Extended Version

Job Scheduling in Multiprogrammed Parallel Systems

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

IBM Research Report RC 19790 (87657), October 1994
Second Revision, 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 **1**

2 A Classification of Multiprogramming Schemes **5**
2.1 Single-Level Scheduling and Two-Level Scheduling 6
2.2 The Two Dimensions of Sharing 7
2.3 The Roots of Divergence 9

3 Partitioning **11**
3.1 Fixed Partitioning 14
3.2 Variable Partitioning According to Requests 14
3.2.1 Combinations of Partitions Based on Powers of Two 15
3.2.2 Flexible Partition Sizes 24
3.2.3 Making Scheduling Decisions 30
3.3 Adaptive Partitioning: Setting the Allocation at Load Time 33
3.4 Dynamic Partitioning: Allocation May Change at Runtime 37
3.5 Processors Allocated Singly 42

4 Independent PEs **43**
4.1 Local Queues 43
4.1.1 Mapping 45
4.1.2 Load balancing 48
4.2 A Global Queue 50
4.2.1 Implementations 50
4.2.2 Making scheduling decisions 55
4.3 Combination of Local and Global Queues 56
4.4 Optimizations for Two-Level Scheduling 57
4.4.1 Chunking from a Global Queue 58
4.4.2 Variations on the Thread Model 60

5 Gang Scheduling **62**
5.1 Definition and Motivation 63
5.2 Implementing Multi-Context-Switching 66
5.3 Gang Scheduling within Predefined Partitions 68
5.4 Gang Scheduling with Dynamic Repartitioning 70
5.5 Theoretical Results 74

6 Implications of the Sharing Scheme **75**
6.1 Interaction with Applications 75
6.2 Impact on System Performance 78
6.3 Individual Users in a Multiuser System 83
6.4 Implementation Issues 88

7 Case Studies and Example Systems **92**
7.1 Batch Queueing Systems 93
7.1.1 The Network Queueing System (NQS) 93
7.1.2 The Portable Batch System (PBS) 94
7.2 Catering for Many Masters 96
7.2.1 Partitions with Different Attributes 96
7.2.2 The Lawrence Livermore Gang Scheduler 100
7.2.3 The Tera MTA 102
7.2.4 The Connection Machine CM-5 102
7.2.5 The Kendall Square Research KSR1 105
7.3 Networks of Workstations 105

8 Conclusions **107**

A Terminology **114**

B The Thread Model **118**

References **122**

Index **167**

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, and a discussion of "threads" is given in Appendix B.

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. However, it should be noted that this is not the only mode in which parallel machines are used.

Many parallel machines are not intended to provide multiprogramming services, because they are perceived as high-performance systems that should be dedicated to production computing. Examples start with many early parallel machines and continue into the present. They include the Illiac IV [45, 77], which was designed to carry out numerical solutions for partial differential equations such as those in weather codes, the Goodyear MPP [50, 470], used by NASA to process satellite images, the IBM GF11 [54, 342], dedicated for a whole year to a single computation involving the masses of eight fundamental particles, in an effort to validate the predictions of quantum chromo-dynamics [93], and the Warp systolic array [24], which is used for the real-time signal processing required in robotic navigation and vision.

The use of dedicated machines is not a privilege of large government agencies and philanthropists in search of basic truths. Businesses as diverse as oil companies, car manufacturers, and wall-street firms are known to employ parallel computers in everyday operations. Typically these machines are dedicated to running the same single application for as long as the machine is in use. In the case of oil companies, this application is oil reservoir modelling based on seismic data. Car and plane manufacturers run computational fluid dynamics applications to evaluate the aerodynamics of new designs. Wall street uses parallel supercomputers for market analysis, based on huge databases of market activity. The advantage of dedicated machines is that the application has complete control over the hardware, with no operating system interference. Thus interprocessor communication can be done with less software layers of protection, leading to higher performance [343, 76].

It is interesting to see how history repeats itself, even after 40 years in a fast-changing industry such as computers. The first uniprocessors were also dedicated to a single application, e.g. census tabulation, and many mainframes remain dedicated to transaction processing or payroll processing to this day. Nevertheless, the idea of batch processing came close on the heels of dedicated machines. Many parallel machines are following in those footsteps, using either signup sheets or automated queueing systems such as NQS to coordinate resource allocation [323]. In parallel systems, the allocation is done in three dimensions: PEs, memory, and time. This allows a high degree of resource utilization to be achieved. For example, a utilization above 80% after factoring in down-time was reported for the experimental Touchstone Delta machine at Caltech [400].

Parallel machines are also taking the next step from batch systems to interactive ones. 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. This includes program development, debugging, testing, and just learning about parallelism. After decades of performing such activities interactively on uniprocessors, most users consider it unacceptable to resort to batch processing on parallel machines.

Some installations, mainly small-scale ones at academic institutions, are even used primarily for interactive use. It is safe to speculate that this will also be the primary use of parallel workstations. Other systems attempt to combine interactive and batch usage modes. For example, it is possible to partition the machine statically into a large batch partition and a smaller interactive partition [400, 422, 606]. Alternatively, it is possible to initially allocate all the resources to batch processing, and preempt PEs and memory in favor of interactive jobs as needed. This leads to the notion of adaptive resource allocation for the batch jobs, using the same techniques as for interactive jobs (e.g. time slicing), but at a much coarser time scale.

This survey is oriented primarily towards issues relating to interactive use. The main consideration is thus shifted from performance, utilization, and throughput to response time and user satisfaction. However, it is practically impossible to draw a line between mechanisms used for batch scheduling and those used in interactive systems. Therefore the full range of schemes developed for multiprogramming on parallel machines are reviewed. This includes both paper designs from academia and practical ones that are commercially available.

The following issues will *not* be surveyed. First, we shall not discuss the classic NP-complete problem of task scheduling a.k.a. the mapping problem [590, 293, 238, 58, 434, 13, 7]. This problem deals with how to schedule a program represented as a graph of tasks with interdependencies on multiple processors. Variants of the problem take granularity considerations and inter-processor communication into account. This problem is relevant to the design of parallel programs that will execute on a given number of processors, because then the overhead associated with runtime scheduling can be avoided [622]. Heuristics for its solution are mentioned in Section 4.1.1. However, it is largely irrelevant to situations where the computation is programmed in a way that does not resemble a task graph, as is often the case for control parallelism and the SPMD[1]indexSPMD model. It is also irrelevant if the availability of PEs is not known in advance, as is the case in multiprogrammed systems. In these cases the scheduling must be done by the parallel operating system, using techniques described in this paper.

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 [478, 477, 548, 525, 300, 35, 326], whereas multiprogramming is more concerned with throughput and fair sharing of resources.

[1]SPMD stand for Single Program Multiple Data [144]. Under this paradigm, the same program is loaded onto all the PEs. Each PE runs a single thread, and they all start execution from the same entry point. However, each PE knows its serial number, operates on data in its local memory, and may branch independently of other PEs.

Likewise, scheduling issues on parallel database machines are not included [75, 159], 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") [233, 109, 247, 480, 226]. 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. This was a design goal in many multiprocessors such as the VAX 11/784, which had four processors, each equivalent to that of a VAX 11/780. But many machines that were designed to execute parallel programs are also used in this mode [464]. This is especially true of small scale bus-based machines such as the Sequent Balance, and also of Cray multiprocessors such as the Cray X-MP, Y-MP, and C-90. This survey does not cover such usage. It deals exclusively with scheduling of parallel programs.

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 more detailed discussion of the thread model is given in Appendix B.

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.

A thread has some *state* information, e.g. its program counter and other registers, and executes within some *context*, e.g. its address space and open files. Switching from one thread to another within the same context requires the state of the running thread to be offloaded from the PE, and the state of the newly scheduled thread to be loaded onto it. Switching from a thread in one context to a thread in another context requires the context information to be changed as well.

The title of this survey refers to *multiprogrammed* systems. We use this term to denote

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. This is a straightforward adaptation of the use of these terms in uniprocessor systems [456].

Scheduling is the decision of which thread of control will execute on a certain PE at a certain time. In some cases the relation of threads to PEs is restricted in some way, e.g. threads belonging to a certain application will only execute on a certain subset of the PEs. If the PEs only execute threads from this application, it implies some sort of *processor allocation* prior to the scheduling. This can also be regarded as a *partitioning* of the system among the applications.

The assignment of threads to PEs may also be further restricted, effectively *mapping* each thread to a specific PE. This can be done by the operating system, based on considerations such as the desire to reduce contention for global data structures. In this case the operating system may later change its mind and change the mapping in the interest of *load balancing*. Alternatively, the mapping of threads to PEs can be done by the application based on knowledge of what the threads do and how they interact. Such mapping depends on the fact that the operating system had already allocated the PEs to the application, and is not going to interfere with their use. Technically, the operating system schedules a single thread on each allocated PE, and the application maps its logical threads onto these system threads. The selection of the next thread to run from those that are ready and memory-resident is called *dispatching*.

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 [322, 456]. 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) [558, 362, 197, 594, 464, 201]. Again, sharing provides users with better access to the system, and — depending on the application — even with interactive response times [518]. Again, this is important both because it enables users to make faster progress towards a solution [574],

and because adequate access for multiple users is often crucial in order to fund costly parallel supercomputers [78, 151].

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 distinction between single-level and two-level scheduling is typically apparent in the operating system interface: either the application creates a set of parallel threads, and then the system is responsible for scheduling them, or the application just requests an allocation of PEs, and manages their use on its own.

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 be numerous and can come 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 [104]. 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 [392]. 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 classifi-

			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.*

cation. 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. This is in contrast with other types of classifications, where some combinations don't make sense, such as the MISD part of Flynn's classification of parallel architectures [214].

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 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 (including the Cray T3E, Connection Machine CM-5, the Intel Paragon, the Meiko CS-2, and SGI multiprocessors). In addition, a number of experimental systems have also been reported in the literature 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 [205, 206]. 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. Nat-

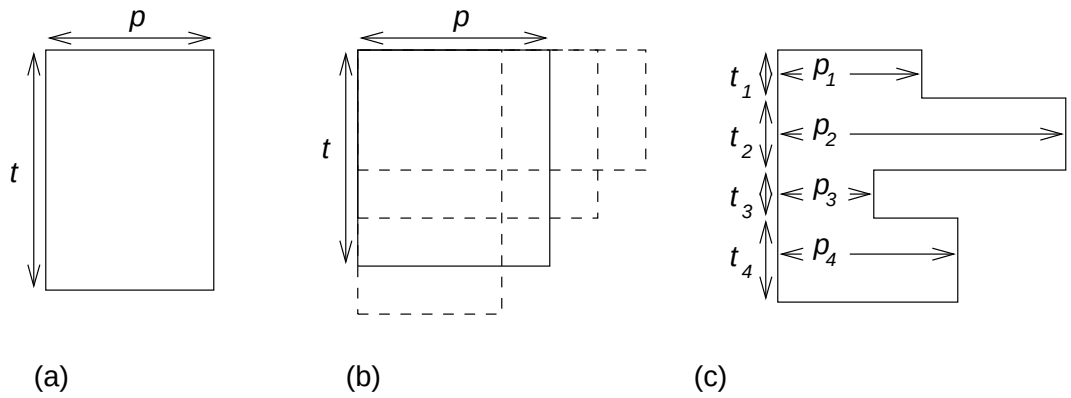


Figure 2: *(a) Rigid jobs define a rectangle in processor-time space. (b) Moldable jobs use one out of a choice of such rectangles. (c) Evolving and malleable jobs both have a profile with a changing number of processors. The difference is that in evolving jobs the changes are initiated by the job, while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

urally, 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 movements. 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 [206].

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) [205]:

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 [581], 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 [633, 580]. 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.

Alternatively, partitioning may be combined with preemption, either on independent PEs within each partition or synchronously across the partition. Systems that provide such combinations are reviewed in other sections. 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[2] 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 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 [381, 489]:

- *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. The architecture might impose restrictions on the ways in which the PEs can be partitioned. The operating system might make partitioning decisions in an oblivious manner, or else it might take system load into account. It might even want to change the partitioning dynamically at runtime.

[2]SIMD stands for Single Instruction-stream Multiple Data-stream, and MIMD stands for Multiple Instruction-stream Multiple Data-stream [214]. 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.

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.*

Applications might require a specific number of PEs, and might even require a specific configuration. On the other hand, applications may be structured so that the number of PEs is set only when the application is loaded, thus allowing the application to run on different sizes of partitions. Applications that run on systems that change the PE allocation at runtime must be able to adjust to whatever number of PEs the system gives them.

We shall classify the partitioning mechanisms into four types, based on how the operating system behaves and the requirements this places on applications [422, 489]: 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 [400, 422]. For example, many commercial parallel 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. Batch jobs are typically identified by the fact that they are submitted via NQS (the Network Queueing System, described in Section 7.1.1). 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. Such mechanisms are provided on the Connection Machine CM-5, the IBM SP2, the Intel Paragon, and the Meiko CS-2, among others. They are described in Section 7. Likewise, the mechanisms described next can be used within the confines of a predefined partition, rather than applying to the whole machine.

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.

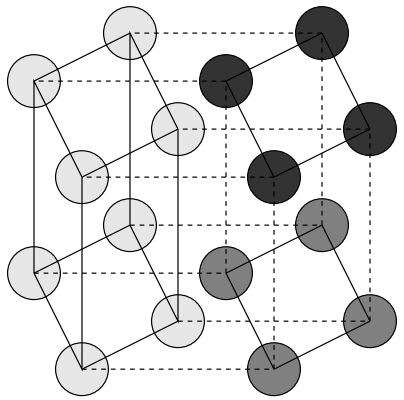


Figure 3: *Example of partitioning a 4-cube into a 3-cube and a pair of 2-cubes. Dashed links are not used in this configuration.*

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 [582]. Each quadrant has an independent sequencer — the unit that interprets the SIMD instruction stream and broadcasts it to 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 very limited. This is so because enabling smaller partitions would require additional sequencers, and would complicate the hardware design. The same consideration also guided the design of the Illiac IV [45], an earlier SIMD machine that was to have four quadrants, although only one was actually built. Other non-SIMD architectures do not have this problem, and allow for finer partitioning. This gives users more flexibility, by providing the option for many small partitions that are used for program development and testing.

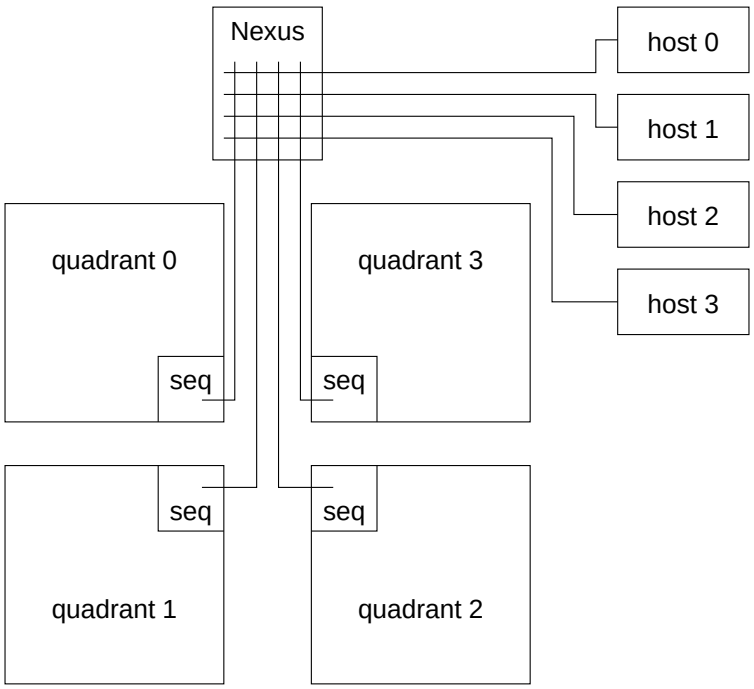


Figure 4: *Partitioning the CM-2 SIMD array (adapted from [582]).*

In many cases the structure of the machine is based on powers of two: PEs are grouped into pairs, then groups of four, eight, and so on. The possible partitions follow the same structure, so partition sizes can be any power of two between 1 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.

Considering the similarities in the methodology of constructing these two architectures, it is not surprising that the mechanisms used to partition them are practically identical. Given that a large partition of the machine is free, it is easy to satisfy a request for a small partition by repeatedly breaking the free large partition in half. The harder problem is uniting small partitions into large ones when they are released, because this unification must match the hardware structure. For example, only adjacent 2-D cubes can be combined into a 3-D cube. We shall start with an overview of partitionable systems based on multistage networks, and then proceed to hypercubes. The question of unification will be treated only for hypercubes, because it is easier to visualize in that context.

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 [503], The NYU Ultracomputer [244, 243], the IBM RP3 [458], PASM [535, 534], TRAC [83, 372, chap. 7], and Cedar (where clusters are connected to the network rather than individual PEs) [222, 319]. In distributed memory machines, all ports are connected to PEs. Examples include the CM-5 [356], the Meiko CS-2, and the IBM SP2 [553].

Multistage networks are generally considered to be a realistic alternative to crossbar switches, which have unit delay but n^2 components. They are arranged as $\log n$ stages of 2×2 switching elements (hence the logarithmic delay), with $n/2$ such elements in each stage. The outputs of each stage are connected to the inputs of the next stage is a certain pattern. Various patterns have been proposed, and much work has been devoted to analyzing the capabilities of the resulting networks. Luckily, most of them turn out to be largely equivalent [619, 530, chap. 5].

As a concrete example, consider the generalized cube network. The input ports to this network are connected to the first stage in a "shuffle" pattern. This pattern derives its name from the perfect shuffling of a pack of cards: the first half ends in the even-numbered positions (counting from 0), and the second half in the odd-numbered positions. The stages are then connected to each other in the "cube-i" pattern, where i is the serial number of the stage. With this pattern, each switch is connected to the corresponding switch in the next stage, and also to the switch whose number (in binary representation) differs only in the ith bit position, counting from the left. An example for $n = 16$ is given in Fig. 5.

At first glance the shuffle pattern before the first stage of this network seems redundant: it just changes the numbering of the ports, without adding any functionality. Its usefulness becomes evident when we partition the network into disjoint parts. The partitioning 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 [533]. 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 [531, 292, 530, 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 [535, 534]. 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. However, the

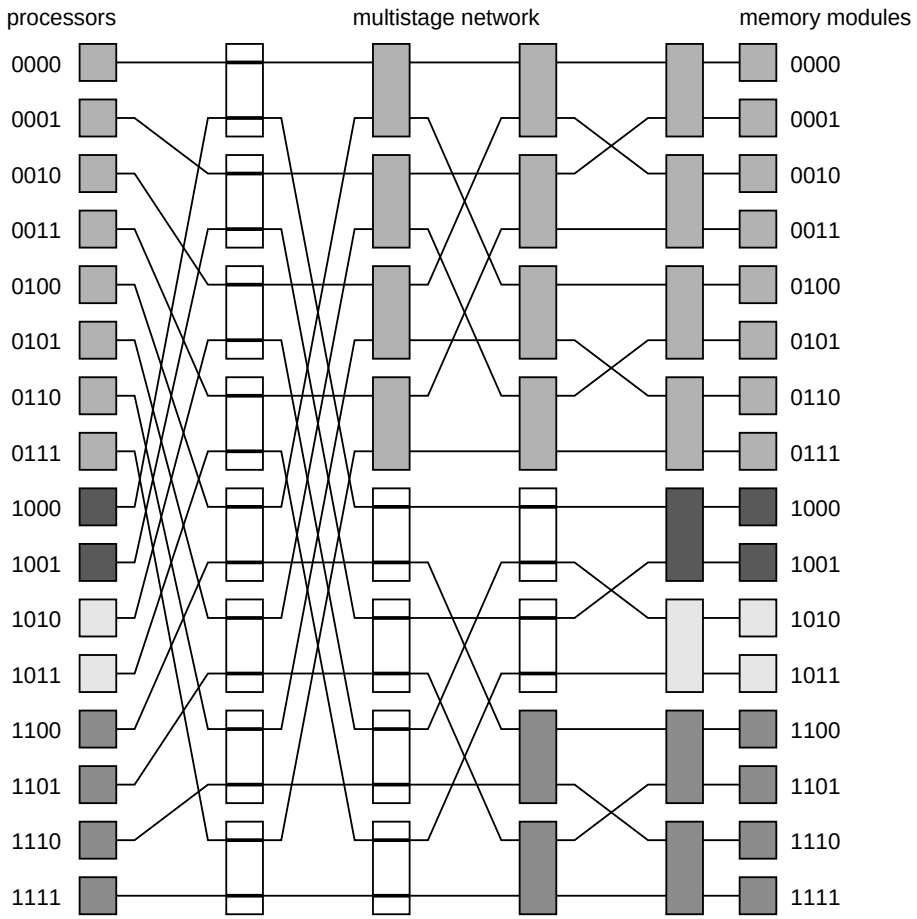


Figure 5: *partitioning a multistage network into independent partitions (adapted from [533]). The elements of each partition are identified by a different shading. Switches set to "straight" are not shaded.*

architecture scales to much larger sizes, with a design point of 1024 PEs and 32 controllers, leading to a minimal partition size of 32 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 [584].

The Texas Reconfigurable Array Processor (TRAC) is based on a banyan network. The prototype has four PEs and nine memory modules. While partitioning is not very meaningful in such a small machine, the architecture is scalable and provides a rich variety of configurations that can be changed at high speed [83, 372, chap. 7].

An interesting variant is the Flagship parallel reduction machine, which is based on a Delta network [610, 611]. In this machine the network is used to propagate global load information in addition to its use for data transfer. Overloaded PEs can inject reductions into the network, and they are automatically routed to the least loaded PE. Applications

that do not have a sufficient degree of parallelism can be limited to a subset of the PEs, based on bit positions, effectively partitioning the machine [561]. This is done to prevent too many remote accesses, which degrade performance.

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 [356]. 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[3]. 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 [513]. A number of commercial offerings have been made since then, the most enduring of which are the iPSC series from Intel [28] and the nCUBE machines [258, 443, 174]. Both provide partitioning as described below.

Many hypercube applications are specifically tailored to the topology (e.g. [41, 495, 475]). Therefore if an application is to run on a partition of the machine, that partition must itself be a hypercube, albeit of a smaller dimensionality[4]. 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. The only complication that might arise is handling faults. The suggested approach is to integrate knowledge about faults into the data structures, and avoid faulty components during allocation [476].

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

[4]If the application does not require a subcube, it is still convenient to allocate a subcube, and then possibly to reclaim the leftover nodes [588].

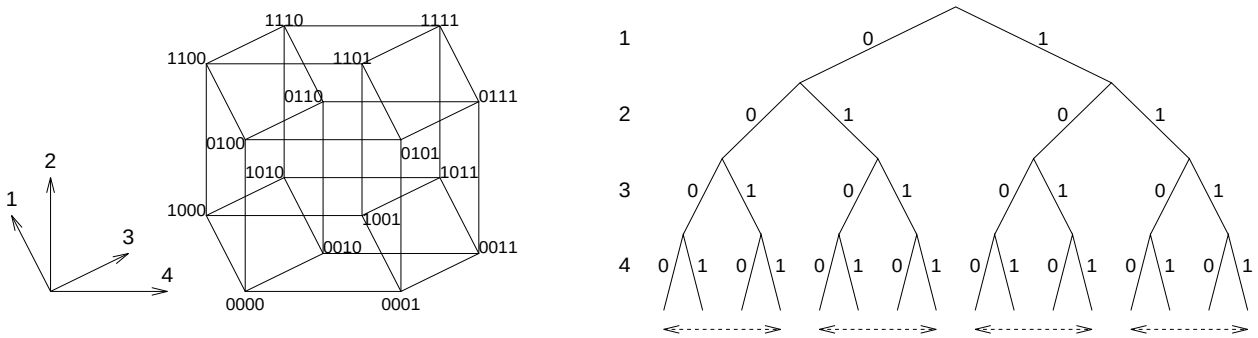


Figure 6: *The buddy system maps dimensions of the hypercube to levels of a binary tree, and uses the tree structure to partition the hypercube.*

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. It is also not true that any pair of, say, $(n-4)$-dimensional subcubes form an $(n-3)$-dimensional 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. If they cannot be joined directly, it might be advantageous to migrate a job running next to one free subcube to the other free subcube [115, 114, 110], thereby creating two adjacent free subcubes that can be joined together. Such migration is also guided by the recognition of the potential to free a large subcube. Therefore the issue of subcube recognition has received much attention in the literature.

The first and simplest method is based on the natural mapping between a hypercube and a buddy system[5], where dimensions of the cube correspond to levels of the tree [113, 173, 146, 35] (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 [331].

In the first dimension, or top level, the whole hypercube is viewed as composed of two subcubes: one comprising PEs with a first bit "0", and the other PEs with a first bit "1". Each of these subcubes is further partitioned in the second dimension, or second level of the tree, according to the second bit in the PEs' IDs. This continues for all n dimensions. This mapping uses only the first bit position to partition the whole cube, so it can recognize just one of the n possible ways to do the partitioning. 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.

Recognition is improved if the PE IDs are regarded as a Gray code [113, 114], rather than as a reflection of the cube's dimensions. In a Gray code, successive elements differ by exactly one bit position. Following the order of the Gray code, we can visit all the PEs in one cycle. As shown at the left of Fig. 7, this can also be interpreted as a tree with

[5]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 [316, 457, 317, pp. 442–445].

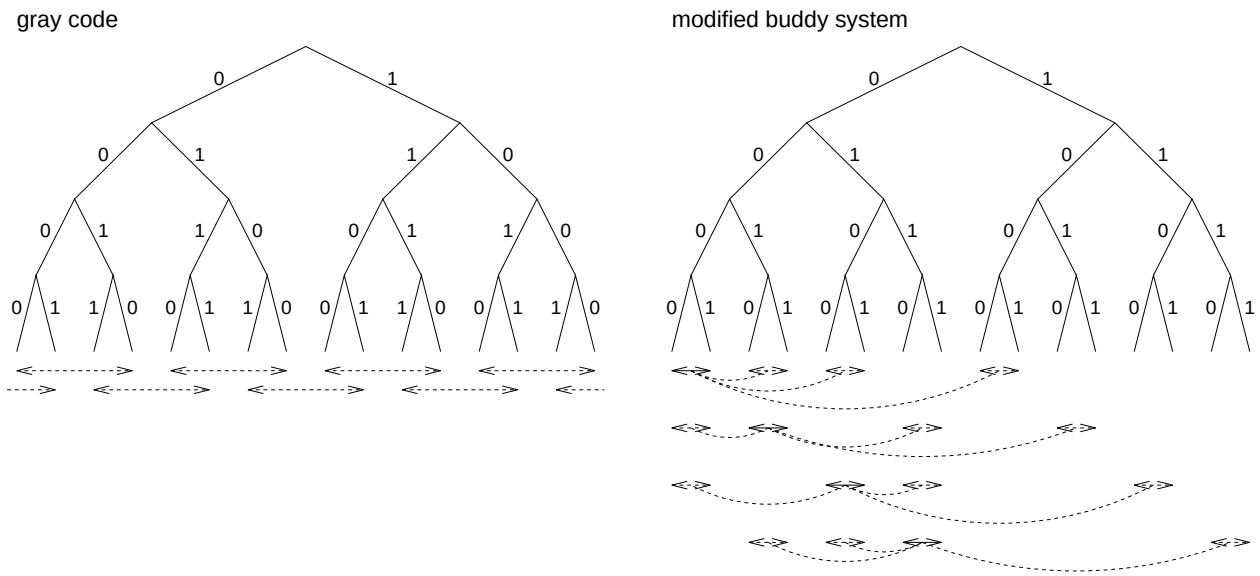


Figure 7: *In the Gray code method, pairs of successive $(k-1)$-dimensional subcubes are considered for creating a k-dimensional subcube. In the modified buddy system, partners along all possible dimensions are considered.*

reflected sub-trees. With this arrangement, every pair of successive $(k-1)$-dimensional subcubes form a legitimate k-dimensional subcube, even if they are not in the same subtree. Thus the Gray code doubles the recognition capability of the buddy system (except for 0-dimensional subcubes and the full cube, which are already fully recognized by the buddy system). Additional subcubes can be recognized by using multiple Gray codes, and full recognition is achieved when $\binom{n}{\lfloor n/2 \rfloor}$ codes are used [113].

Further improvements are achieved if all possible $(k-1)$-dimensional partners are considered, even if they are not adjacent in the tree structure [12]. This has been called the "modified buddy system". A similar method is used in Intel iPSC hypercubes, with the restriction that the address of the first node allocated must be a multiple of the subcube size [46]. Fig. 7 right shows the partners for the first 4 1-dimensional subcubes, resulting in multiple ways to create a 2-dimensional subcube out of the original 4-D hypercube. This can be improved even further by rotating the bit positions used to define the lowest $k-1$ levels in the tree [12]. By searching all levels, full recognition is achieved [10].

Full recognition is also be achieved by using a lattice structure. A lattice is a partially ordered set where every two elements have a unique least upper bound and a unique greatest lower bound [292]. 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

description	*recognition*[*]	*complexity*[†]	*reference*
buddy system	$2^{n+1}-1$	$O(n)$	[113, 331]
gray code	$3 \cdot 2^n - 3$	$O\left(2^n\right)$	[113]
multiple gray codes	3^n	$O\left(\binom{n}{n/2} 2^n\right)$	[113]
modified buddy system: check multiple combinations for creating subcubes	$n2^n+1$	$O\left(k(n-k)2^n\right)$	[12]
free lists for different subcube sizes	3^n	$O\left(n2^{n-k}\right)$[‡]	[309]
tree collapsing and transformations applied to the buddy system	3^n	$O\left(\binom{n}{k} 2^{n-k}\right)$	[123]
gray code combined with two sets of free lists for the different sizes	$3 \cdot 2^n - 3$	$O(n)$	[148]
lattice	3^n	$O\left(n^2\right)$	[292]
graph of prime free subcubes (those not covered by larger free subcubes) and their intersections	3^n	$O\left(\frac{3^{2n}}{n^2}\right)$[§]	[625, 476]
heuristic to maintain an approximation of a maximal set of (disjoint) free subcubes	$< 3^n$	$O\left(n2^n\right)$[#]	[173]
graph of adjacent free subcubes with heuristic to coalesce them when released, using free lists to limit search space	3^n	$O\left(2^{2n}2^{2^n}\right)$[¶]	[627]

[*] The total number of different subcubes recognized. The maximum is 3^n, so such entries imply full recognition.

[†] n is the dimension of the hypercube, and k is the dimension of the requested subcube.

[‡] The worst case is when one free list has $O\left(2^{n-k}\right)$ elements. In a more typical case where list lengths are $O(n)$, the complexity is $O\left(n^3\right)$, or $O\left(n^2\right)$ if done in parallel.

[§] The worst case is based on an estimate of $O\left(\frac{3^n}{n}\right)$ prime cubes [148]. In more typical cases this is expected to be $O(n)$, leading to a complexity of $O\left(n^4\right)$.

[#] A version that actually computes the maximal set of subcubes achieves full recognition at a cost of $O\left(2^{3^n}\right)$.

[¶] This worst case analysis is based on an approximate bound on the number of cycles in the graph, which is itself based on bounds on the number of nodes and their degrees. Assuming $O(n)$ free subcubes, the complexity becomes $O\left(n^2 2^n\right)$.

Table 2: *Listing of subcube recognition algorithms. The complexity measure is for the larger of the allocation and deallocation procedures, which is typically the deallocation.*

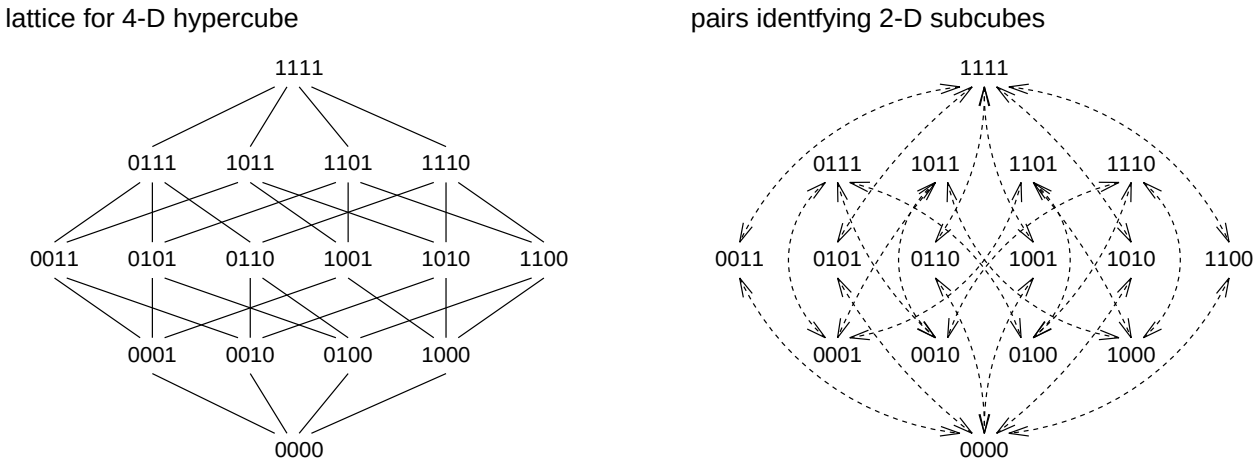


Figure 8: *Subcube identification using a lattice structure.*

larger than or equal to the other. The left part of Fig. 8 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. The splitting and combining algorithms can be performed in a distributed manner, by the affected PEs.

Additional algorithms have also been proposed. Short descriptions and their properties are listed in Table 2. It should be noted that recognition ability is not the only criterion by which these algorithms should be judged. It is also important that the subcubes be recognized quickly [331, 148]. Algorithms based on appropriate simple data structures, while not achieving full recognition, have a complexity of $O(n)$ [331, 148] (which is $O(\log N)$ relative to $N = 2^n$, the number of nodes in the system). Algorithms based on bit vectors have a complexity of $O(2^n) = O(N)$. The more sophisticated graph-based algorithms have an even higher complexity, and are therefore completely impractical for large hypercubes.

Another problem with algorithms that provide full recognition is that they might expose multiple choices. For example, if a freed subcube can be combined with any of a number of adjoining subcubes, which one should be used? A good heuristic is to follow the path that leads to the maximal set of free subcubes, where sets are ordered according to the largest subcubes in them (e.g. a set composed of a 4-cube and a pair of 2-cubes is "bigger" than a set of three 3-cubes) [173]. The rationale is that this reduces the fragmentation, and allows for more efficient allocation in the future. Another possibility is to perform allocation and deallocation such that the free nodes stay in a single connected component [625], or form subcubes that have the least overlap with the allocated one [476].

All the above rests on the assumption that jobs do indeed require subcubes. An alternative is to virtualize the topology: if wormhole routing is used, the question of whether nodes are actually adjacent is not as important as with earlier store-and-forward routing mecha-

nisms. Therefore it is possible to allocate the correct number of non-contiguous nodes, and treat them as a subcube. This will be efficient provided the sets of links used by different subcubes are disjoint [310].

Other systems

Partitioning based on powers of two, using a buddy system, is conceptually simple and easy to implement. Therefore it has also been suggested for systems in which the architecture is not in itself based on powers of two. One example of this approach is a 2-D buddy system proposed for the partitioning of mesh computers [364, 363]. The Cray T3D, which has a 3D torus topology, also requires partitions to be powers of two[6] [474]. The DHC and DQT schemes for gang scheduling are based on a buddy system [197, 204, 217, 273, 272] (DHC is described in Section 5.4). A similar approach has also been proposed in the context of a parallel reduction engine [269].

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 [512, 435] (or an actual implementation of all-to-all communication, as in the VPP500 [407, 592]). 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.

A number of algorithms for variable partitioning have been devised. The simplest approach is to use a central resource manager, which keeps track of the allocation of PEs to partitions [427]. This approach is used on the IBM SP2 [284]. It is adequate because the resource manager is only activated when jobs are submitted or terminated. As this happens at a relatively low rate, the centralized controller does not become a bottleneck.

Another approach is to coordinate the PE allocation using a hierarchical structure. For example, a buddy system can be used where a partition can be composed of a number of blocks from different levels, such that the sum of the allocated PEs is the requested number [375, 606]. Alternatively, one can use a virtual hierarchy of control on the PEs themselves. Such a structure forms the basis of the so called "wave scheduling" mechanism developed for the MICROS distributed operating system [595, 596]. It guarantees that the allocated PEs are close to each other in the network, so it reduces interference among partitions. A similar scheme was proposed in the context of the original CHoPP project, where the goal was to partition a large hypercube among multiple users [558].

[6] This restriction has been alleviated in the newer Cray T3E.

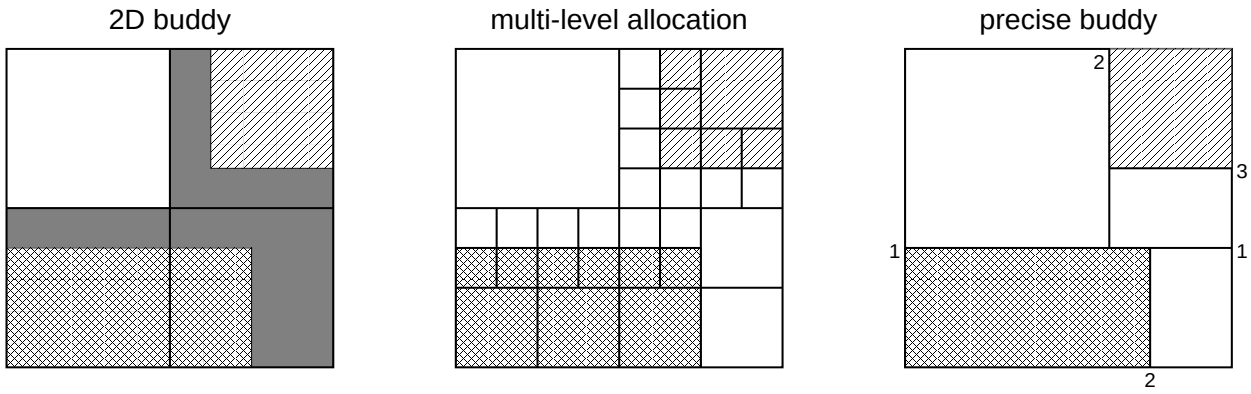


Figure 9: *Allocations for* 6×3 *and* 3×3 *jobs in an* 8×8 *mesh, using variants of the buddy system approach. Fragmentation is shown in gray.*

Submesh allocation

Going back to architectures with topological constraints, there are such architectures that nevertheless allow for partitions with various sizes. A case in point is mesh-based machines, where allocations may be required to be rectangular sub-meshes of various dimensions. Similar algorithms can be used for tori [474].

A simple approach to submesh allocation is the *frame sliding* algorithm [122]. In this algorithm, a rectangular frame representing the exact request size is placed on the mesh, starting with the leftmost and lowest free node. It is then slid across and up in increments of the frame size, until a free region of the required size is found. Note, however, that not all potential locations are checked (so as to shrink the search space and reduce cost), and therefore a free region might be missed.

Another 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 [364, 363]. 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 [375, 606]. 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 [398]. An example comparing the three approaches is given in Fig. 9.

The *interval set* algorithm finds a submesh of the desired size, if it exists anywhere in the system [416, 415]. Interval sets are a data structure used to keep track of allocated nodes in rows of the mesh; in a nutshell, each sequence of busy nodes in the row is an interval, and all such intervals are recorded. When a submesh of $x \times y$ is requested, a condensed representation of the first y rows is generated. This includes intervals of nodes that are busy in any of the y rows. This representation is searched for x adjacent free columns, which

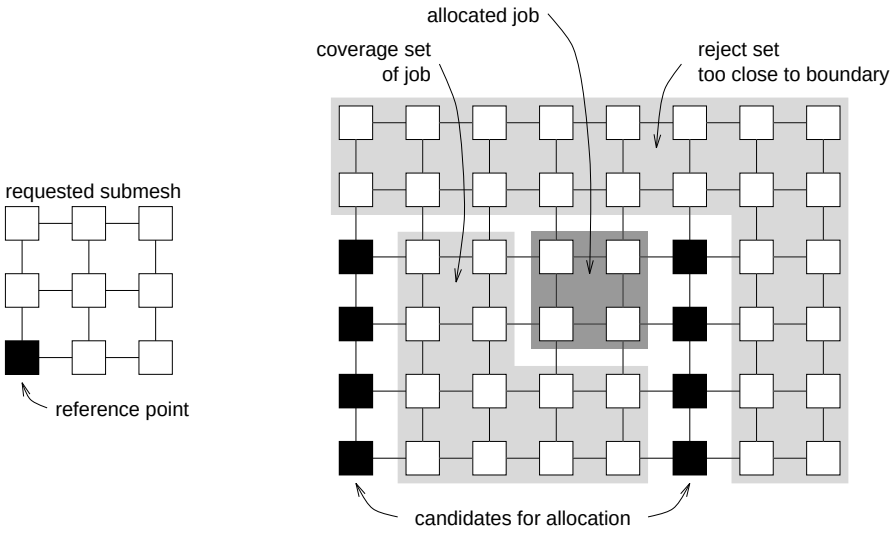


Figure 10: *Submesh allocation based on a coverage set.*

indicate that a free $x \times y$ submesh is available. If none are found, the first row is removed and the next row added in its place. The resulting data structure is again searched for x free columns. This is repeated until a free submesh of the required size is found, or the whole mesh is exhausted.

Another algorithm that finds all possible allocations is based on computing the *coverage sets* of all current jobs [637, 60]. 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. 10. 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) [163].

Other efficient schemes use a list of maximal free submeshes or a list of allocated jobs. Upon allocation, the list of free submeshes is updated using table lookup indexed by the relative position of the free submesh and the allocated one, yielding a set of new smaller free submeshes [474]. Alternatively, an overlapping set of dominant submeshes can be maintained [308]. With the list of jobs, an attempt is made to allocate each new request adjacent to an allocated submesh, or in a corner of the system, thus reducing the search space [149]. If placing the request adjacent to an allocated submesh overlaps with another allocated submesh, the algorithm checks other locations by sliding the request up or down (along the vertical edges of the original allocated submesh) or left and right (along the horizontal edges). After all possible locations are identified, the one with the highest boundary value is chosen

to satisfy the request. The boundary value is computed as the sum of nodes on boundaries of the system and nodes neighboring allocated nodes. This tends to reduce fragmentation. It is also possible to consider busy nodes that are near the allocated submesh, but not adjacent to it, in order to choose the most crowded area and further reduce fragmentation [121]. A final optimization is to perform lookahead into the queue of waiting jobs, and take their requirements into account too [60, 474].

If the exact configuration cannot be found, embeddings can be used [380]. This is also useful for supporting programs that use a different logical structure, e.g. embedding a 3-D torus in a 2-D mesh partition with the same number of PEs. Alternatively, the requirement for a rectangular submesh can be dropped and a non-disjoint partition composed of PEs from different parts of the system used instead [375, 606]. This has the added advantage that faulty PEs do not invalidate whole submeshes [409].

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 [347, 600, 134, 213, 327]. 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 [179, 383]. 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 [338, 213, 235]. A similar formulation is possible using the system *power* as a metric, where power is defined as the throughput divided by the response time [313, 314].

The above schemes find the best partition size for a balance between minimizing response time and maximizing throughput. However, from an individual application's point of view, minimizing response time is more important that maximizing throughput. Therefore, users may be tempted to request a higher number of PEs, even if their application cannot use them as efficiently. Such behavior can lead to reduced performance in a loaded system, because jobs would have to wait longer to obtain the requested PEs.

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.

Choosing the PEs

While flexible partitioning is usually based on the assumption of uniform distances between PEs, this is not always the case. Even with advanced routing mechanisms, some PEs are closer than others. In some cases, especially systems that have a hierarchical structure, the differences can be quite large.

For example, large shared memory machines are often built by connecting multiple clusters of PEs. Examples include Cm* [227, 228], DASH [357, 358], Hector [599], and the KSR1 [216]. In such an architecture, access to memory on a remote cluster can be an order of magnitude more expensive than access to memory in the local cluster. Therefore jobs should be mapped within a single cluster is at all possible [79, 80].

Partitionable hardware

When the partitions exist for the duration of a job's execution, it may be beneficial to set them in hardware. This guarantees that the partitions are independent of each other.

A simple example of this approach is provided by the Fujitsu VPP500 supercomputer, with its crossbar interconnection [407, 592]. Another is the Polyp multiprocessor. In this machine, a set of PEs are connected via switches to a set of buses [387, 382]. By proper settings of the switches, partitions with different numbers of PEs and buses can be created. This allows both the computation power and the communication capacity to be tailored to an application's needs. In addition to being independent of each other, there is also full flexibility in allocating resources to the generated partitions.

Other examples are machines with point-to-point links which may be disconnected. For example, a mesh can be partitioned into two independent parts by disconnecting all the links between two adjacent columns of PEs. A torus can also be divided in this way, provided it is possible to loop back and close the column and row rings when they are disconnected. For example, Fig. 11 shows a 6×4 torus that is partitioned into three tori with dimensions 4×4, 2×3, and 2×1 (Of course, this example is artificially small, and loopback does not add functionality if there are less than 3 PEs in a dimension). The Cray T3D, which is based on a 3-D torus network, lacks this capability. Thus partitioning it results in a set of meshes, rather than tori.

Some systems that provide variable partitions are not physically partitionable. Consequently jobs running on different partitions might interfere with each other, especially by

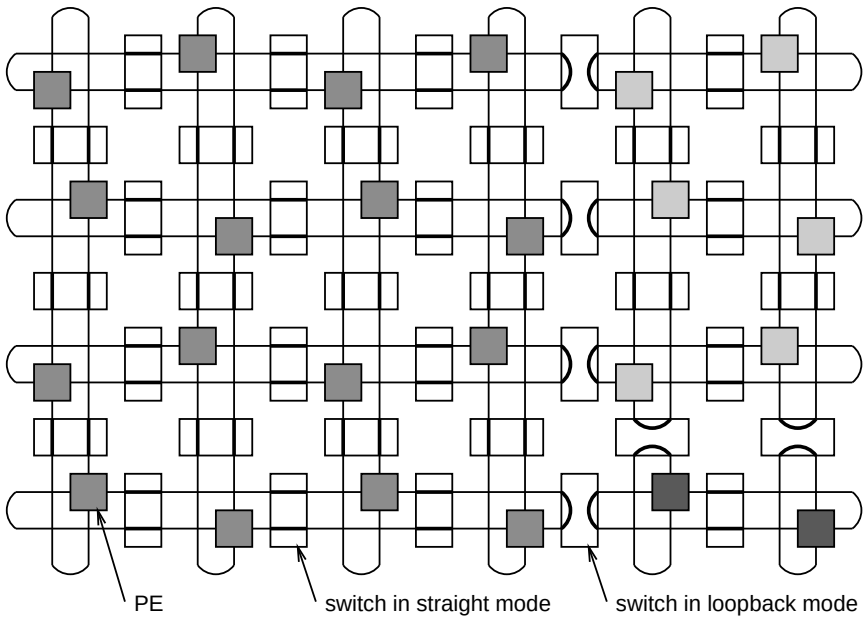


Figure 11: *Partitioning a torus requires switches that are capable of looping back to create small rings when partitioning larger ones.*

loading the communications network [413]. An example is the IBM SP2, which uses a multistage network but allows any subset of PEs to be grouped into a partition [283]. Another example is the Intel Paragon, which uses a mesh architecture with dimensional routing. Thus messages belonging to a concave or disjoint partition may pass through links shared with another partition [375, 403].

Mechanics of job execution

An issue that is often ignored, but which may have a large impact, is the mechanics of job execution. In particular, once nodes are allocated, what additional administrative operations need to be handled? how is the communication set up? how do they access the executable file?

A relatively common approach is to break the problem into several components. Typically this includes [427]

- 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 [401].

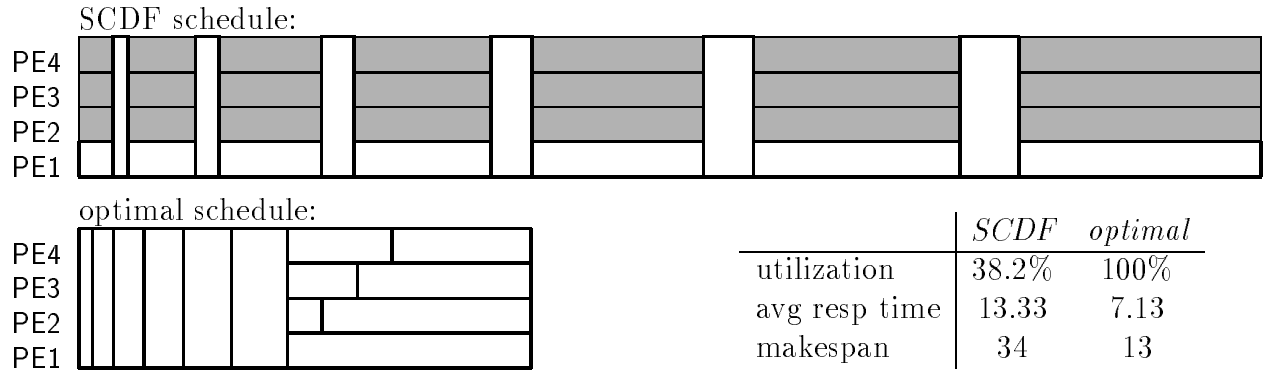
Tearing down a partition and terminating jobs cleanly is just as important as setting things up. In particular, it is crucial that no stray threads be left behind when the job terminates (possibly abnormally) [499]: at best, these orphaned processes pollute the system and consume resources such as entries 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 [295]. A notable exception is the Nexus distributed system, in which the kernel monitors the status of processes, and propagates failure or status messages to other nodes [579]. However, this requires specific user directives that specify what to do in each situation. In the ParPar system, process failure is automatically propagated to other processes in the form of a SIGUSR1 signal. This allows fault-tolerant applications to catch the signal and re-organize the computation, whereas naive applications are terminated cleanly [207].

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" [322, 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 [158], or through compile-time analysis [500, 40]). Second, if short jobs continue to arrive, long jobs might be starved.

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 [384], and the "largest job first" policy [112, 363, 638]. 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 [361, 331]. On the contrary, they may even cause large losses owing to fragmentation.



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

Figure 12: *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.*

The problem with scheduling the smallest job first is rectified by a variant called "smallest cumulative demand first" [384, 360, 519]. 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 [331]. A counter example is shown in Fig. 12. It also suffers from the same disadvantage of shortest job first, namely the requirement to know the execution time in advance. In practice, this disadvantage can be avoided by using a preemptive smallest job first policy with high and low priority queues, which approximates the smallest cumulative demand first policy [360].

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 [127, 125]. 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 [126]. 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. However, the fragmentation can still be quite large [363].

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 [331, 330][7]. 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 [330].

One simple improvement that has been suggested is to change the selection algorithm on-line, depending on the characteristics of the workload. Thus a decreasing order of jobs will be used to improve utilization when large batch jobs are dominant, but small jobs will be scheduled first when the workload is mostly interactive [479].

[7] The more optimistic results in [384, 361] 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.

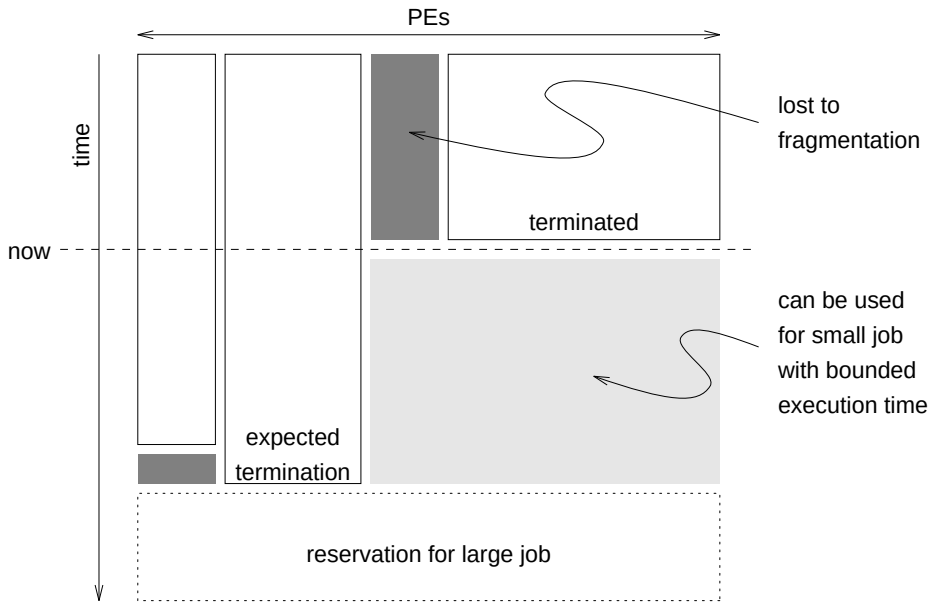


Figure 13: *Runtime bounds on executing jobs allow reservations to be made for large jobs while controlling the amount of resources lost to fragmentation.*

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. A simple version of this approach, called "scan", has been suggested for hypercubes [331, 330, 328]. 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 [410]. Then when the running job terminates, its subcube is reused.

Multiple queues for jobs with different requirements can also be used to match jobs with complementary requirements that can run side by side. For example, this is done in PASM, where the different queues represent different potential partition sizes [584]. 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 [323]. This allows various shape-based heuristics to be used, where the "shape" of a job is determined by the proportions of the rectangle it requires in the processor-time space [287]. 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 [287, 147]. 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. 13). Such scheduling of small jobs in "holes" in the schedule is called *backfilling* [368].

NQS is now used at a large number of sites, with different parallel machines. It is described in Section 7.1.1.

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 [84]. 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 [378]. 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 Kali [318], which also compiles a global address space program for execution on a distributed memory system, and in many message-passing systems, including EUI [39], p4 [94], and PARMACS [95].

While making the above-mentioned guarantee is a commendable gesture, some systems do very little in terms of additional support. For example, the actual partitioning of the PEs may be left to the system administrator [605], or else it may be done by the users themselves in a first-come-first-serve manner [303, 523, 151, 287]. However, some interesting mechanisms for automatic partitioning have also been devised.

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 [235, 633, 519]. 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 [490]. 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. Such problems do not exist if time slicing is used, because then allocation decisions do not have such a dramatic effect on the future. However, the motivation for adaptive and dynamic partitioning is to avoid the overheads and perturbations associated with time slicing, so this is not an acceptable solution.

Minimizing response time is but one goal of operating systems. Another goal is to maximize throughput. A promising policy for satisfying both goals 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 [361, 360, 117, 154]. 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 [514, 628]. 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.

A few other algorithms have been devised that are not derived directly from the equipartition ideal. These algorithms explicitly try to reserve PEs for future arrivals. The first algorithm simply limits the maximal partition size that any single job can obtain [489, 628, 490]. Thus no single job can monopolize the whole system to the detriment of others. It has been

shown that adaptive partitioning with a maximal partition size leads to better overall efficiency than variable partitioning based on the average parallelism in applications, provided the maximum is reduced as the load increases [116].

The second algorithm keeps insurance against a sudden surge in load, by always allocating only a certain fraction of the free PEs, and keeping the rest free. This leads to smaller allocations as the load increases, but somewhat surprisingly, is inferior to the first algorithm [489]. One reason is that this algorithm tends not to utilize all the PEs except under very heavy loads. Under low loads, it would be better to allocate as many PEs as possible [98].

The final algorithm is based on a state machine. At each instant the system is in a certain state, which identifies the ideal partition size [489]. 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 comparison shows that this algorithm is more robust than the others, meaning that on average it performs better for widely different load conditions and for different workload characteristics.

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 [164]. 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. 14 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 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 [108, 31]. 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

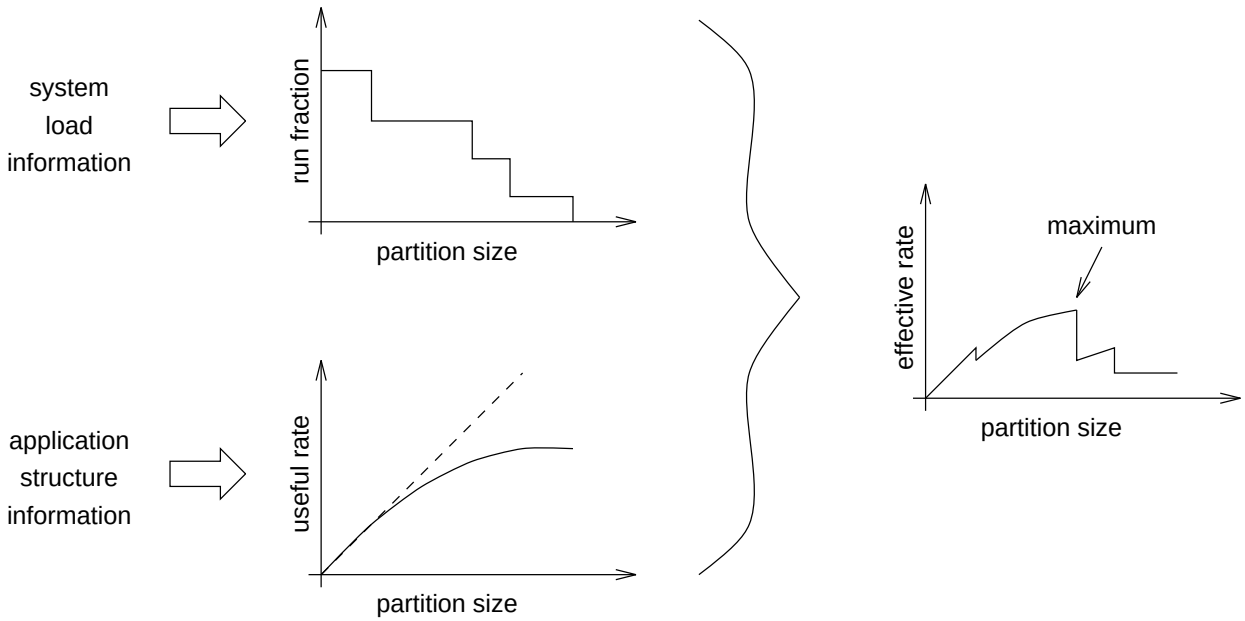


Figure 14: *Combining system load information with a model of application efficiency using different partition sizes provides the optimal size, which will maximize the effective computation rate.*

time it would require on different partition sizes [446, 383]. 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) [520, 341]. 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 [81].

The problem with all the above schemes is that they assume that precise information about the application characteristics is available, and that the application cooperates with the system to find the optimal partition size. Note, however, that this is the optimal size from the system's point of view, i.e. it is designed to improve throughput at the possible expense of response time for any single application. Therefore full cooperation is not always guaranteed. An alternative is for the system to collect the information itself, based on the performance of previous executions of the job [236].

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 [452, 515]. 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 [448]. Therefore, especially if large jobs have good speedup characteristics, it is better to assign large jobs more processors [447].

Theoretical results

The popularity of parallel programs that accept the number of PEs as a parameter that is set at load time has prompted some theoretical studies of this model of PE allocation. The question is phrased thus: given m PEs and n parallelizable jobs, which are characterized by a runtime that is a non-increasing function of the number of PEs allocated to them, find a schedule that minimizes the makespan. If the total amount of work done (i.e. the product of the number of PEs and the runtime) is constant or decreasing with the PE allocation, the solution is trivial: all the PEs are given to one job after the other. Therefore an additional requirement is that the work be nondecreasing (as it typically is, due to overheads [213]), or that each job have a limit on the number of PEs that it can use. This problem is NP-complete. Various heuristics have been developed for the different cases (e.g. for different requirements on the work, and for a possible requirement that the allocated PEs be contiguous). These heuristics lead to schedules that are within a factor of 2 or 3 from the optimal [587, 586, 608, 585].

Note that like many other results in scheduling theory, these algorithms are for *off-line* scheduling, where all the jobs are given in advance. They do not handle on-line scheduling, where jobs arrive in an unpredictable manner. As a result, the feedback effect alluded to earlier is avoided.

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 [581, 633, 383, 395, 101]. For example, when an application enters a sequential phase of computation, its PEs can be reassigned to another application [633, 598, 630]. 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 [360]. This can happen in small installations, but is not expected to be a problem with large parallel machines. 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 [193, 100, 473]. Interestingly, one of the first real implementations of dynamic partitioning on a parallel machine was a port of the Piranha system — which was originally developed for networks of workstations — to the CM-5 parallel computer [101].

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 [633, 395]. This ensures that each job always executes on the assigned number of PEs. However, it may cause thrashing if the load changes frequently.

The "process control" mechanism is more lenient. It allows jobs to create virtual PEs (kernel threads) independently of the server. The server itself is activated periodically and checks the total number of virtual PEs in the system. Jobs that are found to have too many virtual PEs are notified of this condition, and correct it at their convenience [581]. Alternatively the jobs may correct the number of threads by themselves, based on performance measurements [521]. Eventually the total number of virtual PEs becomes equal to the number of physical PEs, thus preventing any overhead due to context switching and synchronization delays. This scheme reduces the coordination that is required in cases where the requirements change at a high rate, at the expense of some context switching activity until the number of virtual PEs is adjusted.

The main drawbacks of process control are the extra context switching, and the need to trust user applications to reduce their number of virtual PEs when they are asked to do so. These issues are addressed by scheduler activations [20], which are a new mechanism for thread manipulation and communication between the kernel and the user-space thread package. It is described in Section 4.4.2.

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, a.k.a. "problem heap", "task queue", "agenda parallelism", "self scheduling", "master-slave", or "farming" [412, 128, 99, 564, 496]. 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 (in some cases, a distinguished master thread is responsible for maintaining the workpile). Additional workers may be added at any time, and existing workers may be removed whenever they finish one chore and before they start another [106, 101]. 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.

It should be noted that the workpile model of application structuring and execution is quite popular, and has been used in various systems. The creation of independent chores can be exposed as part of the user interface in the form of special constructs provided by the programming language. Examples include the `eval` tuple in Linda [230, 9] and futures in Multilisp [253, 324]. A similar concept is the creation of "chares", specified by entry points that allow for remote invocation, in the Chare kernel [209]. This is turn is reminiscent of many concurrent object-oriented languages using a process model, where method invocation may be viewed as executing a chore [118]. In WorkCrews new chores are created in a lazy fashion, so as to reduce overhead if they end up being executed locally [597]. Alternatively, chores can be created by a compiler [466, 465, 414]. A common extension of independent chores is the concept of virtual PEs — the difference being that the different code fragments can interact with each other — with the mapping to physical PEs hidden in the runtime system [581]. Examples include the multiprocessor implementation of concurrent C [124, 224] and various thread packages [130, 612, 59, 192, 87, 105, 55, 529]. The Psyche operating system on the BBN Butterfly provides a similar notion of virtual PEs at the operating system interface [510]. Scheduling in Mach can also be interpreted in this light, with threads as virtual PEs that are executed by whatever number of physical PEs are assigned to the processor set [65]. A variant of this approach is used in the Uniform System, also on the BBN Butterfly. In this system the shared workpile is only used if multiple instantiations of the same chore are required [573]. None of these systems support automatic changes in PE allocation, but they could do so without changing their user interfaces.

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. The "process control" scheme dodges this issue by passing the responsibility completely to the user level. Other 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 [20, 49] and the Psyche system [508, 390], and are described in more detail in Section 4.4.2. 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

[562, 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 [422, 171, 396]. In some cases, such redistribution is supported by the system (albeit with the intention of redistribution for different phases of the computation, not for changes in the available PEs) [23, 378]. 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 [446, 171, 543]. Alternatively, a model where reconfigurations are only allowed at certain points in the application can be used [630, 629, 185, 521]. 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[8].

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. 15) [581, 361, 395, 117, 448, 154].

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 [422, 30]. 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 [421].

A similar consideration applies when the partitions are assigned to phases of jobs, rather than to complete jobs. This allows a job to relinquish a large partition when it enters a serial phase in its computation. However, in order to avoid penalizing such jobs, the serial phase (i.e. the request for a single-PE partition) should be given higher priority than requests for large partitions [620, 278].

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 [396]. A possible solution is to use a *folding* policy rather than an equipartition policy (Fig. 15). 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

[8] Acquisition of PEs at the beginning of parallel loops is common practice on Cray vector multiprocessors. This is typically done using self-scheduling [220].

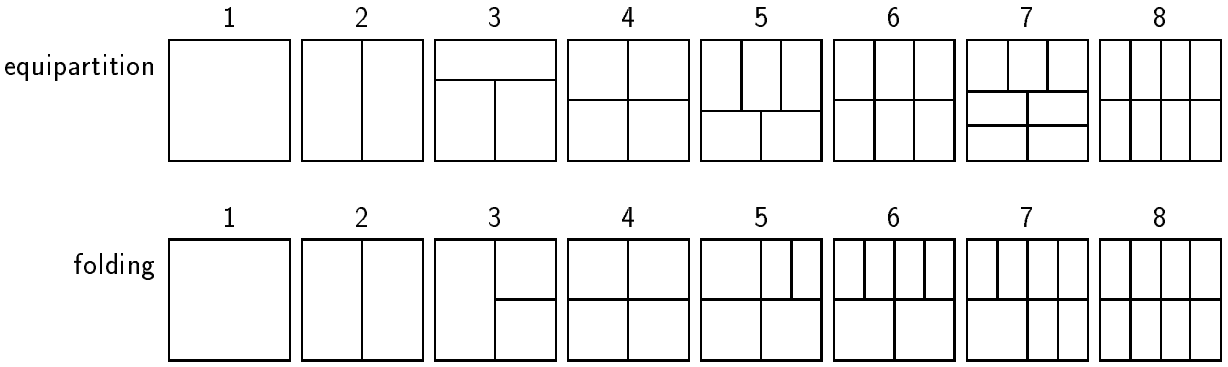


Figure 15: *Conceptual comparison of equipartitioning and folding. In reality, equipartitioning will create non-contiguous partitions so as to reduce the number of PE preemptions.*

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 [396]. 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.

An interesting option available to systems with dynamic partitioning is to evaluate the speedup characteristics of applications as they run, and use this information to determine the optimal partition size. This is done as follows [430, 428, 429]. When a job is submitted, the system executes it for short periods on different partition sizes. Hardware and software monitors are used to gauge the efficiency of the execution, by tabulating processor stalls and synchronization delays. The expected speedup is then computed as the product of the efficiency by the number of processors used. The number of processors that maximizes the speedup is chosen as the target partition size, and serves as an upper bound on the application's partition size. A simpler alternative is to add or delete PEs one at a time, based on the performance of a simple metric such as a barrier synchronization operation [521]. This avoids the inefficiencies of executing the job on an unsuitable number of PEs just to see what happens.

One problem with dynamic partitioning is the overhead of re-partitioning, and possible need to transfer computation state from one processor to another (in distributed memory machines) [446, 441, 543, 289]. The common solution is to limit the rate of such reallocations [422], or to perform them at predefined points in the application where the state is minimal [630, 629, 185, 521]. In addition, it should be noted that proposed dynamic partitioning schemes suffer from the same memory-oblivion as adaptive partitioning. As a result, jobs may find that they have a (small) number of PEs, but cannot use them effectively because they do not have enough memory [452, 515]. This situation is improved if minimal partition

sizes are imposed [421].

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 an artifact of the scheduling mechanism. It is not a goal and does not serve any purpose.

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 [384, 361]. 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 [581] and ASAT (Automatic Self-Allocating Threads) [521].

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 [420], Amoeba [418, 563], and plan 9 from AT&T Bell Labs [462].

A similar mechanism, called Automatic Self Allocating Processors (ASAP), is implemented in hardware in the Convex C2 and C4 system. These machines have up to four PEs, which are connected by a large shared memory and by a large bank of communication registers. The communication registers are used to announce requests to fork new threads. If any PE is idle, it will pick up the new thread and execute it. The communication registers are also used to join threads upon termination [296, 578, sect. 3.4].

Single-PE allocation can also be used beneficially in distributed systems with local queues rather than a central queue. This usually goes under the name "work stealing" [597, 411, 71, 70]. As in the centralized systems, PEs volunteer to join in the computation on their own initiative when they find themselves idle; the difference is that they have to search for work in the local queues of other PEs. This approach is used in the Cilk and Phish systems [70, 71] and some other systems, mainly for functional programming.

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. Systems that do have such a mapping as a goal are described in Section 5.

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. Then optimizations suitable for user-threads and two level scheduling are described.

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 [21, 389]. 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. Indeed, this is used in the Unix compatible OSF/1 AD system [634]. Other example systems that use this approach include distributed memory systems like PEACE on SUPRENUM [505, 506], NX/2 on the iPSC/2 [460], SIMPLEX on the nCUBE [335],, the MOOSE experimental system [497], the original Cosmic Cube [513], and others [146]. The Sylvan system went so far as to add a dedicated kernel-processor to each node [91]; one of the responsibilities of this processor was to perform local scheduling of the node's user-processor and vector unit. An interesting variant is the Time Warp system, which is intended to support discrete event simulation [291].Time Warp In this system local queues are used, but the priorities are determined by each process's *virtual* (simulation) time: the process with the lowest simulation time is allowed to run.

If an arriving message causes some process to rollback, such that its virtual time becomes earlier than that of the currently running process, it will preempt the current process.

Local queues have also been used in shared memory systems, such as StarOS on CM* [228, chap. 6], Chrysalis [352] and Psyche [508, 510] on the BBN butterfly, EMBOS on Elxsi systems [498, 436], the Fujitsu AP1000 array processor [275], PUMA [613], and the EMMA2 system [26]. MEMSY, which has memory modules shared by pairs of PEs, also uses local queues [270].

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 [266, 578, 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 [614, 394]. Operating systems for such machines make use of this feature and support multitasking on each node (e.g. Trillium [92] and Helios [451, 261]).

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 [540, 297, 321, 282]. Each HEP processor could support 128 instruction streams, each with its own program counter and a portion of the register file. These streams belong to up to 16 protection domains, corresponding to jobs. Instructions from the different streams are interleaved into an eight-stage execution pipeline. As each instruction must wait for the previous instruction from the same stream to complete, at least 8 streams are needed to keep the pipeline full. If data from global memory is required, the latency is hidden by using even more streams. A very similar architecture is used in the more recent Tera system [18, 15, 16]. The MIT Alewife takes a slightly different approach: threads are switched only upon a remote access or a synchronization failure, not on every instruction [3, 4]. 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 [141, 140]. 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 [512]. In SPMD programming, this data partitioning is explicitly part of the program [215]. The programmer has to keep a mental picture of how the data is distributed, and insert appropriate code for interprocessor communication. In dataparallel array processing the partitioning is done by the compiler and runtime system. The user writes code that allows parallelism, but does not specify it. A number of programming languages have been designed with this paradigm in mind, including Actus [453, 454, 455], Parallel Pascal [484], and PASM C [340]. Implementations exist not

only on SIMD array processors such as the Connection Machine CM-2 [268, 11], but also on MIMD hypercubes [257].

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 [175, 526]. The typical measure of load, which is the length of the run queue, is also based on research in distributed systems [86, 344][9]. There is some controversy in the literature on whether balanced placement of new threads is enough [177], or maybe migration of active threads (also called "preemptive load balancing") is required as well [332, 150]. 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.

Regrettably, there is little real data concerning these issues in parallel systems. Relying on data from distributed systems is justifiable as long as the interfaces and services provided by the two types of systems are similar. This is not always the case. For example, parallel systems might include an interface that allows an application to spawn a number of parallel threads at once, as opposed to the forking of single processes that is typical in uniprocessor and distributed systems. In addition, the system may guarantee that these threads will actually execute on different PEs. Spreading out the newly created threads like this includes a measure of implicit load balancing. Some initial results show that if a parallel system provides this service, it is not as important to provide explicit load balancing as well [204].

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 [312, 32]. Obviously, this scheme has a bad worst-case behavior, because it is possible that the most highly loaded PE be chosen [175, 269]. 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 [248]. An alternative approach is to first probe a limited number of nodes at random, and then choose the best one [405]. Even probing only two nodes reduces the expected maximum load when mapping n threads from $O(\log n / \log \log n)$ to $O(\log \log n)$ [34].

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. [271]), it is generally felt that this approach will not scale well to large systems. A possible exception is special cases where

[9] More recently, with the advent of heterogeneous systems, the capacity of the different PEs is also becoming an issue [279, 636].

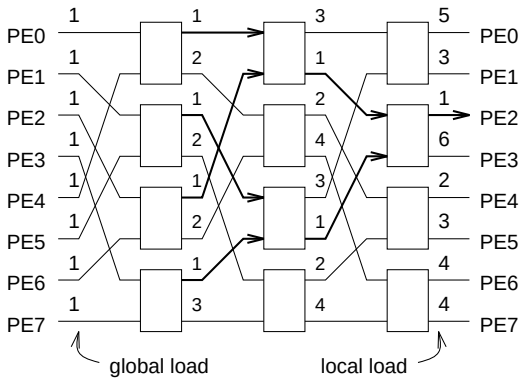


Figure 16: *Dissemination of load information and automatic mapping in Flagship.*

dissemination of load information and mapping are both done in hardware. This approach has been taken in the Flagship graph-reduction machine and in the Heidelberg Polyp. The PEs in Flagship are connected by a Delta multistage network, which is used to propagate load information [561, 610, 611]. 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 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. 16).

In Polyp, a special bus connects all the PEs and contains global information about the highest priority waiting task and lowest priority running task [386, 382]. If these priorities are different, an arbitration mechanism matches one of the PEs with a high priority waiting task with one of the PEs running a low priority task. The high priority task is then scheduled on that PE.

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 [210, 385, 602, 221].

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 [197, 204, 217], in DQT [273, 272], in NETRA [119], and for hypercubes [146, 35, 8]. Actually, the same structure has also been proposed for maintaining and distributing the load information required to map single threads as well, in AMPS [306] and in APERM-2 [269].

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 [95, 604]. HPF, Dataparallel C, and Pandore only support a more restrictive multidimensional mesh topology [378, 257, 23]. 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 [56]. However, in many cases the system does not have enough information, and the mapping has to be done by the user. This is typically considered to be part of program development.

Mapping as part of program development

Mapping as part of program development is quite different from mapping by the operating system. The operating system maps threads from one or more applications, typically without prior knowledge about their behavior. In application development, the programmer assumes a model where a certain fixed number of PEs are dedicated to the application, and the question is how to map program components to these PEs [434].

The most common approach is to represent the computation as a set of tasks with interdependencies (a task graph), and have each thread compute a sequence of tasks [238, 7]. The threads actually represent virtual PEs, and are expected to be mapped to physical PEs on a one-to-one basis at runtime. Whenever a task computed by one thread depends on a task computed by another thread, inter-thread communication is required. Taking communication into account when mapping tasks to threads introduces a number of considerations. One is the communication overhead, which should be added to the price of executing a task [120, 445]. If the tasks are fine-grained, meaning that the time required to compute them is small relative to the communication overhead, it may be necessary to reduce the parallelism in the program by collapsing a number of tasks into one task. Such an increase in granularity reduces the relative cost of communication [501, 120, 377, 232, 434].

Another consideration is the matching of the communication patterns to the interconnection topology of the hardware. It is desirable to map interdependent tasks onto threads that will execute on PEs that have a direct communication link between them [73, 290, 552]. This consideration was important in first-generation message passing machines that used a store-and-forward mechanism for data transfer, but has lost some of its justification in contemporary systems based on wormhole routing. However, mapping to match the topology can still be important with other interconnection mechanisms. For example, the KSR1 uses a uni-directional ring, and processors can perform automatic prefetching by snooping on the

ring. However, this only works if the threads are placed in the correct order around the ring [601].

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 [298]. Most of the literature on this subject relates to distributed systems, not parallel ones (see [609, 526] 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 [507, 440]. 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 [86, 103, 547]. Quick load distribution is achieved by non-local communication to disseminate information about load conditions [371]. 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 [432], 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 [486, 150].

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 [255]. A simple alternative is to perform enough computation locally to cover the costs of migration, and then migrating one thread at a time [6].

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 [135, 72, 623]. The analogy is the diffusion of molecules across membranes, which is driven by the difference in concentrations. This scheme is used in MuNet [254] and Alfalfa [237]. The effectiveness of diffusion depends on the network bandwidth, and it is therefore inefficient for low-dimensionality meshes [556]. 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 [135, 165]. This was shown to be more efficient than diffusion with multiple neighbors at once [135, 615], especially if communication is limited to one port at a time [624].

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 [371]. For good performance, threads should propagate a number of hops along this gradient [528]. However, there is a danger that idle nodes will be flooded with work from all sides, so a reservation protocol is recommended [419]. This scheme is used in Rediflow [305].

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. Thus it is desirable that most of the load balancing activity be done by lightly-loaded PEs (i.e. receiver-initiated [432, 615]). One way to achieve this is by defining thresholds to distinguish between lightly loaded and heavily loaded PEs [526], but nodes still need to collect data to establish correct threshold values (this is avoided in load sharing schemes, where only nodes with no work try to solicit work from other nodes [218]). However, if a few nodes are highly overloaded, it might be beneficial to offload their extra work as fast as possible, so maybe they should take the initiative (i.e. sender-initiated balancing [176, 406, 334]). Of course, there is a tradeoff between the frequency of balancing and the degree of imbalance that is experienced [223].

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 [492]. 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 [522, 325]. One of them is used in the MOSIX system [43, 44, 42]. This started out as a distributed system that supports a single-system Unix image for users. 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 [634][10], and consequently in systems that are based on it, such as the Intel Paragon operating system.

Communication requirements can also be reduced while using a deterministic communication pattern. One approach is to perform the communication among selected intersecting sets of $\sqrt{N}$ nodes (in an N-node system), thereby collectively achieving full knowledge about unbalanced nodes [557]. Another approach is to use a hierarchical control structure, like the

[10] 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.

ones used for mapping based on load conditions. In addition, it is prudent to keep the obtained information for future use, rather than starting with a clean slate each time [334].

Finally, it should be noted that load balancing by the operating system is not related to load balancing as part of the algorithm, in order to achieve some performance goal [463, 495, 433], or load balancing within the application to compensate for data-dependent changes in the load at runtime [616, 165, 565, 542]. These types of load balancing are only relevant in the context of partitioned or batch systems. However, the basic policies used can be similar.

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. We classify systems in which the threads are run to completion as dynamically changing partitioning, rather than as preemptive. Examples, such as SpoC and Amoeba, are discussed in Section 3.5.

The main advantage of using a global queue is that it provides automatic *load sharing*[11]. In effect, the workload is distributed evenly across all the PEs. The disadvantages are possible contention for the queue [426] 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 [487, 388, 389, 106]. 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 number of threads at once each time the queue is accessed, thus amortizing the cost of contention and overhead [468, 426, 281]. 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 (see Section 4.4). 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 [626].

[11] 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 [175, 333]. The term "load balancing" is only meaningful when PEs have individual loads, as when local queues are used.

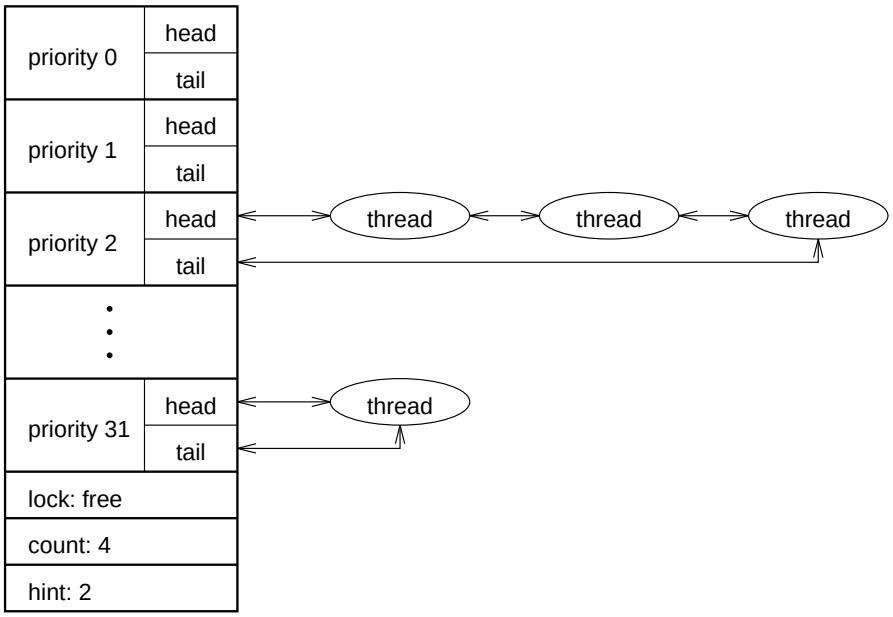


Figure 17: *Priority queue structure in Mach (adapted from [65]). The hint field indicates the highest priority currently in use.*

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 [37]. A lockable shared queue is used in the Hydra system on C.mmp [621, chap. 12], in the Dynix system on the Sequent Balance [570], in Mach [65, 68], in IRIX on Silicon Graphics multiprocessor workstations [370, 48], and in the DASH system which is based on IRIX [357, 106]. It is also used in thread packages such as Presto [59, 192], and in microtasking on multiprocessor Cray supercomputers [220].

Much progress has been made lately regarding the efficient implementation of locks, especially by way of exploiting local caches with hardware coherence [491, 19, 239, 246, 399]. 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 [65] (Fig. 17). 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 [294, 481, 156, 250, 145].

Implementation with fetch-and-add

The problem with locking the queue is that the queue becomes a serial bottleneck [62, 264]. 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 [575].

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 [245, 242] (actually, a fetch-and-increment operation is also enough [219]). 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. 18. 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 [245]. Note that this is not a full implementation, as it does not take care of counter overflow.

A global queue implemented with fetch-and-add is a basic feature of Symunix, the operating system of the NYU Ultracomputer [180, 181]. The implementation of fetch-and-add is a basic feature in the design of the NYU Ultracomputer hardware [244, 160, 14, sect. 10.3.6]. This machine provides access to the shared memory via a multistage Omega network. Fetch-and-add requests aimed at the same memory address are combined in the network, using special hardware. As a result, any number of concurrent fetch-and-adds take the same time to complete as a single read or write memory access. In particular, the accesses are not serialized, and the queue does not become a bottleneck. The combining of independent requests is supposed to prevent the shared index variable from becoming a memory hot spot [459, 353].

```
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 18: *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.*

A distributed global queue

The idea of a distributed global queue might seem to be an oxymoron, but it is not. Such a structure has been proposed for the Heidelberg Polyp multiprocessor [386]. This machine is based on a collection of PEs and global memory modules that are connected by multiple buses. In addition to the data buses, there is a special bus that supports scheduling operations. This bus has two sets of lines that correspond to the systems priority levels. These lines carry the inclusive OR of the inputs from all the PEs. One set is used to signal the priorities of the tasks that are currently running. The other set is used to signal the priorities of new tasks that have been created. Each PE interface monitors these signals. By comparing the priority of the current task to the global lines, each PE can know if its task has the lowest priority in the system. Likewise, it can find if its waiting tasks have the highest priority. If these priorities are different, an arbitration mechanism matches one of the PEs with a high-priority waiting task and one of the PEs with a low-priority running task. The high-priority task is then scheduled to run on this PE.

Mach: partitions with global queues

The approach of partitioning and then scheduling using a global queue shared by all the PEs in the partition was implemented in the Mach operating system[12] [65, 64] and some other systems [90]. The partitioning is based on the concept of a *processor set* (other systems often employ the term "processor pool" [90]). PEs are allocated to the processor set as a result of a special request by the application that created the processor set (possibly a shell). The policy decisions of when to grant such requests are handled by a special server which is not part of the kernel. Owing to this separation between policy and mechanism, it is easy to change the policy when desired. The server may also decide to grant only part of a request, or to reclaim previously allocated PEs.

A global priority queue is associated with each processor set (this is the same queue as depicted in Fig. 17). Threads that are assigned by the application to the processor set are placed in this queue, and executed by the PEs allocated to the processor set. If no PEs are allocated, the threads are effectively suspended from execution.

Mach was developed in an academic environment. The envisioned use of processor sets was to allow users to monopolize PEs without any interference from other users, as might be required in order to generate speedup curves. Such static partitioning which is independent of the actual use to run threads has some drawbacks. It reduces the availability of PEs to other users, might impair fairness, and might cause inefficiency if users do not actually utilize all the PEs they get all the time they have them. On the other hand it might also have some indirect performance benefits. For example, the processor sets can be designed to reflect the machine topology, rather than being allocated arbitrarily, thus improving locality [635]. In

[12] The Mach papers call this "gang scheduling". We reserve "gang scheduling" for schemes like those described in section 5, where a one-to-one mapping exists between threads and PEs. The Mach scheme is closer to "family scheduling" in our terminology.

addition, this scheme reduces the number of PEs sharing a queue (relative to a single global queue), thereby reducing contention for the queue [431].

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 [65].

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[13] relative to the priority of their parent. This ensures that nested spawns are executed in a depth-first order [181]. 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 [57].

It should be noted that in parallel systems, the notion of a priority queue is somewhat different from that in uniprocessors, because P threads execute at once (where P is the number of processors). It is natural to require that the P executing threads be those with the highest priorities. However, this can be hard to implement efficiently, because it requires all the PEs to repeatedly check the queue to see if any thread has a higher priority than the one they are running. In addition, when a high-priority thread does appear, it might cause a cascade of context switches if the PEs happen to access the queue in the order of the priorities of their current threads.

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* [598, 157, 38, 544, 55]. Affinity scheduling becomes more important on machines where the cost of remote access is higher [106]. 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 [249, 576, 598]. 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 [48].

A simple implementation of affinity scheduling uses local modifications to threads' priorities [576, 106]. 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

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

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 3: *Comparison of local and global queues.*

the thread with the highest priority, it boosts the priorities of those threads that used this PE the last time they ran. The boost can depend on the time since the thread last 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 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.

All the above scheduling schemes operate on individual threads, and do not consider the fact that these threads belong to competing jobs. As a result, jobs tend to receive service that is proportional to the number of threads that they spawn. An alternative approach is to give jobs equal degrees of service [361]. This means that threads belonging to a job with lots of threads should have a lower priority than threads belonging to jobs with few threads, or that they should be scheduled to run for shorter time quanta. It is also possible to prioritize jobs, rather than giving them equal service. This can be done by keeping the threads in separate per-job queues. The PEs serve these queues in a round robin manner, and set the time quantum for each scheduled thread according to the priority of the job to which this thread belongs [464].

4.3 Combination of Local and Global Queues

The main features of local and global queues are summarized in Table 3. 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 [194]. In this system, a global queue is used for work distribution, and then local queues are used for the actual execution[14]. 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

[14]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.

already have a high load [492]. Once mapped by `get_work`, threads do not migrate. Similar schemes are used in the KSR1 (see Section 7.2.5) and other systems [507, 414]. 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 [188, 187]. In all these systems, mapping is done at regular intervals, and does not adjust to the load.

In MAXI, performance is further improved by the reuse of thread contexts. If a thread terminates before its time quantum is up, its context is reused for the next thread that is created from the global queue. New thread contexts are created only if all the existing threads are preempted but do not terminate. This is based on the observation that preemption implies either of two conditions: the threads are long, so the overhead of creating new contexts is small relative to the useful computation, or the threads are interdependent, and might be waiting for other threads that have not been commenced yet [527]. A similar optimization exists in many other systems, where threads are run to completion and then their context is reused for the next one. Examples include Scheduler Activations [20, 49], Continuations [166, 152], and the Presto thread package [59, 192].

Another system that uses both local queues and a global queue is the variant of the Mach system developed for the IBM RP3 [85]. The motivation in this case was the observation that when a partition of PEs is allocated to an application, the application might want to exercise some control over the ways in which threads share the use of PEs. If the operating system only provides a global queue or local queues, certain patterns are impossible.

The RP3 Mach scheduling is based on the concept of thread families. Each family has a global queue of threads, and PEs allocated to the family execute threads from this queue. Except for terminology, this is very similar to the processor set mechanism built into the original CMU Mach. One difference is that the RP3 Mach also had the option to map certain threads to specific PEs. At each context switch, a PE would choose the next thread from its local queue or the next thread from the global queue, depending on which had a higher priority. This achieved the automatic load sharing of a global queue, coupled with the advantages of local queues such as the option of using local storage or dedicating processing power to important threads by binding them to PEs.

Finally, 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 [143, 142]. 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.

4.4 Optimizations for Two-Level Scheduling

Scheduling by independent PEs, especially from a global queue, is very common in the second level of two-level scheduling schemes. In this context the scheduling is decoupled from PE allocation, so a number of special optimizations can be applied. Two classes of optimizations

are chunking from the global queue and changes to the thread model so as to provide a better synergy between the kernel and the runtime system.

4.4.1 Chunking from a Global Queue

Scheduling from a global queue can be used by an operating system or by the second level of a two-level system, i.e. the application's runtime system. Indeed, the term "self-scheduling" is often applied to both kinds of systems. In an operating system, it is PEs that schedule themselves from the global queue. In a runtime system, it is worker threads that help themselves to chores from the shared workpile.

However, the two types of system are actually quite different. Perhaps the main difference is the difference in the nature of the entities that are being scheduled. In application self-scheduling, these are typically iterations of nested loops [564, 468, 469, 220]. They are scheduled in FIFO order, with no preemption. Actually "scheduled" might be an overstatement: the whole context stays the same from one iteration to the next, and the only thing that is extracted from the global workpile is the index of the next iteration. In operating systems scheduling from a global queue, the scheduled entities are threads with independent contexts. Threads that exhaust their time quantum are preempted and returned to the queue, and rescheduled only when their priority rises above that of competing threads.

Additional differences exist if the operating system does not use partitioning, i.e. if the same global queue is used for threads from competing applications. If this is the case, the scheduling — that is, the selection of a thread from the queue — includes an element of processor allocation. Self scheduling in applications is only done after PEs have been allocated to run the worker threads. This difference is meaningful because once a specific number of PEs have been allocated to an application, certain optimizations are possible.

The main optimization is chunking. Chunking means that each time a worker accesses the workpile, it takes a number of chores rather than just one. This optimization can result in significant reduction in the contention for the shared workpile, and in overhead for the allocation [337]. For example, if each of P workers takes $1/P$ of the chores, each will only access the workpile once. However, this could compromise the load sharing if the chores are not identical, if the workers do not all start at the same time[15], or if the processors have different speeds [280]. It is therefore advisable to adjust the chunk size as the computation proceeds. Initially, large chunks can be taken. As the computation proceeds and the tolerance for uneven loads decreases, the chunks become successively smaller [337].

A number of schemes that do so have been suggested (see Fig. 19). The formula suggested by *guided self scheduling* is to take $1/P$ of the chores *remaining* in the queue each time [468]. It was proven that if the chores are identical, workers using this scheme will finish all the chores at the same time to within one chore, even if they do not start together. However, if

[15]It is open to debate how much this happens in practice. A recent study of loops from the Perfect benchmark found that static partitioning was perfectly adequate [367]. Another study of multiprocessor Crays found differences in PE allocation times to be a significant problem [186]. A third study found that the distribution of task completion times was quite wide, due to memory contention effects [170].

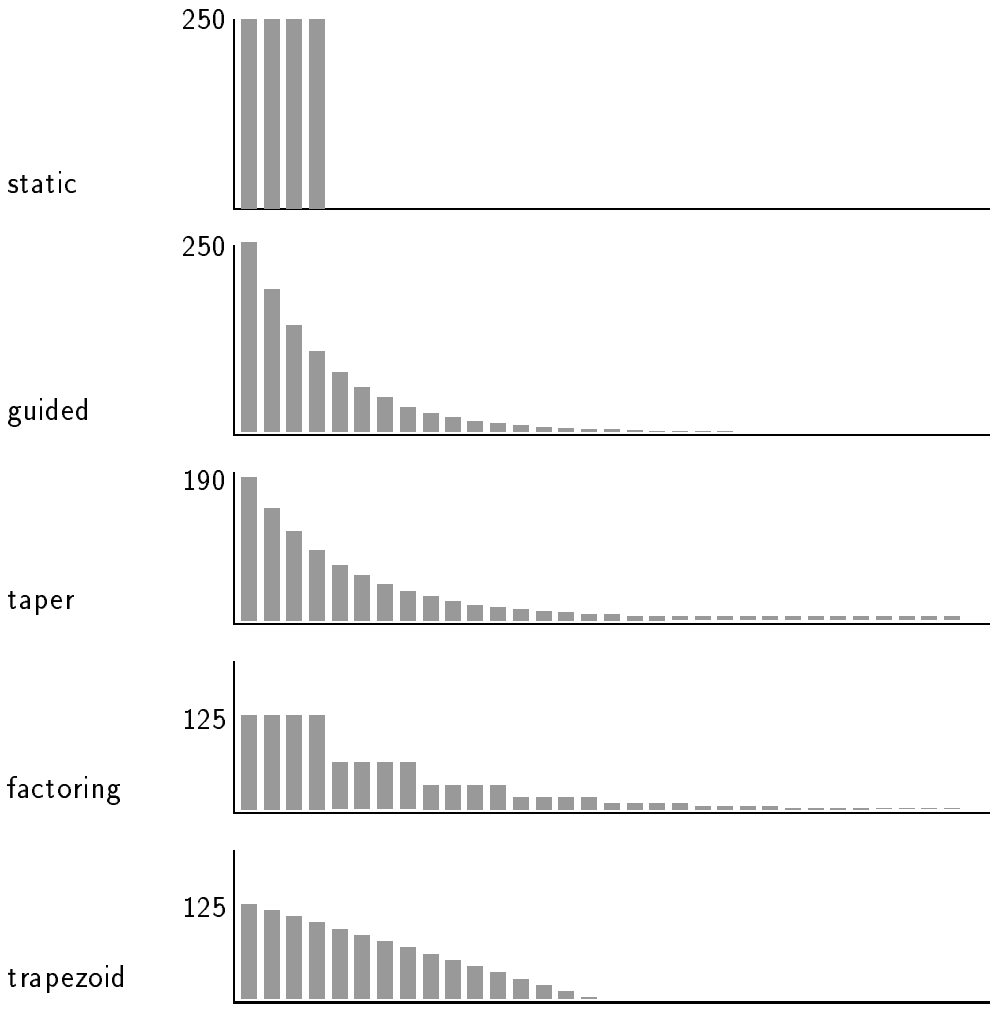


Figure 19: *The chunk sizes created by different self-scheduling schemes, for 1000 chores and* $P = 4$ *(data partly from [281]).*

the time to compute each chore may differ from the others, the first large chunks may take more time than expected, leading to load imbalance.

Taper tries to solve this problem by using runtime statistics of the mean and standard deviation of chore execution times to set the chunk size [379]. Essentially, the chunk size is less than that in guided self scheduling by an amount that is proportional to the variance. In addition, a minimal chunk size is set to reduce excessive overhead towards the end of the loop.

A similar scheme, called *factoring*, also creates smaller and more uniform chunks than guided self scheduling [281]. Chunks come in groups of P, and the size of chunks in each

group is a constant factor of the size of chunks in the previous group. The optimal factor can be calculated based on P and the variance in chore computation times. If the variance is zero, only one group is created, and each worker does $1/P$ of the chores. If the variance is very large, factoring degenerates to self scheduling of a single chore at a time. If the variance is not known, a good heuristic for the first chunk size is $1/2P$, and each subsequent chunk size is half the previous one. The intuition behind the factor of $\frac{1}{2}$ is that if chore computation times are normally distributed, than executing them on dedicated PEs would result in about 50% utilization. Thus at any given time it is best to keep 50% of the work for later, and allocate it to those PEs that finish their share early. However, this is sometimes overly cautious [374]. This observation, together with an initial phase of static scheduling so as to reduce runtime overheads, are the basis of a variant called *safe self-scheduling* [374].

The chunk sizes in guided self scheduling and in factoring decrease exponentially. In *trapezoid self scheduling* they decrease linearly, leading to simpler implementation [589]. While the sizes of the first and last chunk may be set to various values, a conservative use of starting with $1/2P$ and going down to 1 was shown to be a good compromise.

An interesting variant on all the chunking schemes is *cooperative scheduling*. In this scheme, a PE that runs out of work does not only take a chunk of chores for itself; it also maps additional chores to other PEs [426]. This reduces contention for the global queue, but requires locking of the local queues because PEs may access each other's queues.

Finally, it has been noted that chunking by itself may not be enough. If a parallel loop is nested within a sequential loop, additional performance gains are possible if the same chunk is mapped to the same PE each time [389, 555]. This is a special case of affinity scheduling, where the affinity is between successive executions of the loop. Likewise, performance improvements are possible if iterations are re-ordered in the interest of locality [366].

It is debatable whether allocating a number of threads at a time can be applied to scheduling from a global queue by an operating system, because there is no reason to expect the different threads to be homogeneous. However, it is possible to apply this optimization in the special case when a large number of threads are spawned at once, and they are all clones of each other. This is done by the Uniform System on the BBN Butterfly [573], by Chameleon [17], by the Chores system [178], and by the MAXI system on the Makbilan [527].

4.4.2 Variations on the Thread Model

A totally different type of optimization for two-level scheduling involves changes to the thread model. This is based on the observation that both user-level threads and kernel threads have inherent performance drawbacks that cannot be overcome within the constraints of the conventional model.

The problems with user-level threads stem from the fact that the underlying kernel does not know about them and their interactions. The kernel only knows about the kernel threads on which the user threads are multiplexed. Thus if a user thread performs some blocking

operation, the whole kernel thread is blocked with it. If a user thread holds a lock and the kernel thread running it is preempted, the effective lock holding time will grow to include all the time until it is re-scheduled. These problems would be solved if kernel services were used directly, because then the kernel would know about the dependencies. But using kernel services is more expensive, as it requires a trap to privileged code for each operation.

In a nutshell, the problem is one of information and coordination (or lack thereof) [395]. The proposed solutions supply the required information in various ways. For example, it has been suggested that threads be able to request to be temporarily immune to preemption, to prevent cases where the preempted thread holds a lock [181, 265, 632, 320]. This is implemented by setting a special flag in user space just before acquiring a lock. Having the flag in user space avoids the overhead of a kernel call. The kernel checks this flag before preempting a thread, and if it is set, gives the thread some more time. However, no guarantees are made, and the kernel reserves the right to preempt threads that overuse this facility. Another example is handoff scheduling, where a thread that must wait for another thread that is not running explicitly hands its PE to the awaited thread [65, 249]. An alternative formulation of the same idea is priority inheritance, where a thread holding a lock inherits the priority of threads waiting for the lock [265, 603, 349].

Other proposals call for much larger changes. Psyche, for example, includes provisions for first-class user threads, where "first-class" implies that they are recognized by the kernel [390, 508]. One of these provisions is a shared memory segment for communication between the kernel and user-level runtime system, similar to that described above. Another is upcalls that notify the application's runtime library of certain events that might effect its scheduling decisions. These notifications include events like a thread blocking or unblocking while performing a kernel call, a timer expiration, imminent preemption, external invocations that may require a new thread to be created to deal with them, and hardware faults that indicate an exception condition in the current thread. Finally, Psyche provides a scheduler interface convention that allows threads of different types to co-exist and interact [509]. For example, two parallel programs using different thread packages can have a producer-consumer relationship, where consumer threads block if the buffer is empty and are subsequently resumed by threads in the producer program.

"Scheduler activations" are another new mechanism for thread management and manipulation [20, 49]. The scheduler in question is part of the user-level thread package. The activations are upcalls from the kernel to the scheduler. The kernel maintains a constant number of activations, one on each PE. When a PE is added to an application, a scheduler activation is created on that PE, and the kernel upcalls into the user space at a known entry point. The user level scheduler then selects a thread for execution (Fig. 20 left). When a thread blocks or is preempted, the kernel creates an alternative activation on the same PE. This activation selects another thread to run instead of the blocked one. When a thread becomes unblocked, a new activation is created. Typically this comes at the expense of another activation that was running on some PE. The new activation therefore contains information about two threads that are ready to run: the one that became unblocked, and the one that was preempted in order to run the activation. The user-level scheduler can then choose

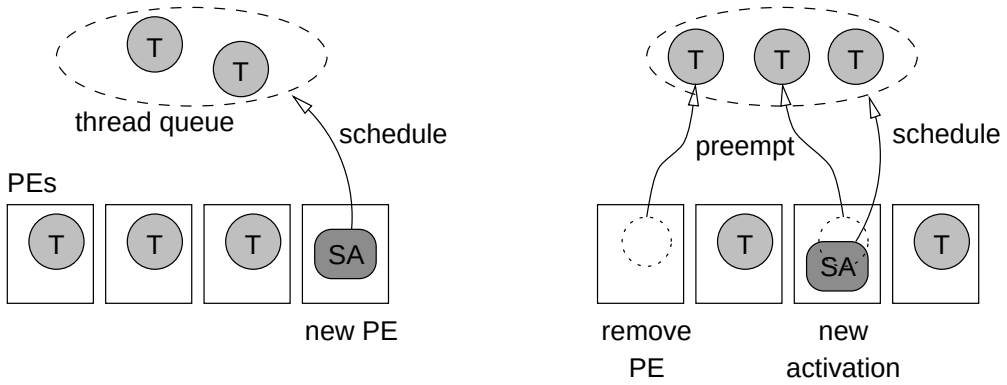


Figure 20: *Handling added and removed PEs with scheduler activations.*

which is more important, or maybe choose a third thread altogether. Similarly, when a PE is taken away from an application, an activation is created on *another* PE, again notifying the user-level scheduler that there are two ready threads but only one PE, and a choice has to be made (right of figure).

A different approach altogether is to forgo all information exchange, and strive to approximate the optimal efficient behavior. For example, in the μSystem, separate threads are designated to perform blocking I/O operations on behalf of those running user code [87]. With cooperative threads, there is an option to create or schedule a new kernel thread whenever an existing one blocks [286, 218]. This prevents PEs from going idle if a kernel thread blocks when performing a blocking operation on behalf of a user thread. MAXI envelopes take this a step further, and also create a new kernel thread if the current user thread exceeds a time quantum[16] [527]. This guarantees correct (albeit possibly inefficient) behavior even if user threads busy-wait for each other, rather than using blocking operations provided by the runtime system. The excess kernel threads are deleted whenever there are no user threads for them to execute.

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 — a belief that is held by a large number of vendors, and also by many researchers [356, 48, 53, 550, 97, 198, 273, 217, 274]. 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

[16] The terminology used in MAXI for kernel and user threads is "envelopes" and "activities", respectively.

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.

The simple version of gang scheduling, where threads are mapped to PEs and do not move, is currently the most popular one. However, more flexible versions have also been proposed [449]. One of these is *migratable* preemption, whereby threads are preempted on one set of PEs and resume on another (typically disjoint) set. This may be used to reduce fragmentation. A more drastic version is *malleable* preemption, in which the job is resumed on a set of PEs with a different size from the original set, thus allowing the job to grow or shrink in accordance with system load. Migratable gang scheduling is trivially implemented in systems with swapping, but malleable gang scheduling is not currently used.

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[17] [438, 198, 499]. 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

[17] 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 [417].

allows threads to be associated with data structures in local memory [106]. Indeed, studies that compared gang scheduling with other scheduling schemes have concluded that gang scheduling is a relatively good policy [376, 511, 361, 249, 133, 106].

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 [203]. 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 [203] (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 [356], Meiko CS-2, Intel Paragon [288], Cray T3E [346], and Silicon Graphics multiprocessor workstations [48].

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 [299].

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 [201]. 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[18]. This can be done based on runtime observation of interactions among threads [196, 541]. 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 [57], `PAR` in Occam [285], and the `spawn` system call in Symunix [182]. Thus, when a set of threads are spawned together by a parallel construct, they are assumed to constitute a gang [438, 197].

Coscheduling and family scheduling

Two other terms that are often mentioned are coscheduling and family scheduling. *Coscheduling*[19] 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 [438]. 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 [354, 133]. Therefore it is not clear that attempting to schedule partial gangs when a full gang cannot be scheduled is worth the effort[20]. Just scheduling arbitrary threads on otherwise idle PEs can probably provide the same gains. Indeed, this is exactly what most systems do. Such scheduling of single threads is called *alternative scheduling* [438].

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 [85]. This can be done using a global queue or using local queues. 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.

[18] Ousterhout used the term "process working set" [438], based on the analogy with the working set of memory pages that must be simultaneously resident in primary memory for efficient computing.

[19] 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.

[20] 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.

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 implies swapping them to secondary storage [203]. 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 [365]. 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" [449], and can help in the reduction of fragmentation [346, 516].

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 [16] (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 [198]. 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 [48].

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 [550]. Another is to use a software broadcast interrupt [198]. A third option is to rely on synchronized clocks, that all cause interrupts on their respective PEs at the same time [217]. Once the processors are interrupted, they perform their local context switch. A number of Unix-based implementations do this using signals [241, 274] or changing the priority [270, 346].

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 [172]. 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 [365]. 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 [63, 234, 2].

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; it is used in the SHARE scheduler for the SP2 [217]. 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 [47] and the Intel Paragon [461]. 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.

A variant of this approach is to maintain multiple virtual circuits between PEs, and use a separate one for each job. In this case, the messages are left in the network until the job is rescheduled and receives them. However, this places a large burden on the network, and is not an efficient use of the limited buffering capabilities that are usually available. Therefore instead of full-fledged virtual circuits, each message is simply tagged with a job ID, and kept in a special queue if this ID does not match the ID of the currently running job on the receiving PE. If this queue overflows, the system is interrupted to offload these messages

and store them elsewhere. This is not anticipated to be a problem, since the window for sending messages from one PE before a context switch and receiving them at another PE after the context switch is relatively short, so there should not be too many such messages. This approach is taken in the *T-next generation machine from MIT.

The third approach is to drain the network as part of the context switch operation [274]. 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 [550], the Connection Machine CM-5 [356] (the CM-5 implementation is described in Section 7.2.4), and the RWC-1 [272].

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 [550], as embodied in the synchronous nearest-neighbor user channels provided in hardware by the K2 architecture [25]. 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")[567]. 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 [137, 467]. 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 (Fig. 21) [567].

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 [222, 339, 319]. 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 by Xylem, the Cedar operating system, 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,

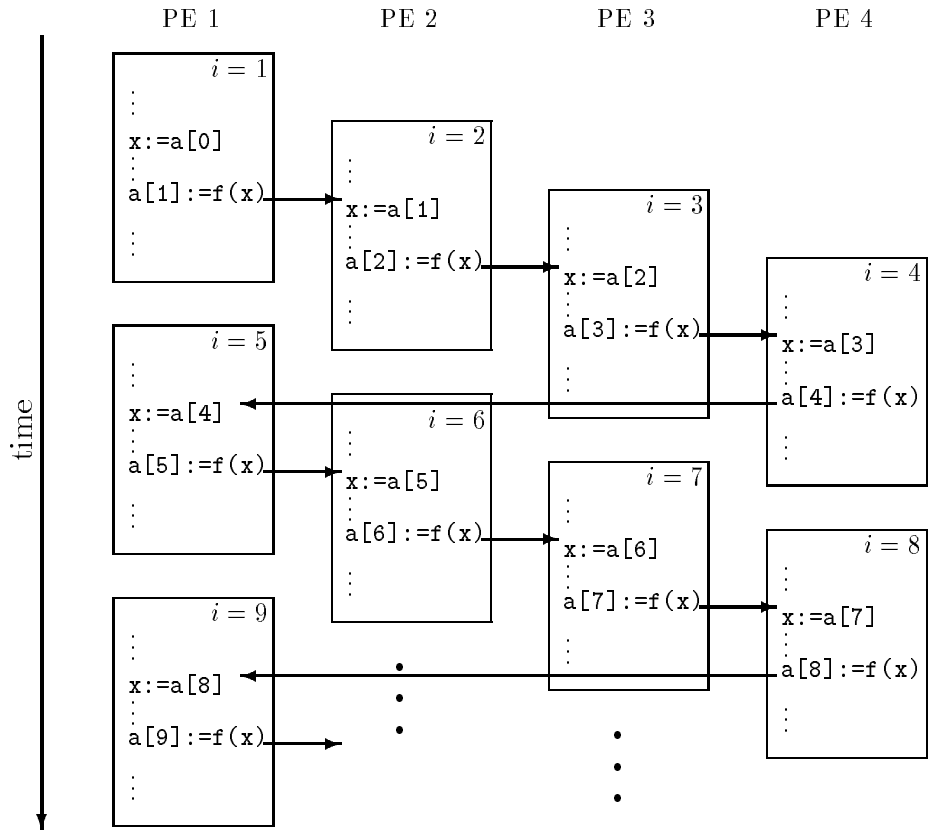


Figure 21: *Skewed execution of iterations from a doacross loop. The interdependencies between iterations imply fine-grain synchronization and a limit on the degree of parallelism (adapted from [567]).*

and if it finds a new task on the global queue, it copies the task to the local cluster [188, 187]. 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 [423].

A mechanism for gang scheduling combined with partitioning of the machine has been implemented in the Connection Machine CM-5 [356]. 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. Such operations are especially useful in the context of the synchronous MIMD model of computation, which is a loose approximation of the SIMD style used in the earlier CM-1 and CM-2 models.

The implementation of gang scheduling on the Connection Machine CM-5 is noteworthy because the architecture employs a buffered multistage network for interprocessor commu-

nication. 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[21] [356]. 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 [299]. 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.

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 [438]. The simplest is the matrix algorithm, which was implemented in the Medusa operating system on CM* [439], in the Meiko CS-2 operating system, in the gang-scheduler used for the BBN Butterfly at Lawrence Livermore National Lab [241, 240], in the experimental gang scheduling runtime library of MAXI, the Makbilan operating system [198], In the SHARE scheduler for the IBM SP2 [217], and in the DQT scheme designed for the RWC1 machine

[21] 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 is structured as a fat tree above them — see Fig. 29.

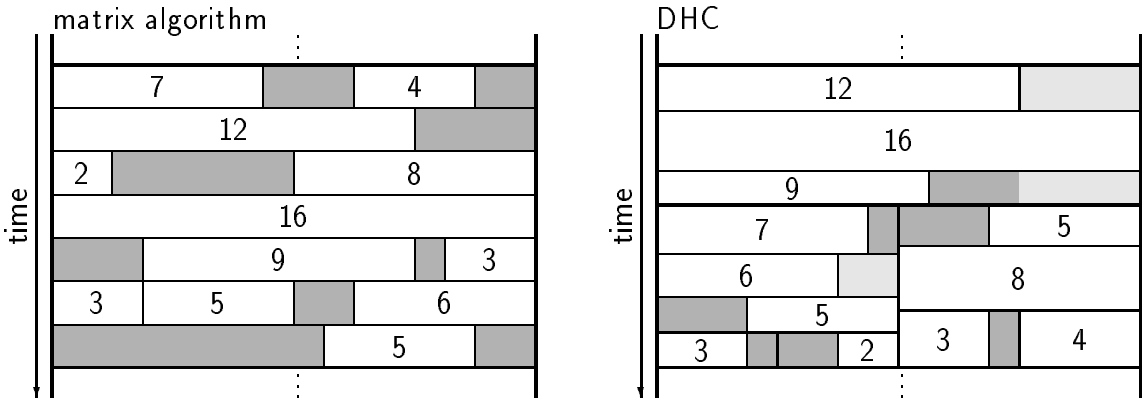


Figure 22: *Example of a scheduling round using the matrix algorithm and DHC. Gray shading indicates fragmentation.*

[273, 272]. 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. 22.

The other two algorithms developed by Ousterhout, which were not implemented, are based on a sliding window metaphor [438]. Threads are arranged in one long sequence, with the understanding that a thread in location i is mapped to PE number i mod 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 [438]. In other words,

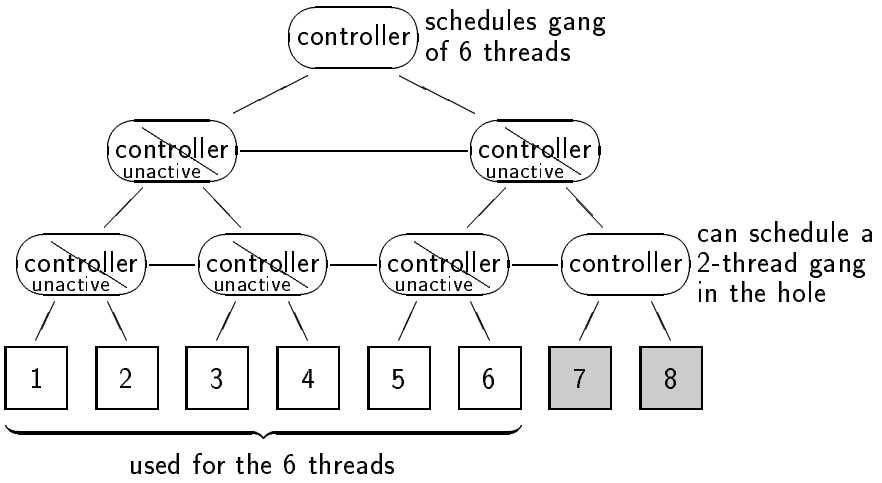


Figure 23: *Distributed Hierarchical Control arranges PEs in a buddy system. Small gangs are mapped to holes left by large ones, and execute at the same time with them.*

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 [197, 204].

The Distributed Hierarchical Control 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. 23). 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. 22.

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. With the Distributed Hierarchical Control scheme, 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 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 [201]. Fig. 23 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. 22. 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. This means that the alternative scheduling is also coordinated. 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 [204]. The time quantums in Fig. 22 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 [217] and the RWC-1 [273, 272]. 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. The Paragon scheduling scheme is described in Section 7.2.1.

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 [203], and described in more detail in Section 7.2.2. A variant based on feedback has also been proposed [516].

Performance guarantees

Due to the use of time slicing, gang scheduling may serve any number of jobs (as long as system tables do not overflow). This is conveninent to users, who can always gain access to the machine. However, it makes future performance unpredictable, because it depends on future load conditions. In some cases, a guaranteed minimal rate of computation is desired. This can be achieved by making reservations for a certain fraction of each scheduling cycle.

This idea in incorporated in the SHARE scheduler, which is based on the DHC structure [217, 545]. Jobs are classified according to their degree of parallelism (rounded up to the nearest power of 2). The system allocates a certain fraction of the time to each class, that is, to each level in the underlying control tree, irrespective of the number of jobs currently in that class. Thus it is possible to support different administrative policies, e.g. give preferential treatment to large jobs.

5.5 Theoretical Results

Off-line algorithms for multiprocessor scheduling, and the complexity of this problem, have been the subject of active research for many years [293, 238, 591, 590]. However, the model employed was usually that the interdependent tasks of only a single job are considered. There has been very little theoretical work on scheduling multiple parallel jobs, that each require multiple PEs simultaneously.

Scheduling parallel jobs can be modeled as 2-D bin packing, where one dimension represents PEs and the other represents time [127]. In the scheduling formulation, the input is a set of jobs, each characterized by the number of PEs it requires and by its runtime. A set of P PEs is given, and the problem is to find a schedule with the minimum possible makespan.

Nonpreemptive scheduling in the general case is NP-complete, as it includes the case of scheduling tasks that each require a single PE [167, 173]. However, preemptive scheduling (as is the case for gang scheduling) admits to polynomial-time algorithms [69, 638, 111, 524, 167]. For example, one algorithm is based on the observation that P is necessarily finite, and therefore there is only a finite number of ways in which the PEs can be divided among different jobs executing simultaneously [69] (this is even simpler for hypercubes, where the partitions are subcubes [111, 524]). The whole set of jobs can be scheduled by using different partitioning schemes for various durations, such that the total cumulative time of each partition size satisfies the requirements of jobs with that degree of parallelism. This leads to a linear programming formulation, where the objective function is to minimize the sum of the times that the different partitioning schemes are used.

The performance implications of gang scheduling have been the subject of less work. However, it has been observed that if a job's threads are scheduled in an uncoordinated manner, they will typically be spread out. As a result the job's response time (measured

until the termination of the last thread) will be long. If all the threads are scheduled at once, on the other hand, there will be no need to wait for threads that trail behind the others, and response time will be shortened [425].

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 [376, 384, 361, 249, 133, 198, 421, 106]. 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. Some of this repeats comments that were made previously, but here they are grouped by functionality rather than by scheduling scheme.

6.1 Interaction with Applications

Multiprocessor scheduling can have a great effect on performance, much more than in uniprocessors. This is so both in terms of the response time of the individual applications, and in terms of system throughput. 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. The way in which competing applications are handled affects the amount of resources that are lost to fragmentation and overhead.

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 [249, 198, 499].

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 [438, 198]. Note that this can be combined with preemptions, leading to gang scheduling, and is not limited to pure partitioning of the machine.

Systems that do not provide one-to-one mapping have to compensate by limiting the amount of time that one thread may wait for another thread that is not running at all. This can be done by simply blocking the waiting thread, and switching the PE to another ready thread. However, context switching may be costly, and it might be that the awaited thread

is running after all. The solution is to use two-phase blocking: first, busy wait for some time, to avoid the context switch overhead if the synchronization can indeed be completed quickly. Only if this fails, perform a context switch [438]. This policy is known to be competitive, meaning that it is guaranteed to limit the overhead to twice the overhead that would be experienced by an optimal off-line policy [301]. The competitive factor can be reduced if the distribution of waiting times is known [369].

Some systems go even farther and add interfaces to allow synchronization considerations to affect the scheduling decisions. The main idea is to enhance the service given to threads that have other threads waiting for them. Three mechanisms have been proposed in the literature: handoff scheduling, priority inheritance, and an interface that allows a thread to request to be temporarily immune to preemption. These were described in Section 4.4.2. A more subtle point is the use of preemptive time slicing in general: this is useful for limiting the time that threads have to wait before they can handle a synchronization event that arrived when another thread (from a competing job) was running [345].

The whole issue of interactions between the scheduling policy and the synchronization mechanisms is irrelevant for architectures where interactions suffer from high latency relative to the context switch overhead — and therefore cannot be considered fine-grained. This can happen in either of two ways. One is when interactions are extremely expensive, as in early message-passing systems. The other is when scheduling is very inexpensive, as in the Denelcor HEP [297, 321]. This machine could support up to 128 contexts in hardware on each PE, and performed a context switch on every instruction. Thus it actually implemented a processor sharing policy, which is equivalent to having all the threads mapped to dedicated (albeit slower) processors. However, if not all jobs fit into the hardware contexts, it is imperative that either all threads of a job be accommodated or none at all. This can be considered a special case of gang scheduling. It is noteworthy that while processor sharing has a positive effect on synchronization, it does not necessarily improve overall response time. This is because it slows down everything, including short threads that would have completed within a single quantum [577].

Finally, synchronization is often accompanied by data transfer. Thus the scheduling policies may also have an effect on buffering requirements. Consider a producer-consumer relationship as a concrete example. If both threads run simultaneously, a small buffer may be sufficient to store produced items waiting for consumption. But if only the producer is running, a large buffer is required to store the backlog of items accumulated until the consumer is scheduled. Buffer overflow in a message-passing system could even lead to deadlock, unless specific precautions are taken [343, 356].

Support for memory locality

Programming for locality is widely recognized as an important paradigm. Specifically, locality of reference reduces interprocessor communication in multicomputers [487] and improves cache performance in multiprocessors [267, 348]. 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 [487, 388, 389]. Hence locality considerations indicate that threads should be mapped to PEs and not moved. The other way the operating system can cause problems 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. A number of studies have demonstrated the adverse effect of context switching on cache performance [408, 249, 598]. 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 [598, 157, 38, 544, 576, 106]. It has even been proposed that affinity hints be included in the programming language [107].

Note that the adverse effects caused by the operating system are related in both cases to the preemption of running threads. Thus memory-usage considerations lead to a preference for non-preemptive scheduling, leaving partitioning as the only means for sharing the machine. However, partitioning by itself does not guarantee efficient use of local memory. If the application does a second level of internal scheduling, it might undermine the potential for exploiting the locality.

The problem of locality does not exist in UMA[22] shared-memory machines, because then all memory is equally distant from all PEs. However, practically no machines support a pure UMA architecture. Just having a cache attached to each PE is enough to create the effect of a NUMA architecture. Black has suggested the following dichotomy to determine whether an architecture should be classified as UMA or NUMA from a scheduling point of view [64]: if performance is served by always scheduling a thread where its data resides, even at the expense of leaving another PE idle, the architecture is NUMA. If, on the other hand, it is better to keep PEs active, even at the expense of remote access to data, the architecture is UMA.

Effect on communication locality

Communication is an important aspect of parallel applications. In many systems, communication costs depend on distance. For example, in early hypercubes it was common to estimate communication overhead based on the number of hops the message had to traverse. Thus allocating PEs from distant parts of the machine to the same job can lead to degraded performance relative to the same job running on a localized set of PEs. However, this dependency has been decreasing in new architectures, due to wormhole routing and the use of lower-dimensionality networks [512, 139, 138].

Even if communication *per se* does not depend on distance, using distant PEs can still

[22] UMA stands for Uniform Memory Access, as opposed to NUMA, which is Non-Uniform Memory Access.

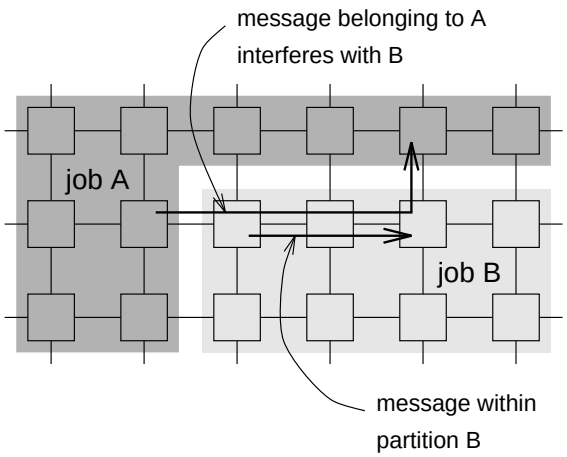


Figure 24: *Due to the predefined routing scheme, concave partitions on the Paragon may cause interference.*

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 [375, 413, 403]. Traffic belonging to the first job then passes through the links in the partition belonging to the second job (Fig. 24).

Locality can also be affected by a mismatch between the application's programming model and the actual architecture, as when embeddings are used [380]. It is irrelevant for "topology-less" architectures like multistage networks (i.e. where the topology is not exposed to the programmer), or UMA machines.

6.2 Impact on System Performance

While the interaction between the scheduling scheme and the application has great impact on the application's performance, it does not tell the whole story. The way in which competing applications are handled is just as important. It 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. This is partially due to synchronization overhead and latency for remote access, as described above, but there are also other factors.

Scheduling schemes based on static partitioning (and gang scheduling) encourages the

use of as many PEs as possible, in an attempt to achieve higher speedups. However, adding PEs typically suffers from diminishing returns, so this may come at the expense of efficiency [179, 600, 134, 213, 133, 327]. In a highly loaded system, it might actually even degrade response time [517]. Adaptive and dynamic partitioning, on the other hand, only gives a job more PEs if there is nothing better to do with them [327, 628, 517, 249, 421, 396]. When multiple jobs compete for resources, each is limited to a relatively small number of PEs, thus boosting its efficiency. A similar principle has been applied in compile-time parallelization and scheduling, where multiple code blocks are executed simultaneously, each with limited parallelism, rather than trying to parallelize each block to use all the PEs [546].

The other side of the coin is that limiting the number of PEs allocated to an application eliminates the possibility of super-unitary speedup, as sometimes happens if many PEs are used simultaneously and the necessary data fits into their combined caches, but not into a smaller number of caches [260, 554]. Even worse, in many systems limiting the number of PEs also limits the amount of primary memory available for the job. If this becomes too small for the job's dataset, paging has to be used. The overhead for the paging typically outweighs the benefits of increased efficiency [452, 515]. Such effects can be reduced by establishing a minimal partition size [421].

Effect on utilization and throughput

One motivation for multiprogramming on uniprocessors is the desire for interactive response times. Another is to increase the utilization of expensive computers. This is achieved by overlapping the computation of one job with the I/O of another, using time slicing [322, 456].

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 [132]. 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 [558, 197, 594, 464].

Another counter argument is that not all jobs are computation intensive. Some require significant amounts of I/O. In the past, such applications were rare, because parallel machines did not have the I/O facilities to support them. But as parallel I/O mechanisms are developed and gain recognition, more and more I/O intensive jobs will appear [153, 202]. The system will then benefit from the possibility of overlapping the I/O of one job with computation of

another, just as in uniprocessors. This can be based on collective I/O operations[23] and gang scheduling. As in uniprocessors, jobs that perform I/O should be given higher priority when they complete the I/O operation [354].

In addition, I/O is not the only thing that threads might wait for. In many cases, considerable idle time can be attributed to waiting for communication or synchronization. If the interaction among threads is coarse-grained, then such waiting time can be overlapped with computation of another thread, possibly belonging to another job [198, 518].

Also, it should be noted that utilization is strongly (inversely) correlated to loss of resources to fragmentation. Time slicing may reduce fragmentation, as shown next.

Loss of resources to fragmentation

One of the drawbacks of partitioning is that it may suffer from internal and external fragmentation. Fragmentation depends on the distribution of partition sizes that are requested and granted, so the degree of harm depends on workload characteristics and how the system matches the workload. 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. If all the applications use the full machine size (or a full predefined partition) all the time, as is possible with dataparallel codes that only define the size when the program is loaded, then there is no fragmentation. Otherwise fragmentation can be a big problem leading to significant losses [364, 363, 204, 375, 402].

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 [581, 633, 133]. Changing the partition size compensates for both internal and external fragmentation. However, this approach may not be suitable for other programming models.

Another solution is to use migration to reduce the fragmentation [114, 110, 346, 449], much as compaction is used to reduce the fragmentation in memory allocation. This is applicable when there are restrictions on how PEs are allocated, as in the Cray T3E which requires a contiguous set of nodes [346], or subcube allocation in hypercubes. The problem with this approach is that migration requires a potentially large context to be copied, which loads communication resources [571]. Also, this does not solve internal fragmentation. The only way to reduce internal fragmentation is to remove the restrictions on PE allocation [375, 606].

A third solution is to reduce the effect of fragmentation by combining partitioning with time-slicing. This works because scheduling is essentially an *on-line* activity, typically with no knowledge of the future. When Partitioning is used without time-slicing, decisions can

[23] In collective I/O, all the PEs perform the I/O at the same time and synchronize as part of the operation. This synchronization point is a good candidate for performing a multi-context-switch.

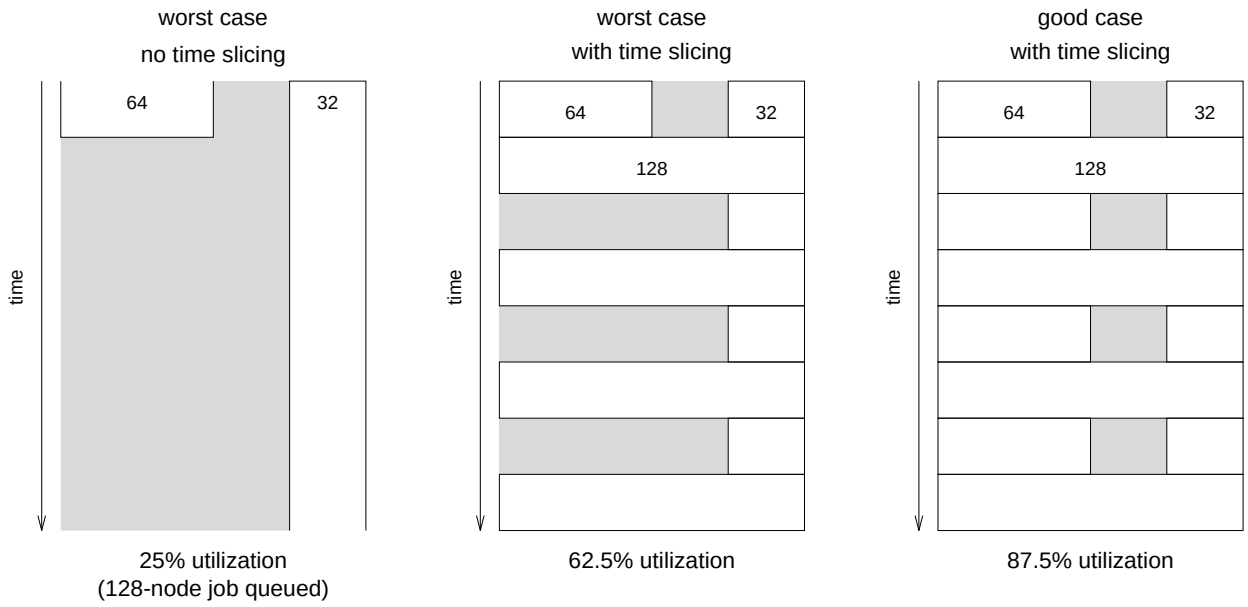


Figure 25: *Example of how the flexibility afforded by time slicing can increase system utilization.*

have a large impact on the future. Consider the following simple case as an example: 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%. And, in order not to starve the 128-node job, it might be necessary to just let the 64-node job run to completion, and settle for 50% utilization.

This bleak outlook changes if gang scheduling is used (Fig. 25) [203, 504]. 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.

When using time-slicing, fragmentation can be reduced even further by giving less service to problematic cases [204]. Using the above example, when the 32-node job runs alone it will receive shorter time slices than the 128-node job. This seems to come at the expense of fairness if you just look at the service that applications receive; it is quite fair, however, if you consider what it costs the system to provide this level of service. It also has the side benefit of keeping the fragmented configuration in the system for a longer period of real time,

thereby increasing the chance that new arrivals will be able to utilize the left-over PEs.

Operating system overhead

Bad scheduling decisions may add overhead by undermining synchronization, reducing locality, and other adverse effects. But there is also direct overhead for the scheduling itself.

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 [162, 161]. 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 [581]. 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 [424].

Context switching operations are not the only source of overhead. Another important issue is contention for locks and shared data structures. In the context of scheduling, contention for a global queue of ready threads can lead to a severe degradation in performance [426, 431]. Another effect of operating system overhead is that it may perturb the execution of an application due to its asynchronous nature. This can be eliminated by coordinating the perturbations on the different PEs, essentially performing a multi-context-switch from the application to the operating system and vice versa [417].

Support for general purpose parallelism

Finally, on a slightly more philosophical level, scheduling is implicitly implicated in the context of support for general purpose parallel computation. A major obstacle in this area is the need to tailor applications to architectures, so as to achieve optimal performance [552]. The alternative is to hide the deficiencies of the architecture, or the mismatch between architecture and application.

The main problem facing parallel applications is the latency of interprocessor communications. Valiant has suggested that this can be hidden by parallel slack: if the logical parallelism in the application is higher than the physical parallelism (i.e., the number of PEs), then when one thread blocks waiting for an interaction to complete another is scheduled in its place. This allows for general-purpose machines that are not specifically designed for a certain type of computation [593, 393]. Actually this concept had been implemented already long before, in the multithreaded architecture of the Denelcor HEP [539, 297, 321].

While that machine is no longer available today, its legacy continues in the Tera machine [18, 96, 15, 16].

As far as scheduling is concerned, this idea is related to independent time-slicing on each PE, using a local queue. The difference is that the scheduling is done in hardware, rather than by the operating system.

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) [322, 212]. 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 [168]. It is also needed in order to exploit various theoretical results about algorithm design and scheduling. For example, a lot of work has been done to develop heuristics for the scheduling of interdependent tasks on a set of PEs [238, 120, 377, 232], but these heuristics are based on the assumption that the PEs are there for the taking and nobody interferes with their use. Likewise, parallel algorithms developed in the PRAM and other models are based on this assumption [302]. Algorithms are simply not designed with the possibility of PEs suddenly switching to run competing applications in mind. The only exception is work done in the context of an extreme model of fault tolerance, that considers the environment in general to be an adversary. This work shows how to emulate a PRAM correctly despite anything that the environment (including the operating system) may do [33].

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. Processor sharing by virtue of hardware contexts, as on HEP, also provides an effect of dedicated hardware. In this case limitations are due to the limited total number of contexts. However, these are typically high enough to be of no concern. 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).

The question remains of how crucial it is to implement the dedicated machine model by actually dedicating a set of PEs, at the same time, for the exclusive use of an application. The answer depends on the characteristics of the application. Providing all the PEs at once reduces the overhead associated with synchronization, especially if busy waiting is used and interactions are fine-grained [198]. When two-phase blocking is used rather than pure busy waiting, there is a danger of increased context switching for synchronization if

the threads do not run simultaneously. If interactions are asynchronous and use buffers to communicate data, the lack of dedicated PEs may lead to buffer overflow. Finally, when PEs are dedicated to the application, so is their memory, which provides strong support for local data referencing. Without dedicated PEs and one-to-one mapping, we might have to pay more for data movement.

In many cases the application reflects the model of computation. If the threads are largely independent and have a small context, the application will not notice that it does not have a dedicated machine. If the threads participate in considerable interactions, and require significant local storage, support for the dedicated machine model is highly desirable. As a concrete example, consider the case of dataparallel applications. In this programming style, large data structures are partitioned across the PEs. Barrier synchronization points typically delimit phases of the computation, sometimes accompanied by data exchanges. These features indicate that good support for the dedicated machine abstraction is required in this case.

Finally, good support for the dedicated machine model improves the robustness of the scheduling scheme. For example, gang scheduling has been shown to be consistently slightly less efficient than a real dedicated machine. Dynamic partitioning, on the other hand, is sometimes much more efficient, but sometimes much less efficient, because of complex interactions between the way the threads and data happen to be mapped. Thus overall gang scheduling is more robust [106].

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 cube multistage networks like those used in PASM and TRAC [531, 533, 372], in the fat tree network used in the Connection Machine CM-5 [356], in crossbars like that of the Fujitsu VPP500 [407, 592], and in hypercubes or meshes where applications run on subcubes or submeshes. Moreover, some systems allow users working in different partitions to use different operating systems. This is a useful feature for system development: new and experimental versions can be tested in one partition, while users continue to use a stable older version in others [52, 151, 350]. It may even be possible to reboot a partition without any effect on jobs running in other partitions [129].

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 [459]. This may block memory accesses from threads in other applications. The severity of this problem and possible solutions are the subject of considerable controversy [14, 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 essence, when a PE runs a thread from a certain application, it loads a memory map in which only memory accessible by this thread is represented. Attempts to access unrepresented memory result in a memory fault. The problem with this scheme is that the maps must be kept consistent on all the PEs. This is known as the TLB[24] consistency problem [67, 488, 566]. The problem does not exist if maps never change, but this can only be done when applications fit into physical memory and never move. In most cases, this places unacceptable limitations on the memory allocation.

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. This increases the cost of message passing. The effect of applications on each other is reduced if whole sets of PEs are allocated to a job together with a partition of the network, rather than scheduling PEs individually. The network is then flushed as part of each context switch. A mechanisms for such flushing exists in the Connection Machine CM-5 [356].

User satisfaction

The introduction of time slicing to uniprocessor systems not only allowed the system to be shared, but also provided interactive response times. This was quickly recognized as a revolutionary improvement in computer use, leading to increased user productivity [471, 51]. The same can be said of time-sliced parallel systems as well. Conversely, in a pure space-slicing system, users may be denied access to a partition of the required size for arbitrarily long times [360].

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) [322, 456]. 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

[24] TLB stands for Translation Lookaside Buffer. This is the common name for a special cache used for frequently accessed mapping information.

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 [485, 450]. As measurements of actual workloads have repeatedly shown that job service demands have a large coefficient of variation [195, 201, 617, 276, 606], the conclusion is that time slicing will lead to reduced response times.

As massively parallel supercomputers become more common, multiuser support becomes more important [78, 151]. Likewise, user frustration with poor access or bad response becomes an issue [251]. Pure space-slicing systems might contribute to user frustration in two ways, especially if the partitioning is not flexible enough. One is if the number of partitions is severely limited. For example, a full-size Connection Machine CM-2 could only be partitioned among four users, because it only had four sequencers [582]. There was no way for multiple users to obtain access to small partitions of the machine for program development and testing.

Another cause for frustration is if an allocation to one user prevents allocation to another, especially when the total number of available PEs is actually sufficient. For example, the allocation of a 4-node subcube in a hypercube can prevent the allocation of a 512-node subcube in which it is contained, even if the total number of idle nodes in the system exceeds 512. This problem may manifest itself in the form of interrupted sessions: if PEs are allocated as part of program execution, it may be possible to run a program once, find a bug, but not be able to run it again because the PEs had been allocated to someone else in the meanwhile [303]. The solution of allocating PEs for the duration of a login session is undesirable because these PEs are never used when the user stops to think or to take a break.

In practice the most common manifestation of these problems is that users are faced with uncertainty regarding when they will be able to run their programs. Reducing the uncertainly is just as important as reducing the response time itself. Methods to do so include placing limits on running jobs, as is done in NQS, and trying to estimate the queueing time of new jobs [368, 225, 164].

Some partitioned systems provide batch scheduling as the only support to ease these difficulties. Others do not provide even this limited support, and implicitly expect users to coordinate the use of the machine verbally among themselves [151, 303]. This is not always easy, as different users often have conflicting goals: some want to use the whole machine for long production runs, while others want small partitions for program development and testing [605, 400]. Electronic signup sheets can be used to reserve a partition. The system can then check who should be allowed to run, and use this information to enable the authorized users to kill unauthorized jobs that interfere with their work [287]. But users might abuse signup sheets by signing up for whole days, weeks in advance. Placing a limit on the number of PEs that users can monopolize, and on the time they can do so, is a possible compromise [64]. However, the limit may be too tight for users who really need the full machine for long

production runs, and might not be tight enough for users who spend large parts of the day waiting for a relatively small number of PEs. The only solution that allows users to feel as if they are sharing the machine at the same time is to use time-slicing. Note that this does not necessarily have to come at the expense of partitioning. Rather, gang scheduling may be used to get the best of both worlds. For example, gang scheduling was implemented on the CM-2 to improve its ability to support multiple users at once [299].

Preemption can also be useful in reducing user frustration due to unexpected program behavior. For example, consider a program that uses busy waiting to synchronize a number of threads. If enough PEs are available and all the threads run simultaneously, the program executes faultlessly. But if less PEs are available, the running threads have to wait in order to synchronize with threads that are not running. If the system is non-preemptive, this scenario results in deadlock. Thus preemption reduces the dependence of program behavior on the number of PEs [549, 57, 563]. While preemption is possibly costly in performance, it should be realized that not all users are perfect. The reduction in user frustration might be worth the cost.

Fairness and accounting

In partitioned systems the notions of fairness and accounting are simple. Each application gets a partition of the machine, and is billed according to the size of the partition and the time it was held. Fairness may be introduced by requiring that concurrently executing applications have equal numbers of PEs in their partitions. Conversely, it is possible to allow applications to acquire larger partitions, and bill them accordingly.

In time-slicing systems, the issues of fairness and accounting boil down to the question of what time slices to give to competing threads and applications. A quest for fairness can be interpreted as implying that time slices should be equal at the job level [361]. Thus 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 [241]: 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 [82, 333, 384, 204]. 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[25].

[25] 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

The problem of user countermeasures is solved by accounting. It seems better to leave sophisticated considerations out when allocating processing power. Rather, accounting should be used to penalize applications that make excessive requests. In other words, both the cost of running a parallel application and the service it receives should be proportional to the number of threads in the application.

A more general solution is to use accounting as a feedback mechanism that influences scheduling. As parallel supercomputers are generally quite expensive, it is often the case that multiple groups of users cooperate in the acquisition of a single machine. Accounting is then used to ensure that users from the different groups receive service that is proportional to their investment. But in order to be effective, the scheduler must take the accounting information into account when deciding on the allocation of resources to new jobs [263, 304, 191].

The naive solution in parallel systems is to partition according to investment: a group that put up 25% of the budget gets a dedicated 25%-partition of the machine. But this means that the full power of the machine is never utilized, and that resources not used by one group cannot be easily transferred to another. In effect, the different user groups could have bought separate, smaller machines. Therefore it is better to use time-slicing, and set scheduling quanta according to the allocation to different user groups. This approach has been used on uniprocessors in the Unix fair share scheduler [263, 304, 191][26], and related policies are incorporated in the Cray T3E scheduling software [346] and in the Tera MTA scheduler [16]. It can also be used to impose job-level fairness, by regarding each job as representing a separate user group which deserves an equal share of the machine. Alternatively, an economic model model can be used to control the allocation. In such a model, each group has an "income rate" proportional to its investment in the machine, and the priority of the group's jobs is proportional to its accummulated "funds". Thus jobs belonging to a group that owns a larger portion of the machine will have a higher priority and run sooner [551].

It should be noted that fragmentation may cause a discrepancy between the service provided to the user and what it costs the operating system to provide this service. For example, a job using a small number of PEs can prevent another large job from running, and leave PEs idle owing to fragmentation. From the system's point of view the cost is a combination of the PEs allocated to the running job and the idle PEs, but the job only benefits from the ones it uses. When gang scheduling is used, determining the time slices based on system cost rather than on user benefit can improve resource utilization [204]. However, only the provided service should be used for accounting.

6.4 Implementation Issues

Choosing a scheduling scheme based on expected functionality and performance is one thing. Actually implementing it is another thing. For example, implementations might be con-

computation in the same job. This will degrade the service to all jobs, including the one that created the spurious threads.

[26] There has also been work on how to set the priorities in the conventional Unix scheduler, which uses decay usage scheduling, to achieve the desired division of resources [259, 189].

strained by architectural features or by lack of hardware support.

Interaction with memory management

Memory management is an important service provided by uniprocessor operating systems. The essence of this service is to strike a balance between the memory requirements of different processes. The solution is to keep each process's working set of pages in the main memory [155]. Most of the time, the process only needs these pages. When a process tries to access a page that is not in main memory, a page fault traps to the operating system, and the page is brought in.

Memory management in multiprocessors has an additional dimension, 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 [74, 66, 502, 351]. Of course, this issue is not relevant for distributed memory machines, where the page must be collocated with the thread that accesses it, or for UMA shared-memory machines, where all memory is equally accessible.

Regrettably, the interaction of memory management with scheduling in parallel systems has received only scant attention [452, 88, 396, 106, 515, 448, 447, 397, 550]. 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 while waiting for a page fault to be serviced [452, 515, 256]. 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 [323]. 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 [448].

While memory availability is indeed growing at an amazing rate, so are processor speeds and user requirements. Therefore requiring jobs to fit into available memory forces users to take memory availability into account when designing programs, and might limit the computations that can be performed [536][27]. Providing virtual memory is increasingly recognized as a necessity [532].

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. However, it should be noted that space slicing also reduces the memory available to each job, and actually time slicing allows more flexibility in the allocation, because memory allocation is decoupled from PE allocation (Fig. 26).

[27] Interestingly, memory availability has also been proposed as the metric for scaling the problem size when analyzing speedup bounds [618, 559].

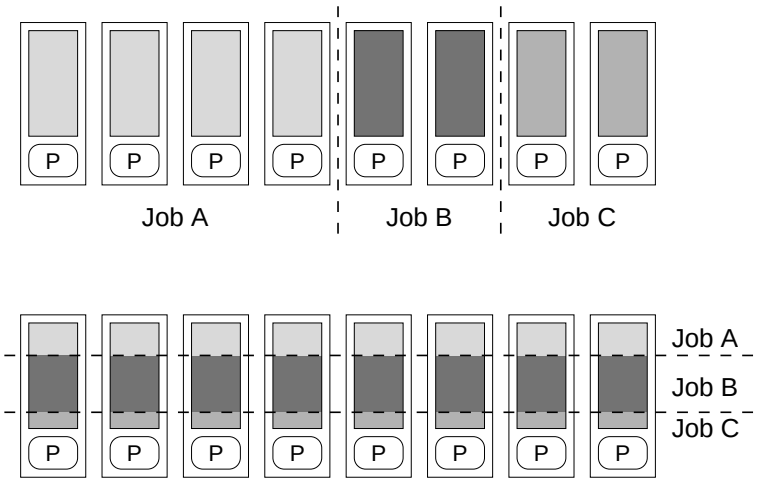


Figure 26: *With space slicing, processor allocation dictates memory allocation due to the "vertical" partitioning (top). With gang scheduling, memory is partitioned "horizontally", so memory allocation is decoupled from processor allocation (bottom).*

Memory management in time slicing systems is based on the distinction 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 [607, 88]. While swapping is a basic feature in most uniprocessor systems (e.g. Unix [36, sect. 6.1 and 9.1]), very little work has been done in this respect in the context of parallel systems [16, 550, 88, 396]. One notable idea, however, is that the residence time of jobs be proportional to their memory footprint [16, 203, 516]. This places an upper bound on the relative overhead of moving the job's state from memory to disk and back.

To summarize, in both cases the problem with memory management is how to avoid stalling PEs when waiting for memory faults to be serviced. An interesting alternative approach to this question is to perform double buffering in memory [583]. This assumes that each job can fit into *half* of the available memory. Successive jobs use the two halfs alternatively. As each job runs using one half of the memory, the other half is used first to offload the results of the previous job, and then to preload the code and input for the next job. This saves the startup costs for new jobs by overlapping them with computation of other jobs.

Finally, it has also been suggested that the problem is too complex for a general solution, so it should be left to the users. Specifically, given that physical memory is scarce, the application itself should be responsible for its optimal management. The mechanisms to do so are control over what pages of virtual memory will be cached or replaced, and facilities

to acquire additional frames of physical memory for bounded durations [256].

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 [531]. 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 [535, 534], or gang scheduling, as on the Connection Machine CM-2 [582, 299]. As another example, a shared memory is required for the implementation of a shared global queue of ready threads.

A more subtle effect of the architecture exists in various instances of hardware support for special operations, or lack thereof. We already mentioned the issue of hardware support for virtual memory. Time slicing puts more strain on memory, because it is shared by a number of applications. If there is not enough memory to go around, virtual memory and paging are needed. The problem is that not all parallel systems support paging or have enough local disks for swap space; the lack of support might preclude the use of time-slicing. Paging also implies asynchronous delays, which undermines the benefits of one-to-one mapping and gang scheduling [607]. When a page fault occurs in a gang-scheduling system, it is necessary to deschedule a whole set of threads, not just a single one [550]. However, this only makes sense if the multi-context-switch overhead is smaller than the page-fault service time [88].

On the other hand, some architectural features make sharing easy. A case in point is architectures with multiple hardware contexts and switching on every instruction cycle, like HEP [297, 321], Tera [18, 16], and Alewife [3]. These architectures provide real processor sharing in hardware: in effect, the whole machine is time-shared by all jobs. The operating system just has to map threads to PEs.

The multi-context-switching operation that is needed to implement gang scheduling is an example of an operation that is unique to parallel systems, and may require substantial hardware support. The main problem is that the context switching operation has to be synchronized across a set of PEs. In particular, it is highly desirable that the time during which they are not running the designated set of threads in parallel be very short relative to the scheduling time quantum. But due to clock drifts and communication latencies, it is not realistic to expect standard interprocessor communication mechanisms to meet this requirement. This is why most systems that implement gang scheduling use a broadcast interrupt to initiate a multi-context-switch [356, 198, 550].

In some architectures, getting all the PEs to do a context switch at the same time is not enough. The reason is that the state of a user's computation is not confined to the PEs: some of it may be in transit in the network. This should also be saved during a multi-context-switch. Again, special hardware support is required. For example, the design of the Connection Machine CM-5 includes a special feature to flush the network upon a context

switch [356]. Of course, it is also necessary to restart these communication operations when the application is scheduled again [365].

When considering the difficulty in implementing a multi-context switch, it is interesting to note that it is a subset of the work required to implement checkpointing and rollback [365]. This is an important observation, because checkpointing and rollback often play a crucial role in fault tolerance schemes. In essence, checkpointing records a snapshot of the program state, and rollback restarts execution from a previous snapshot. Multi-context-switching is actually somewhat simpler: it freezes the execution state without creating a copy of it, and then restarts it later. Thus implementing checkpointing and recovery implies an ability to implement multi-context-switching, with the added benefit of possible relocation.

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, as shown above. 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.

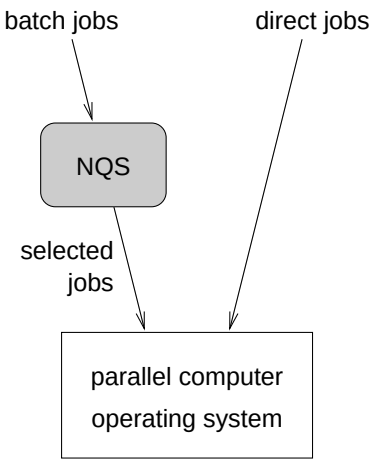


Figure 27: *The place of NQS in a system.*

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 [211]. One of the first was NQS, whose interface has become a *de-facto* industry standard, while the implementation may be different on each machine. Recently other systems have also been proposed, mainly to allow added flexibility for individual installations. 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 to help support efficient use of their Cray 2 and Cray Y-MP multiprocessor supercomputers [323]. It has since been adopted by many installations, as well as by the vendors of various parallel machines.

NQS is essentially a batch scheduling facility. Efficient batch scheduling depends on getting a good job mix, so that the requirements of different jobs complement each other and promote efficient utilization of all available resources. On the Cray multiprocessors, NQS attempts to optimize the use of two resources: memory and CPU time. Submitted jobs must state limits on their CPU and memory usage. Jobs are classified according to these limits, and each class has a separate queue. The scheduler then picks jobs from the different queues to create the desired mix. Jobs that exceed their time limit are killed.

As supercomputers typically have proprietary operating systems, it is impossible to control the scheduling on the target machine directly. Therefore NQS is implemented as a set of scripts that control the load that is placed on the machine. In other words, users submit

time	*number of nodes*			
limit	16	32	64	128
20 minutes	`q16s`	`q32s`	`q64s`	`q128s`
1 hour	`q16m`	`q32m`	`q64m`	`q128m`
3 hours	`q16l`	`q32l`	`q64l`	`q128l`

`Table 4:` *Example of NQS queues used on the 128-node iPSC/860 system at NASA Ames [195].*

jobs to NQS, which buffers the requests in its queues; NQS then submits selected jobs to the machine so as to create the desired job mix. For Cray multiprocessors, the goal is to have slightly less than two batch jobs per processor on average, such that their combined memory requirements do not result in swapping. Two are needed to allow the processor to overlap computation with possible I/O operations. By keeping the load at less than two on average some resources are left free for interactive users that bypass the NQS system (Fig. 27).

NQS was originally designed for throughput-oriented computation with sequential jobs. It has since been extended to support parallel programs as well, on distributed memory architectures. In fact, versions of NQS are provided by most vendors of parallel supercomputers. The common approach is to simply add a specification of PE requirements. Again, multiple classes are created, each with different limits on the number of PEs used and on the runtime. An example is given in Table 4, where there are 4 possible partition sizes and 3 time limits, leading to a total of 12 queues. 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. For example, only queues `q16s`, `q16m`, `q32s`, and `q32m` from Table 4 are enabled during prime time, effectively limiting jobs to 32 PEs and 1 hour. Any larger jobs that are submitted are queued until the other queues become active at night.

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 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 5). 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

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 5: *Example of NQS queues used on the 416-node Paragon at San-Diego Supercomputing Center [606]. 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.*

policies can be expressed. An example is the *Portable Batch System* (PBS), also developed at NASA Ames [262].

PBS structures job control functions using four types of processes. The *job server* is the focus of the system: it maintains the jobs in various queues, and interfaces to the hosts. Each host has a *machine orienter miniserver (MOM)* process that handles the actual execution and monitoring of jobs on that host. Each host also has a *resource monitor* process to gather resource usage information. The scheduling policy is handled by a fourth external process that has access to all the systems knowledge regarding resources, running jobs, and queued jobs. This process makes the decisions, and instructs the job server (and through it, the MOMs) to carry them out.

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 [262]. 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.

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. In addition, other partition types can be defined. In many cases, and I/O partition is used. This includes I/O nodes, that serve as an internal parallel file server [202], and gateway nodes, that interface to external networks and high-speed channels such as HiPPI. In very large machines, a partition of spare nodes responsible for availability and fault tolerance can be used [391].

While the central concepts used in all machines are very similar, the details vary. The following subsections describe some specific examples.

The IBM SP2

The IBM SP2 is a distributed-memory multicomputer based on a multistage network [5, 553]. The nodes are RS/6000 workstations. A number of models are available, based on traditional multi-chip module implementations or on PowerPC microprocessors. Furthermore, the packaging may be in "thin" nodes, "wide" nodes, or "high" nodes. Wide nodes occupy the full width of the rack, and contain more space for memory and peripherals; they are meant to be used as servers or as compute nodes for extremely demanding applications. Thin nodes occupy half the space and can contain less memory and peripherals. High nodes

are 8-processor SMP nodes based on the PowerPC chip. All nodes run the AIX operating system (IBM's version of Unix).

The nodes of an SP2 system can be configured by the system administrator into a number of partitions for different uses. These include the interactive login partition, server partitions (e.g. I/O nodes for a parallel file system [131]), a batch serial partition, and one or more parallel partitions for running parallel jobs.

Scheduling of batch parallel jobs is handled by LoadLeveler, which is based on Condor [373]. This system was originaly designed to run background serial processes using idle cycles in a network of workstations. It uses daemons on the system nodes to collect load information and to control the placement and execution of processes. This infrastructure is also useful for a parallel environment.

In order to schedule parallel jobs, LoadLeveler has been augmented with an external scheduler module, which is based on the EASY scheduler developed at Argonne National Lab for their 128-node IBM SP1 system [368, 538]. EASY (Extensible Argonne Scheduling sYstem) uses a backfilling algorithm to move small jobs ahead in the queue, provided they do not delay the first job in the queue (see Section 3.2.3). However, this algorithm is not mandatory, and installations can tailor the scheduling policy to their liking. This is based on the fact that the scheduler is an external module, and can easily be replaced or modified. The interface between LoadLeveler and the external scheduler allows the scheduler to obtain information about the current status of all nodes in the system, and about the job queue. The scheduler uses this information to decide what job to run, and on which nodes. LoadLeveler then performs the actual loading and execution of the job.

The Intel Paragon

The Intel Paragon is a distributed-memory multicomputer with a 2D mesh topology. Each node has two Intel i860 processors: one for computation, and the other acting as a communication co-processor[28]. Configurations with up to a few thousand nodes have been installed. This represents a second-generation machine, after the first-generation hypercubes (the iPSC, iPSC/2, and iPSC/860), and based on considerable experience with the Touchstone Delta prototype [400]. Processors used by applications run the OSF/1 Mach-based microkernel with a Unix server [634], while the message-passing coprocessor runs code that implements the NX/2 message passing protocols [461]. User applications running on the compute processor are linked with NX/2 libraries that invoke the other processor.

The Paragon provides a generous and flexible partitioning scheme. A hierarchical structure of partitions is maintained, rather like a hierarchical file system [288]. The analogy is extended to protection mechanisms: partitions have owners, and support `read`, `write`, and `execute` permissions for the owner, the owner's group, and other users. `Read` permission allows one to list subpartitions and jobs running in the partition, `write` permission allows

[28] A 3-processor MP node was introduced later, with the option to use all three for computing rather than dedicating one to messaging. The Intel/Sandia TeraFLOP machine, which is based on the Paragon, uses nodes with four Pentium Pro processors [391].

the creation of subpartitions, and `execute` permission allows the execution of jobs in the partition. The root partition, called '.' (dot), contains all the nodes in the system. By convention, there are always at least two subpartitions: `.service` is used to login, edit, compile, and launch jobs. The system automatically maps each new process to the least loaded node in this partition. `.compute` is used to execute parallel jobs. In addition, I/O nodes are often grouped into an I/O partition.

In addition to protection modes, partitions also have specific scheduling characteristics. There are three options:

- *Standard* — normal Unix time slicing is used to share the node among all the processes that are mapped to it. This is used in the service partition.

- *Gang scheduling* — this allows coarse-grain time sharing with coordinated scheduling of the whole application. The time quantum used is also a property of the partition, and can be as large as 24 hours (the default is 10 minutes). It is called the *rollin* quantum, invoking the image of rolling applications into and out of main memory.

  Gang scheduling is done recursively following the partition structure. For each partition, the entities in it (jobs and subpartitions) are examined. Those that do not overlap any other entity are scheduled for a rollin quantum. For entities that do overlap, those which are tied for the highest priority are scheduled one after the other, each for one rollin quantum. The priority of subpartitions is based on the priority of the subpartition itself and the priority of the jobs in it. Context switching is initiated by the *allocator* (the resource manager), using Unix signals. A job can have more than one process on each node, and if so, they all run using the Unix timesharing. Therefore the system actually supports family scheduling in our terminology.

- *Space slicing* — the partition is tiled and no overlap is allowed (except that jobs may overlap unactive subpartitions, that is subpartitions that do not have any active jobs in them). Jobs are then executed to completion.

If a job executes in a partition that uses time slicing, it is possible that a message will arrive at a node that is currently running an unrelated process. Such messages are nevertheless handled by the message passing processor, and received into the address space of the correct (non-executing) process [461].

Batch scheduling is done by NQS. Special provisions are taken to accomodate different node types, e.g. with different amounts of memory [606]. 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. 28.

Each NQS queue is assigned a base priority. The priority of queued jobs depends on the base priority and the queueing time. Whenever there are free nodes in the batch partition, NQS tries to pack the highest priority job onto the mesh. If a large job is queued for a long time, its priority may rise high enough so that nodes will be left idle (rather than running a

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 28: *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.*

smaller queued job) to ensure that it is eventually scheduled to run. The packing is based on a modified 2D buddy system algorithm, which allows buddies that are not powers of 2 [606]. Moreover, jobs can be packed across buddies of different sizes, and a best effort is made at using contiguous buddies. A subpartition is created from the allocated nodes, and it is dedicated to the job. The job is then executed to completion, subject to the time limit of the NQS queue.

The Meiko CS-2

The Meiko Computing Surface CS-2 is a distributed-memory multicomputer, based on SPARC nodes with optional Fujitsu vector units. Standard SUN (or third party) S-Bus peripherals are supported. The nodes are connected by a multistage network, which also supports broadcast and combining operations across ranges of contiguous nodes. Message passing is performed in user-space, with DMA from the network adapter.

A CS-2 can be partitioned by the system administrator into disjoint partitions. These partitions provide authorization domains, by being limited to certain user groups. Partitioning is completely flexible, in the sense that partitions can have any size and include any contiguous subset of the system's nodes. The partitions can be changed without re-booting nodes.

Each node of the CS-2 runs the Solaris operating system, which is a full Unix complete with virtual memory and local scheduling. However, the number and nature of the processes on each node depends on the type of the partition to which it belongs. Each partition runs

a parallel system program — the partition manager — which is responsible for starting user jobs in the partition, and killing them in case of error (e.g. a segmentation violation in one process will also cause the rest of the parallel job, executing on other nodes, to be terminated). The partition manager is actually composed of a collection of daemons, one on each node.

The system software supports four possible types of partitions. The login partition is used for user logins, program development, and so on. Parallel jobs are run on one of the other three types. In the batch partition, jobs receive a subset of the nodes for their exclusive use, and scheduling is manages by NQS. In the time sharing and gang scheduling partitions, jobs share the nodes with each other. The difference is that in a gang Scheduling partition the context switching is coordinated across the nodes, while in a time sharing partition it is not.

7.2.2 The Lawrence Livermore Gang Scheduler

Lawrence Livermore National Lab initiated a large application-driven study of parallel computing in 1989, using a 126-node BBN TC2000 Butterfly machine. In 1993, this machine was replaced by a 256-node Cray T3D. A local gang scheduler was developed in order to better support concurrent activity by users that wanted to use the machine in different modes, for program development, production runs, and benchmarking [241]. Due to the different characteristics of the two machines, the original version for the BBN was modified when ported to the Cray.

The BBN gang scheduler works by acquiring all the free processors from the system when it is booted, and then managing their use by multiple users. Submitted jobs interact with the gang scheduler (via sockets) just before calling the user main entry point. Jobs are classified into four classes [241]. 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.

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

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

The multi-context-switching operations are implemented by signals that are sent from the gang scheduler to all the processes in each job. One signal is used to instruct processes to go to sleep when their time quantum is up, and another to instruct them to wake up when they are to be rescheduled. The local nX scheduler therefore always has only one process to run[29] — the one chosen by the gang scheduler.

Fine-grain gang scheduling as used on the BBN was not efficient enough on the Cray T3D, so "gang-swapping" was used instead [203]. 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 6)[30]. 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.

[29] Programs using Zipcode [537] could have more processes than one process on each PE, and then all the processes belonging to the same job would be enabled. This leads to family scheduling rather than gang scheduling.

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

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 [18, 15]. This is used to tolerate the access latency to shared memory. The basic design is derived from the earlier HEP machine [540, 297, 321, 282]. The following descripion is based on Alverson at al. [16].

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).

7.2.4 The Connection Machine CM-5

The Connection Machine CM-5 from Thinking Machines Corp. is a distributed-memory multicomputer based on SPARC nodes with optional additional vector units [572, 442]. The nodes are connected by two networks: a data network with a 4-ary fat-tree topology, which is implemented as a multistage network (Fig. 29), and a control network, configured as a binary tree that matches the data network's fat tree [356]. The data network is used for

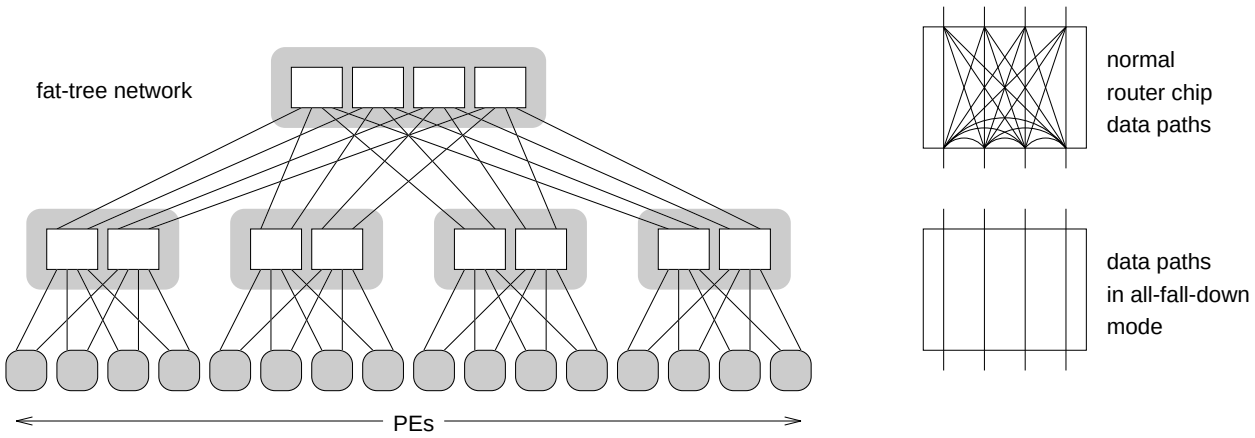


Figure 29: *The CM-5 fat-tree data network is implemented as a multistage network [356]. To flush it, router chips are put in "all-fall-down" mode.*

data transfer, while the control network can be used for special operations such as barrier synchronization, broadcast, and reduction. While no longer under production, the machine is interesting for its extensive support for gang scheduling.

Resource sharing on the CM-5 is done by two means: partitioning and gang scheduling. Partitions are set up by the system administrator, and can be limited to a certain class of jobs or a certain group of users. Each partition consists of one control processor and a set of processing nodes. The processing nodes must be contiguous, and must match a subtree in the tree structure of the networks, with a minimal size of 32 nodes. The relevant subtrees in the data and control networks are dedicated to the applications running in the partition. The partition boundaries are recorded in each node's network adapters when the partition is created. These boundaries are checked automatically when messages are sent, enabling user-mode access to the networks with no operating system overheads for protection. Partitions are completely independent of each other, and activity in one does not have any effect on the others. It is also possible to power down nodes in one partition without any consequences for programs running in other partitions.

Partitions can be enabled for general interactive access or for access through NQS batch queues. In interactive partitions, users log on to the control processor, and then run parallel applications on the partition's nodes. It is also possible to access other partitions by normal Unix commands, e.g. `rsh` to another partition's control processor. Parallel applications are represented by a single Unix process on the control processor. Thus time slicing on the control processor translates to gang scheduling on the processor nodes. The multi-context-switching needed to implement the gang scheduling is initiated by the partition's control processor, which sends a broadcast interrupt to all the nodes via the control network. This notifies all the processing nodes of the impending switch, and puts the dedicated network partition in "all-fall-down" mode. 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[31] (Fig. 29). All messages that are in transit are then routed to arbitrary nodes, and stored there as part of the application state [356]. When the application is re-scheduled, the stored messages are injected into the network again to complete their transit. This ensures that jobs do not interfere with each other even if they are sharing the same partition.

Batch-mode partitions only allow the execution of one job at a time, with no time slicing. Thus all the partition's resources — and especially all its memory — are dedicated to that job. Batch execution is managed by NQS, with separate queues for jobs with different resource requirements and possibly also for different user groups [323]. Queues can be associated with a specific partition, or else they can execute jobs on the most suitable partition based on the partition size and the job's requirements.

User feedback indicates that a minimal partition size of 32 nodes is too big. For example, small installations with only 32 nodes cannot partition the machine at all. Installations with 64 nodes can only create two equal partitions. This is significant considering that small installations are more common than large ones, and that a partition of 4 or 8 nodes is adequate for program development. The reaction to gang scheduling is mixed. On the one hand, users complain about lack of complete control over the machine, which prevents precise performance measurements. On the other hand, they take interactive access to the machine for granted, without realizing that this would not be the case if the machine did not provide gang scheduling.

The distributed job manager (DJM)

The distributed job manager (DJM) was developed at the Minnesota Supercomputing Center based on software from the University of California at Berkeley [404]. It has since been incorporated in the CM-5 system software from Thinking Machines. DJM's functionality is a superset of NQS, by supporting both interactive and batch jobs. Actually DJM does not make this distinction, except for the fact that jobs can be attached to a terminal or detached. When users submit jobs they must specify resource requirements, as they do with NQS. It is also possible to dictate that a job run on a dedicated partition, i.e. with no time slicing. Jobs can be submitted from the user's workstation, so it is no longer necessary to log onto a control processor. DJM then finds the best partition based on the number of jobs and on memory availability. If there is not enough memory, the job is queued.

If a job exceeds its limits, it is not killed immediately. Rather, the system maintains *scores* for both queued jobs and running jobs. The score for queued jobs increases with the time it spends in the queue, while the score for a running job decreases when it exceeds its limits. A running job is killed when a queued job has a higher score; the queued job is then run in its place.

Systems with DJM do not preclude users from logging on to the control processors and launching jobs directly. Such jobs are called "foreign" in DJM terminology. DJM monitors

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

all the activity on the system, so it knows about foreign jobs. These jobs are allowed to execute for a limited amount of time, so as to enable users to run short jobs without going through the DJM queueing mechanism. However, if foreign jobs execute for too long, they are killed by DJM.

7.2.5 The Kendall Square Research KSR1

The KSR1 provides a single address space model of computation [216]. Unlike traditional shared-memory architectures, where data has a unique memory address and possibly some cached replicas, the KSR1 uses an ALLCACHE design with caches but no memory to back them. Data is moved from one cache to another according to usage, so that it will always be close to where it is needed. The implementation is based on a hierarchy of caches with directories that keep track of the location of data and its sharing patterns. The KSR1 is scalable up to 1088 nodes (called "cells"), and uses custom-designed processors. Like the CM-5, this machine is no longer being produced.

The allocation of processing power to a job can be done in either of two ways. One is a direct request to allocate a number of cells. This can be combined with a directive to execute a certain program on those cells, and then the cells are released when the program terminates. If a program is not specified, the cells are allocated to the user's login session, and can be used to execute multiple programs one after the other.

The other approach is to use predefined processor sets, called "pset" [89]. Psets represent a partitioning of the machine as defined by the system administrator. In principle, a pset can include any set of nodes, but better performance is obtained if psets match the hierarchical cache structure. Users can then request that a job be executed on a specific pset. Three types of psets can be created. The first is an interactive pset. This is intended to provide interactive response times, and is used for program development, editing, and the like. The second is a dedicated pset, which is intended to provide the maximal speedup for a single job. The third is a pset with two jobs that both fit into the local caches. This is intended to provide good throughput.

The programmer view of parallelism is based on nested loops. The default is a static partitioning of the iteration space into separate threads, but self-scheduling is also possible. In any case, the number of threads on a pset can be larger than the number of cells. The scheduling is done at two levels. First, there is a pset queue, which is used for load balancing. This is accessed at a coarse granularity, e.g. every couple of seconds. The second level is a local queue on each cell, used for fine-grain switching among the threads mapped to it.

Feedback from user installations indicates that the pset approach is preferred over independent allocation for each job.

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 [329], and another that 60–70% are wasted even at peak usage hours [29]. A number of systems have been designed to utilize this untapped power in order to execute compute intensive parallel applications, including Condor [373, 473, 190, 472], which is the base technology for the IBM LoadLeveler, Piranha [231, 100], the distributed queueing system (DQS) from the University of Florida [169], the Utopia system [636], which was commercialized under the name Load Sharing Facility (LSF), the Stealth scheduler [329], Amber [193], and Hector [493].

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[32]. 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[33] [560, 229]. In technical terms, these requirements translate into monitoring facilities and some way to cope with the changing availability of workstations. The two main schemes that are used are process migration and using malleable jobs. Process migration has the advantage of being transparent to users, at the price of a more complex implementation. Using malleability is simpler, but the user is typically required to write code that will handle the de-alloction of reclaimed workstations [100, 473].

Naturally, the priority given to the workstations' owners implies interference with the performance of the parallel programs being run in the background [315, 359]. 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 [31, 184, 183]. In addition, it is advisable to limit the degree of parallelism of the parallel jobs to about half the cluster size [29].

A secondary issue that is usually ignored in the added load that the parallel jobs impose on the communication network itself. It has been suggested that communication requirements of applications be considered when scheduling them, and those that have large requirements be delayed if the network is already loaded [311].

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 [562, 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 [418, 563].

[32] These requirements can be relaxed to a certain degree by giving the owner higher priority for all the resources (CPU, memory pages, buffer cache frames), rather than exclusive use of the workstation. This leads to the salvation of more resources [329, 31].

[33] Recently, there has been work on adding load considerations to PVM too [102, 472, 440].

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. Those that were reviewed in this paper are summarized in Table 7.

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.

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.

system	*description*	*section*	*reference*
Alewife form MIT	local queues using hardware contexts and switching	4.1	[3, 4]
Alfalfa	load balancing by diffusion	4.1.2	[237]
Amoeba	PEs allocated singly	3.5	[418, 563]
AMPS	hierarchical structure for balanced mapping	4.1.1	[306]
APERM-2	hierarchical structure for balanced mapping	4.1.1	[269]
Chagori on K2 from Zurich	gang scheduling across whole machine	5.3	[550]
CHoPP from Columbia	variable partitioning by virtual hierarchy	3.3	[558]
Chrysalis (BBN Butterfly)	independent PEs with local queues within partitions	4.1	[352]
CM-2 from Thinking Machines	partitioning into quadrants, possible gang scheduling	3.2.1, 5.3	[582, 299]
CM-5 from Thinking Machines	gang scheduling within partitions created by buddy system	5.3, 7.2.4	[355]
Concentrix (Alliant FX/8)	gang scheduling across whole machine	5.3	[567]
Convex C2, C4	hardware self scheduling	3.5	[578, sect. 3.4]
Cosmic Cube	local queues	4.1	[513]
Cray microtasking	self scheduling from a global queue	4.2	[220]
Cray T3D	partitions of power-of-two PEs	3.2	[307]
Cray T3E	partitioninig, gang scheduling, and fair share	5	[346]
Crystal	partitioning by users	3.3	[151]
DASH	global queue with locking	4.2	[357]
DHC design from Jerusalem	gang scheduling with flexible grouping based on buddy system	5.4, 4.1.1	[197, 199]
DQT for RWC-1	gang scheduling with flexible grouping based on buddy system	5.4, 4.1.1	[273]
Dynix (Sequent Balance)	global queue with locking	4.2	[570]
EMBOS (Elxsi)	local queues	4.1	[498, 436]
EMMA2	local queues	4.1	[26]
Flagship	local queues with hardware mapping confined to partitions	4.1.1, 3.2.1	[561, 611]

Table 7: *Summary of the surveyed systems.*

system	*description*	*section*	*reference*
Fujitsu AP1000	single job at a time, local queues with message-driven tasks	4.1	[275]
Fujitsu VPP500	flexible partitioning	3.2.2	[407, 592]
Goodyear MPP	dedicated		[50, 470]
Heidelberg Polyp	distributed global queue with hardware scheduling and flexible partitioning	4.2, 4.1.1	[386]
Helios (transputers)	local queues with hardware scheduling within partitions	4.1	[451, 261]
HEP from Denelcor	local queues using hardware contexts and switching	4.1	[539, 297, 321]
Hydra on C.mmp at CMU	global queue with locking	4.2	[621, chap. 12]
IBM GF11	dedicated		[54, 342]
IBM SP1, SP2	variable partitioning	3.2.2, 7.2.1	[283]
Illiac IV	partitioning into quadrants	3.2.1	[45]
IRIX on SGI multiprocessors	global queue and gang scheduling	4.2, 5.2	
J-machine from MIT	hardware supported message queues act as local queues	4.1	[141, 140]
KSR1 system	flexible partitioning with interactive or dedicated use	7.2.5	[89]
LLNL gang scheduler on BBN/Cray	gang scheduling using matrix algorithm, for interactive, batch, and benchmark jobs	5.4, 7.2.2	[241, 203]
Mach from CMU	family scheduling: global queue with locking, within flexible partitions	4.2	[65, 64]
Mach on IBM RP3	family scheduling: global queue within flexible partitions; also local queues	4.3	[85]
MAXI on Makbilan from Jerusalem	global queue to distribute work, then local queues; experimental gang scheduling using matrix algorithm	4.3, 5.4	[194, 527, 198]
Medusa on CM* at CMU	gang scheduling using matrix algorithm	5.4	[438]
Meiko CS-2	flexible partitioning by administrator; batch, independent time slicing, or gang scheduling in partitions	7.2.1	
MICROS on MICRONET at SUNY	variable partitioning by virtual hierarchy	3.3	[595, 596]

Table 7: *continued.*

system	*description*	*section*	*reference*
MOOSE on hypercubes at Caltech	local queues	4.1	[497]
MOSIX from Jerusalem	load balancing by random pairing	4.1.2	[44, 42]
MuNet	load balancing by diffusion	4.1.2	[254]
nCUBE	partitioning with buddy system, local queues	3.2.1	[258]
NETRA	hierarchical structure for balanced mapping	4.1.1	[119]
NQS	batch queueing on partitioned systems	7.1.1, 3.2.3	[323]
NX/2 (iPSC/2)	partitioning into subcubes, local queues	3.2.1, 4.1	[460]
Orion (on M^3)	family scheduling with pools of processors	4.2	[90]
OSF/1 AD	local Unix-like scheduling plus load balancing	4.1, 4.1.2	[634]
Paragon from Intel	hierarchical partitioning with optional gang scheduling or space slicing	7.2.1	[288]
PASM from Purdue	partitioned multistage network	3.2.1	[584]
PEACE on SUPRENUM	local queues	4.1	[505, 506]
Plan 9 form AT&T	PEs allocated singly	3.5	[462]
Process Control from Stanford	dynamic partitioning which may change at runtime to reflect load	3.4	[581]
Psyche on BBN Butterfly from Rochester	local queues, support for two-level scheduling	4.1, 4.4.2	[510]
PUMA	local queues	4.1	[613]
Rediflow	load balancing by gradient method	4.1.2	[305]
Scheduler activations from Washington	support for two-level scheduling	4.4.2	[20]
SHARE for SP2	gang scheduling with flexible grouping based on buddy system	5.2, 5.4	[217]
SIMPLEX (nCUBE)	local queues	4.1	[335]
SpoC	PEs allocated singly	3.5	[420]
StarOS on Cm* at CMU	local queues	4.1	[228, chap. 6]
Symunix on NYU Ultracomputer	global queue based on fetch-and-add	4.2	[180, 181]
Sylvan	kernel coprocessor for local scheduling	4.1	[91]
Tera MTA	local queues using hardware contexts and switching	4.1, 7.2.3	[18, 96]

Table 7: *continued.*

system	*description*	*section*	*reference*
TRAC from Texas Austin	partitioned multistage network	3.2.1	[83, 372, chap. 7]
Trillium (FPS T-series transputers)	local queues with hardware scheduling	4.1	[92]
Two-level scheduler from Washington	dynamic partitioning which may change at runtime to reflect load	3.4	[633]
Victor from IBM Research	variable partitioning by users	3.3	[523]
VORX on HPC	variable partitioning by users	3.3	[303]
Warp systolic array	dedicated		[24]
Xylem on Cedar at Illinois	gang scheduling on PE clusters	5.3	[187, 188]

Table 7: *finished.*

- 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 [236, 164]. 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 [195, 201, 617, 276, 606, 200]. 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 [16]. 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 [206].

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 "lightweight 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.

B The Thread Model

The term "thread model" encompasses the characteristics of threads as perceived by the programmer. This includes the interfaces used to manipulate threads, and thus to express parallelism. It also includes the performance implications of using threads in different ways. The scheduling mechanism used in a system is often related to the thread model, as it can have a great impact on performance. This appendix examines the various aspects of the thread model and their relationship to scheduling. The relevant aspects of the thread model are the abstraction provided by threads, including dynamic thread creation, thread

weight, the distinction between user threads and system threads, and synchronization among threads.

Thread abstraction

Different parallel programming systems present very different abstractions to the programmer. One of the major distinctions separates those that use control parallelism from those that use data parallelism. With control parallelism, the programmer specifies the exact sequence of instructions performed by each thread, sometimes including constructs that spawn new threads in the process [444]. With data parallelism there is a single logical thread of control, and the parallelism is implied by the fact that the data structures are distributed across multiple PEs [268]. It is then the compiler's job to create the actual sequences of instructions (threads) that will execute on each PE.

A third approach takes the view that the abstract computation is the driving force, not the control structure or the data distribution. For example, the computation can be represented by a directed graph, where nodes are basic computations (usually called "tasks" in this context) and arcs denote dependencies. For example, such a graph may be built when unfolding the interdependent iterations of a doacross loop. Alternatively, the computation can be viewed as an uncoordinated workpile of chores to be done, e.g. the independent iterations of a doall loop, or evaluations injected into tuple space. The actual execution can be carried out in either of two ways. One is by a set of "worker" threads, one on each PE [597, 17, 178]. Creating the worker threads and mapping the tasks or chores to them is then the responsibility of the runtime system. The other is to define a set of virtual processors at compile time, and map the units of work onto them statically [434]. At runtime, each of these virtual processors becomes a thread that executes on a PE.

Dynamic thread creation

It is simple to deal with programs written in the SPMD style, with all threads created at the outset and no changes thereafter. But many applications are structured as a sequence of phases with different degrees of parallelism. This translates into the use of a changing number of threads. It could be taken to imply that threads be created and destroyed on the fly. On the other hand, if thread creation is expensive, an implementation can choose to retain threads from one parallel phase to the next, even if they are not being used [444].

An important aspect of dynamic thread creation is that it is a symptom of a deeper change: a change in resource requirements. It is well known that load balancing improves performance when faced with a dynamically changing workload [483, 175]. This implies that, if PEs are allocated to jobs, the allocation should change in concert with changes in the number of threads. This is exactly what dynamic partitioning does. On the other hand it is possible to avoid the re-partitioning problem by allocating processing power to individual threads in the first place, rather than to jobs. This is done in systems that schedule threads independently.

Dynamic thread creation is more problematic when some guarantees are made about their execution, e.g. in gang scheduling systems that schedule all the threads simultaneously. In this context, it might be impossible to add threads because there is not enough room for them, and deleting threads might leave PEs idle. Therefore it is typically not possible to change the size of a gang after it was created. However, it is possible to create new gangs, that will not necessarily execute simultaneously with the previous ones.

Thread weight

The issue of thread weight is perhaps the most widely debated. The options span conventional heavy-weight Unix processes, implementing light-weight threads in the operating system kernel, or implementing them in a user-space thread package.

Parallel versions of Unix have been quite popular, due to the popularity and availability of Unix itself (and especially the availability of source code). In these systems the unit of parallel execution is typically the Unix process, complete with a separate address space, signal-handling facilities, open files, etc. Early examples include multiprocessor Unix systems [37], Symunix on the NYU Ultracomputer [182], and Dynix on the Sequent Balance [570]. With the advent of parallel machines based on workstation technology in recent years, this approach has received a new push. For example, the Meiko CS-2, Intel Paragon, and IBM SP2 all run a full Unix system on each node.

The problem with Unix processes is that the overhead in creating and destroying them is significant. Therefore they cannot be used as the underlying technology for dynamic thread creation. In the Mach system, the process abstraction was therefore divided into two parts: tasks[34], which represent resource allocation, such as an address space and communication capabilities, and threads, which represent execution within the context of a task [482, 568, 494]. Each thread belongs to a single task, but a task can have multiple threads that all share its resources; the Unix process is equivalent to a task with a single thread. As threads include less state, their manipulation is less expensive than that of a traditional Unix process. The forking of new tasks is also optimized, by using a copy-on-write mechanism to duplicate the address space.

Threads are not unique to Mach. The same abstractions are available in the OSF/1 operating system, which is based on Mach [437, 634]. A similar notion of threads exists in Topaz [569], in PEACE [505, 506], in Panda [61], and in the Portable Common Runtime interface [612]. Variable-weight threads, where some of the resources are allocated per-thread rather then being shared, have also been proposed [27]. This is used in the plan 9 system from AT&T Bell Labs. Finally, threads have been included in the IEEE POSIX standard and implemented in the DCE framework from OSF.

Despite the relatively light weight of threads, their manipulation (including scheduling) still requires a trap to the operating system kernel. This in itself is a costly operation, because the kernel has to protect itself from potentially malicious programs, even if it is just

[34]Note that "task" has a specific technical meaning in the context of Mach, which is totally different from other uses, such as to describe the nodes of a task graph.

switching two threads in the same address space [20]. In addition, even switching light-weight threads requires the CPU's register contents to be stored and restored. This too is costly on modern RISC processors with large register files [22]. Therefore it has been suggested that all thread manipulation should be done in user space, using two-level scheduling [20]. An alternative method for reducing the cost of thread scheduling is to use continuations where possible [166, 152]. With continuations, a blocking thread specifies the context in which it should be resumed, rather than requiring the current context to be stored.

User threads vs. kernel threads

From a programmer's point of view, threads refer to an abstraction provided by the programming environment, not to any specific entity at the operating system level. The mapping between the two depends on the language runtime system. With one-to-one mapping, the user is essentially using the threads supplied by the operating system directly (kernel threads). This has the advantage that all the kernel's thread-manipulation facilities are available, including suspending, resuming, priority control, and use of synchronization mechanisms. It has the disadvantage that all these services require a trap into supervisor mode, and are therefore relatively costly.

Alternatively, the language runtime system can multiplex multiple language-level threads onto a smaller number of processes or kernel threads. This is called user threads, because all thread manipulation occurs in user space. It is the basis for two-level scheduling: the underlying kernel threads are either each mapped onto a separate PE, or they are scheduled preemptively by the operating system. In turn, these threads — sometimes referred to as "virtual PEs" — execute the user threads. Examples of thread packages based on this approach include Cthreads [130], Presto [59, 192], FastThreads [21, 20], WorkCrews [597], Chant [252], Distributed Filaments [218], the Portable Common Runtime [612], the DIA thread library [105], some implementations of Concurrent C [124, 224], and the XERO system [286]. It should be noted that this list is by no means complete; thread packages are not considered to be great innovations, so they tend not to be published in the literature.

User threads are more flexible than kernel threads, because their functionality depends on code at the user level. In fact, multiple flavors of threads may be supported concurrently on the same system [509]. The overhead of various operations, including scheduling, is much lower than for kernel threads. The main drawback of user threads is that they are unknown to the kernel. For example, a user thread might perform an I/O operation. As far as the operating system is concerned, it is the underlying kernel thread that performed the I/O, so that thread is blocked. If only that kernel thread was executing on its PE, that PE remains idle until the I/O operation completes. This reduces the parallelism in the program inappropriately, because there might be other user threads that are ready to run. It would be better if only the user thread were blocked, and the kernel thread was left free to execute other user threads.

Some alternatives to the conventional thread model have been proposed recently to cope with this problem. These include scheduler activations [20], first-class threads [390], and

cooperative threads [286]. The unifying theme of these proposals is the use of callbacks, signals, or software interrupts to notify the user-level scheduler of a change in its state, e.g. that the current thread has been blocked. This information is then used to reschedule in user space. Additional details are given in Section 4.4.2.

Finally, the notions of virtualization and optimal simulation are related to the use of threads [593, 393, 208]. The idea is that if a job's degree of parallelism is much higher than the physical parallelism in the system, so that each PE executes multiple threads, it can switch among these threads and thus hide various latencies and overheads. This is a special case of hardware support for user-level threads. It was used in the design of the HEP and Tera systems [540, 297, 321, 18, 15].

Synchronization among threads

Synchronization among threads can have an explicit effect on scheduling, e.g. when a thread blocks to wait for another thread to release a lock. But there might also be an implicit requirement for simultaneous execution, e.g. through gang scheduling. For example, users may expect that threads that are spawned together actually execute in parallel, and therefore busy waiting can be used to synchronize them. If gang scheduling is not used, this can have grave performance implications [198, 631]. Moreover, if the scheduling is not preemptive at all, it could lead to deadlock [84].

A related issue is the granularity of the interactions. In general, granularity refers to amount of computation performed between successive interactions [336]. Thus in fine-grain interactions, interactions (such as synchronization or transfer of data) are numerous and frequent, whereas coarse-grain interactions are faraway. Monitoring a variety of numerical applications has shown that granularities can be as low as a hew hundred or a thousand instructions [277, 136].

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] S. V. Adve and M. D. Hill, "*A unified formalization of four shared-memory models*". *IEEE Trans. Parallel & Distributed Syst.* **4(6)**, pp. 613–624, Jun 1993.

[3] 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.

[4] 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.

[5] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, "*SP2 system architecture*". *IBM Syst. J.* **34(2)**, pp. 152–184, 1995.

[6] 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.

[7] 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.

[8] 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.

[9] S. Ahuja, N. Carriero, and D. Gelernter, "*Linda and friends*". *Computer* **19(8)**, pp. 26–34, Aug 1986.

[10] S. Al-Bassam, H. El-Rewini, B. Bosa, and T. G. Lewis, "*Processor allocation for hypercubes*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 394–401, Dec 1992.

[11] E. Albert, K. Knobe, J. D. Lukas, and G. L. Steele, Jr., "*Compiling Fortran 8x array features for the Connection Machine computer system*". In *Proc. ACM/SIGPLAN PPEALS (Parallel Programming: Experience with Applications, Languages & Systems)*, pp. 42–56, Jul 1988.

[12] A. Al-Dhelaan and B. Bose, "*A new strategy for processors allocation in an n-cube multiprocessor*". In 8th *Intl. Phoenix Conf. Computers & Communications*, pp. 114–118, Mar 1989.

[13] 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.

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

[15] 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.

[16] 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.

[17] G. A. Alverson and D. Notkin, "*Program structuring for effective parallel portability*". *IEEE Trans. Parallel & Distributed Syst.* **4(9)**, pp. 1041–1059, Sep 1993.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska, "*The interaction of architecture and operating system design*". In 4th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 108–120, Apr 1991.

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

[24] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb, "*The Warp computer: architecture, implementation, and performance*". *IEEE Trans. Comput.* **C-36(12)**, pp. 1523–1538, Dec 1987.

[25] 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.

[26] E. Appiani, D. Bianco, L. Merlo, and L. Roncarolo, "*The EMMA2 multiprocessor operating system*". *Concurrency — Pract. & Exp.* **3(6)**, pp. 541–557, Dec 1991.

[27] Z. Aral, J. Bloom, T. Doeppner, I. Gertner, A. Langerman, and G. Schaffer, "*Variable weight processes with flexible shared resources*". In *Proc. Winter USENIX Technical Conf.*, pp. 405–412, 1989.

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

[29] 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.

[30] 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.

[31] 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.

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

[33] Y. Aumann and M. O. Rabin, "*Clock construction in fully asynchronous parallel systems and PRAM simulation*". *Theoretical Comput. Sci.* **128(1-2)**, pp. 3–30, Jun 1994.

[34] 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.

[35] 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.

[36] M. J. Bach, *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[37] 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.

[38] 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.

[39] V. Bala et al., "*The IBM external user interface for scalable parallel systems*". *Parallel Comput.* **20(4)**, pp. 445–462, Apr 1994.

[40] 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.

[41] P. Banerjee, M. H. Jones, and J. S. Sargent, "*Parallel simulated annealing algorithms for cell placement on hypercube multiprocessors*". *IEEE Trans. Parallel & Distributed Syst.* **1(1)**, pp. 91–106, Jan 1990.

[42] 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.

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

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

[45] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "*The Illiac IV computer*". *IEEE Trans. Comput.* **C-17(8)**, pp. 746–757, Aug 1968.

[46] E. Barszcz, "*Intercube communication for the iPSC/860*". In *Scalable High-Performance Comput. Conf.*, pp. 307–313, 1992.

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

[48] 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.

[49] 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.

[50] K. E. Batcher, "*Design of a massively parallel processor*". *IEEE Trans. Comput.* **C-29(9)**, pp. 836–840, Sep 1980.

[51] W. F. Bauer and R. H. Hill, "*Cost-effectiveness of time-shared computer systems*". In *The Transition to On-Line Computing: Problems and Solutions*, F. Gruenberger (ed.), pp. 79–106, Academic Press, 1967.

[52] B. Beck, "*AAMP: a multiprocessor approach for operating systems and application migration*". *Operating Syst. Rev.* **24(2)**, pp. 41–55, Apr 1990.

[53] M. J. Beckerle, "*Overview of the START(*T) multithreaded computer*". In 38th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 148–156, Feb 1993.

[54] J. Beetem, M. Denneau, and D. Weigarten, "*The GF11 supercomputer*". In 12th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 108–115, Jun 1985.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] B. N. Bershad, E. D. Lazowska, and H. M. Levy, "*PRESTO: a system for object-oriented parallel programming*". *Software — Pract. & Exp.* **18(8)**, pp. 713–732, Aug 1988.

[60] 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.

[61] R. Bhoedjand, T. Rühl, R. Hofman, K. Langendoen, and H. Bal, "*Panda: a portable platform to support parallel programming languages*". In 4th *Symp. Experiences with Distributed & Multiprocessor Syst.*, pp. 213–226, USENIX, Sep 1993.

[62] 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.

[63] R. Bisiani and M. Ravishankar, "*PLUS: a distributed shared-memory system*". In 17th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 115–124, May 1990.

[64] D. L. Black, "*Processors, priority, and policy: Mach scheduling for new environments*". In *Proc. Winter USENIX Technical Conf.*, pp. 1–12, Jan 1991.

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

[66] 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.

[67] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron, "*Translation lookaside buffer consistency: a software approach*". In 3rd *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 113–122, Apr 1989.

[68] D. L. Black, A. Tevanian, Jr., D. B. Golub, and M. W. Young, "*Locking and reference counting in the Mach kernel*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 167–173, Aug 1991.

[69] J. Błażewicz, M. Drabowski, and J. Węglarz, "*Scheduling multiprocessor tasks to minimize schedule length*". *IEEE Trans. Comput.* **C-35(5)**, pp. 389–393, May 1986.

[70] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "*Cilk: an efficient multithreaded runtime system*". *J. Parallel & Distributed Comput.* **37(1)**, pp. 55–69, Aug 1996.

[71] R. D. Blumofe and D. S. Park, "*Scheduling large-scale parallel computations on networks of workstations*". In 3rd *Intl. Symp. High Performance Distributed Comput.*, pp. 96–105, Apr 1994.

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

[73] S. H. Bokhari, "*On the mapping problem*". *IEEE Trans. Comput.* **C-30(3)**, pp. 207–214, Mar 1981.

[74] 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.

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

[76] S. Borkar et al., "*Supporting systolic and memory communication in iWarp*". In 17th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 70–81, May 1990.

[77] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "*The Illiac IV system*". *Proc. IEEE* **60(4)**, pp. 369–388, Apr 1972.

[78] K. C. Bowler, R. D. Kenway, and D. J. Wallace, "*The Edinburgh Concurrent Supercomputer: project and applications*". In *CONPAR 88*, C. R. Jesshope and K. D. Reinartz (eds.), pp. 635–642, Cambridge University Press, 1989.

[79] T. Brecht, "*On the importance of parallel application placement in NUMA multiprocessors*". In 4th *Symp. Experiences with Distributed & Multiprocessor Syst.*, pp. 1–18, USENIX, Sep 1993.

[80] T. B. Brecht, "*An experimental evaluation of processor pool-based scheduling for shared-memory NUMA multiprocessors*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 139–165, Springer Verlag, 1997. Lecture Notes in Computer Science Vol. 1291.

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

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

[83] J. C. Browne, "*TRAC: an environment for parallel computing*". In *COMPCON Spring '84*, pp. 294–298, 1984.

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

[85] 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.

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

[87] P. A. Buhr and R. A. Stroobosscher, "*The μsystem: providing light-weight concurrency on shared-memory multiprocessor computers running UNIX*". *Software — Pract. & Exp.* **20(9)**, pp. 929–964, Sep 1990.

[88] 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.

[89] E. Burke, "*An overview of system software for the KSR1*". In 38th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 295–299, Feb 1993.

[90] H. Burkhart, R. eigenmann, H. Kindlimann, M. Moser, and H. Scholian, "*The M^3 multi-processor laboratory*". *IEEE Trans. Parallel & Distributed Syst.* **4(5)**, pp. 507–519, May 1993.

[91] F. J. Burkowski, "*A vector and array multiprocessor extension of the Sylvan architecture*". In 11th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 4–11, Jun 1984.

[92] 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.

[93] F. Butler, H. Chen, J. Sexton, A. Vaccarino, and D. Weingarten, "*Hadron mass predictions of the valence approximation to lattice QCD*". *Phys. Rev. Lett.* **70(19)**, pp. 2849–2852, 10 May 1993.

[94] R. M. Butler and E. L. Lusk, "*Monitors, messages, and clusters: the p4 parallel programming system*". *Parallel Comput.* **20(4)**, pp. 547–564, Apr 1994.

[95] 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.

[96] D. Callahan and B. Smith, "*A future-based parallel language for a general-purpose highly-parallel computer*". In *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua (eds.), pp. 95–113, MIT Press, 1990.

[97] R. H. Campbell, N. Islam, and P. Madany, "*Choices, frameworks and refinement*". *Computing Systems* **5(5)**, pp. 217–257, Summer 1992.

[98] B. M. Carlson and L. W. Dowdy, "*Static processor allocation in a soft real-time multiprocessor environment*". *IEEE Trans. Parallel & Distributed Syst.* **5(3)**, pp. 316–320, Mar 1994.

[99] N. Carriero and D. Gelernter, "*How to write parallel programs: a guide to the perplexed*". *ACM Comput. Surv.* **21(3)**, pp. 323–357, Sep 1989.

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

[101] 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.

[102] 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.

[103] 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.

[104] 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.

[105] G. Cattaneo, G. Di Giore, and M. Ruotolo, "*Another C threads library*". *SIGPLAN Notices* **27(12)**, pp. 81–90, Dec 1992.

[106] 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.

[107] R. Chandra, A. Gupta, and J. L. Hennessy, "*COOL: an object-based language for parallel programming*". *Computer* **27(8)**, pp. 13–26, Aug 1994.

[108] 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.

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

[110] G-I. Chen and T-H. Lai, "*Constructing parallel paths between two subcubes*". *IEEE Trans. Comput.* **41(1)**, pp. 118–123, Jan 1992.

[111] G-I. Chen and T-H. Lai, "*Preemptive scheduling of independent jobs on a hypercube*". *Inf. Process. Lett.* **28(4)**, pp. 201–206, Jul 1988.

[112] 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.

[113] 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.

[114] 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.

[115] M-S. Chen and K. G. Shin, "*Task migration in hypercube multiprocessors*". In 16th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 105–111, May 1989.

[116] S-H. Chiang, R. K. Mansharamani, and M. K. Vernon, "*Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 33–44, May 1994.

[117] 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.

[118] R. S. Chin and S. T. Chanson, "*Distributed object-based programming systems*". *ACM Comput. Surv.* **23(1)**, pp. 91–124, Mar 1991.

[119] 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.

[120] 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.

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

[122] P-J. Chuang and N-F. Tzeng, "*Allocating precise submeshes in mesh connected systems*". *IEEE Trans. Parallel & Distributed Syst.* **5(2)**, pp. 211–217, Feb 1994.

[123] P-J. Chuang and N-F. Tzeng, "*A fast recognition-complete processor allocation strategy for hypercube computers*". *IEEE Trans. Comput.* **41(4)**, pp. 467–479, Apr 1992.

[124] R. F. Cmelik, N. H. Gehani, and W. D. Roome, "*Experience with multiple processor versions of Concurrent C*". *IEEE Trans. Softw. Eng.* **15(3)**, pp. 335–344, Mar 1989.

[125] 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.

[126] 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.

[127] 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.

[128] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[129] Computer System Architects. *Computer* **25(12)**, p. back cover, Dec 1992. Ad for SuperSet-Plus transputer array.

[130] E. C. Cooper and R. P. Draves, *C Threads*. Technical Report CMU-CS-88-154, Dept. Computer Science, Carnegie-Mellon University, Jun 1988.

[131] P. F. Corbett, D. G. Feitelson, J-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. D. Herr, J. Kavaky, T. R. Morgan, and A. Zlotek, "*Parallel file systems for the IBM SP computers*". *IBM Syst. J.* **34(2)**, pp. 222–248, 1995.

[132] 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.

[133] 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.

[134] 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.

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

[136] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina, "*A quantitative study of parallel scientific applications with explicit communication*". *J. Supercomput.* **10(1)**, pp. 5–24, 1996.

[137] R. Cytron, "*Doacross: beyond vectorization for multiprocessors*". In *Intl. Conf. Parallel Processing*, pp. 836–844, Aug 1986.

[138] W. Dally, "*Network and processor architecture for message-driven computers*". In *VLSI and Parallel Computation*, R. Suaya and G. Britwistle (eds.), chap. 3, Morgan Kaufmann Publishers, Inc., 1990.

[139] W. J. Dally, "*Performance analysis of k-ary n-cube interconnection networks*". *IEEE Trans. Comput.* **39(6)**, pp. 775–785, Jun 1990.

[140] 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.

[141] 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.

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

[143] 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.

[144] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "*A single-program-multiple-data computational mode for EPEX/FORTRAN*". *Parallel Comput.* **7(1)**, pp. 11–24, Apr 1988.

[145] 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.

[146] 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.

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

[148] 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.

[149] 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.

[150] 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.

[151] D. J. De Witt, R. Finkel, and M. Solomon, "*The Crystal multicomputer: design and implementation experience*". *IEEE Trans. Softw. Eng.* **SE-13(8)**, pp. 953–966, Aug 1987.

[152] R. W. Dean, "*Using continuations to build a user-level threads library*". In *USENIX Mach III Symp.*, pp. 137–151, Apr 1993.

[153] J. M. del Rosario and A. N. Choudhary, "*High-performance I/O for massively parallel computers: problems and prospects*". *Computer* **27(3)**, pp. 59–68, Mar 1994.

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

[155] P. J. Denning, "*Working sets past and present*". *IEEE Trans. Softw. Eng.* **SE-6(1)**, pp. 64–84, Jan 1980.

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

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

[158] 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.

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

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

[161] 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.

[162] 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.

[163] 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.

[164] 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.

[165] 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.

[166] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean, "*Using continuations to implement thread management and communication in operating systems*". In 13th *Symp. Operating Systems Principles*, pp. 122–136, Oct 1991.

[167] J. Du and J. Y-H. Leung, "*Complexity of scheduling parallel task systems*". *SIAM J. Discrete Math.* **2(4)**, pp. 473–487, Nov 1989.

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

[169] 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`.

[170] D. Durand, T. Montaut, L. Kervella, and W. Jalby, "*Impact of memory contention on dynamic scheduling on NUMA multiprocessors*". *IEEE Trans. Parallel & Distributed Syst.* **7(11)**, pp. 1201–1214, Nov 1996.

[171] 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.

[172] 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.

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

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

[175] 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.

[176] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "*A comparison of receiver-initiated and sender-initiated adaptive load sharing*". *Perform. Eval.* **6(1)**, pp. 53–68, 1986.

[177] 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.

[178] D. L. Eager and J. Zahorjan, "*Chores: enhanced run-time support for shared-memory computing*". *ACM Trans. Comput. Syst.* **11(1)**, pp. 1–32, Feb 1993.

[179] 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.

[180] 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.

[181] 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.

[182] 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.

[183] 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.

[184] 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.

[185] 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.

[186] G. W. Elsesser, V. N. Ngo, S. Bhattacharya, and W-T. Tsai, "*Processor preallocation and load balancing of DOALL loops*". *J. Supercomputing* **8(2)**, pp. 135–161, Jun 1994.

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

[188] 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.

[189] D. H. J. Epema, "*An analysis of decay-usage scheduling in multiprocessors*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 74–85, May 1995.

[190] 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.

[191] R. B. Essik, "*An event-based fair scheduler*". In *Winter USENIX Conf.*, pp. 147–161, Jan 1990.

[192] J. E. Faust and H. M. Levy, "*The performance of an object-oriented threads package*". In *Object-Oriented Prog. Syst., Lang., & Appl. Conf. Proc.*, pp. 278–288, Oct 1990.

[193] 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.

[194] 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.

[195] 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.

[196] 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.

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

[198] 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.

[199] D. G. Feitelson and L. Rudolph, "*Mapping and scheduling in a shared parallel environment using distributed hierarchical control*". In *Intl. Conf. Parallel Processing*, vol. I, pp. 1–8, Aug 1990.

[200] 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.

[201] 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.

[202] D. G. Feitelson, P. F. Corbett, S. J. Baylor, and Y. Hsu, "*Parallel I/O subsystems in massively parallel supercomputers*". *IEEE Parallel & Distributed Technology* **3(3)**, pp. 33–47, Fall 1995.

[203] 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.

[204] 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.

[205] 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.

[206] 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.

[207] D. G. Feitelson and M. A. Volovic, "*ParPar design document version 0.2*". URL http://www.huji.ac.il/labs/parallel/parpar.html, Jun 1997.

[208] A. Feldmann, M-Y. Kao, J. Sgall, and S-H. Teng, "*Optimal online scheduling of parallel jobs with dependencies*". In 25th *Ann. Symp. Theory of Computing*, pp. 642–651, May 1993.

[209] W. Fenton, B. Ramkumar, V. A. Saletore, A. B. Sinha, and L. V. Kalé, "*Supporting machine independent programming on diverse parallel architectures*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 193–201, Aug 1991.

[210] 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.

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

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

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

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

[215] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors. Vol. I: General Techniques and Regular Problems*. Prentice-Hall, 1988.

[216] S. Frank, H. Burkhardt, III, and J. Rothnie, "*The KSR1: bridging the gap between shared memory and MPPs*". In 38th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 285–294, Feb 1993.

[217] 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.

[218] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews, "*Distributed Filaments: efficient fine-grain parallelism on a cluster of workstations*". In 1st *Symp. Operating Systems Design & Implementation*, pp. 201–213, USENIX, Nov 1994.

[219] 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.

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

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

[222] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, *Cedar*. Technical Report UIUCDCS-R-83-1123, Dept. Computer Science, University of Illinois, Urbana, IL., Feb 1983. Reprinted in K. Hwang, *Supercomputers: Design and Applications*, IEEE Computer Society, pp. 251–275, 1984.

[223] L. Gao, A. L. Rosenberg, and R. K. Sitaraman, "*On trading task reallocation for thread management in partitionable multiprocessors*". In 8th *Symp. Parallel Algorithms & Architectures*, pp. 309–317, Jun 1996.

[224] N. H. Gehani and W. D. Roome, "*Implementing Concurrent C*". *Software — Pract. & Exp.* **22(3)**, pp. 265–285, Mar 1992.

[225] J. Gehring and F. Ramme, "*Architecture-independent request-scheduling with tight waiting-time estimations*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 65–88, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[226] 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.

[227] E. F. Gehringer, A. K. Jones, and Z. Z. Segall, "*The Cm* testbed*". *Computer* **15(10)**, pp. 40–53, Oct 1982.

[228] E. F. Gehringer, D. P. Siewiorek, and Z. Segall, *Parallel Processing: The Cm* Experience*. Digital Press, 1987.

[229] 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.

[230] D. Gelernter, "*Generative communication in Linda*". *ACM Trans. Prog. Lang. & Syst.* **7(1)**, pp. 80–112, Jan 1985.

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

[232] 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.

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

[234] K. Gharachorloo, A. Gupta, and J. Hennessy, "*Hiding memory latency using dynamic scheduling in shared-memory multiprocessors*". In 19th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 22–33, May 1992.

[235] 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.

[236] 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.

[237] 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.

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

[239] 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.

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

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

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

[243] A. Gottlieb, "*An overview of the NYU Ultracomputer project*". In *Experimental Parallel Computing Architectures*, J. J. Dongarra (ed.), pp. 25–95, North-Holland, 1987.

[244] 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.

[245] 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.

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

[247] 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.

[248] 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.

[249] 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.

[250] 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.

[251] J. L. Guynes, "*Impact of system response time on state anxiety*". *Comm. ACM* **31(3)**, pp. 342–347, Mar 1988.

[252] M. Haines, D. Cronk, and P. Mehrotra, "*On the design of Chant: a talking threads package*". In *Supercomputing '94*, pp. 350–359, Nov 1994.

[253] R. H. Halstead, Jr., "*Multilisp: a language for concurrent symbolic computation*". *ACM Trans. Prog. Lang. & Syst.* **7(4)**, pp. 501–538, Oct 1985.

[254] 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.

[255] 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.

[256] 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.

[257] 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.

[258] 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.

[259] J. L. Hellerstein, "*Achieving service rate objectives with decay usage scheduling*". *IEEE Trans. Softw. Eng.* **19(8)**, pp. 813–825, Aug 1993.

[260] D. P. Helmbold and C. E. McDowell, "*Modeling speedup (n) greater than n*". *IEEE Trans. Parallel & Distributed Syst.* **1(2)**, pp. 250–256, Apr 1990.

[261] 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.

[262] 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.

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

[264] 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.

[265] M. Heuser, "*An implementation of real-time thread synchronization*". In *Proc. Summer USENIX Technical Conf.*, pp. 97–105, Jun 1990.

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

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

[268] W. D. Hillis and G. L. Steele, Jr., "*Data parallel algorithms*". *Comm. ACM* **29(12)**, pp. 1170–1183, Dec 1986.

[269] 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.

[270] 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.

[271] 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.

[272] 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.

[273] 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.

[274] 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.

[275] T. Horie and M. Ikesaka, "*AP1000 software environment for parallel programming*". *Fujitsu Scientific & Technical J.* **29(1)**, pp. 25–31, spring 1993.

[276] 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.

[277] J-M. Hsu and P. Banerjee, "*Performance measurement and trace driven simulation of parallel CAD and numeric applications on a hypercube multicomputer*". *IEEE Trans. Parallel & Distributed Syst.* **3(4)**, pp. 451–464, Jul 1992.

[278] J-H. Huang and L. Kleinrock, "*Performance evaluation of dynamic sharing of processors in two-stage parallel processing systems*". *IEEE Trans. Parallel & Distributed Syst.* **4(3)**, pp. 306–317, Mar 1993.

[279] 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.

[280] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein, "*Load-sharing in heterogeneous systems via weighted factoring*". In 8th *Symp. Parallel Algorithms & Architectures*, pp. 318–328, Jun 1996.

[281] 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.

[282] K. Hwang, "*Multiprocessor supercomputers for scientific/engineering applications*". *Computer* **18(6)**, pp. 57–73, Jun 1985.

[283] IBM Corp., *IBM AIX Parallel Environment: Operation and Use*. Order number SH26-7230-0, Sep 1993.

[284] IBM Corp., *SP2 Administration Guide Release 2*. Order number SH26-2486-01, Dec 1994.

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

[286] S. Inohara, K. Kato, A. Narita, and T. Masuda, "*A thread facility based on user/kernel cooperation in the XERO operating system*". In 15th *IEEE Comput. Soc. Intl. Comput. Software & Appl. Conf. (COMPSAC)*, pp. 398–405, Sep 1991.

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

[288] Intel Supercomputer Systems Division, *Paragon User's Guide*. Order number 312489-003, Jun 1994.

[289] N. Islam, A. Prodromidis, and M. S. Squillante, "*Dynamic partitioning in different distributed-memory environments*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 244–270, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[290] L. H. Jamieson, "*Characterizing parallel algorithms*". In *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass (eds.), pp. 65–100, MIT Press, 1987.

[291] D. Jefferson et al., "*Distributed simulation and the Time Warp operating system*". In 11th *Symp. Operating Systems Principles*, pp. 77–93, Nov 1987.

[292] 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.

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

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

[295] 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.

[296] T. Jones, "*Engineering design of the Convex C2*". *Computer* **22(1)**, pp. 36–44, Jan 1989.

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

[298] 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.

[299] 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.

[300] 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.

[301] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki, "*Empirical studies of competitive spinning for a shared-memory multiprocessor*". In 13th *Symp. Operating Systems Principles*, pp. 41–55, Oct 1991.

[302] 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.

[303] H. P. Katseff, R. D. Gaglianello, and B. S. Robinson, "*The evolution of HPC/VORX*". In 2nd *Symp. Principles & Practice of Parallel Programming*, pp. 60–69, Mar 1990.

[304] J. Kay and P. Lauder, "*A fair share scheduler*". *Comm. ACM* **31(1)**, pp. 44–55, Jan 1988.

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

[306] 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.

[307] R. E. Kessler and J. L. Schwarzmeier, "*Cray T3D: a new dimension for Cray Research*". In 38th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 176–182, Feb 1993.

[308] G. Kim and H. Yoon, "*Free submesh list strategy: a best fit submesh allocation in mesh connected multicomputers*". *Parallel Process. Lett.* **6(1)**, pp. 75–86, Mar 1996.

[309] 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.

[310] J-U. Kim, K-H. Shim, and K. H. Park, "*A link-disjoint subcube for processor allocation in hypercube computers*". *Parallel Computing* **22(12)**, pp. 1579–1595, Feb 1997.

[311] O. Kipersztok and J. C. Patterson, "*Intelligent fuzzy control to augment the scheduling capabilities of network queueing systems*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 239–258, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[312] 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.

[313] 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.

[314] 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.

[315] 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.

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

[317] D. E. Knuth, *The Art of Computer Programming. Vol 1: Fundamental Algorithms*. Addison Wesley, 2 ed., 1973.

[318] C. Koelbel, P. Mehrotra, and J. Van Rosendale, "*Supporting shared data structures on distributed memory architectures*". In 2nd *Symp. Principles & Practice of Parallel Programming*, pp. 177–186, Mar 1990.

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

[320] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott, "*Scheduler-conscious synchronization*". *ACM Trans. Comput. Syst.* **15(1)**, pp. 3–40, Feb 1997.

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

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

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

[324] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr, "*Mul-T: a high-performance parallel Lisp*". In *Proc. SIGPLAN Conf. Prog. Lang. Design & Implementation*, pp. 81–90, Jun 1989.

[325] 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.

[326] 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.

[327] 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.

[328] 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.

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

[330] 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.

[331] 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.

[332] 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.

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

[334] P. Krueger and N. G. Shivaratri, "*Adaptive location policies for global scheduling*". *IEEE Trans. Softw. Eng.* **20(6)**, pp. 432–444, Jun 1994.

[335] D. W. Krumme, "*The SIMPLEX operating system*". In 3rd *Conf. Hypercubes, Concurrent Comput., & Appl.*, vol. I, pp. 381–383, Jan 1988.

[336] C. P. Kruskal and C. H. Smith, "*Definitions of granularity*". In *High Performance Computer Systems*, E. Gelenbe (ed.), pp. 257–268, North-Holland, 1988.

[337] C. P. Kruskal and A. Weiss, "*Allocating independent subtasks on parallel processors*". *IEEE Trans. Softw. Eng.* **SE-11(10)**, pp. 1001–1016, Oct 1985.

[338] 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.

[339] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, "*Parallel supercomputing today and the Cedar approach*". *Science* **231(4741)**, pp. 967–974, 28 Feb 1986.

[340] J. T. Kuehn and H. J. Siegel, "*Extensions to the C programming language for SIMD/MIMD parallelism*". In *Intl. Conf. Parallel Processing*, pp. 232–235, Aug 1985.

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

[342] M. Kumar and Y. Baransky, "*The GF11 parallel computer: programming and performance*". *Future Generation Comput. Syst.* **7(2&3)**, pp. 169–179, Apr 1992.

[343] H. T. Kung, "*Deadlock avoidance for systolic communication*". In 15th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 252–260, 1988.

[344] 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.

[345] J. Labarta, S. Girona, and T. Cortes, "*Analyzing scheduling policies using Dimamas*". *Parallel Computing* **23(1-2)**, pp. 23–34, Apr 1997.

[346] 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.

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

[348] 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.

[349] B. W. Lampson and D. D. Redell, "*Experience with processes and monitors in Mesa*". *Comm. ACM* **23(2)**, pp. 105–117, Feb 1980.

[350] R. P. LaRowe, Jr. and C. S. Ellis, "*OS experimentation and a user community coexist under the DUnX kernel*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 158–166, Aug 1991.

[351] 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.

[352] T. J. LeBlanc, M. L. Scott, and C. M. Brown, "*Large-scale parallel programming: experience with the BBN Butterfly parallel processor*". In *Proc. ACM/SIGPLAN PPEALS (Parallel Programming: Experience with Applications, Languages and Systems)*, pp. 161–172, Jul 1988.

[353] G. Lee, C. P. Kruskal, and D. J. Kuck, "*The effectiveness of combining in shared memory parallel computers in the presence of 'hot spots'*". In *Intl. Conf. Parallel Processing*, pp. 35–41, Aug 1986.

[354] 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.

[355] 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, S-W. Yang, and R. Zak, "*The network architecture of the Connection Machine CM-5*". In 4th *Symp. Parallel Algorithms & Architectures*, pp. 272–285, Jun 1992.

[356] 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.

[357] D. Lenoski, J. Laudon, K. Gharachorloo, W-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "*The Stanford Dash multiprocessor*". *Computer* **25(3)**, pp. 63–79, Mar 1992.

[358] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, "*The DASH prototype: logic overhead and performance*". *IEEE Trans. Parallel & Distributed Syst.* **4(1)**, pp. 41–61, Jan 1993.

[359] 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.

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

[361] 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.

[362] M. R. Leuze, L. W. Dowdy, and K. H. Park, "*Multiprogramming a distributed-memory multiprocessor*". *Concurrency — Pract. & Exp.* **1(1)**, pp. 19–33, Sep 1989.

[363] 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.

[364] 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.

[365] 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.

[366] W. Li and K. Pingali, "*Access normalization: loop restructuring for NUMA compilers*". In 5th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 285–295, Oct 1992.

[367] Z. Li and T. N. Nguyen, "*An empirical study of the workload distribution under static scheduling*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 259–263, Aug 1994.

[368] 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.

[369] B-H. Lim and A. Agarwal, "*Waiting algorithms for synchronization in large scale multiprocessors*". *ACM Trans. Comput. Syst.* **11(3)**, pp. 253–294, Aug 1993.

[370] G. E. Lim, "*The Silicon Graphics project supercomputer*". In *Singapore Supercomputing Conf.*, pp. 450–456, World Scientific, Dec 1990.

[371] 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.

[372] G. J. Lipovski and M. Malek, *Parallel Computing: Theory and Comparisons*. John Wiley & Sons, 1987.

[373] 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.

[374] J. Liu, V. Saletore, and T. G. Lewis, "*Safe self-scheduling: a parallel loop scheduling scheme for shared-memory multiprocessors*". *Intl. J. Parallel Programming* **22(6)**, pp. 589–616, Dec 1994.

[375] 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.

[376] 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.

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

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

[379] S. Lucco, "*A dynamic scheduling method for irregular parallel programs*". In *Proc. SIGPLAN Conf. Prog. Lang. Design & Implementation*, pp. 200–211, Jun 1992.

[380] E. Ma and D. G. Shea, "*E-kernel: an embedding kernel on the IBM Victor V256 multiprocessor for program mapping and network reconfiguration*". *IEEE Trans. Parallel & Distributed Syst.* **5(9)**, pp. 977–994, Sep 1994.

[381] 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.

[382] R. Maenner, R. L. Shoemaker, and P. H. Bartels, "*The Heidelberg Polyp system*". *IEEE Micro* **7(1)**, pp. 5–13, Feb 1987.

[383] 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.

[384] 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.

[385] 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.

[386] R. Männer, "*Hardware task/processor scheduling in a polyprocessor environment*". *IEEE Trans. Comput.* **C-33(7)**, pp. 626–636, Jul 1984.

[387] R. Männer, O. Stucky, and W. Ludwig, "*The Heidelberg Polyp multiprocessor project*". In *CONPAR 88*, C. R. Jesshope and K. D. Reinartz (eds.), pp. 456–463, Cambridge University Press, 1989.

[388] 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.

[389] 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.

[390] 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.

[391] 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.

[392] 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.

[393] D. May, "*Towards general-purpose parallel computers*". In *Natural and Artificial Parallel Computation*, M. A. Arbib and J. A. Robinson (eds.), chap. 5, MIT Press, 1990.

[394] 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.

[395] 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.

[396] 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.

[397] 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.

[398] 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.

[399] 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.

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

[401] 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.

[402] 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.

[403] 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.

[404] Minnesota Supercomputer Center, Inc., *The Distributed Job Manager Administration Guide*. 1993. Anonymous ftp `ec.msc.edu:/pub/LIGHTNING/djm_1.0.0_src.tar.Z`.

[405] 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.

[406] R. Mirchandaney, D. Towsley, and J. A. Stankovic, "*Analysis of the effect of delays on load sharing*". *IEEE Trans. Comput.* **38(11)**, pp. 1513–1525, Nov 1989.

[407] 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.

[408] 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.

[409] P. Mohapatra and C. R. Das, "*On dependability evaluation of mesh-connected processors*". *IEEE Trans. Comput.* **44(9)**, pp. 1073–1084, Sep 1995.

[410] 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.

[411] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr., "*Lazy task creation: a technique for increasing the granularity of parallel programs*". *IEEE Trans. Parallel & Distributed Syst.* **2(3)**, pp. 264–280, Jul 1991.

[412] P. Møller-Nielsen and J. Staunstrup, "*Problem-heap: a paradigm for multiprocessor algorithms*". *Parallel Comput.* **4(1)**, pp. 63–74, Feb 1987.

[413] 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.

[414] 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.

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

[416] 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.

[417] 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.

[418] 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.

[419] 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.

[420] 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.

[421] 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.

[422] 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.

[423] 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.

[424] 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.

[425] R. Nelson, D. Towsley, and A. N. Tantawi, "*Performance analysis of parallel processing systems*". *IEEE Trans. Softw. Eng.* **14(4)**, pp. 532–540, Apr 1988.

[426] 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.

[427] 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.

[428] T. D. Nguyen, R. Vaswani, and J. Zahorjan, "*Maximizing speedup through self-tuning of processor allocation*". In 10th *Intl. Parallel Processing Symp.*, pp. 463–468, Apr 1996.

[429] T. D. Nguyen, R. Vaswani, and J. Zahorjan, "*Parallel application characterization for multiprocessor scheduling policy design*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 175–199, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[430] T. D. Nguyen, R. Vaswani, and J. Zahorjan, "*Using runtime measured workload characteristics in parallel processor scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 155–174, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[431] L. M. Ni and G-F. E. Wu, "*Design tradeoffs for process scheduling in shared memory multiprocessor systems*". *IEEE Trans. Softw. Eng.* **15(3)**, pp. 327–334, Mar 1989.

[432] 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.

[433] D. M. Nicol and J. H. Saltz, "*Dynamic remapping of parallel computations with varying resource demands*". *IEEE Trans. Comput.* **37(9)**, pp. 1073–1087, Sep 1988.

[434] 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.

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

[436] R. Olson, "*Parallel processing in a message-based operating system*". *IEEE Software* **2(4)**, pp. 39–49, Jul 1985.

[437] Open Software Foundation, *Design of the OSF/1 Operating System*. Prentice Hall, 1993.

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

[439] 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.

[440] 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.

[441] J. D. Padhye and L. Dowdy, "*Dynamic versus adaptive processor allocation policies for message passing parallel computers: an empirical comparison*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 224–243, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[442] J. Palmer and G. L. Steele, Jr., "*Connection Machine model CM-5 system overview*". In 4th *Symp. Frontiers Massively Parallel Comput.*, pp. 474–483, Oct 1992.

[443] 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.

[444] C. M. Pancake, "*Multithreaded languages for scientific and technical computing*". *Proc. IEEE* **81(2)**, pp. 288–304, Feb 1993.

[445] C. H. Papadimitriou and M. Yannakakis, "*Towards an architecture-independent analysis of parallel algorithms*". *SIAM J. Comput.* **19(2)**, pp. 322–328, Apr 1990.

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

[447] 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.

[448] 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.

[449] 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.

[450] 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.

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

[452] 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.

[453] R. H. Perrott, "*A language for array and vector processors*". *ACM Trans. Prog. Lang. & Syst.* **1(2)**, pp. 177–195, Oct 1979.

[454] R. H. Perrott, D. Crookes, and P. Milligan, "*The programming language ACTUS*". *Software — Pract. & Exp.* **13(4)**, pp. 305–322, Apr 1983.

[455] R. H. Perrott and A. Zarea-Aliabadi, "*Supercomputer languages*". *ACM Comput. Surv.* **18(1)**, pp. 5–22, Mar 1986.

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

[457] J. L. Peterson and T. A. Norman, "*Buddy systems*". *Comm. ACM* **20(6)**, pp. 421–431, Jun 1977.

[458] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "*The IBM research parallel processor prototype (RP3): introduction and architecture*". In *Intl. Conf. Parallel Processing*, pp. 764–771, 1985.

[459] 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.

[460] P. Pierce, "*The NX/2 operating system*". In 3rd *Conf. Hypercubes, Concurrent Comput., & Appl.*, vol. I, pp. 384–390, Jan 1988.

[461] 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.

[462] 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.

[463] C. G. Plaxton, "*Load balancing, selection and sorting on the hypercube*". In *Symp. Parallel Algorithms & Architectures*, pp. 64–73, Jun 1989.

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

[465] C. D. Polychronopoulos, "*Parallel programming issues*". *Intl. J. High Speed Computing* **5(3)**, pp. 413–473, Sep 1993.

[466] C. D. Polychronopoulos, "*Toward auto-scheduling compilers*". *J. Supercomput.* **2(3)**, pp. 297–330, 1988.

[467] C. D. Polychronopoulos and U. Banerjee, "*Processor allocation for horizontal and vertical parallelism and related speedup bounds*". *IEEE Trans. Comput.* **C-36(4)**, pp. 410–420, Apr 1987.

[468] 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.

[469] C. D. Polychronopoulos, D. J. Kuck, and D. A. Padua, "*Execution of parallel loops on parallel processor systems*". In *Intl. Conf. Parallel Processing*, pp. 519–527, Aug 1986.

[470] J. L. Potter (ed.), *The Massively Parallel Processor*. MIT Press, 1985.

[471] L. Press, "*Before the Altair: the history of personal computing*". *Comm. ACM* **36(9)**, pp. 27–33, Sep 1993.

[472] 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.

[473] 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.

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

[475] M. J. Quinn, "*Analysis and implementation of branch-and-bound algorithms on a hypercube multicomputer*". *IEEE Trans. Comput.* **39(3)**, pp. 384–387, Mar 1990.

[476] S. Rai, J. L. Trahan, and T. Smailus, "*Processor allocation in hypercube multiprocessors*". *IEEE Trans. Parallel & Distributed Syst.* **6(6)**, pp. 606–616, Jun 1995.

[477] 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.

[478] 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.

[479] F. Ramme and K. Kremer, "*Scheduling a metacomputer by an implicit voting system*". In 3rd *Intl. Symp. High Performance Distributed Comput.*, pp. 106–113, Apr 1994.

[480] 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.

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

[482] R. F. Rashid, "*Threads of a new system*". *UNIX Review* **4(8)**, pp. 37–49, Aug 1986.

[483] D. A. Reed, "*The performance of multimicrocomputer networks supporting dynamic workloads*". *IEEE Trans. Comput.* **C-33(11)**, pp. 1045–1048, Nov 1984.

[484] A. P. Reeves, "*Parallel Pascal: an extended Pascal for parallel computers*". *J. Parallel & Distributed Comput.* **1(1)**, pp. 64–80, Aug 1984.

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

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

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

[488] B. S. Rosenburg, "*Low-synchronization translation lookaside buffer consistency in large-scale shared-memory multiprocessors*". In 12th *Symp. Operating Systems Principles*, pp. 137–146, Dec 1989.

[489] 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.

[490] 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.

[491] 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.

[492] 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.

[493] 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.

[494] C. H. Russell and P. J. Waterman, "*Variations on UNIX for parallel-processing computers*". *Comm. ACM* **30(12)**, pp. 1048–1055, Dec 1987.

[495] K. W. Ryu and J. JáJá, "*Efficient algorithms for list ranking and for solving graph problems on the hypercube*". *IEEE Trans. Parallel & Distributed Syst.* **1(1)**, pp. 83–90, Jan 1990.

[496] S. Sahni, "*Scheduling master-slave multiprocessor systems*". In *EURO-PAR '95 Parallel Processing*, pp. 611–622, Springer-Verlag, Aug 1995. Lecture Notes in Computer Science Vol. 966.

[497] J. Salmon, S. Callahan, J. Flower, and A. Kolawa, "*MOOSE: a multi-tasking operating system for hypercubes*". In 3rd *Conf. Hypercubes, Concurrent Comput., & Appl.*, vol. I, pp. 391–396, Jan 1988.

[498] J. Sanguinetti, "*Performance of a message based multiprocessor*". *Computer* **19(9)**, pp. 47–55, Sep 1986.

[499] 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.

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

[501] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

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

[503] G. E. Schmidt, "*The butterfly parallel processor*". In 2nd *Intl. Conf. Supercomputing*, vol. I, pp. 362–365, 1987.

[504] 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.

[505] W. Schröder, "*The distributed* PEACE *operating system and its suitability for MIMD message-passing architectures*". In *CONPAR 88*, C. R. Jesshope and K. D. Reinartz (eds.), pp. 27–34, Cambridge University Press, 1989.

[506] W. Schröder-Preikschat, "*PEACE — a software backplane for parallel computing*". *Parallel Comput.* **20(10-11)**, pp. 1471–1485, Nov 1994.

[507] 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.

[508] 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.

[509] M. L. Scott, T. J. LeBlanc, and B. D. Marsh, "*Multi-model programming in Psyche*". In 2nd *Symp. Principles & Practice of Parallel Programming*, pp. 70–78, Mar 1990.

[510] M. L. Scott, T. J. LeBlanc, B. D. Marsh, T. G. Becker, C. Dubnicki, E. P. Markatos, and N. G. Smithline, "*Implementation issues for the Psyche multiprocessor operating system*". *Computing Systems* **3(1)**, pp. 101–137, Winter 1990.

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

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

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

[514] 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.

[515] 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.

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

[517] 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.

[518] S. K. Setia, M. S. Squillante, and S. K. Tripathi, "*Processor scheduling on multiprogrammed, distributed memory parallel computers*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 158–170, May 1993.

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

[520] 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.

[521] 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.

[522] 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.

[523] D. G. Shea, W. W. Wilcke, R. C. Booth, D. H. Brown, Z. D. Christidis, M. E. Giampapa, G. B. Irwin, T. T. Murakami, V. K. Naik, F. T. Tong, P. R. Varker, and D. J. Zukowski, "*The IBM Victor V256 partitionable multiprocessor*". *IBM J. Res. Dev.* **35(5/6)**, pp. 573–590, Sep/Nov 1991.

[524] X. Shen and E. M. Reingold, "*Scheduling on a hypercube*". *Inf. Process. Lett.* **40(6)**, pp. 323–328, Dec 1991.

[525] 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.

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

[527] K. Shteiman, D. Feitelson, L. Rudolph, and I. Exman, "*Envelopes in adaptive local queues for MIMD load balancing*". In *Parallel Processing: CONPAR 92 – VAPP V*, pp. 479–484, Springer-Verlag, Sep 1992. Lecture Notes in Computer Science Vol. 634.

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

[529] W. Shu, "*Run-time support for user-level ultralightweight threads on distributed-memory computers*". *J. Supercomput.* **9(1/2)**, pp. 91–103, 1995.

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

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

[532] H. J. Siegel et al., "*Report of the Purdue workshop on grand challenges in computer architecture for the support of high performance computing*". *J. Parallel & Distributed Comput.* **16(3)**, pp. 199–211, Nov 1992.

[533] 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.

[534] 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.

[535] 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.

[536] J. P. Singh, J. L. Hennessy, and A. Gupta, "*Scaling parallel programs for multiprocessors: methodology and examples*". *Computer* **26(7)**, pp. 42–50, Jul 1993.

[537] A. Skjellum, S. G. Smith, N. E. Doss, A. Pleung, and M. Morari, "*The design and evolution of Zipcode*". *Parallel Comput.* **20(4)**, pp. 565–596, Apr 1994.

[538] J. Skovira, W. Chan, H. Zhou, and D. Lifka, "*The EASY - LoadLeveler API project*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 41–47, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[539] B. J. Smith, "*Latency and HEP*". In *High-Speed Computation*, J. S. Kowalik (ed.), pp. 139–144, Springer-Veralg, 1984. NATO ASI Series, Vol. F7.

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

[541] 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.

[542] A. Sohn, R. Biswas, and H. D. Simon, "*A dynamic load balancing framework for unstructured adaptive computations on distributed-memory multiprocessors*". In 8th *Symp. Parallel Algorithms & Architectures*, pp. 189–192, Jun 1996.

[543] 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.

[544] 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.

[545] M. S. Squillante, F. Wang, and M. Papaefthymiou, "*Stochastic analysis of gang scheduling in parallel and distributed systems*". *Performance Evaluation* **27&28**, pp. 273–296, Oct 1996.

[546] G. N. Srinivasa Prasanna, A. Agarwal, and B. R. Musicus, "*Hierarchical compilation of macro dataflow graphs for multiprocessors with local memory*". *IEEE Trans. Parallel & Distributed Syst.* **5(7)**, pp. 720–736, Jul 1994.

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

[548] 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.

[549] R. A. Steigerwald and M. L. Nelson, "*Concurrent programming in Smalltalk-80*". *SIGPLAN Notices* **25(8)**, pp. 27–36, Aug 1990.

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

[551] I. Stoica, H. Abdel-Wahab, and A. Pothen, "*A microeconomic scheduler for parallel computers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 200–218, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[552] Q. F. Stout, "*Mapping vision algorithms to parallel architectures*". *Proc. IEEE* **76(8)**, pp. 982–995, Aug 1988.

[553] C. B. Stunkel et al., "*The SP2 high-performance switch*". *IBM Syst. J.* **34(2)**, pp. 185–204, 1995.

[554] C. B. Stunkel and W. K. Fuchs, "*An analysis of cache performance for a hypercube multicomputer*". *IEEE Trans. Parallel & Distributed Syst.* **3(4)**, pp. 421–432, Jul 1992.

[555] S. Subramaniam and D. L. Eager, "*Affinity scheduling of unbalanced workloads*". In *Supercomputing '94*, pp. 214–226, Nov 1994.

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

[557] T. T. Y. Suen and J. S. K. Wong, "*Efficiant task migration algorithm for distributed systems*". *IEEE Trans. Parallel & Distributed Syst.* **3(4)**, pp. 488–499, Jul 1992.

[558] H. Sullivan, T. R. Bashkow, and D. Klappholz, "*A large scale, homogeneous, fully distributed parallel machine, II*". In 4th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 118–124, Mar 1977.

[559] X-H. Sun and L. M. Ni, "*Scalable problems and memory-bounded speedup*". *J. Parallel & Distributed Comput.* **19(1)**, pp. 27–37, Sep 1993.

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

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

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

[563] 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.

[564] P. Tang and P-C. Yew, "*Processor self-scheduling for multiple-nested parallel loops*". In *Intl. Conf. Parallel Processing*, pp. 528–535, Aug 1986.

[565] E. Tärnvik, "*Dynamo — a portable tool for dynamic load balancing on distributed memory multicomputers*". *Concurrency — Pract. & Exp.* **6(8)**, pp. 613–639, Dec 1994.

[566] P. J. Teller, "*Translation-lookaside buffer consistency*". *Computer* **23(6)**, pp. 26–36, Jun 1990.

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

[568] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "*Mach threads and the Unix kernel: the battle for control*". In *Proc. Summer USENIX Technical Conf.*, pp. 185–197, Jun 1987.

[569] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr., "*Firefly: a multiprocessor workstation*". *IEEE Trans. Comput.* **37(8)**, pp. 909–920, Aug 1988.

[570] S. Thakkar, P. Gifford, and G. Fielland, "*Balance: a shared memory multiprocessor system*". In 2nd *Intl. Conf. Supercomputing*, vol. I, pp. 93–101, 1987.

[571] S. S. Thakkar and M. Sweiger, "*Performance of an OLTP application on Symmetry multiprocessor system*". In 17th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 228–238, May 1990.

[572] Thinking Machines Corp., *Connection Machine CM-5 Technical Summary.* Nov 1992.

[573] R. H. Thomas and W. Crowther, "*The Uniform System: an approach to runtime support for large scale shared memory parallel processors*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 245–254, Aug 1988.

[574] C. D. Thomborson, "*Does your workstation computation belong on a vector supercomputer?*". *Comm. ACM* **36(11)**, pp. 41–49, Nov 1993.

[575] 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.

[576] 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.

[577] D. Towsley, C. G. Rommel, and J. A. Stankovic, "*The performance of processor sharing for scheduling fork-join jobs in multiprocessors*". In *High Performance Computer Systems*, E. Gelenbe (ed.), pp. 145–156, North-Holland, 1988.

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

[579] A. Tripathi, N. M. Karnik, S. P. Koneru, C. Nock, R. Tewari, K. Day, and T. Noonan, "*Configuration management in the Nexus distributed operating system*". *Concurrency — Pract. & Exp.* **6(4)**, pp. 325–338, Jun 1994.

[580] 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.

[581] 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.

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

[583] D. L. Tuomenoksa and H. J. Siegel, "*Task preloading schemes for reconfigurable parallel processing systems*". *IEEE Trans. Comput.* **C-33(10)**, pp. 895–905, Oct 1984.

[584] 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.

[585] J. Turek, W. Ludwig, J. L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelshohn, and P. S. Yu, "*Scheduling parallelizable tasks to minimize average response time*". In 6th *Symp. Parallel Algorithms & Architectures*, pp. 200–209, Jun 1994.

[586] J. Turek, J. L. Wolf, K. R. Pattipati, and P. S. Yu, "*Scheduling parallelizable tasks: putting it all on the shelf*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 225–236, Jun 1992.

[587] J. Turek, J. L. Wolf, and P. S. Yu, "*Approximate algorithms for scheduling parallelizable tasks*". In 4th *Symp. Parallel Algorithms & Architectures*, pp. 323–332, Jun 1992.

[588] 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.

[589] T. H. Tzen and L. M. Ni, "*Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers*". *IEEE Trans. Parallel & Distributed Syst.* **4(1)**, pp. 87–98, Jan 1993.

[590] 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.

[591] J. D. Ullman, "*NP-complete scheduling problems*". *J. Comput. Syst. Sci.* **10(3)**, pp. 384–393, Jun 1975.

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

[593] L. G. Valiant, "*A bridging model for parallel computation*". *Comm. ACM* **33(8)**, pp. 103–111, Aug 1990.

[594] R. J. van der Pas and J. M. van Kats, "*Parallelism in a multi-user environment*". *Parallel Comput.* **17(2/3)**, pp. 185–196, Jun 1991.

[595] A. M. van Tilborg and L. D. Wittie, "*High-level operating system formation in network computers*". In *Intl. Conf. Parallel Processing*, pp. 131–132, 1980.

[596] A. M. van Tilborg and L. D. Wittie, "*Wave scheduling — decentralized scheduling of task forces in multicomputers*". *IEEE Trans. Comput.* **C-33(9)**, pp. 835–844, Sep 1984.

[597] M. T. Vandevoorde and E. S. Roberts, "*WorkCrews: an abstraction for controlling parallelism*". *Intl. J. Parallel Programming* **17(4)**, pp. 347–366, Aug 1988.

[598] 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.

[599] Z. G. Vranesic, M. Stumm, D. M. Lewis, and R. White, "*Hector: a hierarchically structured shared-memory multiprocessor*". *Computer* **24(1)**, pp. 72–79, Jan 1991.

[600] 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.

[601] T. D. Wagner, E. Smirni, A. W. Apon, M. Madhukar, and L. W. Dowdy, "*Measuring the effects of thread placement on the Kendall Square KSR1*". In 8th *Intl. Parallel Processing Symp.*, pp. 618–624, Apr 1994.

[602] 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.

[603] C. A. Waldspurger and W. E. Weihl, "*Lottery scheduling: flexible proportional-share resource management*". In 1st *Symp. Operating Systems Design & Implementation*, pp. 1–11, USENIX, Nov 1994.

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

[605] D. J. Wallace, "*Large scale applications of transputers: achievement and perspective*". In 9th *Intl. Conf. Computing Methods in Applied Sciences & Engineering*, pp. 436–446, SIAM, 1990.

[606] 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.

[607] 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.

[608] Q. Wang and K. H. Cheng, "*A heuristic of scheduling parallel tasks and its analysis*". *SIAM J. Comput.* **21(2)**, pp. 281–294, Apr 1992.

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

[610] 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.

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

[612] M. Weiser, A. Demers, and C. Hauser, "*The portable common runtime approach to interoperability*". In 12th *Symp. Operating Systems Principles*, pp. 114–122, Dec 1989.

[613] S. R. Wheat, A. B. Maccabe, R. Riesen, D. W. van Dresser, and T. M. Stallcup, "*PUMA: an operating system for massively parallel systems*". *Scientific Programming* **3(4)**, pp. 275–288, Winter 1994.

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

[615] 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.

[616] R. D. Williams, "*Performance of dynamic load balancing algorithms for unstructured mesh calculations*". *Concurrency — Pract. & Exp.* **3(5)**, pp. 457–481, Oct 1991.

[617] 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.

[618] P. H. Worley, "*The effect of time constraints on scaled speedup*". *SIAM J. Sci. Statist. Comput.* **11(5)**, pp. 838–858, Sep 1990.

[619] C. Wu and T. Feng, "*On a class of multi-stage interconnection networks*". *IEEE Trans. Comput.* **C-29(8)**, pp. 694–702, Aug 1980.

[620] Y-J. C. Wu and J-L. C. J. Wu, "*Scheduling parallel programs with non-uniform parallelism profiles*". In *Supercomputing '91*, pp. 502–511, Nov 1991.

[621] W. A. Wulf, R. Levin, and S. P. Harbison, *HYDRA/C.mmp: An Experimental Computer System*. McGraw-Hill, 1981.

[622] L. Xiao-ping and H. Amano, "*A static scheduling scheme for a parallel machine (SM)^2-II*". In *Parallel Arch. & Lang. Europe*, vol. I, pp. 118–135, Springer-Verlag, Jun 1989. Lecture Notes in Computer Science Vol. 365.

[623] 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.

[624] 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.

[625] Q. Yang and H. Wang, "*A new graph approach to minimizing processor fragmentation in hypercube multiprocessors*". *IEEE Trans. Parallel & Distributed Syst.* **4(10)**, pp. 1165–1171, Oct 1993.

[626] 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.

[627] S. Y. Yoon, O. Kang, S. R. Maeng, and J. W. Cho, "*A heuristic processor allocation strategy in hypercube systems*". In 3rd *IEEE Symp. Parallel & Distributed Processing*, pp. 574–581, Dec 1991.

[628] 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.

[629] 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.

[630] 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.

[631] J. Zahorjan, E. D. Lazowska, and D. L. Eager, "*The effect of scheduling discipline on spin overhead in shared memory parallel systems*". *IEEE Trans. Parallel & Distributed Syst.* **2(2)**, pp. 180–198, Apr 1991.

[632] J. Zahorjan, E. D. Lazowska, and D. L. Eager, *Spinning Versus Blocking in Parallel Systems with Uncertainty*. Technical Report 88-03-01, Dept. Computer Science, University of Washington, Mar 1988.

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

[634] 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.

[635] S. Zhou and T. Brecht, "*Processor pool-based scheduling for large-scale NUMA multiprocessors*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 133–142, May 1991.

[636] 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.

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

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

Index

accounting, 88
adaptive partitioning, 33–37
affinity scheduling, 55, 60, 77
agenda parallelism, 39
Alewife, 44
all fall down, 70, 103
Alliant FX/8, 68
Amoeba, 42
AP1000, 44
ASAP, 42
ASAT, 38, 41, 42
average parallelism, 27, 35, 36

backfilling, 33, 97
backward compatability, 11
batch vs. interactive, 3, 14, 93
BBN Butterfly, 44
bin packing, 9, 31, 74
bottleneck, *see* contention
buddy system, 20, 24, 46, 72
    2-D, 25, 99
buffer overflow, 63, 76
busy waiting, 33, 51, 62, 63, 75, 87

C.mmp, 51
cache affinity, 55, 77
cache coherence, 51
Cedar, 57, 68
Chagori, 68
checkpointing, 66, 92
CHoPP, 24
chore, 39
Chrysalis, 44
chunking, 50, 58–60
Cilk, 42
clean termination, 30
clusters, 28, 54, 68
Cm*, 44, 70
CM-2, 15, 70
CM-5, 19, 68, 69, **102–105**
communication interference, 29, 78, 84
compaction, 20, 80
competitive algorithm, 76
complexity, 23
Concentrix, 68
configurable buses, 28
contention, 39, 52, 55, 57
context switch, 4, 67, 104, *see* multi-context-switch
    in hardware, 44, 83, 102
    overhead, 82
Convex C2/C4, 42
coscheduling, 65, 71
Cosmic Cube, 19, 43
coverage set, 26
Cray multiprocessors, 4, 93
Cray T3D, 24, 28, 101
Cray T3E, 80, 88

DASH, 51
database machines
    scheduling, 4
dedicated machine, 2
    illusion of, 83–84
DHC, 47, 72–73
diffusion, 48
divergence, 9–11
DJM, 104
do not disturb, 73
DQT, 47, 71
draining the network, 68, 70, 92, 103
dynamic partitioning, 37–42, 80, 107–112
Dynix, 51

EASY, 97
ecomonic model, 46, 88
effective computation rate, 35
efficiency, 27, 79
Elxsi, 44
embeddings, 27
EMBOS, 44
EMMA2, 44
equipartitioning, 34, 40
evolving job, 11, 37, 40

execution signature, 35

fair share, 88
fairness, 87–88
family scheduling, 57, 65
farming, 39
fat tree, 19, 102
FCFS, 30
feedback, 27
fetch-and-add, 52
fixed partitioning, 14
Flagship, 18, 46
flexible allocation, 1, 6, 24, 64, 80, *see* partitioning, flexibility
folding, 40
fragmentation, 14–15, 25, 32, 42, 73, **80–82**, 88

gang scheduling, 7, 9, 62–75, 112, *see* multi-context-switch
  and priorities, 67, 73
  and swapping, 66, 98, 101
  vs. coscheduling, 65
global queue, 50–56
  chunking, 50, 58–60
  in two-level scheduling, 57–60
  vs. local queue, 56–57
  wait-free, 52
Goodyear MPP, 2
gradient model, 49
granularity, 47, 122
Gray code, 20

handoff, 61
hardware queue, 44
heap, 51
Helios, 44
HEP, 44, 76, 82, 102
heterogeneous networks, 4
hierarchy of queues, 57, 72
hot spot, 84
HPF, 33
Hydra, 51
hypercube, 19
  partitioning, 19–24

I/O, 62, 79–80, 102
IBM GF11, 2
IBM RP3, 57, 84
IBM SP2, 24, 67, **96–97**
Illiac IV, 2, 15
interactive vs. batch, 3, 14, 93
interference, 29, 78, 84
interval set, 25
iPSC, 19, 43
IRIX, 51, 66

J Machine, 44
job, 4
job execution, 29
job history, 36, 113
job mix, 79, 94
job phases, 11, 37, 40, 67, 79
job types, 10, 96, 100

K2, 68
kernel threads vs. user threads, 60–62, 121–122
knowledge about jobs, 27, 35, 36, 113
KSR1, 48, **105**

largest job first, 31
lattice, 21
layered design, 40
LLNL gang scheduler, 70, 74, **100–101**
load balancing, 48–50
  vs. mapping, 45
  implicit, 45
load fluctuations, 33, 35, 37
load metric, 45
load sharing, 50
loading jobs, 29
LoadLeveler, 97
local queue, 43–50
  in hardware, 44
  vs. global queue, 56–57
locality, 13, 54
  proximity of communicating threads, 24, 41, 47, 77–78
  proximity of threads and data, 44, 50, 76–77

lock
on global queue, 51
preempting holding thread, 39, 61
loop scheduling, 58–60, 68

Mach, 39, 51, 54, 57, 120
Makbilan, 56, 70
malleable job, 11, 39, 106
mapping, 45–47, *see* task scheduling
vs. load balancing, 45
the mapping problem, 3
master-slave, 39
mathematical tractability, 11
matrix algorithm, 70
MAXI, 56, 62, 70
maximal partition size, 34
Medusa, 70
Meiko CS-2, 67, 70, 99–100
memory
nodes with different amounts, 94, 96, **98**
memory considerations, 36, 41, 77, 79, 89–91
memory pressure, 37, 66, 89
MEMSY, 44
metacomputing, 4
MICROS, 24
migration, 20, 48–50, 80, 106
minimal partition size, 17, 19, 36, 40, 42, 79
moldable job, 11, 33
Moose, 43
MOSIX, 49
multi-context-switch, 66–68, 91, 101, 104, *see* draining the network
multiple queues, 32, 72, 94
multiprogramming, 5
interactive and batch, 3, 14
motivation, 5–6
pros and cons, 3
multistage network, 17, 84
combining, 52
partitioning, 17
multitasking, 5
multithreading, 44, 102

nCUBE, 20, 43
NETRA, 47
non-contiguous allocation, 24, 27
NOW, 67, **105–106**
NP-completeness, 3, 74
NQS, 32, **93–94**, 98
NUMA, 28, 77, 89
NX/2, 43, 97
NYU Ultracomputer, 52, 84

off-line, 37, 74
on-line identification of gangs, 65
OSF/1 AD, 43, 97
overhead, 82

paging, 89, *see* memory considerations and swapping
Paragon, 67, 97–99
parallel slack, 82
partition size, 27, 34–36, 40–42
on-line tuning, 41
partitioning, 7, **11–42**
adaptive, 33–37
classification, 13
constraints, 12, 38, *see* partitioning, hypercube and mesh
dynamic, 37–42, 80, 107–112
fixed, 14
flexibility, 11–12, 15, 19, 24, 38
hypercube, 19–24
mesh, 25–27
multistage network, 17–19
torus, 28
variable, 14–33
PASM, 17, 32
PBS, 94–95
PE (processing element), 4
PEACE, 43
phases, 11, 37, 40, 67, 79
Phish, 42
plan 9, 42
political scheduling, 11
Polyp, 28, 46, 54
predict queueing time, 35, 86
predictability, 83, 86
preemption, *see* time slicing
of thread holding lock, 39, 61

synchronized, *see* multi-context-switch
priority inheritance, 61
priority queue, 51, 55, 67, 101
problem heap, 39
process, 4
process control, 38, 42
processor allocation, 7
one at a time, 42
program shape, 36
programming model, 11, 39, 80
Psyche, 39
PUMA, 44

queue, *see* local queue, global queue, and multiple queues
hierarchy, 57, 72
queueing time prediction, 35, 86

real-time scheduling, 3
redistribution, 40–41
reservation, 32
responsiveness, 64, 86
rigid job, 10, 14
robustness, 84
RWC-1, 68, 71

scan, 32
scheduler activations, 61–62
scheduling
and synchronization, 75–76, 84
assumptions, 9–11
comparison, 10, 75, 107
short term vs. long term, 90, 102
taxonomy, 7–9
scheduling order, 30–31, 55
self scheduling, 39, 42, 46, **58**
SHARE, 67, 70, 74
shifts, 94–95, 98
SIMD, 12, 15, 17, 64, 70
SIMPLEX, 43
single-level scheduling, **6–7**
SJF, 30
smallest cumulative demand first, 31
smallest job first, 30
space slicing, 9, *see* partitioning
speedup, 27, 79
SPMD, 44
SpoC, 42
standardization, 11
StarOS, 44
starvation, 30, 81, 85
subcube allocation, 19–23
submesh allocation, 25–27
super-unitary speedup, 79
supply and demand, 46
SUPRENUM, 43
swapping, 66, 90, 101
Sylvan, 43
Symunix, 52
synchronization, 75–76, 84

task queue, 39
task scheduling, 3, 47–48, 83
taxonomy, 7–9
Tera MTA, 44, 76, 82, 88, 102
thread, 4, **118–122**
first class, 61
gangs, 64
user vs. kernel, 60–62, 121–122
thread package, 39, 61, 120, 121
throughput machines, 4
time sharing, *see* time slicing
time slicing, 8–9, 86
and fragmentation, 80–82
different quanta, 73, 88
gang scheduling, 7, 62–75
independent, 7, 43–44, 50–56
overhead, 82
TLB consistency, 85
topology, 24, 47, 54
torus, 28, *see* submesh allocation
TRAC, 18
transputer, 44
Trillium, 44
2-D buddy system, 25, 99
two-level scheduling, **6–7**, 11, 57–62, 107–112
two-phase blocking, 76

UMA, 38, 77
Uniform System, 39

upcall, 39, 61
user frustration, 12, 86
user threads vs. kernel threads, 60–62, 121–122
utilization, 79–82
    achieved, 2, 31

variable partitioning, 14–33
virtual PEs, 38, 39, 47, 82, 119, 121
VPP500, 24, 28

wait-free queue, 52
Warp, 2
wave scheduling, 24
work stealing, 42
workpile, 39, 80

Xylem, 68