Theory and Practice in Parallel Job Scheduling

Dror G. Feitelson[1], Larry Rudolph[1], Uwe Schwiegelshohn[2], Kenneth C. Sevcik[3], and Parkson Wong[4]

[1] Institute of Computer Science
The Hebrew University, 91904 Jerusalem, Israel
{feit,rudolph}@cs.huji.ac.il
[2] Computer Engineering Institute
University Dortmund, 44221 Dortmund, Germany
uwe@carla.e-technik.uni-dortmund.de
[3] Department of Computer Science
University of Toronto, Toronto, Ontario, Canada M5S 3G4
kcs@cs.toronto.edu
[4] MRJ, Inc., NASA Contract NAS 2-14303
Moffett Field, CA 94035-1000, USA
parkson@nas.nasa.gov

Abstract. The scheduling of jobs on parallel supercomputer is becoming the subject of much research. However, there is concern about the divergence of theory and practice. We review theoretical research in this area, and recommendations based on recent results. This is contrasted with a proposal for standard interfaces among the components of a scheduling system, that has grown from requirements in the field.

1 Introduction

The scheduling of jobs on parallel supercomputers is becoming the subject of much research activity. See, for example the proceedings of three workshops [40], and a survey of a large number of proposed and implemented approaches [19]. It has become a distinct research topic from the largely unrelated, but better known problem of DAG scheduling [1]. DAG scheduling assumes that all tasks have fixed and specified dependencies, whereas job scheduling assumes that the jobs are mostly independent.

This paper is about scheduling jobs on distributed memory massively parallel processors (MPPs), which currently dominate the supercomputing arena. In terms of scheduling, on such machines memory is typically allocated in conjunction with the processors, rather than being treated as a distinct resource. However, this does not preclude a shared address space model of computation, and indeed many recent systems provide hardware support for different levels of memory sharing.

There are a growing number of high performance computing facilities that support large diverse workloads of parallel jobs on multicomputers that have tens to thousands of processors. The typical way that they are currently used is that:

1. The system is divided into "partitions" consisting of different numbers of processors. Most processors are allocated to partitions devoted to serving parallel jobs. One partition is typically set aside for support of interactive work through time-slicing of its processors. Another may be devoted to service tasks, such as running a parallel file system. The configuration of partitions may be changed on a regular basis (for example, by providing larger partitions for parallel jobs at night or over weekends, at the expense of the interactive partition).
2. A (large) number of queues are established, each one corresponding to a specific combination of job characteristics. (For example, one queue might correspond to jobs that require as many as 32 processors, and are expected to run no longer than 15 minutes.) Some queues are served at higher priority than others, so the user tends to submit a job to the highest priority queue for which the job qualifies based on its expected resource requirements.
3. Each partition is associated with one or more queues, and its processors serve as a pool for those queues. Whenever some processors are free, the associated queues are searched in order of priority for one that is non-empty. The first job in that non-empty queue is then activated in the partition, and it runs until it completes, provided the number of free processors is sufficient. Within each queue jobs are processed strictly in first-come-first-served order.

Thus:

- the number of processors assigned to a job is fixed by the user;
- once initiated the job runs to completion.

While there exist some innovations that have been introduced into production systems, such as non-FCFS service and support for swapping, the general trend is to retain the same framework, and moreover, to cast it into a standard. Many studies, however, show that more flexibility in both the scheduler actions and the way programs make use of parallelism result in better performance. But there is hope for convergence [25]. For example, theoretical analysis underscores the effectiveness of preemption in achieving low average response times, and also shows that considerable benefits are possible if the scheduler is allowed to tailor the partition sizes in accordance with the current system load. Notably, much of this work is based on workload models that are derived from measurements at supercomputer installations.

We survey the theoretical background in Section 2, and the specific recommendations that are made in Section 3. The standardization effort based on practical work at large installations is reviewed in Section 4. Finally, we discuss this state of affairs and present our conclusions in Section 5.

2 Survey of Theoretical Results

Various kinds of scheduling or sequencing problems have been addressed since the fifties by theoretical researchers from the areas of computer science, operations research, and discrete mathematics. The challenge of efficient job management on computers has frequently been named as a key reason to address this kind of problems. This is especially true for job scheduling on parallel systems with a large number of processors or nodes. Hence a direct use of many of these theoretical results in real applications would seem to be natural. However, many of these theoretical resutls rely on a creative set of assumptions, in order to make their proofs tractable. This divergence from reality not only make them hard to use in practice, but also the diversity of divergence makes them hard to compare with each other.

2.1 The Diversity of Divergence

This section covers many of the assumptions of theoretical work, by presenting a rough classification of different theoretical models. This includes different cost metrics, different features and operations available on the modeled system, and different algorithmic approaches.

Cost metrics For the discussion of the various cost metrics we use the following notations:

$$\begin{aligned} t_i &= \text{completion time of job } i \\ s_i &= \text{release time of job } i \\ w_i &= \text{weight of job } i \\ d_i &= \text{deadline of job } i \end{aligned}$$

The *completion* time t_i is the time when the computer system has finally completed work on this job. Note that no information is provided on whether the job has been successfully completed or whether it has been removed from the system for other reasons. The *release* time s_i is the earliest time the computer system can start working on job i. Usually, the release time of a job is identical with its submital or arrival time, i.e. the first time when the system becomes aware of the new job. However, sometimes it is assumed that the scheduling system is aware of all jobs at time 0, but job i cannot be started before some time $s_i \geq 0$. The *weight* w_i of a job is a way to prioritize one job over another. The *deadline* d_i is the time by which a job must complete its execution. There is no generally accepted definition as to what happens if the deadline is not met for a specific job, i.e. $t_i > d_i$.

Obviously, the role of the scheduler is the allocation of limitted system resources to competing jobs. A job should somehow be charged for its resource comsumption. Often the cost of a schedule is simply the sum of the individual job costs. This cost function serves as basis to compare and evaluate different schedules. Assuming a job system τ the following metrics are commonly used:

$$
\begin{aligned}
\max_{i \in \tau} t_i &= \text{Makespan (throughput)} \\
|\{i \in \tau | t_i > d_i\}| &= \text{Deadline misses} \\
\sum_{i \in \tau} w_i t_i &= \text{Weighted completion time} \\
\sum_{i \in \tau} w_i (t_i - s_i) &= \text{Weighted flow (response) time} \\
\sum_{i \in \tau} w_i \max\{0, t_i - d_i\} &= \text{Weighted tardiness}
\end{aligned}
$$

Note that response time and flow time usually have the same meaning. The origin of these criteria often goes back to the fifties. For instance Smith [82] showed in 1956 that the sum of the weighted completion times for a system of jobs on a single processor can be minimized if the tasks are scheduled by increasing execution time to weight ratio, the so called Smith ratio. If all jobs have unit weight this algorithm becomes the well known shortest-job first method.

These metrics allow a relatively simple evaluation of algorithms which may be one reason for their popularity, but there are some subtle difference in them. A schedule with optimal weighted completion time also has the optimal weighted flow time. This equality does not hold, however, when they deviate by even a constant factor from the optimum as shown by Kellerer et al. [42] and by Leonardi and Raz [47].

In reality, the metrics attempt to formalize the real goals of a scheduler:

1. Satisfy the users.
2. Maximize the profit.

For instance, a reduction of the job response time will most likely improve user satisfaction.

Example 1. Assume that a job i needs approximately 3 hours of computation time. If the user submits the job in the morning (9am) he may expect to receive the results after lunch. It probably does not matter to him whether the job is started immediately or delayed for an hour as long as it is done by 1pm. Any delay beyond 1pm may cause annoyance and thus reduce user satisfaction, i.e. increase costs. This corresponds to tardiness scheduling. However, if the job is not completed before 5pm it may be sufficient if the user gets his results early next morning. Moreover, he may be able to deal with the situation easily if he is informed at the time of submital that execution of the job by 5pm cannot be expected. Also, if the user is charged for the use of system resources, he may be willing to postpone execution of his job until nighttime when the charge is reduced.

The use of metrics such as throughput and response time in many commercial installations may be due to the simplicity of the evaluation, or it may be a sign of some non-obvious influence from theory. On the other hand, a good management policy for a commercial system may require that different metrics are used during different times of the day: During daytime many users will actually wait for the completion of their submitted jobs. Thus a response time metric is appropriate. However, during the night it is best to maximize the throughput of jobs.

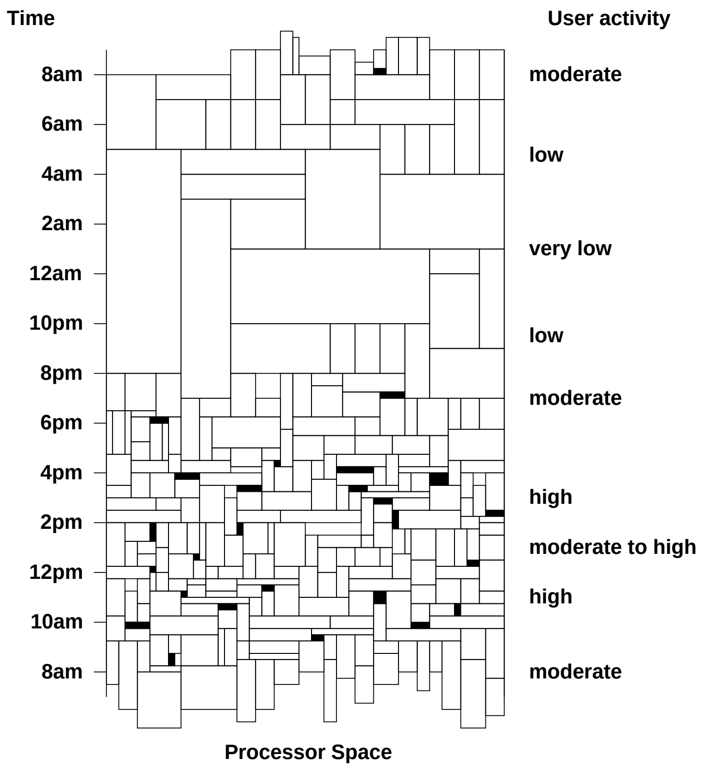

Fig. 1. *Workload of a parallel computer over the course of a day.*

Fig. 1 shows the load of a multiprocessor over the course of a day. For reasons of simplicity each job is described as a rectangle. Black rectangles denote idle processors due to fragmentation. However, note that multiprocessors do not necessarily require a linear one-dimensional processor space. But this way it is easier to visualize jobs. As shown in the figure, during periods of high user activity small jobs are given preference even if some processors remain idle due to fragmentation of the processor space. Jobs are allocated resources such that the shortest response time is achieved. On the other hand during periods of low user activity large batch jobs are started. Also moldable jobs are run in a way to increase efficiency, i.e. using fewer processors while requiring longer execution time.

Recent studies, e.g. Charkrabarti et al. [7], explicitly address the problem of bicriteria scheduling where scheduling methods are introduced which generate good schedules with respect to the makespan and the weighted completion time metric.

The Model A large variety of different machine and scheduling models have been used in studies of scheduling problems. The constraints incorporated into these models directly affect operations of the scheduler. They are at least partly inspired by the way real systems are managed and how parallel applications are written. The following roughly classifies these models according to five criteria:

1. Partition Specification Each parallel job is executed in a *partition* that consists of a number of processors. The size of such a partition may depend on the multiprocessor, the application, and the load of multiprocessor [25]. Moreover, the size of the partition of a specific job may change during the lifetime of this job in some models:

- **Fixed.** The partition size is defined by the system administrator and can be modified only by reboot.
- **Variable.** The partition size is determined at submission time of the job based on user request.
- **Adaptive.** The partition size is determined by the scheduler at the time the job is initiated, based on the system load, and taking the user request into account.
- **Dynamic.** The partition size may change during the execution of a job, to reflect changing requirements and system load.

Feldmann et al. [26] have considered fixed partitions generated by different architectures such as hypercubes, trees, or meshes. Many other authors use the variable partitioning paradigm, in which each job requires a specific number of processors but can be scheduled on any subset of processors of the system. An example of a theoretical study based on the adaptive approach is the work of Turek et al. [88]. Here, the application does not require a specific number of processors, but can use different numbers. However, once a partition for a job has been selected its size cannot change anymore. Finally, in dynamic partitioning the size of a partition may change at run time. This model has, for instance, been used by Deng et al. [11].

2. Job Flexibility As already mentioned advanced partitioning methods must not only be supported by the multiprocessor system but by the application as well. Therefore, Feitelson and Rudolph [25] characterize applications as follows:

- **Rigid jobs.** The number of processors assigned to a job is specified external to the scheduler, and precisely that number of processors are made available to the job throughout its execution.
- **Moldable jobs.** The number of processors assigned to a job is determined by the system scheduler within certain constraints when the job is first activated, and it uses that many processors throughout its execution.

- **Evolving jobs.** The job goes through different phases that require different numbers of processors, so the number of processors allocated may change during the job's execution in response to the job requesting more processors or relinquishing some. Each job is allocated at least the number of processors it requires at each point in its execution (but it may be allocated more to avoid the overheads of reallocation at each phase).
- **Malleable jobs.** The number of processors assigned to a job may change during the job's execution, as a result of the system giving it additional processors or requiring that the job release some.

The manner in which an application is written determines which of the four types is used[1]. The more effort devoted to writing the application with the intention of promoting it to one of the later types, the better a good scheduler can perform with respect to both the particular job, and the whole workload. Generally, a scheduler can start a job sooner if the job is moldable or even malleable than if it is rigid [87].

If jobs are *moldable*, then processor allocations can be selected in accordance with the current system load, which delays the onset of saturation as system load increases [25]. It is generally not difficult to write an application so that it is moldable, and is able to execute with processor allocations over some range (e.g., any power of two from four to 256). *Evolving* jobs arise when applications go through distinct phases, and their natural parallelism is different in different phases. For such jobs, system calls are inserted at the appropriate points in the application code to indicate where parallelism changes [96].

Certain parallel applications, such as those based on the "work crew" model[2], can be modified to be *malleable* relatively easily. An increased processor allocation allows more processors to take work from the queue, while a reduction means that some processors cease picking up work and are deallocated [67]. In most cases, however, it is more difficult to support malleability in the way an application is written. One way of attaining a limited form of malleability is by creating as many threads in a job as the largest number of processors that would ever be used, and then using multiplexing (or *folding* [51,38]) to have the job execute on a lesser number of processors. Alternatively, a job can be made malleable by inserting application specific code at particular synchronization points to repartition the data in response to any change in processor allocation. The latter approach is somewhat more effective, but it requires more effort from the application writer, as well as significantly more system support. Much of the required mechanisms for supporting malleable job scheduling is present in facilities for checkpointing parallel jobs [66]. Hence, a combined benefit can be derived if processor allocations are changed only at times when checkpoints are taken.

[1] Some theoretical studies use different terminology. For example, Ludwig and Tiwari [50] speak about "malleable" jobs which are equivalent to moldable jobs in our terminology.

[2] In the "work crew" model, processors pick up relatively small and independent units of computation from a central queue.

For stable applications that are widely and frequently used by many users, the effort required to make them malleable may be well justified. Otherwise, it is probably not worthwhile to write applications so that they are malleable.

3. Level of Preemption Supported Another issue is the extent to which individual threads or entire jobs can be preempted and potentially relocated during the execution of a job:

- **No Preemption.** Once a job is initiated it runs to completion while holding all its assigned processors throughout its execution.
- **Local Preemption.** Threads of a job may be preempted, but each thread can only be resumed later on the same processor. This kind of preemption does not require any data movement between processors.
- **Migratable Preemption.** Threads of a job may be suspended on one processor and subsequently resumed on another.
- **Gang Scheduling.** All active threads of a job are suspended and resumed simultaneously. Gang scheduling may be implemented with or without migration.

While many theoretical scheduling studies only use a model without preemption, more recently preemption has also been taken into account. Schwiegelshohn [71] uses a gang scheduling model without migration. The work of Deng et al. [11] is based upon migratable preemption.

In a real system the preemption of a job requires that all the job's threads be stopped in a consistent state (i.e., without any messages being lost), and the full state of each thread must be preserved. The memory contents associated with the job may be either explicitly written out of memory, or may be implicitly removed over time by a process such as page replacement. Whether or not the data of a job is removed from memory when the job is preempted depends in part on the memory requirements of the preempting job and the total amount of memory available.

In addition migratable preemption needs the mechanism of moving a thread from one processor to another, while preserving all existing communication paths to other threads. Also, when a thread is migrated, its associated data must follow. In message passing systems, this requires that the data be copied from one processor to another. In shared memory systems, the data can be transferred a cache line or page at a time as it is referenced. Preemption may have great benefit in leading to improved performance, even if it is used infrequently and on only a small fraction of all jobs.

Preemption in real machines has an overhead cost, e.g. Motwani et al. [55] address the overhead by minimizing the number of preemptions. In order to compare a preemptive schedule with non-preemptive ones Schwiegelshohn [71] includes a time penalty for each preemption.

4. Amount of Job and Workload Knowledge Available Systems differ in the type, quantity, and accuracy of information available to and used by the scheduler.

Characteristics of individual jobs that are useful in scheduling include (i) the total processing requirement, and (ii) the speedup characteristics of the job. Full knowledge of the latter requires knowing the execution time of the job for each number of processors on which it might be executed. Partial knowledge is provided by characteristics such as average parallelism (the average number of processors busy when the job is allocated ample processors), and maximum parallelism (the largest number of processors that a job can make use of at any point in its execution).

Workload information is also useful in choosing a scheduling policy. For example, workload measurements at a number of high-performance computing facilities have indicated that the variability in processing requirements among jobs is extreme, with most jobs having execution times of a few seconds, but a small number having execution times of many hours. The coefficient of variation[3], or CV, of the service times of jobs has been observed to be in the four to seventy range at several centers [8,64,21]. This implies that a mechanism to prevent short jobs from being delayed by long jobs is mandatory.

The knowledge available to the scheduler may be at one of the following levels:

- **None.** No prior knowledge is available or used in scheduling, so all jobs are treated the same upon submission.
- **Workload.** Knowledge of the overall distribution of service times in the workload is available, but no specific knowledge about individual jobs. Again, jobs are treated the same, but policy attributes and parameters can be tuned to fit the workload.
- **Class.** Each submitted job is associated with a "class", and some key characteristics of jobs in the class are known, including, for example, estimates of processing requirement, maximum parallelism, average parallelism, and possibly more detailed speedup characteristics.
- **Job.** The execution time of the job on any given number of processors is known exactly.

Job knowledge, which is defined to be exact, is unrealistic in practice. However, assuming omniscience in modeling studies makes it possible to obtain an optimistic bound on performance that is not attainable in practice. Presuming job knowledge in modeling studies sets a standard in performance against which practically realizable scheduling algorithms, which use class knowledge at most, can be compared.

On-line scheduling has been addressed more frequently in recent years. For instance Shmoys et al. [78] discussed makespan scheduling if the job characteristics are not known until the release time and the execution time requirements of the job are also not available before the job has been executed to completion. They show that any algorithm with all jobs available at time 0 can be converted

[3] The coefficient of variation of a distribution is its standard deviation divided by its mean.

to an algorithm that handles dynamic arrivals with a competitive factor at most twice larger.

Many systems make no effort at all to use information that accompanies a job submission to estimate the resource requirements of the job. However, smallest demand type disciplines (e.g., "Least Work First" (LWF)) can be used to yield low average response times if the resource demand of each job is known (precisely or approximately). For example, in many current systems jobs are submitted to one of a large number of queues, and the queue selected indicates such information as the number of processors needed, a limit on the execution time, and other parameters. Thus, each queue corresponds to a job class. In these systems, this information can be used implicitly through the way queues are assigned to (fixed) partitions, and the relative priorities assigned to the queues.

Any information provided by the user relating to job resource requirements must be used carefully both because it is prohibitively difficult for the user to consistently provide information with high accuracy, and also because the user may be motivated to deceive the scheduler intentionally. Thus, sources from which to gain knowledge about job resource requirements must be broadened to include:

- consider user provided information (while recognizing that it is historically quite unreliable, in part because users aren't careful about making good estimates);
- measure efficiency during execution and increase processor allocations only for jobs that are using their currently allocated processors effectively;
- keep track of execution time and speedup knowledge from past executions on a class by class basis, and use that information.

All identifying characteristics associated with the submital of a job can potentially be used to determine its class. These characteristics include the user id, the file to be executed, the memory size specified, and possibly others. An estimate of the efficiency [59] or the execution time [31] of a job being scheduled can be obtained from retained statistics on the actual resource usage of jobs from the same (or a similar) class that have been previously submitted and executed. A small database can be kept to record resource consumption of jobs on a class by class basis. This is very useful particularly for large jobs that are executed repeatedly.

5. Memory Allocation For high performance applications, memory is usually the critical resource.

This is particularly true in shared-memory systems, where the allocation of memory is relatively decoupled from the allocation of processors. Thus there are two types of memory to consider:

- **Distributed Memory.** Typically each processor and its associated memory is allocated as a unit. Message passing is used to access data in remote memory.

– **Shared Memory.** Access cost to shared memory can either be uniform (UMA) or nonuniform (NUMA) for all the processors. With UMA, there is the potential for more equitable allocation of the memory resources. With NUMA, the performance is sensitive to the allocation of a job to processors and its data to memories.

Mostly, memory requirements have been ignored in real scheduling systems and are not even part of the model in theoretical studies, although this is changing [65,73,62,61].

Algorithmic Methods The intractability for many scheduling problems has been well studied [28]. Examples in the specific area of parallel job scheduling include preemptive and non-preemptive gang scheduling by Du and Leung [13] using makespan, and Bruno, Coffman, and Sethi [6] and McNaughton [53] who use weighted completion time as optimization criterion.

With the intractability of many scheduling problems being established, polynomial algorithms guaranteeing a small deviation from the optimal schedule appear more attractive. Some polynomial algorithms are still very complex, [70], while others are particular simple algorithms, like list scheduling methods [41,95]. The latter promises to be of the greatest help for the selection of scheduling methods in real systems.

Although many of the approximation algorithms have a low computational complexity and produce schedules that are close to the optimum, they are usually not the method of choice in commercial installations.

The worst case approximation factor is usually of little relevance to a practical problem since a schedule that approaches the worst case is often unacceptable for a production schedule. For instance, the makespan approximation factor for list schedules of non-preemptive parallel job schedules is 2. In other words, up to 50% of the nodes of a multiprocessor system may be left idle. However, these high costs are only encountered for a few job systems which may never be part of real workloads.

Turek et al. [89] proposed "SMART" schedules for the off-line non-preemptive completion time scheduling of parallel jobs. They prove an approximation factor of 8 [72] and give a worst case example with a deviation of 4.5. However, applying the algorithm on job systems obtained from the traces of the Intel Paragon at the San Diego Supercomputing Center gave an average deviation from the optimum by 2. This result was further improved to the factor 1.4 by using the job order of the SMART schedule as input for a list schedule [33]. But note that SMART generates non-preemptive off-line schedules and requires the knowledge of the execution time of all jobs. The consideration of more complex constraints may make any general approximation algorithm impossible [42,47].

The evaluation of any scheduler can be either done by comparing its schedule against the optimal schedule or against schedules generated by other methods. Sleator and Tarjan [80] introduced the notion of competitive analysis. An on-line algorithm is competitive if it is guaranteed to produce a result that is within a constant factor of the optimal result. Only the deviation from the optimal schedule can determine whether there is enough room for improvement to motivate further algorithmic research. Unfortunately, the optimal schedule cannot be obtained easily, but an analysis of an approximation algorithm can use lower bounds for the optimal schedule to determine the competitive factor, e.g. the squashed area bound introduced by Turek et al. [89].

Moreover, the theoretical analysis may be able to pinpoint the conditions which may lead to a bad schedule. These methods can also be applied to any practical approach and help to determine critical workloads. If the evaluation of real traces reveals that such critical workloads rarely or even never occur then they can either be ignored or the approach can be enhanced with a procedure to specifically handle those situations.

For instance Kawaguchi and Kyan's LRF schedule [41] can be easily extended to parallel jobs. As long as no parallel job requires more than 50% of the processors, this will only increase the approximation factor from 1.21 to 2 [87]. However, if jobs requiring more processors are allowed in addition, no constant approximation factor can be guaranteed.

2.2 Some Specific Studies

Workload Characterization Several workload characterization studies of production high-performance computing facilities have been carried out. They reveal characteristics of actual workloads that can be exploited in scheduling.

Feitelson and Nitzberg [21] noted that repeated runs of the same application occurred frequently, and later runs tended to have similar resource consumption patterns as the corresponding earlier ones. Hotovy [37] studied a quite different system, yet found many of the same observations to hold. Gibbons [30] also analyzed workload data from the Cornell site in addition to two sites where parallel applications are executed on a network of workstations, concluding that in all three systems classifying the jobs by user, execution script, and requested degree of parallelism led to classes of jobs in which execution time variability is much lower than in the overall workload. The common conclusion is that much information about a job's resource requirements can be uncovered without demanding the user's cooperation.

Feitelson [17] studied the memory requirements of parallel jobs in a CM-5 environment. He found that memory is a significant resource in high-performance computing, although he observed that users typically request more processors than naturally correspond to their memory requirements.

Jann et al. [39] have produced a workload model based on measurements of the workload on the Cornell Theory Center SP2 machine. This model is intended to be used by other researchers, leading to easier and more meaningful comparison of results. Nguyen at al. [58] have measured the speedup characteristics of a variety of applications.

Batch Job Scheduling In an effort to improve the way current schedulers behave, several groups have modified NQS implementations to allow queue reordering in order to achieve better packing. Lifka et al. [49,79] have developed a scheduler on top of LoadLeveler with the feature that the strict FCFS order of activating jobs is relaxed. In this scheduler, known as "EASY", jobs are scheduled in a FCFS order to run at the earliest time that a sufficient number of processors are available for them. However, this can mean that smaller jobs may be executed before bigger jobs that arrived earlier, whenever they can do so without delaying the previously scheduled jobs[4]. It was found that user satisfaction was greatly increased since smaller jobs tended to get through faster, because they could bypass the very big ones.

Henderson [35] describes the Portable Batch System (PBS), another system in which performance gains are achieved by moving away from strict FCFS scheduling. Wan et al. [92] also implement a non-FCFS scheduler that uses a variation of a 2-D buddy system to do processor allocation for the Intel Paragon.

Thread-oriented scheduling Nelson, Towsley, and Tantawi [57] compare four cases in which parallel jobs are scheduled in either a *centralized* or *de-centralized* fashion, and the threads of a job are either spread across all processors or all executed on one processor. They found that best performance resulted from centralized scheduling and spreading the threads across processors. Among the other options, decentralized scheduling of split tasks beat centralized scheduling with no splitting under light load, but the reverse is true under heavy load.

Dynamically Changing A Job's Processor Allocation Because the efficiency of parallel jobs generally decreases as their processor allocation increases, it is necessary to decrease processor allocations to moldable jobs as the overall system load increases in order to avoid system saturation (see Sevcik [77]). Zahorjan and McCann [97] found that allocating processors to evolving jobs according to their dynamic needs led to much better performance than either run-to-completion with a rigid allocation or round-robin. For the overhead parameters they chose, round-robin beat run-to-completion only at quite low system loads.

Ghosal et al. [29] propose several processor allocation schemes based on the *processor working set* (PWS), which is the number of allocated processors for which the ratio of execution time to efficiency is minimized. (The PWS differs from the average parallelism of the job by at most a factor of two [16].) The best of the variants of PWS gives jobs at most their processor working set, but under heavy load gives fewer and fewer processors to each job, thus increasing efficiency and therefore system capacity.

Setia, Squillante, and Tripathi [74] use a queuing theoretic model to investigate how parallel processing overheads cause efficiency to decrease with larger processor allocations. In a later study [75], they go on to show that dynamic partitioning of the system beats static partitioning at moderate and heavy loads. Naik, Setia and Squillante [56] show that dynamic partitioning allows much better performance than fixed partitioning, but that much of the difference in

[4] Actually, EASY only guarantees that the *first* job in the queue will not be delayed.

performance can be obtained by using knowledge of job characteristics, and assigning non-preemptive priorities to certain job classes for admission to fixed partitions.

McCann and Zahorjan [51] found that "efficiency-preserving" scheduling using folding allowed performance to remain much better than with equipartitioning (EQUI) as load increases. Padhye and Dowdy [60] compare the effectiveness of treating jobs as moldable to that of exploiting their malleability by folding. They find that the former approach suffices unless jobs are irregular (i.e., evolving) in their pattern of resource consumption. Similarly, in the context of quantum-based allocation of processing intervals, Chiang et al. [9] showed that static processor allocations (for which jobs need only be moldable) led to performance nearly as good as that obtained by dynamic processor allocation (which requires that jobs be malleable).

Foregoing Optimal Utilization Downey [12] studies the problem of scheduling in an environment where moldable jobs are activated from an FCFS queue, and run to completion. He suggests how to use predictions of the expected queuing time for awaiting the availability of different numbers of processors in order to decide when a particular job should be activated. The tradeoff is between starting a job sooner with fewer processors and delaying its start (causing processors to be left idle) until a larger number of processors is available. Algorithms that leave processors idle in anticipation of future arrivals were also investigated by Rosti et al. [69] and Smirni et al. [81].

The Need for Preemption A number of studies have demonstrated that despite the overheads of preemption, the flexibility derived from the ability to preempt jobs allows for much better schedules.

The most often quoted reason for using preemption is that time slicing gives priority to short running jobs, and therefore approximates the Shortest-Job First policy, which is known to reduce the average response time. This is especially true when the workload has a very high variability (which is the case in real production systems). Parsons and Sevcik [64] show the importance of preemption under high variance by comparing versions with and without preemption of the same policy. Good support for short running jobs is important because it allows for interactive feedback.

Another use of preemption that is also known from conventional uniprocessor systems is that it allows the overlap of computation and I/O. This is especially important in large scale systems that perform I/O to mass storage devices, an operation that may take several minutes to complete. Lee et al. [45] have shown that some jobs are more sensitive to perturbations than others, therefore some jobs have a stronger requirement for gang scheduling. However, all parallel jobs benefit from *rate-equivalent* scheduling, that is all threads get to run for the same fraction of the wallclock time, but not necessarily simultanously.

Preemption is also useful to control the share of resources allocated to competing jobs. Stride and lottery scheduling use the notion of tickets to fairly allocate resources, including CPU time [90,91]. Each job gets a proportion of the CPU, according to the proportion of tickets assigned to the job. A time line is then produced for each job containing the periods when the job is scheduled to run. That is, the time quantums are placed at strides along the timeline. The timelines from all the jobs are pushed down onto a single timeline, and idle time squeezed out.

In parallel systems, preemption is also useful in reducing fragmentation. For example, with preemption it is not necessary to accumulate idle processors in order to run a large job. Feitelson and Jette [20] demonstrate that the preemptions inherent in time-slicing allow the system to escape from bad processor allocation decisions, boosting utilization over space-slicing for rigid jobs, and avoiding the need for non-work conserving algorithms. Also, preemtion is needed in order to migrate processes to actively counter fragmentation.

Finally, in many computing centers it was noted that a non-negligible number of parallel batch jobs failed to run more than a minute due to reasons such as an incorrectly specified data file. Therefore, it might be reasonable that jobs should be started immediately after submission, then interrupted after 1 minute and finally resumed and completed at a later time.

Time-Slicing and Space-Slicing Scheduling Many variations of scheduling algorithms based on time-slicing and space-slicing have been proposed and evaluated. Time-slicing is motivated by the high variability and imperfect knowledge of service times, as described above, while space-slicing is motivated by the goal of having processors used with high efficiency.

Time slicing is typically implemented by gang scheduling, that is, all the threads in a job are scheduled (and de-scheduled) simultaneously. Gang scheduling is compared to local scheduling and is found to be superior by Feitelson and Rudolph [24]. Squillante et al. [85] and Wang et al. [94] have analyzed a variation of gang scheduling that involves providing service cyclically among a set of fixed partition configurations, each having a number of partitions equal to some power of two. They find that long jobs benefit substantially from this approach, but only at the cost of longer response times for short jobs. Feitelson and Rudolph [23] and Hori et al. [36] analyze a more flexible policy in which there is time slicing among multiple active sets of partitions. Lee et al. [45] study the interaction of gang scheduling and I/O, and found that many jobs may tolerate the perturbations caused by I/O, that I/O bound jobs suffer under gang scheduling, and therefore argue in favor a flexible gang scheduling.

Several studies have revealed that EQUI does very well, even when some moderate charge for the overhead of frequent preemptions is made [86,48]. Squillante [84] provides an analysis of the performance of dynamic partitioning. Deng et al. show that EQUI is optimally competitive [11]. Dussa et al. [14] compares space-slicing against no partitioning, and finds that space-partitioning pays off.

Knowledge-Based Scheduling Majumdar, Eager and Bunt showed that, under high variability service time distributions, round-robin (RR) was far better than FCFS, but that policies based on knowledge of the processing requirement (such as least work first) were still better. Knowledge of the average parallelism of a job makes it possible to allocate each job an appropriate number of processors to make it operate at a near-optimal ratio of execution time to efficiency [16]. With the knowledge of how many processors each job uses, policies for packing the jobs into frames for gang scheduling are investigated by Feitelson [18]. Feitelson and Rudolph [22] describe a discipline in which processes that communicate frequently are identified, and it is assured that the corresponding threads are all activated at the same time. Similar schemes in which co-scheduling is triggered by communication events were described by Sobalvarro and Weihl [83] and by Dusseau, Arpaci, and Culler [15].

Taking system load and minimum and maximum parallelism of each job into account as well, still higher throughputs can be sustained [77]. Chiang et al. [8] show that use of knowledge of some job characteristics plus permission to use a single preemption per job allows run-to-completion policies to approach ideal (i.e., no overhead) EQUI, and Anastasiadis et al. [3] show that, by setting the processor allocation of moldable jobs based on some known job characteristics, disciplines with little or no preemption can do nearly as well as EQUI.

Other Factors in Scheduling McCann and Zahorjan [52] studied the scheduling problem where each job has a minimum processor allocation due to its memory requirement. They find that a discipline based on allocation by a buddy system consistently does well. Alverson et al. [2] describe the scheduling policy for the Tera MTA, which includes consideration of memory requirements. Brecht [5] has carried out an experimental evaluation of scheduling in systems where processors are identified with clusters or *pools*, and intracluster memory access is faster than intercluster access. A surprising result is that *worst-fit* scheduling, where each job is allocated to the pool with the most available processors, beats *best-fit* scheduling, where jobs are placed where they come closest to filling out a pool. This is a result of using a model of evolving jobs, where it is best to leave these jobs space to grow. Yue [96] describes the creation of evolving jobs by selecting (in the compiler) at the top of each loop what degree of parallelism should be used for that loop.

Experiments with Parallel Scheduling Many of the results of the modeling studies described above have been corroborated by experimental studies in which various policies were implemented in real systems.

Gibbons has experimented with a number of scheduling disciplines for scheduling rigid jobs in a network of workstations environment. His conclusions include:

- Activating jobs in Least Expected Work First (LEWF) order rather than FCFS reduces the resulting average response time by factors from two to six in various circumstances.
- If service times are unknown or if only estimates are available, then “backfilling” (as in EASY) reduces average response times by a factor of two or

more. (If service times are known exactly, then back-filling has less relative benefit.)
- Whether back-filling is used or not, knowledge of service times is very helpful (particularly if preemption is supported). Having job knowledge and using it leads to response times that are a factor of three to six smaller than for the case of no knowledge. When the knowledge is restricted to class knowledge based on the a small database that records execution characteristics of jobs, the average response times are roughly half those with no knowledge.
- If some knowledge (class or job) is available, then preemption is much less valuable than in the case where no knowledge is available and bad decisions are made (which can only be corrected by preemption).

Parsons has experimented with a broader class of disciplines, most of which exploit moldable and malleable jobs. His positive observations include:

- If migratable preemption is supported at low cost, then very good performance can be achieved, even if no service time knowledge is available. (Also, malleability is not of much additional benefit.)
- If only local preemption is supported, then class knowledge of service times is needed in order to do well by using LEWF order for activation.
- When preemption is not supported, class knowledge and LEWF order are helpful (roughly halving average response times), but not as much as with local preemption supported.
- In (typical) environments where the distribution of service times has very high variance, LERWF does very well when some service time knowledge is available; otherwise, if malleability doesn't lead to excessive overhead, then a simple rule like EQUI does well.

Some additional observations on the negative side are:

- The value of local preemption is restricted by the fragmentation that occurs because jobs must be restarted on the same set of processors on which they previously ran. (In this case, either clever packing strategies or even fixed partitioning are beneficial, because the dependencies among jobs are then limited [25].)
- Even with moldable jobs, performance is poor unless preemption is supported, because if inappropriate allocations are occasionally made to very long jobs, then only preemption can remedy the situation.

3 Recommendations and Future Directions

The current state-of-the-art regarding scheduling on large-scale parallel machines is to use simple and inflexible mechanisms. In essence, the number of processors used for each job is chosen by the user, some sufficiently large partition acquired, and the job is run to completion. A few recent systems support preemption, so that a parallel job can be interrupted and possibly swapped out of memory, but many installations choose not to use this option due to high associated overheads and lack of adequate I/O facilities.

This section presents six recommendations to improve the performance of state-of-the-art schedulers. The recommendations are based on the conclusions of the investigations, analysis, and simulations described above.

A great deal has been learned about how "in theory" multiprogrammed multiprocessor scheduling should be done. As always, it is not easy and sometimes impossible to put theoretical results into practice, especially in production environments. Note, however, that all of the following suggested approaches have been demonstrated to be feasible through prototype implementations and experimentation.

Recommendation 1: Provide system support for parallel job preemption. Preemption is crucial to obtaining the best performance. However, this should be coordinated across the nodes running the job. One such form of coordination is gang scheduling, where all the threads run simultaneously on their respective nodes. Alternatively, rate-equivalent scheduling can be used, meaning that all threads get to run for the same fraction of the wallclock time, but not necessarily simultanously. Note that gang scheduling implies rate-equivalent scheduling.

Preemption is also a precondition for changing the number and identity of processors assigned to a job during runtime which is desirable to best handle evolving and malleable jobs, but the marginal gain in performance is not substantial. Hence it is justified only if it can be provided with little additional effort (as a part of checkpointing procedures, for example).

Recommendation 2: Write applications to be moldable and, if it is natural, then to be evolving. Since system loads vary with time, and users generally do not know when they submit a job what the load conditions will be at the time the job is activated, it is desirable that jobs be moldable rather than rigid, so that available processors can be fully exploited at light load, but still efficient use of processors can be assured at heavy load.

If jobs are naturally evolving (such as a cyclic fork join structure, with relatively long sequential periods), then writing the job as evolving (with annotations or commands to dynamically acquire and release processors) makes it possible to greatly increase the efficiency with which the job utilizes the processors assigned to it.

Writing jobs to be malleable is much more work, and this is typically justified only for applications that consume a significant portion of a system's capacity, because they are either very large or invoked very frequently.

Recommendation 3: When system efficiency is of utmost importance, then base processor allocations on both job characteristics and the current load on the system. Jobs make more efficient use of their assigned processors when they have fewer than when they have more. Hence, as the workload volume increases, it may be necessary to reduce the number of processors assigned on average to each job. At light load, processor availability is not an issue, so each job can be given as many processors as it can use, even if they are not being used at high efficiency. At heavy load, the multiprocessing overhead merely detracts from the overall system capacity, so giving jobs a small number of processors (even one in the limit as long as memory requirements don't preclude this extreme possibility) is the most appropriate action. By doing this, the throughput capacity of the system can be maximized.

When specific processor allocations are selected by users, they tend to be overly aggressive or optimistic. The numbers selected by users are typically suitable for light load conditions, but they lead to unacceptably low processor efficiency at heavy load. Consider a case where there are N statistically identical jobs to run on P processors. Assuming the jobs are moldable, the scheduler has the options to either (1) run them one at a time with all P processors, or (2) run them in pairs with half the processors each. Both the mean and the variance of the response times are lower with the latter approach unless [76]:

$$S(P) > \left[2 - \frac{2}{N+2}\right] \cdot S(P/2)$$

This condition seldom holds when either the number of processors or the number of jobs is moderately large.

Since users cannot practically know the load on the system at the time they submit a job, it is best if they identify a range of acceptable processor allocations, and then leave the choice within that range to the scheduler. The current workload volume can be taken into account either by just observing the occupancy of the queues in which jobs await initiation, or by tracking some prediction of overall load as it varies in daily or weekly cycles.

Recommendation 4: To improve average response times, give priority to jobs that are most likely to complete soon, using preemption when necessary. In uniprocessor scheduling, it is known that RR scheduling protects against highly variable service time distributions by making average response time independent of the service time distribution (assuming preemption overhead is negligible). Further, if the service time distribution is know to have high variability, then feedback (FB) disciplines can exploit this, and yield lower average response times as the variability of the service time distribution grows [10].

When no knowledge of service times is available and malleability can be exploited, the ideal EQUI discipline, which attempts to assign an equal number of processors to each job available for execution is optimal. EQUI is analogous to RR in a uniprocessor context in its ability to schedule relatively well even with no service time knowledge. If malleability is impractical due to lack of system support or jobs aren't written to exploit it, then some form of preemptive scheduling based on time-slicing, such as gang-scheduling, should be used.

In current practice, if queues for jobs with smaller execution times tend to have higher priority, then this is consistent with the idea of using available service time knowledge to favor the jobs that are expected to complete most promptly. If better knowledge of job service times than queue identities is available, then it is best to try to activate the jobs in order of increasing expected remaining service time [63]. If the service times are known to be highly variable, but the service times of individual jobs cannot be predicted in advance, then the discipline that executes the job with least acquired service first is best because it emulates the behavior of least expected remaining work first.

Recommendation 5: Make use of information about job characteristics that is either provided directly, or measured, or remembered. Users already provide information about the execution characteristics of their jobs, in the encoded form of a queue identifier. User supplied estimates cannot be directly believed, but the information is generally positively correlated with truth, and that is sufficient to make better scheduling possible. (A good scheduling policy will penalize users who intentionally misestimate the characteristics of the jobs they submit.)

Assuming malleable jobs, some job characteristics (such as efficiency) can be measured while the job is executing and the system can take appropriate action with respect to giving additional processors, or taking some away from the job. Finally, if some historical information is retained, then observed behavior of previous jobs with certain characteristics can be used to predict (approximately) the behavior of new jobs with similar characteristics.

Recommendation 6: Develop New Models Based on the behavior and shortcomings of real machines, new models should capture relevant aspects such as the following:

1. different preemption penalty costs associated with local preemption and job migration,
2. a relation between execution time and allocated processors for moldable, evolving, and malleable jobs,
3. prevention of job starvation by guaranteeing a completion time for each job at the submission time of the job,
4. pricing policies that are based on some combination of resource consumption by the job, and job characteristics that may or may not be known at the time the job is submitted,
5. cyclic load patterns that motivate delaying some large jobs to time periods of lower overall demand (e.g., "off hours").

4 The PSCHED Standard Proposal

Theoretical research like that described in Section 2 tends to focus on algorithmics and easily measurable metrics, while abstracting away from the details. System administrators, on the other hand, cannot abstract away from real-life concerns. They are also faced with unmeasurable costs and constraints, such as interoperability (will machines work together?) and software lifetime (how soon will parts of the system need to be replaced?). Moreover, achieving the maturity and stability required of production software is much harder than building a prototyope. Finally, they need to cater to users and administrators with many different needs, leading to the creation of rather elaborate systems [4,44].

As a result of such concerns, there is much interest in standardizing various software components. In recent years, message passing libraries were standardized through the MPI effort. Similarly, the PSCHED proposal aims at standardizing the interactions among various components involved in parallel job scheduling.

4.1 Background

Deferred processing of work under the control of a scheduler has been a feature of most proprietary operating systems from the earliest days of multi-user systems in order to maximize utilization of the computer.

The arrival of the UNIX system proved to be a dilemma to many hardware providers and users because it did not include the sophisticated batch facilities offered by the proprietary systems. This omission was rectified in 1986 by NASA Ames Research Center who developed the Network Queuing System (NQS) as a portable Unix application that allows the routing and processing of batch "jobs" in a network. To encourage its usage, the product was later put into the public domain.

The supercomputing technical committee began as a "Birds Of a Feather" (BOF) at the January 1987 Usenix meeting. There was enough general interest to form a supercomputing attachment to the /usr/group working groups. The /usr/group working groups later turned into the IEEE POSIX standard effort.

Due to the strong hardware provider and customer acceptance of NQS, it was decided to use NQS as the basis for the POSIX Batch Environment amendment in 1987. Other batch systems considered at the time included CTSS, MDQS, and PROD. None were thought be have both the functionality and acceptability of NQS. This effort was finally approved as a formal standard on December 13, 1994 as IEEE POSIX 1003.2d. The standard committee decided to postpone addressing issues such as programmatic interface and resource control. The supercomputing working group has since been inactive.

PBS was developed at NASA Ames Research Center as a second generation batch queue system that conforms to the IEEE Std. 1003.2d-1994. The project started in June 1993, and was first released in June 1994 [35].

However, both NQS and PBS were designed to schedule serial jobs, and have no understanding of the needs of parallel jobs. The only support for parallelism is regarding "processors" as another resource during allocation, on the same standing as time, memory, or software licenses. To run efficiently, all parts of a parallel job needed to be scheduled to run at the same time. Without support from the batch queue system, most of the large installation of MPP systems had reverted to space slicing and an "all jobs run to completion" policy.

4.2 Outline of Psched

The idea of creating a metacenter is the force behind the PSCHED project at NASA Ames Research Center. A metacenter is a computing resource where jobs can be scheduled and run on a variety of machines physically located in different facilities [34]. This concept ran into several road blocks:

- Some schedulers are tightly integrated with the message passing library: Condor and PVM.
- Almost all schedulers are tightly integrated with the batch queue system.
- Lack of support for parallel jobs.

The Numerical Aerospace Simulation facility (NAS), as part of a Cooperative Research Agreement involving several NASA centers, IBM, Pratt and Whitney, Platform Computing and others, has formed an informal group with the goal of developing a set of "standard" API calls relating to job and resource management systems. The goal of the PSCHED API is to allow a site to write a scheduler that could schedule a variety of parallel jobs: MPI-2, PVM, and SMP multi-tasking jobs to run on a collection of different machines.

To achieve this goal, we intend to standardize the interfaces between the different modules: message passing libraries, task manager, resource manager, and scheduler (see Fig. 2). The specific roles of these components are

Task Manager: An entity that provides task management services such as: spawn a task on a node, local or remote; deliver a signal from one task to another task within the same parallel application; and interface with a resource management function to provide information about nodes assigned to the set of tasks which make up a parallel application, to obtain additional resources (nodes), to free resources (nodes) no longer required, and to notify tasks of the need to checkpoint, suspend, and/or migrate.

Resource Manager: An entity that provides resource management services such as: monitor the resources available in the system, reserve or allocate resources for tasks, and release or deallocate resources no longer needed by tasks.

Scheduler: An entity that schedules jobs. The scheduler is responsible for determining which task should be run on the system according to some site specific policy and the resources available in the system.

The PSCHED API is not an effort to standardize how any of these modules should be implemented. It is an effort to identify the minimal functionality needed from each module and then standardize its interface. For example, the scheduler is a user of the interfaces provided by the task manager and the resource manager. The scheduler waits for scheduling events from the task manager.

The PSCHED API is divided into two areas:

- A set of calls for use by parallel processing jobs to spawn, control, monitor, and signal tasks under the control or management of the job/resource management system. This set of calls should meet the needs of MPI-II, PVM, and other message passing implementations.

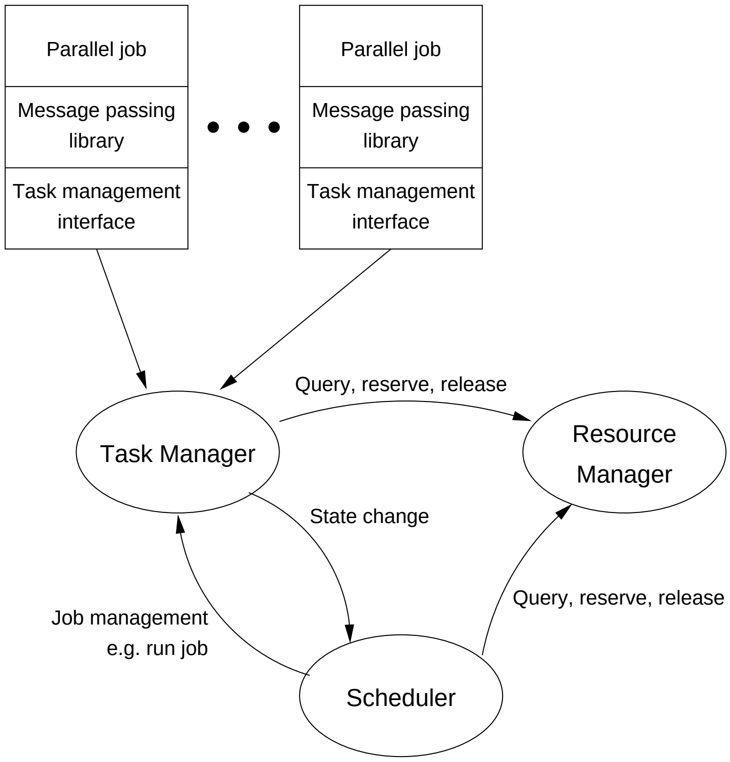

Fig. 2. *Components of the PSCHED environment.*

– A set of calls to be used by batch job schedulers. These calls will allow the development of consistent job/resource schedulers independent of the job/resource management system used. The calls are intended to provide a standard means of obtaining information about the resources available in the processing environment and about the supply of jobs and their requirements.

Let us take a look at an example of how a parallel job would spawn a sub-task, adding more nodes to the running job. The job will call the message passing library, for example `MPI_Spawn` in MPI-II. The message passing library will interface with the task manager to spawn the new task to add more nodes to the existing task. The task manager will inform the scheduler of the request. The scheduler will make a decision based on its scheduling policy. If the policy allows the job to expand, the scheduler will request additional resources from the resource manager, then inform the task manager to start the new sub-task and allow the job to proceed.

```
main()
{
    tm_handle handle[3];

    /* connect to 3 different machines */
    tm_connect(server_1, NULL, &handle[0]);
    tm_connect(server_2, NULL, &handle[1]);
    tm_connect(server_3, NULL, &handle[2]);

    while (1) {
        /* wait for events from any of the servers */
        tm_get_event(handle, 3, &which_handle, &event, &job_id, &args);
        ack = process_event(handle[which_handle], event, job_id, args);
        /* acknowledge the event */
        tm_ack_event(handle[which_handle], event, job_id, ack);
    }
}

process_event(handle, tm_event, job_id, ...)
{
    switch (tm_event) {
        PSCHED_EVENT_JOB_ARRIVED:
            /* call policy routine */
            scheduler_policy(job_id, &run_this_job, resource_needed);
            /* if decided to run job, reserve the resources needed
             * and run the job */
            if (run_this_job) {
                rm_reserve(resource_list, &resource);
                tm_run_job(handle, job_id, resource);
            }
            break;
        PSCHED_EVENT_JOB_EXITED:
            /* release the reources */
            if (resource != PSCHED_RESOURCE_NULL)
                rm_release(resource);
            /* pick a new job to run */
            break;
        PSCHED_EVENT_YIELD:
            /* job is ready to yield some resource, determine
             * whether we want to shrink or expand the job.
             * call the task manager if any action is taken. */
            break;
        PSCHED_EVENT_CHECKPOINT:
            /* a good time to migrate the job if we wanted to */
            break;
        PSCHED_EVENT_REQUEUED:
            /* pick a new job to run */
            break;
```

Fig. 3. *Example skeleton of PSCHED code components.*

```
        PSCHED_EVENT_ADD_RESOURCE:
            /* run policy routines */
            scheduler_policy(job_id, &run_this_job, additional_resource);
            if (run_this_job) {
                    rm_reserve(resource_list, &resource);
                    tm_add_resource(handle, job_id, resource);
            } else {
                    /* tell server we can't fulfill the request
                     * or suspend the run and wait for the resource */
            }
            break;
        PSCHED_EVENT_RELEASE_RESOURCE:
            rm_release(rm_handle, resource);
            break;
        default:
            return UNKNOWN_EVENT;
    }
    return SUCCESS;
}

scheduler_policy(...)
{
    /* this is what the scheduler writters will concentrate on */
}
```

Fig. 3. (cont.)

4.3 Implication on the Programming and Scheduling of Parallel Jobs

Obvious benefits of a standard like PSCHED include:

- real traces can be used in simulations to develop better algorithms
- new algorithms could be directly applied to running systems
- modularity of very complex pieces of software allows a mix and match of:
  - batch / task management system
  - scheduler
  - communication library (e.g., MPI, PVM)
  - scheduling simulators

The PSCHED API will be flexible enough to address some of the problems identified in Section 3 such as shrinking and expanding jobs, checkpointing and migrating jobs.

Hopefully this set of "standard" interfaces will free researchers from the need to port their work to different systems and let them concentrate on innovative scheduling algorithm and scheduler design. This will also make production machines more readily available for researchers. An example of such a scheduler is given in Fig. 3. Once written it should be very easily ported to another environment. The tm_ and rm_ calls are interfaces to the task manager and the resource manager respectively.

Areas that need standardization but are not currently addressed by PSCHED include:

- Moving jobs from one batch queue system to another.
- The accounting information kept by the batch queue system.

5 Discussion and Conclusions

The relationship between theory and practice is an interesting one. Sometimes theory is ahead of practice, and suggests novel approaches and solutions that greatly enhance the state of the art. Sometimes theory straggles behind, and only provides belated justification for well known practices. It is not yet clear what role it will play in the field of parallel job scheduling.

The question of how much theory contributes to practice also depends on the metrics used to measure performance and quality. In the field of job scheduling, the three most common metrics are throughput, utilization, and response time. Throughput and utilization are actually related to each other: if we assume that the statistics of the workload are essentially static, then executing more jobs per unit time on average also leads to a higher utilization. This can go on until the system saturates. If users are satisfied with the system, and the system does not saturate, more jobs will be submitted, leading to higher utilization and throughput. The role of the scheduler is therefore to delay the onset of saturation, by reducing fragmentation and assuring efficient usage of processors [25]. Also, good support for batch jobs can move some of the load to off hours, further increasing the overall utilization.

In practice utilization is a very commonly used metric, as it is easy to measure and reflects directly on the degree to which large investments in parallel hardware are used efficiently. Throughput figures are hardly ever used. Reported utilization figures vary from 50% for the NASA Ames iPSC/860 hypercube [21], through around 70% for the CTC SP2 [37], 74% for the SDSC Paragon [92] and 80% for the Touchstone Delta [54], up to more than 90% for the LLNL Cray T3D [20]. Utilization figures in the 80–90% range are now becoming more common, due to the use of more elaborate batch queueing mechanisms [49,79,92] and gang scheduling [20]. These figures seem to leave only little room for improvement.

However, it should be noted that these figures only reflect one factor contributing to utilization. The real utilization of the hardware is the product of two factors: the fraction of PEs allocated to users, and the efficiency with which these PEs are used. The figures above relate to the first factor, and depend directly on the scheduling policies; they show that current systems can allocate nearly all the resources, with little loss to fragmentation. But the efficiency with which the allocated resources are used depends more on the application being run, and can be quite low. However, the system can still have an effect, because in most applications the efficiency trails off as processors are added. Thus allocating less processors under high loads should improve the second factor, and lead to higher overall utilization [43,68,51]. This is possible with moldable or malleable jobs, but not with rigid ones.

The case of the response time metric is more complex, because little direct evidence exists. Theory suggests that preemption be used to ensure good response times for small jobs [64], especially since workloads have a high variability in computational requirements [21]. This comes close on the heels of actual systems that implement gang scheduling for just this reason [46,32,27,20].

Actually two metrics may be used to gauge the responsiveness of a system: the actual response time (or turnaround time, i.e. the time from submittal to termination), or the slowdown (the ratio of the response time on a loaded system to the response time on a dedicated system). Using actual response times places more weight on long jobs, and "doesn't care" if a short job waits a few minutes, so it may not reflect the users' notion of responsiveness. Slowdown reflects the rather reasonable notion that responsiveness should be measured against requirements, meaning that users should expect their jobs to take time that is proportional to the computation performed. However, for very short jobs, the denominator becomes very small, leading to a large slowdown, even though the actual response time may be quite short, well within the interactive range. It may therefore be best to combine the two metrics. Let T represent the response time on the loaded system, T_d the response time on a dedicated system, and T_h the threshold of interactivity (i.e. the time users are willing to wait). The combined metric for responsiveness as percieved by users would then be

$$R = \begin{cases} T/T_d & \text{if } T_d > T_h \\ T/T_h & \text{if } T_d < T_h \end{cases}$$

For long jobs, this is the normal slowdown. For short jobs, this is the slowdown *relative to the interactivity threshold*, rather than relative to the very short runtime on a dedicated system. If we use T_h as the unit of time, then for short jobs the expression degenerates to the response time. We suggest the name "bounded slowdown" for this metric, as it is similar to the slowdown metric, but bounded away from high values for very short jobs.

Two possible roles for theory, that have relatively few parallels in practice, are how to use knowledge about specific jobs [76], and how to tune algorithmic parameters [93]. In practice, knowledge about jobs is limited to that supplied by the users, typically in the form of choosing a queue with a certain combination of resource limits. This approach has two main drawbacks: first, it leads to a combinatorical explosion of queues, that are hard to deal with. Second, even with very many queues, the resolution in which requirements are expressed is necessarily very coarse, and user estimates are notoriously inaccurate anyway. Recent more theoretical work shows how data can be acquired automatically by the system, rather than relying on the users [59,31,12].

At the same time that theoretical work is focusing, at least to some degree, on practical concerns, practice in the field seems to be rather oblivious of this development. One reason is that the larger and more advanced installations have been developing rather elaborate scheduling facilities, which achieve reasonable results, so the pressure for searching for additional improvements outside is diminished. Another reason is the overwhelming concern for backwards compatability, portability, and interoperability, which leads to standards based on common practices and discourages innovations. It should be hoped, however, that the developed standards will be flexible enough to allow unanticipated advances to be incorporated in the future.

References

1. I. Ahmad, "*Editorial: resource management of parallel and distributed systems with static scheduling: challenges, solutions, and new problems*". *Concurrency — Pract. & Exp.* **7(5)**, pp. 339–347, Aug 1995.
2. G. Alverson, S. Kahan, R. Korry, C. McCann, and B. Smith, "*Scheduling on the Tera MTA*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 19–44, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
3. S. V. Anastiadis and K. C. Sevcik, "*Parallel application scheduling on networks of workstations*". *J. Parallel & Distributed Comput.*, Jun 1997. (to appear).
4. 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.
5. T. B. Brecht, "*An experimental evaluation of processor pool-based scheduling for shared-memory NUMA multiprocessors*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, 1997. Lecture Notes in Computer Science (this volume).
6. J. Bruno, E. G. Coffman, Jr., and R. Sethi, "*Scheduling independent tasks to reduce mean finishing time*". *Comm. ACM* **17(7)**, pp. 382–387, Jul 1974.
7. S. Chakrabarti, C. Phillips, A. S. Achulz, D. B. Shmoys, C. Stein, and J. Wein, "*Improved approximation algorithms for minsum criteria*". In *Intl. Colloq. Automata, Lang., & Prog.*, pp. 646–657, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1099.
8. S-H. Chiang, R. K. Mansharamani, and M. K. Vernon, "*Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 33–44, May 1994.
9. S-H. Chiang and M. K. Vernon, "*Dynamic vs. static quantum-based parallel processor allocation*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 200–223, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.
10. R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*. Addison-Wesley, 1967.
11. X. Deng, N. Gu, T. Brecht, and K. Lu, "*Preemptive scheduling of parallel jobs on multiprocessors*". In 7th *SIAM Symp. Discrete Algorithms*, pp. 159–167, Jan 1996.
12. A. B. Downey, "*Using queue time predictions for processor allocation*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, 1997. Lecture Notes in Computer Science (this volume).
13. J. Du and J. Y-H. Leung, "*Complexity of scheduling parallel task systems*". *SIAM J. Discrete Math.* **2(4)**, pp. 473–487, Nov 1989.

14. K. Dussa, K. Carlson, L. Dowdy, and K-H. Park, "*Dynamic partitioning in a transputer environment*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 203–213, May 1990.
15. A. Dusseau, R. H. Arpaci, and D. E. Culler, "*Effective distributed scheduling of parallel workloads*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 25–36, May 1996.
16. D. L. Eager, J. Zahorjan, and E. D. Lazowska, "*Speedup versus efficiency in parallel systems*". *IEEE Trans. Comput.* **38(3)**, pp. 408–423, Mar 1989.
17. D. G. Feitelson, "*Memory usage in the LANL CM-5 workload*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, 1997. Lecture Notes in Computer Science (this volume).
18. D. G. Feitelson, "*Packing schemes for gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 89–110, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.
19. D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems*. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.
20. D. G. Feitelson and M. A. Jette, "*Improved utilization and responsiveness with gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, 1997. Lecture Notes in Computer Science (this volume).
21. 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.
22. 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.
23. D. G. Feitelson and L. Rudolph, "*Evaluation of design choices for gang scheduling using distributed hierarchical control*". *J. Parallel & Distributed Comput.* **35(1)**, pp. 18–34, May 1996.
24. 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.
25. D. G. Feitelson and L. Rudolph, "*Toward convergence in job schedulers for parallel supercomputers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–26, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.
26. A. Feldmann, J. Sgall, and S-H. Teng, "*Dynamic scheduling on parallel machines*". *Theoretical Comput. Sci.* **130(1)**, pp. 49–72, Aug 1994.
27. H. Franke, P. Pattnaik, and L. Rudolph, "*Gang scheduling for highly efficient distributed multiprocessor systems*". In 6th *Symp. Frontiers Massively Parallel Comput.*, pp. 1–9, Oct 1996.
28. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.
29. D. Ghosal, G. Serazzi, and S. K. Tripathi, "*The processor working set and its use in scheduling multiprocessor systems*". *IEEE Trans. Softw. Eng.* **17(5)**, pp. 443–453, May 1991.
30. R. Gibbons, *A Historical Application Profiler for Use by Parallel Schedulers*. Master's thesis, Dept. Computer Science, University of Toronto, Dec 1996. Available as Technical Report CSRI-TR 354.

31. R. Gibbons, "*A historical application profiler for use by parallel schedulers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, 1997. Lecture Notes in Computer Science (this volume).
32. B. Gorda and R. Wolski, "*Time sharing massively parallel machines*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 214–217, Aug 1995.
33. R. L. Graham, "*Bounds on multiprocessing timing anomalies*". *SIAM J. Applied Mathematics* **17(2)**, pp. 416–429, Mar 1969.
34. A. S. Grimshaw, J. B. Weissman, E. A. West, and E. C. Loyot, Jr., "*Metasystems: an approach combining parallel processing and heterogeneous distributed computing systems*". *J. Parallel & Distributed Comput.* **21(3)**, pp. 257–270, Jun 1994.
35. 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.
36. 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.
37. S. Hotovy, "*Workload evolution on the Cornell Theory Center IBM SP2*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 27–40, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.
38. N. Islam, A. Prodromidis, and M. S. Squillante, "*Dynamic partitioning in different distributed-memory environments*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 244–270, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.
39. J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riodan, "*Modeling of workload in mpps*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, 1997. Lecture Notes in Computer Science (this volume).
40. *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162. *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1997. Lecture Notes in Computer Science (this volume).
41. T. Kawaguchi and S. Kyan, "*Worst case bound of an LRF schedule for the mean weighted flow-time problem*". *SIAM J. Comput.* **15(4)**, pp. 1119–1129, Nov 1986.
42. H. Kellerer, T. Tautenhahn, and G. J. Wöginger, "*Approximability and nonapproximability results for minimizing total flow time on a single machine*". In 28th *Ann. Symp. Theory of Computing*, pp. 418–426, 1996.
43. R. Krishnamurti and E. Ma, "*An approximation algorithm for scheduling tasks on varying partition sizes in partitionable multiprocessor systems*". *IEEE Trans. Comput.* **41(12)**, pp. 1572–1579, Dec 1992.
44. R. N. Lagerstrom and S. K. Gipp, "*PScheD: political scheduling on the CRAY T3E*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, 1997. Lecture Notes in Computer Science (this volume).
45. W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph, "*Implications of I/O for gang scheduled workloads*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, 1997. Lecture Notes in Computer Science (this volume).

46. C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong-Chan, S-W. Yang, and R. Zak, "*The network architecture of the Connection Machine CM-5*". *J. Parallel & Distributed Comput.* **33(2)**, pp. 145–158, Mar 1996.
47. S. Leonardi and D. Raz, "*Approximating total flow time on parallel machines*". In 29th *Ann. Symp. Theory of Computing*, 1997.
48. S. T. Leutenegger and M. K. Vernon, "*The performance of multiprogrammed multiprocessor scheduling policies*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 226–236, May 1990.
49. 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.
50. W. Ludwig and P. Tiwari, "*Scheduling malleable and nonmalleable parallel tasks*". In 5th *SIAM Symp. Discrete Algorithms*, pp. 167–176, Jan 1994.
51. 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.
52. C. McCann and J. Zahorjan, "*Scheduling memory constrained jobs on distributed memory parallel computers*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 208–219, May 1995.
53. R. McNaughton, "*Scheduling with deadlines and loss functions*". *Management Science* **6(1)**, pp. 1–12, Oct 1959.
54. P. Messina, "*The Concurrent Supercomputing Consortium: year 1*". *IEEE Parallel & Distributed Technology* **1(1)**, pp. 9–16, Feb 1993.
55. R. Motwani, S. Phillips, and E. Torng, "*Non-clairvoyant scheduling*". *Theoretical Comput. Sci.* **130(1)**, pp. 17–47, Aug 1994.
56. V. K. Naik, S. K. Setia, and M. S. Squillante, "*Performance analysis of job scheduling policies in parallel supercomputing environments*". In *Supercomputing '93*, pp. 824–833, Nov 1993.
57. R. Nelson, D. Towsley, and A. N. Tantawi, "*Performance analysis of parallel processing systems*". *IEEE Trans. Softw. Eng.* **14(4)**, pp. 532–540, Apr 1988.
58. T. D. Nguyen, R. Vaswani, and J. Zahorjan, "*Parallel application characterization for multiprocessor scheduling policy design*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 175–199, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.
59. T. D. Nguyen, R. Vaswani, and J. Zahorjan, "*Using runtime measured workload characteristics in parallel processor scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 155–174, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.
60. J. D. Padhye and L. Dowdy, "*Dynamic versus adaptive processor allocation policies for message passing parallel computers: an empirical comparison*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 224–243, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.
61. E. W. Parsons and K. C. Sevcik, "*Benefits of speedup knowledge in memory-constrained multiprocessor scheduling*". *Performance Evaluation* **27&28**, pp. 253–272, Oct 1996.
62. E. W. Parsons and K. C. Sevcik, "*Coordinated allocation of memory and processors in multiprocessors*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 57–67, May 1996.

63. E. W. Parsons and K. C. Sevcik, "*Implementing multiprocessor scheduling disciplines*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, 1997. Lecture Notes in Computer Science (this volume).
64. 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.
65. V. G. J. Peris, M. S. Squillante, and V. K. Naik, "*Analysis of the impact of memory in distributed parallel processing systems*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 5–18, May 1994.
66. J. Pruyne and M. Livny, "*Managing checkpoints for parallel programs*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 140–154, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.
67. 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.
68. 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.
69. 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.
70. S. K. Sahni, "*Algorithms for scheduling independent tasks*". *J. ACM* **23(1)**, pp. 116–127, Jan 1976.
71. U. Schwiegelshohn, "*Preemptive weighted completion time scheduling of parallel jobs*". In 4th *European Symp. Algorithms*, pp. 39–51, Springer-Verlag, Sep 1996. Lecture Notes in Computer Science Vol. 1136.
72. U. Schwiegelshohn, W. Ludwig, J. L. Wolf, J. J. Turek, and P. Yu, "*Smart SMART bounds for weighted response time scheduling*". *SIAM J. Comput.* To appear.
73. S. K. Setia, "*The interaction between memory allocation and adaptive partitioning in message-passing multicomputers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 146–165, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
74. S. K. Setia, M. S. Squillante, and S. K. Tripathi, "*Analysis of processor allocation in multiprogrammed, distributed-memory parallel processing systems*". *IEEE Trans. Parallel & Distributed Syst.* **5(4)**, pp. 401–420, Apr 1994.
75. S. K. Setia, M. S. Squillante, and S. K. Tripathi, "*Processor scheduling on multiprogrammed, distributed memory parallel computers*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 158–170, May 1993.
76. K. C. Sevcik, "*Application scheduling and processor allocation in multiprogrammed parallel processing systems*". *Performance Evaluation* **19(2-3)**, pp. 107–140, Mar 1994.
77. 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.
78. D. Shmoys, J. Wein, and D. Williamson, "*Scheduling parallel machines on-line*". *SIAM J. Comput.* **24(6)**, pp. 1313–1331, Dec 1995.

79. J. Skovira, W. Chan, H. Zhou, and D. Lifka, "*The EASY - LoadLeveler API project*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 41–47, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.
80. D. D. Sleator and R. E. Tarjan, "*Amortized efficiency of list update and paging rules*". *Comm. ACM* **28(2)**, pp. 202–208, Feb 1985.
81. E. Smirni, E. Rosti, G. Serazzi, L. W. Dowdy, and K. C. Sevcik, "*Performance gains from leaving idle processors in multiprocessor systems*". In *Intl. Conf. Parallel Processing*, vol. III, pp. 203–210, Aug 1995.
82. W. Smith, "*Various optimizers for single-stage production*". *Naval Research Logistics Quarterly* **3**, pp. 59–66, 1956.
83. P. G. Sobalvarro and W. E. Weihl, "*Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 106–126, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
84. M. S. Squillante, "*On the benefits and limitations of dynamic partitioning in parallel computer systems*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 219–238, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
85. M. S. Squillante, F. Wang, and M. Papaefthymiou, "*Stochastic analysis of gang scheduling in parallel and distributed systems*". *Performance Evaluation* **27&28**, pp. 273–296, Oct 1996.
86. 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.
87. J. Turek, W. Ludwig, J. L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelshohn, and P. S. Yu, "*Scheduling parallelizable tasks to minimize average response time*". In 6th *Symp. Parallel Algorithms & Architectures*, pp. 200–209, Jun 1994.
88. J. Turek, J. L. Wolf, and P. S. Yu, "*Approximate algorithms for scheduling parallelizable tasks*". In 4th *Symp. Parallel Algorithms & Architectures*, pp. 323–332, Jun 1992.
89. J. J. Turek, U. Schwiegelshohn, J. L. Wolf, and P. Yu, "*Scheduling parallel tasks to minimize average response time*". In 5th *ACM-SIAM Symp. Discrete Algorithms*, pp. 112–121, Jan 1994.
90. C. A. Waldspurger and W. E. Weihl, "*Lottery scheduling: flexible proportional-share resource management*". In 1st *Symp. Operating Systems Design & Implementation*, pp. 1–11, USENIX, Nov 1994.
91. C. A. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. Ph.D. dissertation, Massachusetts Institute of Technology, Technical Report MIT/LCS/TR-667, Sep 1995.
92. M. Wan, R. Moore, G. Kremenek, and K. Steube, "*A batch scheduler for the Intel Paragon with a non-contiguous node allocation algorithm*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 48–64, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.
93. F. Wang, H. Franke, M. Papaefthymiou, P. Pattnaik, L. Rudolph, and M. S. Squillante, "*A gang scheduling design for multiprogrammed parallel computing environments*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 111–125, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

94. F. Wang, M. Papaefthymiou, and M. Squillante, "*Performance evaluation of gang scheduling for parallel and distributed multiprogramming*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, 1997. Lecture Notes in Computer Science (this volume).
95. Q. Wang and K. H. Cheng, "*A heuristic of scheduling parallel tasks and its analysis*". *SIAM J. Comput.* **21(2)**, pp. 281–294, Apr 1992.
96. K. K. Yue and D. J. Lilja, "*Loop-level process control: an effective processor allocation policy for multiprogrammed shared-memory multiprocessors*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 182–199, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
97. J. Zahorjan and C. McCann, "*Processor scheduling in shared memory multiprocessors*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 214–225, May 1990.