Condensed Version

Job Scheduling in Multiprogrammed Parallel Systems

Dror G. Feitelson*
Institute of Computer Science
The Hebrew University, 91904 Jerusalem, Israel

Version of August 1997

Abstract

Scheduling in the context of parallel systems is often thought of in terms of assigning tasks in a program to processors, so as to minimize the makespan. This formulation assumes that the processors are dedicated to the program in question. But when the parallel system is shared by a number of users, this is not necessarily the case. In the context of multiprogrammed parallel machines, scheduling refers to the execution of threads from competing programs. This is an operating system issue, involved with resource allocation, not a program development issue.

Scheduling schemes for multiprogrammed parallel systems can be classified as one or two leveled. Single-level scheduling combines the allocation of processing power with the decision of which thread will use it. Two level scheduling decouples the two issues: first, processors are allocated to the job, and then the job's threads are scheduled using this pool of processors. The processors of a parallel system can be shared in two basic ways, which are relevant for both one-level and two-level scheduling. One approach is to use time slicing, e.g. when all the processors in the system (or all the processors in the pool) service a global queue of ready threads. The other approach is to use space slicing, and partition the processors statically or dynamically among the different jobs. As these approaches are orthogonal to each other, it is also possible to combine them in various ways; for example, this is often done in gang scheduling. Systems using the various approaches are described, and the implications of the different mechanisms are discussed. The goals of this survey are to describe the many different approaches within a unified framework based on the mechanisms used to achieve multiprogramming, and at the same time document commercial systems that have not been described in the open literature.

Keywords: parallel system, scheduling, multiprogramming, time slicing, space slicing, processor allocation, mapping, two-level scheduling, partitioning, gang scheduling.

*The original version of this work was done while at the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598

Contents

1 Introduction

In uniprocessors, scheduling is the activity of deciding which thread of control gets to run on the CPU. In multiprocessors and multicomputers, scheduling has another dimension: not only deciding when a thread will run, but also *where* it will run, i.e. on which Processing Element (PE). Thus parallel systems allow a two-dimensional division of resources among competing jobs, both in time and in space.

Apart from the allocation of computing resources to competing jobs, there is also the question of allocation to cooperating threads within a single job. This can be done by the operating system, by the language runtime system, or by the application itself. The scope of this paper is oriented more towards scheduling by the operating system, but some of the mechanisms that are surveyed are equally applicable to scheduling at a higher level. We start with a distinction between two basic approaches to scheduling in multiprogrammed parallel systems: single-level and two-level. The single level approach combines the issues of PE allocation to the job and scheduling threads on those PEs. The two-level approach decouples the two issues. First, the operating system allocates PEs to the job, and then the application itself (or the language runtime library) schedules the threads. This eliminates operating system overhead from each scheduling decision, and allows for application-specific optimizations.

While single-level scheduling and two-level scheduling are conceptually very different, the mechanisms used are largely overlapping. Indeed, two-level scheduling can often be viewed as a combination of two scheduling mechanisms, one for PE allocation and the other for scheduling on a given set of PEs, where each of the two can also be considered a single-level scheduling scheme in its own right. The bulk of the survey therefore focuses on a framework for describing scheduling mechanisms, emphasizing the fact that most mechanisms can be used alone or as part of a two-level scheme.

The major distinction among different mechanisms is whether they are based on time slicing or space slicing. This distinction is extremely important, because space slicing is often associated with exclusive allocation of the PEs to a given application. This entails some degree of loss of control by the operating system. Time slicing is more flexible, and scheduling decisions have less impact on future performance. However, this comes at the price of higher overheads.

Section 2 presents a classification of scheduling mechanisms based on the use of time slicing, space slicing, or both. Specific systems are then surveyed according on the mechanisms used: partitioning, whether fixed, variable, adaptive, or dynamic (Section 3), independent PEs with global or local queues (Section 4), and gang scheduling, where PEs are scheduled in unison (Section 5). For each scheme, we identify the models of computation for which it is useful, and discuss the interaction of the scheduling scheme with other aspects of the parallel operating system activities, such as thread mapping, load balancing, and memory management. The implications of various combinations and options are explored in Section 6. Finally, Section 7 surveys a number of real systems which use different mechanisms to support different classes of applications. A glossary of terms is given in Appendix A.

Scope of this survey

The issues mentioned above are not relevant for all parallel systems; as the title implies, we are dealing only with multiprogrammed machines, and specifically those that support interactive use. Let us first delimit the scope of this survey by saying what will not be covered.

First, we shall not discuss the classic NP-complete problem of algorithmic scheduling, a.k.a. the mapping problem [346, 177, 142, 36, 266, 7, 5]. This problem deals with how to schedule a program represented as a graph of tasks with interdependencies on multiple processors.

Second, scheduling under real-time constraints is also not discussed, as it is quite different from scheduling in multiprogrammed general-purpose systems. Real-time systems try to find a schedule that satisfies a set of designated deadlines [289, 288, 330, 317, 183, 23, 198], whereas multiprogramming is more concerned with throughput and fair sharing of resources. Likewise, scheduling issues on parallel database machines are not included [43, 88], as they too are quite different from general purpose systems.

Third, we will not discuss scheduling on heterogeneous networks (usually identified by the new buzzword "metacomputing") [138, 59, 150, 290, 134]. The issue there is to match parts of the application to the most appropriate hardware, taking other load into account. Thus schedulers for metacomputing environments must interact with the local schedulers of (parallel) machines that compose the environment, such as those described in this survey.

Finally, many multiprocessor multiprogrammed systems are used as throughput machines, running independent sequential jobs on the different PEs [283]. This survey does not cover such usage. It deals exclusively with scheduling of parallel programs. Likewise, we do not discuss machines dedicated to a single (parallel) application, as there is obviously no role for scheduling in this case.

What is left is general-purpose parallel machines, that are used simultaneously by multiple users to execute both interactive and batch jobs. It can be argued that this leads to inefficiency and lesser utilization, because interactive response times require time slicing to be used, leading to higher system overheads and reduced cache efficiency. Nonetheless, the market requirement is there, and most vendors of parallel machines make a point of their support for interactive use. An attempt was made to survey the full range of ideas, spanning both paper designs from academia and practical ones that are commercially available.

A note on terminology

For the sake of self-containment and completeness, the following paragraphs define our usage of some basic terminology. Additional terms are added as needed throughout the paper. All the terms are also summarized in Appendix A.

Throughout this paper, we use the term *PE* (processing element) to denote the constituents of a parallel system. A PE typically includes a CPU, some memory, and mechanisms to communicate with other PEs in the system (either directly or through the use of shared memory). We use the term *thread* to indicate the entity that may be scheduled to run on a PE. Depending on the context, this may be something like a full Unix process or else it could be a light-weight process. In addition, we make no a-priori distinction between kernel threads and user threads. Such distinctions are only made in those cases where the described mechanisms are only suitable for one type but not for the other.

A *job* is an application in execution, as known by the operating system. Thus the set of threads that together execute a given application typically comprise a job. The exception is when a set of *processes*, which are autonomous as far as the operating system knows, are actually part of a single application. In this case, the operating system regards each one as a separate job.

Multiprogramming denotes systems that support the concurrent execution of multiple independent jobs. In parallel systems, this can be done either by context switching among threads from the different jobs, or by partitioning the machine and executing different jobs on different partitions. The term *multitasking* will be used to denote cases where a number of threads — from the same job or from competing jobs — execute concurrently on the same PE, using time slicing. Thus these terms are not synonymous.

Scheduling is the aggregate set of decisions that determine which thread will execute on a certain PE at a certain time. It may be composed of a combination of *processor allocation*, i.e. *partitioning* of the system among the applications, and *dispatching*, that is selecting the next thread from those that are ready and memory-resident.

2 A Classification of Multiprogramming Schemes

Multiprogramming is the activity of executing multiple jobs concurrently on the same machine. Textbooks on operating systems introduce multiprogramming as a solution to the problem of low resource utilization, where an adequate mix of jobs with complementary requirements keep all parts of the system busy [195, 279]. For example, I/O operations from one job can be overlapped with useful computation for another job, rather than leaving the CPU idle until the I/O completes. This is achieved by time slicing, which also creates the opportunity for interactive response times. For many users, the capacity for interactive use is even more important than the improved utilization.

The same considerations apply to parallel systems. Again, utilization is improved by a proper mix of jobs, but with the added dimension of having jobs with different degrees of parallelism running side by side (i.e., one job utilizes PEs left over by another). Again, sharing provides users with better access to the system, and — depending on the application — even with interactive response times. Again, this is important both because it enables users to make faster progress towards a solution, and because adequate access for multiple users is often crucial in order to fund costly parallel supercomputers.

The importance of multiprogramming is clear, and many systems provide this feature. However, they do so in a variety of ways. In this section, we look at different classifications of multiprogramming schemes, and at the reasons that led to the development of different schemes.

2.1 Single-Level Scheduling and Two-Level Scheduling

Probably the most basic dichotomy in parallel scheduling is the distinction between single-level scheduling and two-level scheduling. In single-level scheduling the act of allocating a processing resource is combined with the act of deciding which thread will use this resource. In two-level scheduling, these two aspects of scheduling are decoupled. The first level deals with resource allocation, and the second with its use.

The problem with single-level scheduling is the fear that leaving all scheduling to the operating system is too expensive, and not responsive enough to application needs. Note that many scheduling decisions are a result of synchronization conditions among the threads of the application. For example, one thread may block waiting for another thread to send a message or reach a synchronization point. In fine grain applications, such interactions can occur at high rates. Paying the operating system overhead for each one would be prohibitively expensive. In addition, the operating system cannot optimize the scheduling because it lacks information about the application's characteristics and patterns of interaction.

The proposed solution is two-level scheduling. The operating system just allocates the computing resources, i.e. PEs and memory. This is done at a relatively low rate related to the job submission rate. Because PEs are allocated to jobs, sharing is done by space slicing, and applications might have more threads than PEs. The application itself (or the runtime system) then does the actual fine-grain scheduling of threads on the allocated PEs, in a way that satisfies the synchronization constraints. This level of internal scheduling provides high flexibility in resource allocation. For example, it is possible to create systems where the PE allocation changes at runtime, and the application is expected to adjust accordingly. This approach is suitable for systems where the computation is represented as a task graph or as a workpile of chores, which are executed by a variable number of worker threads. Such systems are reviewed in Section 3.4.

It should be noted, however, that two-level scheduling is not universally accepted. This is largely due to the fact that its use is somewhat limited in its applicability. It is perfectly suitable for relatively small, shared memory machines, using the workpile of chores programming model, because then chores can indeed be executed on any PE. It is less suitable for distributed memory architectures, especially if programs are written in the prevalent SPMD style.

In fact, it is possible to identify three types of single-level scheduling that are commonly used. One is single-level scheduling that corresponds to the first level of two-level scheduling, i.e. processor allocation. Such systems just partition the PEs among the submitted jobs, and then run a single thread on each PE. For example, this approach is suitable for programs written in the SPMD style and executed in batch mode. It is also very simple to implement, and has low operating system overhead. Consequently, it has received widespread use on many systems. Partitioning is reviewed in Section 3.

Another approach corresponds to the second level of two-level scheduling, i.e. time slicing to execute more threads than there are PEs. The differences from two-level scheduling are that all the PEs are used rather than only a pre-allocated subset, and that all threads are considered, rather than only those belonging to a certain application. However, the mechanisms used are the same. Furthermore, scheduling multiple threads on a smaller number of PEs is a relatively simple extension of uniprocessor systems, where the number of PEs happens to be 1. Therefore time slicing systems have also been quite popular; they are reviewed in Section 4.

The time slicing systems described above are distinguished by the fact that the scheduling on each PE is independent of the others. A third approach to single-level scheduling is to perform coordinated scheduling on all the PEs (or on subsets of PEs). This is known as *gang scheduling*. The justification is that if all the threads in an application are scheduled to run simultaneously on different PEs, the application sees the same environment as it would on a dedicated machine. In particular, synchronization constraints will be limited to those that are inherent to the application, and will not result in operating system activity. On the other hand, gang scheduling provides more flexible resource sharing than pure partitioning, because time slicing is also used. While harder to implement than the previous two types of single-level scheduling, gang scheduling has the benefit of providing applications with a better approximation of a dedicated machine. Therefore this approach is gaining in approval, as witnessed by the increasing number of commercial system that provide it. Gang scheduling systems are reviewed in Section 5.

2.2 The Two Dimensions of Sharing

Other surveys of scheduling and resource allocation have used various classifications. For example, Casavant and Kuhl classify different scheduling algorithms according to their algorithmic design [56]. Mauny *et al.* take a broader look at resource allocation in general (including vector registers, memory, etc.), and study its interactions with the programming model and architecture [236]. Our classification of scheduling schemes for multiprogrammed parallel systems is based on the way in which computing resources are shared: temporal sharing, spatial sharing, or both. Fig. 1 presents a taxonomy based on these options. Naturally, scheduling schemes used in real machines are not designed specifically to ease their classification. Therefore this classification is not always completely adequate for the description of certain systems. However, it provides insights into the basic choices.

The main observation is that the mechanisms of time-slicing are largely independent of those for space-slicing. Thus the classification can be represented by a two-dimensional grid, with time-slicing on one axis and space-slicing on the other. The fact that practically all the squares in the grid are occupied with examples of real systems testifies to the independence of the mechanisms for the two sharing schemes.

On the time-slicing axis, the main distinction is between mechanisms that apply to each PE individually and mechanisms that handle a group of PEs as a single unit. Mechanisms for independent PEs are further divided into those that use local queues, thereby requiring

			time slicing			
			yes			no
			independent PEs		gang scheduling	
			global queue	local queues		
space slicing	yes	flexible	Mach	Paragon/service Meiko/timeshare KSR/interactive transputers Tera/streams Chrysalis	Medusa Butterfly@LLNL Cray T3E Meiko/gang Paragon/gang SGI/gang Tera/PB MAXI/gang	IBM SP2, Victor Meiko/batch Paragon/slice KSR/batch 2-level/bottom TRAC, MICROS Amoeba
		structured		NX/2 on iPSC/2 nCUBE	CM-5 Cedar DHC on SP2 DQT on RWC-1	Cray T3D CM-2 PASM hypercubes
	no		IRIX on SGI NYU Ultra Dynix 2-level/top Hydra/C.mmp	StarOS Psyche Elxsi AP1000	MasPar MP2 Alliant FX/8 Chagori on K2	Illiac IV MPP GF11 Warp

Figure 1: *Grid of combinations of mechanisms for time-slicing and space-slicing.*

that threads be mapped to PEs before they can be scheduled, and those that use a shared global queue, thereby blending the actions of mapping and scheduling. These mechanisms are described in detail and compared in Section 4.

Mechanisms that perform time-slicing on groups of PEs actually implement gang scheduling, which we define to mean preemptive scheduling of a certain set of threads *simultaneously* on distinct PEs, with a one-to-one mapping of threads to PEs (i.e. either all these threads execute or none execute). Gang scheduling requires coordinated context switching across the PEs (also called *multi-context-switching*), which is harder to implement than independent context switching. However, this approach is gaining in popularity, and a number of recent commercial systems provide gang scheduling. These systems and others are reviewed in Section 5.

It should be noted that the groups of PEs that perform coordinated context switching may be created as part of a space-slicing scheme that partitions the machine. On the other hand, there may be no partitioning at all: all the PEs can be used for only one job at a time. The same applies for groups of PEs sharing the use of a global queue. This is another manifestation of the independence of time-slicing from space-slicing.

Space-slicing mechanisms are reviewed in Section 3. In essence, space-slicing is a bin packing problem: how to fit applications side by side, with best utilization of the available PEs. Many different heuristics have been proposed. Some use a hierarchical structure to guide the partitioning. Others limit the number of options that have to be considered by performing the partitioning within a framework of predefined static partitions. Still others relay the problem to the application programmer, by requiring applications to adjust to whatever number of PEs the system can provide under the current loading conditions.

The large number of possible mechanisms and the fact that they are independent of each other implies that the design space for multiprogrammed parallel systems is very large. To help evaluate these options, Section 6 lists the implications of the various sharing schemes in terms of performance and functionality.

2.3 The Roots of Divergence

As seen from the grid of Fig. 1, many different scheduling schemes have been proposed and implemented. While similar machines often use similar scheduling schemes, different types of machines are often scheduled in very dissimilar ways. Moreover, there is a rather wide gap between theoretical studies and schemes proposed by academia on the one hand, and what is done in practice in large installations on the other hand.

The reasons for such divergence are that the assumptions leading to and justifying the different schemes are usually quite different [123, 124]. Thus, while all the schemes are designed to solve the general problem of "how to schedule parallel jobs on a parallel machine", the detailed assumptions make each instance of this problem distinct from the others. Naturally, this leads to different solutions. Regrettably, it also means that it is often hard if not meaningless to try and compare the different solutions with each other — in many cases, this is like comparing apples to oranges.

A basic conceptual difference that underlies much of the divergence is the type of system and how it is used. Many theoretical studies involve off-line systems, and search for optimal solutions given that everything is known up front and nothing changes. Real systems, however, operate in an on-line open environment, and need to contend with unpredictable arrivals of new work.

A more concrete source of differences is assumptions about the system architecture. For example, the availability of shared memory makes the use of a shared queue natural and straightforward, while its absence makes it harder to implement. Furthermore, if the shared memory is centrally located, rather than being distributed among the PEs, its allocation is completely decoupled from the allocation of processors. This means that it is easy to re-allocate processors at runtime, because such reallocations do not require any data move-

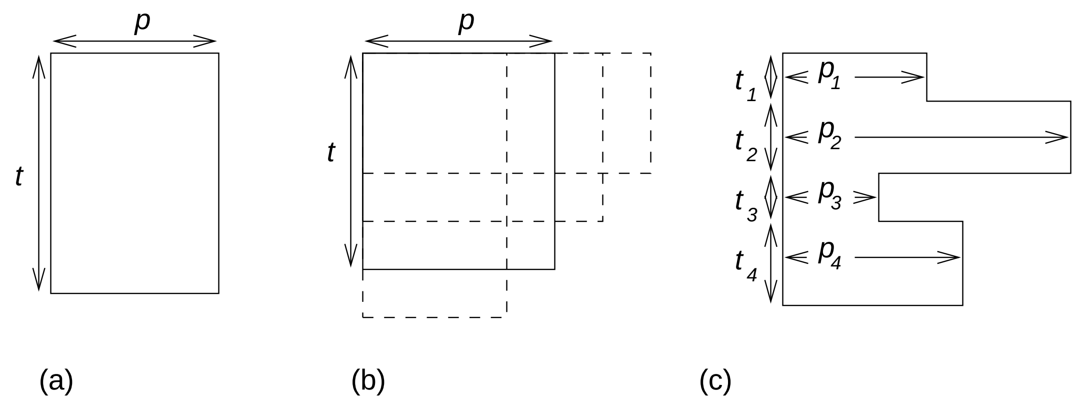


Figure 2: *(a) Rigid jobs define a rectangle in processor-time space. (b) Moldable jobs use one out of a choice of such rectangles. (c) Evolving and malleable jobs both have a profile with a changing number of processors. The difference is that in evolving jobs the changes are initiated by the job, while in malleable ones they are initiated by the system.*

ments. If global memory is not available, then processor reallocation becomes a more expensive operation. In addition, there may be different assumptions about the basic operations supported by the system software (preemption, migration, swapping), and the amount of information available to the scheduler [124].

Another source of differences is assumptions about the jobs and their capabilities in relation to processor allocation. The following classification has been proposed (Fig. 2) [123]:

Rigid: jobs that require a certain predefined number of processors. They will not run on less, and will not utilize more. The system has no choice but to grant the requested number.

Moldable: jobs that allow the number of processors to be set at the outset, but it cannot change thereafter. Thus the system may affect the number of PEs used, for example reducing it if the job is submitted under conditions of heavy load.

Evolving: jobs with changing requirements, e.g. a sequence of serial and parallel phases. At the beginning of each phase, the job requests the system for the resources it needs for this phase, and at the end of the phase it releases them. This allows the system to re-assign processors that fall out of use.

Malleable: jobs that can adjust to changing allocations at runtime. This allows the most flexibility to the system: when PEs become available, they can be immediately allocated to running jobs so as to reduce fragmentation; when new jobs arrive, they can run immediately using PEs preempted from other jobs.

Quite naturally, malleable jobs have been assumed in many academic studies, and the resulting systems have been shown to be very efficient. However, this model is much less popular with users, and is not supported by most programming environments. As a consequence, the suggested scheduling schemes cannot be used in most real production systems, leaving system administrators with other "suboptimal" solutions.

In addition to explicit assumptions about the environment and workload, the gap between theory and practice is also fueled by simple non-technical concerns. Practitioners who need to support large communities of real users are bound by such issues as standard user interfaces, backward compatibility, the need to support political scheduling, etc. Academics have the freedom to ignore such dull issues. On the other hand, academics are sometimes limited by questions of mathematical tractability, whereas practitioners may devise complex and elaborate schemes that defy any type of rigorous analysis.

3 Partitioning

Space-slicing is done by partitioning the machine among a number of applications that execute side by side. This approach is motivated by the desire to reduce operating system overhead on context switching [342], and by the desire to exploit architectural decisions not to provide support for virtual memory and paging, but rather to give all the physical memory to a single application.

Sometimes partitioning is the only mode of sharing. In this case, the operating system is actually involved more in PE allocation than in scheduling. The application's runtime system may then schedule user threads on these PEs, leading to a two-level scheduling scheme [366, 341]. This is especially common on shared-memory multiprocessors. The second level of scheduling is often done using a global queue, with the same considerations and optimizations as described in Section 4.2, e.g. taking processor affinity into account. On distributed-memory multicomputers, it is more common to have only one thread on each PE. This section concentrates on the mechanisms of partitioning *per se*.

In general, partitioning is limited by the hardware. Not every machine can be partitioned. For example, SIMD[1] architectures cannot be partitioned unless it is possible to associate an instruction interpretation unit with each partition, together with a mechanism to broadcast the instructions to the PEs in the partition. Likewise, some interconnection topologies cannot be partitioned effectively. For example, a mesh can be partitioned easily, but partitioning a torus requires special switches that re-link the row and column rings. Hypercubes can only be partitioned into subcubes.

Given that the hardware allows for partitioning, the main issue that remains is where to place the partition boundaries. Some systems do not provide much of a choice. This makes the implementation simple, but risks waste and user frustration due to mismatch

[1]SIMD stands for Single Instruction-stream Multiple Data-stream, and MIMD stands for Multiple Instruction-stream Multiple Data-stream [129]. In SIMD architectures there is a single instruction decoding unit, which broadcasts the instructions to multiple processing units. Thus each instruction is applied synchronously and in parallel to multiple data elements that reside in the same address on distinct PEs. In MIMD architectures PEs execute independently of each other, except for explicit synchronization.

between the available partitioning patterns and the actual requirements. Other systems allow any partitioning that one might wish for. This provides welcome flexibility, but requires sophisticated algorithms to make the partition decisions. The following criteria have been proposed to evaluate and compare partitionable systems [228, 295]:

- *Independence* — distinct partitions should be as independent as possible. In particular, they should not share hardware such as switches or links.
- *Flexibility* — it should be possible to allocate partitions of arbitrary sizes, composed of arbitrary subsets of PEs. Otherwise, PEs will be lost to fragmentation.
- *Adaptiveness* — the partitioning should reflect the requirements of the workload, so as to promote good utilization. As the workload typically changes with time, so should the partitioning.
- *Cost and complexity* — should be low. This refers both to the hardware needed to build the system, and to the algorithms used to run it.
- *Modularity and scalability* — solutions should be useful for large systems.

The classification of partitioning schemes is complicated by the fact that three players are involved: the architecture, the operating system, and the applications. We shall classify the partitioning mechanisms into four types, based on how the operating system behaves and the requirements this places on applications [260, 295]: fixed, variable, adaptive, and dynamic. These are summarized in Table 1 and discussed in Subsections 3.1 through 3.4, respectively. As indicated in the table, the progress towards dynamic partitioning is motivated by the desire to eliminate fragmentation and improve resource utilization. However, this comes at the price of reduced locality and independence: PEs from different parts of the machine may be called to take part in executing the same job, and local state may be wiped out when a PE is re-allocated to another job. Therefore dynamic partitioning has been proposed mainly for small-scale shared memory machines.

If time slicing is not used, jobs will be queued when the requested resources are not available. A review of FCFS, SJF, and other algorithms for scheduling queued jobs onto partitions is therefore included in Section 3.2.3. This effect is reduced under dynamic partitioning, as PEs can then be preempted from jobs that are already running. Alternatively, queueing can be eliminated by combining partitioning and time slicing, as in gang scheduling. This is especially relevant for fixed and variable partitioning.

Section 3.5 mentions another option, that of allocating PEs one at a time.

3.1 Fixed Partitioning

Fixed partitions are set by the system administrator, typically for reasons related to access control. This allows certain parts of the machine to be dedicated to certain groups of users, possibly according to their investment in procuring the machine. Alternatively, the partitions can be designated for different job classes [243, 260]. For example, many commercial parallel

type	*operating system*	*application runtime*	*advantages*	*disadvantages*
fixed	predefined partitions	parallelism hardcoded or a parameter	simple, preserves locality	internal fragmentation, limited multiprogramming, arbitrary queueing
variable	allocation according to request (possibly rounded up)	hardcoded parallelism	matches requests, preserves locality	external (and possibly internal) fragmentation, arbitrary queueing
adaptive	allocation subject to load when launched	parallelism is a parameter	preserves locality, adapts to load, improved efficiency	external fragmentation, some queueing
dynamic	allocation changes at runtime to reflect changes in load and requirements	express changes in potential parallelism, and adapt to changes in available parallelism	no fragmentation, queueing only under high load, adapts to load, improved efficiency	does not preserve locality, partitions not independent, restriction on programming model

Table 1: *The four types of partitioning.*

systems include provisions for creating a batch partition and an interactive partition. This provides adequate resources for the interactive part of the workload, without starving the batch part. The sizes of these partitions can be changed automatically twice a day, to accommodate the differences between daytime activity and nighttime activity. However, the partitioning is not changed to accommodate individual jobs.

A major concern with fixed partitioning is internal fragmentation: jobs get all the PEs in the partition, or none. If the job only requires a small number of PEs, the rest are left unused. A partial solution is to create several partitions with different sizes, so that users can choose the most appropriate one. However, internal fragmentation can still occur. In addition, excessive load on the best partition might induce users to choose a less appropriate one. Another solution is to use a second level of variable partitioning or time slicing within the fixed partitions, rather than applying these ideas to the whole machine.

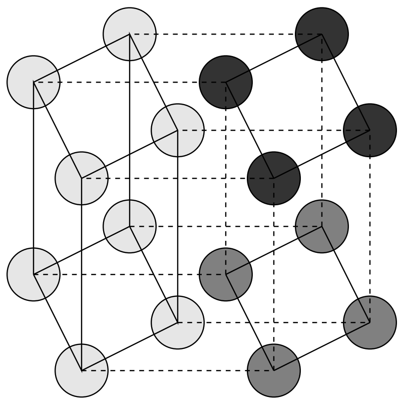


Figure 3: *Example of partitioning a 4-cube into a 3-cube and a pair of 2-cubes. Dashed links are not used in this configuration.*

3.2 Variable Partitioning According to Requests

Variable partitioning is similar to fixed partitioning, except for the fact that partition sizes are not predefined — rather, they are set on the fly according to incoming requests. This is done in either of two ways: by using combinations of predefined partition sizes, or by creating arbitrary partitions.

The most common reason for using combinations of predefined partitions is that the partitioning is required to match the architecture of the machine. For example, this is the case in the partitioning of hypercubes. Jobs on hypercubes typically rely on the hypercube topology, and therefore cannot execute on an arbitrary subset of nodes. Rather, the hypercube is partitioned into subcubes (Fig. 3). In this case, there is no internal fragmentation. However, internal fragmentation may occur in cases where the allocation is rounded up to one of the predefined sizes, but the job does not utilize the extra PEs.

If the architecture does not impose any restrictions, arbitrary partitions can be created. This eliminates internal fragmentation, because a job is never allocated more PEs than it can use. However, external fragmentation remains an issue because a set of idle PEs might be left which is not large enough to satisfy the requests of any queued jobs.

The following subsections review partitioning based on powers of two, which is a very common design, and arbitrary partitioning. Then the question of scheduling order is addressed.

3.2.1 Combinations of Partitions Based on Powers of Two

Many parallel machines are not just an ensemble of PEs. Rather, they are built as a group of constituents, each of which may be further divided into even smaller parts, before single PEs are reached. For example, a full-sized Connection Machine CM-2, with 65536 single-bit PEs, is composed of four quadrants of 16384 PEs each [343]. Each quadrant has an independent sequencer — the unit that interprets the SIMD instruction stream and broadcasts it to

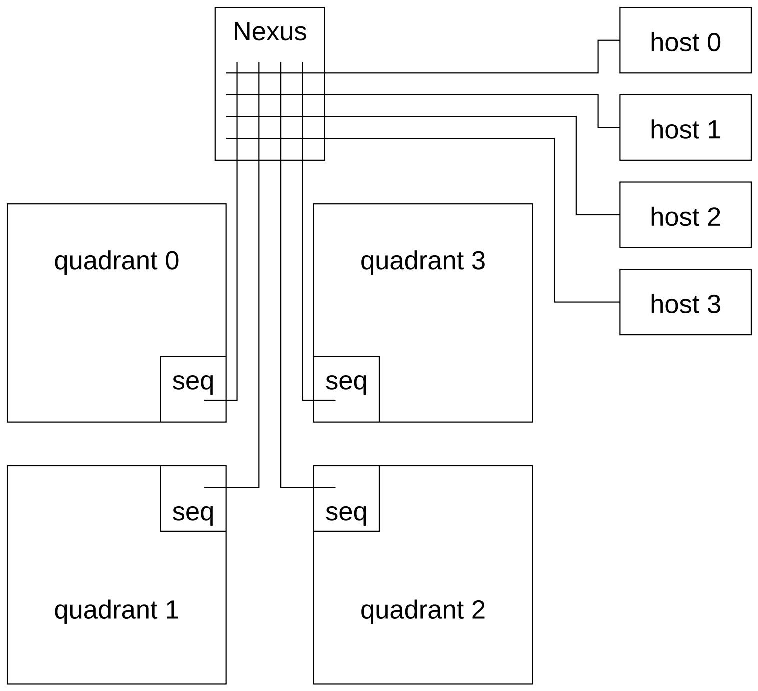


Figure 4: *Partitioning the CM-2 SIMD array (adapted from [343]).*

the PEs — and can thus be used to run a separate program. Thus the machine can be partitioned into a maximum of four partitions. Alternatively, two or four quadrants can be combined and used to run a single program. This is controlled by the nexus, which is a 4×4 crosspoint switch that interfaces the host front-ends to the sequencers (Fig. 4). The repertoire of partition sizes offered by the CM-2 is thus rather limited.

In many other cases the structure of the machine is still based on powers of two, but more levels of partitioning are supported. In the CM-5, the basic unit is 32 PEs. In Cedar, it is 8. In the Cray T3D, each allocatable node contains a pair of PEs. In hypercubes, single PEs can be allocated. These basic units are grouped into pairs, then groups of four, eight, and so on, and the possible partitions follow the same structure Hence partition sizes can be any power of two between some minimum and the full machine.

Two important examples of this are parallel machines based on multistage networks and hypercubes. In the multistage machines, each partition contains a power-of-two PEs, the same number of memory modules, and a unique part of the network. In hypercubes each partition is a subcube, that is a hypercube of a smaller dimension. Both these architectures create independent partitions, but suffer from lack of flexibility that may lead to fragmentation.

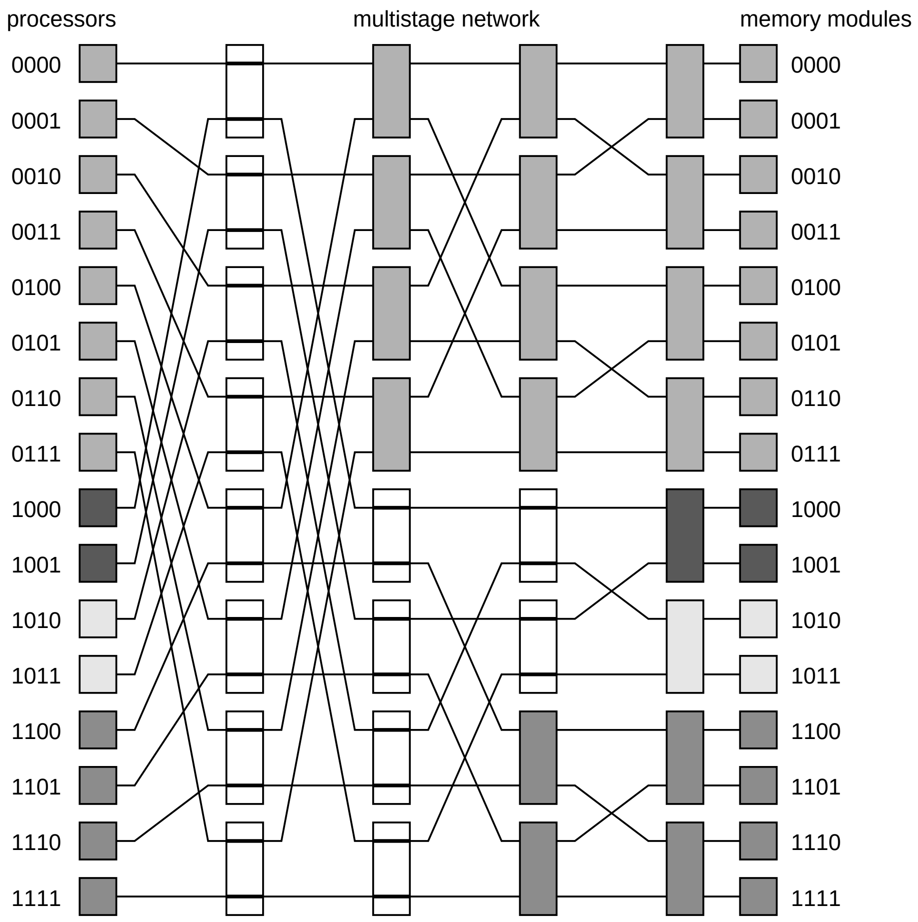


Figure 5: *partitioning a multistage network into independent partitions (adapted from [322]). The elements of each partition are identified by a different shading. Switches set to "straight" are not shaded.*

Partitioning multistage networks

Multistage networks connect n input ports to n output ports, with a $\log n$ delay and a $\frac{1}{2}n \log n$ component count. Such networks are used both for shared memory machines and for distributed memory machines. In shared memory machines, the input ports are PEs and the output ports are memory modules; examples include the BBN Butterfly, The NYU Ultracomputer, the IBM RP3, PASM, TRAC, and Cedar (where clusters are connected to the network rather than individual PEs). In distributed memory machines, all ports are connected to PEs. Examples include the CM-5. the Meiko CS-2, and the IBM SP2.

As a concrete example, consider the generalized cube network. An example for $n = 16$ is given in Fig. 5. Partitioning this network is achieved by setting some of the switches to the "straight" mode, meaning that they just connect their top input to their top output, and their bottom input to their bottom output [322]. Setting the first stage to "straight" partitions the machine into two halves. Within each partition, setting an additional stage to "straight" halves that partition. The setting shown in Fig. 5 results in partitions of 8, 2, 2, and 4 PEs. These partitions may be described by the serial numbers of the PEs and memory modules included in them as $\{0, x, x, x\}$, $\{1, 0, 0, x\}$, $\{1, 0, 1, x\}$, and $\{1, 1, x, x\}$, where x represents either 0 or 1. Networks can also be partitioned in other ways, e.g. by setting the last stage to "straight" rather than the first one, resulting in partitions that are defined by other bit patterns [321, 176, 320, chap. 5].

A number of prototype systems have used the idea of partitioning a multistage network. Perhaps the best known is PASM, a partitionable SIMD/MIMD machine built at Purdue [324, 323]. This architecture imposes a limit on how small partitions can be, because — like the CM-2 — it needs a control module to allow each partition to operate in SIMD mode. The prototype has 16 PEs and 4 controllers, so partitions can have 4, 8, or 16 PEs. Only predefined pairs of partitions, chosen by bit patterns similar to those shown above, can be combined to create larger partitions. The matching of jobs that can execute side by side is done by keeping separate queues for the different partition sizes [344].

Research prototypes are not the only machines that use partitionable multistage networks. The Connection Machine CM-5 is based on a network that is logically seen as a fat tree (a tree in which links near the root are "thicker" and provide more bandwidth so as to prevent congestion), but actually implemented as a multistage network [214]. Unlike PASM or TRAC, the CM-5 is a distributed memory machine. The machine is partitioned among competing jobs by partitioning the network. Each partition includes 2^k PEs for some $k \geq 5$ (i.e., the minimal partition size is 32), with the adjoining $k/2$ stages of the network[2]. Thus there is no network interference from other jobs. A separate partition is devoted to I/O devices. It is accessed via that part of the network that is farther away from the PEs.

Finally, it is worth noting that not all machines based on multistage networks that support partitioning do so by partitioning the network. The Meiko CS-2, IBM SP2, and BBN Butterfly allow arbitrary nodes to be grouped into the same partition. This increases the flexibility of PE allocation, at the price of creating partitions that are not independent.

Partitioning hypercubes

Hypercubes are multicomputers with a power-of-two PEs, which are interconnected in a hypercube pattern. A hypercube with 2^n PEs is said to be n-dimensional. The 2^n PEs are identified by unique n-bit long IDs. PEs with IDs that differ in exactly one bit position have a direct link connecting them. The ancestor of all contemporary hypercubes is the Cosmic Cube built at Caltech in the early '80s [308]. A number of commercial offerings have been made since then, the most enduring of which are the iPSC series from Intel [17] and the nCUBE machines [158, 271, 100]. Both provide partitioning as described below.

Many hypercube applications are specifically tailored to the topology. Therefore if an application is to run on a partition of the machine, that partition must itself be a hypercube,

[2]It is $k/2$ stages rather than k stages because 4×4 switches are used.

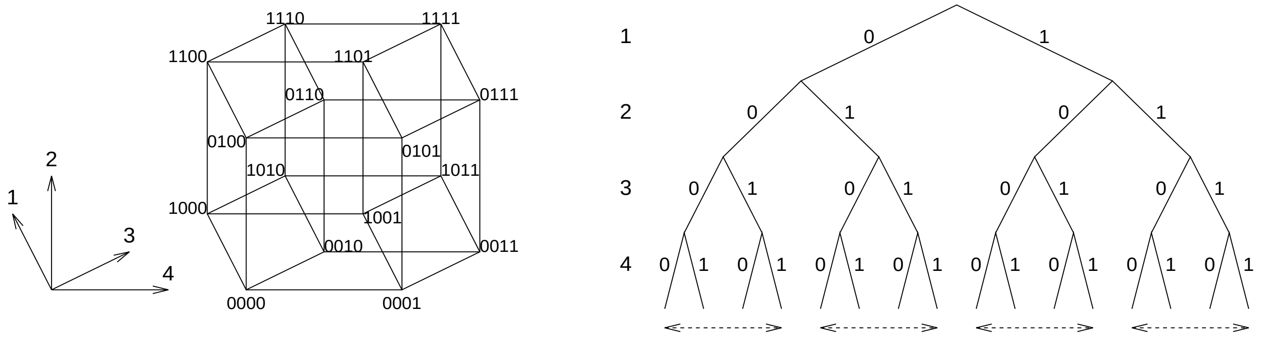


Figure 6: *The buddy system maps dimensions of the hypercube to levels of a binary tree, and uses the tree structure to partition the hypercube.*

albeit of a smaller dimensionality[3]. It is easy to see that there are many ways in which to satisfy this requirement. For example, all the nodes whose IDs start with the bit "1" in an n-dimensional hypercube form an $(n-1)$-dimensional hypercube. Likewise, it is possible to use any other of the n bit positions to partition the hypercube into two halves which are hypercubes smaller by one dimension.

It is relatively straightforward to satisfy requests for the allocation of a subcube. If a free subcube of the requested size exists, it is allocated. If there is no free subcube of the requested size, but there exists a larger free subcube, then that subcube is successively divided in two until a subcube of the desired size is obtained.

It is harder to re-unite subcubes after an application terminates. Needless to say, not any arbitrary set of 2^{n-1} nodes in an n-dimensional hypercube form a subcube. Thus if two jobs terminate after being executed on disjoint subcubes, we need to be able to recognize whether or not these subcubes can be joined into a larger subcube. Therefore the issue of subcube recognition has received much attention in the literature.

The simplest method is based on the natural mapping between a hypercube and a buddy system[4], where dimensions of the cube correspond to levels of the tree [61, 99, 79, 23] (Fig. 6). This scheme is used in the nCUBE system. Implementations can use a bit vector or more sophisticated data structures based on the buddy system tree [203]. At each level, only one bit position is considered for the partitioning, so there is only one way in which subcubes can be united. In the example of Fig. 6, the recognized 2-D subcubes are marked with arrows. These are the horizontal 2-D subcubes of the 4-D hypercube on the left. Other 2-D subcubes are not recognized.

Many schemes that offer improved recognition have been proposed, including the use of Gray codes [61, 62], free lists [187, 81], or a lattice structure. A lattice is a partially ordered

[3]If the application does not require a subcube, it is still convenient to allocate a subcube, and then possibly to reclaim the leftover nodes [345].

[4]In a buddy system, a set of resources is allocated in blocks whose sizes are defined by a recurrence equation. Here, we consider the binary buddy system, where $B_i = 2 \times B_{i-1}$. This was originally proposed for fast allocation of contiguous memory segments [192].

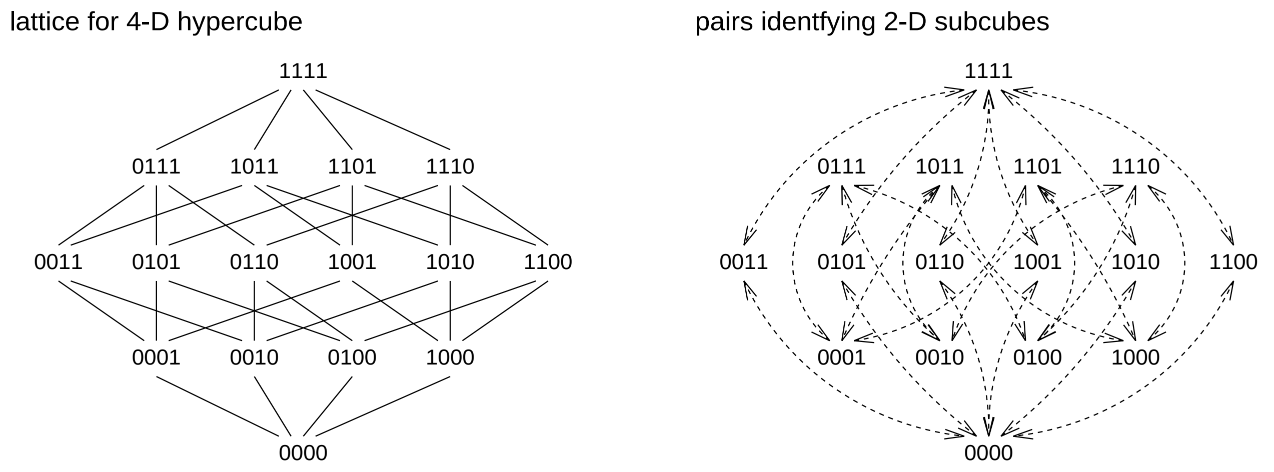


Figure 7: *Subcube identification using a lattice structure.*

set where every two elements have a unique least upper bound and a unique greatest lower bound [176]. PE IDs in a hypercube form a lattice if we use the following order relation: the bit string $x_1 x_2 \ldots x_n$ is smaller than or equal to the bit string $y_1 y_2 \ldots y_n$ if $x_i \leq y_i$ for every $1 \leq i \leq n$. Using this structure, every two comparable elements in the lattice define a subcube comprising all the elements that are smaller than or equal to one of them, but larger than or equal to the other. The left part of Fig. 7 shows the lattice corresponding to the 4-D hypercube. The right part uses arrows to indicate the 24 pairs of elements which have two other elements smaller than one, but larger than the other. Such pairs define 2-D subcubes, thus giving full recognition of all such subcubes in the original 4-D hypercube.

3.2.2 Flexible Partition Sizes

The architectural considerations that motivate the use of combinations of predefined partition sizes are mainly based on the topology of the interconnection network. But with modern multicomputers, the importance of topology is decreasing. This is largely due to improved hardware routing mechanisms, that support an abstraction of a fully connected network with uniform distances among all pairs of PEs [307, 267] (or an actual implementation of all-to-all communication, as in the VPP500 [248, 347]). This allows arbitrary partitions to be made, including partitions composed of PEs attached to arbitrary ports of the interconnection network. Among other things, this feature is important because it allows faulty processors to be configured out of the system without invalidating a whole partition.

Submesh allocation

An interesting special case is mesh-based machines, where allocations may be required to be rectangular sub-meshes of various dimensions (similar algorithms can be used for tori [287]). In this case the question is finding a free submesh of the required size.

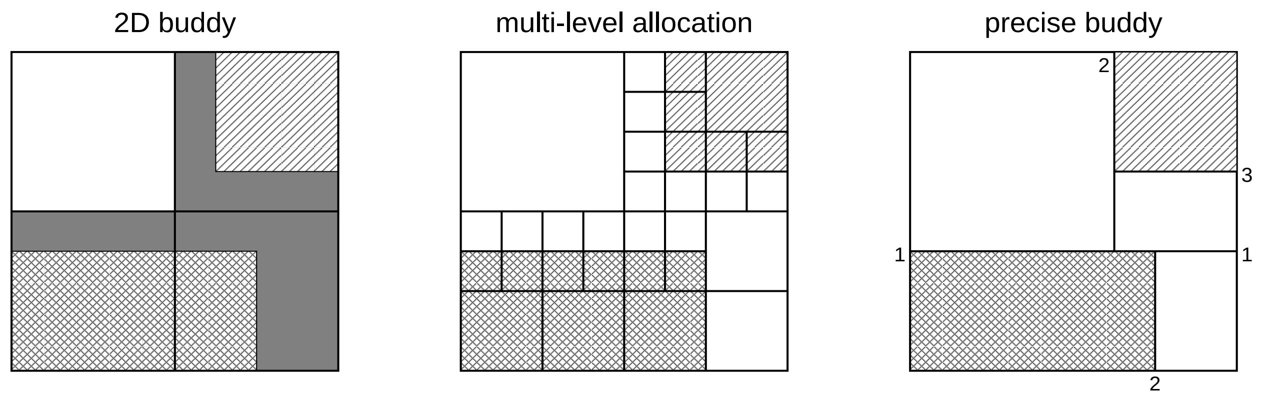


Figure 8: *Allocations for 6×3 and 3×3 jobs in an 8×8 mesh, using variants of the buddy system approach. Fragmentation is shown in gray.*

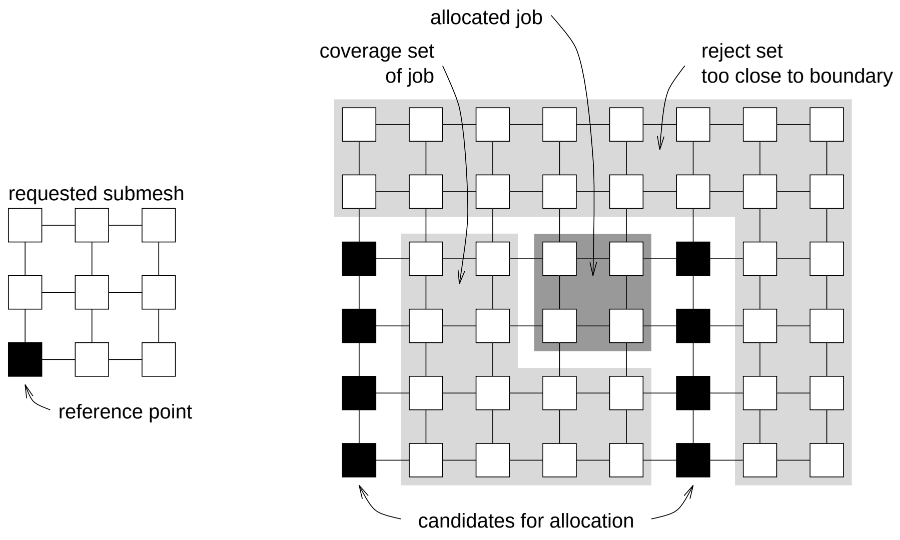


Figure 9: *Submesh allocation based on a coverage set.*

A simple approach mentioned above is to use a 2-D version of the *buddy system*. However, this is limited to square systems where the side is a power of two. Moreover, allocations are also squares with sides that are powers of two, leading to significant internal fragmentation [219, 218]. These problems are solved by using the buddy system only to identify free submeshes, and allocating a number of free submeshes of different sizes to satisfy each request [224, 352]. Such a scheme is used by NQS to pack batch jobs on the Intel Paragon. The price is that the allocation is not necessarily a rectangle, and may even be non-contiguous. Another interesting modification is to use a *precise* buddy system, in which buddy sizes are not predefined powers of two, but rather they are determined by the sizes of requests [241]. An example comparing the three approaches is given in Fig. 8.

Another algorithm that finds all possible allocations is based on computing the *coverage* *sets* of all current jobs [369, 37]. The coverage set includes all those nodes that cannot be used as the origin (lower left node) for the allocation, because they are too close to an existing job. In addition, there is a *reject set* of nodes that are too close to the system boundary (for a torus, the reject set is empty). An example is shown in Fig. 9. The extent of the coverage and reject sets depends on the size of the request. Thus only nodes not in these sets need be considered as locations for the origin of a new allocation. A first-fit allocation uses the first such node found. A best-fit version checks all of them to find the one that will cause least additional fragmentation, by choosing a candidate in a corner of the smallest free space that has the most busy neighbors. If no candidates are found, the dimensions of the request can be switched (e.g. look for a 5×3 submesh rather than a 3×5 submesh) [92].

As with subcube allocation, many other algorithms have been proposed, e.g. using interval sets [254, 253] or lists of allocated jobs [82, 66].

Choosing the partition size

It is well known that adding more and more PEs suffers from diminishing returns, and might even cause a degradation in performance [210, 349, 72, 128, 199]. Considerable work has been done to find the optimal number of PEs that should be used. The unifying theme in the obtained results is that some knowledge of the behavior of the program is required, e.g. the degree of useful parallelism in it. For example, the *average parallelism* of the application can be used for the partition size [103, 229]. This has been shown to guarantee that the obtained speedup is at least half the speedup that would be obtained with the optimal partition size. Also, the efficiency is guaranteed to exceed 50%.

In general, the efficiency drops as more PEs are added, while the speedup rises to a maximum. A good target function is therefore to maximize the *product* of these two metrics. Denoting the execution time on p PEs by T_p, the speedup is $S(p) = T_1/T_p$ and the efficiency is $E(p) = T_1/(p \cdot T_p)$. Thus the optimal partition size is the p that maximizes $\frac{T_1^2}{p \cdot T_p^2}$. Using the uniprocessor execution time T_1 as the unit of time, this expression is equivalent to $\frac{S(p)}{p \cdot T_p}$. With this formulation, the objective function can be interpreted as maximizing the speedup per unit of cost, where cost is measured in PE seconds [206, 128, 139]. A similar formulation is possible using the system *power* as a metric, where power is defined as the throughput divided by the response time [189, 190].

The problem with setting the partition size according to the job's request is that there is a hidden feedback effect. If the system is lightly loaded, a job can reduce its response time by using more than the optimal number of PEs. Granted, it would not use the additional PEs efficiently, but if it doesn't use them, they will just be idle. Therefore asking for more PEs makes sense. But if the load is high, asking for too many PEs will cause the job to spend more time in the queue waiting for the PEs to be allocated. So in this case asking for less PEs may actually reduce the response time. The feedback mechanism works through the system load: requests for PEs change the load, and the load affects the manner in which requests are granted. It is hidden because the jobs do not have any information about system load. This problem is solved by the adaptive partitioning schemes described in the next section.

Mechanics of job execution

An issue that is often ignored, but which may have a large impact, is the mechanics of job execution. A relatively common approach is to break the problem into several components. Typically this includes [264]

- A system-wide *resource manager*, which is responsible for the allocation decisions.
- *Node managers* on all the nodes, to handle the local scheduling of threads and to implement the decisions of the global resource manager.
- A *job manager* or *partition manager* for each job, to represent the job to the system and communicate the job's needs to the resource manager.

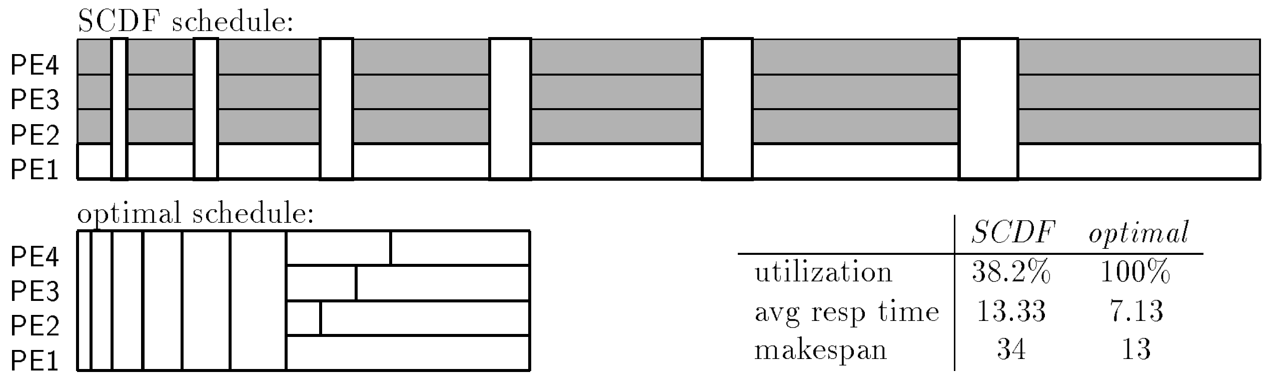
Setting up the partition, acquiring the required resources, and loading the executable may take a significant amount of time. For example, measurements on an IBM SP2 showed that this process can take from a few seconds to hundreds of seconds as the number of PEs grows from 1 to 32 [1]. In the Mach system these overheads were reduced by using hierarchical structure within each partition [244].

Tearing down a partition and terminating jobs cleanly is just as important as setting things up. In particular, it is crucial that no stray threads be left behind when the job terminates (possibly abnormally) [300]: at best, these orphaned processes pollute the system and consume resources such as entries in system tables. At worse, they keep their allocation of PEs and prevent the execution of other jobs. Practically all commercially available job management systems for parallel computers have problems in this respect [179].

3.2.3 Making Scheduling Decisions

When partitioning is used without preemption, it may be the case that submitted jobs have to wait until sufficient PEs become available for them to run. The system is then faced with the question of the order in which the queued jobs should be executed. The simplest approach is first-come-first-serve (FCFS). However, just as in uniprocessor systems, other orders could result in better performance.

The favorite heuristic in uniprocessors is "shortest job first" [195, sect. 8.3.1]. The idea is that jobs which only require the CPU for a short time be scheduled first, and then jobs that require the CPU for longer durations. The rationale is that if a short job waits for a long job, both jobs will have a long response time. But if the short job is allowed to execute first, it will have a short response time, thus reducing the overall *average* response time. However, this approach has two drawbacks. First, it requires the execution time to be known in advance, which is usually not the case (although it might be possible to estimate if the same program is executed repeatedly [87], or through compile-time analysis [301, 26]). Second, if short jobs continue to arrive, long jobs might be starved.



	SCDF	*optimal*
utilization	38.2%	100%
avg resp time	13.33	7.13
makespan	34	13

Figure 10: *A counter example showing suboptimal performance using the "smallest cumulative demand first" scheduling scheme. The workload consists of alternating sequential and parallel jobs with cumulative demands of 2, 3, 4, ... 14 units.*

In partitioned parallel systems, jobs may also be measured by the number of PEs that they require. This has the advantage that the PE requirements are known in advance. Two opposite options for using this information have been investigated: the "smallest job first" policy [230], and the "largest job first" policy [60, 218, 370]. Scheduling the smallest job first is motivated mainly by the obvious analogy with scheduling the shortest job first. However, it turns out to perform poorly, because jobs that require few PEs do not necessarily terminate quickly [217, 203]. On the contrary, they may even cause large losses owing to fragmentation.

The problem with scheduling the smallest job first is rectified by a variant called "smallest cumulative demand first" [230, 216, 313]. Under this policy jobs are ordered according to the product of the number of PEs and the expected execution time, so jobs with high priority are guaranteed to require few PEs *and* to terminate quickly. However, in a partitioned environment this policy is not much better than the original smallest job first policy [203]. A counter example is shown in Fig. 10. It also suffers from the same disadvantage of shortest job first, namely the requirement to know the execution time in advance.

Scheduling large jobs first is motivated by results in bin packing, which indicate that a simple first-fit algorithm achieves better packing if the packed items are sorted in decreasing size [69, 67]. If the item sizes divide each other and also divide the bin capacity — which is the case for jobs that require subcubes from a hypercube, for example — a perfect packing is achieved [68]. In terms of scheduling, this means that scheduling the larger jobs first may be expected to cause less fragmentation, and therefore higher resource utilization, than FCFS.

Despite the intuitive appeal of some of these scheduling policies, studies indicate that they do not necessarily perform better than a straightforward FCFS strategy in a partitioned environment [203, 202][5]. Moreover, systems using these schemes tend to saturate under lighter loads than FCFS. Using uniform requests for subcubes of different sizes in a hypercube as a concrete example, saturation may occur at loads as low as 40–50% of capacity [202].

[5]The more optimistic results in [230, 217] are due to a model in which threads are independent, kept in a global queue, and PEs are allocated singly. In such a model, there is no loss to fragmentation.

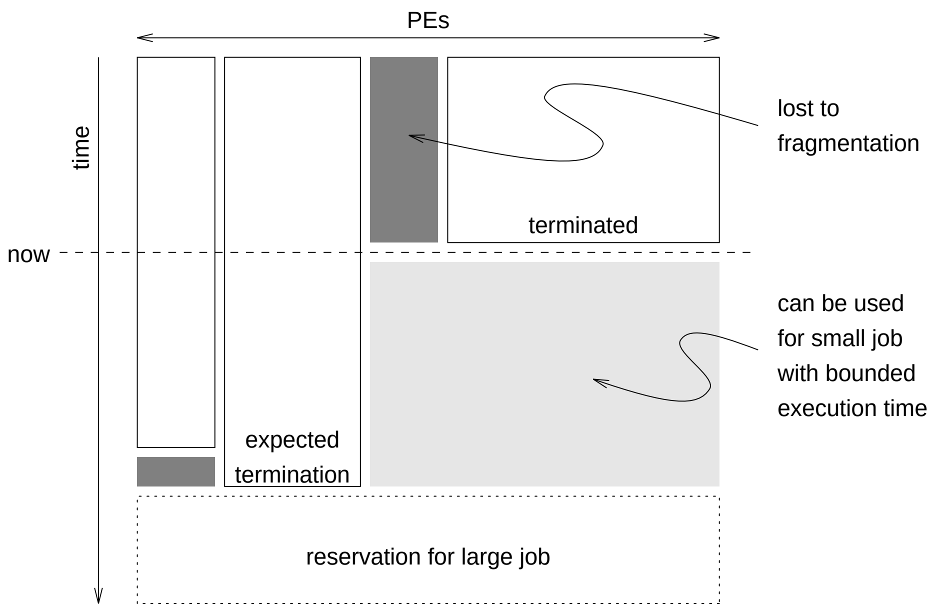

PEs
time
now
lost to fragmentation
terminated
can be used for small job with bounded execution time
expected termination
reservation for large job

Figure 11: *Runtime bounds on executing jobs allow reservations to be made for large jobs while controlling the amount of resources lost to fragmentation.*

Other promising alternatives use a distinct queue for each possible subcube size, and take specific steps to reduce the fragmentation. One approach groups jobs together before they are scheduled, rather then scheduling them on an individual basis [203, 202, 200]. The idea is to always schedule all waiting jobs of the same size. Thus when one job terminates another can immediately be scheduled in its place, and there is no fragmentation. However, this can only be applied effectively when there are multiple jobs of each size, which implies large queueing delays. Thus the opposite approach may be better: whenever a job of a given size is already running, queue any additional requests for subcubes of the same size instead of running them together [250]. Then when the running job terminates, its subcube is reused.

A similar approach is used in NQS, a network queueing system designed to support submission of batch jobs to shared parallel machines. The different queues specify different limits on the number of PEs and the execution time [196]. The limit on runtime allows the time when PEs will become available to be estimated. When jobs requiring a large number of PEs are queued, such knowledge allows PEs to be reserved for a certain time in the future. Note that such a reservation might leave PEs idle until the time that the large job is started. This loss of resources can be reduced in two ways. First, reservations can be forbidden if they lead to excessive waste [175, 80]. Second, these PEs can be used to schedule other queued jobs, provided the runtime bounds on those jobs indicate that they will terminate by the reservation time (Fig. 11). Such scheduling of small jobs in “holes” in the schedule is called *backfilling* [221].

3.3 Adaptive Partitioning: Setting the Allocation at Load Time

Many parallel applications are written so as to be executable on different numbers of PEs. But this does not mean that the number of PEs can change *during execution*. Systems which partition the PEs adaptively according to requirements and load may thus be divided into two types. The first is systems which allocate a certain number of PEs to a job when it is loaded, and guarantee that it will have that number of PEs whenever it executes. Such systems provide a more convenient environment for application writers, because they isolate the application from the fluctuating load conditions in the system. These systems are called *adaptive*, and jobs that fit this model are called *moldable*; they are reviewed in this section. The other type is systems that propagate load fluctuations to the applications, and require the applications to adapt to changing resource allocations in response to changing loads. Such systems are called *dynamic*, and the jobs are called *malleable*; they and are reviewed in Section 3.4.

As noted, the motivation for adaptive partitioning is the added convenience for programmers. For example, the application may divide the work among the available PEs at the outset, and then use barrier synchronization points implemented by busy waiting to coordinate the computation. If the application is executed on a different number of PEs, it will partition the work accordingly and perform flawlessly. But if a PE is reclaimed by the system during execution, the others will spin forever at the next synchronization point [46]. The assumption that the number of PEs does not change may even be embedded in the programming environment. For example, the HPF programming model assumes that the number of PEs is not necessarily known at compilation, but is fixed throughout any given execution [227]. A set of intrinsic functions are provided to inquire about the number of PEs and their arrangement. These intrinsics can also be used directly in array declarations. Similar assumptions are made in many message-passing systems, including EUI, p4, and PARMACS.

Setting the partition size by the system

Given that the system is capable of allocating a partition of the machine to each job, the question remains of what size partition the job should get. The simple answer of giving each job whatever it requests only applies if the sum of requests from all the jobs is less than the total number of PEs. If the sum is larger than the number of PEs, jobs are required to make do with less [139, 366, 313]. In the extreme case, each job may be reduced to only one PE. While this obviously increases the actual run time relative to the case where each job uses as many PEs as it wants, it also guarantees that short jobs do not have to wait for long ones.

By necessity, adaptive partitioning uses an on-line algorithm to determine the partition size that will be allocated to a new job. As the future is not known, a difficult choice has to be made. One option is to always keep some PEs on the side, in anticipation of additional arrivals [296]. If this is done, then resources are explicitly wasted by the system, limiting the achievable utilization. The other option is to allocate all the PEs if there is sufficient demand. This suffers the consequence that additional arrivals will have to be queued for arbitrarily long times, until one of the current jobs terminates.

A promising policy is *equipartition*, which strives for equal-sized partitions for all current jobs (except that each job's request is an upper bound on how many PEs it gets). This allows all the jobs to run simultaneously, so they don't have to wait in the queue. The jobs' response times then reflects their computation requirements more directly than if they have to wait arbitrarily long for other jobs to terminate. In addition, PEs are not given to jobs that do not utilize them well, so throughput is also served.

Equipartition can be interpreted as a spatial analogue of the well known processor sharing ideal, leading to good overall performance [217, 216, 63, 84]. In fact, equipartition itself is an ideal, that is only approximated by dynamic partitioning. In the context of adaptive partitioning, partition sizes cannot be changed after the initial allocation, even if new jobs continue to arrive. The algorithm for equipartition in this framework is as follows [309, 363]. As each job arrives, there either are free PEs available or not. If there are, the job is allocated PEs according to its request (or all of them, if less are available). If there are no free PEs, the job is queued. When a job terminates, its PEs are divided equally among all the jobs in the queue at that moment. If there are more jobs than freed PEs, some jobs will be left in the queue.

Another algorithm is based on a state machine. At each instant the system is in a certain state, which identifies the ideal partition size [295]. For a system with P PEs, the states are $\{P, \frac{P}{2}, \frac{P}{3}, ..., 1\}$. Jobs are allocated partitions whose size is determined by the state. The transition from one state to another is governed by the load on the system, as measured by the queue length. If the queue length is approximately constant and equal to the number of partitions in this state, then no transition takes place. If the queue grows larger, a transition to a state with more numerous but smaller partitions is induced. Thus more of the queued jobs can be serviced at once. If the queue becomes shorter, the transition is to a state with larger partitions. This reduces the danger of leaving idle PEs while some applications are still active.

A more sophisticated approach is to base the decision on predictions of the queueing time, based on estimates of how long current jobs will run [93]. This allows the system to weigh the two alternatives: either allocate a small number of PEs immediately, leading to a longer runtime, or wait for more PEs to become available, hopefully leading to shorter execution and a faster overall response time.

Setting the partition size in cooperation with the application

The drawback of all these algorithms is that they are oblivious of jobs' actual requirements. An alternative is to derive the best partition size in collaboration with the application. This can be done by combining information about system load with information about application characteristics. Fig. 12 shows an example for a system that combines time-slicing with partitioning. When the application is loaded, the system provides information about the expected run fraction for different partition sizes. For example, a small partition would be

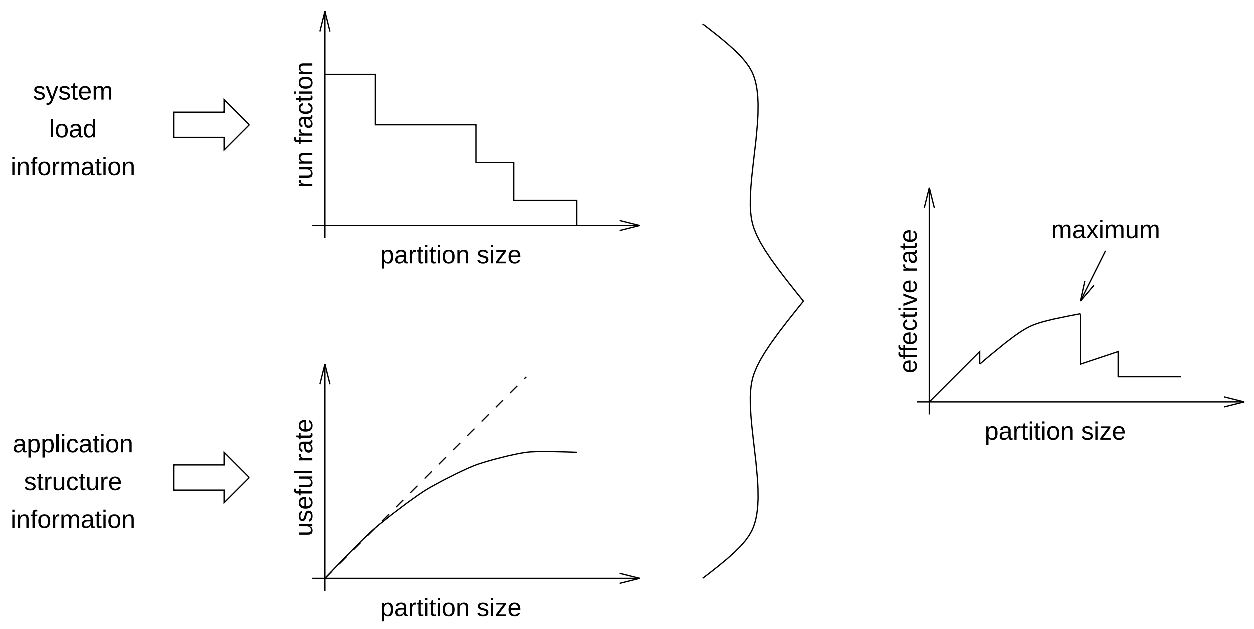


Figure 12: *Combining system load information with a model of application efficiency using different partition sizes provides the optimal size, which will maximize the effective computation rate.*

shared with one other job, leading to a run fraction of 50%. For a larger partition, the run fraction would only be 33%, and so on. The application has a model of its useful computation rate for different partition sizes. This is typically sublinear, because of added overhead when more PEs are being used. By creating a pointwise product of these two graphs, one can find the expected effective computation rate for different partition sizes [58, 20]. The size that provides the maximum effective rate is then used.

A similar idea can be applied to systems that use partitioning without time slicing. In this case, the application supplies knowledge about its *execution signature*, i.e. how much time it would require on different partition sizes [272, 229]. The system combines this with the load information to derive the optimal partition size, that would lead to the best throughput. A simpler scheme uses the *program shape*, which is the cumulative distribution of the parallelism profile (i.e., what fraction of the time does the program utilize each number of PEs) [314, 207]. This allows the minimal, average, and maximal parallelism to be found. In highly loaded systems, the job would get the minimal number, which are then guaranteed to be kept busy. In an unloaded system, the job should get up to the maximal number if there is nothing else to do with them. In general, information about the total work and the efficiency of the job is beneficial [44].

Memory considerations

All the above algorithms, whether based on application characteristics or on system load, share one major oversight: they ignore memory requirements. In shared memory systems, this can lead to overallocation of global memory. In distributed memory machines, an application can be assigned to a set of PEs that do not have sufficient memory for its data. This implies that virtual memory support is required (or else, programmers need to use overlays). The problem is that the overhead for paging can negate any benefits of running the programs side by side [278, 310]. The obvious remedy is to consider memory requirements when allocating PEs. This applies both to primary memory and to secondary storage.

Another more subtle point is that the number of processors allocated obviously affects the runtime of the job. Assuming virtual memory is not used, the runtime is equal to the memory residence time. Therefore allocating less processors under load conditions leads to longer memory residence and higher memory pressure [274]. Therefore, especially if large jobs have good speedup characteristics, it is better to assign large jobs more processors [273].

3.4 Dynamic Partitioning: Allocation May Change at Runtime

While the guarantee of a non-changing number of PEs may be convenient for application writers, it might exacerbate the problem of fragmentation and compromise fairness. Changing the partition size at run time allows the system to respond to load changes, both in terms of new and terminated jobs and in terms of changing requirements of a running job [342, 366, 229, 238, 53]. For example, when an application enters a sequential phase of computation, its PEs can be reassigned to another application [366, 348, 365]. When it enters a parallel phase again, enough PEs will be re-assigned to it to ensure efficient progress. In particular, newly arrived jobs can run immediately without being queued, unless the total number of jobs is larger than the number of PEs in the system [216]. The concept of dynamic partitioning is also applicable to running parallel programs on networks of workstations, where workstations become available at unpredictable times and are reclaimed by their owners at unpredictable times [113, 52, 286].

An important pre-requisite for PE re-allocation to work is that there be no topological constraints. It must be possible to create partitions from arbitrary subsets of PEs, without any performance ramifications. Thus this approach is mainly suitable for architectures such as bus-based or crossbar systems with a uniform-access shared memory. It is obviously inapplicable to SIMD architectures. Another consequence of the requirement for arbitrary partitioning is that the generated partitions are typically not independent: the interconnection network is shared by all the partitions, so activity in one partition may effect the performance of another.

Actual implementations of dynamic partitioning typically rely on a centralized server that maintains load information. For example, the server may be activated only when the system load changes; it then redistributes the PEs as appropriate [366, 238]. This ensures that each job always executes on the assigned number of PEs. However, it may cause thrashing if the load changes frequently.

Programming model

The fact that the partition size may change at runtime has a profound effect on how the application is structured. In particular, the application must accept whatever number of PEs the operating system allocates it, and must be ready to accept changes in this allocation (i.e., it must be malleable). Practically the only programming model that meets this requirement is the *workpile* model. In this model the application is structured as multiple independent chores, which are kept in a workpile. A number of worker threads running on distinct PEs pick chores from the workpile in an undefined order, and execute them. Additional workers may be added at any time, and existing workers may be removed whenever they finish one chore and before they start another [57, 53]. Such changes in the number of workers may change the order in which chores are computed and the rate in which they are completed, but this does not affect the outcome of the computation.

On the other hand, using the workpile model presents two problems. First, the shared data structures needed to distribute chores to workers may become a serial bottleneck. Second, there is the question of handling the preemption of a PE that is in the middle of a chore. Some systems enhance the user interface in order to allow the user-level software to deal with such preemption. This includes upcalls that notify the application of certain events that might effect its scheduling decisions, such as blocking and unblocking in kernel calls, timer expiration, or program faults. Examples include scheduler activations [13, 32] and the Psyche system [305, 234]. The problem with this approach is that it violates the layered design of the system by creating a tight coupling between the kernel and the user-level runtime system [335, p. 184]. This adds complexity to the implementation and reduces portability.

Finally, it should be noted that using a workpile model is not a pre-requisite for using dynamic partitioning. Adapting to a dynamically changing number of PEs is possible in other programming models as well, but it requires significant effort on the side of the program developer. For example, jobs written for distributed-memory machines can adjust to a changing number of PEs by redistributing their data structures [260, 97, 239]. As this involves considerable overhead, it is imperative that the frequency of reconfigurations be kept low in practice: otherwise the price of reconfiguration might even outweigh the benefits of changing the PE allocation [272, 97, 327]. Alternatively, a model where reconfigurations are only allowed at certain points in the application can be used [365, 364, 109, 315]. These points are chosen such that repartitioning is significantly easier than at other points in the computation, e.g. at the beginning of a new parallel loop[6].

[6]Acquisition of PEs at the beginning of parallel loops is common practice on Cray vector multiprocessors. This is typically done using self-scheduling [132].

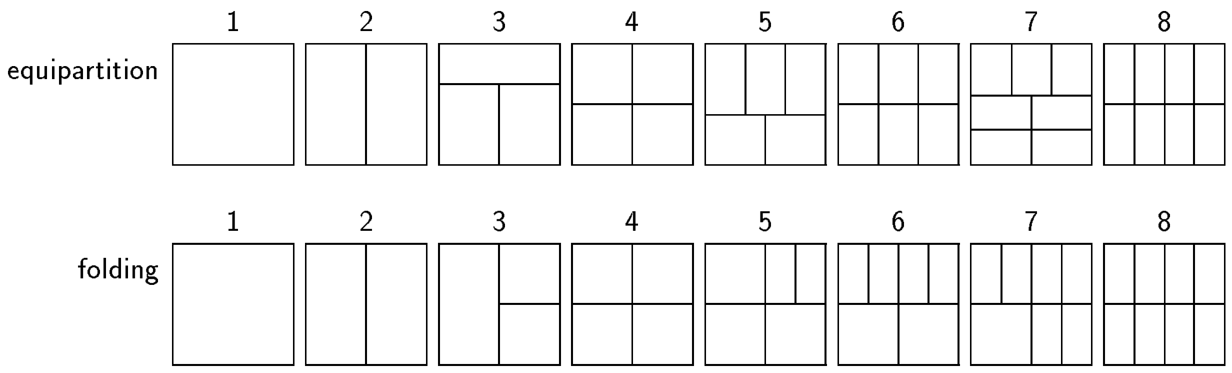


Figure 13: *Conceptual comparison of equipartitioning and folding. In reality, equipartitioning will create non-contiguous partitions so as to reduce the number of PE preemptions.*

Setting the partition sizes

The main reason to change the partition sizes at runtime is a desire to improve fairness on the one hand, and resource utilization on the other. Thus when a new job arrives, a fair share of processors should be preempted from jobs already in the system, and given to the new job. When a job terminates, its processors should be divided among the other jobs. In essence, this is the equipartition policy (Fig. 13) [342, 217, 238, 63, 274, 84].

However, it is not always best to strive for an equal partitioning of the processors. Some applications may have an upper limit on the number of processors they can use effectively. Therefore the allocation must take program requirements into account. In addition, different classes of jobs may require different qualities of service: it is perfectly OK to preempt PEs from a batch job in favor of an interactive one, but it is undesirable to do so the other way round [260, 19]. A possible mechanism to guarantee a certain level of service is to prescribe a minimal partition size, and possibly different sizes for different job classes [259].

Efficiency can also become an issue. Consider a job that is written so that the work is divided into 8 equal pieces (e.g. 8 chores). Running such a job on a partition of 7 PEs would not accrue any benefits over a partition of only 4 PEs, leading to waste of resources [239]. A possible solution is to use a *folding* policy rather than an equipartition policy (Fig. 13). This policy assumes that the total number of PEs is a power of two, and that all applications are designed to execute optimally on the full machine. When a new job arrives, the largest current partition is folded in half, thus freeing half of its PEs for the new job. The result is that all jobs in the system have partition sizes that are either of two adjacent powers of two. Such partition sizes provide the same load balancing properties and communication locality as the full machine, so jobs can be expected to achieve high efficiency [239]. This approach is applicable to hypercubes, where jobs typically require a subcube in order to run, and therefore cannot use the equipartitioning scheme. However, this would not help if the optimal partition sizes for different jobs are not multiples of each other, e.g. not all powers of two.

3.5 Processors Allocated Singly

Dynamically changing partitions can also be created by the individual allocation of single PEs. The fact that a number of PEs end up executing threads belonging to the same application, and thus implicitly defining a partition allocated to that application, is then an artifact of the scheduling mechanism.

The main advantage of this approach is that there is no fragmentation: any PE can be allocated to any job, and jobs never get more PEs than they actually need. This leads to high utilization and performance [230, 217]. The model of computation is typically that of multiple independent threads, that do not interact with each other. Otherwise, there is danger of deadlock if multiple jobs request additional PEs while hanging on to what they already have.

In fact, some versions of dynamic partitioning operate in this manner. Instead of explicitly allocating PEs to jobs, the jobs create threads that then execute on available PEs. If there are too many threads in the system, the jobs voluntarily reduce the number of threads. If there are free PEs, the jobs create new threads to utilize them. Examples of this approach include process control [342] and ASAT (Automatic Self-Allocating Threads) [315].

A more explicit form of single PE allocation has been taken in a couple of coarse-grain systems, which are designed for a distributed programming style more than for a parallel one. The scheduling mechanism is simply a global queue of independent threads. The difference between this and the global queue mechanisms described in Section 4.2 is that here the threads are run to completion, without preemption. Systems the use this approach include SpoC [258], Amoeba [256, 336], and plan 9 from AT&T Bell Labs [282].

4 Independent PEs

The second major class of multiprogramming mechanisms is time slicing on independent PEs. The fact that the PEs are independent implies that they handle the threads on an individual basis. As far as each PE is concerned, there is no grouping of threads. However, such grouping can exist at the system level. For example, the operating system may decouple the question of processor allocation from that of scheduling, by creating a partition of PEs that only know of threads from one specific application. Note also that the individual handling of threads means that dynamic thread creation is handled as well, and does not require special support.

The mechanisms described in this section are designed to deal with a many-to-few mapping of threads to PEs. This covers both the case of PEs that service threads from different applications, typical of single-level scheduling, and PEs that only know of threads from one application, as would happen in a two-level scheme. Of course, if it so happens that the load on the system is low, the relation may degenerate to a one-to-one mapping. However, such a mapping is not a goal or part of the design of independently scheduled PEs.

Given the many-to-few relation of threads to PEs, the question is whether to map threads to PEs and then use a local queue on each one, or to have all the PEs share a single global queue. The choice also depends on the architecture, and specifically on the availability of shared memory. Without shared memory, a global queue is hard to implement efficiently. The body of this section describes and compares these two options.

4.1 Local Queues

Using a local run queue on each PE is natural for distributed memory machines. It is also suitable for shared memory machines provided there is some local memory as well. Given that only one PE uses each queue, they suffer from no contention and need no locks [14, 233]. However, when using local queues, a separate distribution mechanism is required to map threads to PEs.

With local queues, each PE is exactly like a uniprocessor. Thus this is the natural choice for the Unix compatible OSF/1 AD system. Other example systems that use this approach include distributed memory systems like PEACE on SUPRENUM, NX/2 on the iPSC/2, and SIMPLEX on the nCUBE. The Sylvan system went so far as to add a dedicated kernel-processor to each node; one of the responsibilities of this processor was to perform local scheduling of the node's user-processor and vector unit. Local queues have also been used in shared memory systems, such as StarOS on CM*, Chrysalis and Psyche on the BBN butterfly, EMBOS on Elxsi systems, the Fujitsu AP1000 array processor, PUMA, and the EMMA2 system. MEMSY, which has memory modules shared by pairs of PEs, also uses local queues.

Some architectures support a local queue and context switching in hardware. A prime example is the transputer, which is a building block of many parallel systems [163, 340, chap. 5]. Each transputer is capable of implementing the programming model of the Occam language, including multiple parallel threads. This is done by efficient context switching implemented in hardware [357, 237]. Operating systems for such machines make use of this feature and support multitasking on each node (e.g. Trillium [50] and Helios [277, 159]).

Other architectures employ multithreaded processors, that essentially perform context switching in hardware on every instruction. In fact, the first commercial multiprocessor, the Denelcor HEP, was based on this design [325, 180, 194], and a very similar architecture is used in the more recent Tera system [11, 9, 10]. The MIT Alewife takes a slightly different approach: threads are switched only upon a remote access or a synchronization failure, not on every instruction [2, 3]. In all these architectures the role of the operating system is limited to mapping multiple threads to each PE.

A third class of systems with hardware scheduling that amounts to local queues is represented by the MIT J Machine. This is a message-driven architecture, where threads are created and scheduled in response to incoming messages [75, 74]. In effect, this architecture uses the hardware-maintained queue of arriving messages as the ready queue.

The use of local queues implies that threads are mapped to PEs. This provides a sense of locality and context. Threads can keep data in local memory, thereby supporting programming paradigms that exploit locality. For example, a very useful programming paradigm is to partition the problem data set, and perform the same processing on the different parts, with some communication for coordination along the boundaries [307]. This is used in the prevalent SPMD and dataparallel programming styles, the difference being whether the partitioning and coordination are done explicitly by the programmer or implicitly by the compiler.

The main consideration involved in mapping threads to PEs is the requirement to balance the loads. This is largely based on data from distributed systems, where is was shown that load balancing is crucial for adequate performance [101, 318]. The typical measure of load, which is the length of the run queue, is also based on research in distributed systems [48, 208][7]. There is some controversy in the literature on whether balanced placement of new threads is enough [102], or maybe migration of active threads is required as well [204, 83]. This is an important issue because migration suffers more overhead, owing to the larger context at runtime, whereas initial balancing fails to adapt to changing conditions when threads terminate. Initial mapping of threads is reviewed in Section 4.1.1, and load balancing by migrating threads is reviewed in Section 4.1.2.

4.1.1 Mapping

The initial placement of threads is often referred to as *mapping*, with the implication that threads are mapped to PEs and do not migrate to other PEs later. The question is how to map threads to PEs.

One simple approach that has received some attention is to use random mapping: for each new thread, choose some PE at random and map the thread to that PE [188, 21]. Obviously, this scheme has a bad worst-case behavior, because it is possible that the most highly loaded PE be chosen [101, 165]. This can be overcome by using a two-step approach: first choose some PE at random, and then map the thread to the least loaded of its immediate neighbors [151]. An alternative approach is to first probe a limited number of nodes at random, and then choose the best one [247]. Even probing only two nodes reduces the expected maximum load when mapping n threads from $O(\log n/\log\log n)$ to $O(\log\log n)$ [22].

Mapping can also be done based on load information to begin with, skipping the initial random phase. The problem is how to maintain global load information. While some schemes for maintaining such information have been devised (e.g. [167]), it is generally felt that this approach will not scale well to large systems. A possible exception is special cases where dissemination of load information and mapping are both done in hardware. This approach has been taken in the Flagship graph-reduction machine. The PEs in Flagship are connected by a Delta multistage network, which is used to propagate load information [334, 355, 356]. At each network switch, the lowest load is propagated. Thus all the PEs know the load on the least-loaded PE in the machine. When a PE generates a new graph node for reduction, it compares its own load with the global minimum. If the local load is higher than the

[7]More recently, with the advent of heterogeneous systems, the capacity of the different PEs is also becoming an issue [172, 368].

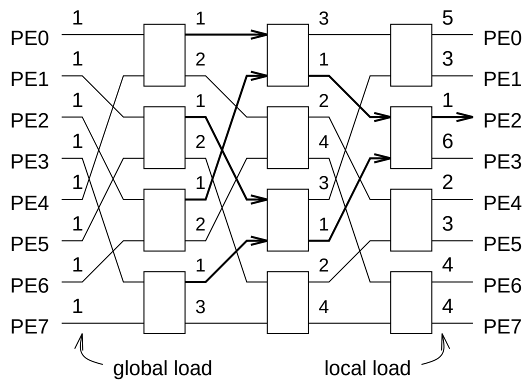


Figure 14: *Dissemination of load information and automatic mapping in Flagship.*

minimum by a sufficient amount, the new graph node is injected into the network. It is then forwarded automatically to the least loaded PE (Fig. 14).

Other approaches are to distribute the load using self scheduling, and to use economic models. With self scheduling the load is distributed using a global queue, and then local queues are used to actually perform the computation. The acquisition of new work can be done at a rate that is inversely proportional to the local load, so as to improve load balancing, as described in Section 4.3 below. With economic models, PEs communicate with each other to create a market of work to be done and available resources, and matching is done based on supply and demand [125, 231, 350, 133].

It is interesting to note that all the above approaches refer to the mapping of single threads. There has been a limited amount of work on mapping parallel jobs as a whole to different parts of the system. A design that has been suggested a number of times is to partition the system using a buddy-system structure. This creates a hierarchy of control points: the root controls the whole system, its two children each control half of the system, the grandchildren control quarters, and so on. This control structure is used to maintain data about load on different parts of the system. When a request to map a new job arrives, it is handled by the controllers at the level that has enough PEs for the job's size. This approach is used in DHC [120, 121, 130], in DQT [169, 168], in NETRA [64], and for hypercubes [79, 23, 6]. Actually, the same structure has also been proposed for maintaining and distributing the load information required to map single threads as well, in AMPS [186] and in APERM-2 [165].

Mapping based on load conditions is appropriate if threads just do local work. But if threads communicate with each other, the communication pattern induced by the mapping should also be taken into account. Note that the quest to reduce communication requires knowledge about the interactions among the threads. The PARMACS message passing library and MPI allow such interactions to be expressed via the virtual process topology [51, 351]. HPF, Dataparallel C, and Pandore only support a more restrictive multidimensional mesh topology [227, 157, 15]. Alternatively, the system can make an educated guess about communication patterns; for example, it is reasonable to assume that when one thread spawns another thread, the two will communicate [34].

4.1.2 Load balancing

The second major issue involved in the use of local queues is load balancing, by means of migrating threads after they have commenced execution. The importance of load balancing depends in the degree of imbalance, and hence on the mapping: if all new threads are placed on the same PE, for example, load balancing becomes very important [181]. Most of the literature on this subject relates to distributed systems, not parallel ones (see [354, 318] for surveys). The difference is that in parallel systems the connectivity is much greater, which allows more state information to be collected and more context to be moved. Also, parallel systems sometimes make explicit statements about mapping and load distribution, whereas in distributed systems processes are typically created locally and one at a time. Finally, in parallel systems the threads are expected to communicate, so when they are migrated messages have to be forwarded and all the routing framework has to be updated [304, 270]. This added complexity also reduces the appeal of using load balancing.

Important features of a load balancing algorithm are that it distribute load quickly and that it be stable [48, 55, 329]. Quick load distribution is achieved by non-local communication to disseminate information about load conditions [222]. Stability means that the algorithm should not over-react to every small change; it is achieved by balancing only if the difference in loads is above a certain threshold [265], or by limiting the number of times that each specific thread can migrate. In addition, the technicalities of process migration are important, including how to avoid leaving any state on the original PE, and how to reduce the period during which the process is suspended [293, 83].

The two algorithmic aspects of load balancing are choosing which thread to migrate, and choosing where to migrate it. The chosen thread should have enough computation left to justify the overhead of migration. Information regarding the distribution of thread lifetimes is therefore useful, and indicates that long-lived threads have a higher probability to continue execution longer [155]. A simple alternative is to perform enough computation locally to cover the costs of migration, and then migrating one thread at a time [4].

Two schemes for choosing the target for migration are diffusion and the gradient model. Diffusion is based on an analogy between work and substance: at each load-balancing step, work passes between every pair of connected PEs according to the difference in their loads [73, 41, 360]. The analogy is the diffusion of molecules across membranes, which is driven by the difference in concentrations. This scheme is used in MuNet [154] and Alfalfa [141]. The effectiveness of diffusion depends on the network bandwidth, and it is therefore inefficient for low-dimensionality meshes [332]. A special version for hypercubes, called "dimension exchange", iterates on the dimensions of the hypercube and balances the load among neighbors in one dimension at a time [73, 94]. This was shown to be more efficient than diffusion with multiple neighbors at once [73, 358], especially if communication is limited to one port at a time [361].

The gradient model is more directly goal-oriented. The system maintains a "pressure surface" representing the load. The gradient of this surface points to the nearest PE with a local minima of load [222]. For good performance, threads should propagate a number of hops along this gradient [319]. However, there is a danger that idle nodes will be flooded with work from all sides, so a reservation protocol is recommended [257]. This scheme is used in Rediflow [185].

Maintaining the information required for load balancing can be expensive. In addition, the activity of load balancing in itself is overhead that comes at the expense of useful computation. A nice solution that includes both receiver and sender-initiated balancing is to perform the balancing by a thread that competes with other threads. On PEs with low load this thread will be scheduled often, so these PEs will spend much of their time trying to find more work. On highly loaded PEs the load balancing thread will seldom be scheduled, allowing them to concentrate on useful work [298]. The pairing for balancing in this scheme is random. This reduces the cost of communication to maintain the required load information, and achieves good results with high probability.

Other schemes that use randomization to reduce the cost of collecting load information have also been proposed [316, 197]. One of them is used in the MOSIX system [28, 29, 27]. Each node maintains a short load vector, say with 4 entries. The first is the local load. At certain intervals, it sends the first two entries to a randomly chosen other node. When it happens to receive such a load message, the new data is integrated into the load vector and the oldest data discarded. The data in the load vector is used to gauge the average load in the system, and to select a target for process migration if the local node finds that it is overloaded. This scheme has been adopted for use in OSF/1 AD [367][8], and consequently in systems that are based on it, such as the Intel Paragon operating system.

4.2 A Global Queue

A global queue is easy to implement in shared memory machines. It is not a realistic option for distributed memory machines. In any case, in this section we deal with preemptive systems: threads are picked from the global queue, executed for some time, and then returned to the queue.

The main advantage of using a global queue is that it provides automatic *load sharing*[9]. In effect, the workload is distributed evenly across all the PEs. The disadvantages are possible contention for the queue [263] and lack of memory locality. With a global queue threads typically run on a different PE each time they are scheduled, so local memory cannot be used as effectively [294, 232, 233, 57]. This includes use of the local cache on each PE.

It might seem possible to improve the performance of a global queue by removing a

[8]It is part of the Unix personality layer built above the Mach kernel. Thus the entities that are balanced are full Unix processes, not Mach threads.

[9]The term "load sharing" originated in distributed systems, where it describes systems that do not allow a PE to be idle if there is work waiting at another PE [101, 205]. The term "load balancing" is only meaningful when PEs have individual loads, as when local queues are used.

number of threads at once each time the queue is accessed, thus amortizing the cost of contention and overhead [284, 263, 173]. However, this is much more relevant to second-level scheduling by the application, after PEs have been allocated, than to scheduling from a global queue by the operating system. The reason is that an application knows that all the threads belong to the same application, and knows the number of PEs devoted to executing them. An operating system does not have such knowledge, and might accidentally schedule all the threads of a certain application to the same PE, causing it to loose the possible benefits of parallelism.

4.2.1 Implementations

A global queue is a shared data structure, so access to the queue should be done with care. The main methods used are locking and using wait-free primitives, as described below. There has also been some work on fault tolerance, i.e. ways to guarantee that each element in the queue be taken exactly once [362].

Implementation with locking

The naive implementation of a global queue uses locks to protect the shared queue, and allow different PEs to add and delete threads. For example, it has been observed that adding locks to data structures is the main thing that needs to be done to parallelize the Unix kernel [24]. A lockable shared queue is used in the Hydra system on C.mmp, in the Dynix system on the Sequent Balance, in Mach, in IRIX on Silicon Graphics multiprocessor workstations, and in the DASH system which is based on IRIX. It is also used in thread packages such as Presto, and in microtasking on multiprocessor Cray supercomputers.

Much progress has been made lately regarding the efficient implementation of locks, especially by way of exploiting local caches with hardware coherence [297, 12, 143, 149, 242]. The main idea behind these implementations is that it is possible for each PE to busy-wait on a variable in its own cache, thereby avoiding any extra load on the communication network. But the variable in the cache represents a variable in the shared memory, so when a PE releases the lock, all it has to do is to touch the shared variable. The cache coherence mechanism propagates this effect to the waiting PEs.

The queue itself is basically the same as in uniprocessors. This typically implies that entries are sorted by priorities, and multi-level feedback is used rather than maintaining strict FIFO semantics. For example, the Mach system structures the queue as an array of 32 queues, corresponding to the 32 possible priority levels [39] (Fig. 15). Suggestions have also been made to use a heap structure to maintain the queue. Such a structure allows concurrent access by a number of PEs, that each need to lock only part of the queue [178, 291, 85, 153, 78].

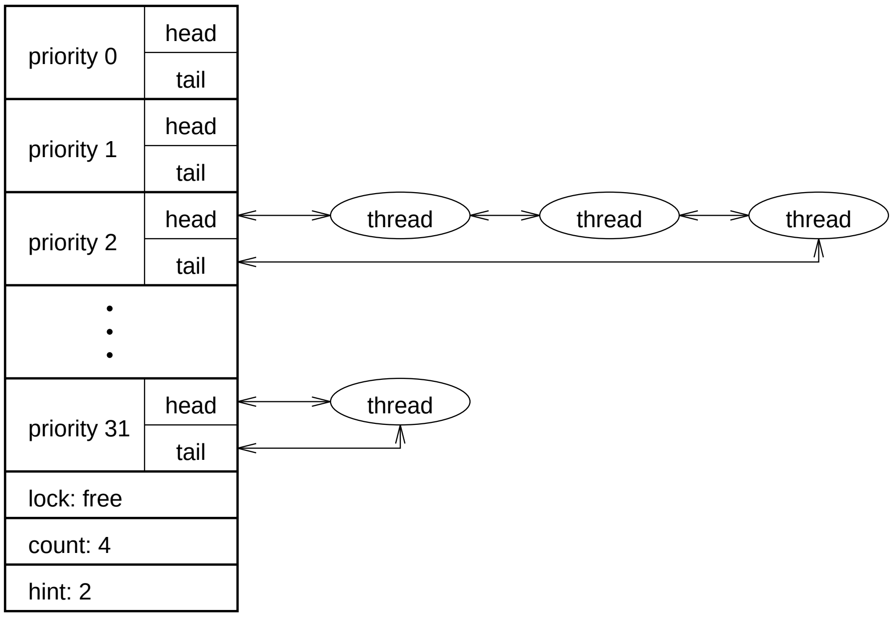


Figure 15: *Priority queue structure in Mach (adapted from [39]). The hint field indicates the highest priority currently in use.*

Implementation with fetch-and-add

The problem with locking the queue is that the queue becomes a serial bottleneck [38, 162]. Denote the average time that the lock is held by τ, and the scheduling time quantum by T. Each PE only holds the lock for τ/T of the time. But if there are more than T/τ PEs in the system, than the extra PEs will always be waiting for the lock. These PEs are unable to do any useful work. Therefore the solution of protecting a global queue with a lock cannot be expected to scale up to massively parallel systems with thousands of PEs. Indeed, measurements on a 4-PE machine show that already 13.7% of the attempts to access the global queue found it locked, and the trend indicated that considerable contention should be expected for larger numbers of PEs [338].

The alternative is to use a bottleneck-free implementation of queues, that does not require locking. Such an implementation, based on the fetch-and-add primitive, was a driving force in the design of the NYU Ultracomputer [147, 146, 104, 89] (actually, a fetch-and-increment operation is also enough [131]). The implementation is based on a predefined array of entries. A shared variable contains the index of the next available entry. PEs that want an entry perform a fetch-and-add on this variable, with an increment of 1. Many such operations may be performed simultaneously. By definition, the final outcome is that the variable is incremented by the total sum of the individual increments, so that it contains the index of the next free entry after all the requests are granted. Each of the PEs receives a different partial sum as a return value from the fetch-and-add. Using this return value as an index into the queue creates a unique matching of PEs to queue entries. Once a PE is uniquely matched with an entry, there is no need to lock the entry itself (but it is still necessary to synchronize the PE that inserts a certain entry with the one that deletes that same entry).

Code segments that implement queue insertion and deletion are shown in Fig. 16. The first check that the queue is not full when trying to insert, or empty when trying to delete, seems to be redundant considering the following fetch-and-add operation. However, it is required in order to avoid race conditions [148]. Note that this is not a full implementation, as it does not take care of counter overflow.

4.2.2 Making scheduling decisions

The order in which threads are scheduled from the global queue is determined by their relative priorities. Most implementations use the same algorithms that were used in uniprocessor systems. For example, the Mach scheduler implements a priority queue that is modeled after the Unix scheduler. Thread priorities are based on their recent CPU usage plus a base priority that reflects system load [39].

The scheduler in the Symunix system on the NYU Ultracomputer is also based on the Unix scheduler, with priorities calculated according to a base priority and CPU usage. The base priority is used to increase the priority of threads from nested spawns[10] relative to the priority of their parent. This ensures that nested spawns are executed in a depth-first order [105]. This order is deemed appropriate because it prevents the system from being flooded with threads in cases of large deeply-nested spawns. Giving nested threads a lower priority would result in breadth-first execution, which improves fairness among sibling threads [35].

Another consideration in the scheduling of threads from a global queue is matching threads with the most appropriate PE for them to run on. In general-purpose systems where PEs are functionally equivalent, this boils down to the question of data that may reside in the PE's cache. The scheduling policy that tries to schedule threads on the same PE on which they ran most recently, under the assumption that this particular PE might still have some relevant data in its cache, is called *affinity scheduling* [348, 86, 25, 328, 33]. Affinity scheduling becomes more important on machines where the cost of remote access is higher [57]. However, the performance improvement is typically small, as in most cases not much is left in the cache after a number of other applications have been scheduled [152, 339, 348]. It has therefore been suggested that affinity be temporally delimited: it would only be established if a thread runs on a PE for a minimal amount of time, and it would expire after a certain delay when the thread is de-scheduled [31].

A simple implementation of affinity scheduling uses local modifications to threads' priorities [339, 57]. Each thread has a global priority that depends on its accumulated runtime, like processes in a Unix system. However, when a given PE scans the queue and looks for the thread with the highest priority, it boosts the priorities of those threads that used this PE the last time they ran. The last thread to run on the PE is given a larger priority boost. This increases the effective scheduling quantum, by increasing the probability that a thread

[10]The `spawn` system call is a generalization of `fork` which creates multiple processes at once.

```
shared struct {
   int   data;
   int   turn;
   int   in_count, out_count;
} q[N];
shared int count, head, tail;

insert( data )
   if (count < N)
      if (faa(count,1) < N)
         index := faa(tail,1) mod N
         myturn := 2 * faa(q[index].in_count,1)
         wait until q[index].turn = myturn
         q[index].data := data
         q[index].turn := myturn + 1
      else
         faa(count,-1)
         fail: queue is full
   else
      fail: queue is full

delete( buffer )
   if (count > 0)
      if (faa(count,-1) > 0)
         index := faa(head,1) mod N
         myturn := 2 * faa(q[index].out_count,1) + 1
         wait until q[index].turn = myturn
         buffer := q[index].data
         q[index].turn := myturn + 1
      else
         faa(count,1)
         fail: queue is empty
   else
      fail: queue is empty
```

Figure 16: *Implementation of a shared queue using fetch-and-add.* count, head, *and* tail *are global variables for the number of used cells in the queue array, the index of the first one, and the index of the last one.* N *is the size of the array. Apart from the data, each cell has three fields (*turn, in_count, *and* out_count, *initialized to zero) to synchronize threads that deposit data into this cell with those that extract it.*

issue	*local queues*	*global queue*
suitability	distributed memory	shared memory
load distribution	requires explicit mapping and load balancing	automatic load sharing
locality	maintained	partial at best, if affinity scheduling is used
overhead	low, no contention	requires locks, and may suffer from contention

Table 2: *Comparison of local and global queues.*

be rescheduled immediately on the same PE. It does not allow the thread to monopolize the PE, because the priority is still reduced as the thread accumulates more and more runtime.

4.3 Combination of Local and Global Queues

The main features of local and global queues are summarized in Table 2. As can be seen, each has its advantages and disadvantages. The choice between the two can be avoided by using both, and trying to benefit from the good characteristics of each. For example, this is done in the MAXI system on the Makbilan research multiprocessor [116]. In this system, a global queue is used for work distribution, and then local queues are used for the actual execution[11]. The mapping of new threads is done by a system `get_work` thread that competes with local threads, so lightly loaded PEs get more new threads than PEs that already have a high load [298]. Once mapped by `get_work`, threads do not migrate. Similar schemes are used in the KSR1 and other systems [304, 252]. A variant is used on the Cedar multiprocessor, where the unit of allocation is not a single thread but rather a task with eight threads, that executes on a cluster of eight PEs [111, 110]. In all these systems, mapping is done at regular intervals, and does not adjust to the load.

A hierarchical system of queues has also been suggested in order to reduce the contention for the global queue, while maintaining a degree of load distribution [77, 76]. In this system all new threads are placed in a global queue, and each PE has a local queue. In addition, there exists a hierarchy of queues in between. When a PE's local queue is empty, it tries to get more work from its parent. If the parent queue is also empty, the request is propagated upwards. Transfers of work between queues at higher levels involve increasing numbers of threads, because they are expected to be serviced by more PEs.

[11]In principle, these queues contain threads from separate applications. In practice, the MAXI implementation was only capable of running a single application at a time, because memory management was not built into the system.

5 Gang Scheduling

In multiprogrammed parallel systems, threads come in groups belonging to different applications. Threads within a group cooperate with each other, while competing with threads in other groups. It is open to debate whether the operating system should acknowledge this distinction and do something about it. The scheduling mechanisms described in this section reflect the belief that it is important to schedule cooperating threads together. We start with a definition of gang scheduling and an exposition of the reasons for using it. Then we investigate the implementation of coordinated context switching across multiple PEs. Finally, we describe systems that combine gang scheduling with partitioning in various ways.

5.1 Definition and Motivation

The term "gang scheduling" has been used with a variety of meanings in the literature. In general, they have the common theme of giving a job multiple PEs at once. But in order to distinguish gang scheduling from other schemes, such as variable partitioning and family scheduling, a more precise definition is needed. We therefore define gang scheduling formally to be a scheduling scheme that combines these three features:

1. Threads are grouped into gangs.
2. The threads in each gang execute simultaneously on distinct PEs, using a one-to-one mapping.
3. Time slicing is used, with all the threads in a gang being preempted and rescheduled at the same time.

In many cases, all the threads in the job are considered to be a single gang. Thus the number of threads in the job conveys the PE requirements of the job.

Gang scheduling leads to several desirable properties. First, gang scheduling supports the abstraction of a dedicated machine for each job, and does not impose any restrictions on the programming model (as opposed to dynamic partitioning). This has many faces. For example, gang scheduling promotes efficient fine-grain interactions among the threads in a gang, based on the fact that they are executing simultaneously. Thus it is possible to use busy waiting for synchronization, without fear of waiting for a thread that is not running[12] [268, 122, 300]. In addition, asynchronous message passing can be used without the risk of buffer overflow, and hardware communication devices can be accessed directly in user mode without need for protection mechanisms. Moreover, a one-to-one mapping also allows threads to be associated with data structures in local memory [57]. Indeed, studies

[12] An interesting special case occurs in systems that run a full Unix on each node. In this case the operating system might interfere with a thread even if there are no competing threads on the same PE. Specifically, asynchronous interrupt handling on different PEs increases the worst-case communication latency of the application. Synchronizing these interruptions — in effect, gang scheduling among the application and the operating system — leads to improved and predictable performance [255].

that compared gang scheduling with other scheduling schemes have concluded that gang scheduling is a relatively good policy [225, 306, 217, 152, 71, 57].

It should be noted, however, that other schemes — most notably variable partitioning — provide the same support. The advantage of gang scheduling (and the second desirable property) is that it provides interactive response times for short jobs, by virtue of using preemption [117]. Just as in uniprocessor systems, periodic preemption prevents long jobs from monopolizing system resources, and guarantees that every job in the system will execute within a relatively short time. Admittedly, preemption suffers from overhead and might reduce cache performance — but, on the other hand, it might actually improve utilization by providing the system with more flexibility in resource allocation [117] (these and other arguments are discussed in Section 6). Therefore gang scheduling is an appealing policy, as reflected by the fact that it is provided on a number of commercial platforms, such as the Connection Machine CM-5, Meiko CS-2, Intel Paragon, Cray T3E, and Silicon Graphics multiprocessor workstations.

Another reason for using gang scheduling is that it allows sharing in architectures that cannot be partitioned. For example, it is impossible to provide flexible partitioning of a SIMD machine, because each partition requires an instruction decoding unit and facilities to broadcast the instructions to all the PEs. This problem is solved by time slicing these devices among the different jobs. This approach was used on the CM-2 to improve user access to the machine [182].

While our definition of gang scheduling requires that time-slicing be used, this is not necessarily the only sharing scheme. Space-slicing can be used as well. When a number of gangs are to be scheduled to run side by side, the question is how to match them so as to make the best use of the available PEs [115]. One possibility is to use a set of predefined partitions; such systems are described in Section 5.3. Alternatively, it is possible to have time-slicing across the machine, and do the partitioning on the fly for each context switch. This increases the flexibility of how the machine is shared; for example, it allows applications that require all the PEs to co-exist with applications that require small sets of PEs. Systems based on this approach are described in Section 5.4.

What is a gang?

The grouping of threads into gangs deserves further elaboration. In most cases, all the threads in the job are considered to be a single gang. For example, this approach is suitable for the SPMD style of programming. It means that the threads of each job are spread across different PEs, and either all of them are running or no threads from the job are running.

However, gangs comprising just part of the threads of a given job can also be used. This allows the number of threads in the job to exceed the number of PEs in the system. The criterion for grouping is simple: threads that interact at fine granularity should be scheduled together, and therefore they should be grouped into a gang[13]. This can be done

[13]Ousterhout used the term "process working set" [268], based on the analogy with the working set of memory pages that must be simultaneously resident in primary memory for efficient computing.

based on runtime observation of interactions among threads [119, 326]. Alternatively, it has been suggested that syntactic program structures can be used to define gangs. Relevant structures are `parfor` or `parbegin` in languages like ParC [35], `PAR` in Occam [174], and the `spawn` system call in Symunix [106]. Thus, when a set of threads are spawned together by a parallel construct, they are assumed to constitute a gang [268, 120].

Coscheduling and family scheduling

Two other terms that are often mentioned are coscheduling and family scheduling. *Coscheduling*[14] was originally defined by Ousterhout to describe systems where the operating system attempts to schedule a set of threads simultaneously on distinct PEs, as in gang scheduling, but if it cannot then it resorts to scheduling only a subset of the threads simultaneously [268]. The fraction of the gang that typically gets scheduled together is used to measure the coscheduling efficacy.

The distinction between gang scheduling and coscheduling is subtle but significant. The main difference is that gang scheduling allows guarantees about the performance to be made. This is so because applications execute in an environment that is essentially the same as a dedicated machine, except for some additional overheads. Coscheduling, on the other hand, has unknown performance implications. If the application's threads are largely independent, they can make progress even if the whole gang is not scheduled. In this case, coscheduling can be highly beneficial. But if the threads synchronize with each other at fine granularity, there is little virtue to scheduling just some of them [213, 122, 71]. Therefore it is not clear that attempting to schedule partial gangs when a full gang cannot be scheduled is worth the effort[15].

Family scheduling is a variant of gang scheduling where the number of threads is allowed to be larger than the number of PEs. Thus, the operating system is involved in two levels of time slicing: first, there is the coordinated scheduling of the job as a whole across a set of PEs, and then there is the internal scheduling of the job's threads on these PEs [47]. The difference from two-level scheduling schemes is that the whole job may be preempted (two-level scheduling typically employs non-preemptive partitioning at the job level), and both levels are done by the operating system rather than leaving the second one for the application runtime system.

Relationship with swapping

As large scale parallel applications often require large amounts of memory, it is sometimes not possible to have multiple jobs memory resident. In such cases, switching among jobs

[14]Historically, coscheduling preceded gang scheduling. It is fair to say that all the work on gang scheduling is an outgrowth of Ousterhout's original work on coscheduling.

[15]This may also depend on what part is scheduled each time. An interesting idea is that *rate equivalent* scheduling may be beneficial, i.e. that all threads get the same average service rate, even if they are not scheduled simultaneously.

implies swapping them to secondary storage [117]. Indeed, some systems use the terms "gang scheduling" and "swapping" as synonyms. Such simultaneous swapping of all the job's threads and address spaces indeed qualifies as gang scheduling, because either all the threads execute or none do. However, the higher overheads of swapping imply that this cannot be done in an interactive time scale.

An interesting observation relating to gang-scheduling-via-swapping is that it is related to checkpointing [220]. As such, it is possible to restart the job on a different set of PEs than the one used originally. Therefore swapping leads to "migratable preemption" [275], and can help in the reduction of fragmentation [209, 311].

While simultaneous swapping is a type of gang scheduling, it need not be the only one. It is possible to have both fine-grain gang scheduling among the memory-resident jobs, and coarse grain gang scheduling by means of swapping. A variant of this approach is used on the Tera Multi-Threaded Architecture, where threads are loaded into separate hardware contexts that are switched on each cycle [10] (see Section 7.2.3).

5.2 Implementing Multi-Context-Switching

The implementation of gang scheduling hinges on the *multi-context-switch* operation, i.e. the coordinated context switching across a set of PEs. This has two aspects: the coordination required to cause all PEs to context switch at the same time, and saving the distributed program state.

The synchronization of the context-switch operation is typically handled by a central controller. The controller need not be predefined: a floating controller can be used, where any PE that notices a certain condition (e.g. all threads are blocked) induces the next multi-context-switch [122]. A variant of this is used in IRIX on SGI multiprocessor workstations: the PE that selects the first member of a gang from the global queue interrupts other PEs that are running low-priority processes so that they will schedule the other gang members [31].

The controller coordinates the context switching by causing an operating system trap on all the relevant processors. The requirement on this trap is that the variability in the exact time that it occurs on the different PEs be small relative to the scheduling time quantum. One possibility is to use special hardware, as in the K2 architecture [331]. Another is to use a software broadcast interrupt [122]. A third option is to rely on synchronized clocks, that all cause interrupts on their respective PEs at the same time [130]. Once the processors are interrupted, they perform their local context switch. A number of Unix-based implementations do this using signals [145, 170] or changing the priority [166, 209].

An interesting alternative suggested for networks of workstations is to let the coordination of context switches take care of itself. This is based on an application model where relatively long phases of computation are interleaved with phases of intense communication. The crucial observation is that the first processes entering the communication phase will necessarily block, waiting for the others to catch up [98]. When the last process enters the communication phase, all the rest will have relatively high priorities due to the multi-feedback queueing mechanism typically used on workstations. Therefore they will tend to run immediately when unblocked, allowing them to complete the communication phase efficiently.

Handling communication

Saving the program state on a single PE typically involves no more than saving the CPU register values. In a parallel machine, this has to be done on all the PEs. However, the program might have additional state that is neither in memory nor in the registers, but in transit from one place to another. In general, it is necessary to save such communication state together with the computation state. The following discussion uses message-passing terminology, because such architectures are more susceptible to this problem. However, it should be interpreted to include messages generated automatically to access remote memory, especially in machines that allow multiple outstanding accesses, as with relaxed consistency models.

The problem with messages that are in transit when a context switch occurs is what to do with them when they finally arrive at their destination PE. One approach is to simply drop such messages, and re-send them the next time that the application is scheduled to run [130]. This requires the sender to buffer the message until an acknowledgement is received, because it might have to re-send it later. The message passing software must use sequence numbers, acknowledgements, and timeouts to ensure that all messages are indeed delivered in the end. It should be noted, however, that most systems have such mechanisms anyway to deal with transmission errors. Moreover, this approach has the advantage of not requiring any hardware support, so it can be implemented on any machine.

Another approach is to tag messages with a job ID. When an arriving message does not belong to the currently running job, it is handled anyway. This approach is used on the Meiko CS-2 [30] and the Intel Paragon [281]. Note that it requires access to the descheduled job's address space, and that the network adapter (or communication coprocessor) must be smart enough to generate and check the job IDs. Otherwise, user-level communication cannot be used. Security issues are resolved by using kernel-maintained page tables which are accessible by the device handling the messages.

The third approach is to drain the network as part of the context switch operation [170]. While this increases the overhead for context switching, it provides each job with exclusive access to a clean network. This eliminates any need for privileged support, and opens the door to efficient user-level communication. Security is provided by mapping the communication devices into user space, and using existing hardware protection mechanisms. This approach is used in the the K2 [331], the Connection Machine CM-5 [214], and the RWC-1 [168].

5.3 Gang Scheduling within Predefined Partitions

The simple approach is to first partition the machine into sets of disjoint PEs, and then perform gang scheduling within each partition independently of the others. Actually, partitioning is not strictly necessary, as it is possible to simply schedule *all* the PEs as one unit. This approach disregards program needs, and might lead to waste if the system has many more PEs than are needed. Therefore it may not be suitable for large systems.

Chagori on the K2 prototype and Concentrix on the Alliant FX/8 are examples of systems that schedule all the PEs as a unit. In Chagori, the motivation for simultaneous scheduling is the desire to provide applications with efficient synchronization [331], as embodied in the synchronous nearest-neighbor user channels provided in hardware by the K2 architecture [16]. This is a relatively small machine (16 PEs organized as a 4×4 torus), so the loss due to possible fragmentation is not expected to be high. The coordination of the multi-context-switching required to implement gang scheduling is done by special hardware called the Torus Synchronization Unit. This combines a hardware interrupt to start the multi-context-switch with an acknowledgement specifying that the user channels are no longer active transferring user messages. These channels can then be used by the scheduling algorithm.

The Alliant FX/8 is even smaller than the K2, with only eight PEs (called "Computation Elements") [337]. The envisioned use is to improve the performance of Fortran loops, and specifically doacross loops. Such loops have dependencies among the iterations, so they can only be parallelized to a limited degree. Even so, efficient synchronization is required in order to enforce the dependencies. These features motivate the use of a small number of PEs that execute consecutive iterations in a skewed manner [337].

Simultaneous scheduling on the Alliant PEs is also done in the Cedar research multiprocessor, which is constructed from a number of Alliant FX/8 systems connected to a shared memory via an Omega network [193]. Each Alliant machine is called a cluster, and such clusters are the unit of PE allocation. The scheduling is done at two levels: allocation of work to clusters, and scheduling on each cluster. The allocation is done from a global queue of tasks, where each task specifies the state of a whole cluster (i.e. 8 Alliant PEs). A server process executes periodically on each cluster, and if it finds a new task on the global queue, it copies the task to the local cluster [111, 110]. The Alliant scheduler executes all the tasks mapped to the cluster by context switching all eight PEs from one task to another. The final effect is of gang scheduling within partitions defined by the clusters. Of course, if a job spans multiple clusters, it is not gang scheduled across all of them, which can lead to performance degradation [261].

A mechanism for gang scheduling combined with partitioning of the machine has been implemented in the Connection Machine CM-5 [214]. The coordinated scheduling of threads is needed in that machine so as to utilize the hardware support for various collective operations, as embodied in the machine's control network. For example, the control network can be used to perform barrier synchronization, various reduction operations, and broadcast.

The implementation of gang scheduling on the Connection Machine CM-5 is noteworthy because the architecture employs a buffered multistage network for interprocessor communication. When a partition of PEs switches from one set of threads to another, the partition of the network connecting them has to be flushed. This is done by putting the network in "all fall down" mode (this terminology is motivated by the common graphical rendering of a CM-5, where the PEs are drawn in a linear sequence at the bottom, and the network

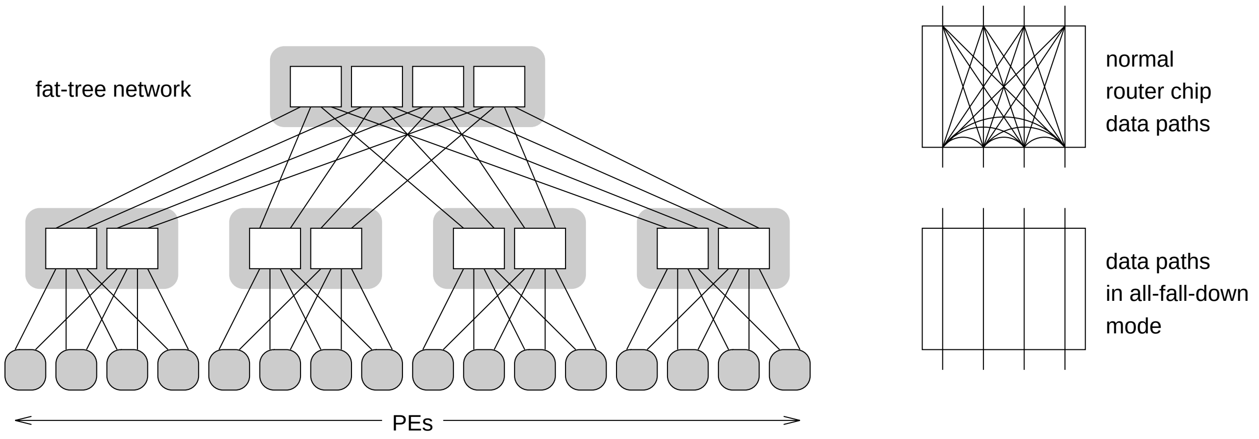


Figure 17: *The CM-5 fat-tree data network is implemented as a multistage network [214]. To flush it, router chips are put in "all-fall-down" mode.*

is structured as a fat tree above them — see Fig. 17). [214]. In this mode, a one-to-one mapping exists between the input ports and output ports of the router chips, so there is no contention[16] Messages are then routed to the nearest node, rather than to their real destination; there they are saved together with the application state. When the application is later scheduled again, these messages are re-injected into the network to complete their transit.

The problem of saving network state did not exist in the earlier CM-2 model, which was a massively parallel SIMD machine. Gang scheduling on the CM-2 was orchestrated by a timesharing daemon that ran on the front-end host [182]. The timesharing daemon could set a flag telling the current process to put itself to sleep. Each job checked this flag before each CM operation was broadcast to the nodes. This mechanism was used so as not to interrupt a CM operation in the middle. Thus it was guaranteed that at the instant of switching from one job to another, there was no activity on the networks. The nodes themselves did not have to do anything — they just went on executing instructions, oblivious to the fact that the instructions now come from a different stream.

5.4 Gang Scheduling with Dynamic Repartitioning

Using fixed partitions runs the risk of significant resource loss due to fragmentation. If all gangs are not of the same size, it is therefore desirable to change the partitioning at each multi-context-switch. This implies that context switching must be coordinated across groups of PEs, and not only within groups. The problem with this approach is the difficulty of doing the partitioning on the fly. The solution is to look for a suitable partitioning only when the load changes, not at each context switch. When a new application arrives or an old one terminates, applications are matched together so as to utilize as many PEs as possible. Then at each context switch the next set of matching applications is scheduled.

[16]The mapping is actually a permutation that can be set by the operating system.

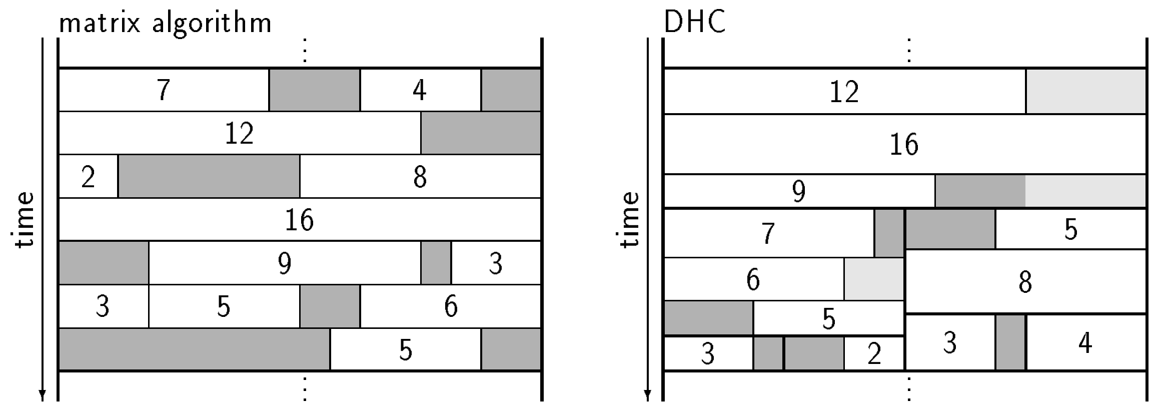


Figure 18: *Example of a scheduling round using the matrix algorithm and DHC. Gray shading indicates fragmentation.*

Synchronous switching

The most common approach to the implementation of dynamic repartitioning is to perform the context switching synchronously across the whole machine. This is done regardless of how the partitioning is supposed to change during the context switching operation. PEs in all the different groups always switch simultaneously, so moving a PE from one group to another during a switch is no problem.

The three coscheduling algorithms developed by Ousterhout fall into this category [268]. The simplest is the matrix algorithm, which was implemented in the Medusa operating system on CM* [269], in the Meiko CS-2 operating system, in the gang-scheduler used for the BBN Butterfly at Lawrence Livermore National Lab [145, 144], in the experimental gang scheduling runtime library of MAXI, the Makbilan operating system [122], In the SHARE scheduler for the IBM SP2 [130], and in the DQT scheme designed for the RWC1 machine [169, 168]. The idea of this algorithm is to view scheduling space as a matrix, where columns represent PEs and rows represent scheduling slots. Gangs of interacting threads are mapped to rows of the matrix. If space allows, a number of gangs can be mapped to the same row. This can be identified by keeping rows linked to each other according to their capacity for additional threads. When a gang terminates, rows may be unified if they use non-overlapping sets of PEs. This can be recognized by performing the logical AND of bitmaps that represent the PEs that have threads mapped to them in each row.

The scheduling proceeds by simultaneous switching of all the PEs from the thread in one row to the thread in the next row. PEs that do not have a thread assigned in the next row may execute any other thread that happens to be mapped to them (this is why the algorithm was originally classified as "coscheduling" rather than "gang scheduling"). However, they remember the row that they were supposed to use, and fall into step again with the other PEs at the next synchronized context switch. An example of a scheduling round, in which each slot is scheduled once, is given in Fig. 18.

The other two algorithms developed by Ousterhout, which were not implemented, are based on a sliding window metaphor [268]. Threads are arranged in one long sequence, with the understanding that a thread in location i is mapped to PE number $i \bmod P$ (where P is the number of PEs in the system). In the “continuous” algorithm, threads belonging to the same gang must be placed within a span of P places in this sequence. In the “undivided” algorithm, threads from the same gang must occupy adjacent places. Scheduling is performed by sliding a window of width P across the sequence, and scheduling those threads that are currently in the window. At the end of each scheduling time quantum, the window is advanced until the leftmost thread in the window is the leftmost thread of a gang that had not been gang-scheduled yet (it is possible that the thread itself had been scheduled in the previous window position, but its whole gang was not). When all gangs have been scheduled, a new sweep is started. These algorithms — and especially the undivided algorithm — have a slightly larger coscheduling effectiveness than the matrix algorithm [268]. In other words, they have a greater tendency to schedule larger fractions of gangs simultaneously. However, it is not clear that this leads to real benefits, as explained in Section 5.1.

Subsets switch independently

It is necessary to synchronize the context switching in different groups of PEs if PEs need to move from one group to the other as part of the context switch. But this is required only if one of the groups must grow. There is no need to synchronize if the groups only split into smaller groups. This observation is used in the design of the “Distributed Hierarchical Control” (DHC) scheme [120, 121].

The DHC scheme partitions the PEs using a buddy system arrangement, that is by successively dividing them into halves. A separate (logical) controller is associated with each partition (Fig. 19). The scheduling is carried out in cycles. Each cycle starts with all the PEs in one group, which is used by the top controller to execute whatever very large gangs are currently in the system (specifically, gangs with so many threads that they need more than half the PEs). After all such gangs have been scheduled for the duration of their respective time slices, the PEs are split into two groups. Each of these groups has a separate controller, which proceeds to schedule the gangs mapped to its group. These controllers do not need to synchronize with each other, and context switches on the two groups of PEs are independent. The splitting continues as smaller gangs are scheduled by lower levels of controllers. At the end of the cycle, all the groups are reunited again to form one large group, and the next cycle starts from the top controller. An example of such a cycle is given in Fig. 18.

The importance of removing extra synchronization is that it decouples groups of PEs with different loads. This allows the time slices to be set differently on different groups of PEs, so as to optimize the execution of different gangs. It also improves the scalability of the system, by removing any components that require full knowledge about the system state. In fact, each controller only needs knowledge about the largest gangs mapped to its group of PEs.

Three complementary mechanisms are used to deal with the issue of fragmentation. The

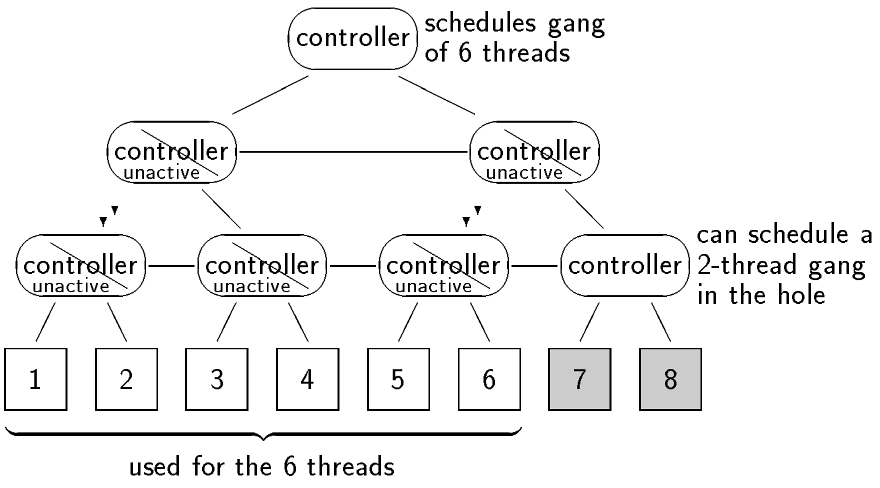


Figure 19: *Distributed Hierarchical Control arranges PEs in a buddy system. Small gangs are mapped to holes left by large ones, and execute at the same time with them.*

first is applied when a new gang is spawned, and involves choosing the least loaded group of PEs that is large enough for this gang. The second mechanism is applied if the new gang is smaller than the full group (which is always a power of two). In this case, the gang has to be split into two, with part being mapped on one half of the PEs in the group and the rest on the other half. Such splitting is always done so that one half is completely utilized, which increases the chance of leaving a large unused group in the other half. These two mechanisms combined ensure that small gangs tend to find a place in holes left by large ones, providing better packing than alternatives such as first-fit or best-fit [115]. Fig. 19 provides an example: when a six-thread gang is mapped onto a group of eight PEs, four threads are mapped onto one half and only two onto the other. This leaves two adjacent PEs unloaded, which makes them a prime target for mapping any new two-thread gang.

The third mechanism is applied during scheduling. When large gangs are scheduled by the top controller, small gangs that are mapped onto leftover PEs are also scheduled. Such opportunities are indicated by the light gray shading in Fig. 18. In the above example, the two-thread gang will be scheduled together with the six-thread gang, during the time slot allocated to the six-thread gang. In addition, it allows the controllers to discriminate in favor of large gangs, and give them larger quantums. This allows large gangs, which make better use of the machine, to receive good service. It doesn't hurt small gangs, because they often manage to execute in holes left by the large gangs [121]. The time quantums in Fig. 18 are computed based on the number of threads in each gang.

Gang scheduling systems based on the ideas of DHC were implemented for the IBM SP2 [130] and the RWC-1 [169, 168]. Gang scheduling on the Intel Paragon uses a similar hierarchical structure based on the nesting of partitions. However, the partition sizes are flexible and not limited to powers of two.

Lazy gang scheduling

Taking the idea of independent switching to the extreme leads to the notion of lazy gang scheduling. This does away with predefined quanta and round-robin scheduling altogether. Instead, each job has a maximal wait time associated with it, based on its class: interactive and debug jobs have short wait times, while batch jobs may wait a very long time. Each time a job's wait time is exceeded, its priority rises, and the lowest priority jobs in the system are preempted to make space for it. The scheduled job then runs for a certain "do-not-disturb" time, which is proportional to its memory footprint. After this period, it itself becomes a candidate for preemption if another high-priority job is waiting. This style of gang scheduling is used on the Cray T3D at LLNL [117], and described in more detail in Section 7.2.2. A variant based on feedback has also been proposed [311].

6 Implications of the Sharing Scheme

A number of studies have compared the effectiveness of different scheduling schemes, typically using their impact on the performance of specific applications as a metric [225, 230, 217, 152, 71, 122, 259, 57]. This section summarizes these results and adds many additional considerations, such as the support for different programming models and the influence on user satisfaction. These considerations are divided into four areas: the interaction between scheduling and the application, the impact on system performance, the support for individual users in the context of a multiuser system, and implementation issues.

6.1 Interaction with Applications

A good match between the scheduling policy and the application can be instrumental in increasing the effectiveness of the application's use of system resources.

Effect on synchronization

Synchronization delays depend on the scheduling policy, because the main issue in synchronization is the temporal alignment of different threads. A mismatch between the scheduling and synchronization mechanisms might result in significant performance penalties [152, 300]. The greatest impact occurs if the threads interact at a fine granularity, i.e. interactions are numerous and come in rapid succession. In this case it is best for the threads to be mapped to PEs using a one-to-one mapping. This allows the synchronization to be handled by busy waiting, saving the overhead of context switching [268, 122].

Support for memory locality

Programming for locality is widely recognized as an important paradigm. Specifically, locality of reference reduces interprocessor communication in multicomputers [294] and improves

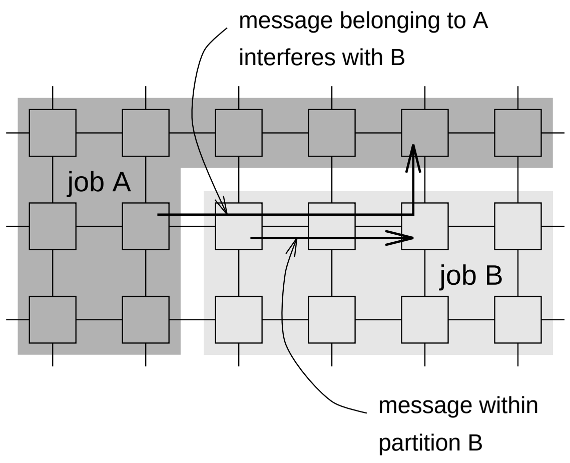


Figure 20: *Due to the predefined routing scheme, concave partitions on the Paragon may cause interference.*

cache performance in multiprocessors [164, 211]. But programming for locality can only work if data stored in a local memory is still in the local memory when it is needed. The operating system should take care not to undermine this paradigm.

The operating system can cause problems in two ways. One is by changing the mapping of threads to PEs. Load balancing or load sharing schemes that migrate threads from one PE to another, leaving data behind, compromise the locality of reference [294, 232, 233]. The other is by causing important data to be flushed as a side effect of context switching. This effect becomes more noticeable the smaller the memory, and is therefore especially relevant for caches [249, 152, 348]. Affinity scheduling, where threads are scheduled back onto previously used PEs so as to benefit from data that may still reside in their caches, tries to counter this effect [348, 86, 25, 328, 339, 57].

Effect on communication locality

Communication is an important aspect of parallel applications. In many systems, communication costs depend on distance. However, this dependency has been decreasing in new architectures, due to wormhole routing and the use of lower-dimensionality networks.

Even if communication *per se* does not depend on distance, using distant PEs can still degrade performance due to increased contention: messages that traverse a larger distance need to pass through more links, so the overall load on the network increases. This is especially troublesome if it induces a lack of independence, that is if messages belonging to one job cause a degradation in the network performance of another. For example, this has been known to happen on the Intel Paragon, when one job is allocated a set of PEs that form a concave or disjoint group, and another job runs in the middle, effectively surrounded by PEs belonging to the first job [224, 251, 246]. Traffic belonging to the first job then passes through the links in the partition belonging to the second job (Fig. 20).

6.2 Impact on System Performance

The way in which competing applications are handled affects the amount of resources that are lost to fragmentation, system overheads, and throughput.

Impact on efficiency

An application that executes on a certain number of PEs for a certain time typically does not utilize them fully. Indeed, adding PEs typically suffers from diminishing returns, so it improves speedup only at the price of reduced efficiency [103, 349, 72, 128, 71, 199]. This is a problem for scheduling schemes based on static partitioning (including and gang scheduling). Adaptive and dynamic partitioning, on the other hand, only gives a job more PEs if there is nothing better to do with them [199, 363, 312, 152, 259, 239]. When multiple jobs compete for resources, each is limited to a relatively small number of PEs, thus boosting its efficiency.

Effect on utilization and throughput

In uniprocessors, multiprogramming improves utilization by overlapping the computation of one job with the I/O of another, using time slicing [195, 279]. The question of whether the same effect applies also to parallel systems is somewhat contentious. It has been argued that parallel machines are (or at least should be) used for extremely computation-intensive applications, and that good programmers expend significant effort to utilize the PEs efficiently. Therefore there is little idle time to use for other applications. Specifically, I/O can be overlapped with computation *within* the application by using asynchronous I/O operations [70]. Moreover, switching from one application to another has its overheads, and corrupts cache state. Therefore it might end up costing more than the gains it provides.

One counter argument is that many applications do not have uniform resource requirements throughout their execution. This is expressed by changes in the degree of parallelism. Effective resource use then dictates that PEs that cannot be used by one application be given to another. For example, a sequential phase of one application can be overlapped with a parallel phase of another. Another counter argument is that not all jobs are compute-intensive. Some require significant amounts of I/O, which they cannot mask by their own computation. Therefore the possibility of overlapping the I/O of one job with computation of another is still important.

Loss of resources to fragmentation

One of the drawbacks of static partitioning is that it may suffer from fragmentation [219, 218, 121, 224, 245]. Internal fragmentation occurs if the partition size is larger than the requested size, or if the degree of parallelism in the program changes during execution. External fragmentation occurs when unallocated PEs are left idle because they cannot form a partition that is suitable for any queued job.

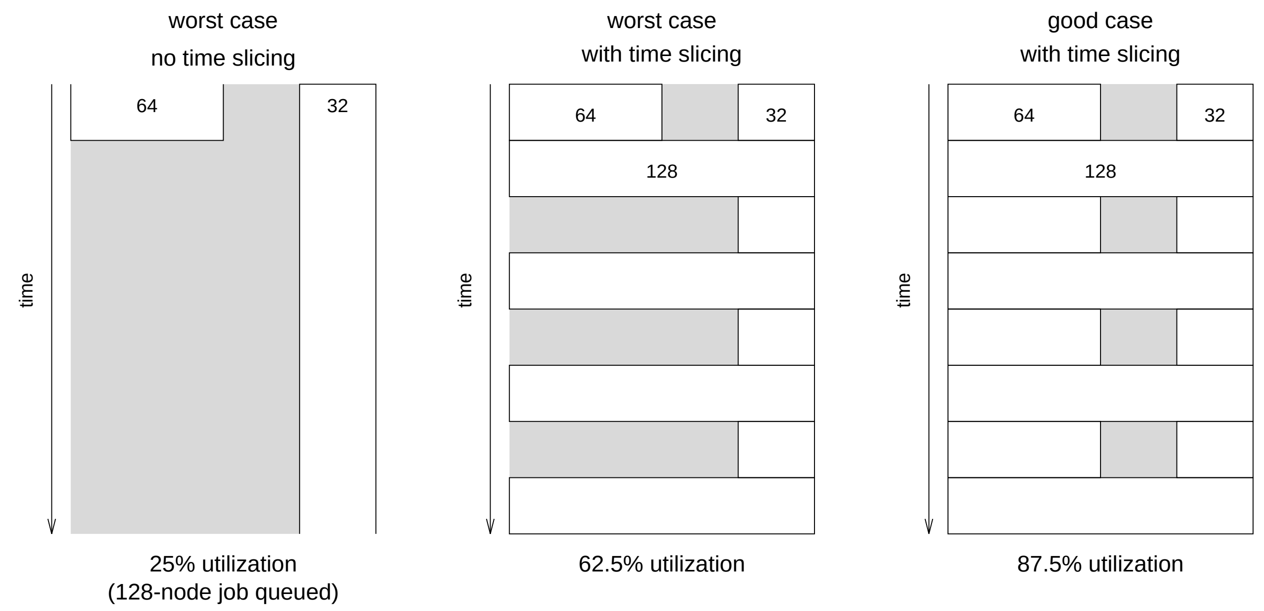


Figure 21: *Example of how the flexibility afforded by time slicing can increase system utilization.*

A number of solutions have been proposed for the fragmentation problem. One solution is to impose a programming model in which applications can use whatever number of processors is available, such as the workpile model. The system can then partition the machine without any fragmentation, and even change the partitioning dynamically to respond to changing loads [342, 366, 71].

Another solution is to reduce the effect of fragmentation by combining partitioning with time-slicing, so as to reduce the effect of scheduling decision on the future. Consider the following simple case as an example (Fig. 21): a 128-node system is currently running a 64-node job, and there are a 32-node job and a 128-node job in the queue. The question is, should the 32-node job be scheduled to run concurrently with the 64-node job? Two outcomes are possible. If the 32-node job is scheduled and it terminates before the 64-node job, resource utilization is improved from 50% possibly up to 75%. But if the 64-node job terminates soon after the 32-node job is scheduled, and the 32-node job runs for a long time, the utilization drops from 50% to 25%. But with gang scheduling, the 32-node job can run in the same time-slot with the 64-node job, while the 128-node job gets a separate time-slot. As long as all three jobs are active, the utilization is 87.5%. Even if the 64-node job terminates, leaving the 32-node job to run alone in its time-slot, the utilization is 62.5%. Naturally, a few percentage points should be shaved off these figures to account for context-switching overhead. Nevertheless, this is a unique case where time-slicing, despite its added overhead, can lead to better resource utilization than batch scheduling [117, 303].

Operating system overhead

One study conducted on a multiprogrammed multiprocessor found that 24% of the total CPU cycles were lost to operating system overhead, and 2/3 of this (16%) was attributed to multiprogramming [91, 90]. This was further classified as base overhead for the context switching itself, which was about 5% irrespective of the workload, and workload-dependent overhead such as additional paging that would not be necessary if the programs executed on a dedicated machine. Naturally, these results depend strongly on system parameters such as the scheduling time quantum. It is always possible to reduce the relative overhead by increasing the time quantum, and the price of slower responsiveness.

The multiprogramming overhead would be avoided if time-slicing was not used, and indeed one of the main arguments for partitioning is that it reduces the overhead for context switching [342]. This is supported to some degree by another study, using a dedicated single-user machine. This study found that 5–21% of the time went to operating system overhead, but only about 2% was attributed to context switching [262].

6.3 Individual Users in a Multiuser System

A major goal of multiprogrammed systems is to support multiple users at once, giving each the illusion of having a dedicated machine (even if it is smaller and slower) [195, 127]. Without proper care, it is easy to compromise this illusion.

Support for the dedicated machine model

The fiction of a dedicated machine for each user is important because this is the only model in which a user can program. It fosters predictable program behavior and allows for performance tuning of applications [95]. It is also needed in order to exploit various theoretical results about algorithm design [184] and task scheduling [142, 65, 226, 137], which are based on the assumption that the PEs are dedicated.

Pure partitioning with one-to-one mapping provides the most direct support for the dedicated machine model. However, the number of simultaneous jobs sharing the machine is limited by the number of PEs (or the maximal number of partitions that can be created, if the minimal partition size has more than a single PE). In small installations, this can be a significant limitation. Gang scheduling also supports the illusion of a dedicated machine effectively, and does not have a hard limit (except, of course, for the size of the system tables).

User separation and protection

With pure partitioning users have less chance to affect each other. If the network is also partitioned, traffic generated by one application does not load that part of the network used by another. Such partitioning is possible in multistage networks, in crossbars, and in hypercubes or meshes where applications run on subcubes or submeshes.

Conversely, in non-partitioned machines, one job may degrade the performance of another. Probably the most infamous example is found in machines like the NYU Ultracomputer and the IBM RP3. These machines provide time-slicing, and PEs are scheduled independently using a shared global queue, so threads belonging to distinct jobs may execute simultaneously. The architecture calls for a shared memory that is accessed via a multistage network. If one of the applications has a memory hot-spot, the whole network may suffer from tree saturation [280]. This may block memory accesses from threads in other applications. The severity of this problem and possible solutions are the subject of considerable controversy [8, sect. 10.3.8].

Sharing does not only effect performance; if proper care is not taken, sharing may compromise user data integrity. In shared memory machines, memory assigned to one application must be protected from access by another application. This is done using normal hardware protection mechanisms that were developed for uniprocessors. In message passing systems, protection means that threads in one application should not be able to send messages directly to PEs running threads from another application. If space-slicing is used, this means that certain PEs should be out of bounds. If time-slicing is used, this means that messages should be addressed to specific threads, not to PEs. Naturally, handling messages addressed to threads other than the running thread causes performance perturbations. In addition, the fact that an identification tag must be added to each message implies that message passing is done via kernel calls, unless special hardware support is provided.

User satisfaction

User satisfaction with system access is usually quantified by the mean response time of submitted jobs. This measure is reduced by always executing the job(s) that have least computing to do, that is scheduling the shortest job first (SJF) [195, 279]. But SJF requires service demand to be known in advance, which is typically not the case, and runs the risk of starving long jobs. Luckily, it is possible to make important choices based on very limited knowledge. Rather than requiring the service demand of each job to be known, it is enough to know the coefficient of variation of the distribution of service demands. If this coefficient is less than 1, it means that jobs have similar demands. In this case it is best to execute them in FIFO order, because at each instant the currently running job may be expected to have less additional demand than a newly arrived job (in other words, the new job is not likely to be shorter). On the other hand, if the coefficient of variation is larger than 1, the distribution of service demands is wide. In this case preemption should be used, because there is a good chance that new jobs will be shorter than those currently in the system [292, 276]. As measurements of actual workloads have repeatedly shown that job service demands have a large coefficient of variation [118, 115, 359, 171, 352], the conclusion is that time slicing will lead to reduced response times.

Fairness and accounting

A quest for fairness can be interpreted as implying that all jobs receive equal service. For example, if one job has more threads than another, each of its threads will run for less time. But this notion of fairness undermines the whole concept of multiprocessing [145]: in effect, when users try to exploit the parallelism in their programs, the system tries to degrade the service they receive.

An alternative interpretation is that the time a job waits for service be proportional to the total amount of service it requires [45, 205, 230, 121]. In other words, when the competition for system resources causes jobs to run slower than on a dedicated machine, all the jobs should be slowed down by the same factor. Using this interpretation, fairness means that *equivalent* jobs should get the same amount of service, not *all* jobs. As the service requirement is generally not known in advance, it is taken to be the number of threads. The problem with this approach is that it gives higher priority to jobs with many threads. Users may abuse this policy by increasing the degree of parallelism in their programs, even if the additional PEs are not used as effectively[17]. This may be solved by accounting: users that want better service, pay for it.

Finally, the goal of fairness might actually be wrong: administrative reasons sometimes favor one job over another. This leads to schedulers that allocate resources according to unequal targets, such as the Unix fair-share schedulers [161]. Related policies are incorporated in the Cray T3E scheduling software [209] and in the Tera MTA scheduler [10].

6.4 Implementation Issues

Implementations might be constrained by architectural features or by lack of hardware support.

Interaction with memory management

Memory management in multiprocessors has an extra dimension that does not exist in uniprocessors, just like scheduling. The question is not only if a page is in memory or swapped out, but also in which memory module it is placed. Significant work has been done in the context of NUMA architectures on the automatic migration of pages from one memory to another, so as to improve locality of reference [42, 40, 302, 212].

Regrettably, the interaction of memory management with scheduling in parallel systems has received only scant attention [278, 49, 239, 57, 310, 274, 273, 240, 331]. This interaction has great importance. Systems that use non-preemptive partitioning typically do not provide any virtual memory or paging, because they cannot afford the overhead of idling a PE

[17]It is often said that users might also create spurious threads that do not perform any useful computation. However, this would not lead to any benefits. If the scheduler is fair at the thread level, the spurious threads will just compete with all the other threads in the system, including the threads that actually do the computation in the same job. This will degrade the service to all jobs, including the one that created the spurious threads.

while waiting for a page fault to be serviced [278, 310, 156]. All the memory is dedicated to a single application at a time, and the application is explicitly required to fit into the available memory. In systems that provide dynamic partitioning, this can be as small as the memory associated with a single PE. If memory is shared, the system must know the memory requirements of different jobs in advance so as not to overcommit memory [196]. Note that as load increases, jobs may be allocated smaller and smaller partitions, in an attempt to service more jobs at once. But this increases the residence time of the jobs, and therefore increases the memory pressure as well [274].

Systems that use preemption, on the other hand, cannot afford not to support memory management as well. When PEs are shared among a number of jobs, so is the memory. The more jobs there are, the harder it is to satisfy their memory requirements at once. This draws the line between short-term scheduling and long-term scheduling: long term scheduling is involved in deciding which jobs will have some memory allocated to them, and which will be swapped out altogether. Short term scheduling is involved in the decision as to which threads will run at a given instant, out of the threads belonging to the memory-resident jobs. Swapping is preferrable to demand paging, because the paging activity on the different PEs may be uncorrelated, and thus cause synchronization problems [353, 49]. While swapping is a basic feature in most uniprocessor systems, very little work has been done in this respect in the context of parallel systems [10, 331, 49, 239]. One notable idea, however, is that the residence time of jobs be proportional to their memory footprint [10, 117, 311]. This places an upper bound on the relative overhead of moving the job's state from memory to disk and back.

Interaction with the architecture and hardware support

Architectural features can have a decisive impact on how the machine is shared among multiple users. For example, a dedicated machine (or partition) is obviously needed for SIMD architectures because of the single instruction decoding unit that controls all the PEs [321]. It is simply not possible to share a SIMD machine by using independent time-slicing on different PEs. Current implementations either use only space-slicing, as in the PASM prototype [324, 323], or gang scheduling, as on the Connection Machine CM-2 [343, 182]. As another example, a shared memory is required for the implementation of a shared global queue of ready threads.

Ease of Implementation

Another consideration distinguishing between different scheduling schemes is the ease of their implementation. The easiest to implement is partitioning: once the PEs are allocated the operating system need do nothing more. Indeed, a number of commercial systems have followed this approach. A close second is the use of local queues, or perhaps a shared global queue; this approach borrows heavily from experience with uniprocessor systems.

Hybrid approaches like gang scheduling are harder to implement. Table 1 bares testimony to this fact: while not a statistically valid sample, it shows that there are many more systems that use either time-slicing or space-slicing, but not both. However, it it encouraging that some important commercial systems such as Mach, the Connection Machine CM-5, the Meiko CS-2, and the Intel Paragon also support a hybrid approach.

7 Case Studies and Example Systems

While it is possible to make many contradicting statements about the superiority of one scheduling scheme over another, many of them do not apply to real-life situations. Real systems often have to cater to many masters. Some want peak performance for long, computationally-intensive jobs. Others want fast interactive response times. A third group insists it needs exclusive use of the machine so as to run benchmarks with no interference whatsoever. Finally, administrators tend to place the greatest emphasis on resource utilization. Therefore real systems sometimes end up supplying different levels of service to different jobs, and using different mechanisms at once. This section presents some case studies of such systems. Some are management systems developed at national labs and supercomputing centers, and others are components of commercial offerings from vendors of specific parallel machines.

We start with the simplest form of systems, those used to queue batch jobs for later execution. Then we turn to systems that serve jobs of multiple types at the same time.

7.1 Batch Queueing Systems

A basic distinction is between batch and interactive (or "direct") jobs. Batch jobs may explicitly be delayed by the system, and executed whenever it is convenient. This allows the system to optimize the running job mix, by choosing jobs judiciously from the queue. A number of batch queueing systems have been designed [126], including NQS and PBS. Systems designed mainly for networks of workstations, such as DQS and LSF, are mentioned below in Section 7.3.

7.1.1 The Network Queueing System (NQS)

The Network Queueing System (NQS) was developed at NASA Ames research Center for their multiprocessor Cray supercomputers [196]. It attempts to create a good job mix, so that the requirements of different jobs complement each other and promote efficient utilization of all available resources. This is done based on a classification of jobs into queues with different resouce limits, most notably on the number of PEs used and on the run time. The batch scheduler then uses the limits to identify jobs that can run side by side using space slicing. In addition, certain queues can be enabled or disabled at different times, so as to provide better support for different workloads.

Implementations of NQS often provide support for other aspects of job classification as well. For example, separate queues can be created for high and low priority jobs, and for

time	*number of nodes*							
limit	1	4	8	16	32	64	128	256
15 min		`q4t`						
1 hour		`q4s`	`q8s`	`q16s`	`q32s`	`q64s`		
			`qf8s`	`qf16s`	`qf32s`			
4 hours					`q32m`	`q64m`	`q128m`	`q256m`
					`qf32m`	`qf64m`	`qf128m`	`qf256m`
12 hours	`q1l`				`q32l`	`q64l`	`q128l`	
					`qf32l`	`qf64l`	`qf128l`	`qf256l`
						`qstandby`		
						`qfstandby`		

`Table 3:` *Example of NQS queues used on the 416-node Paragon at San-Diego Supercomputing Center [352]. Queues with 'f' indicate use of nodes with 32 MB of memory, while the others use nodes with only 16 MB. The standby queues have lower priority, and are also charged at a lower rate.*

different levels of additional resources, such as on-line disk space and memory. This leads to a combinatorical explosion of queues, and makes it harder for users to choose the appropriate one (see e.g. Table 3). Moreover, efficient scheduling depends on exact classification of jobs, which translates to more burden on users. Nevertheless, NQS has been successful in boosting the utilization of many parallel machines, and is a *de-facto* standard in Unix-based supercomputing environments. It is the basis for the POSIX 1003.15 standard.

7.1.2 The Portable Batch System (PBS)

Recently, however, the limitations of NQS are becoming apparent. The major limitation is the lack of control available to system administrators for tuning and special services — all they can do is to configure different queues with different priorities and resource limits. This has prompted work on more general scheduling infrastructures, where elaborate scheduling policies can be expressed. An example is the *Portable Batch System* (PBS), also developed at NASA Ames [160]. It derives its flexibility from the fact that the scheduling policy is handled by an external process, that has access to all the systems knowledge regarding resources, running jobs, and queued jobs.

As an example, a policy that can be implemented with this infrastructure and cannot be implemented in NQS was devised for the SP2 machine at NASA Ames [160]. Special features in this policy include

1. Time is divided into multiple shifts, which are different during weekdays and weekends. This allows buffer zones at the beginning and end of prime time (NQS only supports prime and non-prime time).
2. Job limits change dynamically based on load conditions. Thus during prime time 32-

node jobs can run for up to 4 hours if at least 32 additional nodes are free. Jobs using the last 32 nodes are limited to 10 minutes.

3. Jobs are not started if they might not complete before the end of their shift.

7.2 Catering for Many Masters

The scheduling mechanisms described in Sections 3 through 5 are typically based on a certain preconception of how jobs behave and what services they need. For example, a central queue is designed to serve jobs composed of largely independent threads, dynamic partitioning assumes malleable jobs, and so on. But in a real system the workload is usually very diverse. Applications may be programmed in different styles and have different characteristics, and jobs may have different priorities. A real scheduler has to take these differences into account.

Most large scale distributed memory machines deal with this problem by creating several partitions, each with different attributes. Such systems are described in Section 7.2.1. A more innovative approach is to gang schedule the different jobs using different scheduling characteristics. Such a system was developed at Lawrence Livermore National Lab, and is now available for Cray T3D machines; it is described in Section 7.2.2.

7.2.1 Partitions with Different Attributes

At the highest level, most systems employ two types of partitions: a login partition and a compute partition. The login partition serves as an internal host: when users log onto the machine, their shell process runs on one of the PEs in the login partition. These PEs can also be used for small serial jobs, such as editing and compilation. Only large parallel jobs are executed on the compute partition. Often, a separate, third partition responsible for I/O is used. In very large machines, an additional partition of spare nodes responsible for availability and fault tolerance can be used [235].

In some cases, additional distinctions are made. The compute partition may be subdivided into a number of smaller partitions, that have different characteristics. In many cases, this is used to support different types of jobs. For example, one subpartition may be dedicated to small and short jobs, so as to insure reasonable response time, while another is used for large batch jobs, in the interest of utilization.

An interesting example is provided by the Intel Paragon implementation of NQS. Special provisions are taken to accomodate different node types, e.g. with different amounts of memory [352]. To do so, the nodes in the system are divided into homogeneous *node sets*. Node sets are in turn grouped into *node groups*. NQS queues can be linked to these node groups, and this linkage can change as a function of time of day and day of week. An example is given in Fig. 22.

Node sets:

S1	32 MB nodes
S2	small partition of 16 MB nodes
S3	large partition of 16 MB nodes

NQS queues and node groups:

queue	*time limit*	*node group*	*usage*
q1	short	p+np: S1	short jobs requiring large memory
q2	long	np: S1	long jobs / large memory: only at night
q3	short	p: S2,S3 np: S1,S2,S3	short small jobs avoid large mem during day run anywhere at night
q4	long	p: S2 np: S2,S3	long jobs with small mem and few processors run preferentially on S2
q5	long	p: S3 np: S1,S2,S3	long jobs with small mem and many processors run preferentially on S3

Figure 22: *Example of using node sets and node groups to map jobs on heterogeneous nodes (courtasy of Reagan Moore). p: prime time. np: non-prime time.*

7.2.2 The Lawrence Livermore Gang Scheduler

The gang scheduler developed at Lawrence Livermore National Lab, initially on a BBN Butterfly and then ported to a Cray T3D, uses a finer classification of jobs. Originally, four classes were used [145]. The default class is "interactive". These jobs are gang-scheduled with a time quantum of 10 seconds. The time quantum can be adjusted in the range of 5 to 20 seconds, in the interest of providing a fair share of the resources to different users. Thus users that run multiple jobs concurrently will get a shorter quantum for each job.

The second class is "production", or batch. Production jobs are only run when there are no interactive jobs in the system, or when they can run alongside an interactive job and utilize processors that would otherwise be idle. If multiple production jobs are present, time-slicing is used to execute all of them, but the time quantum is set to 10 minutes to prevent thrashing due to page faults resulting from a context switch. Interactive jobs are automatically demoted to production status if they execute for more than 50 time slices (i.e. 500 seconds).

The third program class is "benchmark". These are jobs that are executed for the purpose of accurate timing measurements, such as those required to generate speedup curves. Jobs in this class are guaranteed exclusive use of the machine, up to a certain time limit. During the day, benchmark jobs preempt the currently running job and get the machine for up to 60 seconds. During the night, the time limit is 2 hours. Programs running in other modes can also request to run a short part in benchmark mode so as to perform timing measurements.

The fourth and final class is "standby". These are jobs that run for free when there is

Job Class	*Priority*	*Wait Time*	*Do-not-disturb Time Multiplier*	*Processor Limit*
Interactive	4	0 Sec	10 Sec	256
Debug	4	300 Sec	1 Year	96
Production	3	1 Hour	10 Sec	256
Benchmark	2	1 Year	1 Year	64
Standby	1	1 Year	3 Sec	256

Table 4: *Daytime scheduling parameters for different job classes on the LLNL Cray T3D.*

nothing else to run on the machine. This class was not actually used, because accounting was not an issue in the context of the study. All three other modes were used extensively, by a large community of real users.

Fine-grain gang scheduling as used on the BBN was not efficient enough on the Cray T3D, so "gang-swapping" was used instead [117]. This was based on the job-swapping service provided by the Cray operating system, which was not available on the Butterfly. As swapping is itself a costly operation (swapping out and in again on the whole machine takes about 8 minutes), its overhead has to be amortized by sufficient useful computation. This is done by using feedback scheduling and time quanta that depend on the partition size.

The LLNL Cray gang scheduler keeps all the non-running jobs in a priority queue. Scheduling decisions are directed by a set of parameters (see Table 4)[18]. The priorities are used to decide which job to service next. If this job has been in the queue for more than its wait time, an attempt to free PEs for it by preempting other jobs is made. However, such jobs are preempted only if their current quantum is already longer than their partition size times the do-not-disturb time multiplier. The 1 year wait time value indicates the job will only run if the PEs are idle; the 1 year time multiplier indicates the job will not be preempted, but the number of nodes used by non-preemptable jobs is limited. Thus jobs are preempted selectively only when the PEs are needed to service another job with higher priority, rather than using periodic preemption. In practice, it turned out that only about 7% of the jobs were ever preempted.

7.2.3 The Tera MTA

The Tera Multi-Threaded Architecture (MTA) is a unique machine that departs from the common wisdom of utilizing off-the-shelf components in order to benefit from the mass market of PC and workstations. Instead, Tera employs custom processors with 128 hardware contexts, allowing threads (called "instruction streams") to be switched on every instruction [11, 9]. This is used to tolerate the access latency to shared memory. The basic design is derived from the earlier HEP machine [325, 180, 194]. The following descripion is based on Alverson at al. [10].

[18]Debug is a new, fifth class of jobs.

The 128 streams may belong to up to 16 protection domains, corresponding to jobs. The set of streams belonging to a single protection domain is called a team. A job may have multiple teams on different processors, and this can change dynamically: new teams are created by a system call, and they retire if they have nothing to do. In addition, teams can acquire and release streams as needed. This is done by special hardware instructions and does not require operating system intervention.

Scheduling is done at several levels. First, the memory scheduler determines which jobs to load into memory; the others are swapped out. From the set of memory resident jobs, the processor scheduler determines which jobs to load into processor protection domains. Once loaded, stream scheduling is handled by the hardware at each instruction cycle.

Tera also makes a distinction between large (batch) jobs and small (interactive) jobs. Memory is statically partitioned into two parts, dedicated to these two types of jobs, with a separate memory scheduler handling the swapping in each one. The residence time of a job in memory, before it is eligible for being swapped out again, is proportional to its memory requirement, so as to amortize the swapping overhead. This is rounded up to a power of two times some minimal interval, in the interest of easier packing. The time a job remains swapped out is set according to its relative priority, when compared against competing jobs. Jobs are swapped one at a time, to make optimal use of the I/O bandwidth available for swapping, and reduce unutilized memory due to jobs that are only partially swapped in and therefore cannot run.

There are also two types of processor schedulers. One is a local processor scheduler on each processor, that handles single-team jobs. Such jobs are scheduled according to conventional Unix feedback mechanisms. The other is a system-wide scheduler for multi-team jobs, which gang schedules the teams on different processors (called the PB scheduler).

7.3 Networks of Workstations

It is often observed that the idle cycles of workstations and personal computers in many large institutions amount to huge computing capabilities. One study has found an average of 91% of the resources are thus wasted [201], and another that 60–70% are wasted even at peak usage hours [18]. A number of systems have been designed to utilize this untapped power in order to execute compute intensive parallel applications, including Condor [223, 286, 112, 285], which is the base technology for the IBM LoadLeveler, Piranha [136, 52], the distributed queueing system (DQS) from the University of Florida [96], the Utopia system [368], which was commercialized under the name Load Sharing Facility (LSF), the Stealth scheduler [201], Amber [113], and Hector [299].

The main challenge in these systems is to improve utilization without noticeable interference with the work of the workstation owner. This leads to two requirements: the identification of idle workstations, and the ability to give them up immediately when the owner returns. Meeting these requirements is the main feature that distinguished the above systems from other systems that allow users to obliviously spawn off processes to multiple hosts, such as PVM[19] [333, 135]. In technical terms, these requirements translate into monitoring facilities and process migration facilities.

Naturally, the priority given to the workstations' owners implies interference with the performance of the parallel programs being run in the background [191, 215]. The performance degradation becomes worse when more workstations are used and when the parallel application is more fine-grained, because then there is an increased probability that one of the processes will be delayed, and that this delay will propagate to all the other processes. A partial solution is to run only one parallel job at a time, so as to avoid interference among the parallel jobs [20, 108, 107]. In addition, it is advisable to limit the degree of parallelism of the parallel jobs to about half the cluster size [18].

Finally, it should be noted that an alternative model is also possible: that the workstations constitute a processor pool, with no constraints imposed by ownership [335, p. 193]. This model is similar to a shared parallel machine, and simplifies the issues of resource allocation. It is used in the Amoeba system [256, 336].

8 Conclusions

Summary

As more users become interested in the use of parallel processing, there is increasing pressure to multiprogram parallel machines. This can be done in two dimensions: temporal, as in conventional time slicing, and spatial, by partitioning the machine and giving each user a distinct partition. These two modes can also be combined, e.g. by gang scheduling within partitions. A large variety of approaches has been implemented in different systems.

A key observation is that mechanisms for partitioning are largely orthogonal to mechanisms for time slicing. It is therefore possible to use either type by itself, or combine mechanisms of both types. The most popular combinations are the following:

Partitioning: only space slicing is used. This approach is typically used in distributed memory machines, which dominate the domain of large-scale machines. It is well suited for batch processing, and can also be used for interactive work provided jobs are short and enough PEs are generally available.

Global queue: all the PEs serve the same queue, which holds all the threads in the system. This approach is common in small-scale UMA shared memory systems, such as those with bus-connected processors.

Gang scheduling: gangs of threads (typically all the threads in an application) are scheduled simultaneously on distinct PEs. Time-slicing is used to share the PEs' resources, using synchronized multi-context-switching. This can be done across the whole machine, or it can be combined with partitioning and done within the confines of a single partition.

[19]Recently, there has been work on adding load considerations to PVM too [54, 285, 270].

Two-level scheduling: a combination of partitioning to allocate PEs to each job, and time slicing to run the job's threads on those PEs. The thread management is typically done in user space.

The state of the art

Views of the state of the art in scheduling on multiprogrammed parallel machines depend on who you ask. Especially troubling is the wide division between published academic work and practice in the field.

Most research results from academia point to a strong belief in two-level scheduling with dynamic partitioning. This scheme is repeatedly shown to provide better response times and throughput than competing schemes, due to four main features:

- There is no loss of resources to fragmentation.
- There is no overhead for context switching, except that for redistributing the PEs when the load changes. The second level of scheduling within the application is assumed to require less overhead.
- There is no waste of CPU cycles on busy waiting for synchronization, as threads can be blocked inexpensively.
- The degree of parallelism provided to each job is automatically decreased under load conditions, leading to better efficiency. This is often hidden from application programmers by a user-level thread library.

However, dynamic partitioning has its limitations. A major drawback is that as load increases, partitions become smaller and may not contain enough memory for the application's dataset. Therefore the integration of dynamic partitioning with memory management has to be addressed. Another limitation is the restriction to programming environments that use the workpile programming model, and have a runtime system that is closely integrated with the system scheduler. Other programming models, such as the popular SPMD model, are not supported. Even with a suitable environment, implementation considerations indicate that two-level scheduling is mainly suitable for UMA shared-memory machines.

Indeed, two-level scheduling with dynamic partitioning is not practiced outside of the academic institutions that promote it. The large-scale parallel computing market is currently dominated by distributed-memory machines, most of which support a message-passing programming model. Programmers invest significant effort in structuring their applications so as to achieve high performance in this environment. The common wisdom is that performance is improved by programming for locality, and reducing communication requirements. In practice this implies distributing the dataset among the PEs, and grouping messages so as to amortize startup costs. This style of programming is explicitly based on the notion that the programmer owns the machine, and controls the use of resources. Language and compiler design also follow this principle. In particular, little work is being done on how to adjust to changes in available resources, so as to accommodate competing jobs.

When partitioning is used as the only means to share a machine, each partition does indeed behave much like a dedicated machine. Current programming practices are thus supported, even if this comes at the expense of some fragmentation. The problem is that as the number of users who want access to parallel machines grows, system administrators dictate the use of smaller partitions, and at peak hours even these are not always obtainable. The users lose twice: they cannot obtain enough resources to run really big production codes, and they also cannot get the continuous interactive access needed for program development and testing. This state of affairs is increasingly leading to the conclusion that system administrators need better scheduling tools for resource management.

Obviously, there is no simple solution to the conflict between users desire for a dedicated system and system administrators need to manage limited resources. A promising compromise is to use gang scheduling. By time slicing whole jobs, gang scheduling provides the illusion of a dedicated (albeit slower) machine, just like time slicing on uniprocessors. This can be used in a number of ways. In large installations, the parallel machine is typically partitioned into two or more partitions. Most of the machine is then used in batch mode, to execute long production runs. At the same time, a small part of the machine is gang scheduled and provides an interactive environment for program development and the execution of short jobs. The mechanisms used for gang scheduling can also be used to checkpoint and restart batch jobs, allowing time-slicing of the batch partitions between jobs that take weeks or months and jobs that merely take a few hours. In small installations, gang scheduling can be used to switch between the interactive workload and the batch workload. Thus batch jobs are automatically run on the whole machine at off hours, but interactive jobs can always reclaim it. While gang scheduling incurs some overhead, it can actually increase the overall utilization over partitioning, because of reduced fragmentation. Most vendors of parallel machines either provide gang scheduling capabilities already, or have expressed an interest in doing so.

Research directions

Parallel operating systems have been widely neglected as a field of research, and implementations have often been based on adaptations of uniprocessor or distributed systems. There is room for significant improvements. Regarding the specific area of scheduling in multiprogrammed systems, an obvious goal is to reconcile the differences between academia and practice. To do so, it is instructive to ferret out the roots of the gap between the two. It seems that the problem lies in the milieu in which the work is done, and the resulting requirements.

Academic work is done in an environment that is forgiving on one hand while being severe on the other. It allows various idealistic assumptions to be made, even if they do not match contemporary real-world constraints, as long as they are explicitly documented. But it requires results to be demonstrated unequivocally within the setting created by these assumptions, using well defined and quantifiable metrics. This sometimes leads to an effect of looking for a lost coin under the street light, where known techniques are applied to analyzable systems, with certain disregard to additional requirements that would make the system too hard to handle. The results are then postulated to hold in general, whereas in reality the additional requirements undermine the whole basis for the analysis.

Rather than giving specific examples of work that might suffer from such circumstances, let us attempt to identify those aspects of scheduling in multiprogrammed parallel systems that would most benefit from additional research. One such issue is that of *requirements*. Making intelligent scheduling decisions depends on the availability of information regarding the workload. So far systems simply had to make do without such information, or else users had to supply it in the form of bounds on resource use. There are interesting research opportunities in the area of automatic classification of parallel jobs, and assessment of their resource requirements [140, 93]. This should probably involve some cooperation between the compiler and the runtime system. In addition, there is a dire need for information about statistical properties of parallel workloads in general, e.g. the distribution of partition sizes that are requested in a general-purpose multiprogrammed system. Such parameters can be found by analyzing traces from existing systems [118, 115, 359, 171, 352, 114]. Of course, care should be taken not to interpret the results too broadly, as the workload on a given system necessarily reflects the properties of the system, and cannot automatically be assumed to apply to other systems that provide other services.

A long term research goal is to arrive at a holistic view of parallel operating systems, that *integrates* all the system services. Scheduling should not be addressed as an independent issue; rather, the interactions between scheduling, parallel I/O, and memory management should be acknowledged. For example, collective I/O operations can be integrated with gang scheduling to allow overlap between the computation of one job and the I/O of another. Memory availability can be used to differentiate between short-term scheduling and long-term scheduling, where whole jobs are swapped out to make space for others. These ideas are standard in uniprocessor systems, but are just beginning to migrate to parallel ones [10]. System integration also includes the cooperation between operating system, runtime system, and compiler. It is made difficult by the requirement to maintain compatibility with previous systems, some of which were designed under different assumptions. Also, there is the question of standardization in the interest of portability and interoperability among system components from different vendors [124].

Finally, the *human* aspect of scheduling has been completely absent so far. User satisfaction is extremely important, because how the provided service is *perceived* may have more impact on what machines are bought and used than more objective measures of performance. Moreover, user attitudes should be used to guide scheduling policies. For example, it has been shown that allocating equal resources to different jobs results in overall low mean response times, that are relatively insensitive to various workload parameters. However, this assumes that all jobs are equally important. In a real system, users typically have opinions about what levels of service are (un)acceptable. These opinions are often correlated with resource requirements: interactive jobs with low requirements call for prompt service, whereas extremely large jobs can tolerate some delays.

Concluding remarks

Multiprogramming in parallel systems is far from being a closed deal. While there is growing recognition of the need for multiprogramming, there are still those who claim that parallel supercomputers are too expensive to be used to support users directly, and they should be dedicated to the execution of supercomputer class jobs. As for the implementation of multiprogramming, much progress has been made in recent years, but there is still much to be done. An important observation is that it is hard to add multiprogramming to a system that was not designed with such use in mind. Multiprogramming support must be built into the system design from the start, because it has considerable effect on many different system attributes. In some cases, multiprogramming can benefit greatly from specialized hardware support. As more systems are designed with explicit regard to multiprogramming, we can expect performance to improve and objections to dwindle.

Acknowledgements

Thanks to Brent Gorda and Moe Jette for information about the LLNL gang scheduler, Jim Cownie for information about the Meiko CS-2, Joe Brandenburg, Reagan Moore, and Andy Pfiffer for information about the Intel Paragon, Gérard Vichniac for information about the KSR1, and Chris Johnson for information about the Fujitsu AP1000.

A Terminology

The following list explains the terminology as it is used in this paper.

Adaptive Partitioning — a partitioning scheme that sets the partition size allocated to new jobs according to the load at the time of their arrival.

Affinity Scheduling — re-scheduling a thread on the same PE it used before, based on the assumption that some of its data might still be in the cache on that PE. Relevant for systems that do not map threads to PEs.

Blocking — when a thread relinquishes its PE because it must wait for some synchronization event (e.g. the arrival of a message).

Chore — the unit of parallel work in an application, especially in the case where work is represented by an unordered workpile. Chores are typically executed to completion, without preemption. See also **task** and **thread**.

Coscheduling — A variant of gang scheduling that does not guarantee that all the threads will always run simultaneously. This is based on the assumption that running a fraction of the threads is better than nothing.

Dispatching — selecting the next thread to run from those that are ready and memory-resident. Relevant to time slicing systems.

Dynamic Partitioning — a partitioning scheme that changes the partition size allocated to jobs at runtime, to reflect changes in job requirements and system load.

Evolving Job — one that has different requirements (re number of PEs) in different phases of the computation.

Family Scheduling — A scheduling scheme that combines these features:

1. Threads are grouped into families, typically all the threads in the job.
2. Time slicing is used, with each family receiving a set of PEs for its exclusive use each time.
3. Each family has more threads than the number of PEs it gets to run on, so whenever it is scheduled there is an additional level of internal time slicing among the threads of the family.

Fixed Partitioning — the use of predefined partitions oblivious to job requirements and system load.

Flexible Partitioning — when any subset of PEs can be grouped into a partition. For example, in hypercubes the partitions must form subcubes, so the partitioning is not flexible.

Fragmentation — the situation where some PEs are left idle because of the partitioning mechanism. **Internal fragmentation** occurs when a partition contains more PEs than the application requires (e.g. the next power of two). **External fragmentation** occurs when a whole partition is left idle because the next queued job requires a larger partition.

Gang Scheduling — a scheduling scheme that combines these three features:

1. Threads are grouped into gangs.
2. All the threads in a gang are always scheduled to execute simultaneously on distinct PEs, using a one-to-one mapping.
3. Time slicing is used, with all the threads in a gang being preempted and rescheduled at the same time.

Group Scheduling — a collective name for schemes that guarantee that a group of threads run simultaneously. Includes static partitioning and gang scheduling.

Global Queue — a ready queue shared by more than one PE. Implies that threads are not mapped to PEs.

Handoff — when one thread yields the PE in favor of a specified other thread.

Job — the sum of executing entities that comprise a single application, as known to the operating system. This typically means a set of threads running on different PEs.

Load Balancing — the activity of migrating threads from one PE to another so that the loads on the different PEs will be equal. The term is also used to describe the goal of certain mapping schemes, even if migration is not used later to adjust to changing conditions. In either case, this is only relevant for systems that map threads to PEs.

Load Sharing — a property of parallel systems whereby no PE will be idle if there are any ready threads in the system. This might apply to systems that keep ready threads in a global queue and do not map them to PEs, or to systems that map threads to PEs and migrate one thread at a time from loaded PEs to idle ones, rather than attempting to balance the loads.

Local Queue — a ready queue containing threads mapped to a specific PE.

Malleable Job — one that can adjust to changes in the allocation of PEs at runtime.

Mapping — the association of threads with PEs. Mapping at runtime should not be confused with "the mapping problem", which is part of program development, and involves the decision of which PE will run which task in the program. This is only relevant if the PEs are dedicated, their number is known in advance, and the program is represented as a task graph.

Migration — relocating threads from one PE to another in the interest of load balancing or reducing fragmentation, in systems where threads are mapped to PEs.

Moldable Job — one that allows the number of PEs used to be set when it is launched, but then the number must stay fixed.

Multiprocessing — the use of multiple PEs for the same job. Synonymous to parallel processing.

Multiprogramming — the concurrent execution of multiple jobs on the machine. In parallel systems, this can be achieved by time slicing, space slicing, or a combination of both.

Multitasking — time slicing on a single PE. The threads mapped to this PE can belong to the same job or to different jobs.

Multi-Context-Switch — the operation of switching from the execution of one job to another, in a gang scheduling system.

Partitioning — a synonym for **space slicing**. See also **fixed partitioning**, **variable partitioning**, **adaptive partitioning**, and **dynamic partitioning**.

PE (Processing Element) — generic name for a node in a parallel machine. This is the unit for allocating processing power. In multicomputers, a PE typically includes a CPU and local memory. In a multiprocessor, it might be just a CPU, typically with a cache.

Preemption — the de-scheduling of a thread (or job) in order to give its PE(s) to another. This is done without the tread's concent, as opposed to **blocking**.

Priority Queue — a queue such that new items can be inserted anywhere in it, according to their priority relative to items already in the queue. Often implemented as a set of queues for the different priorities.

Process — an autonomous execution unit. In uniprocessors, a process is a job in execution. This includes a thread of control and its resources, or context. In some parallel systems, especially those based on Unix, "process" may be used as a synonym for **thread**, in the sense that multiple processes are written such that they communicate and cooperate. However, this is unknown to the operating system.

Processor Allocation — allocating PEs for the exclusive use of a certain job.

Ready Queue — a queue of threads that are ready to run.

Rigid Job — one that requires a predefined number of PEs in order to run.

Runtime System — part of a programming system that is linked with applications, and supports the abstractions of the programming system at runtime.

Single-Level Scheduling — scheduling schemes that combine the allocation of processing power with the decision of which thread will use it.

Space Sharing — a synonym for **Space slicing**. We prefer "space slicing", because "space sharing" sounds as if the same space is being shared, whereas actually the machine is being partitioned into disjoint spatial pieces.

Space Slicing — sharing of a parallel machine by partitioning it and allocating different PEs to different jobs.

Static Partitioning — a collective name for partitioning schemes where partitions do not change at runtime: fixed, variable, and adaptive partitioning.

Task — the unit of parallelism in an application, especially when the application is represented as a graph of tasks with interdependencies. Tasks are typically executed to completion, without preemption. See also **thread** and **chore**. In the context of the

Mach system, **task** refers to an abstraction used for resource allocation, e.g. an address space.

Task Graph — the representation of a parallel program as a directed graph, where nodes are basic blocks of code (tasks), and arcs represent data and control dependencies among them.

Thread — the unit of parallelism in an application, especially in applications based on control parallelism. In this context, "thread" is short for "thread of control" or "light-weight process". The state associated with a thread includes its program counter, CPU registers, and call stack. Depending on the system, threads may execute to completion or else they may be preempted. See also **task** and **chore**.

Time Sharing — a synonym for **time slicing**.

Time Slicing — sharing of a PE or a set of PEs by context switching among a number of jobs.

Two-Level Scheduling — a collective name for schemes that decouple the allocation of PEs from the decision of what parts of the program will be executed on each one. The first level, **processor allocation**, is done by the operating system at a low rate determined by the job submission rate. The second level, that of scheduling program parts on these PEs, is done by the language runtime system or the application itself.

Variable Partitioning — a flexible partitioning scheme that sets the partition size allocated to new jobs according to their requests.

Workpile — an unordered collection of independent chores to be computed. The lack of a specified order distinguishes a workpile from a queue, which is ordered.

Yielding — when a thread voluntarily relinquishes the PE it is running on.

References

[1] G. A. Abandah and E. S. Davidson, "*Modeling the communication performance of the IBM SP2*". In 10th *Intl. Parallel Processing Symp.*, pp. 249–257, Apr 1996.

[2] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. kurihara, B-H. Lim, G. Maa, and D. Nussbaum, "*The MIT Alewife machine: a large-scale distributed-memory multiprocessor*". In *Scalable Shared Memory Multiprocessors*, M. Dubois and S. S. Thakkar (eds.), pp. 239–261, Kluwer Academic, 1992.

[3] A. Agarwal, J. Kubiatowicz, D. Kranz, B-H. Lim, D. Yeung, G. D'Souza, and M. Parkin, "*Sparcle: an evolutionary processor design for large-scale multiprocessors*". *IEEE Micro* **13(3)**, pp. 48–61, Jun 1993.

[4] G. Aharoni, D. G. Feitelson, and A. Barak, "*A run-time algorithm for managing the granularity of parallel functional programs*". *J. Functional Programming* **2(4)**, pp. 387–405, Oct 1992.

[5] I. Ahmad, "*Editorial: resource management of parallel and distributed systems with static scheduling: challenges, solutions, and new problems*". *Concurrency — Pract. & Exp.* **7(5)**, pp. 339–347, Aug 1995.

[6] I. Ahmad, A. Ghafoor, and G. C. Fox, "*Hierarchical scheduling of dynamic parallel computations on hypercube multicomputers*". *J. Parallel & Distributed Comput.* **20(3)**, pp. 317–329, Mar 1994.

[7] H. H. Ali and H. El-Rewini, "*On the intractability of task allocation in distributed systems*". *Parallel Process. Lett.* **4(1&2)**, pp. 149–157, Jun 1994.

[8] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing.* Benjamin/Cummings, 1989.

[9] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith, "*Exploiting heterogeneous parallelism on a multithreaded multiprocessor*". In *Intl. Conf. Supercomputing*, pp. 188–197, Jul 1992.

[10] G. Alverson, S. Kahan, R. Korry, C. McCann, and B. Smith, "*Scheduling on the Tera MTA*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 19–44, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[11] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "*The Tera computer system*". In *Intl. Conf. Supercomputing*, pp. 1–6, Jun 1990.

[12] T. E. Anderson, "*The performance of spin lock alternatives for shared-memory multiprocessors*". *IEEE Trans. Parallel & Distributed Syst.* **1(1)**, pp. 6–16, Jan 1990.

[13] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "*Scheduler activations: effective kernel support for the user-level management of parallelism*". *ACM Trans. Comput. Syst.* **10(1)**, pp. 53–79, Feb 1992.

[14] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "*The performance implications of thread management alternatives for shared-memory multiprocessors*". *IEEE Trans. Comput.* **38(12)**, pp. 1631–1644, Dec 1989.

[15] F. André, J-L. Pazat, and H. Thomas, "*Pandore: a system to manage data distribution*". In *Intl. Conf. Supercomputing*, pp. 380–388, Jun 1990.

[16] M. Annaratone, M. Fillo, K. Nakabayashi, and M. Viredaz, "*The K2 parallel processor: architecture and hardware implementation*". In 17th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 92–101, May 1990.

[17] R. Arlauskas, "*iPCS/2 system: a second generation hypercube*". In 3rd *Conf. Hypercubes, Concurrent Comput., & Appl.*, vol. I, pp. 38–42, Jan 1988.

[18] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, "*The interaction of parallel and sequential workloads on a network of workstations*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 267–278, May 1995.

[19] I. Ashok and J. Zahorjan, "*Scheduling a mixed interactive and batch workload on a parallel, shared memory supercomputer*". In *Supercomputing '92*, pp. 616–624, Nov 1992.

[20] M. J. Atallah, C. L. Black, D. C. Marinescu, H. J. Siegel, and T. L. Casavant, "*Model and algorithms for coscheduling compute-intensive tasks on a network of workstations*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 319–327, Dec 1992.

[21] W. C. Athas and C. L. Seitz, "*Multicomputers: message-passing concurrent computers*". *Computer* **21(8)**, pp. 9–24, Aug 1988.

[22] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "*Balanced allocations*". In 26th *Ann. Symp. Theory of Computing*, pp. 593–602, May 1994.

[23] D. Babbar and P. Krueger, "*On-line hard real-time scheduling of parallel tasks on partitionable multiprocessors*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 29–38, Aug 1994.

[24] M. J. Bach and S. J. Buroff, "*Multiprocessor UNIX operating systems*". *AT&T Bell Labs Tech. J.* **63(8, part 2)**, pp. 1733–1749, Oct 1984.

[25] J. E. Bahr, S. B. Levenstein, L. A. McMahon, T. J. Mullins, and A. H. Wottreng, "*Architecture, design, and performance of Application System/400 (AS/400) multiprocessors*". *IBM J. Res. Dev.* **36(6)**, pp. 1001–1014, Nov 1992.

[26] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "*A static performance estimator to guide data partitioning decisions*". In 3rd *Symp. Principles & Practice of Parallel Programming*, pp. 213–223, Apr 1991.

[27] A. Barak, S. Guday, and R. G. Wheeler, *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag, 1993. Lecture Notes in Computer Science Vol. 672.

[28] A. Barak and A. Litman, "*MOS: a multiprocessor distributed operating system*". *Software — Pract. & Exp.* **15(8)**, pp. 725–737, Aug 1985.

[29] A. Barak and A. Shiloh, "*A distributed load-balancing policy for a multicomputer*". *Software — Pract. & Exp.* **15(9)**, pp. 901–913, Sep 1985.

[30] E. Barton, J. Cownie, and M. McLaren, "*Message passing on the Meiko CS-2*". *Parallel Comput.* **20(4)**, pp. 497–507, Apr 1994.

[31] J. M. Barton and N. Bitar, "*A scalable multi-discipline, multiple-processor scheduling framework for IRIX*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 45–69, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[32] P. Barton-Davis, D. McNamee, R. Vaswani, and E. D. Lazowska, "*Adding scheduler activations to Mach 3.0*". In *USENIX Mach III Symp.*, pp. 119–136, Apr 1993.

[33] F. Bellosa, "*Locality-information-based scheduling in shared-memory muliprocessors*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 271–289, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[34] M. Beltrametti, K. Bobey, and J. R. Zorbas, "*The control mechanism for the Myrias parallel computer system*". *Computer Architecture News* **16(4)**, pp. 21–30, Sep 1988.

[35] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph, "*ParC—an extension of C for shared memory parallel processing*". *Software — Pract. & Exp.* **26(5)**, pp. 581–612, May 1996.

[36] D. Bernstein, M. Rodeh, and I. Gertner, "*On the complexity of scheduling problems for parallel/pipelined machines*". *IEEE Trans. Comput.* **38(9)**, pp. 1308–1313, Sep 1989.

[37] S. Bhattacharya and W-T. Tsai, "*Lookahead processor allocation in mesh-connected massively parallel multicomputer*". In 8th *Intl. Parallel Processing Symp.*, pp. 868–875, Apr 1994.

[38] G. E. Bier and M. K. Vernon, "*Measurement and prediction of contention in multiprocessor operating systems with scientific application workloads*". In *Intl. Conf. Supercomputing*, pp. 9–15, Jul 1988.

[39] D. L. Black, "*Scheduling support for concurrency and parallelism in the Mach operating system*". *Computer* **23(5)**, pp. 35–43, May 1990.

[40] D. L. Black, A. Gupta, and W-D. Weber, "*Competitive management of distributed shared memory*". In 34th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 184–190, Spring 1989.

[41] J. E. Boillat, "*Load balancing and Poisson equation in a graph*". *Concurrency — Pract. & Exp.* **2(4)**, pp. 289–313, Dec 1990.

[42] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox, "*NUMA policies and their relation to memory architecture*". In 4th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 212–221, Apr 1991.

[43] H. Boral and D. J. DeWitt, "*Processor allocation strategies for multiprocessor database machines*". *ACM Trans. Database Syst.* **6(2)**, pp. 227–254, Jun 1981.

[44] T. B. Brecht and K. Guha, "*Using parallel program characteristics in dynamic processor allocation policies*". *Performace Evaluation* **27&28**, pp. 519–539, Oct 1996.

[45] P. Brinch Hansen, "*An analysis of response ratio scheduling*". In *IFIP Congress, Ljubljana*, pp. TA-3 150–154, Aug 1971.

[46] R. Bryant, H-Y. Chang, and B. Rosenburg, "*Experience developing the RP3 operating system*". *Computing Systems* **4(3)**, pp. 183–216, Summer 1991.

[47] R. M. Bryant, H-Y. Chang, and B. S. Rosenburg, "*Operating system support for parallel programming on RP3*". *IBM J. Res. Dev.* **35(5/6)**, pp. 617–634, Sep/Nov 1991.

[48] R. M. Bryant and R. A. Finkel, "*A stable distributed scheduling algorithm*". In 2nd *Intl. Conf. Distributed Comput. Syst.*, pp. 314–323, Apr 1981.

[49] D. C. Burger, R. S. Hyder, B. P. Miller, and D. A. Wood, "*Paging tradeoffs in distributed-shared-memory multiprocessors*". *J. Supercomput.* **10(1)**, pp. 87–104, 1996.

[50] G. D. Burns, A. K. Pfiffer, D. L. Fielding, and A. A. Brown, "*Trillium operating system*". In 3rd *Conf. Hypercubes, Concurrent Comput., & Appl.*, vol. I, pp. 374–376, Jan 1988.

[51] R. Calkin, R. Hempel, H-C. Hoppe, and P. Wypion, "*Portable programming with the PARMACS message passing library*". *Parallel Comput.* **20(4)**, pp. 615–632, Apr 1994.

[52] N. Carriero, E. Freedman, D. Gelernter, and D. Kaminsky, "*Adaptive parallelism and Piranha*". *Computer* **28(1)**, pp. 40–49, Jan 1995.

[53] N. Carriero, E. Freeman, and D. Gelernter, "*Adaptive parallelism on multiprocessors: preliminary experience with Piranha on the CM-5*". In *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (eds.), pp. 139–151, Springer-Verlag, Aug 1993. Lecture Notes in Computer Science Vol. 768.

[54] J. Casas, R. Konuru, S. W. Otto, R. Prouty, and J. Walpole, "*Adaptive load migration systems for PVM*". In *Supercomputing '94*, pp. 390–399, Nov 1994.

[55] T. L. Casavant and J. G. Kuhl, "*Effect of response and stability on scheduling in distributed computing systems*". *IEEE Trans. Softw. Eng.* **14(11)**, pp. 1578–1588, Nov 1988.

[56] T. L. Casavant and J. G. Kuhl, "*A taxonomy of scheduling in general-purpose distributed computing systems*". *IEEE Trans. Softw. Eng.* **14(2)**, pp. 141–154, Feb 1988.

[57] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "*Scheduling and page migration for multiprocessor compute servers*". In 6th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 12–24, Nov 1994.

[58] H. Y. Chang and B. S. Rosenburg, "*"ration function" that lets a parallel program adapt its processor requirements to system load*". *IBM Technical Disclosure Bulletin* **32(8B)**, pp. 123–125, Jan 1990.

[59] S. J. Chapin and E. H. Spafford, "*Support for implementing scheduling algorithms using MESSIAHS*". *Scientific Programming* **3(4)**, pp. 325–340, Winter 1994.

[60] G-I. Chen and T-H. Lai, "*Scheduling independent jobs on hypercubes*". In 5th *Symp. Theoretical Aspects of Computer Science*, pp. 273–280, Springer-Verlag, Feb 1988. Lecture Notes in Computer Science Vol. 294.

[61] M-S. Chen and K. G. Shin, "*Processor allocation in an n-cube multiprocessor using Gray codes*". *IEEE Trans. Comput.* **C-36(12)**, pp. 1396–1407, Dec 1987.

[62] M-S. Chen and K. G. Shin, "*Subcube allocation and task migration in hypercube multiprocessors*". *IEEE Trans. Comput.* **39(9)**, pp. 1146–1155, Sep 1990.

[63] S-H. Chiang and M. K. Vernon, "*Dynamic vs. static quantum-based parallel processor allocation*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 200–223, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[64] A. N. Choudhary, J. H. Patel, and N. Ahuja, "*NETRA: a hierarchical and partitionable architecture for computer vision systems*". *IEEE Trans. Parallel & Distributed Syst.* **4(10)**, pp. 1092–1104, Oct 1993.

[65] W. W. Chu, L. J. Holloway, M-T. Lan, and K. Efe, "*Task allocation in distributed data processing*". *Computer* **13(11)**, pp. 57–69, Nov 1980.

[66] Y-K. Chu and D. T. Rover, "*An efficient two-dimensional mesh partitioning strategy*". *Parallel Process. Lett.* **5(4)**, pp. 623–634, Dec 1995.

[67] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "*Approximation algorithms for bin-packing — an updated survey*". In *Algorithm Design for Computer Systems Design*, G. Ausiello, M. Lucertini, and P. Serafini (eds.), pp. 49–106, Springer-Verlag, 1984.

[68] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "*Bin packing with divisible item sizes*". *J. Complex.* **3(4)**, pp. 406–428, Dec 1987.

[69] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan, "*Performance bounds for level-oriented two-dimensional packing algorithms*". *SIAM J. Comput.* **9(4)**, pp. 808–826, Nov 1980.

[70] P. F. Corbett, D. G. Feitelson, J-P. Prost, and S. J. Baylor, "*Parallel access to files in the Vesta file system*". In *Supercomputing '93*, pp. 472–481, Nov 1993.

[71] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos, "*Multiprogramming on multiprocessors*". In 3rd *IEEE Symp. Parallel & Distributed Processing*, pp. 590–597, 1991.

[72] Z. Cvetanovic, "*The effects of problem partitioning, allocation, and granularity on the performance of multiple processor systems*". *IEEE Trans. Comput.* **C-36(4)**, pp. 421–432, Apr 1987.

[73] G. Cybenko, "*Dynamic load balancing for distributed memory multiprocessors*". *J. Parallel & Distributed Comput.* **7**, pp. 279–301, 1989.

[74] W. J. Dally, J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler, "*The message-driven processor: a multicomputer processing node with efficient mechanisms*". *IEEE Micro* **12(2)**, pp. 23–39, Apr 1992.

[75] W. J. Dally and D. S. Wills, "*Universal mechanisms for concurrency*". In *Parallel Arch. & Lang. Europe*, vol. I, pp. 19–33, Springer-Verlag, Jun 1989. Lecture Notes in Computer Science Vol. 365.

[76] S. P. Dandamudi, "*Reducing run queue contention in shared memory multiprocessors*". *Computer* **30(3)**, pp. 82–89, Mar 1997.

[77] S. P. Dandamudi and P. S. P. Cheng, "*A hierarchical task queue organization for shared-memory multiprocessor systems*". *IEEE Trans. Parallel & Distributed Syst.* **6(1)**, pp. 1–16, Jan 1995.

[78] S. K. Das, M. C. Pinotti, and F. Sarkar, "*Optimal and load balanced mapping of parallel priority queues in hypercubes*". *IEEE Trans. Parallel & Distributed Syst.* **7(6)**, pp. 555–564, Jun 1996.

[79] D. Das Sharma, G. D. Holland, and D. K. Pradhan, "*Subcube level time-sharing in hypercube multicomputers*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 134–142, Aug 1994.

[80] D. Das Sharma and D. K. Pradhan, "*Job scheduling in mesh multicomputers*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 251–258, Aug 1994.

[81] D. Das Sharma and D. K. Pradhan, "*A novel approach for subcube allocation in hypercube multiprocessors*". In 4th *IEEE Symp. Parallel & Distributed Processing*, pp. 336–345, Dec 1992.

[82] D. Das Sharma and D. K. Pradhan, "*Submesh allocation in mesh multicomputers using busy-list: a best-fit approach with complete recognition capability*". *J. Parallel & Distributed Comput.* **36(2)**, pp. 106–118, Aug 1996.

[83] D. De Paoli, A. Goscinski, M. Hobbs, and P. Joyce, "*Performance comparison of process migration with remote process creation mechanisms in RHODOS*". In 16th *Intl. Conf. Distributed Comput. Syst.*, pp. 554–561, May 1996.

[84] X. Deng and P. Dymond, "*On multiprocessor system scheduling*". In 8th *Symp. Parallel Algorithms & Architectures*, pp. 82–88, Jun 1996.

[85] N. Deo and S. Prasad, "*Parallel heap: an optimal parallel priority queue*". *J. Supercomput.* **6(1)**, pp. 87–98, 1992.

[86] M. Devarakonda and A. Mukherjee, "*Issues in implementation of cache-affinity scheduling*". In *Proc. Winter USENIX Technical Conf.*, pp. 345–357, Jan 1992.

[87] M. V. Devarakonda and R. K. Iyer, "*Predictability of process resource usage: a measurement-based study on UNIX*". *IEEE Trans. Softw. Eng.* **15(12)**, pp. 1579–1586, Dec 1989.

[88] D. DeWitt and J. Gray, "*Parallel database systems: the future of high performance database systems*". *Comm. ACM* **35(6)**, pp. 85–98, Jun 1992.

[89] S. R. Dickey and R. Kenner, "*Hardware combining and scalability*". In 4th *Symp. Parallel Algorithms & Architectures*, pp. 296–305, Jun 1992.

[90] R. T. Dimpsey and R. K. Iyer, "*Modeling and measuring multiprogramming and system overheads on a shared memory multiprocessor: case study*". *J. Parallel & Distributed Comput.* **12(4)**, pp. 402–414, Aug 1991.

[91] R. T. Dimpsey and R. K. Iyer, "*Performance degradation due to multiprogramming and system overheads in real workloads: case study on a shared memory multiprocessor*". In *Intl. Conf. Supercomputing*, pp. 227–238, Jun 1990.

[92] J. Ding and L. N. Bhuyan, "*An adaptive submesh allocation strategy for two-dimensional mesh connected systems*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 193–200, Aug 1993.

[93] A. B. Downey, "*Using queue time predictions for processor allocation*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 35–57, Springer Verlag, 1997. Lecture Notes in Computer Science Vol. 1291.

[94] K. M. Dragon and J. L. Gustafson, "*A low-cost hypercube load-balancing algorithm*". In 4th *Conf. Hypercubes, Concurrent Comput., & Appl.*, pp. 583–589, Mar 1989.

[95] M. Dubois, C. Scheurich, and F. A. Briggs, "*Synchronization, coherence, and event ordering in multiprocessors*". *Computer* **21(2)**, pp. 9–21, Feb 1988.

[96] D. W. Duke, T. P. Green, and J. L. Pasko, "*Research toward a heterogeneous networked computing cluster: the Distributed Queueing System version 3.0*". Mar 1994. Anonymous ftp `ftp.scri.fsu.edu:/pub/DQS/DQS_3_1_1.tar.Z`.

[97] K. Dussa, K. Carlson, L. Dowdy, and K-H. Park, "*Dynamic partitioning in a transputer environment*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 203–213, May 1990.

[98] A. Dusseau, R. H. Arpaci, and D. E. Culler, "*Effective distributed scheduling of parallel workloads*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 25–36, May 1996.

[99] S. Dutt and J. P. Hayes, "*Subcube allocation in hypercube computers*". *IEEE Trans. Comput.* **40(3)**, pp. 341–352, Mar 1991.

[100] B. Duzett and R. Buck, "*An overview of the nCUBE 3 supercomputer*". In 4th *Symp. Frontiers Massively Parallel Comput.*, pp. 458–464, Oct 1992.

[101] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "*Adaptive load sharing in homogeneous distributed systems*". *IEEE Trans. Softw. Eng.* **SE-12(5)**, pp. 662–675, May 1986.

[102] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "*The limited performance benefits of migrating active processes for load sharing*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 63–72, May 1988.

[103] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "*Speedup versus efficiency in parallel systems*". *IEEE Trans. Comput.* **38(3)**, pp. 408–423, Mar 1989.

[104] J. Edler, A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. J. Teller, and J. Wilson, "*Issues related to MIMD shared-memory computers: the NYU Ultracomputer approach*". In 12th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 126–135, 1985.

[105] J. Edler, A. Gottlieb, and J. Lipkis, "*Considerations for massively parallel UNIX systems on the NYU Ultracomputer and IBM RP3*". In *EUUG (European UNIX system User Group) Autumn '86 Conf. Proc.*, pp. 383–403, Sep 1986. Another version appeared in *Proc. Winter USENIX Technical Conf.*, pp. 193–210, Jan 1986.

[106] J. Edler, J. Lipkis, and E. Schonberg, "*Process management for highly parallel UNIX systems*". In *Proc. Workshop on UNIX and Supercomputers*, pp. 1–18, USENIX, Sep 1988.

[107] K. Efe and V. Krishnamoorthy, "*Optimal scheduling of compute-intensive tasks on a network of workstations*". *IEEE Trans. Parallel & Distributed Syst.* **6(6)**, pp. 668–673, Jun 1995.

[108] K. Efe and M. A. Schaar, "*Performance of co-scheduling on a network of workstations*". In 13th *Intl. Conf. Distributed Comput. Syst.*, pp. 525–531, May 1993.

[109] K. Ekanadham, J. Moreira, and V. K. Naik, "*Application oriented resource management on large scale parallel systems*". In *Proc. ICPP Workshop Challanges for Parallel Processing*, pp. 56–63, Aug 1995.

[110] P. Emrath, "*Xylem: an operating system for the Cedar multiprocessor*". *IEEE Software* **2(4)**, pp. 30–37, Jul 1985.

[111] P. A. Emrath, M. S. Anderson, R. R. Barton, and R. E. McGrath, "*The Xylem operating system*". In *Intl. Conf. Parallel Processing*, vol. I, pp. 67–70, Aug 1991.

[112] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne, "*A worldwide flock of condors: load sharing among workstation clusters*". *Future Generation Comput. Syst.* **12(1)**, pp. 53–65, May 1996.

[113] M. J. Feeley, B. N. Bershad, J. S. Chase, and H. M. Levy, "*Dynamic node reconfiguration in a parallel-distributed environment*". In 3rd *Symp. Principles & Practice of Parallel Programming*, pp. 114–121, Apr 1991.

[114] D. G. Feitelson, "*Memory usage in the LANL CM-5 workload*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 78–94, Springer Verlag, 1997. Lecture Notes in Computer Science Vol. 1291.

[115] D. G. Feitelson, "*Packing schemes for gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 89–110, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[116] D. G. Feitelson, Y. Ben-Asher, M. Ben Ezra, I. Exman, L. Picherski, L. Rudolph, and D. Zernik, "*Issues in run-time support for tightly-coupled parallel processing*". In 3rd *Symp. Experiences with Distributed & Multiprocessor Syst.*, pp. 27–42, USENIX, Mar 1992.

[117] D. G. Feitelson and M. A. Jette, "*Improved utilization and responsiveness with gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 238–261, Springer Verlag, 1997. Lecture Notes in Computer Science Vol. 1291.

[118] D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[119] D. G. Feitelson and L. Rudolph, "*Coscheduling based on runtime identification of activity working sets*". *Intl. J. Parallel Programming* **23(2)**, pp. 135–160, Apr 1995.

[120] D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.

[121] D. G. Feitelson and L. Rudolph, "*Evaluation of design choices for gang scheduling using distributed hierarchical control*". *J. Parallel & Distributed Comput.* **35(1)**, pp. 18–34, May 1996.

[122] D. G. Feitelson and L. Rudolph, "*Gang scheduling performance benefits for fine-grain synchronization*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 306–318, Dec 1992.

[123] D. G. Feitelson and L. Rudolph, "*Toward convergence in job schedulers for parallel supercomputers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–26, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[124] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "*Theory and practice in parallel job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–34, Springer Verlag, 1997. Lecture Notes in Computer Science Vol. 1291.

[125] D. Ferguson, Y. Yemini, and C. Nikolaou, "*Microeconomic algorithms for load balancing in distributed computer systems*". In 8th *Intl. Conf. Distributed Comput. Syst.*, pp. 491–499, Jun 1988.

[126] F. Ferstl, "*Job- and resource-management systems in heterogeneous clusters*". *Future Generation Comput. Syst.* **12(1)**, pp. 39–51, May 1996.

[127] R. A. Finkel, *An Operating Systems Vade Mecum*. Prentice-Hall Inc., 2nd ed., 1988.

[128] H. P. Flatt and K. Kennedy, "*Performance of parallel processors*". *Parallel Comput.* **12(1)**, pp. 1–20, 1989.

[129] M. J. Flynn, "*Very high-speed computing systems*". *Proc. IEEE* **54(12)**, pp. 1901–1909, Dec 1966.

[130] H. Franke, P. Pattnaik, and L. Rudolph, "*Gang scheduling for highly efficient distributed multiprocessor systems*". In 6th *Symp. Frontiers Massively Parallel Comput.*, pp. 1–9, Oct 1996.

[131] E. Freudenthal and A. Gottlieb, "*Process coordination with fetch-and-increment*". In 4th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 260–268, Apr 1991.

[132] M. Furtney, "*Parallel processing at Cray Research, Inc.*". In *Software for Parallel Computers*, R. H. Perrott (ed.), pp. 133–154, Chapman & Hall, 1992.

[133] R. A. Gagliano, M. D. Fraser, and M. E. Schaefer, "*Auction allocation of computing resources*". *Comm. ACM* **38(6)**, pp. 88–102, Jun 1995.

[134] J. Gehring and A. Reinefeld, "*MARS—a framework for minimizing the job execution time in a metacomputing environment*". *Future Generation Comput. Syst.* **12(1)**, pp. 87–99, May 1996.

[135] G. A. Geist and V. S. Sunderam, "*The evolution of the PVM concurrent computing system*". In 38th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 549–557, Feb 1993.

[136] D. Gelernter and D. Kaminsky, "*Supercomputing out of recycled garbage: preliminary experience with Piranha*". In *Intl. Conf. Supercomputing*, pp. 417–427, Jul 1992.

[137] A. Gerasoulis and T. Yang, "*A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 276–291, Dec 1992.

[138] A. Ghafoor and J. Yang, "*A distributed heterogeneous supercomputing management system*". *Computer* **26(6)**, pp. 78–86, Jun 1993.

[139] D. Ghosal, G. Serazzi, and S. K. Tripathi, "*The processor working set and its use in scheduling multiprocessor systems*". *IEEE Trans. Softw. Eng.* **17(5)**, pp. 443–453, May 1991.

[140] R. Gibbons, "*A historical application profiler for use by parallel schedulers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 58–77, Springer Verlag, 1997. Lecture Notes in Computer Science Vol. 1291.

[141] B. Goldberg and P. Hudak, "*Alfalfa: distributed graph reduction on a hypercube multiprocessor*". In *Proc. Workshop Graph reduction*, pp. 94–113, Springer Verlag, Sep 1986. Lecture Notes in Computer science Vol. 279.

[142] M. J. Gonzalez, Jr., "*Deterministic processor scheduling*". *ACM Comput. Surv.* **9(3)**, pp. 173–204, Sep 1977.

[143] J. R. Goodman, M. K. Vernon, and P. J. Woest, "*Efficient synchronization primitives for large-scale cache-coherent multiprocessors*". In 3rd *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 64–75, Apr 1989.

[144] B. Gorda and R. Wolski, "*Time sharing massively parallel machines*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 214–217, Aug 1995.

[145] B. C. Gorda and E. D. Brooks III, *Gang Scheduling a Parallel Machine.* Technical Report UCRL-JC-107020, Lawrence Livermore National Laboratory, Dec 1991.

[146] A. Gottlieb, "*Avoiding serial bottlenecks in ultraparallel MIMD computers*". In 28th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 354–359, 1984.

[147] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "*The NYU Ultracomputer — designing an MIMD shared memory parallel computer*". *IEEE Trans. Comput.* **C-32(2)**, pp. 175–189, Feb 1983.

[148] A. Gottlieb, B. Lubachevsky, and L. Rudolph, "*Basic techniques for the efficient coordination of very large numbers of cooperating sequential processes*". *ACM Trans. Prog. Lang. & Syst.* **5(2)**, pp. 164–189, Apr 1983.

[149] G. Graunke and S. Thakkar, "*Synchronization algorithms for shared-memory multiprocessors*". *Computer* **23(6)**, pp. 60–69, Jun 1990.

[150] A. S. Grimshaw, J. B. Weissman, E. A. West, and E. C. Loyot, Jr., "*Metasystems: an approach combining parallel processing and heterogeneous distributed computing systems*". *J. Parallel & Distributed Comput.* **21(3)**, pp. 257–270, Jun 1994.

[151] D. C. Grunwald, B. A. A. Nazief, and D. A. Reed, "*Empirical comparison of heuristic load distribution in point-to-point multicomputer networks*". In 5th *Distributed Memory Comput. Conf.*, pp. 984–993, 1990.

[152] A. Gupta, A. Tucker, and S. Urushibara, "*The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 120–132, May 1991.

[153] A. K. Gupta and A. G. Photiou, "*Load balanced priority queues on distributed memory machines*". In 6th *Parallel Arch. & Lang. Europe*, pp. 689–700, Springer-Verlag, Jul 1994. Lecture Notes in Computer Science Vol. 817.

[154] R. H. Halstead, Jr. and S. A. Ward, "*The MuNet: a scalable decentralized architecture for parallel computation*". In 7th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 139–145, May 1980.

[155] M. Harchol-Balter and A. B. Downey, "*Exploiting process lifetime distributions for dynamic load balancing*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 13–24, May 1996.

[156] K. Harty and D. R. Cheriton, "*Application-controlled physical memory using external page-cache management*". In 5th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 187–197, Sep 1992.

[157] P. J. Hatcher, M. J. Quinn, A. J. Lapadula, B. K. Seevers, R. J. Anderson, and R. R. Jones, "*Data-parallel programming on MIMD computers*". *IEEE Trans. Parallel & Distributed Syst.* **2(3)**, pp. 377–383, Jul 1991.

[158] J. P. Hayes, T. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "*A microprocessor-based hypercube supercomputer*". *IEEE Micro* **6(5)**, pp. 6–17, Oct 1986.

[159] F. Hemery, D. Lazure, E. Delattre, and J-F. Mehaut, "*An analysis of communication and multiprogramming in the Helios operating system*". *Microprocessing & Microprogramming* **32(1–5)**, pp. 137–144, Aug 1991.

[160] R. L. Henderson, "*Job scheduling under the portable batch system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 279–294, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[161] G. J. Henry, "*The fair share scheduler*". *AT&T Bell Labs Tech. J.* **63(8, part 2)**, pp. 1845–1857, Oct 1984.

[162] M. Herlihy, B-H. Lim, and N. Shavit, "*Low contention load balancing on large-scale multiprocessors*". In 4th *Symp. Parallel Algorithms & Architectures*, pp. 219–227, Jun 1992.

[163] A. J. G. Hey, "*Supercomputing with transputers - past, present and future*". In *Intl. Conf. Supercomputing*, pp. 479–489, Jun 1990.

[164] M. D. Hill and J. R. Larus, "*Cache considerations for multiprocessor programmers*". *Comm. ACM* **33(8)**, pp. 97–102, Aug 1990.

[165] R. Hofman and W. G. Vree, "*Distributed hierarchical scheduling with explicit grain size control*". *Future Generation Comput. Syst.* **8(1-3)**, pp. 111–119, Jul 1992.

[166] F. Hofmann, M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand, C-U. Linster, T. Thiel, and S. Turowski, "*MEMSY: a modular expandable multiprocessor system*". In *Parallel Computer Architectures*, A. Bode and M. Dal Cin (eds.), pp. 15–30, Springer Verlag, 1993. Lecture Notes in Computer Science Vol. 732.

[167] J. W. Hong, M. A. Bauer, and J. M. Bennett, "*Integration of the directory service in distributed system management*". In *Intl. Conf. Parallel & Distributed Syst.*, pp. 142–149, Dec 1992.

[168] A. Hori et al., "*Time space sharing scheduling and architectural support*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 92–105, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[169] A. Hori, Y. Ishikawa, H. Konaka, M. Maeda, and T. Tomokiyo, "*A scalable time-sharing scheduling for partitionable, distributed memory parallel machines*". In 28th *Hawaii Intl. Conf. System Sciences*, vol. II, pp. 173–182, Jan 1995.

[170] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda, "*Implementation of gang-scheduling on workstation cluster*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 126–139, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[171] S. Hotovy, "*Workload evolution on the Cornell Theory Center IBM SP2*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 27–40, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[172] C-C. Hui and S. T. Chanson, "*A hydro-dynamic approach to heterogeneous dynamic load balancing in a network of computers*". In *Intl. Conf. Parallel Processing*, vol. III, pp. 140–147, Aug 1996.

[173] S. F. Hummel, E. Schonberg, and L. E. Flynn, "*Factoring: a method for scheduling parallel loops*". *Comm. ACM* **35(8)**, pp. 90–101, Aug 1992.

[174] INMOS Ltd., *Occam Programming Manual.* Prentice-Hall, 1984.

[175] Intel Corp., *iPSC/860 Multi-User Accounting, Control, and Scheduling Utilities Manual.* Order number 312261-002, May 1992.

[176] M. Jeng and H. J. Siegel, "*A distributed management scheme for partitionable parallel computers*". *IEEE Trans. Parallel & Distributed Syst.* **1(1)**, pp. 120–126, Jan 1990.

[177] D. S. Johnson, "*The NP-completeness column: an ongoing guide*". *J. Algorit.* **4(2)**, pp. 189–203, Jun 1983.

[178] D. W. Jones, "*Concurrent operations on priority queues*". *Comm. ACM* **32(1)**, pp. 132–137, Jan 1989.

[179] J. P. Jones and C. Brickell, *Second Evaluation of Job Queueing/Scheduling Software: Phase 1 Report.* Technical Report NAS-97-013, NAS High Performance Processing Group, NASA Ames Research Center, Jun 1997.

[180] H. F. Jordan, "*Experience with pipelined multiple instruction streams*". *Proc. IEEE* **72(1)**, pp. 113–123, Jan 1984.

[181] B. S. Joshi, S. H. Hosseini, and K. Vairavan, "*A methodology for evaluating load balancing algorithms*". In 2nd *Intl. Symp. High Performance Distributed Computing*, pp. 216–223, Jul 1993.

[182] B. U. Kahle, W. A. Nesheim, and M. Isman, "*Unix and the Connection Machine operating system*". In *Proc. Workshop on UNIX and Supercomputers*, pp. 93–107, USENIX, Sep 1988.

[183] D. D. Kandlur, D. L. Kiskis, and K. G. Shin, "*HARTOS: a distributed real-time operating system*". *Operating Systems Rev.* **23(3)**, pp. 72–89, Jul 1989.

[184] R. M. Karp and V. Ramachandran, "*Parallel algorithms for shared-memory machines*". In *Handbook of Theoretical Computer Science. Vol A: Algorithms and Complexity*, J. van Leeuwen (ed.), pp. 869–941, Elsevier/MIT Press, 1990.

[185] R. M. Keller and F. C. H. Lin, "*Simulated performance of a reduction-based multiprocessor*". *Computer* **17(7)**, pp. 70–82, Jul 1984.

[186] R. M. Keller, G. Lindstrom, and S. Patil, "*A loosely-coupled applicative multi-processing system*". In *AFIPS Natl. Comput. Conf.*, vol. 48, pp. 613–622, Jun 1979.

[187] J. Kim, C. R. Das, and W. Lin, "*A top-down processor allocation scheme for hypercube computers*". *IEEE Trans. Parallel & Distributed Syst.* **2(1)**, pp. 20–30, Jan 1991.

[188] D. Klappholz and H-C. Park, "*Parallelized process scheduling for a tightly-coupled MIMD machine*". In *Intl. Conf. Parallel Processing*, pp. 315–321, Aug 1984.

[189] L. Kleinrock, "*Power and deterministic rules of thumb for probabilistic problems in computer communications*". In *Intl. Conf. Communications*, vol. 3, pp. 43.1.1–43.1.10, Jun 1979.

[190] L. Kleinrock and J-H. Huang, "*On parallel processing systems: Amdahl's law generalized and some results on optimal design*". *IEEE Trans. Softw. Eng.* **18(5)**, pp. 434–447, May 1992.

[191] L. Kleinrock and W. Korfhage, "*Collecting unused processing capacity: an analysis of transient distributed systems*". *IEEE Trans. Parallel & Distributed Syst.* **4(5)**, pp. 535–546, May 1993.

[192] K. C. Knowlton, "*A fast storage allocator*". *Comm. ACM* **8(10)**, pp. 623–625, Oct 1965.

[193] J. Konicek et al., "*The organization of the Cedar system*". In *Intl. Conf. Parallel Processing*, vol. I, pp. 49–56, Aug 1991.

[194] J. S. Kowalik (ed.), *Parallel MIMD Computation: The HEP Supercomputer and its Applications.* MIT Press, 1985.

[195] S. Krakowiak, *Principles of Operating Systems.* MIT Press, 1988.

[196] W. T. C. Kramer and J. M. Craw, "*Effective use of Cray supercomputers*". In *Supercomputing '89*, pp. 721–731, Nov 1989.

[197] O. Kremien, J. Kramer, and J. Magee, "*Scalable, adaptive load sharing for distributed systems*". *IEEE Parallel & Distributed Technology* **1(3)**, pp. 62–70, Aug 1993.

[198] A. W. Krings, R. M. Kieckhafer, and J. S. Deogun, "*Inherently stable real-time priority list dispachers*". *IEEE Parallel & Distributed Technology* **2(4)**, pp. 49–59, winter 1994.

[199] R. Krishnamurti and E. Ma, "*An approximation algorithm for scheduling tasks on varying partition sizes in partitionable multiprocessor systems*". *IEEE Trans. Comput.* **41(12)**, pp. 1572–1579, Dec 1992.

[200] P. Krueger and D. Babbar, "*The effect of precedence and priority constraints on the performance of scan scheduling for hypercube multiprocessors*". *J. Parallel & Distributed Comput.* **39(2)**, pp. 95–104, Dec 1996.

[201] P. Krueger and R. Chawla, "*The stealth distributed scheduler*". In 11th *Intl. Conf. Distributed Comput. Syst.*, pp. 336–343, May 1991.

[202] P. Krueger, T-H. Lai, and V. A. Dixit-Radiya, "*Job scheduling is more important than processor allocation for hypercube computers*". *IEEE Trans. Parallel & Distributed Syst.* **5(5)**, pp. 488–497, May 1994.

[203] P. Krueger, T-H. Lai, and V. A. Radiya, "*Processor allocation vs. job scheduling on hypercube computers*". In 11th *Intl. Conf. Distributed Comput. Syst.*, pp. 394–401, May 1991.

[204] P. Krueger and M. Livny, "*A comparison of preemptive and non-preemptive load distributing*". In 8th *Intl. Conf. Distributed Comput. Syst.*, pp. 123–130, Jun 1988.

[205] P. Krueger and M. Livny, "*The diverse objectives of distributed scheduling policies*". In 7th *Intl. Conf. Distributed Comput. Syst.*, pp. 242–248, Sep 1987.

[206] D. J. Kuck, "*On the speedup and cost of parallel computation*". In *The Complexity of Computational Problem Solving*, R. S. Anderseen and R. P. Brent (eds.), pp. 63–78, University of Queensland Press, St. Lucia, Queensland, Australia, 1976.

[207] M. Kumar, "*Measuring parallelism in computation-intensive scientific/engineering applications*". *IEEE Trans. Comput.* **37(9)**, pp. 1088–1098, Sep 1988.

[208] T. Kunz, "*The influence of different workload descriptions on a heuristic load balancing scheme*". *IEEE Trans. Softw. Eng.* **17(7)**, pp. 725–730, Jul 1991.

[209] R. N. Lagerstrom and S. K. Gipp, "*PScheD: political scheduling on the CRAY T3E*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 117–139, Springer Verlag, 1997. Lecture Notes in Computer Science Vol. 1291.

[210] T-H. Lai and S. Sahni, "*Anomalies in parallel branch-and-bound algorithms*". *Comm. ACM* **27(6)**, pp. 594–602, Jun 1984.

[211] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "*The cache performance and optimizations of blocked algorithms*". In 4th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 63–74, Apr 1991.

[212] R. P. LaRowe, Jr., C. S. Ellis, and L. S. Kaplan, "*The robustness of NUMA memory management*". In 13th *Symp. Operating Systems Principles*, pp. 137–151, Oct 1991.

[213] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph, "*Implications of I/O for gang scheduled workloads*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 215–237, Springer Verlag, 1997. Lecture Notes in Computer Science Vol. 1291.

[214] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong-Chan, S-W. Yang, and R. Zak, "*The network architecture of the Connection Machine CM-5*". *J. Parallel & Distributed Comput.* **33(2)**, pp. 145–158, Mar 1996.

[215] S. T. Leutenegger and X-H. Sun, "*Distributed computing feasibility in a non-dedicated homogeneous distributed system*". In *Supercomputing '93*, pp. 143–152, Nov 1993.

[216] S. T. Leutenegger and M. K. Vernon, *Multiprogrammed Multiprocessor Scheduling Issues.* Research Report RC 17642 (#77699), IBM T. J. Watson Research Center, Nov 1992.

[217] S. T. Leutenegger and M. K. Vernon, "*The performance of multiprogrammed multiprocessor scheduling policies*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 226–236, May 1990.

[218] K. Li and K-H. Cheng, "*Job scheduling in a partitionable mesh using a two-dimensional buddy system partitioning scheme*". *IEEE Trans. Parallel & Distributed Syst.* **2(4)**, pp. 413–422, Oct 1991.

[219] K. Li and K-H. Cheng, "*A two-dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system*". *J. Parallel & Distributed Comput.* **12(1)**, pp. 79–83, May 1991.

[220] K. Li, J. F. Naughton, and J. S. Plank, "*An efficient checkpointing method for multicomputers with wormhole routing*". *Intl. J. Parallel Programming* **20(3)**, pp. 159–180, Jun 1991.

[221] D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 295–303, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[222] F. C. H. Lin and R. M. Keller, "*The gradient model load balancing method*". *IEEE Trans. Softw. Eng.* **SE-13(1)**, pp. 32–38, Jan 1987.

[223] M. J. Litzkow, M. Livny, and M. W. Mutka, "*Condor - a hunter of idle workstations*". In 8th *Intl. Conf. Distributed Comput. Syst.*, pp. 104–111, Jun 1988.

[224] W. Liu, V. Lo, K. Windisch, and B. Nitzberg, "*Non-contiguous processor allocation algorithms for distributed memory multicomputers*". In *Supercomputing '94*, pp. 227–236, Nov 1994.

[225] S-P. Lo and V. D. Gligor, "*A comparative analysis of multiprocessor scheduling algorithms*". In 7th *Intl. Conf. Distributed Comput. Syst.*, pp. 356–363, Sep 1987.

[226] V. M. Lo, "*Heuristic algorithms for task assignment in distributed systems*". *IEEE Trans. Comput.* **37(11)**, pp. 1384–1397, Nov 1988.

[227] D. B. Loveman, "*High Performance Fortran*". *IEEE Parallel & Distributed Technology* **1(1)**, pp. 25–42, Feb 1993.

[228] Y-W. Ma and R. Krishnamurti, "*The architecture of REPLICA — a special-purpose computer system for active multi-sensory perception of 3-dimensional objects*". In 11th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 30–37, Jun 1984.

[229] S. Majumdar, D. L. Eager, and R. B. Bunt, "*Characterisation of programs for scheduling in multiprogrammed parallel systems*". *Performance Evaluation* **13(2)**, pp. 109–130, 1991.

[230] S. Majumdar, D. L. Eager, and R. B. Bunt, "*Scheduling in multiprogrammed parallel systems*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 104–113, May 1988.

[231] T. W. Malone, R. E. Fikes, K. R. Grant, and M. T. Howard, "*Enterprise: a market-like task scheduler for distributed computing environments*". In *The Ecology of Computation*, B. A. Huberman (ed.), pp. 177–205, North-Holland, 1988.

[232] E. P. Markatos and T. J. LeBlanc, "*Load balancing vs. locality management in shared-memory multiprocessors*". In *Intl. Conf. Parallel Processing*, vol. I, pp. 258–265, Aug 1992.

[233] E. P. Markatos and T. J. LeBlanc, "*Using processor affinity in loop scheduling on shared-memory multiprocessors*". *IEEE Trans. Parallel & Distributed Syst.* **5(4)**, pp. 379–400, Apr 1994.

[234] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, "*First-class user-level threads*". In 13th *Symp. Operating Systems Principles*, pp. 110–121, Oct 1991.

[235] T. G. Mattson, D. Scott, and S. Wheat, "*A TeraFLOP supercomputer in 1996: the ASCI TFLOP system*". In 10th *Intl. Parallel Processing Symp.*, pp. 84–93, Apr 1996.

[236] J. Mauney, D. P. Agrawal, Y. K. Choe, E. A. Harcourt, S. Kim, and W. J. Staats, "*Computational models and resource allocation for supercomputers*". *Proc. IEEE* **77(12)**, pp. 1859–1874, Dec 1989.

[237] D. May, R. Shepherd, and C. Keane, "*Communicating process architecture: Transputers and Occam*". In *Future Parallel Computers*, P. Treleaven and M. Vanneschi (eds.), pp. 35–81, Springer-Verlag, 1987. Lecture Notes on Computer Science Vol. 272.

[238] C. McCann, R. Vaswani, and J. Zahorjan, "*A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors*". *ACM Trans. Comput. Syst.* **11(2)**, pp. 146–178, May 1993.

[239] C. McCann and J. Zahorjan, "*Processor allocation policies for message passing parallel computers*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 19–32, May 1994.

[240] C. McCann and J. Zahorjan, "*Scheduling memory constrained jobs on distributed memory parallel computers*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 208–219, May 1995.

[241] B. E. Melhart, C. A. Morgenstern, and T. Nute, "*A compendium of processor allocation strategies for two-dimensional mesh connected systems*". *Concurrency — Pract. & Exp.* **7(5)**, pp. 497–514, Aug 1995.

[242] J. M. Mellor-Crummey and M. L. Scott, "*Synchronization without contention*". In 4th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 269–278, Apr 1991.

[243] P. Messina, "*The Concurrent Supercomputing Consortium: year 1*". *IEEE Parallel & Distributed Technology* **1(1)**, pp. 9–16, Feb 1993.

[244] D. S. Milojičić, D. L. Black, and S. Sears, "*Operating system support for concurrent remote task creation*". In 9th *Intl. Parallel Processing Symp.*, pp. 486–492, Apr 1995.

[245] D. Min and M. W. Mutka, "*Effects of job size irregularity on the dynamic resource scheduling of a 2-D mesh multicomputer*". In 5th *Parallel Arch. & Lang. Europe*, pp. 476–487, Jun 1993. Lecture Notes in Computer Science Vol. 694.

[246] D. Min and M. W. Mutka, "*A model for analyzing interaction in 2-D mesh wormhole-routed multicomputers*". *Parallel Computing* **22(5)**, pp. 675–699, Aug 1996.

[247] R. Mirchandaney, D. Towsley, and J. A. Stankovic, "*Adaptive load sharing in heterogeneous distributed systems*". *J. Parallel & Distributed Comput.* **9(4)**, pp. 331–346, Aug 1990.

[248] K. Miura, M. Takamura, Y. Sakamoto, and S. Okada, "*Overview of the Fujitsu VPP500 supercomputer*". In 38th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 128–130, Feb 1993.

[249] J. C. Mogul and A. Borg, "*The effect of context switches on cache performance*". In 4th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 75–84, Apr 1991.

[250] P. Mohapatra, C. Yu, and C. R. Das, "*A lazy scheduling scheme for hypercube computers*". *J. Parallel & Distributed Comput.* **27(1)**, pp. 26–37, May 1995.

[251] S. Q. Moore and L. M. Ni, "*The effects of network contention on processor allocation atrategies*". In 10th *Intl. Parallel Processing Symp.*, pp. 268–273, Apr 1996.

[252] J. E. Moreira and C. D. Polychronopoulos, "*Autoscheduling in a distributed shared-memory environment*". In *Languages & Compilers for Parallel Comput.*, K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (eds.), pp. 453–469, Springer-Verlag, Aug 1994. Lecture Notes in Computer Science Vol. 892.

[253] C. Morgenstern, "*Methods for precise submesh allocation*". *Scientific Computing* **3(4)**, pp. 353–364, Winter 1994.

[254] C. Morgenstern and P. Fouque, "*Efficient submesh allocation using interval sets*". In 27th *Hawaii Intl. Conf. System Sciences*, vol. II, pp. 493–501, Jan 1994.

[255] R. Mraz, "*Reducing the variance of point-to-point transfers for parallel real-time programs*". *IEEE Parallel & Distributed Technology* **2(4)**, pp. 20–31, Winter 1994.

[256] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "*Amoeba: a distributed operating system for the 1990s*". *Computer* **23(5)**, pp. 44–53, May 1990.

[257] F. J. Muniz and E. J. Zaluska, "*Parallel load balancing: an extension to the gradient model*". *Parallel Comput.* **21(2)**, pp. 287–301, Feb 1995.

[258] A. J. Musciano and T. L. Sterling, "*Efficient dynamic scheduling of medium-grained tasks for general purpose parallel processing*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 166–175, Aug 1988.

[259] V. K. Naik, S. K. Setia, and M. S. Squillante, "*Performance analysis of job scheduling policies in parallel supercomputing environments*". In *Supercomputing '93*, pp. 824–833, Nov 1993.

[260] V. K. Naik, S. K. Setia, and M. S. Squillante, "*Scheduling of large scientific applications on distributed memory multiprocessor systems*". In 6th *SIAM Conf. Parallel Processing for Scientific Computing*, vol. II, pp. 913–922, Mar 1993.

[261] C. Natarajan, S. Sharma, and R. K. Iyer, "*Impact of loop granularity and self-preemption on the performance of loop parallel applications on a multiprogrammed shared-memory multiprocessor*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 174–178, Aug 1994.

[262] C. Natarajan, S. Sharma, and R. K. Iyer, "*Measurement-based characterization of global memory and network contention, operating system and parallelization overheads: case study on a shared-memory multiprocessor*". In 21st *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 71–80, Apr 1994.

[263] R. D. Nelson and M. S. Squillante, "*Analysis of contention in multiprocessor scheduling*". In *Performance '90*, P. J. B. King, I. Mitrani, and R. J. Pooley (eds.), pp. 391–405, North Holland, 1990.

[264] B. C. Neuman and S. Rao, "*The prospero resource manager: a scalable framework for processor allocation in distributed systems*". *Concurrency — Pract. & Exp.* **6(4)**, pp. 339–355, Jun 1994.

[265] L. M. Ni, C-W. Xu, and T. B. Gendreau, "*A distributed drafting algorithm for load balancing*". *IEEE Trans. Softw. Eng.* **SE-11(10)**, pp. 1153–1161, Oct 1985.

[266] M. G. Norman and P. Thanisch, "*Models of machines and computation for mapping in multicomputers*". *ACM Comput. Surv.* **25(3)**, pp. 263–302, Sep 1993.

[267] S. F. Nugent, "*The iPSC/2 direct-connect communication technology*". In 3rd *Conf. Hypercubes, Concurrent Comput., & Appl.*, vol. I, pp. 51–60, Jan 1988.

[268] J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.

[269] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "*Medusa: an experiment in distributed operating system structure*". *Comm. ACM* **23(2)**, pp. 92–105, Feb 1980.

[270] B. J. Overeinder, P. M. A. Sloot, R. N. Heederik, and L. O. Hertzberger, "*A dynamic load balancing system for parallel cluster computing*". *Future Generation Comput. Syst.* **12(1)**, pp. 101–115, May 1996.

[271] J. F. Palmer, "*The NCUBE family of high performance parallel computer systems*". In 3rd *Conf. Hypercubes, Concurrent Comput., & Appl.*, vol. I, pp. 847–851, Jan 1988.

[272] K-H. Park and L. W. Dowdy, "*Dynamic partitioning of multiprocessor systems*". *Intl. J. Parallel Programming* **18(2)**, pp. 91–120, Apr 1989.

[273] E. W. Parsons and K. C. Sevcik, "*Benefits of speedup knowledge in memory-constrained multiprocessor scheduling*". *Performance Evaluation* **27&28**, pp. 253–272, Oct 1996.

[274] E. W. Parsons and K. C. Sevcik, "*Coordinated allocation of memory and processors in multiprocessors*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 57–67, May 1996.

[275] E. W. Parsons and K. C. Sevcik, "*Implementing multiprocessor scheduling disciplines*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 166–192, Springer Verlag, 1997. Lecture Notes in Computer Science Vol. 1291.

[276] E. W. Parsons and K. C. Sevcik, "*Multiprocessor scheduling for high-variability service time distributions*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 127–145, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[277] Perihelion Software Ltd., *The Helios Parallel Operating System*. Prentice Hall, 1991.

[278] V. G. J. Peris, M. S. Squillante, and V. K. Naik, "*Analysis of the impact of memory in distributed parallel processing systems*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 5–18, May 1994.

[279] J. Peterson and A. Silberschatz, *Operating System Concepts*. Addison-Wesley, 1983.

[280] G. F. Pfister and V. A. Norton, "*"Hot-spot" contention and combining in multistage interconnection networks*". *IEEE Trans. Comput.* **C-34(10)**, pp. 943–948, Oct 1985.

[281] P. Pierce and G. Regnier, "*The Paragon implementation of the NX message passing interface*". In *Scalable High-Performance Comput. Conf.*, pp. 184–190, May 1994.

[282] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "*Plan 9 from Bell Labs*". In *Proc. Summer 1990 UKUUG Conf. (UK Unix User Group)*, pp. 1–9, Jul 1990.

[283] C. D. Polychronopoulos, "*Multiprocessing versus multiprogramming*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 223–230, Aug 1989.

[284] C. D. Polychronopoulos and D. J. Kuck, "*Guided self scheduling: a practical scheduling scheme for parallel supercomputers*". *IEEE Trans. Comput.* **C-36(12)**, pp. 1425–1439, Dec 1987.

[285] J. Pruyne and M. Livny, "*Interfacing Condor and PVM to harness the cycles of workstation clusters*". *Future Generation Comput. Syst.* **12(1)**, pp. 67–85, May 1996.

[286] J. Pruyne and M. Livny, "*Parallel processing on dynamic resources with CARMI*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 259–278, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[287] W. Qiao and L. M. Ni, "*Efficient processor allocation for 3D tori*". In 9th *Intl. Parallel Processing Symp.*, pp. 466–471, Apr 1995.

[288] K. Ramamritham, J. A. Stankovic, and P-F. Shiah, "*Efficient scheduling algorithms for real-time multiprocessor systems*". *IEEE Trans. Parallel & Distributed Syst.* **1(2)**, pp. 184–194, Apr 1990.

[289] K. Ramamritham, J. A. Stankovic, and W. Zhao, "*Distributed scheduling of tasks with deadlines and resouce requirements*". *IEEE Trans. Comput.* **38(8)**, pp. 1110–1123, Aug 1989.

[290] F. Ramme, T. Römke, and K. Kremer, "*A distributed computing center software for the efficient use of parallel computer systems*". In *High-Performance Computing and Networking*, W. Gentzsch and U. Harms (eds.), pp. 129–136, Springer-Verlag, April 1994. Lecture Notes in Computer Science Vol. 797.

[291] V. N. Rao and V. Kumar, "*Concurrent access of priority queues*". *IEEE Trans. Comput.* **37(12)**, pp. 1657–1665, Dec 1988.

[292] R. C. Regis, "*Multiserver queueing models of multiprocessing systems*". *IEEE Trans. Comput.* **C-22(8)**, pp. 736–745, Aug 1973.

[293] M. Richmond and M. Hitchens, "*A new process migration algorithm*". *Op. Sys. Rev.* **31(1)**, pp. 31–42, Jan 1997.

[294] A. Rogers and K. Pingali, "*Process decomposition through locality of reference*". In *Proc. SIGPLAN Conf. Prog. Lang. Design & Implementation*, pp. 69–80, Jun 1989.

[295] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson, "*Robust partitioning schemes of multiprocessor systems*". *Performance Evaluation* **19(2-3)**, pp. 141–165, Mar 1994.

[296] E. Rosti, E. Smirni, G. Serazzi, and L. W. Dowdy, "*Analysis of non-work-conserving processor partitioning policies*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 165–181, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[297] L. Rudolph and Z. Segall, "*Dynamic decentralized cache schemes for MIMD parallel processors*". In 11th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 340–347, Jun 1984.

[298] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal, "*A simple load balancing scheme for task allocation in parallel machines*". In 3rd *Symp. Parallel Algorithms & Architectures*, pp. 237–245, Jul 1991.

[299] S. H. Russ, B. Flachs, J. Robinson, and B. Heckel, "*Hector: automated task allocation for MPI*". In 10th *Intl. Parallel Processing Symp.*, pp. 344–348, Apr 1996.

[300] W. Saphir, L. A. Tanner, and B. Traversat, "*Job management requirements for NAS parallel systems and clusters*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 319–336, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[301] V. Sarkar, "*Determining average program execution times and their variance*". In *Proc. SIGPLAN Conf. Prog. Lang. Design & Implementation*, pp. 298–312, Jun 1989.

[302] C. Scheurich and M. Dubois, "*Dynamic page migration in multiprocessors with distributed global memory*". *IEEE Trans. Comput.* **38(8)**, pp. 1154–1163, Aug 1989.

[303] B. Schnor, "*Dynamic scheduling of parallel applications*". In *Parallel Computing Technologies*, V. Malyshkin (ed.), pp. 109–116, Springer-Verlag, Sep 1995. Lecture Notes in Computer Science Vol. 964.

[304] K. Schwan, B. Blake, W. Bo, and J. Gawkowski, "*Global data and control in multicomputers: operating system primitives and experimentation with a parallel branch-and-bound algorithm*". *Concurrency — Pract. & Exp.* **1(2)**, pp. 191–218, Dec 1989.

[305] M. L. Scott, T. J. LeBlanc, and B. D. Marsh, "*Design rationale for Psyche, a general-purpose multiprocessor operating system*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 255–262, Aug 1988.

[306] M. K. Seager and J. M. Stichnoth, *Simulating the Scheduling of Parallel Supercomputer Applications.* Technical Report UCRL-102059, Lawrence Livermore National Laboratory, Sep 1989.

[307] C. L. Seitz, "*Concurrent architectures*". In *VLSI and Parallel Computation*, R. Suaya and G. Birtwistle (eds.), chap. 1, Morgan Kaufmann Publishers, Inc., 1990.

[308] C. L. Seitz, "*The Cosmic Cube*". *Comm. ACM* **28(1)**, pp. 22–33, Jan 1985.

[309] S. Setia and S. Tripathi, *An Analysis of Several Processor Partitioning Policies for Parallel Computers.* Technical Report CS-TR-2684, University of Maryland, May 1991.

[310] S. K. Setia, "*The interaction between memory allocation and adaptive partitioning in message-passing multicomputers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 146–165, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[311] S. K. Setia, "*Trace-driven analysis of migration-based gang scheduling policies for parallel computers*". In *Intl. Conf. Parallel Processing*, Aug 1997.

[312] S. K. Setia, M. S. Squillante, and S. K. Tripathi, "*Analysis of processor allocation in multiprogrammed, distributed-memory parallel processing systems*". *IEEE Trans. Parallel & Distributed Syst.* **5(4)**, pp. 401–420, Apr 1994.

[313] K. C. Sevcik, "*Application scheduling and processor allocation in multiprogrammed parallel processing systems*". *Performance Evaluation* **19(2-3)**, pp. 107–140, Mar 1994.

[314] K. C. Sevcik, "*Characterization of parallelism in applications and their use in scheduling*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 171–180, May 1989.

[315] C. Severance, R. Enbody, and P. Petersen, "*Managing the overall balance of operating system threads on a multiprocessor using automatic self-allocating threads (ASAT)*". *J. Parallel & Distributed Comput.* **37(1)**, pp. 106–112, Aug 1996.

[316] E. Shamir and E. Upfal, "*A probabilistic approach to the load-sharing problem in distributed systems*". *J. Parallel & Distributed Comput.* **4**, pp. 521–530, 1987.

[317] K. G. Shin and Y-C. Chang, "*Load sharing in distributed real-time systems with state-change broadcasts*". *IEEE Trans. Comput.* **38(8)**, pp. 1124–1142, Aug 1989.

[318] N. G. Shivaratri, P. Krueger, and M. Singhal, "*Load distributing for locally distributed systems*". *Computer* **25(12)**, pp. 33–44, Dec 1992.

[319] W. Shu, "*Adaptive dynamic process scheduling on distributed memory parallel computers*". *Scientific Programming* **3(4)**, pp. 341–352, Winter 1994.

[320] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies.* McGraw-Hill, 2nd ed., 1990.

[321] H. J. Siegel, "*The theory underlying the partitioning of permutation networks*". *IEEE Trans. Comput.* **C-29(9)**, pp. 791–801, Sep 1980.

[322] H. J. Siegel, W. G. Nation, C. P. Kruskal, and L. M. Napolitano, Jr., "*Using the multistage cube network topology in parallel supercomputers*". *Proc. IEEE* **77(12)**, pp. 1932–1953, Dec 1989.

[323] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "*An overview of the PASM parallel processing system*". In *Tutorial: Computer Architecture*, D. D. Gajski, V. M. Milutinović, H. J. Siegel, and B. P. Furht (eds.), pp. 387–407, IEEE Computer Society Press, 1987.

[324] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "*PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition*". *IEEE Trans. Comput.* **C-30(12)**, pp. 934–947, Dec 1981.

[325] B. J. Smith, "*A pipelined, shared resource MIMD computer*". In *Intl. Conf. Parallel Processing*, pp. 6–8, 1978.

[326] P. G. Sobalvarro and W. E. Weihl, "*Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 106–126, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[327] M. S. Squillante, "*On the benefits and limitations of dynamic partitioning in parallel computer systems*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 219–238, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[328] M. S. Squillante and E. D. Lazowska, "*Using processor-cache affinity information in shared-memory multiprocessor scheduling*". *IEEE Trans. Parallel & Distributed Syst.* **4(2)**, pp. 131–143, Feb 1993.

[329] J. A. Stankovic, "*Stability and distributed scheduling algorithms*". *IEEE Trans. Softw. Eng.* **SE-11(10)**, pp. 1141–1152, Oct 1985.

[330] J. A. Stankovic and K. Ramamritham, "*The Spring kernel: a new paradigm for real-time operating systems*". *Operating Systems Rev.* **23(3)**, pp. 54–71, Jul 1989.

[331] P. Steiner, "*Extending multiprogramming to a DMPP*". *Future Generation Comput. Syst.* **8(1-3)**, pp. 93–109, Jul 1992.

[332] R. Subramanian and I. D. Scherson, "*An analysis of diffusive load balancing*". In 6th *Symp. Parallel Algorithms & Architectures*, pp. 220–225, Jun 1994.

[333] V. S. Sunderam, "*PVM: a framework for parallel distributed computing*". *Concurrency — Pract. & Exp.* **2(4)**, pp. 315–339, Dec 1990.

[334] G. S. H. Tan and W-N. Chin, "*Neighbourhood scheduling for a multiprocessor*". In *Intl. Conf. Parallel & Distributed Systems*, pp. 472–479, Dec 1992.

[335] A. S. Tanenbaum, *Distributed Operating Systems.* Prentice Hall, 1995.

[336] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum, "*Experiences with the Amoeba distributed operating system*". *Comm. ACM* **33(12)**, pp. 46–63, Dec 1990.

[337] J. A. Test, "*Multi-processor management in the Concentrix operating system*". In *Proc. Winter USENIX Technical Conf.*, pp. 173–182, Jan 1986.

[338] J. Torrellas, A. Gupta, and J. Hennessy, "*Characterizing the caching and synchronization performance of a multiprocessor operating system*". In 5th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 162–174, Oct 1992.

[339] J. Torrellas, A. Tucker, and A. Gupta, "*Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors*". *J. Parallel & Distributed Comput.* **24(2)**, pp. 139–151, Feb 1995.

[340] A. Trew and G. Wilson (eds.), *Past, Present, Parallel: A Survey of Available Parallel Computing Systems.* Springer-Verlag, 1991.

[341] S. K. Tripathi, G. Serazzi, and D. Ghosal, "*Processor scheduling in multiprocessor systems*". In *Parallel Computation*, H. P. Zima (ed.), pp. 208–225, Springer-Verlag, 1992. Lecture Notes in Computer Science Vol. 591.

[342] A. Tucker and A. Gupta, "*Process control and scheduling issues for multiprogrammed shared-memory multiprocessors*". In 12th *Symp. Operating Systems Principles*, pp. 159–166, Dec 1989.

[343] L. W. Tucker and G. G. Robertson, "*Architecture and applications of the Connection Machine*". *Computer* **21(8)**, pp. 26–38, Aug 1988.

[344] D. L. Tuomenoksa and H. J. Siegel, "*Task scheduling on the PASM parallel processing system*". *IEEE Trans. Softw. Eng.* **SE-11(2)**, pp. 145–157, Feb 1985.

[345] S. W. Turner, L. M. Ni, and B. H. C. Cheng, "*Contention-free 2D-mesh cluster allocation in hypercubes*". *IEEE Trans. Parallel & Distributed Syst.* **44(8)**, pp. 1051–1055, Aug 1995.

[346] J. D. Ullman, "*Complexity of sequencing problems*". In *Computer and Job-Shop Scheduling Theory*, E. G. Coffman, Jr. (ed.), chap. 4, John Wiley & Sons, 1976.

[347] T. Utsumi, M. Ikeda, and M. Takamura, "*Architecture of the VPP500 parallel supercomputer*". In *Supercomputing '94*, pp. 478–487, Nov 1994.

[348] R. Vaswani and J. Zahorjan, "*The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors*". In 13th *Symp. Operating Systems Principles*, pp. 26–40, Oct 1991.

[349] D. F. Vrsalovic, D. P. Siewiorek, Z. Z. Segall, and E. F. Gehringer, "*Performance prediction and calibration for a class of multiprocessors*". *IEEE Trans. Comput.* **37(11)**, pp. 1353–1365, Nov 1988.

[350] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta, "*Spawn: a distributed computational economy*". *IEEE Trans. Softw. Eng.* **18(2)**, pp. 103–117, Feb 1992.

[351] D. W. Walker, "*The design of a message passing interface for distributed memory concurrent computers*". *Parallel Comput.* **20(4)**, pp. 657–673, Apr 1994.

[352] M. Wan, R. Moore, G. Kremenek, and K. Steube, "*A batch scheduler for the Intel Paragon with a non-contiguous node allocation algorithm*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 48–64, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[353] K. Y. Wang and D. C. Marinescu, "*Correlation of the paging activity of individual node programs in the SPMD execution model*". In 28th *Hawaii Intl. Conf. System Sciences*, vol. I, pp. 61–71, Jan 1995.

[354] Y-T. Wang and R. J. T. Morris, "*Load sharing in distributed systems*". *IEEE Trans. Comput.* **C-34(3)**, pp. 204–217, Mar 1985.

[355] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant, "*Flagship: a parallel architecture for declarative programming*". In 15th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 124–130, 1988.

[356] P. Watson, "*The FLAGSHIP parallel machine*". In *Multiprocessor Computer Architectures*, T. J. Fountain and M. J. Shute (eds.), pp. 57–81, North-Holland, 1990.

[357] C. Whitby-Strevens, "*The transputer*". In 12th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 292–300, Jun 1985.

[358] M. H. Willebeek-LeMair and A. P. Reeves, "*Strategies for dynamic load balancing on highly parallel computers*". *IEEE Trans. Parallel & Distributed Syst.* **4(9)**, pp. 979–993, Sep 1993.

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

[360] C-Z. Xu and F. C. M. Lau, "*Optimal parameters for load balancing using the diffusion method in k-ary n-cube networks*". *Inf. Process. Lett.* **47(4)**, pp. 181–187, Sep 1993.

[361] C. Xu, B. Monien, R. Lüling, and F. C. M. Lau, "*An analytical comparison of nearest neighbor algorithms for load balancing in parallel computers*". In 9th *Intl. Parallel Processing Symp.*, pp. 472–479, Apr 1995.

[362] I-L. Yen and F. B. Bastani, "*Robust parallel resource management in shared memory multiprocessor systems*". In 9th *Intl. Parallel Processing Symp.*, pp. 458–465, Apr 1995.

[363] C. Yu and C. R. Das, "*Limit allocation: an efficient processor management scheme for hypercubes*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 143–150, Aug 1994.

[364] K. K. Yue and D. J. Lilja, "*Efficient execution of parallel applications in multiprogrammed multiprocessor systems*". In 10th *Intl. Parallel Processing Symp.*, pp. 448–456, Apr 1996.

[365] K. K. Yue and D. J. Lilja, "*Loop-level process control: an effective processor allocation policy for multiprogrammed shared-memory multiprocessors*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 182–199, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[366] J. Zahorjan and C. McCann, "*Processor scheduling in shared memory multiprocessors*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 214–225, May 1990.

[367] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala, "*An OSF/1 UNIX for massively parallel multicomputers*". In *Proc. Winter USENIX Conf.*, pp. 449–467, Jan 1993.

[368] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "*Utopia: a load sharing facility for large, heterogeneous distributed computer systems*". *Software — Pract. & Exp.* **23(12)**, pp. 1305–1336, Dec 1993.

[369] Y. Zhu, "*Efficient processor allocation strategies for mesh-connected parallel computers*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 328–337, Dec 1992.

[370] Y. Zhu and M. Ahuja, "*On job scheduling on a hypercube*". *IEEE Trans. Parallel & Distributed Syst.* **4(1)**, pp. 62–69, Jan 1993.