling in Shared-Memory Multiprocessors, Ph.D. thesis in preparation, University of Toronto (1993).

[BP65] R.E. Barlow and F. Proschan, *Mathematical Theory of Reliability*, John Wiley and Sons (1965).

[CD73] E.G. Coffman and P.J. Denning, *Operating Systems Theory*, Prentice Hall, 1973.

[CD92] B.M. Carlson and L.W. Dowdy, Static Processor Allocation in a Soft Real-Time Multiprocessor Environment, to appear in *IEEE Trans. on Parallel and Distributed Systems* (1993).

[CDDLM91] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos, Multiprogramming on Multiprocessors, *Proc. 3rd IEEE Symp. on Par. and Distr. Proc.*, Dallas (December 1991), 590-97.

[CMM67] R.W. Conway, W.L. Maxwell, and L.W. Miller, *Theory of Scheduling*, Addison-Wesley, 1967.

[D88] L.W. Dowdy, On the Partitioning of Multiprocessor Systems, Tech. Rpt. 88-06, Dept. of Comp. Science, Vanderbilt University, (March 1988).

[DCDP90] K. Dussa, B. Carlson, L. Dowdy, and K.-H. Park, Dynamic Partitioning in a Transputer Environment, Proc. 1990 ACM Sigmetrics Conf., *Performance Evaluation Review 18*, (May 1990), 203-213.

[EZL89] D.L. Eager, J. Zahorjan, and E.D. Lazowska, Speedup Versus Efficiency in Parallel Systems, *IEEE Trans. on Computers C-38*, 3 (March 1989), 408-23.

[GG75] M. Garey and R. Graham, Bounds for Multiprocessor Scheduling with Resource Constraints, *SIAM J. Comp. 4*, 2 (June 1975), 187-200.

[GST91] D. Ghosal, G. Serazzi, and S. Tripathi, Processor Working Set and its Use in Scheduling Multiprocessor Systems, *IEEE Trans. on Soft. Eng. TSE-17*, 5 (May 1991), 443-53.

[GTS91] A. Gupta, A. Tucker, and L. Stevens, Making Effective Use of Shared-Memory Multiprocessors: The Process Control Approach, Tech. Rpt. CSL-TR-91-475A, Computer Systems Laboratory, Stanford University (July 1991).

[GTU91] A. Gupta, A. Tucker, and S. Urushibara, The Impact of Operating System Scheduling Policies and Synchronization Methods of the Performance of Parallel Applications, Proc. 1991 ACM Sigmetrics Conf., *Performance Evaluation Review 19* (May 1991), 120-32.

[K87] M. Kumar, Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications, *IEEE-Trans. on Comp. 37*, 9 (September 1988), 1088-98.

[Li91] H. Li, Scheduling in Multiprogrammed Multiprocessors, Ph.D. Depth Report, University of Toronto, (1991).

[LDP89] M.R. Leuze, L.W. Dowdy, and K.-H. Park, Multiprogramming a Distributed Memory Multiprocessor, *Concurrency: Practice and Experience 1*, 1 (September 1989), 19-33.

[LG87] S.-P. Lo and V.D. Gligor, A Comparative Analysis of Multiprocessor Scheduling Algorithms, *Proc. 7th Int. Conf. on Dist. Comp. Systems* (Sept. 1987).

[LV90] S.T. Leuteneggar and M.K. Vernon, The Performance of Multiprogrammed Multiprocessor Scheduling Policies, Proc. 1990 ACM Sigmetrics Conf., (May 1990), *Performance Evaluation Review 18*, 226-36.

[MDT92] D.A. Menasce, S.C. daSilva Porto, and S.K. Tripathi, Processor Assignment in Heterogeneous Parallel Architectures, *Proc. Int. Parallel Proc. Symp.* (March 1992), Los Angeles.

[MEB88] S. Majumdar, D.L. Eager, and R.B. Bunt, Scheduling in Multiprogrammed Parallel Systems, Proc. 1988 ACM Sigmetrics Conf., *Performance Evaluation Review 16*, (May 1988), 104-13.

[MEB91] S. Majumdar, D.L. Eager, and R.B. Bunt, Characterization of Programs for Scheduling in Multiprogrammed Parallel Systems, *Performance Evaluation 13*, 2 (October 1991), 109-30.

[ML91] E.P. Markatos and T.J. LeBlanc, Memory Conscious Scheduling in Shared-Memory Multiprocessors, Computer Science Dept., University of Rochester (1991).

[MN91] A.M. Makowski and R. Nelson, Distributed Parallelism Considered Harmful, Tech. Rept. RC 17448, IBM Research (Dec. 1991).

[MN92] A.M. Makowski and R. Nelson, Optimal Scheduling for a Distributed Parallel Processing Model, Tech. Rept. 17449, IBM Research (Feb. 1992).

[MS88] A.J. Musciano and T.L. Sterling, Efficient Dynamic Scheduling of Medium-grained Tasks for General Purpose Parallel Processing, *Proc. Int. Conf. on Par. Proc.* (1988), 166-175.

[MVZ91] C. McCann, R. Vaswani, and J. Zahorjan, A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors, to appear in *ACM Trans. on Computer Systems* (1993).

[Ni89] D.M. Nicol, Optimal Partitioning of Random Programs Across Two Processors, *IEEE Trans. on Soft. Eng. 15*, 2 (February 1989), 134-41.

[NTT88] R. Nelson, D. Towsley, and A.N. Tantawi, Performance Analysis of Parallel Processing Systems, *IEEE Trans. on Software Eng. SE-14*, 4 (April 1988), 532-40.

[O82] J.K. Ousterhout, Scheduling Techniques for Concurrent Systems, *Proc. 3rd Int. Conf. on Distr. Comp. Syst*. (October 1982), 22-30.

[PD89] K.H. Park and L.W. Dowdy, Dynamic Partitioning of Multiprocessor Systems, *Int. J. of Parallel Programming 18*, 2 (1989), 91-120.

[RSDSC92] E. Rosti, E. Smirni, L.W. Dowdy, G. Serazzi, and B.M. Carlson, Robust Partitioning Policies of Multiprocessor Systems, Vanderbilt University (1992), to appear in *Performance Evaluation*.

[Sa90] J. Salmon, Parallel Hierarchical N-body Methods, Ph.D. thesis, California Institute of Technology, Report CRPC-90-14 (December 1990).

[Se74] K.C. Sevcik, Scheduling for Minimum Total Loss Using Service Time Distributions, *J.ACM 21,* 1 (January 1974), 66-75.

[Se89] K.C. Sevcik, Characterizations of Parallelism in Applications and Their Use in Scheduling, Proc. 1989 ACM Sigmetrics Conf. (May 1989), *Performance Evaluation Review 17*, 171-80.

[Sr91] N. Srikantiah, Processor Scheduling in Hierarchical NUMA Multiprocessors, M.Sc. Thesis, Dept. of Comp. Sci., University of Toronto (1991).

[SL90] M.S. Squillante and E.D. Lazowska, Using Processor Cache Affinity Information in Shared-Memory Multiprocessor Scheduling, Tech. Rpt. 89-06-1, Dept. of C.S. and Eng., University of Washington, (February 1990), to appear in *IEEE Trans. on Parallel and Distributed Systems*.

[SST92] S.K. Setia, M.S. Squillante, and S.K. Tripathi, Analysis of Processor Allocation in Multiprogrammed Parallel Processing Systems, Tech. Rept. CS-TR-2840, Dept. of Computer Science, Univ. of Maryland (Feb. 1992).

[ST91] S. Setia and S.K. Tripathi, An Analysis of Several Processor Partitioning Policies for Parallel Computers, CS-TR-2684, C.S. Dept., University of Maryland (May 1991).

[SZ92] K.C. Sevcik and S. Zhou, Performance Benefits and Limitations of Large NUMA Multiprocessors, Tech. Rept. CSRI-256, Computer Systems Research Institute, University of Toronto, submitted for publication (November 1992).

[TRS90] D. Towsley, G. Rommel, and J.A. Stankovic, Analysis of Fork-Join Program Response Times on Multiprocessors, *IEEE Trans. on Parallel and Distributed Systems 1,* 3 (July 1990), 286-303.

[TG89] A. Tucker and A. Gupta, Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors, *Proc. 12th ACM Symp. on Oper. Sys. Princ.* (Dec. 1989).

[TWPY92] J. Turek, J.L. Wolf, K.R. Pattipati, and P.S. Yu, Scheduling Parallelizable Tasks: Putting it All on the Shelf, Proc. 1992 ACM Sigmetrics Conf., *Performance Evaluation Review 20,* (May 1992), 225-236.

[TWY92] J. Turek, J.L. Wolf, and P.S. Yu, Approximate Algorithm for Scheduling Parallelizable Tasks, *Proc. SPAA '92* (1992).

[ZB91] S. Zhou and T. Brecht, Processor Pool Based Scheduling For Large-Scale NUMA Multiprocessors, Proc. ACM Sigmetrics Conf., *Performance Evaluation Review 19,* (May 1991), 133-142.

[ZM90] J. Zahorjan and C. McCann, Processor Scheduling in Shared Memory Multiprocessors, Proc. 1990 ACM Sigmetrics Conf., (May 1990), *Performance Evaluation Review 18*, 214-25.

Appendix 1. Activating b versus b − 1 Applications

Compare the original and new scheduling plans as introduced in section 6.3. The differences in the two plans are indicated in figure A1. Application 1 finishes sooner because it uses $p_1 + p_b$ processors rather than p_1. Applications 2 through $b-1$ are unaffected. Application b finishes later in the new plan because, although its execution time is reduced due to using more processors, it is activated only after application 1 completes. Finally, some of the applications $b+1$ to N will likely be delayed because applications 1 and b execute less efficiently due to using more processors. Not taking into account the details of scheduling applications $b+1$ to N, we assume the delay is averaged over them by having fraction p_1/P finish at $T_1(p_1+p_b) + T_b(p_1+p_b)$ instead of at $T_1(p_1)$, and fraction p_b/P finish at $T_1(p_1+p_b) + T_b(p_1+p_b)$ instead of at $T_b(p_b)$. Thus, the new plan is an improvement if and only if the reduction in application 1's completion time is greater than the sum of the expected delays in completion time to applications b and $b+1$ through N. Letting $T_{tot} = T_1(p_1+p_b) + T_b(p_1+p_b)$, the new plan is an improvement if

$$T_1(p_1) - T_1(p_1+p_b) \; > \; (T_{tot} - T_b(p_b)) + \frac{N-b}{P}\left[p_1(T_{tot} - T_1(p_1)) + p_b(T_{tot} - T_b(p_b))\right]$$

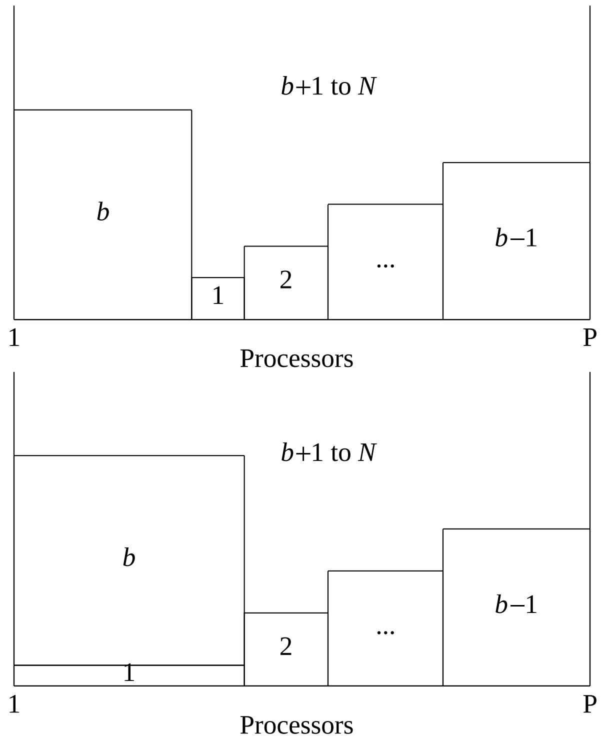

Figure A1. A comparison of the original and new scheduling plans.

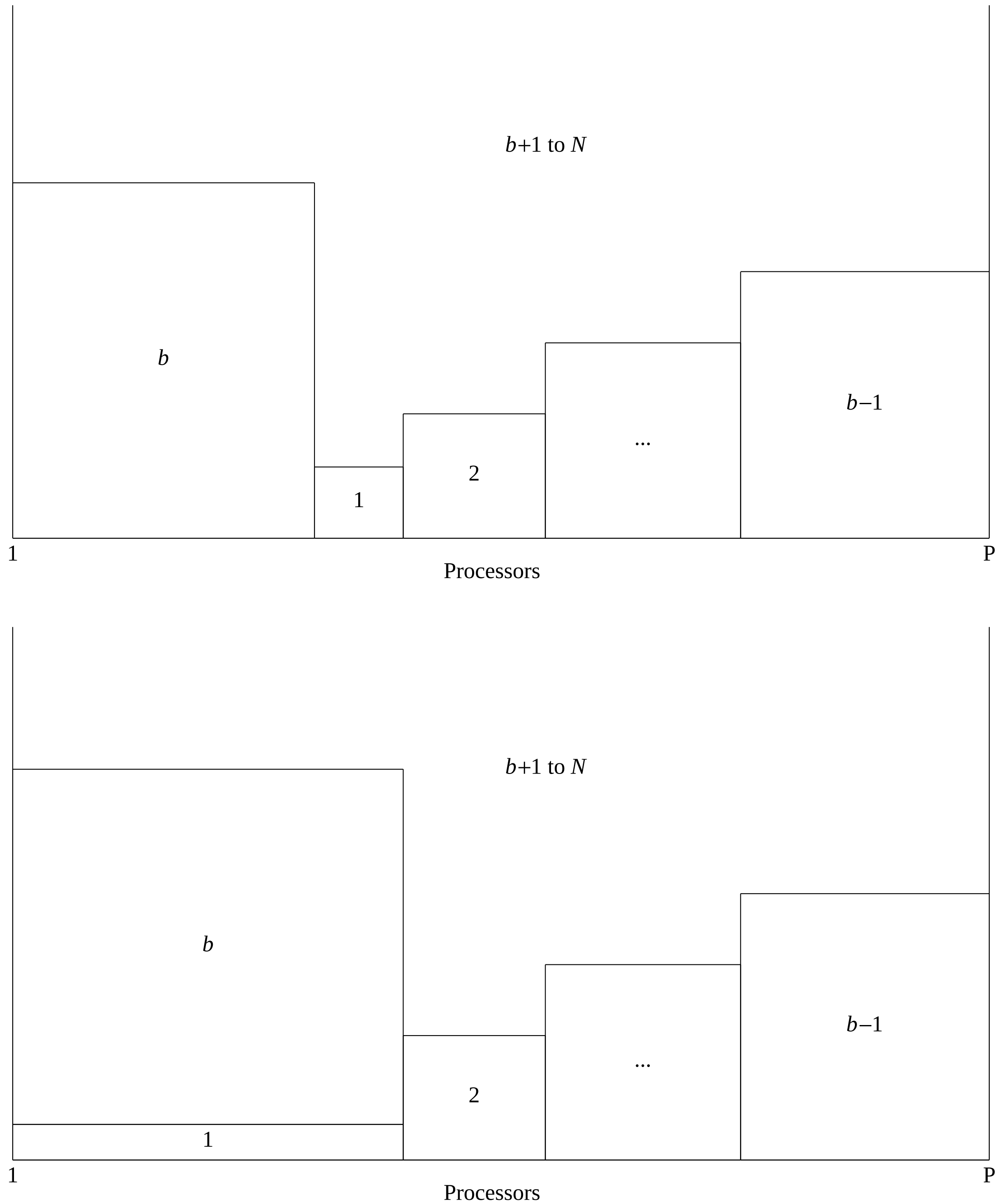

Figure A1. A comparison of the original and new scheduling plans.

<end of report>