# LIMITED DISTRIBUTION NOTICE
# (NOT TO BE REMOVED FROM DOCUMENT):

**This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communication and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).**

# A Survey of Scheduling in Multiprogrammed Parallel Systems

Dror G. Feitelson
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

**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 three ways, which are relevant for both one-level and two-level scheduling. Some systems 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. Other systems use space slicing, and partition the processors statically or dynamically among the different jobs. The third mode is to use both time slicing and space slicing, as in gang scheduling. Systems using all three approaches are described, and the implications of the various 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.

## COMMENTS?

This is not a final version. Comments regarding inaccuracies, omissions, alternative viewpoints, and experience with real systems are very welcome. Address comments to:


email: feit@watson.com.ibm
mail: Dror Feitelson
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
voice: (914) 945-1402
fax: (914) 945-2141

# Contents


**1 Introduction** **1**

**2 A Classification of Multiprogramming Schemes** **5**
2.1 Single-level Scheduling and Two-Level Scheduling . . . . . 5
2.2 The Two Dimensions of Sharing . . . . . 7
2.3 Mechanisms of Multiprogramming . . . . . 9

**3 Partitioning** **11**
3.1 Fixed Partitioning . . . . . 13
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 . . . . . 22
3.2.3 Making Scheduling Decisions . . . . . 24
3.3 Adaptive Partitioning: Setting the Allocation at Load Time . . . . . 27
3.4 Dynamic Partitioning: Allocation May Change at Runtime . . . . . 32
3.5 Processors Allocated Singly . . . . . 35

**4 Independent PEs** **36**
4.1 Local Queues . . . . . 36
4.1.1 Mapping . . . . . 38
4.1.2 Load balancing . . . . . 40
4.2 A Global Queue . . . . . 42
4.3 Combination of Local and Global Queues . . . . . 48
4.4 Optimizations for Two-Level Scheduling . . . . . 49
4.4.1 Chunking from a Global Queue . . . . . 49
4.4.2 Variations on the Thread Model . . . . . 52

**5 Gang Scheduling** **53**
5.1 Definition and Motivation . . . . . 54
5.2 Implementing Multi-Context-Switching . . . . . 56
5.3 Gang Scheduling within Predefined Partitions . . . . . 57
5.4 Gang Scheduling with Dynamic Repartitioning . . . . . 59
5.5 Theoretical Results . . . . . 63

**6 Implications of the Sharing Scheme** **63**
6.1 Interaction with Applications . . . . . 63
6.2 Impact on System Performance . . . . . 67
6.3 Individual Users in a Multiuser System . . . . . 70
6.4 Implementation Issues . . . . . 75

**7 Catering for Many Masters** **78**
7.1 The Network Queueing System (NQS) . . . . . 78
7.2 The Lawrence Livermore Gang Scheduler . . . . . 80
7.3 The Connection Machine CM-5 . . . . . 81
7.4 The Meiko CS-2 . . . . . 84
7.5 The Intel Paragon . . . . . 85
7.6 The Kendall Square Research KSR1 . . . . . 86
7.7 Networks of Workstations . . . . . 87

**8 Conclusions** **87**

**A Terminology** **94**

**B The Thread Model** **97**

**References** **101**

# 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 CPU. Thus parallel systems allow a two-dimensional division of resources among competing jobs, both in time and in space.

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

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

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

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

*Scope of this survey*

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

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 [34, 58], which was designed to carry out numerical solutions for partial differential equations such as those in weather codes, the Goodyear MPP [37, 328], used by NASA to process satellite images, the IBM GF11 [40, 236], 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 [69], and the Warp systolic array [16], 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 [237, 57].

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 [221]. 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 [281].

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, such as the DEC Firefly [398, 351], the Xerox Dragon [286], the IBM TOP-1 [365], and the SPUR project from Berkeley [187]. 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 [281, 291]. 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 algorithmic scheduling, a.k.a. the mapping problem [417, 201, 166, 43, 298]. 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 [442]. 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] 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 [332, 331, 381, 366, 206, 24], whereas multiprogramming is more concerned with throughput and fair sharing of resources. Likewise, scheduling issues on parallel database machines are not included [56, 117], as they too are quite different from general purpose systems.

[1]SPMD stand for Single Program Multiple Data [108]. 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.

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 [322]. 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 and Cray Y-MP. 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 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 [314].

*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.

# 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 [220, 314]. 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) [389, 143, 420, 322]. Again, sharing provides users with better access to the system, and — depending on the application — even with interactive response times [359]. Again, this is important both because it enables users to make faster progress towards a solution [403], and because adequate access for multiple users is often crucial in order to fund costly parallel supercomputers [59, 112].

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.

## 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 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 rate. 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

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. Other surveys of scheduling and resource allocation have used other classifications. For example, Casavant and Kuhl classify different scheduling algorithms according to their design [76]. 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 [275].

As noted above, the current trend is towards sharing parallel machines among many users. The only way to share uniprocessors is by time slicing. But with parallel machines, it is also possible to partition the machine spatially. This leads to the options of sharing among users by using time slicing, space slicing, or both. Fig. 1 presents a taxonomy based on these options. Naturally, scheduling schemes used in real machines are not designed specifically to ease their classification. Therefore this classification is not always completely adequate for the description of certain systems. However, it provides insights into the basic choices.

The first branch of the taxonomy, that of pure *time slicing*, is a direct extension of current practices on uniprocessors. Indeed, many systems that are based on parallelizations of conventional uniprocessor systems use this approach. It is especially common with shared-memory machines, where the same data structures that were originally designed for uniprocessors are now accessed by multiple PEs. The subdivisions reflect different options in parallel use of a ready queue: either the whole machine serves the queue as a single unit, or each PE serves the queue independently, or they each have a separate queue.

The third branch, that of pure *space slicing*, is more common in distributed memory machines. This is the simplest approach to implement: the machine is *partitioned* among the users, who are then left to their own devices. The operating system does not provide any additional support (except possibly for message passing [287, 149, 318, 319]). This reduces system overhead, and prevents the system from interfering with user applications. The subdivisions reflect different levels of flexibility in making the partitions.

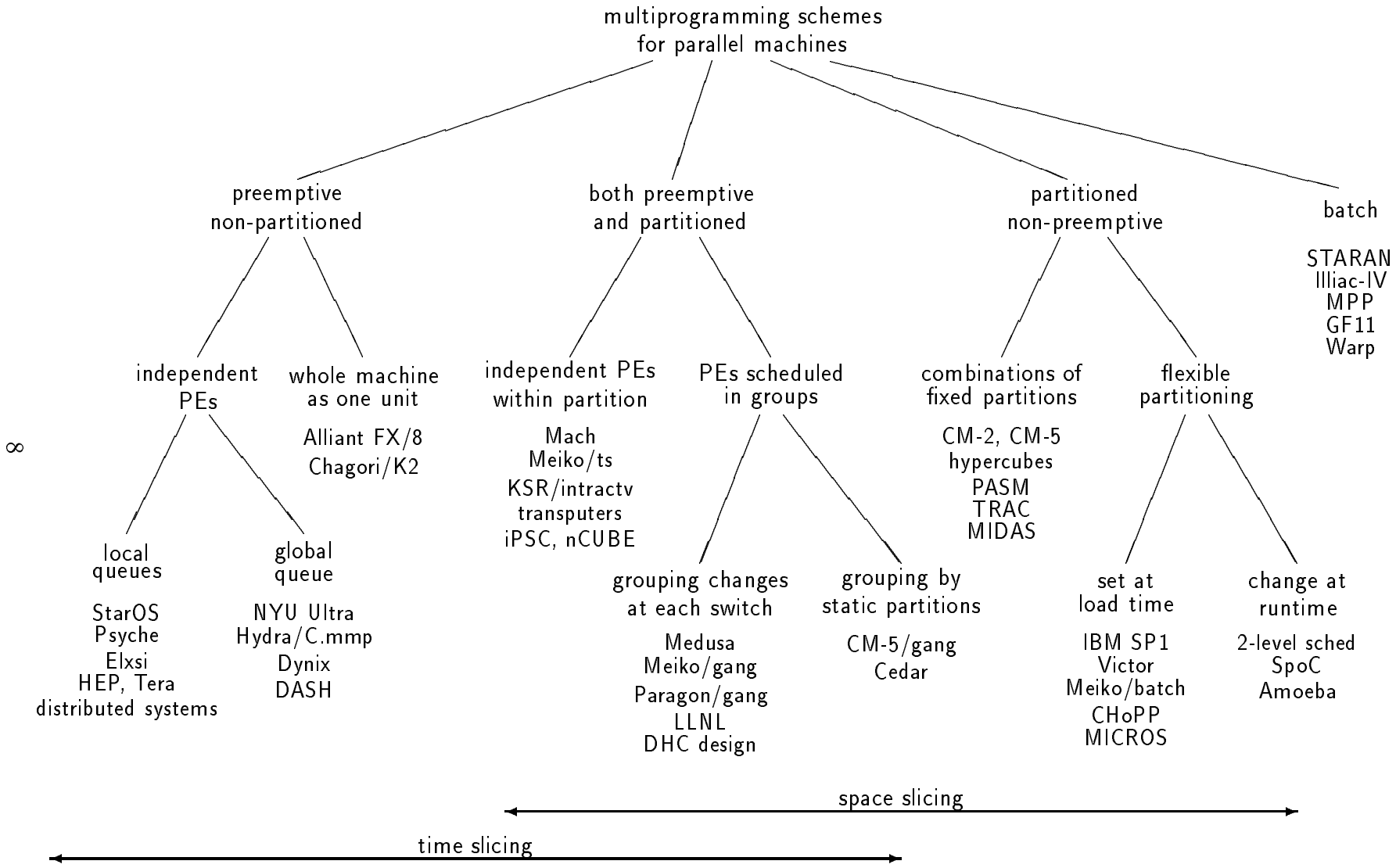



**Figure 1:** *Taxonomy of scheduling policies for multiprogrammed parallel machines.*

The second branch, representing hybrid time-slicing and space-slicing, is somewhat more complex than the other two. One of the interesting new scheduling schemes is *gang scheduling*, which is typically used loosely to indicate that the scheduling of a number of PEs is correlated in some way. We shall reserve this term for a more specific meaning, namely when the operating systems schedules a certain set of threads *simultaneously* on distinct PEs, with a one-to-one mapping of threads to PEs. With gang scheduling, either all these threads execute or none execute. We further restrict this term to imply preemptive scheduling, with the understanding that all the threads are preempted at the same time. The reason for excluding non-preemptive gang scheduling is that scheduling a set of threads using a one-to-one mapping on a set of PEs, and running them to completion, is exactly what pure space-slicing systems do. Thus a better term for it is simply "partitioning". Note that the application might use the non-preemptive threads to multiplex many user-level threads, but this is no concern of the underlying operating system.

The fourth branch, batch scheduling across the whole machine, is shown for completion. This is typically used in machines that are dedicated to a single application.

### 2.3 Mechanisms of Multiprogramming

One way to classify multiprogrammed systems is by the concepts that are applied. Another is by the mechanisms which are used to implement these concepts. The classification in Fig. 1 is based on the concepts of time-slicing and space-slicing. In this subsection we present another view of the same classification, based on mechanisms.

The main observation is that the mechanisms of time-slicing are largely independent of those for space-slicing. Thus the classification of Fig. 1 can be represented by a two-dimensional grid, with time-slicing on one axis and space-slicing on the other (Table I). 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 [150].

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 by 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 (according to our definition of this term). As gang scheduling requires coordinated context switching across the PEs (also called *multi-context-switching*), these are harder to implement. However, this approach is gaining in popularity: at least three recent commercial systems provide gang scheduling (the Connection Machine CM-5, the Intel Paragon, and the Meiko CS-2), and a number of experimental systems have also been reported in the literature [245, 383, 73, 144, 192]. These systems and others are reviewed in Section 5.

| | | | time slicing | | | |
|---|---|---|---|---|---|---|
| | | | yes | | | no |
| | | | independent PEs | | gang scheduling | |
| | | | global queue | local queues | | |
| space slicing | yes | flexible | Mach | Meiko/ts<br>KSR/intractv<br>transputers<br>Chrysalis | Medusa<br>Meiko/gang<br>Paragon/gang<br>LLNL<br>MAXI/gang | IBM SP1, Victor<br>Meiko/batch<br>KSR/batch<br>TRAC, MICROS<br>2-level/bottom<br>Amoeba |
| | | structured | – | NX/2 on iPSC/2<br>nCUBE | CM-5<br>Cedar<br>DHC design | CM-2<br>PASM<br>hypercubes |
| | no | | NYU Ultra<br>Dynix<br>2-level/top<br>Hydra/C.mmp | StarOS<br>Psyche<br>Elxsi<br>HEP, Tera | Alliant FX/8<br>Chagori on K2 | Illiac IV<br>MPP<br>GF11<br>Warp |

**Table I:** *Grid of combinations of mechanisms for time-slicing and space-slicing.*

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, they 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.

# 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 [409], 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 [447, 408]. 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

[2]SIMD stands for Single Instruction-stream Multiple Data-stream, and MIMD stands for Multiple Instruction-stream Multiple Data-stream [150]. 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.

sophisticated algorithms to make the partition decisions. The following criteria have been proposed to evaluate and compare partitionable systems [266, 339]:

- *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. 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 [291, 339]: fixed, variable, adaptive, and dynamic. These are summarized in Table II 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.

| *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 II:** *The four types of partitioning.*

### 3.1 Fixed Partitioning

Fixed partitions are set by the system administrator, typically for reasons related to access control. This allows certain fractions 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 [281, 291]. 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). 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

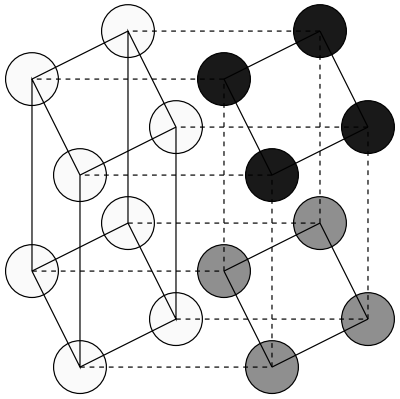


**Figure 2:** *Example of partitioning a 4-cube into a 3-cube and a pair of 2-cubes. Dashed links are not used in this configuration.*

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 Meiko CS-2, and the Intel Paragon. 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. 2). In this case, there is no internal fragmentation. However, internal fragmentation may occur in cases where the allocation is rounded up to one of the predefined sizes, but the job does not utilize the extra PEs.

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

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

### 3.2.1 Combinations of Partitions Based on Powers of Two

Many parallel machines are not just an ensemble of PEs. Rather, they are built as a group of constituents, each of which may be further divided into even smaller parts, before single PEs are reached. For example, a full-sized Connection Machine CM-2, with 65536 single-bit PEs, is composed of four quadrants of 16384 PEs each [410]. 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 $4 \times 4$ crosspoint switch that interfaces the host front-ends to the sequencers.

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 [34], 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.

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 [349], The NYU Ultracomputer [171, 170], the IBM RP3 [316], PASM [373, 372], TRAC [61, 258, chap. 7], and

Cedar (where clusters are connected to the network rather than individual PEs) [156, 218]. In distributed memory machines, all ports are connected to PEs. Examples include the CM-5 [245], the Meiko CS-2, and the IBM SP1 [386].

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 $2 \times 2$ 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 [439, 369, 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 $i$th bit position, counting from the left. An example for $n = 16$ is given in Fig. 3.

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 the their bottom output. 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. 3 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 [370, 200, 369, 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 [373, 372]. 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 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 [412].

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 mean-

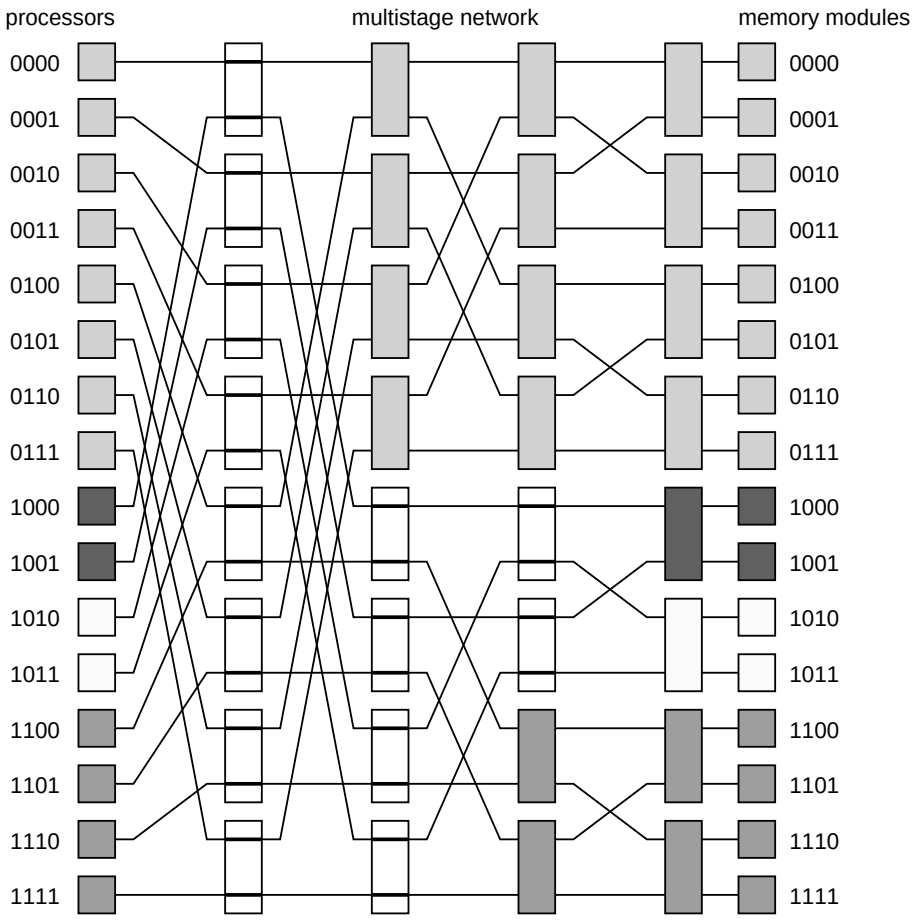



**Figure 3:** *partitioning a multistage network into independent partitions. The elements of each partition are identified by a different shading. Switches set to "straight" are not shaded.*

ingful in such a small machine, the architecture is scalable and provides a rich variety of configurations that can be changed at high speed [61, 258, chap. 7].

An interesting variant is the Flagship parallel reduction machine, which is based on a Delta network [432, 433]. 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 reouted 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 [392]. 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, but actually implemented as a multistage network [245]. 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 SP1, 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 [357]. A number of commercial offerings have been made since then, the most enduring of which are the iPSC series from Intel [20] and the nCUBE machines [180, 305, 127]. Both provide partitioning as described below.

Many hypercube applications are specifically tailored to the topology (e.g. [30, 343, 330]). 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. It is easy to see that there are many ways in which to satisfy this requirement. For example, all the nodes whose IDs start with the bit "1" in an $n$-dimensional hypercube form an $(n-1)$-dimensional hypercube. Likewise, it is possible to use any other of the $n$ bit positions to partition the hypercube into two halves which are hypercubes smaller by one dimension.

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

It is harder to re-unite subcubes after an application terminates. Needless to say, not any arbitrary set of $2^{n-1}$ nodes in an $n$-dimensional hypercube form a subcube. 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 [85, 84, 80], thereby creating two 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[4], where dimensions of the cube correspond to levels of the tree [83, 126, 109, 24]

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

[4]In a buddy system, a set of resources is allocated in blocks whose sizes are defined by a recurrence

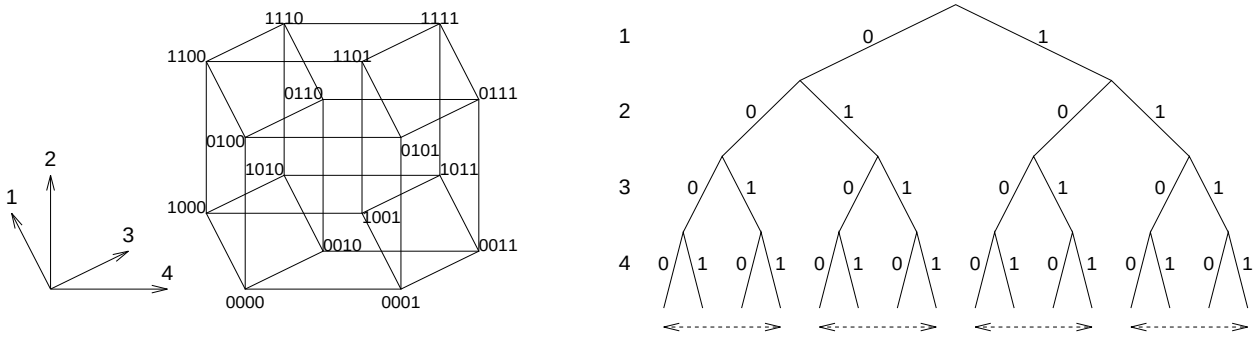



**Figure 4:** *The buddy system maps dimensions of the hypercube to levels of a binary tree, and uses the tree structure to partition the hypercube.*

(Fig. 4). 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 [226].

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. 4, 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 [83, 84], 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. 5, this can also be interpreted as a tree with 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. 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 [83].

Further improvements are achieved if all possible $(k-1)$-dimensional partners are considered, even if they are not adjacent in the tree structure [6]. 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 [35]. Fig. 5 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

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 [215, 315, 216, pp. 442–445].

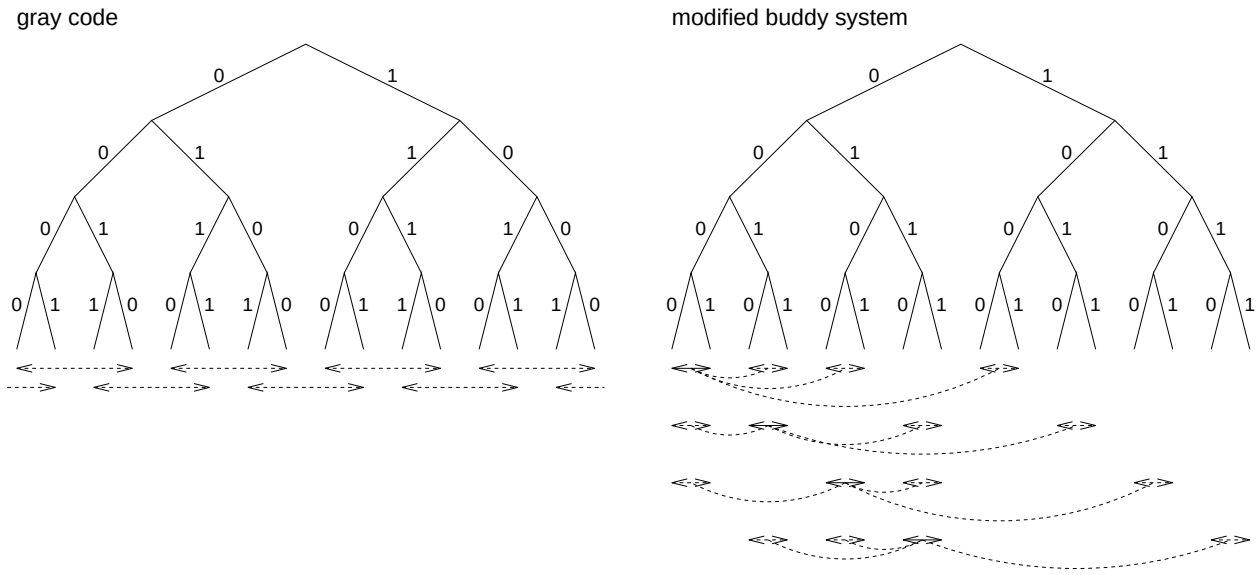



**Figure 5:** *In the Gray code method, pairs of successive $(k-1)$-dimensional subcubes are considered for creating a $k$-dimensional subcube. In the modified buddy system, partners along all possible dimensions are considered.*

can be improved even further by rotating the bit positions used to define the lowest $k-1$ levels in the tree [6].

Full recognition may 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 [200]. 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

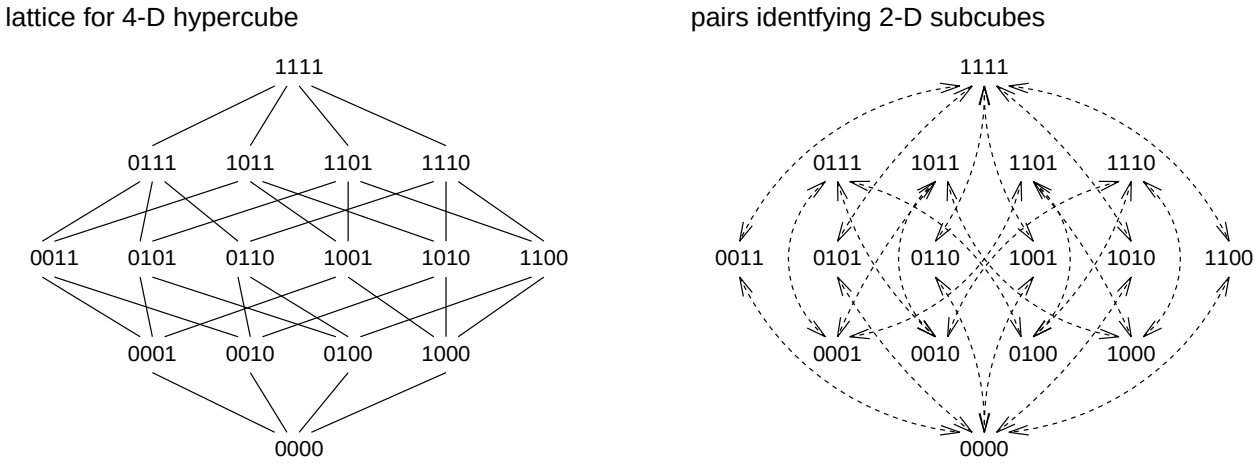



**Figure 6:** *Subcube identification using a lattice structure.*

| *description* | *recognition** | *complexity*† | *reference* |
|---|---|---|---|
| buddy system | $2^{n+1}-1$ | $O(n)$ | [83, 226] |
| gray code | $2^{n+2}-3$ | $O\left(2^n\right)$ | [83] |
| multiple gray codes | $3^n$ | $O\left(\binom{n}{n/2}2^n\right)$ | [83] |
| modified buddy system: check multiple combinations for creating subcubes | $n2^n+1$ | $O\left(k(n-k)2^n\right)$ | [6] |
| free lists for different subcube sizes | $3^n$ | $O\left(n2^{n-k}\right)$‡ | [212] |
| tree collapsing and transformations applied to the buddy system | $3^n$ | $O\left(\binom{n}{k}2^{n-k}\right)$ | [91] |
| gray code combined with two sets of free lists for the different sizes | $2^{n+2}-3$ | $O(n)$ | [111] |
| lattice | $3^n$ | $O\left(n^2\right)$ | [200] |
| graph of prime free subcubes (those not covered by larger free subcubes) and their intersections | $3^n$ | $O\left(\frac{3^{3n}}{n^2}\right)$§ | [429] |
| heuristic to maintain an approximation of a maximal set of (disjoint) free subcubes | $< 3^n$ | $O\left(n2^n\right)$# | [126] |
| graph of adjacent free subcubes with heuristic to coalesce them when released, using free lists to limit search space | $3^n$ | $O\left(2^{2n}2^{2^n}\right)$¶ | [443] |

* 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 [111]. 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 III:** *Listing of subcube recognition algorithms. The complexity measure is for the larger of the allocation and deallocation procedures, which is typically the deallocation.*

every $1 \leq i \leq n$. Using this structure, every two comparable elements in the lattice define a subcube comprising all the elements that are smaller than or equal to one of them, but larger than or equal to the other. The left part of Fig. 6 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.

Additional algorithms have also been proposed. Short descriptions and their properties are listed in Table III. 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 [226, 111]. Algorithms based on appropriate simple data structures, while not achieving full recognition, have a complexity of $O(n)$ [226, 111] (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 may become 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) [126]. 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 [429].

*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 [251, 250]. The Cray T3D, which has a 3D torus topology, also requires partitions to be powers of two. The DHC and DQT schemes for gang scheduling are based on a buddy system [143, 192] (DHC is described in Section 5.4). A similar approach has also been proposed in the context of a parallel reduction engine [190].

### 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 [356, 299]. 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 [294]. This approach is used on the IBM SP1. 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 virtual hierarchy of control. Such a structure forms the basis of the so called "wave scheduling" mechanism developed for the MICROS distributed operating system [421, 422]. 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 [389].

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 partitions are required to be rectangular sub-meshes. The most straightforward approach is to use 2-D versions of the well known first-fit or best-fit algorithms, in an analogy to memory allocation on uniprocessors [450, 90]. If the exact configuration cannot be found, embeddings can be used [265]. 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.

*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 [239, 425, 101, 148, 224]. 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 [131, 268]. 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 [232, 148, 164].

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.

*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 Polyp multiprocessor. In this machine, a set of PEs are connected via switches to a set of buses [271, 267]. 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 resource 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. 7 shows a $6 \times 4$ torus that is partitioned into three tori with dimensions $4 \times 4$, $2 \times 3$, and $2 \times 1$ (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 loading the communications network. An example is the IBM SP1, which uses a multistage network but allows any subset of PEs to be grouped into a partition [99].

### 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.

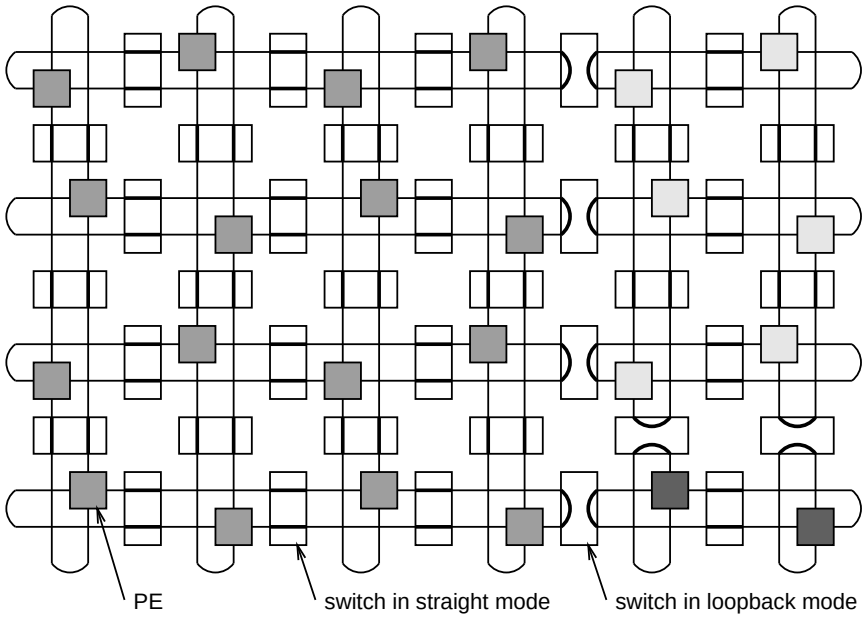
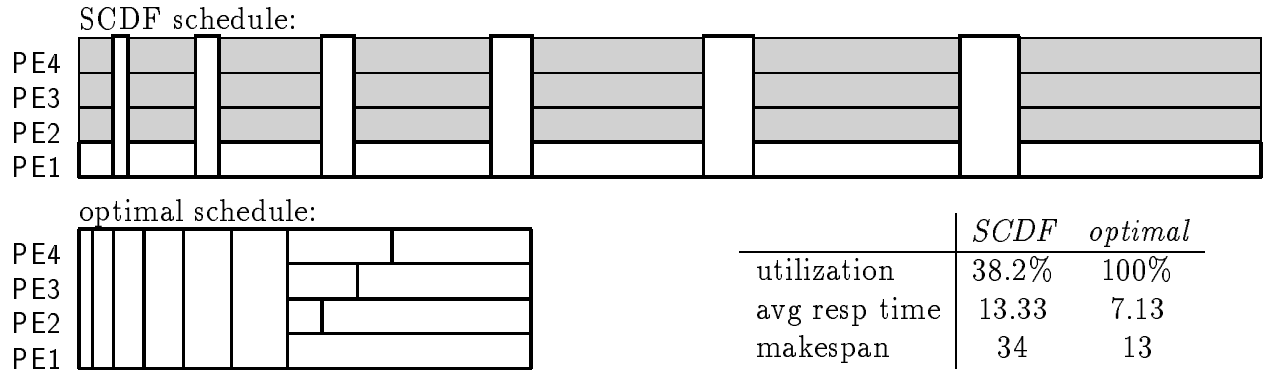



**Figure 7:** *Partitioning a torus requires switches that are capable of looping back to create small rings when partitioning larger ones.*

The favorite heuristic in uniprocessors is "shortest job first" [220, 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 [116], or through compile-time analysis [346, 29]). 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 [269], and the "largest job first" policy [82, 250, 451]. 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 [249, 226]. On the contrary, they may even cause large losses owing to fragmentation.

The problem with scheduling the smallest job first is rectified by a variant called "smallest cumulative demand first" [269, 248, 360]. 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 [226]. A counter example is shown in Fig. 8. It also suffers from the same disadvantage of shortest




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

**Figure 8:** *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.*

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 [248].

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 [93]. 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 [94]. 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 [250].

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 [226][5]. Moreover, systems using these schemes tend to saturate under lighter loads than FCFS. Using uniform requests for subcubes of different sizes in a hypercube as a concrete example, saturation may occur at loads as low as 40–50% of capacity [226].

A promising alternative that achieves much better performance is to group jobs together before they are scheduled, rather then scheduling them on an individual basis. This actually amounts to taking specific steps to reduce the fragmentation. A simple version of this approach, called "scan", has been suggested for hypercubes [226]. The idea is to use a distinct queue for each possible subcube size, and 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.

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

[5] The more optimistic results in [269, 249] 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.

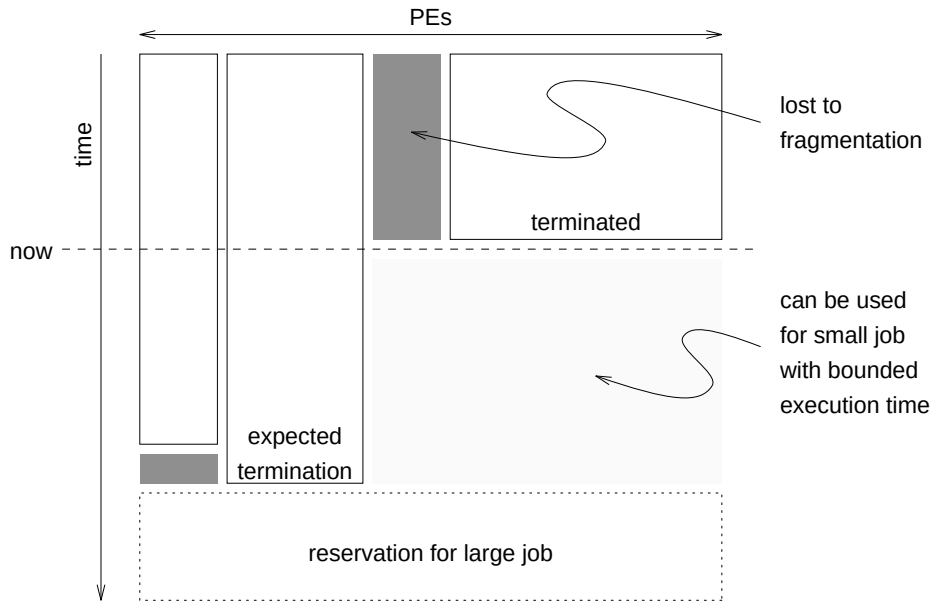



**Figure 9:** *Runtime bounds on executing jobs allow reservations to be made for large jobs while controlling the amount of resources lost to fragmentation.*

in PASM, where the different queues represent different potential partition sizes [412]. 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 [221]. 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 [198]. 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 [198, 110]. 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. 9).

NQS is now used at a large number of sites, with different parallel machines. It is described in Section 7.1.

### 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, called *adaptive*, 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 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 [62]. 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 [262]. 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 [217], which also compiles a global address space program for execution on a distributed memory system, and in many message-passing systems, including EUI [28], p4 [70], and PARMACS [71].

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 [428], or else it may be done by the users themselves in a first-come-first-serve manner [209, 363, 112, 198]. 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 [164, 447, 360]. 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. 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 [249, 248]. 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 [358]. 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.

Four other algorithms have been devised that are not derived directly from the equipartition ideal. The first algorithm simply limits the maximal partition size that any single job can obtain [339, 444]. 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 [87].

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 [339]. One reason is that this algorithm tends not to utilize all the PEs except under very heavy loads.

Another algorithm uses greedy allocation: if an arriving job requests more PEs than are available, it is allocated all the available PEs [444]. This avoids wasting PEs in anticipation of future arrivals, but risks long queueing delays if such arrivals do indeed occur.

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 [339]. 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

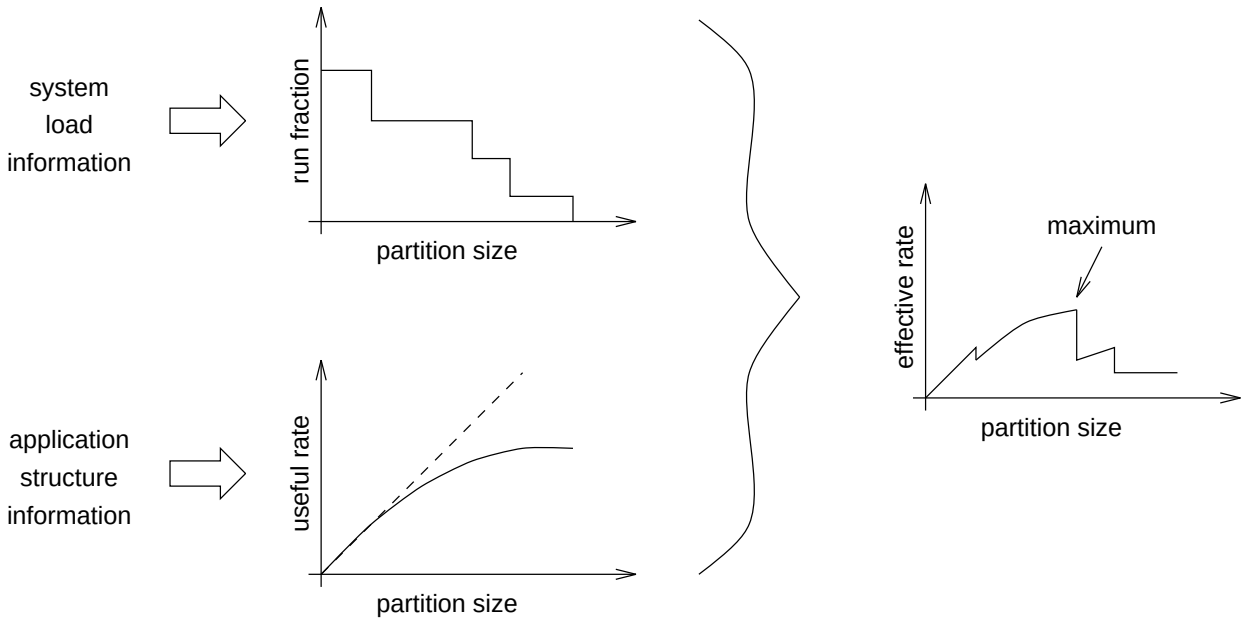



**Figure 10:** *Combining system load information with a model of application efficiency using different partition sizes provides the optimal size, which will maximize the effective computation rate.*

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 other two, meaning that on average it performs better for widely different load conditions and for different workload characteristics.

*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. 10 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 [79, 22]. The size that provides the maximum effective rate is then used.

A similar idea can be applied to systems that use partitioning without time slicing. In this case, the application supplies knowledge about its *execution signature*, i.e. how much time it would require on different partition sizes [308, 268]. 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) [361, 235]. 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.

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.

*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 [310]. The obvious remedy is to consider memory requirements when allocating PEs. This applies both to primary memory and to secondary storage.

*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 [148]), 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 [415, 414, 430, 413].

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 [409, 447, 268, 278]. For example, when an application enters a sequential phase of computation, its PEs can be reassigned to another application [447, 424]. 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 [248]. This can happen in small installations, but is not expected to be a problem with large parallel machines.

An important pre-requisite for such 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 unapplicable 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 that were reported in the literature rely on a centralized server that maintains load information. Two different schemes have been proposed. In the two-level scheduler designed at the University of Washington, the server is activated only when the system load changes; it then redistributes the PEs as appropriate [447, 278]. 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" scheduler designed in Stanford University 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 [409]. Eventually the total number of virtual PEs becomes equal to the number of physical PEs, thus preventing any overhead due to context switching. 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 this implementation 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 [12], which are a new mechanism for thread manipulation and communication between the kernel and the user-space thread package. This 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. Practically the only programming model that meets this requirement is the *workpile* model, a.k.a. "farming", "problem heap", "task queue", "agenda parallelism", or "self scheduling" [285, 95, 74, 394]. In this model the application is structured as multiple independent chores, which are kept in a workpile. A number of worker threads running on distinct PEs pick chores from the workpile in an undefined order, and execute them. Additional workers may be added at any time, and existing workers may be removed whenever they finish one chore and before they start another. 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 [160, 4] and futures in Multilisp [177, 222]. In WorkCrews new chores are created in a lazy fashion, so as to reduce overhead if they end up being executed locally [423]. Alternatively, chores can be created by a compiler [324, 323]. In many cases the independent chores are directly referred to as virtual PEs, with the mapping to physical PEs hidden in the runtime system [409]. Examples include the multiprocessor implementation of concurrent C [92, 157] and various thread packages [97, 434, 44, 138, 77]. The Psyche operating system on the BBN Butterfly provides a similar notion of virtual PEs at the operating system interface [354]. 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 [48]. 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 same chore are required [402]. 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 [12] and the Psyche system [352, 274], and are described in more detail in Section 4.4.2.

Finally, it should be noted that using a workpile model is not a pre-requisite for using

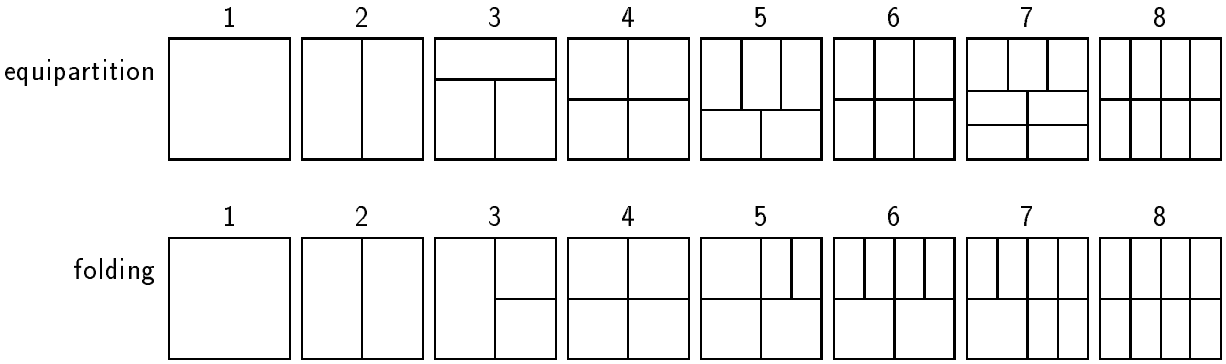



**Figure 11:** *Conceptual comparison of equipartitioning and folding. In reality, equipartitioning will create non-contiguous partitions so as to reduce the number of PE preemptions.*

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 [291, 125]. 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) [15, 262]. As this involves considerable overhead, it is imperative that the frequency of reconfigurations be kept low in practice. Occasionally the price of reconfiguration might even outweigh the benefits of changing the PE allocation [308, 125].

*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. 11).

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 [291, 21]. 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 [290].

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 [440, 193].

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). Runing 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 [279]. A possible solution is to use a *folding* policy rather than an equipartition policy (Fig. 11). This policy assumes that the total number of PEs is a power of two, and that all applications are designed to execute optimally on the full machine. When a new job arrives, the largest current partition is folded in half, thus freeing half of its PEs for the new job. The result is that all jobs in the system have partition sizes that are either of two adjacent powers of two. Such partition sizes provide the same load balancing properties and communication locality as the full machine, so jobs can be expected to achieve high efficiency [279]. 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.

Finally, 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 [310]. This situation is improved if minimal partition sizes are imposed [290].

### 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 [269, 249]. 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.

This approach 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 [289], Amoeba [288, 393], and plan 9 from AT&T Bell Labs [320].

A similar mechanism, called Automatic Self Allocating Processors (ASAP), is implemented in hardware in the Convex C2 system. This machine has 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 [203, 407, sect. 3.4].

# 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 describes 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 [13, 273]. 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 [448]. Other example systems that use this approach include distributed memory systems like NX/2 on the iPSC/2 [318], SIMPLEX on the nCUBE [229], the MOOSE experimental system [344], the original Cosmic Cube [357], and others [109]. The Sylvan system went so far as to add a dedicated kernel-processor to each node [67]. One of the responsibilities of this processor was to perform local scheduling of the node's user-processor and vector unit. Local queues have also been used in shared memory systems, such as StarOS on CM* [158, chap. 6], Chrysalis [243] and Psyche [352, 354] on the BBN butterfly, EMBOS on Elxsi systems [345, 300], and the EMMA2 system [18].

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 [186, 407, 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 [435, 277]. Operating systems for such machines make use of this feature and support multitasking on each node (e.g. Trillium [68] and Helios [309, 182]).

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 [377, 204, 219, 195]. 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 [10, 8]. The MIT Alewife takes a slightly different approach: threads are switched only upon a remote access or a synchronization failure, not on every instruction [2, 3]. In all these architectures the role of the operating system is limited to mapping multiple threads to each PE.

A third class of systems with hardware scheduling that amounts to local queues is represented by the MIT J Machine. This is a message-driven architecture, where threads are created and scheduled in response to incoming messages [107, 106]. 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 [356]. In SPMD programming, this data partitioning is explicit as part of program [151]. 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 [311, 312, 313], Parallel Pascal [336], and PASM C [234]. Implementations exist not only on SIMD array processors such as the Connection Machine CM-2 [189, 5], but also on MIMD hypercubes [179].

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 [128, 367]. The typical measure of load, which is the length of the run queue, is also based on research in distributed systems [64, 238]. There is some controversy in the literature on whether balanced placement of new threads is enough [129], or maybe migration of active threads (also called "preemptive load balancing")

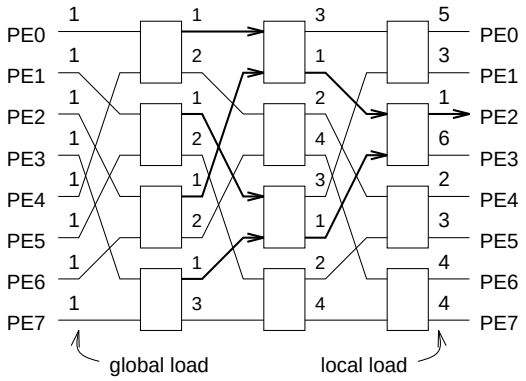



**Figure 12:** *Dissemination of load information and automatic mapping in Flagship.*

is required as well [227]. 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 [145].

### 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 [213]. Obviously, this scheme has a bad worst-case behavior, because it is possible that the most highly loaded PE be chosen [128, 190]. 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 [174]. An alternative approach is to first probe a limited number of nodes at random, and then choose the best one [283].

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

dissemination of load information and mapping are both done in hardware. This approach has been taken in the Flagship graph-reduction machine and in the Heidelberg Polyp. The PEs in Flagship are connected by a Delta multistage network, which is used to propagate load information [392, 432, 433]. 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. 12).

In Polyp, a special bus connects all the PEs and contains global information about the highest priority waiting task and lowest priority running task [270, 267]. 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.

Another approach is to distribute the load using self scheduling from a global queue, and then use local queues 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.

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 [143, 145], in DQT [192], in NETRA [88], and for hypercubes [109, 24]. Actually, the same structure has also been proposed for maintaining and distributing the load information required to map single threads as well, in AMPS [211] and in APERM-2 [190].

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 [71, 427]. HPF, Dataparallel C, and Pandore only support a more restrictive multidimensional mesh topology [262, 179, 15]. Alternatively, the system can make an educated guess about communication patterns; for example, it is reasonable to assume that when one thread spawns another thread, the two will communicate [41]. 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 [298].

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 [166]. 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 [89, 307]. 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 [347, 89, 261, 162, 298].

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 [54, 199, 384]. 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 [426].

### 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. Most of the literature on this subject relates to distributed systems, not parallel ones (see [431, 367] 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.

Important features of a load balancing algorithm is that it distribute load quickly and that it be stable [64, 75, 380]. Quick load distribution is achieved by non-local communication to disseminate information about load conditions [257]. 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 [296], or by limiting the number of times that each specific thread can migrate.

Two load balancing schemes that have been applied to parallel systems 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 [102, 53]. The analogy is the diffusion of molecules across membranes, which is driven by the difference in concentrations. This scheme is used in MuNet [178] and Alfalfa [165]. The effectiveness of diffusion depends on the network bandwidth, and it is therefore inefficient for low-dimensionality meshes [387]. 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 [102, 120]. This was shown to be more efficient than diffusion with multiple neighbors at once [102, 436].

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 [257]. This scheme is used in Rediflow [210].

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. One way to achieve this is by defining thresholds to distinguish between lightly loaded and heavily loaded PEs [367], but nodes still need to collect data to establish correct threshold values. Another solution 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 [341]. 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 [362, 223]. One of them is used in the MOSIX system [32, 33, 31]. 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 [448][6], 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

[6]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.

unbalanced nodes [388]. Another approach is to use a hierarchical control structure, like the ones used for mapping based on load conditions.

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 [321, 343, 297], or load balancing within the application to compensate for data-dependent changes in the load at runtime [437, 120]. 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*[7]. In effect, the workload is distributed evenly across all the PEs. The disadvantages are possible contention for the queue 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 [337, 272, 273].

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 [326, 194]. 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.

*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 [26]. A lockable shared queue is used in the Hydra system on C.mmp [441, chap. 12], in the Dynix system on the Sequent Balance [399], in Mach [48, 51], in Irix on Silicon Graphics multiprocessor workstations [256], and in the DASH system which is based on Irix [246]. It is

[7] 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 [128, 228]. The term "load balancing" is only meaningful when PEs have individual loads, as when local queues are used.

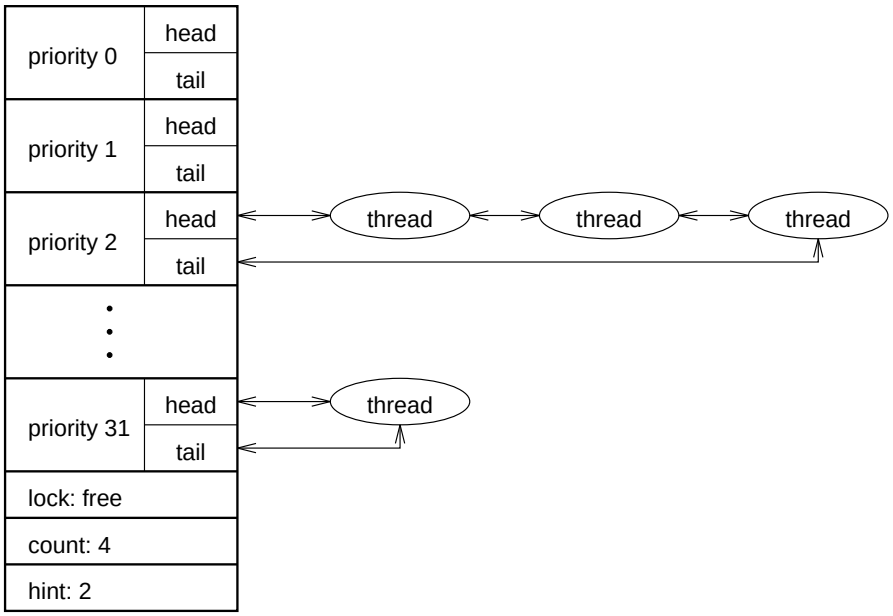



**Figure 13:** *Priority queue structure in Mach (adapted from [48]). The hint field indicates the highest priority currently in use.*

also used in thread packages such as Presto [44, 138], and in microtasking on multiprocessor Cray supercomputers [155].

Much progress has been made lately regarding the efficient implementation of locks, especially by way of exploiting local caches with hardware coherence [340, 11, 167, 173, 280]. 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, 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 [48] (Fig. 13). 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 [202, 333].

*Implementation with fetch-and-add*

The problem with locking the queue is that the queue becomes a serial bottleneck [45, 184]. Denote the average time that the lock is held by $\tau$, and the scheduling time quantum by $T$. Each PE only holds the lock for $\tau/T$ of the time. But if there are more than $T/\tau$ 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 [404].

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 [172, 169]. Actually, a fetch-and-increment operation is also enough [154]. 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. 14. 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 [172]. 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 [132, 133]. The implementation of fetch-and-add is a basic feature in the design of the NYU Ultracomputer hardware [171, 118, 7, 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 [317, 244].

*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 [270]. 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.

```
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 14:** *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.*

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[8] [48, 47] and some other systems [66]. The partitioning is based on the concept of a *processor set* (other systems often employ the term "processor pool" [66]). 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. 13). 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 [449]. In addition, this scheme reduces the number of PEs sharing a queue (relative to a single global queue), thereby reducing contention for the queue [295].

*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 [48].

[8] 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.

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[9] relative to the priority of their parent. This ensures that nested spawns are executed in a depth-first order [133]. 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 [42].

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 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* [424, 115, 27, 378]. 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 [175, 405, 424]. However, using cache affinity as a guideline for scheduling can only improve performance, so there is no reason not to use it.

A simple implementation of affinity scheduling uses local modifications to threads' priorities [405]. Each thread has a global priority that depends on its accumulated runtime, like processes in a Unix system. However, when a given PE scans the queue and looks for the thread with the highest priority, it boosts the priorities of those threads that used this PE the last time they ran. The last thread to run on the PE is given a larger priority boost. This increases the effective scheduling quantum, by increasing the probability that a thread 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 [249]. 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

[9] 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 IV:** *Comparison of local and global queues.*

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 [322].

### 4.3 Combination of Local and Global Queues

The main features of local and global queues are summarized in Table IV. 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 [139]. In this system, a global queue is used for work distribution, and then local queues are used for the actual execution[10]. The mapping of new threads is done by a system `get_work` thread that competes with local threads, so lightly loaded PEs get more new threads than PEs that already have a high load [341]. Once mapped by `get_work`, threads do not migrate. A similar scheme is used in the KSR1 system, as described in Section 7.6. 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 [137, 136]. In both 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 [368]. A similar optimization exists in many other systems, where threads are run to completion and then their context is

[10] 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.

reused for the next one. Examples include Scheduler Activations [12], Continuations [121], and the Presto thread package [44, 138].

Another system that uses both local queues and a global queue is the variant of the Mach system developed for the IBM RP3 [63]. 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 [86]. 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 [394, 326, 327, 155]. They are scheduled in FIFO order, with no preemption. Actually "scheduled" might be an overstate-

ment: 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 [231]. 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, or if the workers do not all start at the same time[11]. 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 [231].

A number of schemes that do so have been suggested (see Fig. 15). The formula suggested by *guided self scheduling* is to take $1/P$ of the chores *remaining* in the queue each time [326]. 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 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 [264]. 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 [194]. 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

[11] 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 [254].

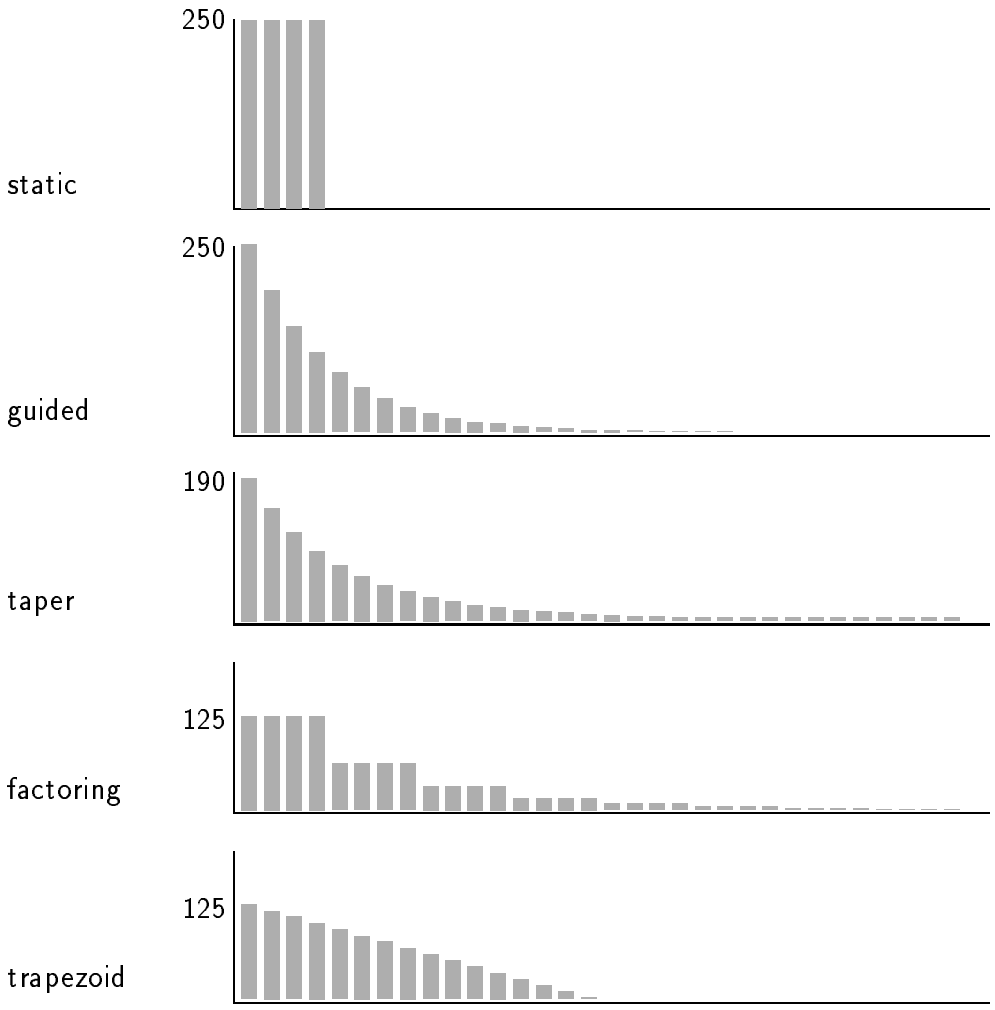



**Figure 15:** *The chunk sizes created by different self-scheduling schemes, for 1000 chores and $P = 4$ (data partly from [194]).*

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.

The chunk sizes in guided self scheduling and in factoring decrease exponentially. In *trapezoid self scheduling* they decrease linearly, leading to simpler implementation [416]. 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.

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 [273]. 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 [253].

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 [402], by Chameleon [9], by the Chores system [130], and by the MAXI system on the Makbilan [368].

### 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) [278]. 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 [133, 185, 446]. 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 [48, 175]. 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 [185].

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 [274, 352]. 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 [353]. 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 [12]. 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. 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 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.

A different approach altogether is to forgo all information exchange, and strive to approximate the optimal efficient behavior. For example, with cooperative threads, there is an option to create a new kernel thread whenever an existing one blocks [197]. 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[12] [368]. 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. We start with a definition of gang scheduling and an exposition of the reasons for using it. Then we inves-

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

tigate the implementation of coordinated context switching across multiple PEs. Finally, we describe systems that combine gang scheduling with partitioning in various ways.

### 5.1 Definition and Motivation

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

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

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

Gang scheduling leads to two desirable properties. First, gang scheduling promotes efficient fine-grain interactions among the threads in a gang, based on the fact that they are executing simultaneously. For example, it is possible to use busy waiting for synchronization, without fear of waiting for a thread that is not running [302, 144]. In addition, asynchronous message passing can be used without the risk of buffer overflow, and hardware communication devices can be accessed directly in user mode without need for protection mechanisms. Moreover, a one-to-one mapping also allows threads to be associated with data structures in local memory. Indeed, studies that compared gang scheduling with other scheduling schemes have concluded that gang scheduling is a relatively good policy [260, 355, 249, 175, 100].

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. 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, might reduce performance, and is not always needed. These and other arguments are discussed in Section 6. Nevertheless, gang scheduling is an appealing policy, as reflected by the fact that it is provided on a number of commercial platforms, such as the CM-5, Meiko CS-2, and Intel Paragon.

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 [205].

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. One possibility is to use a set of predefined partitions; such systems are described in Section 5.3. Alternatively, it is possible to have time-slicing across the machine, and do the partitioning on the fly for each context switch. This increases the flexibility of how the machine is shared; for example, it allows applications that require all the PEs to co-exist with applications that require small sets of PEs. Systems based on this approach are described in Section 5.4.

*What is a gang?*

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

However, gangs comprising just part of the threads of a given job can also be used. This allows the number of threads in the job to exceed the number of PEs in the system. The criterion for grouping is simple: threads that interact at fine granularity should be scheduled together, and therefore they should be grouped into a gang[13]. This can be done based on runtime observation of interactions among threads [142]. 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 [42], `PAR` in Occam [196], and the `spawn` system call in Symunix [134]. Thus, when a set of threads are spawned together by a parallel construct, they are assumed to constitute a gang [302, 143].

*Coscheduling and family scheduling*

Two other terms that are often mentioned are coscheduling and family scheduling. *Coscheduling*[14] was originally defined by Ousterhout to describe systems where the operating system attempts to schedule a set of threads simultaneously on distinct PEs, as in gang scheduling, but if it cannot then it resorts to scheduling only a subset of the threads simultaneously [302]. 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

[13]Ousterhout used the term "process working set" [302], based on the analogy with the working set of memory pages that must be simultaneously resident in primary memory for efficient computing.

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

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 [100]. Therefore it is not clear that attempting to schedule partial gangs when a full gang cannot be scheduled is worth the effort. 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* [302].

*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 [63]. 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.

### 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. This controller has the means to cause 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 [383]. Another is to use a software broadcast interrupt. 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 [144].

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 the communication state together with the computation state [252]. 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 [46, 163, 1].

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. 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 [36] and the Intel Paragon [319]. Note that it requires access to the descheduled job's address space, and therefore 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.

The third approach is to drain the network as part of the context switch operation. 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 Connection Machine CM-5 [245] and the K2 [383].

### 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 [383], as embodied in the synchronous nearest-neighbor user channels provided in hardware by the K2 architecture [17]. This is a relatively small machine (16 PEs organized as a $4 \times 4$ 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

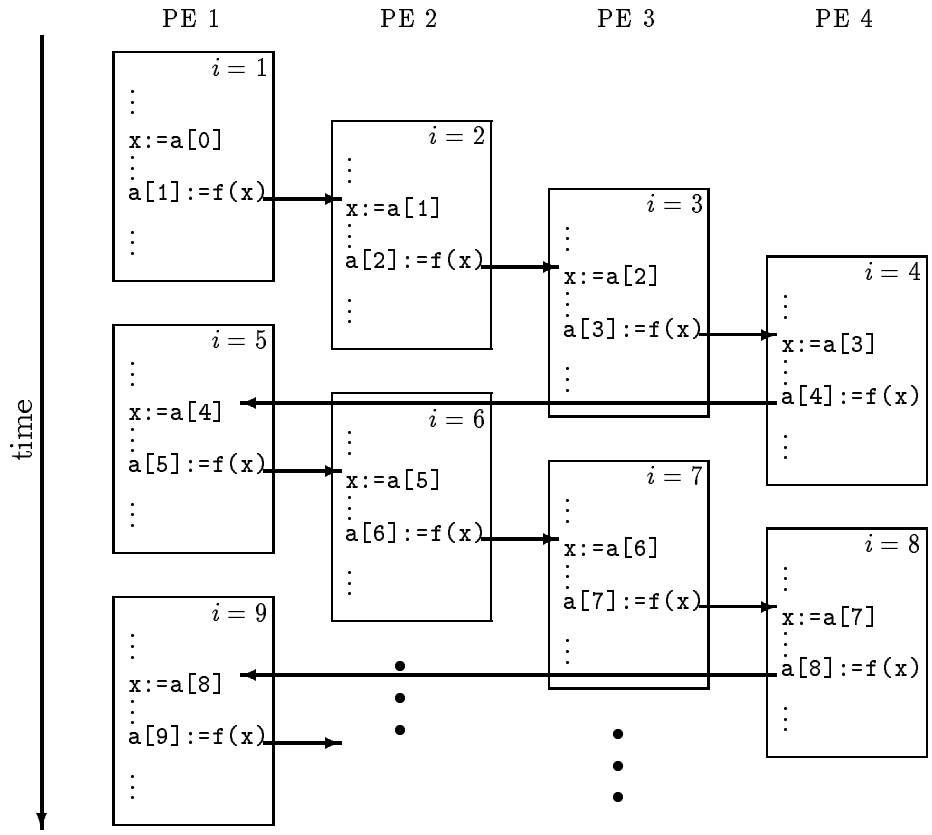



**Figure 16:** *Skewed execution of iterations from a doacross loop. The interdependencies between iterations imply fine-grain synchronization and a limit on the degree of parallelism (adapted from [396]).*

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") [396]. 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 [103, 325]. 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. 16) [396].

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 [156, 233, 218]. Each Alliant machine is called a cluster, and such clusters are the unit of PE allocation. The scheduling is done at two levels: allocation of work to clusters, and scheduling on each cluster. The allocation is done from a global queue of tasks, where each task specifies the state of a whole cluster (i.e. 8 Alliant PEs). A server process executes periodically on each cluster, and if it finds a new task on the

global queue, it copies the task to the local cluster [137, 136]. 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 [292].

A mechanism for gang scheduling combined with partitioning of the machine has been implemented in the Connection Machine CM-5 [245]. 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 communication. When a partition of PEs switches from one set of threads to another, the partition of the network connecting them has to be flushed. This is done by putting the network in "all fall down" mode[15] [245]. 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 [205]. 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.

[15] 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. 21.

Then at each context switch the next set of matching applications is scheduled. For example, this approach is taken in the Intel Paragon.

*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 [302]. The simplest is the matrix algorithm, which was implemented in the Medusa operating system on CM* [303], in the Meiko CS-2 operating system, in the Intel Paragon operating system, in the gang-scheduler used for the BBN Butterfly at Lawrence Livermore National Lab [168], in the experimental gang scheduling runtime library of MAXI, the Makbilan operating system [144], and in DQT [192]. 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. 17.

The other two algorithms developed by Ousterhout, which were not implemented, are based on a sliding window metaphor [302]. Threads are arranged in one long sequence, with the understanding that a thread in location $i$ is mapped to PE number $i \bmod P$ (where $P$ is the number of PEs in the system). In the "continuous" algorithm, threads belonging to the same gang must be placed within a span of $P$ places in this sequence. In the "undivided" algorithm, threads from the same gang must occupy adjacent places. Scheduling is performed by sliding a window of width $P$ across the sequence, and scheduling those threads that are currently in the window. At the end of each scheduling time quantum, the window is advanced until the leftmost thread in the window is the leftmost thread of a gang that had not been gang-scheduled yet (it is possible that the thread itself had been scheduled in the previous window position, but its whole gang was not). When all gangs have been scheduled, a new sweep is started. These algorithms — and especially the undivided algorithm — have a slightly larger coscheduling effectiveness than the matrix algorithm [302]. In other words,

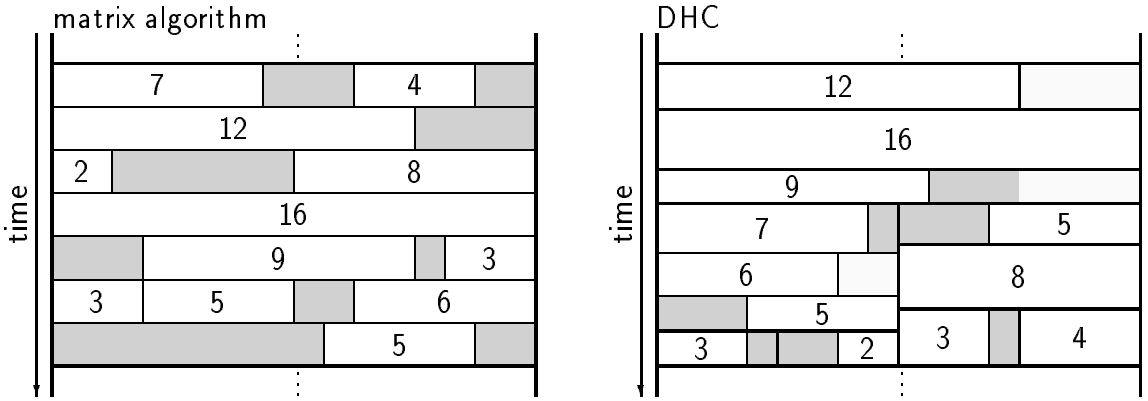



**Figure 17:** *Example of a scheduling round using the matrix algorithm and DHC. Gray shading indicates fragmentation.*

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 only happens 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" scheme [143, 145].

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. 18). 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. 17.

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.

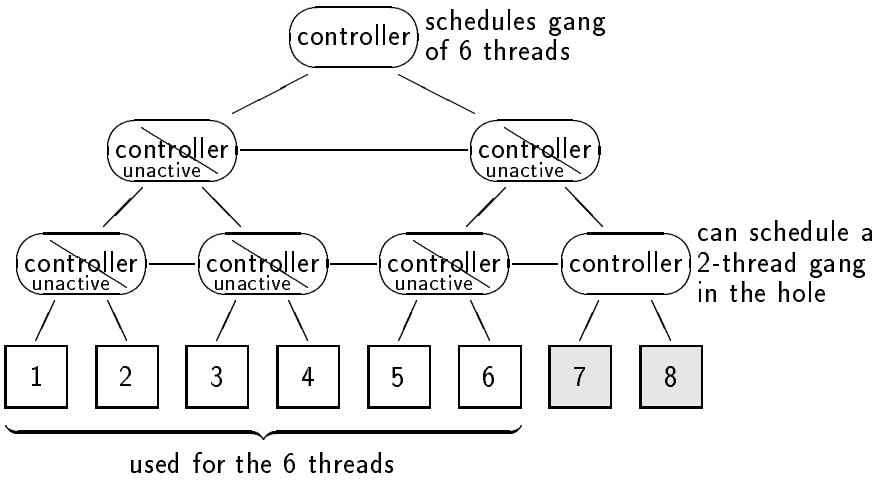



**Figure 18:** *Distributed Hierarchical Control arranges PEs in a buddy system. Small gangs are mapped to holes left by large ones, and execute at the same time with them.*

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. Fig. 18 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. 17. 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 [145]. The time quantums in Fig. 17 are computed based on the number of threads in each gang.

### 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 [201, 166, 418, 417]. However, the model employed was that the scheduled jobs each execute on a single PE. There has been very little theoretical work on scheduling where the jobs themselves are parallel, i.e. they 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. In the scheduling formulation, the input is a set of tasks, 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 [122, 126]. However, preemptive scheduling (as is the case for gang scheduling) admits to polynomial-time algorithms [52, 451, 81, 364, 122]. 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 [52] (this is even simpler for hypercubes, where the partitions are subcubes [81, 364]). 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 [293].

## 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 [260, 269, 249, 175, 100, 144, 290]. This section summarizes these results and adds many additional considerations, such as the support for different programming models and the influence on user satisfaction. These considerations are divided into four areas: the interaction between scheduling and the application, the impact on system performance, the support for individual users in the context of a multiuser system, and implementation issues.

### 6.1 Interaction with Applications

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 applica-

tions, 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 [175, 144].

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 [302, 144]. 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 [302]. 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 [207]. The competitive factor can be reduced if the distribution of waiting times is known [255].

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.

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 scheduling is very inexpensive, as in the Denelcor HEP [204, 219]. 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. The other is when interactions are extremely expensive, as in early message-passing systems. 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 [406].

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 [237, 245].

*Support for memory locality*

Programming for locality is widely recognized as an important paradigm. Specifically, locality of reference reduces interprocessor communication in multicomputers [337] and improves cache performance in multiprocessors [188, 240]. 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 [337, 272, 273]. 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 [284, 175, 424]. 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 [424, 115, 27, 378, 405]. It has even been proposed that affinity hints be included in the programming language [78].

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[16] 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 [47]: 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

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

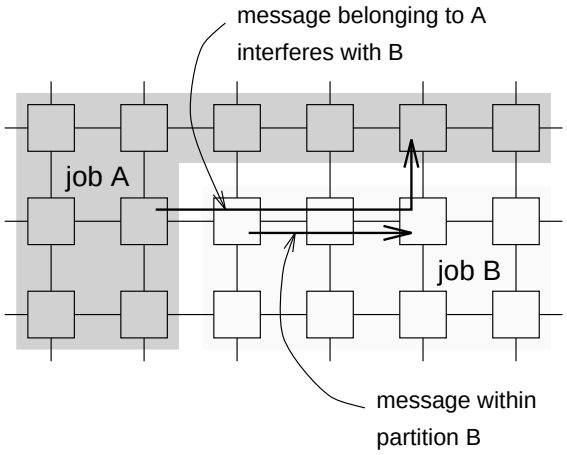



**Figure 19:** *Due to the predefined routing scheme, concave partitions on the Paragon may cause interference.*

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 [356, 105, 104].

Even if communication per se does not depend on distance, using distant PEs can still degrade performance due to increased contention: messages that traverse a larger distance need to pass through more links, so the overall load on the network increases. This is especially troublesome if it induces a lack of independence, that is if messages belonging to one job cause a degradation in the network performance of another. For example, this has been known to happen on the Intel Paragon, when one job is allocated a set of PEs that form a concave group, and another job runs in the middle, effectively surrounded by PEs belonging to the first job. Traffic belonging to the first job then passes through the links in the partition belonging to the second job (Fig. 19).

Locality can also be affected by a mismatch between the application's programming model and the actual architecture, as when embeddings are used [265]. 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 [131, 425, 101, 148, 100, 224]. Dynamic partitioning, on the other hand, only gives a job more PEs if there is nothing better to do with them [175, 290, 279]. 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 [379].

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 [181, 385]. 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 [310]. Such effects can be reduced by establishing a minimal partition size [290].

*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 [220, 314].

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 [98]. 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 [389, 143, 420, 322].

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 [113, 140]. 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[17] and gang scheduling.

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 [144, 359].

*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 [251, 250, 146].

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 [409, 447, 100]. 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 [84, 80], 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 subcube allocation in hypercubes. The problem with this approach is that migration requires a potentially large context to

[17] 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.

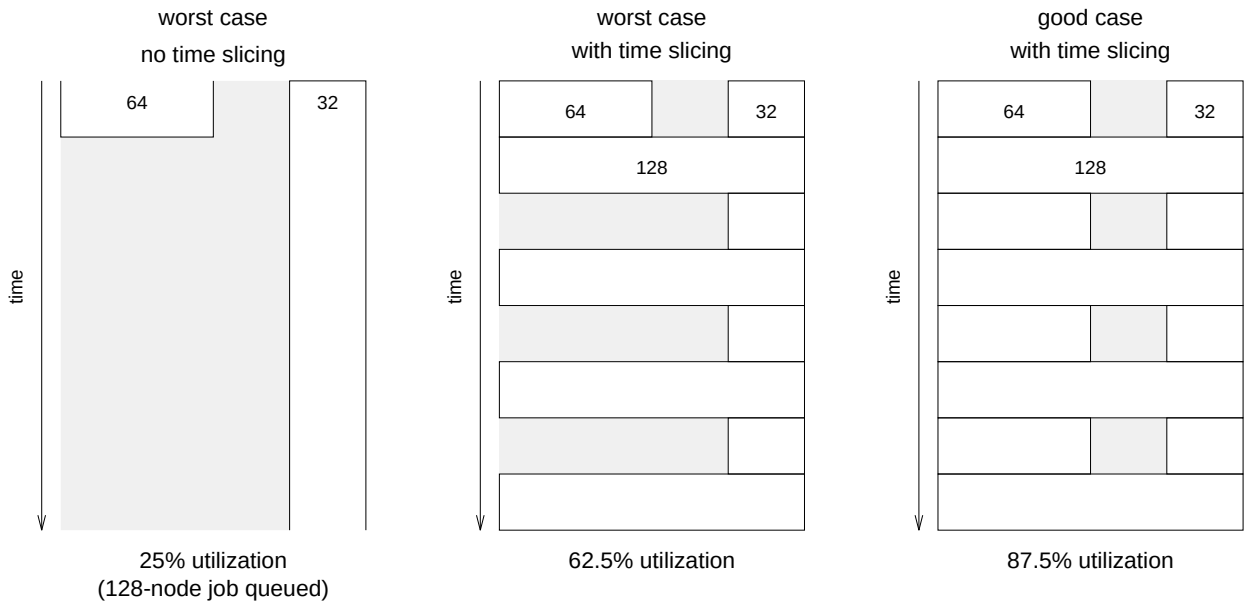



**Figure 20:** *Example of how the flexibility afforded by time slicing can increase system utilization.*

be copied, which loads communication resources [400]. Also, this does not solve internal fragmentation.

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 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. 20). 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 [145, 146]. 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.

*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 scheduling [119]. This 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 [409]. Partitioning potentially also reduces paging, because all the memory is allocated to the same job for the duration of its execution. This issue is elaborated below.

*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 [384]. 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 [419, 276]. Actually this concept had been implemented already long before, in the multithreaded architecture of the Denelcor HEP [376, 204, 219]. While that machine is no longer available today, its legacy continues in the Tera machine [10, 72, 8].

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) [220, 147]. 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 [123]. 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 [166, 89, 261, 162], 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 [208]. 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 [23].

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 [144]. 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.

*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 [370, 369, 258], in the fat tree network used in the Connection Machine CM-5 [245], 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 [39, 112, 241]. It may even be possible to reboot a partition without any effect on jobs running in other partitions [96].

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 [317]. This may block memory accesses from threads in other applications. The severity of this problem and possible solutions are the subject of considerable controversy [7, 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[18] consistency problem [50, 338, 395]. 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

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

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 [245].

*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 [329, 38]. 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 [248].

As massively parallel supercomputers become more common, multiuser support becomes more important [59, 112]. Likewise, user frustration with poor access or bad response becomes an issue [176]. 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 [410]. 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 [209]. 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.

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 [112, 209]. 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 [428, 281]. 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 user to kill unauthorized jobs that interfere with his work [198]. 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 [47]. 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 [205].

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 [382, 42, 393]. 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 [249]. 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 [168]: 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 [60, 228, 269, 145]. 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[19].

[19]It is often said that users might also create spurious threads that do not perform any useful computation.

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 [183].

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 [183]. 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.

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 [145]. 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 constrained 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

However, this would not lead to any benefits. If the scheduler is fair at the thread level, the spurious threads will just compete with all the other threads in the system, including the threads that actually do the computation in the same job. This will degrade the service to all jobs, including the one that created the spurious threads.

processes. The solution is to keep each process's working set of pages in the main memory [114]. 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 [55, 49, 348, 242]. 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 has received only scant attention [310, 65, 279, 383]. 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 [310]. 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 [221]. While memory availability is indeed growing at an amazing rate, so are processor speeds and user requirements. Therefore this approach forces users to take memory availability into account when designing programs, and might limit the computations that can be performed [374][20]. Providing virtual memory is increasingly recognized as a necessity [371].

Systems that use preemption, on the other hand, cannot afford not to support memory management as well. When PEs are shared among a number of jobs, so is the memory. The more jobs there are, the harder it is to satisfy their memory requirements at once. This draws the line between short-term scheduling and long-term scheduling: long term scheduling is involved in deciding which jobs will have some memory allocated to them, and which will be swapped out altogether. Short term scheduling is involved in the decision as to which threads will run at a given instant, out of the threads belonging to the memory-resident jobs. While swapping is a basic feature in most uniprocessor systems (e.g. Unix [25, sect. 6.1 and 9.1]), very little work has been done in this respect in the context of parallel systems [383, 65, 279].

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 [411]. 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

[20] Interestingly, memory availability has also been proposed as the metric for scaling the problem size when analyzing speedup bounds [438, 390].

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. If more than 2 jobs are being time sliced, they will have more memory available to them.

*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 [370]. 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 [373, 372], or gang scheduling, as on the Connection Machine CM-2 [410, 205]. 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. 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 [383]. However, this only makes sense if the multi-context-switch overhead is smaller than the page-fault service time [65].

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 [204, 219], Tera [10], and Alewife [2]. 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 typically requires 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 [245, 144, 383].

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 [245]. Of course, it is also necessary to restart these communication operations when the application is scheduled again [252].

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 [252]. 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.

*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 I 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 Catering for Many Masters

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.

## 7.1 The Network Queueing System (NQS)

The Network Queueing System (NQS) was developed at NASA Ames to help support efficient use of their Cray 2 and Cray Y-MP multiprocessor supercomputers [221]. It has since been adopted by many installations, as well as by the vendors of various parallel machines.

| *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 V:** *Example of NQS queues used on the 128-node iPSC/860 system at NASA Ames [141].*

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. 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.

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 replace the specification of memory requirements with 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 V, 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 V 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. 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.

### 7.2 The Lawrence Livermore Gang Scheduler

Lawrence Livermore National Lab conducted a three-year Massively Parallel Computing Initiative from October 1989 to October 1992, to study the use of microprocessor-based parallel machines as a replacement for traditional supercomputers. The study was application driven, so a major goal was to parallelize various applications and measure their performance on a substantial parallel machine. The machine acquired for this study was a 126-node BBN TC2000. This is a shared memory machine, where the processors are connected to the shared memory modules via a multistage network. Each processor is a Motorola 88000 running at 20 MHz.

The native operating system on the TC2000, which is called nX, only supports space slicing. The nodes are divided into two: a public cluster, which is used for login, editing, etc., and a free cluster of nodes that can be allocated to parallel jobs. A local gang scheduler was developed at LLNL 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 [168]. The gang scheduler works by acquiring all the free processors from nX when the system 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 [168]. The default class is "interactive". These jobs are gang-scheduled with a time quantum of 10 seconds. The time quantum can be adjusted in the range of 5 to 20 seconds, in the interest of providing a fair share of the resources to different users. Thus users that run multiple jobs concurrently will get a shorter quantum for each job.

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

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

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

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[21] — the one chosen by the gang scheduler. A special timing function is provided that gives the time that the program actually ran, after accounting for unscheduled time. However, this is rounded to the nearest second at each time slice, so it is not very accurate, especially for interactive jobs.

It should be noted that the gang scheduler could be replaced on the fly, allowing system administrators to tune various parameters (e.g. time slices and benchmark time limits) as needed. In addition to the gang scheduler itself, an interactive tool called the "gangster" was developed. This allowed users to check the status of their jobs and interact with them. For example, if an interactive job was demoted to production status because it took too long, the user could bump it up to interactive status again. It was also possible to investigate the load on the machine, and instruct the gang scheduler to use certain nodes in preference over others. This runtime control interface was very useful in enabling users to interact with the system and affect its behavior.

While the original project has ended, the BBN machine is still in active use. Research on parallel processing at LLNL is also continuing with a newly acquired Cray T3D. The gang scheduler is in the process of being ported to the T3D.

### 7.3 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 [401, 304]. 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. 21), and a control network, configured as a binary tree that matches the data network's fat tree [245]. The data network is used for data transfer, while the control network can be used for special operations such as barrier synchronization, broadcast, and reduction. The machine has been available since the end of 1992. Standard configurations range from 32 to 16384 nodes; the largest installation has 1024 nodes. In addition, there are a number of control processors. The control processors run the Unix-compatible CMost operating system, while processing nodes run a local microkernel.

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

[21] Programs using Zipcode [375] 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.

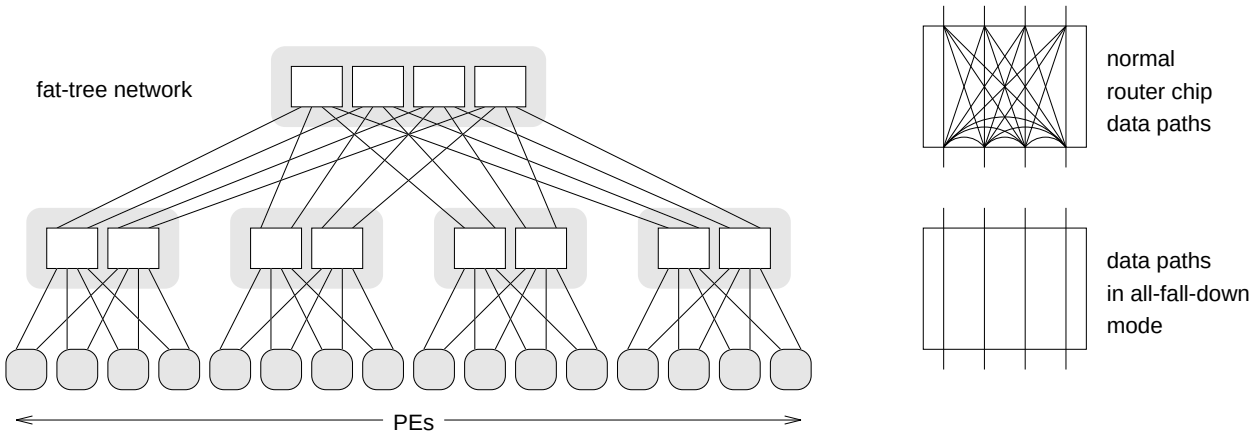



**Figure 21:** *The CM-5 fat-tree data network is implemented as a multistage network [245]. To flush it, router chips are put in "all-fall-down" mode.*

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.

I/O devices are also contained in separate partitions. For example, it is possible to have two file systems on two separate partitions, each with an I/O control processor and multiple disk storage nodes. These file systems are mounted on the processing partitions' control processors, as are additional file systems from external file servers. The data transfer between jobs and the I/O partitions uses a part of the network that is disjoint from that used by any specific job, so again there is no mutual interference. However, distinct I/O operations from separate jobs are serialized by the I/O control processor [263].

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 (Fig. 21). All messages that are in transit are then routed to arbitrary nodes, and stored there as part of the application state [245]. When the application is re-scheduled, the stored messages are injected into the network again to complete their transit. This ensures that jobs to 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 [221]. 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 [282]. 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 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.

### 7.4 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. Standard configurations range from 8 to over 256 nodes. The largest installation contains 256 vector nodes and some 20 scalar nodes for login, network connections, and persistent I/O. The machine has been available since the end of 1992.

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 segment 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.

There are four possible types of partitions.

**Login** — these processors are used by login shells and for editing, compilation, and submitting parallel jobs to other partitions. User logins are load balanced across the nodes in the login partition.

**Batch** — the partition is used for batch scheduling, with each job getting a subset of the nodes for its exclusive use. This can lead to extensive fragmentation, but it is not an issue unless a job requires a contiguous set of nodes. Queueing and allocation are managed by NQS.

**Time sharing** — the nodes are not exclusively allocated, so each node can have processes belonging to several different parallel jobs. In this case, the Unix scheduler time slices between the processes, with no coordination among the different nodes.

**Gang scheduling** — the nodes are shared by multiple jobs, but scheduling is coordinated using the matrix algorithm. As each new job arrives, it is placed in the first scheduling slot with enough free space. The partition manager puts all the processes on all the

nodes to sleep, except the ones in the current slot. The Unix scheduler on each node then schedules its local chosen process, which is the only one that is not asleep.

Multi-context-switching is coordinated by the partition manager. When a time quantum is up, the root process in the partition manager gets an alarm. It then broadcasts a request to the other partition manager processes. These processes are blocked waiting for such a request to arrive. The partition manager processes have a higher priority than the user application processes, so when the request arrives, they immediately execute. They then put the current user process to sleep and wake the one in the next slot. As all the partition manager processes on the different nodes do this at the same time, in response to a broadcast message, the effect is that the jobs in each slot are gang scheduled. Time quantums of several seconds are typically used, to amortize the cost of paging in and out the memory images of large programs.

When multiple processes exist on each node, as in time sharing and gang scheduling partitions, it is necessary to ensure that messages are delivered to the correct process. To do so, the network adapters have their own TLB and page table for each process. Thus they can continue to perform DMA into and out of a process's address space even if that process is not currently running [36]. However, if a page fault occurs due to this activity, the main processor is interrupted in order to handle it. The page tables are maintained by the kernel, allowing the network adapter itself to operate in user mode, thus reducing message passing overhead.

Typical installations have a login partition with scalar nodes, and another partition with vector nodes for parallel processing. The time sharing and batch types are used more often than the gang scheduling.

### 7.5 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. Configurations include from 8 to over 1800 such nodes. 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 [281]. Each node runs the OSF/1 Unix-compatible operating system [448], in conjunction with the NX/2 message passing system [319].

Sharing a Paragon among groups of users is based on partitioning. Partitions are set up by the system administrator. They are typically a rectangular sub-mesh, but they could be any other shape. Partitions are used in either of two ways. One is to run batch jobs. The other is to run interactive jobs with gang scheduling. Typically, the system is divided into a batch partition and an interactive partition, with different sizes for prime time and night time.

The batch partition is manages by NQS. 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, using a modified 2D buddy system algorithm. The packing works for meshes that are not powers of two, jobs can be packed across buddies of different size, and a best effort is made at using contiguous buddies. The allocated nodes are dedicated to the job. The job is then executed to completion, subject to the time limit of the NQS queue.

The implementation of gang scheduling uses the matrix algorithm. Each scheduling slot is called a "tile", invoking the image of covering a rectangular partition of the machine. Jobs are assigned to tiles according to space availability. Some jobs might specify that they need a submesh with certain dimensions, or even a specific subset of nodes. Other jobs just specify the total number of nodes desired. Such requests are used by the system to fill in gaps left by other allocations, under the constraint that the allocated nodes form a single connected component. The algorithm used is to allocate a linear sequence of nodes, with possible folding over.

### 7.6 The Kendall Square Research KSR1

The KSR1 provides a single address space model of computation [152]. 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.

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". 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 memory. 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.

### 7.7 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 [225]. A number of systems have been designed to utilize this untapped power, including Condor [259], which is the base technology for the IBM LoadLeveler, the stealth scheduler [225], Piranha [161], Amber [153], and the distributed queueing system (DQS) from the University of Florida [124].

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[22]. Meeting these requirements is the main feature that distinguished the above systems from other systems that allow users to spawn off processes to multiple hosts, such as PVM [391, 159].

Naturally, the priority given to the workstations' owners implies interference with the performance of the parallel programs being run in the background [214, 247]. 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 [22, 135].

## 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 VI.

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 marketplace. It is well suited for batch pro-

[22] 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 [225, 22].

| *system* | *description* | *section* | *reference* |
|---|---|---|---|
| Alewife form MIT | local queues using hardware contexts and switching | 4.1 | [2, 3] |
| Alfalfa | load balancing by diffusion | 4.1.2 | [165] |
| Amoeba | PEs allocated singly | 3.5 | [288, 393] |
| AMPS | hierarchical structure for balanced mapping | 4.1.1 | [211] |
| APERM-2 | hierarchical structure for balanced mapping | 4.1.1 | [190] |
| Chagori on K2 from Zurich | gang scheduling across whole machine | 5.3 | [383] |
| CHoPP from Columbia | dynamic partitioning by virtual hierarchy | 3.3 | [389] |
| Chrysalis (BBN Butterfly) | independent PEs with local queues within partitions | 4.1 | [243] |
| CM-2 from Thinking Machines | partitioning into quadrants, possible gang scheduling | 3.2.1, 5.3 | [410, 205] |
| CM-5 from Thinking Machines | gang scheduling within partitions created by buddy system | 5.3, 7.3 | [245] |
| Concentrix (Alliant FX/8) | gang scheduling across whole machine | 5.3 | [396] |
| Convex C2 | hardware self scheduling | 3.5 | [407, sect. 3.4] |
| Cosmic Cube | local queues | 4.1 | [357] |
| Cray microtasking | self scheduling from a global queue | 4.2 | [155] |
| Cray T3D | partitions of power-of-two PEs | 3.2 | |
| Crystal | partitioning by users | 3.3 | [112] |
| DASH | global queue with locking | 4.2 | [246] |
| DHC design from Jerusalem | gang scheduling with dynamic grouping based on buddy system | 5.4, 4.1.1 | [143, 145] |
| DQT | gang scheduling with dynamic grouping based on buddy system | 5.4, 4.1.1 | [192] |
| Dynix (Sequent Balance) | global queue with locking | 4.2 | [399] |
| EMBOS (Elxsi) | local queues | 4.1 | [345, 300] |
| EMMA2 | local queues | 4.1 | [18] |
| Flagship | local queues with hardware mapping confined to partitions | 4.1.1, 3.2.1 | [392, 433] |
| Goodyear MPP | dedicated | | [37, 328] |

**Table VI:** *Summary of surveyed systems.*

| *system* | *description* | *section* | *reference* |
|---|---|---|---|
| Heidelberg Polyp | distributed global queue with hardware scheduling and flexible partitioning | 4.2, 4.1.1 | [270] |
| Helios (transputers) | local queues with hardware scheduling within partitions | 4.1 | [309, 182] |
| HEP from Denelcor | local queues using hardware contexts and switching | 4.1 | [376, 204, 219] |
| Hydra on C.mmp at CMU | global queue with locking | 4.2 | [441, chap. 12] |
| IBM GF11 | dedicated | | [40, 236] |
| IBM SP1 | variable partitioning | 3.2.2 | [99] |
| Illiac IV | partitioning into quadrants | 3.2.1 | [34] |
| J-machine from MIT | hardware supported message queues act as local queues | 4.1 | [107, 106] |
| KSR1 system | flexible partitioning with interactive or dedicated use | 7.6 | |
| LLNL gang scheduler on BBN Butterfly | gang scheduling using matrix algorithm, for interactive, batch, and benchmark jobs | 5.4, 7.2 | [168] |
| Mach from CMU | family scheduling: global queue with locking, within dynamic partitions | 4.2 | [48, 47] |
| Mach on IBM RP3 | family scheduling: global queue within dynamic partitions; also local queues | 4.3 | [63] |
| MAXI on Makbilan from Jerusalem | global queue to distribute work, then local queues; experimantal gang scheduling using matrix algorithm | 4.3, 5.4 | [139, 368, 144] |
| Medusa on CM* at CMU | gang scheduling using matrix algorithm | 5.4 | [302] |
| Meiko CS-2 | flexible partitioning by administrator; batch, independent time slicing, or gang scheduling in partitions | 7.4 | |
| MICROS on MICRONET at SUNY | dynamic partitioning by virtual hierarchy | 3.3 | [421, 422] |
| MOOSE on hypercubes at Caltech | local queues | 4.1 | [344] |
| MOSIX from Jerusalem | load balancing by random pairing | 4.1.2 | [33, 31] |
| MuNet | load balancing by diffusion | 4.1.2 | [178] |
| nCUBE | partitioning with buddy system, local queues | 3.2.1 | [180] |
| NETRA | hierarchical structure for balanced mapping | 4.1.1 | [88] |

**Table VI:** *Summary of surveyed systems (continued).*

| *system* | *description* | *section* | *reference* |
|---|---|---|---|
| NQS | batch queueing on partitioned systems | 7.1, 3.2.3 | [221] |
| NX/2 (iPSC/2) | partitioning into subcubes, local queues | 3.2.1, 4.1 | [318] |
| Orion (on $M^3$) | family scheduling with pools of processors | 4.2 | [66] |
| OSF/1 AD | local Unix-like scheduling plus load balancing | 4.1, 4.1.2 | [448] |
| Paragon from Intel | partitioning with optional gang scheduling using matrix algorithm | 7.5 | |
| PASM from Purdue | partitioned multistage network | 3.2.1 | [412] |
| Plan 9 form AT&T | PEs allocated singly | 3.5 | [320] |
| Process Control from Stanford | dynamic partitioning which may change at runtime to reflect load | 3.4 | [409] |
| Psyche on BBN Butterfly from Rochester | local queues, support for two-level scheduling | 4.1, 4.4.2 | [354] |
| Rediflow | load balancing by gradient method | 4.1.2 | [210] |
| Scheduler activations | support for two-level scheduling | 4.4.2 | [12] |
| SIMPLEX (nCUBE) | local queues | 4.1 | [229] |
| SpoC | PEs allocated singly | 3.5 | [289] |
| StarOS on Cm* at CMU | local queues | 4.1 | [158, chap. 6] |
| Symunix on NYU Ultracomputer | global queue based on fetch-and-add | 4.2 | [132, 133] |
| Sylvan | kernel coprocessor for local scheduling | 4.1 | [67] |
| Tera | local queues using hardware contexts and switching | 4.1 | [10, 72] |
| TRAC from Texas Austin | partitioned multistage network | 3.2.1 | [61, 258, chap. 7] |
| Trillium (FPS T-series transputers) | local queues with hardware scheduling | 4.1 | [68] |
| Two-level scheduler from Washington | dynamic partitioning which may change at runtime to reflect load | 3.4 | [447] |
| Victor from IBM Research | dynamic partitioning by users | 3.3 | [363] |
| VORX on HPC | dynamic partitioning by users | 3.3 | [209] |
| Warp systolic array | dedicated | | [16] |
| Xylem on Cedar at Illinois | gang scheduling on PE clusters | 5.3 | [136, 137] |

**Table VI:** *Summary of surveyed systems (continued).*

cessing, 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.

**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.
- 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. Another limitation is that 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 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 and reduced the overall efficiency of the system, installations that multiprogram parallel machines typically do not try to squeeze top performance out of the machine at the same time, so they are willing to pay some overhead for this convenience. 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 place for significant improvement. Regarding the specific area of scheduling in multiprogrammed systems, an obvious goal is to reconcile the differences between academia and practice. For example, two-level scheduling with dynamic partitioning can be made more useful by addressing support for a programming model that looks more like a dedicated

machine. In particular, the system should take the overhead of rearranging the data layout into account when deciding on PE reallocation. The generation of the additional code to support this should be done by the compiler or language runtime system, without placing any additional load on the programmer. There is also ample room for improvement in the implementation of gang scheduling. Goals include increased flexibility in the assignment of resources, reduced fragmentation, and reduced overhead for multi-context-switching.

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. 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 [141]. 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 have not migrated yet to parallel ones.

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 is overall low mean response times, that are relatively insensitive to various workload parameters [249, 248]. 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 the workload characteristics. Moreover, 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 for information about the LLNL gang scheduler, Jim Cownie for information about the Meiko CS-2, Joe Brandenburg and Reagan Moore for information about the Intel Paragon, and Gérard Vichniac for information about the KSR1.

## 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.

**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.

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

**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.

**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.

**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** — an execution of an application program. This typically involves 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.

**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, in system where threads are mapped to PEs.

**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.

**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** — 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**.

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

**Ready Queue** — a queue of threads that are ready 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.

**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 [306]. 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 [189]. 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 [423, 9, 130]. 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 [298]. 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 [306].

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 [335, 128]. 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 is 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 [26], Symunix on the NYU Ultracomputer [134], and Dynix on the Sequent Balance [399]. 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 SP1 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[23], which represent resource allocation, such as an address space and communication capabilities, and threads, which represent execution within the context of a task [334, 397, 342]. 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 [301, 448]. A similar notion of threads exists in Topaz [398], in Peace [350], and in the Portable Common Runtime interface [434]. Variable-weight threads, where some of the resources are allocated per-thread rather then being shared, have also been proposed [19]. 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 switching two threads in the same address space [12]. In addition, even switching light-weight threads requires the CPU's register contents to be stored and restored. This too is costly on

[23]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.

modern RISC processors with large register files [14]. Therefore it has been suggested that all thread manipulation should be done in user space, using two-level scheduling [12]. An alternative method for reducing the cost of thread scheduling is to use continuations where possible [121]. 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 [97], Presto [44, 138], FastThreads [13, 12], WorkCrews [423], the Portable Common Runtime [434], the DIA thread library [77], some implementations of Concurrent C [92, 157], and the XERO system [197]. 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 [353]. 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 [12], first-class threads [274], and cooperative threads [197]. 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.

*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 [144, 445]. Moreover, if the scheduling is not preemptive at all, it could lead to deadlock [62].

A related issue is the granularity of the interactions. In general, granularity refers to amount of computation performed between successive interactions [230]. Thus in fine-grain interactions, interactions (such as synchronization or transfer of data) are numerous and frequent, whereas coarse-grain interactions are faraway.

## References


[1] 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.

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

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

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

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] M. Annaratone, 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.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

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

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

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

[38] 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.

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

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

[41] 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.

[42] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph, "*ParC — an extension of C for shared memory parallel processing*". *Software — Pract. & Exp.*, to appear.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

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

[55] 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.

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

[57] 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.

[58] 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.

[59] 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.

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

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

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

[63] 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.

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

[65] D. C. Burger, R. S. Hyder, B. P. Miller, and D. A. Wood, "*Paging tradeoffs in distributed-shared-memory multiprocessors*". In *Supercomputing '94*, Nov 1994.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

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

[74] 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.

[75] 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.

[76] 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.

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

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

[79] 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.

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

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] S. P. Cheng and S. Dandamudi, "*Scheduling in parallel systems with a hierarchical organization of tasks*". In *Intl. Conf. Supercomputing*, pp. 377–386, Jul 1992.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

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

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

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

[98] 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.

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

[100] 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.

[101] 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.

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

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

[104] 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.

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

[106] 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.

[107] 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.

[108] 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.

[109] 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.

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

[111] 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.

[112] 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.

[113] 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.

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

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

[116] 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.

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

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

[119] 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.

[120] 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.

[121] 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.

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

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

[124] 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`.

[125] 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.

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

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

[128] 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.

[129] 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.

[130] 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.

[131] 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.

[132] 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.

[133] 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.

[134] 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.

[135] 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.

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

[137] 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.

[138] 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.

[139] 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.

[140] D. G. Feitelson, P. F. Corbett, S. J. Baylor, and Y. Hsu, *Satisfying the I/O Requirements of Massively Parallel Supercomputers.* Research Report RC 19008 (83016), IBM T. J. Watson Research Center, Jul 1993.

[141] D. G. Feitelson and B. Nitzberg, *Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860.* Reserach Report RC 19773 (87585), IBM T. J. Watson Research Center, Oct 1994.

[142] D. G. Feitelson and L. Rudolph, *Coscheduling Based on Run-Time Identification of Activity Working Sets.* Research Report RC 18416 (80519), IBM T. J. Watson Research Center, Oct 1992.

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

[144] 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.

[145] 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.

[146] D. G. Feitelson and L. Rudolph, "*Wasted resources in gang scheduling*". In 5th *Jerusalem Conf. Information Technology*, pp. 127–136, IEEE Computer Society Press, Oct 1990.

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

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

[149] J. Flower and R. Williams, "*Parallel programming in comfort*". In 3rd *Conf. Hypercubes, Concurrent Comput., & Appl.*, vol. I, pp. 811–822, Jan 1988.

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

[151] 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.

[152] 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.

[153] M. J. Freeley, 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.

[154] 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.

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

[156] 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.

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

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

[159] 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.

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

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

[162] 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.

[163] 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.

[164] 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.

[165] 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.

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

[167] 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.

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

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

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

[171] 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.

[172] 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.

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

[174] 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.

[175] 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.

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

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

[178] 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.

[179] 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.

[180] 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.

[181] 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.

[182] 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.

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

[184] 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.

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

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

[187] M. Hill et al., "*Design decisions in SPUR*". *Computer* **19(11)**, pp. 8–22, Nov 1986.

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

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

[190] 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.

[191] 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.

[192] A. Hori, Y. Ishikawa, H. Konaka, M. Maeda, and T. Tomokiyo, "*A scalable time-sharing scheduling for partitionable, distributed memory parallel machines*". In 27th *Hawaii Intl. Conf. System Sciences*, Jan 1995.

[193] 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.

[194] 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.

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

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

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

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

[199] 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.

[200] 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.

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

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

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

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

[205] 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.

[206] 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.

[207] 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.

[208] 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.

[209] 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.

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

[211] 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.

[212] 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.

[213] 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.

[214] 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.

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

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

[217] 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.

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

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

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

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

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

[223] 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.

[224] 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.

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

[226] 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.

[227] 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.

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

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

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

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

[232] 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.

[233] 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.

[234] 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.

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

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

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

[238] 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.

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

[240] 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.

[241] 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.

[242] 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.

[243] 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.

[244] 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.

[245] 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.

[246] 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.

[247] 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.

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

[249] 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.

[250] 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.

[251] 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.

[252] 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.

[253] 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.

[254] 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.

[255] 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.

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

[257] 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.

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

[259] 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.

[260] 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.

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

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

[263] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler, "*sfs: a parallel file system for the CM-5*". In *Proc. Summer USENIX Conf.*, pp. 291–305, Jun 1993.

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

[265] 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.

[266] 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.

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

[268] 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.

[269] 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.

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

[271] 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.

[272] 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.

[273] 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.

[274] 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.

[275] 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.

[276] 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.

[277] 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.

[278] 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.

[279] 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.

[280] 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.

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

[282] 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`.

[283] 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.

[284] 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.

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

[286] L. Monier and P. Shindhu, "*The architecture of the Dragon*". In 30th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 118–121, Feb 1985.

[287] T. N. Mudge, G. D. Buzzard, and T. S. Abdel-Rahman, "*A high performance OS for the NCUBE*". In *Hypercube Multiprocessors 1987*, M. T. Heath (ed.), SIAM, 1987.

[288] 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.

[289] 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.

[290] 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.

[291] 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.

[292] 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.

[293] 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.

[294] B. C. Neuman and S. Rao, "*Resource management for distributed parallel systems*". In 2nd *Intl. Symp. High Performance Distributed Comput.*, pp. 316–322, Jul 1993.

[295] 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.

[296] 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.

[297] 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.

[298] 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.

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

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

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

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

[303] 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.

[304] 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.

[305] 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.

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

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

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

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

[310] 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.

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

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

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

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

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

[316] 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.

[317] 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.

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

[319] 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.

[320] 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.

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

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

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

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

[325] 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.

[326] 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.

[327] 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.

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

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

[330] 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.

[331] 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.

[332] 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.

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

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

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

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

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

[338] 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.

[339] 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.

[340] 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.

[341] 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.

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

[343] 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.

[344] 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.

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

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

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

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

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

[350] 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.

[351] M. D. Schroeder and M. Burrows, "*Performance of Firefly RPC*". In 12th *Symp. Operating Systems Principles*, pp. 83–90, Dec 1989.

[352] 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.

[353] 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.

[354] 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.

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

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

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

[358] 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.

[359] 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.

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

[361] 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.

[362] 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.

[363] 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.

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

[365] S. Shimizu, N. Oba, T. Nakada, M. Ohara, and A. Moriwaki, "*Design choices for the TOP-1 multiprocessor workstation*". *IBM J. Res. Dev.* **35(5/6)**, pp. 591–602, Sep/Nov 1991.

[366] 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.

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

[368] 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.

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

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

[371] 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.

[372] 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.

[373] 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.

[374] 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.

[375] 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.

[376] 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.

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

[378] 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.

[379] 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.

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

[381] 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.

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

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

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

[385] 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.

[386] C. B. Stunkel, D. G. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao, "*The SP1 high-performance switch*". In *Scalable High-Performance Comput. Conf.*, pp. 150–157, May 1994.

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

[388] 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.

[389] 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.

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

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

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

[393] 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.

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

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

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

[397] 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.

[398] 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.

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

[400] 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.

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

[402] 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.

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

[404] 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.

[405] J. Torrellas, A. Tucker, and A. Gupta, "*Benefits of cache affinity scheduling in shared-memory multiprocessors: a summary*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 272–274, May 1993.

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

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

[408] 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.

[409] 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.

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

[411] 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.

[412] 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.

[413] 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.

[414] 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.

[415] 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.

[416] 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.

[417] 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.

[418] J. D. Ullman, "*NP-complete scheduling problems*". *J. Comput. Syst. Sci.* **10(3)**, pp. 384–393, Jun 1975.

[419] L. G. Valiant, "*A bridging model for parallel computation*". *Comm. ACM* **33(8)**, pp. 103–111, Aug 1990.

[420] 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.

[421] 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.

[422] 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.

[423] M. T. Vandevoorde and E. S. Roberts, "*WorkCrews: an abstraction for controlling parallelism*". *Intl. J. Parallel Programming* **17(4)**, pp. 347–366, Aug 1988.

[424] 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.

[425] 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.

[426] 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.

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

[428] 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.

[429] H. Wang and Q. Yang, "*Prime cube graph approach for processor allocation in hypercube multiprocessors*". In *Intl. Conf. Parallel Processing*, vol. I, pp. 25–32, Aug 1991.

[430] 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.

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

[432] 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.

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

[434] 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.

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

[436] 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.

[437] R. D. Williams, "*Performance of dynamic load balancing algorithms for unstructured mesh calculations*". *Concurrency — Pract. & Exp.* **3(5)**, pp. 457–481, Oct 1991.

[438] P. H. Worley, "*The effect of time constraints on scaled speedup*". *SIAM J. Sci. Statist. Comput.* **11(5)**, pp. 838–858, Sep 1990.

[439] C. Wu and T. Feng, "*On a class of multi-stage interconnection networks*". *IEEE Trans. Comput.* **C-29(8)**, pp. 694–702, Aug 1980.

[440] 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.

[441] W. A. Wulf, R. Levin, and S. P. Harbison, *HYDRA/C.mmp: An Experimental Computer System*. McGraw-Hill, 1981.

[442] 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.

[443] S. Y. Yoon, O. Kang, S. R. Maeng, and J. W. Cho, "*A heuristic processor allocation strategy in hypercube systems*". In 3rd *spdp*, pp. 574–581, Dec 1991.

[444] 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.

[445] 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.

[446] 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.

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

[448] 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.

[449] 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.

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

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