Parallel Job Scheduling: Issues and Approaches

Dror G. Feitelson[1] and Larry Rudolph[2*]

[1] IBM T. J. Watson Research Center
P. O. Box 218, Yorktown Heights, NY 10598
feit@watson.ibm.com

[2] Institute of Computer Science
The Hebrew University, 91904 Jerusalem, Israel
rudolph@theory.lcs.mit.edu

Abstract. Parallel job scheduling is beginning to gain recognition as an important topic that is distinct from the scheduling of tasks within a parallel job by the programmer or runtime system. The main issue is how to share the resources of the parallel machine among a number of competing jobs, giving each the required level of service. This level of scheduling is done by the operating system. The four most commonly used or advocated techniques are to use a global queue, use variable partitioning, use dynamic partitioning, and use gang scheduling. These techniques are surveyed, and the benefits and shortcomings of each are identified. Then additional requirements that are not addressed by current systems are outlined, followed by considerations for evaluating various scheduling schemes.

1 Introduction

Parallel supercomputers are an expensive, scarce resource that often must be shared among a large community of users. But, the successful scheduling schemes for uniprocessors do not readily translate to address the challenges posed by parallelism. This paper examines some of these challenges, surveys current solutions, and points to future areas of research.

Although parallel computers are more difficult to use effectively than their sequential counterparts, many people are willing to pay the price because their applications need the additional resources. An application executed on a parallel computer can potentially complete in a shorter time, and make use of much larger aggregate cache capacity, physical memory size, and I/O bandwidth. Of course, when an application must compete for a share of these resources, the payoff becomes far less attractive.

Resource allocation to competing jobs is done by the system scheduler. Scheduling on a parallel computer is complex since it involves scheduling over two dimensions, time and space, and at two levels, jobs and threads. There are a number of different approaches to job scheduling. The processors can be partitioned to allow several jobs to execute in parallel. In addition, when a job is running, its constituent threads may also need to be scheduled.

---

* Currently on sabbatical at the MIT Laboratory for Computer Science.

Discussing job scheduling on a parallel computer is hard because it means so many different things to so many different people. The large variety of parallel programming languages, parallel computer architectures, and parallel operating systems, means that there is no one ideal scheduling strategy. The problem is exacerbated by the large variety of vague goals. For example, it is not clear if the response time of individual jobs, i.e. the makespan, should take preference over the overall system throughput. Also, it is hard to compare actual schedulers in commercial products handling real workloads with academic studies of idealized schedulers exercised by synthetic workloads.

The scheduling requirements depend on the environment. For parallel computers used as dedicated machines for solving single large applications and benchmarks or allocated to a small group of researchers for evaluation purposes, schedulers are not so important. But when large parallel computers are to be shared by a wide range of users, some with large batch jobs and others with smaller interactive jobs, proper scheduling and resource allocation becomes a critical issue. Very few people are willing to wait a month to get one hour on a supercomputer; the application usually could be finished sooner on a dedicated workstation. It may be necessary to satisfy individual job needs at the expense of total system throughput.

This paper first outlines the range of objects that are to be scheduled. A survey of the scheduling approaches is then presented. The paper concludes with several comments about the requirements for job scheduling and the evaluation of scheduling schemes. Throughout, it also serves to put the other papers in this volume in context.

2 Background

A major problem in talking about "scheduling on parallel machines" is that this expression means different things to different people. We therefore start by distinguishing among the different classes of objects that are to be scheduled on parallel systems. The major objects are *jobs* and *threads*. Jobs are autonomous programs that execute in their own protection domain. One job should not be allowed to access the memory of another job nor interfere with the other job's message-space. A job may consist of many threads, i.e. a parallel job, and may be batch or interactive. We use the term "thread" to specify the thing that can execute in parallel in a parallel program. It is synonymous with terms like "chore" or "activity" as sometimes used by others.

We prefer thread rather than "task", as this latter term has been used with innumerable conflicting meanings in the literature. Furthermore, we reserve "process" to denote an autonomous execution unit. Thus in uniprocessors the terms job and process are synonymous. In parallel systems, if the operating system is aware of the fact that multiple executing entities are parts of the same application, we call them threads and the whole thing together is a job. If, on the other hand, the interacting entities are independent as far as the operating system is concerned, we call them processes. For the operating system, each of these pro-

		Communication pattern	
		Static	*Dynamic*
Thread creation	*Static*	DAG	SPMD, HPF
	Dynamic	Dataflow Functional	Nested parallelism Unix fork-join

Fig. 1. *A simple classification of parallel programming styles. The range is actually much more complex than what is shown here.*

cesses is a separate job, whereas for the user they collectively represent a single job.

While a parallel job consists of threads whose execution is specified in a parallel program, there is no single universally agreed upon parallel programming language. Figure 1 gives a very broad classification of parallel programming languages. The threads can be static or dynamic and the communication between them can be static or dynamic. For example, in a Single Program, Multiple Data (SPMD) language, the number of threads is fixed and is often equal to the number of processors, but the communication patterns between the threads can change dynamically during the execution.

Threads can be very light-weight, as in the case of dataflow or functional programming with very limited communication — data is passed to the thread upon its creation, and the thread passes on data when it terminates. In this case, there are few restrictions on where the thread can execute. On the other hand, threads can be heavy-weight involving lots of computation and many communications with other threads. In a message-passing systems, such threads often cannot be moved once begun since most message-passing libraries use physical processor addresses for messages to avoid a level of indirection for each communication.

The machine architecture thus has a serious impact on the scheduler. Figure 2 gives a very coarse classification. In machines in which processors are assembled in a restricted topology such as a mesh or hypercube, jobs are assigned to specific partitions of the machines, e.g. subcubes. In symmetric multiprocessors with

	Shared memory	*Message passing*
Complete network	UMA SMPs SGI Challenge	IBM SP2 NOWs
Mixed	KSR	TMC CM-5
Visible topology	NUMA Cm*, DASH	Intel Paragon Cray T3D Hypercubes

Fig. 2. *A simple classification of parallel machine classes. "Mixed" denotes cases that have characteristics of both complete networks and topological restrictions. (SMP = Symmetric Multiprocessor; UMA = Uniform Memory Access; NUMA = Non-Uniform Memory Access)*

uniform access to shared memory, it is easy to stop a thread and reschedule it on any other processor. Some machines offer a mixture: the KSR is billed as an UMA machine, due to the automatic migration of data to wherever it is needed, but has a clustered structure. The CM-5 offers a complete-network view in terms of message passing, but can only be partitioned according to subtrees of its fat-tree network.

The following subsections review some of the classes of scheduling. Each class may have its own goal or be directed at a particular class of machines or languages.

2.1 Static Scheduling

In order to understand the theoretical limitations of scheduling, a restricted version of parallel programs is considered in which there is nearly complete knowledge of the computation. An application is modeled as a directed acyclic graph (DAG) where each node denotes a task and directed edges or arcs denote dependencies among the tasks: the task at the origin of the arc must complete before the task at its end may commence. These dependencies typically represent data transfers, i.e. the task at the sink of a directed arc uses some data values that are generated by the task at the source of the arc. Both nodes and arcs may have weights attached to them. The weight of a node denotes the required amount of computation, while the weight of an arc denotes the amount of data that is transferred.

When the system is modeled as a set of homogeneous processors, that can each execute one task at a time, the objective is usually to minimize the makespan [31]. In some cases, communication costs are also explicitly modeled. Thus if two neighboring tasks are assigned to different processors, the second one is delayed by an amount of time proportional to the weight of the arc between them, but if they are assigned to the same processor, there is no such delay.

Many restricted cases of DAG scheduling have been shown to be NP-complete, meaning that it is impractical to look for an optimal schedule. This is true even for very restricted special cases, e.g. when communication is free and all tasks have unit execution times [48, 18].

Many parallel programming styles are close to the DAG model, although with substantially less information. Dataflow, functional programming, and other side-effect free programming models can be viewed as DAGs. Consequently, various heuristics have been developed to approximate good schedules. One approach is to assign priorities to nodes in the DAG according to their distance from the termination node. Another concentrates on the critical path in the DAG. When communication costs are taken into account, the main issue is how to partition the DAG into clusters of nodes, and then schedule these clusters on processors. This is a compromise between two conflicting forces: keeping nodes separate increases parallelism at the cost of communication, whereas clustering them causes serialization but saves on communication [29, 34]. If the interconnection network has some specific topology, this can also be taken into account [5].

2.2 Scheduling in the Runtime System

While DAG scheduling has generated a large body of research, much of it cannot be applied to programming environments that use other representations. In many such environments, the programmer is actually shielded from the details of how the program is mapped onto the machine. The details are then handled by the environment's runtime system.

One example of this approach is the use of thread packages [33]. The application is structured as a set of interacting threads. The environment supplies functions for thread creation, synchronization, and termination. The runtime system is responsible for implementing these functions when they are called.

Another example is the use of parallelizing compilers. Parallelizing compilers usually extract parallelism from the loops of sequential programs. At runtime, the different loop iterations are scheduled for execution on distinct processors [28]. A typical approach is to assign decreasing chunks of iterations to the different processors, so as to balance the load on one hand while avoiding extra coordination on the other. Again, the programmer does not have to worry about how this is done.

2.3 Scheduling in the Operating System

Scheduling by the programmer or runtime system aims to satisfy the individual needs of the program in question. But when multiple programs must co-exist in the same system, it is necessary to balance the needs of the individual with those of the community in general. Such considerations and mechanisms lie in the realm of the operating system.

The fact that the mechanisms and considerations of scheduling at the operating system level are sometimes very similar to those applied at the application and runtime level has, in our opinion, hindered progress — it is not clear who is in charge of the problem. The spectrum of opinions range from those who claim that most scheduling should be done at the application level, keeping the operating system out of the loop [2], to those that prefer to let the operating system do all the work [4]. Interrupts, context switches, and memory swapping are overheads that can be very costly on a parallel supercomputer. However, as parallel systems become more commonly used, there is growing recognition of the need for resource management at the operating system level. For example, this is where jobs can be isolated from each other, giving each the fiction of a dedicated (virtual) machine, and where accounting functions should be carried out. It is this class of scheduling that is addressed in the rest of this paper.

Many requirements make scheduling difficult. For example, it is important that different jobs execute in different protection domains. That is, one job cannot access the memory or message space of another job. A second difficulty is that on many machines, partitions have to match the topology of the machine, e.g. only subcubes can be used on a hypercube. Yet another problem is that in many message passing systems, source and destination addresses are hardwired so that a thread cannot be easily migrated between processorss. Therefore multiple different approaches are used in different systems, as described in Section 3.

2.4 Administrative Scheduling

Finally, some scheduling decisions are made at an administrative rather than at a technical level. For example, the head of a computing facility may decide that a certain project should get exclusive access to a certain machine for three weeks, in order to achieve some project goals. Or, it may be the policy to encourage short massively parallel jobs instead of longer jobs of smaller parallelism. Or, what should be the ratio of batch jobs to interactive ones. While no research is done on this type of activity, it undeniably has a significant effect on many users. Mechanisms that enable such policy control are of interest. Finally, there is the issue of accounting and how to charge users for the computational resources they consume.

3 Current Approaches

In order to multiprogram a parallel machine, the operating system has to decide when to execute each job and on which processors. In general, it is possible to use time slicing (jobs share the use of the same processors), space slicing (each processor is allocated to a specific job until its completion), or a combination of both. Time slicing may not always be a viable option. On some systems it may be costly or even impossible to switch protection domains, or applications may use massive amounts of physical memory that could take many minutes to page in and out. Similarly, space slicing may not be reasonable if the network is not partitionable, protection cannot be assured, or applications require the full parallelism of he machine. Since many parallel machines are superlinear in their cost as a function of processors, extensive use of partitioning does not make economic sense. An interesting observation is that the two sharing schemes are largely orthogonal, so various combinations can be tried. Indeed, a remarkable variety of approaches have been devised over the years (see Fig. 3) [11].

The following subsections survey the four most popular approaches: global queue, variable partitioning, dynamic partitioning with two-level scheduling, and gang scheduling. Of these, two emphasize the use of space slicing (variable and dynamic partitioning), and two the use of time slicing (global queue and gang scheduling). For each, we briefly identify where it is applicable (usually depends on the machine architecture), and highlight its strengths and weaknesses.

3.1 Global Queue

Perhaps the simplest way to implement a parallel operating system scheduler is to run a copy of a uniprocessor system on each node, while sharing the main data structures, specifically the run queue. Threads that are ready to run are

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

Fig. 3. *Examples of systems that use different combinations of space slicing and time slicing (updated from [11]).*

placed in this queue. Processors pick the first thread from the queue, execute it for a certain time quantum, and then return it to the queue. This approach is especially common on small-scale bus-based UMA shared memory machines, such as Sequent multiprocessors [46] and SGI multiprocessor workstations [3], and is also used in the Mach operating system [4].

The main merit of a global queue is that it provides automatic load sharing. No processor is idle if there is any waiting thread in the system. However, this comes at a price. One problem is contention for the global queue, which grows with the number of processors. Another is that threads will typically execute on different processors each time they are scheduled. As a result, threads cannot stash data in local memory, and their cache state is wiped out with each rescheduling. This effect is countered to a certain degree by *affinity* scheduling, which attempts to re-schedule threads on the same processors they used before [47]. Alternatively, it is possible to provide global load balance with local queues and occasional migration to overcome any imbalance [39].

A third problem with a global queue is that the threads in a single application are scheduled in an uncoordinated manner. This may be fine if the threads do not interact with each other. But in many parallel programs the threads do interact and synchronize with each other. If the interaction is at a high rate (i.e. a fine granularity), the lack of coordination in the scheduling implies that often the interacting partners will not be executing at the same time. Therefore the interactions will not be able to proceed, inducing extra context switches and overhead [16].

Finally, scheduling from a global queue has the interesting property that the service a job receives is proportional to the number of threads that it spawns. It is subject to debate whether this is good or bad. On one hand, this is a natural way for jobs that require more computation to get the necessary resources. On the other hand, it impairs fairness and might be susceptible to user counter measures. These problems can probably be solved by suitable accounting practices.

3.2 Variable Partitioning

Another simple approach to multiprogramming parallel machines is to partition the processors into disjoint sets and execute each job in a distinct partition. There are different approaches to partitioning, and the following taxonomy is common (see Fig. 4) [11]: *Fixed* partitioning is when the partition sizes are set in advance by the system administrator. Repartitioning requires a reboot. *Variable* is when the set of nodes are partitioned according to user requests when jobs are submitted. That is, a large partition can be divided into smaller partitions to allow several small jobs to execute in parallel. Partitions are fused when the jobs terminate. *Adaptive* is when the partitions are automatically set by the system according to current load when the job is submitted. *Dynamic* is when the size can change at runtime to reflect changes in requirements and load. Of these, the most popular are variable and dynamic partitioning.

	Parameters taken into account		
Scheme	*User request*	*System load*	*Changes*
Fixed	no	no	no
Variable	yes	no	no
Adaptive	yes	yes	no
Dynamic	yes	yes	yes

Fig. 4. *A taxonomy of partitioning schemes.*

Variable partitioning is popular on distributed memory machines, and is used on Intel and nCUBE hypercubes, on the IBM SP2, on the Intel Paragon, on the Meiko CS-2, and on the Cray T3D. The non-uniform memory access feature or the lack of a shared address space of these machines make the previous global queue approach impractical. The advantages of running a job on a dedicated partition with variable partitioning is that this gives a good approximation of a dedicated machine. It places the needs of individual jobs over that of the system. The program has control over the distribution of data in the memories of the processors on which it runs. There is no cache interference, and no operating system overheads. Depending on the system topology, there may also be no network interference (in hypercubes, for example, each subcube uses a disjoint set of links, but in systems that use a multistage network, such as an IBM SP2, some links may be shared by different partitions).

Of course, variable partitioning also has some disadvantages. These disadvantages stem from the possible mismatch between the available processors and the user requests. Two distinct problems may occur. One is fragmentation, which occurs when the available (free) processors are insufficient to satisfy the requests of any submitted jobs, so these processors are left idle. The other problem is that submitted jobs can be queued for a long time until the requested processors become available. This is especially severe when big jobs are involved: a job that requires all the processors cannot run as long as other jobs are running, and once it does run no other job can commence. Even if allocated processors are momentarily idle due to an I/O operation, they cannot be given to another job. Therefore variable partitioning is more suitable for batch processing than for interactive work. Indeed, the inability to run a program when desired sometimes causes significant user frustration, especially during the program development phase. To reduce this effect, sophisticated batch systems are designed [24, 21, 27].

3.3 Dynamic Partitioning with Two-Level Scheduling

One way to reduce the waiting time of queued jobs is to prevent jobs from monopolizing too many processors when the system is heavily loaded. This is done by adaptive partitioning schemes, and several ideas about how to allocate the processors have been proposed [38]. The problem with adaptive partitioning, however, is that once a number of processors are allocated to a job this number is fixed until the job terminates. It does not change in response to load changes, and cannot be changed to reflect changes in the degree of parallelism in the job.

Changes in allocation during execution are provided by dynamic partitioning. To support such behavior, applications are required to use a programming model that can both express changes in requirements and handle system-induced changes in allocation. This is typically done by using a workpile model, where the work to be done is represented as an unordered pile of tasks or chores, and the computation is carried out by a set of worker threads, one per processor, that take one chore at a time from the workpile. This decouples the work (the chores) from the agents of computation (the workers), and allows for adjustment to different numbers of processors by changing the number of workers. The result is a two-level scheduling scheme: the operating system deals with the allocation of processors to jobs, while the applications handle the scheduling of chores on those processors.

Dynamic partitioning with two-level scheduling is probably the most studied parallel scheduling scheme, and has repeatedly been shown to be superior to other schemes [30, 20] either by analytic means or through simulation with synthetic workloads. This is due to a number of factors:

- There is no loss of resources to fragmentation.

- There is no overhead for context switching, except that for redistributing the processors 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 thread blocking incurs little overhead.
- The degree of parallelism provided to each job is automatically decreased under heavy load conditions, leading to better efficiency. Programming environments may shelter the application programmer from having to deal with such dynamic changes explicitly.

However, dynamic partitioning does have its drawbacks. It does not support popular programming styles such as the SPMD (used by MPI) and dataparallel (used by HPF) models. Moreover, it could lead to extensive queueing if sufficient processors are not available. And the overheads for repartitioning, which include switching processors from one protection domain to another, may negate the benefits that were expected [44].

In addition, dynamic partitioning requires extensive coordination between the operating system and the application's internal scheduler, so as to handle the changes in processor allocation efficiently [2]. For example, if a processor is taken away from an application, the whole application might deadlock if the thread running on that processor happened to be holding a lock. To prevent such scenarios, the application's runtime system should be notified so it can take appropriate action. This means that the operating system and the programming environment runtime system must be designed together, limiting their portability and ability to work in other environments. A possible solution is to only change the processor allocation at certain points in the program, e.g. at the beginning of a new parallel loop [50].

As a result of the above shortcomings, dynamic partitioning has so far been used only to a very limited degree, mainly in the context of running parallel jobs on networks of workstations, where the workstation must be returned to its owner when required [8, 36].

3.4 Gang Scheduling

The only way to guarantee interactive response times is via time slicing. However, if done in an uncoordinated manner, as with a global queue, this can lead to large inefficiencies. Rather, the context switching should be coordinated across the processors. Thus all the threads in the job will execute at the same time, allowing them to interact at a fine granularity. In addition, the threads can be mapped permanently to processors, thus allowing them to make use of local memory and maybe to benefit from sustained cache state. This solution is completely general, and works for any programming model. In fact, it decouples the application from the operating system. It is used by some vendors, e.g. the CM-5 from Thinking Machines [26], the IRIX system on SGI multiprocessors [3], the Intel Paragon [23], and the Meiko CS-2, and has also been studied in academia and research prototypes [32, 13, 19, 22].

An interesting variant of gang scheduling is based on the observation that coordinated scheduling is only needed if the job's threads interact frequently [16]. Therefore the rate of interaction can be used to drive the grouping of threads into gangs [17, 43]. Other variants include *coscheduling*, which attempts to schedule a large subset of the gang if it is impossible to schedule all the threads at once [32], and *family scheduling*, which allows more threads than processors and uses a second level of internal time slicing [7].

The price of gang scheduling is that the overall system performance is not always optimal (although it does favor individual jobs). There is some interference in the cache, and overhead for the context switching. In addition, there may be some processor fragmentation. However, due to the time slicing, its effect is less severe than in variable partitioning [14, 15]. Indeed, most studies find that gang scheduling is nearly as efficient as dynamic partitioning [9].

4 The Requirements

Literally hundreds of papers have been written about job scheduling in parallel systems (see [11]). However, in many respects, we are at the beginning of the beginning. The split between the academic focus on dynamic partitioning and the practical use of variable partitioning and gang scheduling is a symptom of disagreement even about the fundamental questions.

System design is (or at least, should be) driven by requirements. Thus one source of disagreement about the merits of different designs is controversy over exactly what are the requirements that need to be satisfied. Some requirements reflect first principles, e.g. that jobs must be isolated from each other, that an accounting service be provided, that the scheduler be aware of the fact that multiple processes belong to the same job (and all should be killed if one terminates abnormally), and that systems where processors have somewhat different configurations have to be supported efficiently [40, 37]. In addition, it is important to understand the workload that must be supported.

Obviously, the workload is related to the ultimate use of the parallel machine. It seems that parallel supercomputers are used for three main reasons:

- **Short response time.** Parallelism enables a computation to complete in less time, and this may make a qualitative rather than just quantitative difference. For example, reducing the time to compute a 3-day forecast from a week to a day makes it relevant. At the interactive level, running a circuit simulation in a matter of minutes rather than hours allows an engineer to iteratively tinker with parameters, leading to increased productivity.
- **Large resource requirements.** Parallel supercomputers allow more resources to be harnessed to solving the same problem than other systems. The resources in question can be both compute power and memory. An example is the GF-11, which took about a year to perform a calculation aimed at verifying the theory of Quantum Chromo-Dynamics. Of course paging can provide large virtual memory, but for large parallel supercomputers the time penalty for paging significant amounts of memory is enormous.

– **Because it's there.** Parallel processing is intriguing and challenging because it is inherently different from the sequential human stream of consciousness (as opposed to the underlying structure of the brain, which is massively parallel). This fascinates and attracts many people. Moreover, in the last few years, it has been the "in" thing to do. This is an important factor, because users are human beings who do not always perform a full cost/benefit analysis before choosing a course of action. The fact that many vendors report that entry-level systems account for a large fraction of their sales, as opposed to large-scale systems, bears testimony to the fact that many parallel systems are not used to solve extremely large or time-sensitive problems, as implied by the previous two categories.

In addition, there are secondary reasons such as development and debugging of parallel application. These activities are important because their requirements are typically different from those of the application being developed: less resources are needed, and an interactive response time is crucial.

So what does all of this mean for parallel scheduling? Above everything else, it means that arguments to the effect that parallel supercomputers should only be used in batch mode for large problems are wrong. There is a very diverse set of requirements, spanning a spectrum from rather small interactive jobs, through large but time-sensitive jobs, to very large and not-time-sensitive jobs that can be satisfied by a batch system [12]. The problem is that these different classes may be present on the same system at the same time, and each must be serviced according to its unique requirements. This wide distribution of requirements implies that some sort of time slicing has to be used [35].

An important aspect of workload requirements that is often overlooked is memory and secondary storage. In fact, many respectable commercial parallel systems (e.g. the CM-5) do not support memory paging: applications are required to fit into physical memory. When a machine is dedicated to executing a very large application for a very long time, the overhead of a fancy scheduler is not worthwhile and, similarly, the overhead of a paging system may not be worthwhile. If the machine is shared among a number of applications, their combined requirements are required to fit into memory. Many proposed scheduling schemes make the same assumption, and fall apart once memory considerations are introduced. The simple solution adopted by some recent systems, such as the IBM SP2 and Meiko CS-2, is to have independent paging on each node. This has the undesirable side effect that processes can be blocked asynchronously for relatively long periods, preventing fine-grain synchronization and communication from taking place [49]. To quote a somewhat overused phrase, "there must be a better way".

The main thing to remember is that scheduling is not an isolated issue. Scheduling is but one service provided by the operating system. The solution to the scheduling problem must be integrated with solutions to other problems, e.g. memory management and I/O. The different parts of the system must work together to create a cohesive whole in a way that makes sense. To illustrate the point, here is a list of simple ideas, for the simple case of an SPMD or dataparallel model of computation:

- The system can use collective I/O operations as collective context-switching points for gang scheduling.
- The system can use a page fault by one process to trigger a prefetch for the same page in other processes.
- Swapping can be done by collective I/O, where the program images on all processors are downloaded to multiple disks in parallel.

Other ideas have also been proposed [10, 41]. Regrettably, many systems do not take such a comprehensive approach. The reason is that creating a useful cohesive system is an order of magnitude[3] harder than creating a working prototype, and that is already hard enough. However, the fact that a cohesive system must be the ultimate goal should not be forgotten, together with the implications of this goal, in terms of restrictions on the scheduling scheme so that it will not conflict with other sub-systems.

In this context, it is fair to note that requirements go both ways. If the users require support for both interactive and batch jobs, possibly with very large memory demands, the scheduling subsystem may respond that such support can only be achieved if enough I/O capabilities are provided to perform swapping at a high rate. This is not always the case, as installations skimp on I/O resources to reduce system costs, leaving the scheduler in a no-win situation. The challenge is to design the system so that it can take advantage of such resources if they are available, and to quantify the effect of doing without them if they are not, so users can make an informed choice [1]. Ignoring the issue is not a viable alternative.

Finally, one should remember that the operating system is there to serve applications, and through them, to serve users, and the client is always right. Thus the operating system should provide opportunities, not impose restrictions. In the context of scheduling, it does not do to support only certain models of computation, and limit users to their use. It is hard enough to program parallel machines effectively without restricting the available tools and idioms. In addition, it is important to provide consistent and predictable service, including provisions for fairness and control [45].

5 Evaluation

Designs must be evaluated carefully to gauge their effectiveness. For evaluation to be meaningful it should measure something meaningful. In system design, the meaningful metric is adherence to the requirements, i.e. support for the expected workload. This has two aspects. The first is functionality (e.g. interactive

[3] Brooks estimates a factor of three for each of the two main hurdles, regardless of the order in which they are undertaken; these are (1) system integration and (2) creating product-quality, tested, documented, and maintainable software [6].

response time and virtual memory). The second is performance (e.g. low average response time and high throughput)[4].

The most precise form of evaluation is to implement a full system and measure its performance when used in a production environment. However, this is not a realistic option in many cases, and misses the whole point of being able to gauge the performance implications of a certain design without the expense and delay involved in a full implementation. This does not mean that measurements of real systems are not important — they are important, but not as a design tool, rather as a tool to assess the need for additional improvements and as a yardstick for other systems.

Evaluation of designs prior to implementation therefore depends on simulation and analysis. Simulation often has the advantage that it can provide a more accurate characterization of the system, because any details that are felt to be important can be simulated. Analysis has the advantage that it may lead to equations that provide a concise description of system behavior as a function of its parameters.

The quality of the results of both simulation and analysis hinges on the quality of the inputs that are used. In particular, the workload model must faithfully represent the workload that will be imposed on the real system. Regrettably, there is very little data about real systems, and far too many papers include the phrase "due to lack of any real data, we assume...". This again points out the utility of measurements of real systems, albeit not measurements of the system performance per se but rather measurements of user behavior [12].

The usefulness of the results obtained from simulation or analysis depends on the metrics that are used. Amazingly little work has gone into defining and evaluating metrics. Even the straightforward metrics such as average response time and throughput have problems. With response time, the question is whether to use absolute values (which gives larger weight to large jobs) or to normalize the response time by the amount of work that is done, and use the slowdown as the metric. Throughput and utilization actually depend on the arrival process more than on the system itself (unless it becomes saturated). The most meaningful metric also depends on the type of system. For example, response time is the most important metric for interactive systems, and is largely meaningless for batch processing (assuming there is no starvation). On the other hand, makespan can be used to gauge the effectiveness of a batch system (which often operates in an off-line mode), but is useless for interactive systems. Finally, when combining the results of multiple experiments, care must be taken to use the correct method for averaging [42].

Instead of using multiple independent metrics, it has sometimes been proposed to combine them into a derived metric. For example, "power" is loosely defined as the throughput divided by the response time, so it goes up when throughput goes up or when response time goes down [25]. However, it is not clear that this simple equation captures the relative importance of the two orig-

[4] Note that we make a distinction between the qualitative requirement of interactive response time and the quantitative measure of low average response time.

inal metrics. Maybe a more complicated equation would give better predictive power.

Finally, there is a whole range of metrics that are not used because they are hard to measure and quantify — the metrics related to user satisfaction. The hidden assumption is that user satisfaction is linearly correlated with the measured metrics, such as average response time. But this is not necessarily the case. User perception of the quality of service may be different from that predicted by simple algebra. For example, the difference between response times of 10 seconds and 100 seconds can be very meaningful, because the first is barely interactive whereas the latter is in that inconvenient range where you cannot work continuously but also cannot go to get coffee. The difference between 10 minutes and 100 minutes may be less meaningful. For humans, important metrics are "below a couple of seconds" and "lack of surprises" (which can be translated as low variability).

6 Conclusions

The issue of job scheduling has suffered due to the common lack of distinction between job scheduling by the operating system and static or dynamic scheduling within an application by the programmer or runtime system. Nevertheless, hundreds of papers about parallel job scheduling have been published. Despite this large body of work, we seem to be at the beginning of a long road, and much remains to be done. Specific topics that cry out for further research include:

- Integration of scheduling with other system services, and most notably, with memory management. This includes requirements that the scheduling sub-system places on the hardware and I/O sub-system, and interactions between the different sub-systems.
- Better characterization of the workloads that are found on general purpose parallel systems. This includes the jobs themselves (how many processors they use, how much memory they need, how long they run), and issues such as the arrival process.
- Evaluation of alternatives that is both fair (i.e. use the best possible implementation of each scheme) and informative (i.e. use the same workloads and metrics, and make it the right metrics).

Of course, there is also ample place for more work on scheduling schemes for both common parallel systems and emerging new types of systems, such as multithreaded architectures [1] and NOWs [36]. The future will tell which of these survive the ultimate test, that of satisfying real users.

References

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

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.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
4. D. L. Black, "*Scheduling support for concurrency and parallelism in the Mach operating system*". *Computer* **23(5)**, pp. 35–43, May 1990.
5. S. H. Bokhari, "*On the mapping problem*". *IEEE Trans. Comput.* **C-30(3)**, pp. 207–214, Mar 1981.
6. F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1975.
7. 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.
8. N. Carriero, E. Freedman, D. Gelernter, and D. Kaminsky, "*Adaptive parallelism and Piranha*". *Computer* **28(1)**, pp. 40–49, Jan 1995.
9. R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "*Scheduling and page migration for multiprocessor compute servers*". In 6th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 12–24, Nov 1994.
10. C. Connelly and C. S. Ellis, "*Scheduling to reduce memory coherence overhead on coarse-grain multiprocessors*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
11. D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems.* Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.
12. 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.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
13. D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.
14. 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.
15. 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.
16. 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.
17. D. G. Feitelson and L. Rudolph, "*Coscheduling based on runtime identification of activity working sets*". *Intl. J. Parallel Programming* **23(2)**, pp. 135–160, Apr 1995.
18. M. J. Gonzalez, Jr., "*Deterministic processor scheduling*". *ACM Comput. Surv.* **9(3)**, pp. 173–204, Sep 1977.
19. B. C. Gorda and E. D. Brooks III, *Gang Scheduling a Parallel Machine.* Technical Report UCRL-JC-107020, Lawrence Livermore National Laboratory, Dec 1991.

20. 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.
21. R. L. Henderson, "*Job scheduling under the portable batch system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
22. 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.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
23. Intel Supercomputer Systems Division, *Paragon User's Guide.* Order number 312489-003, Jun 1994.
24. O. Kipersztok and J. C. Patterson, "*Intelligent fuzzy control to augment the scheduling capabilities of network queueing systems*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
25. L. Kleinrock and J-H. Huang, "*On parallel processing systems: Amdahl's law generalized and some results on optimal design*". *IEEE Trans. Softw. Eng.* **18(5)**, pp. 434–447, May 1992.
26. 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.
27. D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
28. D. J. Lilja, "*Exploiting the parallelism available in loops*". *Computer* **27(2)**, pp. 13–26, Feb 1994.
29. V. M. Lo, "*Heuristic algorithms for task assignment in distributed systems*". *IEEE Trans. Comput.* **37(11)**, pp. 1384–1397, Nov 1988.
30. 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.
31. 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.
32. J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.
33. C. M. Pancake, "*Multithreaded languages for scientific and technical computing*". *Proc. IEEE* **81(2)**, pp. 288–304, Feb 1993.
34. C. H. Papadimitriou and M. Yannakakis, "*Towards an architecture-independent analysis of parallel algorithms*". *SIAM J. Comput.* **19(2)**, pp. 322–328, Apr 1990.
35. 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.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
36. 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.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

37. M. E. Rosenkrantz, D. J. Schneider, R. Leibensperger, M. shore, and J. Zollweg, "*Requirements of the Cornell Theory Center for resource management and process scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
38. 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.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
39. 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.
40. W. Saphir, L. A. Tanner, and B. Traversat, "*Job management requirements for NAS parallel systems and clusters*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
41. S. Setia, "*The interaction between memory allocation and adaptive partitioning in message-passing multicomputers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
42. J. E. Smith, "*Characterizing computer performance with a single number*". *Comm. ACM* **31(10)**, pp. 1202–1206, Oct 1988.
43. P. G. Sobalvarro and W. E. Weihl, "*Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
44. M. S. Squillante, "*On the benefits and limitations of dynamic partitioning in parallel computer systems*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
45. I. Stoica, H. Abdel-Wahab, and A. Pothen, "*A microeconomic scheduler for parallel computers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
46. S. Thakkar, P. Gifford, and G. Fielland, "*Balance: a shared memory multiprocessor system*". In 2nd *Intl. Conf. Supercomputing*, vol. I, pp. 93–101, 1987.
47. 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.
48. 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.
49. K. Y. Wang and D. C. Marinescu, "*Correlation of the paging activity of individual node programs in the SPMD execution model*". In 28th *Hawaii Intl. Conf. System Sciences*, vol. I, pp. 61–71, Jan 1995.
50. K. K. Yue and D. J. Lilja, "*Loop-level process control: an effective processor allocation policy for multiprogrammed shared-memory multiprocessors*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.