Toward Convergence in Job Schedulers for Parallel Supercomputers

Dror G. Feitelson and Larry Rudolph

Institute of Computer Science
The Hebrew University, 91904 Jerusalem, Israel
{feit,rudolph}@cs.huji.ac.il

Abstract. The space of job schedulers for parallel supercomputers is rather fragmented, because different researchers tend to make different assumptions about the goals of the scheduler, the information that is available about the workload, and the operations that the scheduler may perform. We argue that by identifying these assumptions explicitly, it is possible to reach a level of convergence. For example, it is possible to unite most of the different assumptions into a common framework by associating a suitable cost function with the execution of each job. The cost function reflects knowledge about the job and the degree to which it fits the goals of the system. Given such cost functions, scheduling is done to maximize the system's profit.

1 Introduction

Both theoreticians and practitioners have been investigating and implementing various types of schedulers, and analyzing their performance over a wide range of workloads, leading to a large and varied body of knowledge [13]. However, many of the assumptions as to the type of workload and the goals of the scheduler are incompatible. We argue that the best features can and should be combined.

The following principles are common features of all scheduling systems:

- The scheduler services all jobs that are submitted.
- Jobs that provide optional resource requirement specifications are rewarded.
- Jobs that are coded in a "schedulingly friendly" style are rewarded.
- Accounting and quality of service are the tools used to reward jobs.

The road to convergence starts with an explicit understanding of the differences that are to be bridged. The differences stem from different basic assumptions relating to performance metrics (Section 2), workload (Section 3), and scheduler actions (Section 4). This paper therefore begins by reviewing these assumptions. We can identify several common combinations of assumptions, which have led to the creation of isolated "clusters of assumptions" (Section 5). Then, we propose several ways to achieve convergence (Section 6).

Before continuing, the following plea for cooperation is issued. At the very minimum, we wish that all articles about job schedulers, either real or paper design, make clear their assumptions about the workload, the permissible actions allowed by the system, and the metric that is being optimized.

2 Assumptions About the Goals of a Job Scheduler

There are many scheduling systems for parallel computers and even more that are being proposed and analyzed. The systems are widely disparate both in what they hope to accomplish and in the ways they hope to accomplish it. This section reviews some the various, sometimes conflicting goals of schedulers.

2.1 Run Jobs

The primary goal of all schedulers is to enable the successful execution of a job, hopefully a parallel one, on a parallel machine. While obvious, this goal should never be forgotten. In particular, maximizing secondary goals should not starve certain class of jobs,

Secondary scheduling goals, described in the following subsections, vary and depend on satisfying the needs of the group versus the needs of the individual. These goals can be broadly classified as being system centric or user centric. Some are measurable well-defined metrics, while others are functional desiderata. They are summarized in Table 1.

	user centric	*system centric*
metric	response time	utilization throughput
function	run jobs	
	emulate dedicated machine	administrative preferences

Table 1. *Classification of scheduler goals.*

2.2 Maximize Utilization of the Machine

It might appear obvious that a scheduler should maximize the utilization of the machine. Utilization can be defined in either of two ways: either as the percentage of CPU cycles *actually used* for productive computation, or as the percentage of CPU cycles *allocated* to user jobs that pay for them. The difference is that the first definition integrates the efficiency of user code into the equation, while the latter makes a clear distinction between the allocation of resources by the operating system and their use by the user.

The problem with utilization as a metric is that it is largely dependent on system load (Fig 1). Consider a simple queuing model of an operating system scheduler: requests to run jobs arrive, and they are serviced by the system. When load is low and all jobs can be serviced, the utilization is equal to the load. When load is high and the system saturates, utilization is equal to the saturation point. Therefore the goal of a system designer is not to increase utilization *per se* but

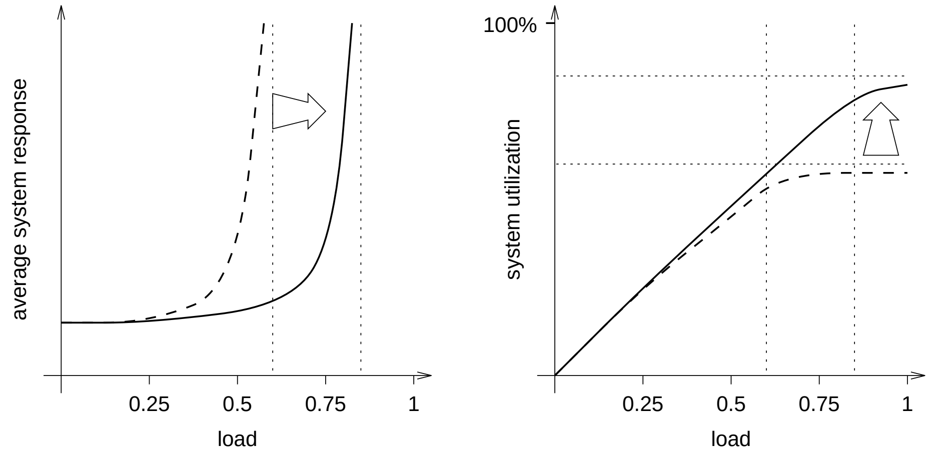

Fig. 1. *Utilization depends on the system load and on how efficiently the system handles it, i.e. at what point does it saturate. Arrows indicate improvement in system efficiency.*

rather to delay the onset of saturation. In other words, A "good" system will be able to sustain a higher load before becoming saturated, which means that a higher utilization is possible if the load demands it.

Another problem with the utilization metric is that adopting it may lead to starvation of certain jobs. For example, if the job stream includes many jobs that require all the processors in the system, and only a handful of jobs that require fewer processors and cause significant fragmentation, it is best from a utilization point-of-view to only schedule the jobs that need all the processors.

2.3 Maximize Throughput

Throughput is the number of jobs completed per unit time. The throughput metric is similar to the utilization metric in the sense that it is affected by system load and efficiency. But, whereas utilization is maximized when there are mainly massively parallel jobs executing for long time periods, the throughput metric is maximized when there are many small (in parallelism and in CPU usage) jobs.

The rationale behind this metric is that the higher the throughput the more users are satisfied. In general, maximizing the average throughput also minimizes the average response time for a job. This is true only when there is no knowledge about the execution time of a job. If that is known, then scheduling the shortest jobs first will reduce the average response time without affecting the average throughput rate.

Throughput has the same problem as utilization: by focusing on the average values, the system may undermine the primary goal. A parallel job mix may be difficult to schedule and can cause significant fragmentation [16]. For example, a 27 node job on a 32 node parallel machine leaves an awkward 5 nodes free. If one is interested in maximizing the average number of jobs processed by the system, it might be better to ignore jobs that cause fragmentation altogether.

2.4 Reduce Average Response time

Reducing average response time is a very common goal, especially in interactive systems. While there is some debate about the exact definition of "response time", most researchers use it as a synonym for "turnaround time," i.e. from job submittal[1]to job completion time, rather than the time till when the first output is produced [41]).

One problem with the usual response time metric is its use of absolute values. Consider a job J_a that responds in one day and another job J_b that responds in one minuite. Both jobs have the same computational requirement, then there might be something wrong with the scheduler. On the other hand, if job J_a requires 24 hours of computation time, then the one day response time is pretty good whereas if job J_b only required 1 microsecond of computation time, then the one minute response time may be bad. Jobs can be perceived as having different weights, depending on their run time. A possible solution to this problem is to use the average *slowdown* as a metric instead, where slowdown is the ratio of the time it takes to run the job on a loaded system divided by the time it takes on a dedicated system (this is sometimes also called the "response ratio" [4]). This normalizes all jobs to the same scale.

Another problem with this metric is its linear regard to time. Actually response time should be measured as perceived by those who are interested, e.g. human users. For humans, the difference between a response of 1ms and 100ms is immeasurable, but the difference between 1s and 100s is very annoying [18].

Finally, it should be noted that not all jobs require the same level of service in terms of response time. Interactive jobs require interactive response times, preferably of not more than a couple of seconds. Time critical jobs require application-dependent response times (e.g. tomorrow's weather forecast should be ready in time to be useful). And some jobs do not have any specific time constraints. In fact, most parallel systems make a distinction between batch jobs and direct jobs, with batch jobs executed only at night or when the machine would otherwise be idle. However most efforts at modeling do not take this distinction into account.

2.5 Fairness vs. Aadministrative Preferences

Fairness is not often advocated as a requirement on its own accord, but it underlies the requirements for maximizing throughput and minimizing average response time. But should all jobs be treated the same? For example, we have already noted that batch jobs do not require short response times.

[1] We are following Steve Hortney's campaign to use the term *job submittal* in place of the masocistic term *job submission*; despite the fact that jobs are at the mercy of the scheduler.

Since all jobs are not created equal, it is often desirable to give preference to certain classes of jobs. For example, is there any preference to schedule two 8 node jobs in place of a single 16 node job? There is no abstract answer to this question; it is dictated by the management personnel of the supercomputer. Due to the high cost of parallel supercomputers, and their resulting use as shared resources that are specifically targeted at large computational problems, some installations do indeed try to encourage highly parallel jobs at the expense of those with only moderate parallelism.

Encouraging highly parallel jobs can be viewed as "fairness to threads." A job with more threads, that exhibits a larger degree of parallelism, is assumed to require more computational resources, and is therefore given better service. That is, administrative preferences may cause the system to be unfair to to users or to jobs (that is, all jobs are not considered equal).

2.6 Give the Illusion of a Dedicated Parallel Machine

Multiuser workstations and other non-parallel computers attempt to provide the user with the illusion of a dedicated machine. This is especially true for interactive jobs. When a parallel computer supports multiple users via time slicing or space slicing, it is generally desirable to provide the illusion of a dedicated parallel machine. We define this to mean that if a job receives $1/k$ of the total CPU cycles, then the job should take about k times as long to complete as it would on a dedicated machine, without taking any special actions.

To understand the issue here, consider a job scheduler that allows the individual activities of each parallel job to be time sliced independently. An activity may then waste many CPU cycles waiting for a message to be sent by another activity that is currently not executing. Had the machine been dedicated to the job, this wasted time would not occur. Thus, a user might be charged more CPU time, just because the scheduler decided to execute several jobs in an uncoordinated fashion (a simple solution for this case is therefore to use gang scheduling [17]).

2.7 Issues That Are Often Ignored

An important observation is that most simple metrics have simple failure modes in which they cause starvation for a class of jobs that do not promote the predefined metric. For example,

- Maximizing utilization may not schedule jobs that cause fragmentation
- Maximizing throughput may not schedule large jobs
- Minimizing response time ignores the fact that batch jobs do not need it nor want to pay for it

A more subtle observation is that a scheduling-centric metric cannot account for interactions with other resources that may become depleted first. For example, memory is a critical resource and if it is not managed correctly, an application may suffer from thrashing. Consider for a moment a job that consists of two processes, a consumer and a producer. The producer sends messages to the consumer process. If the rate of production is equal to the rate of consumption, and the two processes execute simultaneously in parallel, memory can be used efficiently. On the other hand, if they do not execute simultaneously, each message may need to be buffered. Buffering consumes system resources and could cause other jobs to frequently page fault. Unchecked scheduling of parallel jobs may quickly deplete "swap" disk space. Many large installations provide massive storage systems with latency times measured in the minutes. Such time frames must be handled differently from the times involved with a simple page fault.

In the area of functional requirements, the need to support whatever users may want to do is often overlooked. There are a number of examples of over-sophisticated schedulers that may end up limiting their users:

- Users sometimes want full control over the number of processors used to run their jobs, e.g. in order to generate speedup curves. A scheduler that sets the number automatically and does not provide an override mechanism makes this impossible to achieve.
- Different applications are easier to write in different programming styles, and users also have their personal preferences. Schedulers that limit the styles that are supported may thus alienate users that would rather use another style. This applies, for example, to schedulers that require all jobs to be able to adapt to changes in resource allocation at runtime, something that is difficult to achieve in certain cases.

Finally, in modern complex systems it is often the case that the scheduler must interact with external agents, e.g. as part of a system for heterogeneous computing [20,30]. As part of such interactions, the scheduler might need to make reservations for future use. This functionality is often missing, and the performance implications (e.g. loss of resources due to reservations) are usually not included in models and analysis.

3 Assumptions About the Workload

Although the basic object that is to be scheduled is a job, there is a major division on the characteristics of jobs and what the scheduler knows about them. We restrict our attention to *parallel jobs*. That is, jobs that are composed of independent communicating activities. There is an underlying assumption that communication time is fast, e.g. the time to communicate a word of information is only about an order of magnitude longer than the time required for a CPU to fetch a word from its memory. We exclude client-server type jobs and other distributed computing jobs.

The definition of a parallel job from the point of view of a scheduler, unfortunately, is not so clear cut. There are many styles of parallel programs, and many structures that are imposed by some runtime systems and compilers. We identify four types of jobs based on the number of processors to be used by the job. This number may be specified by the user, either within the program itself or as part of job submittal specification, or it may be dictated by the scheduler. In addition, the number of processors may be fixed at the start of program execution or may change during the course of the computation. There are thus four classes (Table 2). Although one of the problems with the field is that there are too many conflicting terms, we risk adding to the complexity by proposing yet another set of terms. We feel that these terms may help highlight the differences.

who decides number	*when is it decided* *at submittal*	*during execution*
user	Rigid	Evolving
system	Moldable	Malleable

Table 2. *Classification of job types based on specifying number of processors used.*

3.1 Rigid Jobs

A *rigid* job requires a certain number of processors in order to execute, as specified by the user at job submittal time; there may be other resource requirements as well, such as CPU time and memory, but we'll focus on the requirement for processors. A rigid job will not execute with fewer processors and will not make use of any additional processors. The scheduler does not know anything about the job besides the number of processors it needs.

¿From the programmer's perspective, the reasons for using a rigid formulation vary. In some cases it is simply what the system interface supports, so even jobs that are written as moldable jobs must be submitted as rigid ones. Applications written in High Performance Fortran are usually inflexible and there are often optimal decompositions based on the problem size. For example, it might be very inefficient to decompose a job with an array of size 100 into 17 processors.

3.2 Evolving Jobs

An *evolving job* is one that may change its resource requirements during execution. Note that it is the application itself that initiates the changes. The system must satisfy the requests or the job will not be able to continue its execution. Again, the scheduler knows nothing about the job except for its current requirement for processors.

Although such jobs are not common, there is much activity in the community to define a standard for dynamic processor requests. Such facilities already exist in the PVM interface [21], and they are also in the process of being incorporated into the MPI-2 standard.

The reason for the interest in this feature is that many parallel jobs are composed of multiple phases, and each phase has different characteristics. In particular, different phases may contain different degrees of parallelism. By telling the scheduler about these changes, it is possible for jobs to obtain additional resources when needed, and relinquish them when they can be used more profitably elsewhere (thereby reducing the cost of running the job). Also, this type of jobs is commonly modeled by task graphs with changing widths [55].

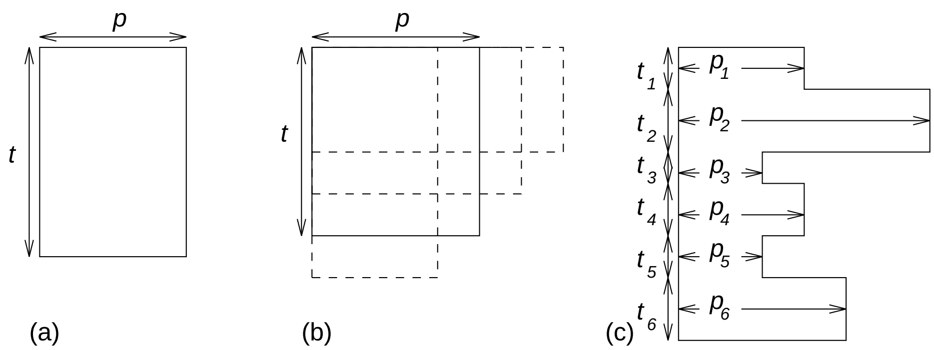

Fig. 2. *(a) Rigid jobs define a rectangle in processor-time space. (b) Moldable jobs use one out of a choice of such rectangles. (c) Evolving and malleable jobs both have a profile with a changing number of processors. The difference is that in evolving jobs the changes are initiated by the job, while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

3.3 Moldable Jobs

A job may be flexible in the number of processors that it requires and may allow the system to dictate the allocated number of processors. There are two types of such flexible jobs, which we call *moldable* and *malleable*. With moldable jobs, the number of processors is set at the beginning of execution, and the job initially configures itself to adapt to this number. After it begins execution, the job cannot be reconfigured. It has already conformed to the mold.

If the number of processors is selected by the user and presented to the scheduler as a requirement, the job is actually rigid from the scheduler's point of view. But, given a range of choices, the scheduler can set the number of processors based on knowledge about the system load and competing jobs, knowledge that is typically not available to the user. This has been called "adaptive partitioning" in the literature [44].

A moldable job can be made to execute over a wide range of processors. There is often a minimum number of processors on which it can execute, and above that number, additional processors can be used to improve performance, up to some saturation point. The resource requirement of a minimal number of processors is usually due to memory and response time constraints. ¿From a local efficiency point of view, there is a best number of processors for the job, at the knee of the speedup curve. But since the scheduler cares about maximizing some overall system performance properties, it might be best if the job is executed at another point. In any case, knowledge about application characteristics is typically required [46].

Programs written using the SPMD style, e.g. with the MPI library package, are often moldable. Moreover, workload traces from real parallel systems show that indeed the same application may run several times using different partition sizes [14].

3.4 Malleable Jobs

The most flexible type of jobs are *malleable* ones that can adapt to changes in the number of processors during execution. The main programming styles that permits this flexibility consists of many short independent tasks that access shared data in a very stylized way. For example, if all the tasks have no side effects, then to reduce the number of processors, some tasks are terminated and restarted later on the remaining processors [43].

It is fairly well accepted to call changing the number of processors at runtime "dynamic partitioning". We prefer to call the jobs "malleable" rather than "dynamic" because the term "dynamic" does not indicate who is doing the dynamic allocation. On the other hand, the shape of a malleable object can be changed by an outside entity, while an evolving object is one that changes of its own accord. However, note that evolving and malleable should usually come together, because one job's evolution will cause others to have to reconfigure.

Analyzing the benefits of dynamic partitioning and malleable jobs has been the subject of much recent research [36,7,28,39,34,23]. This research typically compares the cost of reconfiguration with the resulting improvement in overall performance. But such comparisons do not give a full picture. In many cases, changing the number of processors allocated to a job requires complex interactions between the operating system and the application's runtime system [2]. For example, if the thread running on a certain processor holds a lock and then the processor is taken away, there may be no way to free the lock. Implementing the required interfaces to solve such problems naturally complicates the system, and makes it harder to implement, which is one reason why malleable jobs are currently not supported on any commercial parallel machines.

An interesting benefit of malleable jobs is that the option for changes can be used to allow the system to collect information about the job at runtime, by trying several configurations and checking the resulting performance. This information can later be used to guide allocation decisions [37]. This approach has obvious advantages over requiring the information to be available in advance, as is needed for moldable jobs.

4 Assumptions About Permissible Actions

A scheduler must execute in the environment of an existing operating system and machine architecture. This environment restricts the operations it is allowed to perform. In some machines, the operating system provides a *single system image*. That is, it does not matter from which processor an action is executed, they are all identical. Shared Memory Parallel Processors (SMPs) often have this feature and it is also being explored in some distributed systems [29]. When there is no single system image, it is difficult to migrate tasks. The machine architecture may impose restrictions on the types of processor partitions available and the ability to share access to the communication substrate.

The most limited system has partition sizes of a fixed number of processors, and allows only one job to execute from start to finish in a partition at any time. The scheduler simply needs to keep track of empty partitions and map incoming jobs to the appropriate partition. At the extreme, there would be only one partition and so only one job can execute at a time. But most systems allow many more powerful options.

4.1 Flexible Partitioning

Nowadays, most systems allow the processors to be partitioned on a job by job basis. This is sometimes referred to as *space slicing*. The exact number of processors may be forced to match the topology of the machine, as in hypercube topologies in which partitions must also be hypercubes but of a smaller dimension. But in many cases, especially when the network topology is hidden from the programmer, there are no such restrictions and partitioned may be formed using arbitrary subsets of processors.

A rigid job is submitted for execution along with a specification of the number of processors that it requires. The scheduler then creates a partition of that size and schedules the job to execute within that partition [53,32,20,1,9,31,33]. With moldable jobs, it is the scheduler that selects the partition size [44].

Evolving and malleable jobs require partitions that are not only flexible but can also change dynamically at runtime. This places an added burden both on the programmer, who must write application code that requests and adapts to such changes, and on the scheduler, that must handle the re-allocation decisions and coordinate them with the applications.

One common heuristic for dynamic partitioning is to strive for equal sized partitions (usually called "equipartitioning") [35]. The problem with this approach is that it might require all jobs to be interrupted whenever something changes. An alternative is to use *folding* [35]. With folding, the number of processors allocated to a job can only grow or shrink by factors of 2. That is, the partition size may be halved or doubled. When a partition is halved, the runtime system may choose to simply "fold over" the application, and time-slice two tasks on each processor. Thus an application that has a balanced workload over a particular partition size is likely to remain balanced after a folding operation. Many speedup curves resemble step functions, with poor speedup values for non powers of two number of processors. However, there is some debate over the benefits of folding [28,39].

4.2 Preemption and Time Slicing

Dynamic partitioning, discussed above, requires certain processors to be preempted and re-allocated in order to accommodate load changes. Another type of preemption is that used in order to time-slice multiple applications, as is commonly done on uniprocessors. We identify this feature for special attention since many systems cannot support such preemption due to limitations on the message-passing architecture. For example, some machines require that the interface to the network be set up before a job begins execution. There may be no easy way to switch jobs without compromising the integrity of the messages. On some systems, switching message space protection from one job to another is permitted but is a very time consuming operation.

An extension of preemption is the ability to preempt all the members of a parallel job at the same time, as well as restarting all the members of another job. This is called "gang scheduling". It is generally agreed that if time slicing is used, then it should be implemented via gang scheduling, rather than letting each processor do its own uncoordinated time slicing. The reason is that with gang scheduling all the processes of a given job execute simultaneously on different processors, thus supporting the use of fine-grain interactions [17]. An interesting observation is that it is desirable to also preempt the network, i.e. to flush any traffic that belongs to the job that is being de-scheduled, so as to present a clean slate to the new job [26].

Gang scheduling suffers the overhead of context switching and corrupts cache state, but for a large enough time quantum these overheads can be made insignificant [23]. On the other hand, time slicing in general reduces the average response time provided the distribution of execution times has a large variance [40], which in fact it typically occurs [14].

4.3 Migration

Migration refers to the ability of a scheduler to move an executing job or some of its components to other processors. As such it is an extension of preemption: a task stops running on a certain processor, and it restarts on another processor. Reasons for migration include packing in order to reduce fragmentation [6,12], and the need to withdraw from a workstation when its owner returns [42].

Migration is simple on shared memory machines, because threads do not have any state that is local to the processor except for their cache and TLB footprints. The challenge is to ensure that interacting threads map to distinct processors.

Migration is significantly more problematic in distributed memory machines as it requires migrating the local memory which can be a very expensive operation. The ability to migrate a task is often hindered by systems whose message passing libraries specify physical processor numbers as source or destination fields for messages. Note that the elimination of virtual to physical processor mapping increases the speed of a communication. Many systems map the network FIFO queues into user space; disconnecting and reconnecting may also require significant overhead.

4.4 Change Job Execution Order

A scheduler may be able to process jobs in an order different from the job submittal order. Many batch systems have some such flexibility [32,24,53]. Of course, this flexibility is only useful if there is some information as to the resource requirements of the waiting jobs as well as any deadlines or response time requirements.

We mention this option since it easily leads to violation of the primary goal of a scheduler – the execution of every job. Some aging mechanism is required to ensure that jobs are not passed over for arbitrarily long time periods.

5 Research Clusters

The spectrum of job schedulers for parallel machines may be expected to span a large part of all the different options for assumptions about goals, metrics, and workloads. In fact this is not so. Several "clusters" have formed, each with its own set of assumptions, and often oblivious of the others. This section identifies and characterizes these "clusters of assumptions," as a prelude to suggestions for some degree of convergence in the next section. The clusters are summarized in Table 3. The most controversial part is probably our classification of the goals. Mostly, if the scheduling system ensured that jobs run with all their required resources in a timely fashion, response time was considered as a goal. We evaluated the goals of utilization and throughput by considering how the scheduler treats large batch jobs and small interactive jobs.

5.1 Rigid Jobs and Variable Partitioning

Maybe the simplest scheduling scheme is to reduce the role of the operating system to that of a processor rental agency. Jobs request a certain number of processors, and the system provides them for exclusive use if they are available. The only goal is to (eventually) run the jobs. No knowledge about job behavior is assumed, and no special actions need be supported, except some measure of partitioning the machine. This scheme has been called "variable partitioning" or "pure space sharing" in the literature.

Despite its simplicity and the resulting drawbacks in terms of responsiveness, fragmentation, and reliability, this scheme is widely used. It is especially common on large distributed memory machines [53,27,20,9,31]. The reason is that it gets the job done, albeit not optimally, but with relatively little investment in system development. In an industry where time-to-market is a crucial element of success, this is a true virtue [1]. As a result, users sometimes have to revert to signup sheets as the actual processor allocation mechanism.

Because variable partitioning cannot run jobs immediately if the requested number of processors is not available, jobs often have to be queued. As a result this scheme implies a batch mode of operation. With sufficient backlog, it is then possible to select an execution order that improves system utilization

	variable partitioning	*gang scheduling*	*shared queue*	*adaptive partitioning*	*dynamic partitioning*
goals					
run jobs	yes	yes	yes	yes	yes
utilization	no			yes	yes
throughput	no		yes	yes	yes
response	no	yes		no	
admin					
dedicated	yes	yes	yes		no
workloads					
rigid	yes	yes	yes	yes	no
evolving	no	no	yes	no	yes
moldable	yes	yes	yes	yes	no
malleable	no	no	no	no	yes
actions					
partitioning	yes	yes	no	yes	yes
preemption	no	yes	yes	no	yes
synchronized	n/a	yes	no	n/a	no
migration	no	no	yes	no	
ordering	yes	n/a	n/a		

Table 3. *"Clusters" of common combinations of assumptions.*

and throughput [32,48]. Thus even this simple scheme has bred some interesting research. In addition, it has prompted research into improvements such as adaptive and dynamic partitioning (see below).

5.2 Rigid Jobs and Gang Scheduling

Gang scheduling has been re-invented many times over because it is an intuitive extension of timesharing on uniprocessors. It also supports interactive use and gives the illusion of using a dedicated machine, without placing restrictions on the programming model and without assuming knowledge about the workload. It has therefore enjoyed considerable popularity among vendors, at least in the form of "hype" (all vendors claim to support some form of gang scheduling) but good implementations also exist. There have been a number of experimental implementations [38,15,12,54,22,25] that demonstrate its usefulness.

Academically speaking, gang scheduling has repeatedly been shown to be inferior to dynamic partitioning (see below), but only by a small margin [23,8]. The main drawbacks cited are interference with cache state, and possible loss of resources to fragmentation. As the first can be lessened by using long time quanta [23], and recent research suggests that the second is not so severe [12], it seems that the advantages of gang scheduling generally outweigh its drawbacks.

However, there are still some unresolved issues. The main one concerns the possible interaction between gang scheduling and over-committing memory resources. Time slicing between two active jobs requires more memory that executing these jobs sequentially. The direct solution is to provide adequate swapping to disks, but so far little research has been done on this issue, and parallel systems are notoriously underpowered in terms of I/O. Another criticism of gang scheduling is the lack of fault tolerance — if a processor fails, many jobs may have to be aborted. While important, this problem is not unique to gang scheduling: it is also present in other scheduling schemes and in other components of parallel operating systems, e.g. message passing facilities.

In summary, the finer the granularity of interaction between members of a parallel job, the more gang scheduling is required.

5.3 Evolving Jobs and a Shared Queue

Another simple extension of timesharing on uniprocessors is to use a shared queue. Each processor chooses a process from the head of the centralized ready queue, executes it for a time quantum, and then returns it to the tail of the queue. As processors are not allocated permanently to jobs, the number of processes in a job may change during runtime without causing any problems. This scheme is especially suitable for shared memory multiprocessors, and indeed it is used on many bus-based systems [50,3].

Using a shared queue as described above may suffer from three drawbacks: contention for the queue, frequent migration of processes, and lack of coordinated scheduling. The issue of possible contention has lead to the design of wait-free queues, where different processes can access the queue simultaneously by using suitable hardware primitives, such as fetch-and-add. Indeed, this was one of the driving forces in the design of the NYU Ultracomputer, and its support for fetch-and-add via a combining multistage network [11]. However, the idea of combining network has not caught on because of their added complexity and design costs.

Migration occurs in this scheme because processes are typically executed on a different processor each time they arrive at the head of the queue. As a result, any state that may be left in a processor's cache is lost. It has been suggested that this effect can be reduced by using affinity scheduling, where an effort is made to re-schedule the process on the same processor as used last time [49,10]. However, it is not clear to what degree data indeed remains in the cache, and in any case, affinity scheduling is largely equivalent to just using longer time quanta [51].

The third issue, lack of coordinated scheduling, may cause problems for applications where the processes interact with each other frequently. The only solution is to use gang scheduling. While gang scheduling and a shared queue seem to be in conflict with each other, a scheme that integrates both has been designed in the context of the IRIX operating system for SGI multiprocessor workstations [3].

Finally, it should be noted that this scheme benefits from similarity with runtime systems and thread packages that run within the confines of a single job.

5.4 Moldable Jobs and Adaptive Partitioning

As noted above, variable partitioning is a simple but somewhat inefficient scheduling scheme. The inefficiencies result both from fragmentation, where the remaining processors are insufficient for any queued job, and from the fact that jobs may request more processors than they can use efficiently. It is therefore an interesting question to assess the degree to which efficiency can be improved by adding flexibility and information about the characteristics of different jobs.

The model adopted for this line of research is that jobs can be molded to run on different numbers of processors, and some information about their average parallelism or execution profile is provided. This allows the system to judiciously choose partition sizes, without significantly affecting the programming model. Thus when the system is lightly loaded, jobs are allowed to use larger numbers of processors, even if they do not utilize them efficiently, but when system load increases, jobs are cut down to size [44,47,46]. It has also been suggested that the system keep some processors idle on the side in anticipation of additional arrivals [45].

In summary, this approach has generated a rather large body of research, but it has yet to lead to any implementations in real systems.

5.5 Malleable Jobs and Dynamic Partitioning

A more extreme approach to system optimization calls for sacrificing common programming models along with the illusion of a dedicated machine in order to promote efficiency. In some sense, this approach demonstrates the best performance that can be achieved, given full system flexibility in the allocation of resources, and jobs that are willing to cooperate with the system (and through it, with each other).

The programming model requires each job to accurately inform the system their resource requirements, and be able to adapt to changes in resource allocation that result from fluctuations in system load. The system uses information about the characteristics of the jobs to decide on the optimal allocation: jobs are only given additional processors if there is nothing better to do with them. When a new job arrives, some processors are taken from existing jobs and given to the new job, so that it will not have to wait. When a job terminates, its processors are distributed among the other jobs, so as not to waste them [52,34,35].

A good implementation requires co-design of the operating system, the runtime system, and the programming environment [2]. Indeed, no production implementation for parallel machines have been reported so far, despite much research that shows the benefits of this approach in terms of efficiency. On the other hand, this approach has the unique advantage that jobs may be able to tolerate system faults: a faulty processor is similar to one that is taken away and given to another job. Likewise, jobs running on a network of workstations will be able to tolerate workstations that drop out of the processor pool when they are reclaimed by their owners. Therefore this approach has lately become prominent in the context of network computing [5,43].

6 Steps Towards Convergence

The field of job scheduling for parallel processing is in flux. It is being driven by the needs of growing numbers of installations. Sadly, there seems to be a growing divergence between the practical approaches adopted by actual users and the more sophisticated approaches advocated by theoreticians. Our goal here is to point out ways in which this gap can be bridged, and show how the different communities can benefit from the work of each other.

6.1 Step One: Be Explicit About the Differences

One of the problems in the field is the difficulty in relating the various research results to each other. In some cases such comparisons are actually comparing apples with oranges as if they were equivalent, in some the comparison is dismissed because it mistakenly seems to be irrelevant, and in some cases it is just hard to see whether or not the comparison makes sense.

The root of the problem is with the significant implicit assumptions and imprecise, confusing terminology. For example, dynamic partitioning research papers usually do not make explicit the assumption that jobs are coded in a style that tolerates changes in resource allocation at runtime, and that jobs are willing to cooperate in order to achieve greater overall (system) efficiency. Dynamic partitioning also assumes that the speedup functions for the jobs are not trivial step functions in which no speedup is achieved until a critical number of processors is made available and that additional processors do not affect job performance; such speedup functions describe rigid jobs. Likewise, work on gang scheduling usually does not make explicit the assumption that all programming styles must be supported with no changes. Gang scheduling research assumes that most jobs are not "embarrassingly parallel jobs" that require infrequent interaction. When these assumptions are clarified, it is evident that the two schemes operate in different frameworks. It is true that both strive for greater system efficiency, but each does so in a different framework based on different assumptions about the workload, so a comparison between them is debatable at best.

The other problem is one of terminology. For example *gang scheduling* means the same thing as *coscheduling* except that coscheduling will also schedule only some threads of a parallel job instead of leaving idle processor. Other researchers use the terms *hard* and *soft* gang scheduling to denote these differences. Such practices make it harder for readers to figure out what the results are about, and increase the cognitive load. While it is true that existing terminology is not always optimal and may suffer from historical artifacts, it is still usually better to stick with the established terms.

To summarize, we recommend that each paper should state the assumptions use consistent terminology, and resist the urge to define new terms or give new meaning to old ones.

6.2 Step Two: Acknowledge Deficiencies and Search for Solutions

Each scheme, by way of being dependent on a set of assumptions, has weaknesses when the assumptions are violated. Rather than assuming them away, one should try to overcome them by incorporating the ideas of other schemes. This may make the difference between a theoretical proposal and a real system.

For example, major weaknesses of variable partitioning are the loss of resources to fragmentation and the lack of responsiveness. Different schemes have been proposed to overcome one or the other of these problems: gang scheduling improves responsiveness, while adaptive and dynamic partitioning reduce fragmentation and improve throughput, provided the assumptions regarding the workload are met. But each scheme still has weaknesses when its assumptions are not met in full.

While gang scheduling improves system responsiveness through the use of time slicing, and also alleviates the ill-effects of fragmentation to some degree, it does not allow for optimizations based on global knowledge of the system load. Thus each job runs on a predefined number of processors, and this number cannot be changed. Efficiency may be improved if support for malleable jobs is included. Then, jobs are indeed malleable can be re-shaped to improve efficiency based on knowledge of the speedup curve. If the workload does not include sufficient small jobs, malleable jobs can also be re-shaped to fill in holes and reduce fragmentation.

Dynamic partitioning improves efficiency by eliminating fragmentation and reducing the processor allocation towards the optimal operation point for each job when load is high. However, it must still deal with unfavorable conditions, such as non-malleable jobs, jobs that do not provide the required information about their operation characteristics, and situations in which too many jobs have been submitted to run all of them at once. These problems can be addressed using mechanisms of adaptive partitioning and gang scheduling.

The bottom line is that combining ideas from different scheduling schemes can lead to important benefits for real systems. This does not mean that research on the individual schemes is not important — on the contrary, it is definitely necessary to focus on a narrow scheme and reduce the number of variables in order to perform a detailed analysis. But when it comes to building real systems, it is necessary to take a broader view.

6.3 Step Three: Broaden the Scope

Since expensive supercomputers should address a broad spectrum of application programs, a job scheduler should be able to handle a workload consisting of all sorts of jobs, be they rigid, evolving, moldable, or malleable, and all levels of resource specifications. Moreover, a scheduler should implement the constantly evolving policies and goals of the computer installation. Although we do not propose a scheduler that achieves these goals, it is important to state take steps in this direction.

Consider, for a moment, an 8 processor system and two parallel jobs. Suppose the jobs are malleable and the system is extremely flexible in that the two jobs can be executed in a gang scheduled, time-sliced manner with 8 nodes allocated to each job, or it can use space slicing and dedicate 4 nodes to each job. Suppose further that each job achieves linear speedup. Should the scheduler use space or time slicing? If the jobs require the same execution time, then there should be little difference, and system overheads should dictate the choice. Now suppose one job was submitted first and is allocated 8 nodes exclusively. When the second job arrives, is it more expensive to repartition the first job or to gang schedule them? The answer now depends on the computational times of the jobs – the number of context switches times their cost versus the repartitioning cost.

As the load on the system changes, the fraction of system resources allocated to each job changes. In time sharing system, it is easy to see this change. In space sharing systems, dynamic repartitioning attempts to share the resource of the processor. However, it is important to keep in mind that there are other resources such as physical memory and disk swap space.

So, one point is to understand the tradeoffs between the various modes of sharing.

Specifically, we propose the use of *cost functions* that reflect the policies and goals of the computer installation. For a given workload and cost function, the aim of a scheduler is to maximize the revenues of the system. There may be many different schedulers, some better suited to specific workloads and cost functions.

The cost function is defined on the execution of a job. It may be totally fictional; that is users do not pay money to have their job executed, but there is almost always some accounting when there is a scarce resource. Different job types can have different cost functions. The abstract notion of money that the system receives by executing a particular job at a particular time allows a scheduler to handle all types of jobs. Instead of ill defined notions such as "gets better service" there is a precise accounting for service. Policy decisions can be reflected through the cost function and not through the scheduler allocation algorithms.

Incentives The more flexible and fully specified a job, the easier it is to schedule. It may be difficult to write a malleable or even a moldable job, and the user may not wish to take the time to uncover a job's resource requirements and speedup curves. So, there should be an incentive to the user to write a malleable program and to provide the system with lots of information about the program execution requirements. There are many types of incentives, but the two most popular choices are reduced cost and improved service; the adage "time is money" may equate the two.

Since the biggest drawback of scheduling of rigid jobs is fragmentation, it should be possible to use malleable jobs to fill in the leftover space. In addition, some fraction of the processors should be reserved for malleable jobs. The exact fractional value is clearly a system specific policy decision.

It seems natural to schedule jobs in the order that they arrive (or by some priority measure). But what resources should be given to flexible jobs? The system might have a different opinion than the user. For example, the user may want fast response time, therefore desire the maximum number of processors, while the system may want to satisfy the maximum number of users and thus allocate the minimal size to a malleable job. Given a 32 processor machine and two jobs, one rigid job requiring 24 processors and the other completely malleable, the system may allocate only 8 processors to the malleable one, thereby giving preferential treatment to the rigid job.

In other words, a reasonable scheduling strategy is to first take care of rigid jobs. Then, any remaining, unassigned processors due to fragmentation can be evenly distributed among the flexible ones. When new jobs arrive, the existing malleable jobs can be squeezed to their minimal size, and the new job allocated processors according to an equipartitioning strategy.

But should malleable jobs always be squeezed to their minimal size? Is it fair to slow down malleable jobs for the sake of executing another rigid job? Will users learn to choose the resource requirements that make their job complete sooner independently of the job's real requirements? For example, malleable jobs may be declared as rigid ones to prevent their interruptions. There is no universally correct answer to these concerns. The tradeoff can only be solved given the goals of the computer installation. A possible approach is to reserve a fraction of the system resources for each type of job. Particular classes can be encouraged and given preferential treatment by varying the fraction of resources. Such a strategy, however, restricts the ability of the scheduler and the policy maker. What happens when there are no rigid jobs? Are the resources wasted?

Using Cost Functions By defining a cost function, there is then something to maximize. For a given workload and cost function, some schedulers will maximize revenues better than others. But no matter the quality of the scheduler, changing the cost function will affect how the users view the machine. This provides flexibility to the system manager and allows exploration of the cost function space.

As a by-product, there is a common goal for theoreticians and practitioners: maximize revenues. Of course the theoretician may assume that there is much that is specified with the job, while the practitioner may have to approximate, infer, or guess at this information.

Note that there are actually two types of cost functions. The one just addressed concerns the when, where, and how a job is scheduled. It reflects the value to the scheduler of executing a job at a particular time. This indirectly affects the quality of service received by individual jobs.

It makes sense also to speak of a potentially different cost function that defines how much the user will have to pay. One can encourage certain types of jobs by charging users differently. It may be necessary to have two different functions since the scheduler gives preference to higher valued jobs whereas users give preference to lower valued jobs.

We assume that changes in the value of a job to the scheduler do not directly affect the offered workload. Of course once users figure out which jobs execute early, they may change the jobs submitted for execution. On the other hand, changes in the cost of a job to the user will quickly change the contents of the offered workload.

We shall ignore the cost to a user and concentrate on the cost function that defines the value of a job to the scheduler.

In what follows, we consider various components of a cost function. Real cost functions are expected to be developed in close collaboration with management personnel of the supercomputer center in order to address policy. The development follows the ideas found elsewhere [19].

The Importance of a Job It may be decided that the amount of parallelism of a job is important. Suppose a job is executed on p processors. When computing the *importance* of a job, a modified value of p may be used, such as $p' = k_0 p$. If $k_0 < 1$ then parallelism is discouraged; if $k_0 > 1$ then it is encouraged. Similarly, the total amount of CPU time used by a job, t, might be deemed important. So another constant is required to scale the time: $t' = k_1 t$.

The policy might be to encourage the use of medium sized parallelism, say 32 processors. The knee of the cost function should therefore be at this preferred number.

Other resource requirements can be similarly scaled in a way to reflect the importance of a job at the given installation. The final importance of a job is then some function of the scaled components. A simple function is addition, e.g. $t' + p'$, or multiplication, e.g. $t' \cdot p'$, but with multiplication, elapsed time or time per processor may make more sense since total time already incorporates the amount of parallelism. The scaled time per processor is simply t'/p.

If a job has a dynamic parallelism profile, then these values can be defined piecewise over periods when the number of processors is fixed, and then added up, as in

$$\text{Importance}(J) = \sum_i \frac{t_i'}{p_i} \, p_i'$$

The Global Affect of a Job Executing one job may preclude the scheduler from executing another, less valuable job, but it may also leave idle resources. Thus there is a second component to a job that must take account for what else is happening in the machine. Let us call this the *affect* of executing a job.

Suppose there are several jobs executing at any one time in a space slicing manner. During this snapshot in time, there are L idle processors. Then one could assign a value of $-\frac{p_i}{P} L$ as the affect of the job during this time slice (i.e. the cost is this factor multiplied by the time).

The affect of a job as defined is bad. The more wasted processors, the more negative the value. This function gives proportional blame for the idle resources based on the size of a job.

Deadlines and Response Time No matter how important is a job, without some accounting of deadlines or response time, there is no reason for a job to be scheduled at a particular time. The scheduler simply waits until it has enough jobs that it can pack together without waste. When there is a deadline associated with a job, then the cost function can reflect some policy. For example, the value of a job may decrease the longer it misses its deadline. A function that has the value continue to decrease (below zero) ensures that each job will eventually be executed[2].

(a)

Job	Type	CPU	Min PEs	Max PEs	Arrival	Deadline
J_0	Rigid	6	6	6	0	1
J_1	Rigid	6	6	6	0	1
J_2	Malleable	10	1	8	0	2
J_3	Malleable	10	1	8	0	2

(b)

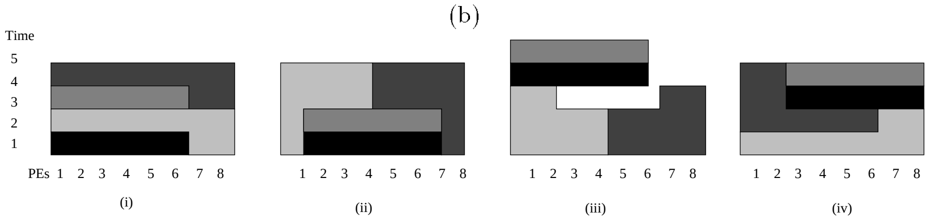

Fig. 3. *Example of scheduling rigid and malleable jobs under different cost functions. (a) characteristics of the jobs in the workload. (b) possible schedules.*

One suggested function [19] is:

$$(\text{deadline} - \text{response time})/\text{Importance}$$

This makes sense once one recalls that our definition of importance is based on the total use of resources. The suggested function therefore scales the delay according to the resources used. The idea is that if a one month job misses its deadline by a day, it is not as bad as a 5 minute job missing its deadline by a day.

Consider the example of four jobs in a machine with 8 processors shown in Fig. 3. They all arrive at time 0 and want immediate service. There are several ways to schedule these four jobs; four examples are shown in the figure, although there are others. Which is the right schedule? The answer is that there is no right or wrong schedule; there is only those that maximize the cost function. Consider

[2] Note that this differs from the strategy of airlines that want to maximize their on-time performance, so that once a flight is delayed by more than 15 min, there is little pressure to minimize the delay. A better analogy is with the construction industry where fines are levied for each month delay in completing a building.

a simple cost function, like the one defined above, and assume that importance is taken to be CPU time. The cost is then

$$\mathrm{Cost}(J) = \frac{\text{deadline} - \text{response time}}{\text{CPU}}$$

With this function, schedule (i) gives the best score since shorter jobs have a some what worse penalty for missing their deadlines. Increasing the importance of the smallness of a job, say by using an importance function that squares the computation requirements: i.e. $1/\mathrm{CPU}^2$, configuration (ii) is the best. If bigger jobs are more important, then schedule (iii) would be chosen, assuming that fragmentation or low system utilization is less critical than missing deadlines. Finally, we note that schedule (iv) would not be chosen since the malleable jobs complete at the same time as in schedule (i) and the rigid jobs finish later. Extending the deadlines of the rigid jobs would make the two schedules equivalent.

We note that the types of simple cost and importance functions just discussed, often results in malleable jobs that start execution with few processors and then expand their parallelism as their deadlines approach.

7 Conclusions

Job schedulers should support a workload of all types of jobs, with a varying amount of information concerning their resource needs specified either at job submittal time or during job execution. The system manager should be able to define a cost function that captures the policy goals of the computation center. The question is how to get there from where we are today.

First, it is important to continue the investigation into current workloads to get a better understanding of the resource requirements of parallel applications as well as how resources affect their performance. Some of this can be gotten from runtime monitoring and gathered from job statistics at the supercomputer centers. Another important input is from the designers of parallel programming languages and programming environments, who may come up with new requirements and desiderata.

Second, many on-line schedulers use amortized cost analysis to make decisions that are within a constant factor of the optimal, off-line algorithm. But, without an explicit statement of what it is that the scheduler is trying to maximize, there can be no way to evaluate the success of an on-line algorithm.

Finally, it is important to keep the assumptions made by different researchers in mind. In particular, it is necessary to always check to what degree the results depend upon these assumptions, and to keep a lookout for ideas that may be applicable to a wider class of jobs and systems than envisioned initially.

References

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

2. 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.
3. J. M. Barton and N. Bitar, "*A scalable multi-discipline, multiple-processor scheduling framework for IRIX*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 45–69, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
4. P. Brinch Hansen, "*An analysis of response ratio scheduling*". In *IFIP Congress, Ljubljana*, pp. TA–3 150–154, Aug 1971.
5. N. Carriero, E. Freedman, D. Gelernter, and D. Kaminsky, "*Adaptive parallelism and Piranha*". *Computer* **28(1)**, pp. 40–49, Jan 1995.
6. 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.
7. S-H. Chiang and M. Vernon, "*Dynamic vs. static quantum-based parallel processor allocation*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
8. 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.
9. D. Das Sharma and D. K. Pradhan, "*A fast and efficient strategy for submesh allocation in mesh-connected parallel computers*". In *IEEE Symp. Parallel & Distributed Processing*, pp. 682–689, Dec 1993.
10. M. Devarakonda and A. Mukherjee, "*Issues in implementation of cache-affinity scheduling*". In *Proc. Winter USENIX Technical Conf.*, pp. 345–357, Jan 1992.
11. 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.
12. D. G. Feitelson, "*Packing schemes for gang scheduling*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
13. D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems*. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.
14. D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
15. D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.
16. D. G. Feitelson and L. Rudolph, "*Evaluation of design choices for gang scheduling using distributed hierarchical control*". *J. Parallel & Distributed Comput.*, 1996. to appear.
17. 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.
18. D. G. Feitelson and L. Rudolph, "*Parallel job scheduling: issues and approaches*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–18, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

19. M. Frank, V. Lee, W. Lee, K. Mackenzie, and L. Rudolph, "*An online scheduler respecting job cost functions for parallel processors*". Manuscript in preperation, M.I.T. Cambridge, MA, 1996.
20. J. Gehring and F. Ramme, "*Architecture-independent request-scheduling with tight waiting-time estimations*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
21. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM 3 User's Guide and Reference Manual*. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1994.
22. B. Gorda and R. Wolski, "*Time sharing massively parallel machines*". In *Intl. Conf. Parallel Processing*, Aug 1995.
23. 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.
24. R. L. Henderson, "*Job scheduling under the portable batch system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 279–294, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
25. A. Hori et al., "*Time space sharing scheduling and architectural support*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 92–105, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
26. A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda, "*Implementation of gang-scheduling on workstation cluster*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
27. S. Hotovy, "*Workload evolution on the Cornell Theory Center IBM SP2*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
28. N. Islam, A. Prodromidis, and M. Squillante, "*Dynamic partitioning in different distributed-memory environments*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
29. Y. A. Khalidi, J. Bernabeu, V. Matena, K. Shirriff, and M. Thadani, "*Solaris MC: a Multi Computer OS*". In *Proc. USENIX Conf.*, Jan 1996.
30. A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C-L. Wang, "*Heterogeneous computing: challenges and opportunities*". *Computer* **26(6)**, pp. 18–27, Jun 1993.
31. P. Krueger, T-H. Lai, and V. A. Dixit-Radiya, "*Job scheduling is more important than processor allocation for hypercube computers*". *IEEE Trans. Parallel & Distributed Syst.* **5(5)**, pp. 488–497, May 1994.
32. D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 295–303, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
33. W. Liu, V. Lo, K. Windisch, and B. Nitzberg, "*Non-contiguous processor allocation algorithms for distributed memory multicomputers*". In *Supercomputing '94*, pp. 227–236, Nov 1994.
34. 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.

35. 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.
36. T. D. Nguyen, R. Vaswani, and J. Zahorjan, "*Parallel application characterization for multiprocessor scheduling policy design*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
37. T. D. Nguyen, R. Vaswani, and J. Zahorjan, "*Using runtime measured workload characteristics in parallel processor scheduling*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
38. J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.
39. J. D. Padhye and L. W. Dowdy, "*Preemptive versus non-preemptive processor allocation policies for message passing parallel computers: an empirical comparison*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
40. E. W. Parsons and K. C. Sevcik, "*Multiprocessor scheduling for high-variability service time distributions*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 127–145, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
41. J. Peterson and A. Silberschatz, *Operating System Concepts*. Addison-Wesley, 1983.
42. J. Pruyne and M. Livny, "*Managing checkpoints for parallel programs*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
43. J. Pruyne and M. Livny, "*Parallel processing on dynamic resources with CARMI*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 259–278, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
44. 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.
45. E. Rosti, E. Smirni, G. Serazzi, and L. W. Dowdy, "*Analysis of non-work-conserving processor partitioning policies*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 165–181, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
46. K. C. Sevcik, "*Application scheduling and processor allocation in multiprogrammed parallel processing systems*". *Performance Evaluation* **19(2-3)**, pp. 107–140, Mar 1994.
47. 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.
48. J. Skovira, W. Chan, H. Zhou, and D. Lifka, "*The EASY - LoadLeveler API project*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
49. 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.
50. S. Thakkar, P. Gifford, and G. Fielland, "*Balance: a shared memory multiprocessor system*". In 2nd *Intl. Conf. Supercomputing*, vol. I, pp. 93–101, 1987.

51. J. Torrellas, A. Tucker, and A. Gupta, "*Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors*". *J. Parallel & Distributed Comput.* **24(2)**, pp. 139–151, Feb 1995.
52. 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.
53. M. Wan, R. Moore, G. Kremenek, and K. Steube, "*A batch scheduler for the Intel Paragon MPP system with a non-contiguous node allocation algorithm*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
54. F. Wang, M. Papaefthymiou, M. Squillante, L. Rudolph, P. Pattnaik, and H. Franke, "*A gang scheduling design for multiprogrammed parallel computing environments*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
55. J. Zahorjan and C. McCann, "*Processor scheduling in shared memory multiprocessors*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 214–225, May 1990.