Job Scheduling on Parallel Systems

Jonathan Weinberg

University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0505

Abstract

Parallel systems such as supercomputers are valuable resources which are each commonly shared among a community of users. The problem of job scheduling is to determine how that sharing should be done in order to maximize the system's utility. This problem has been extensively studied for well over a decade, yielding a great breadth of knowledge and techniques. In this work, we survey the ideas and approaches that have proven most influential to how jobs are scheduled on today's large-scale parallel systems. With this background in mind, we discuss how deployed scheduling policies can be improved to meet existing requirements and how trends in parallel processing are currently altering those requirements.

1 Introduction

Parallel systems, such as supercomputers, are valuable resources that are commonly shared by communities of users. Users continually submit jobs to the system, each with unique resource and service-level requirements as well as value to the user and resource owner. The charge of job scheduling is to decide when and how each job should execute in order to maximize the system's aggregate utility to its owners.

For well over a decade, the field of job scheduling has been the subject of great scrutiny, producing a sizeable body of work [20] and increasing returns on HPC investments by millions of dollars. Despite this progress, one may argue that the problem of scheduling on parallel systems may not be closer to being solved today than it was a decade ago. Scheduling is an inherently reactive discipline, mirroring trends in HPC architectures, parallel programming language models, user demographics, and administrator priorities. No scheduling strategy is optimal for all of today's scenarios, let alone all of tomorrow's.

In recent years, relative stability in the aforementioned forces has gradually moved large supercomputer installations towards workable though imperfect de facto standards. The production of large Massively Parallel Processing (MPP) machines today centers around MIMD architectures [23] in pure or shared distributed memory configurations, such as Cache Coherent Non-Uniform Memory Access. This architectural trend has given rise to the dominance of rigid programming models such as MPI, and consequently of complementary scheduling policies such as batch queued space-sharing and its variants.

Nevertheless, as parallel processing is increasingly heralded as the savior of Moore's Law, a shift in user demographics, system architectures, and programming models is underway. No longer the exclusive realm of supercomputers, parallel processing is moving onto commodity clusters, geographically distributed *grid* ensembles, and even the desktop. Concurrently, the rigidity and explicit parallelism of MPI is slowly giving way to alternative programming models which challenge traditional scheduling assumptions.

In this work, we illuminate the issues and approaches that have defined how parallel jobs are scheduled in today's production environments and highlight as yet unresolved issues in the field. Before this backdrop, we discuss current trends in parallel processing and emphasize their implications for the future of job scheduling.

2 Definitions and Assumptions

Because the terms involved have held multiple meanings and implied various assumptions over time, we define them here for this survey.

This work is concerned primarily with *job scheduling*, a discipline whose purpose is to decide when and where each job should be executed from the perspective of the system. This is related to, but distinct from, the study of *application/task scheduling* [39, 31] where the question is how a single parallel application should schedule each of its threads.

We use the term *supercomputer* to reference a computer capable of massively parallel processing and shared among a community of users. Such machines today typically scale to thousands of processors that often cooperate in a Multiple Instruction Multiple Data (MIMD) architecture [23] and offer a pure distributed memory or distributed shared memory interface. The processor pools of such machines are normally homogenous.

We use the term *job* to refer to some parallel program, composed of multiple concurrent *threads*, submitted to the system for execution. As such, a job is inherently associated with a submission time and scheduling algorithms must make *online* scheduling decisions based on the current state of an ordered job stream.

Each job is characterized along two dimensions: its *length* as measured by execution time and its *width* or *size* as measured by the number of threads; we will assume that each of a job's threads executes on a separate processor.

Job sizes are not necessarily fixed before or during execution, but are often so in practice. The literature categorizes job sizes according to the following behavioral taxonomy [21]:

Rigid - The number of processors made available to the job is specified by the user and is external to the scheduler. Exactly that number of processors is made available to the job throughout its execution

Moldable - The number of processors assigned to the job is determined by the scheduler but within certain constraints, potentially provided by the user. Once the job begins, it uses the same number of processors throughout its execution.

Evolving - The job goes through different phases that require different numbers of processors. The number of processors allocated may change during execution in response to the job requesting or relinquishing some.

Malleable - The number of processors assigned to the job may change during execution at the discretion of the scheduler.

In production environments, jobs are almost always rigid, though research has shown that moldable jobs can increase throughput in a variety of scheduling approaches [58, 9]. Evolving and malleable jobs are not widely supported due to increased programming burden, migration and checkpointing overheads, and lack of operating system support.

Given a set of jobs and of available processors, a *schedule* is an ordered series of mappings between some threads of those jobs and the processors. In practice, schedules are readily subject to change as dictated by job stream conditions but advanced reservations are sometimes possible [25].

The term *time-sharing* refers to any scheduling approach whereby threads can be preempted by others during execution and restarted later. The number of jobs that each processor can execute concurrently is known as the *multiprogramming level*. Contrastingly, *space-sharing* approaches provide a thread exclusive use of a processor until its execution is complete or a maximum time limit has been eclipsed and the thread is terminated. Space-sharing approaches manage time by placing each job in a queue and executing all of its threads concurrently upon release from that queue.

This divide in approaches reflects a duality of job requirement sets. *Interactive* jobs that require low latency are usually executed using time-sharing, while *batch* jobs that require unperturbed performance are executed on dedicated processors using space-sharing. Supercomputing facilities often meet the requirements of both categories by statically partitioning a machine's processors into time-sharing and space-sharing subsets.

3 What Is a Good Schedule?

For such a well studied problem, the evaluation of scheduling algorithms has proven surprisingly elusive. The objective of scheduling, and of system administration in general, is to maximize system utility. Unfortunately, utility is not directly observable, but rather is some subjective and context-specific function of many factors. Should one job starve to decrease the running time of five others by 10%? How much more productive will a user be if he knew exactly when his job will run? What if he knew within 20%? These determinations must be informed by context well outside the scope of the scheduler and even then, objectivity is impossible.

We can identify certain qualities as desirable for schedulers, including *performance*, *fairness*, and *predictability*. Because these qualities are largely intangible, many observable metrics intended to mirror them, particularly performance, have emerged and enjoyed wide use.

However, consensus exists neither on how these metrics relate to the desired qualities, nor on how these desired qualities relate to utility. It is simply posited that each relates along some unknown, non-decreasing function. Objectively evaluating the effects of scheduling decisions on a specific quality is therefore difficult; evaluating the relative effects of each metric on utility is nearly impossible.

Observable metrics enable us to make only very weak statements regarding scheduling policies, such as "an increase in metric A, while all other factors are held equal, will cause a non-negative change in system utility." Unfortunately, all other factors are rarely held equal; metric A may be in conflict with metric B while their relative effects on utility are unknown. The value of metrics is therefore to support administrative decision-making by describing scheduling tradeoffs in regrettably non-uniform units.

In this section we discuss each of the three desired scheduling qualities mentioned above, some of the metrics that have been used to observe each, and the tradeoffs that exist among them.

3.1 Performance

The most often evaluated scheduling quality is performance. For online scheduling algorithms, this quality is often measured using variations on *response time* [15, 50]. Response time, also known as *flow time* or *turnaround time*, is the amount of time elapsed between a job's submission and completion. The intuition behind this metric is that users are happier with speedier response times, though the exact correlation is not clear [21].

The *slowdown* of a job, sometimes referred to as *stretch*, is the ratio of a job's response time with respect to its runtime on an unloaded system [5]. Feitelson et. al have observed that this metric overemphasizes the importance of extremely short jobs and proposed a *bounded slowdown* [21]:

$$bounded\_slowdown = max\{\frac{T_w + T_r}{max\{T_{ru}, \tau\}}, 1\}$$

where T_w is the job's waiting time, T_r the runtime, T_{ru} the runtime on an unloaded system. Naturally, bounded slowdown is sensitive to the value of τ.

A troublesome quality of both slowdown and bounded slowdown is that jobs with the same response time and processor time can have different slowdown numbers. For example, a job running immediately on 1 processor for 100 seconds will have a slowdown of 1, while a 10 processor job that waits in a queue for 90 seconds and runs for 10 seconds will have a slowdown of 10, even though it has the same 100s response time and utilizes the same 100 cpu seconds. In response, Zotkin et. al. have proposed a *per processor slowdown* (pp-slowdown), derived by simply dividing the slowdown by the number of processors used [66].

Another problem with the slowdown metrics is the tie between performance and job lengths. As T_r and T_{ru} increase, the impact of T_w on the slowdown is diminished. This can have undesirable consequences for polices that try to minimize these metrics. On a space-sharing system for example, such a policy encourages jobs that use fewer processors because they run longer and often start more quickly [8]. This contradiction with the goals of parallel processing has led some to prefer response time to slowdown as a performance metric.

To compare two scheduling schemes, the workload must be held equal. For this reason, many studies use *makespan* [46, 60, 45, 50], a throughput measure that denotes the amount of time required for a particular machine to execute some closed set of jobs. The intuition is that a shorter makespan on some job set might indicate higher throughput in production. Higher throughput presumably means more utility.

Of course, high performance alone is insufficient for high utility. For instance, a scheduler that only chooses fast running jobs that minimize fragmentation in the schedule may easily achieve high throughput and low productivity. This is because it is trading fairness for performance.

3.2 Fairness

Unless told otherwise, a scheduler must assume that utility is best served by providing comparable service to every job. This is because the job mix is a product of external policy and market forces that cannot be anticipated by the scheduler. Providing uneven service levels to certain jobs risks the general utility of the machine. Fairness is therefore an important quality of schedules, though it is seldom quantified.

Most scheduling algorithms make the minimum fairness guarantee that no job will be starved, that is, each job will eventually execute. Stronger fairness guarantees are contingent on the scheduling scheme. In space-sharing, fairness may imply some first-come-first-serve (FCFS) ordering or that a job will not be delayed by any job that is behind it in the queue [38]. In time-sharing, it may be that each thread receives an equal slice of the processor or a slice weighted by the job size [22].

As evidenced by the makespan example above, fairness is often directly at odds with performance and evaluating tradeoffs between the two is difficult.

3.3 Predictability

Predictability is the gap between a job's response or flow time and the user's expectation as created through previous experience. Predictability can indirectly increase productivity by enabling users to anticipate job completion times and plan resource usage accordingly. Some have proposed that predictability, under other realistic assumptions, may be even more central to the user experience than performance [61].

4 Job Scheduling on MPP Supercomputers

The dominant resource for parallel processing in recent years has been the MPP supercomputer. Consequently, parallel job scheduling research in this space has matured into a reasonably understood topic. In this section, we highlight some of the influential issues and ideas that have helped shape this maturation.

4.1 Space-Sharing

The simplest way to schedule a parallel system is with a queue. Each job is submitted to the queue and, upon reaching the head, is executed to completion while all other jobs wait. The queue can be hypothetically FIFO, but the scheme extends to priority queues without loss of generality.

Though providing maximum fairness and predictability, this scheme is inefficient. Since each application utilizes only a subset of the system's processors, those processors not in the subset are left idle during execution. This effect is known as *fragmentation* and its reduction is the primary focus of much scheduling research.

The most natural extension to the queue scheme is space-sharing, which is the simple idea of allowing another job in the queue to execute on the idle processors if enough are available. This is primarily how supercomputers are scheduled today.

Deceptively however, even simple scheduling models such as queued space-sharing hide many assumptions, resulting in keen research interest. In the remainder of this section, we will discuss some of the heuristics used to select the next job to execute and the implications, assumptions, and repercussions of such choices.

4.1.1 Backfilling

The most basic queued space-sharing approach is known as blocking First-Come-First-Serve (FCFS) [44]. Under this scheme, if sufficient idle processors exist to serve the next job in the queue, that job is executed. Otherwise, the queue blocks until sufficient resources become available.

This approach remains prone to severe fragmentation with system utilization rates between 50-80% [30, 16, 27]. Because the queue is only accessed at the head, a wide job may block others behind it from executing while it waits for a large portion of the machine to become available. *Backfilling* is the idea that while the wide job waits, the scheduler may choose to execute some narrower jobs situated further back in the queue. The question is, which job should jump ahead?

The first implementation of a backfilling scheduler was the Extensible Argonne Scheduling sYstem (EASY) [34]. The scheduler was deployed on the Argonne National Laboratory's 128-node IBM SP system and was successful enough to be eventually incorporated into IBM's commercial LoadLeveler scheduling software [28].

EASY backfilling, as it has come to be known, works by allowing a narrower job J_n, to jump in front of a waiting wide job J_w, so long as the execution of J_n does not delay the projected start of J_w. The job furthest ahead in the queue that satisfies these width and length requirements is selected for backfilling.

This scheme relies on a significant assumption, that is, that job lengths are known a priori. Argonne's approach, which preponderates today, was to simply ask the users for an expected runtime. Though this approach has proven serviceable, the problem of job length estimation under disparate assumptions has created an active field of research as described in Section 4.1.2.

The other problem with EASY backfilling is fairness, as cutting can cause unfairness even if not to the job at the head of the queue. This is the fundamental observation motivating *conservative backfilling* [38].

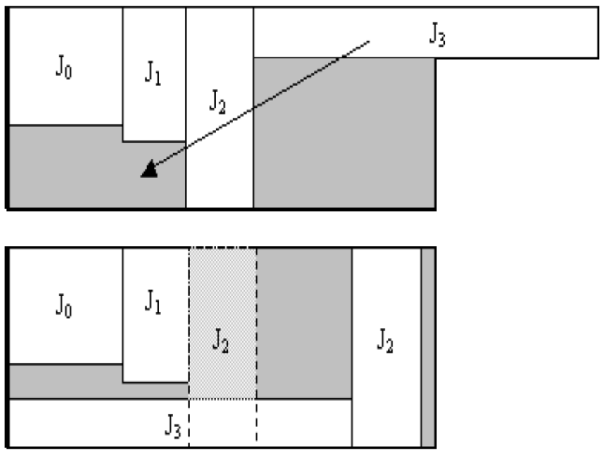

Figure 1. EASY backfilling can cause unfairness.

Figure 1 demonstrates how this unfairness can occur. In the figure, jobs are ordered from left to right and the Y-axis represents the number of processors. Job J_0 is currently executing, leaving too few processors to execute J_1. The first job that can satisfy the length and width requirements of the EASY backfilling algorithm is J_3, so it is scheduled. However, though it has no effect on J_1, the execution of J_3 delays the start time of J_2.

Conservative backfilling only backfills when the maneuver causes no job to be delayed. This approach trades some of the performance of the more aggressive EASY scheme for an increase in fairness, though the extent is naturally workload-dependant [38].

The tradeoffs between conservative and EASY backfilling can be generalized to a *number of reservations*, where EASY backfilling makes a single reservation for the job at the head of the queue and conservative backfilling makes one for each job in the queue. Some schedulers, such as Maui [26], allow administrators to explicitly set this number; Chaing et. al. have suggested that 2-4 is a good compromise between performance and fairness [7]. Alternative approaches such as dynamic reservation policies which make reservations based on observed runtime delays have also been proposed [54].

4.1.2 Estimating Job Lengths

As mentioned in the previous section, backfilling is predicated on knowledge of the job lengths before execution. The approach taken by Lifka's EASY scheduler, to simply ask the users to submit an expected runtime along with the job, is in wide use today. Unfortunately, estimates gathered in this manner are notoriously inaccurate.

Figure 2 displays the percentage of requested runtime actually used by jobs over a two year period at the San Diego Supercomputer Center's IBM SP2 installation. The large spike at 100% is indicative not of accurate estimates, but of jobs being killed upon exceeding their respective runtime estimates. The distribution is almost even across all percentages, meaning that user-supplied runtime estimates are essentially arbitrary.

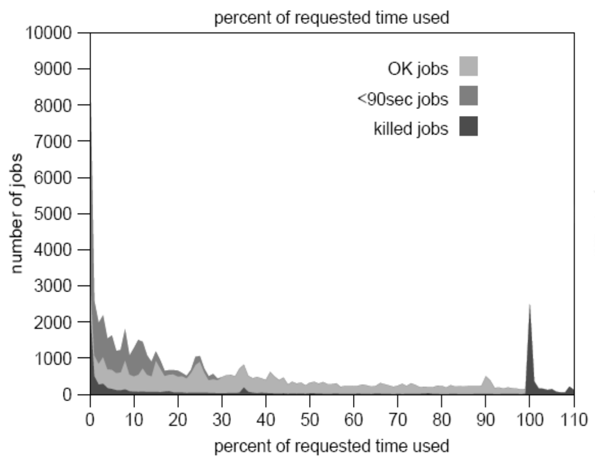

Figure 2. Percent of user runtime estimate used by users at SDSC (128-node IBM SP2, April 1998 - April 2000)

Conventional wisdom has explained this behavior through the *padding hypothesis* which suggests that users overestimate runtimes because their motivation to avoid having their jobs killed is much stronger than their motivation to enable better packing. Subsequent research however, has suggested that padding is not entirely to blame and that users may be unable to provide tight runtime estimates, even when the threat of job termination is removed and a monetary incentive is provided for tight estimates [32].

This has led some to propose that the scheduler should automatically generate runtime estimates instead of users. The most commonly used and successful approach for doing this is through analysis of historical runtime logs. It is well documented that users tend to run identical or similar jobs many times consecutively [16, 13]. A number of experimental results have shown that this phenomenon can be exploited to produce accurate runtime estimates [38, 47, 24].

Nevertheless, no such automated prediction schemes are deployed today. There are several reasons for this. First, if the system underestimates the runtime in a backfilling context, a user's job will be killed. Mu'alem and Feitelson's technique for example, underestimates once for approximately every five predictions [38]. Administrators are therefore justifiably uneasy about adopting such an approach until other assumptions are changed.

Secondly, some have interestingly argued that more accurate runtime estimates are not critical for scheduling. Numerous studies have shown that inaccuracy in this regard is not detrimental to performance, but on the contrary, may actually be beneficial [38, 47, 66]. This has motivated suggestions that doubling user runtime estimates [38, 66] or applying other randomization [42] may actually increase throughput. If this is true, then why fuss about accuracy?

The answer is that performance is not the only desirable quality of a schedule. Indeed, inaccurate runtime estimates have profound effects on fairness and predictability.

To see this, we must first understand why wildly exaggerated runtime estimates result in greater system throughput. This happens because the premature termination of jobs causes fragmentation in the schedule. In backfilling, that fragmentation is mitigated by executing shorter jobs from the back of the queue. It is no surprise that this arrangement increases performance as scheduling theory has long recognized that the Shortest Job First (SJF) heuristic results in optimal throughput [29]. SJF is not widely used on today's production installations because of its inadequate fairness. Coaxing an SJF algorithm out of a backfilling scheduler through inflated runtime estimates may therefore not be desirable, though his analysis has gone largely unrecognized [64, 38, 47, 66, 42].

Erratic runtime estimates can also manifest unfairness through a phenomenon known as *pseudo-delay* [26]. An example is shown in Figure 3. Job J_1 is prevented from executing at the same time as J_0. Relying on false runtime estimates, the scheduler decides to backfill J_2. Soon thereafter, J_0 completes execution, but J_1 cannot begin because of the decision to backfill J_2. The backfilling fairness guarantee that J_1 would not be delayed by any job behind it in the queue is broken.

Like fairness, system predictability also suffers because of poor runtime estimates. For queued space-sharing, this unpredictability is manifested as inconsistent queue wait times which are often quite significant with respect to overall flow times. As described in Section 3.3, predictability has a tangible impact on productivity. Statistically meaningful queue time predictors have therefore become a topic of research interest.

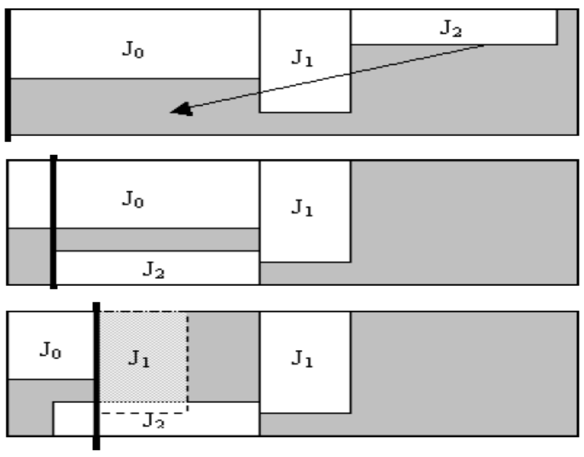

Figure 3. Example of pseudo-delay

4.1.3 Predicting Queue Times

Queue time predictability can be correlated with productivity through many scenarios. The most obvious example is a user with accounts on numerous machines who wants his job to finish most quickly. More subtle is the lost utility caused when the user's job does not finishing in time to provide useful output (e.g. a one day weather simulation finishing in twenty five hours) or the schedule perturbation caused when the user submits to multiple sites in order to guarantee the earliest possible start time.

Increasingly, distributed grid applications must also schedule their components across a set of machines which may exhibit disparate queue times for each type of subtask. Meaningful queue time predictions are an important component of effective grid workflow scheduling [51, 35].

A third example arises when jobs are moldable. Such jobs must decide whether to begin execution when some set of processors becomes available or to wait for a larger set to come free. Such a decision must compare the additional wait time versus the speedup provided by extra processors. Predictability of the queue wait times therefore has a direct impact on the response time of such an application.

Much research has sought to derive statistically meaningful queue time predictions. In 1997, Downey observed that job runtimes on the San Diego Supercomputer Center's Paragon system trended towards a uniform log distribution [11]. Using this observation, he showed that reasonable start time estimates could be calculated for the job at the head of a FCFS queue by calculating the probability of enough processors coming available before a certain time. This technique was also applied to processor allocation for moldable jobs and shown to increase average response times and system throughput [12].

Both Gibbons [24] and Smith [47] show that more accurate job runtime estimates can be derived from historical job logs. They derive these estimates for each job in the system and predict the queue time for any job by simulating the schedule. Smith's results indicate that the technique can predict queue times to within 30-60% on average for four different workloads and three different scheduling strategies. The greater difficulty with this approach is that it requires detailed knowledge of the often capriciously parameterized scheduling policy of the target machine. This is burdensome for a single supercomputing installation and impractical for a distributed grid infrastructure.

Recently, Brevik and Wolski have proposed a more generalized approach called the *Binomial Method Batch Predictor* (BMBP) [6]. The BMBP derives a maximum bound for a job's waiting time by using only historically observed queue wait times. The approach thus obviates the need for specific knowledge of a machine's scheduling policy. Whether or not these bounds are tight enough to be useful remains to be evaluated.

4.2 Time-Sharing

Aside from space-sharing, the other major approach to parallel scheduling is time-sharing. The term time-sharing, or time-slicing, refers to the sharing of a processor's time among threads of different parallel programs. In such approaches, each processor executes a thread for some time, pauses it, and begins executing a new thread. Applications therefore exhibit short wait times but execute more slowly than under the dedicated set of processors provided by space-sharing.

Today, the context switching overheads and complex job resource requirements of time-sharing, particularly memory management, have led most supercomputing installations to prefer the simplicity and predictability of space-sharing for executing performance-critical applications. Time-sharing is most often used to execute interactive jobs that do not necessarily require peak performance.

Though time-sharing has been largely marginalized for high performance scientific computing, its applicability to emerging scenarios remains promising. Interactivity is a critical requirement if parallel processing is to move beyond scientific supercomputing and into widespread deployment. Parallel processing on the desktop for example, is inextricably interactive. The same is true of web application infrastructures and consequently, many dynamic grid computing scenarios.

4.2.1 Local Scheduling

The simplest way to implement a parallel time-sharing scheduler is to run a uni-processor system on each node and share a global run queue. Threads that are ready to execute are placed in the queue. When a processor becomes available, it simply removes the next thread from the queue, executes it for some time, and returns it to the back of the queue. This approach was once widely used in small-scale uniform memory access machines [4, 55].

An obvious advantage of this approach is fairness. Each thread receives an equal share of the machine and priority mechanisms are straightforward to enable.

Local scheduling, however, is beset by numerous shortcomings. Contention for the global queue is a potential performance bottleneck, frequent context switching disturbs cache locality, and thread migration can be costly across processors, particularly when large chunks of data need be ported from one memory bank to another. *Affinity scheduling*, which avoids thread migration by scheduling each thread on the same processor during each time slice, has been proposed to counteract some of these shortcomings to a limited extent [37].

Affinity scheduling however, does not address local scheduling's most significant shortfall: the uncoordinated execution of an application's threads. One thread of an application may block for much of its time slice while waiting for communication from another, currently inactive thread of the application [19]. Even when the operating system is able to recognize this situation and preempt the blocking thread, excessive context-switching, known as *processor thrashing*, may result in significant slowdown [41]. Consequently, codes with fine-grained communication between threads are unlikely to perform well under local scheduling.

The inefficiency of uncoordinated thread execution must be addressed by a more application-centric approach such as *gang scheduling*.

4.2.2 Gang Scheduling

The most accepted form of time-sharing is gang scheduling, an approach by which all threads of an application are executed concurrently as one *gang* [41]. This approach is regarded as a confluence of space-sharing and time-sharing techniques and has been shown to outperform local scheduling in numerous studies [19, 41, 14, 17].

Under gang scheduling, applications can perform more fine-grained communications without suffering a significant performance penalty. Further, since gang scheduling assigns threads to processors, the approach enjoys all the benefits of affinity scheduling, including some local cache efficiencies and an obviated need for memory porting.

Gang scheduling, however, is limited in other respects. In addition to inheriting the drawbacks of memory management and context switching overheads from its time-sharing heritage, it also inherits many of the fragmentation issues of space-sharing [18].

Several variations and relaxations of gang scheduling have been proposed to overcome this challenge. One approach is *dynamic coscheduling* [50]. First proposed for commodity clusters, DCS observes that only threads that communicate often need be scheduled together and attempts to reduce fragmentation by scheduling each thread when a message arrives for it. DCS is part of a class of communication-driven approaches for clusters including spin blocking [40], periodic boost [40], and coordinated coscheduling [3].

Also, job moldability can be leveraged to reduce fragmentation in gang scheduling [9] just as is done under space-sharing [58].

Another inefficiency of gang scheduling, at least with respect to other time-sharing approaches, is its handling of I/O-intensive jobs [33, 43, 65]. Such jobs cause degradation in both processor and I/O efficiency because gang scheduling fails to overlap I/O requests with computation. Processor efficiency suffers when threads idly await I/O results, neither making progress nor allowing compute intensive threads to execute. I/O efficiency degrades when an I/O request returns, but the subsequent request cannot be issued because a compute-intensive job occupies the processor.

To improve I/O performance, certain relaxations in gang scheduling protocols have been proposed. *Flexible gang scheduling* [33] achieves optimal response times by customizing the length of each application's time-slices according to observed behavior. Alternatively, *paired gang scheduling* [62] coschedules I/O and compute-bound codes on the same processors and allows a local scheduler at each processor to choose which thread to run.

5 Improving Supercomputer Scheduling

Although MPP job scheduling is a rather mature field, there are still many improvements that could be addressed by the research community.

In backfilling systems, it seems the responsibility of estimating job runtimes is misplaced. Though users might specify the maximum amount of time for which they are willing to "finance" a specific job, relieving them of length estimation duties may be desirable for several reasons.

Most immediate is usability. Job runtime estimation creates a burden on the user that requires sedulous and error-prone submission script maintenance. Further, providing such estimates requires a level of expertise regarding both the application and hardware that most users evidently do not possess. As the number of supercomputing installations grows and users acquire accounts on multiple machines, such expertise will become increasingly scarce.

Despite their trouble, it is clear that users are either unable or unwilling to provide accurate runtime estimates. The inaccuracy exits both because many users do not posses the required expertise and because the current conditions do not motivate them to acquire it. While the scheduler is interested in the tightest bounds possible, users are only motivated to supply estimates that yield wait times with similar levels of user satisfaction. Perhaps this explains why a recent study of accounting logs at six supercomputing installations over several years has found that twenty values account for 90% of runtime estimates [57].

There are many examples. If a user submits a three hour job at 7:00 PM, it likely makes little difference to him if the job starts within one or five hours. The possibility of an earlier start time would be unlikely to motivate the user to even edit the submit script.

Suppose further that some user wants to perform a parameter sweep of an application by submitting twenty jobs. He writes a program that will automatically generate and submit twenty different run scripts. If he knows that each job will begin execution within one hour, he has little motivation to study the scaling of the application carefully and invest the additional programming burden of creating customized runtime estimates in each script.

This example highlights an even larger issue, that is, that runtime estimates require a "man in the loop". The participation of expert users is not a luxury that schedulers will always enjoy. Distributed grid applications for example, will need to dynamically submit jobs to various resources with no human intervention. Producing accurate runtime estimates under such assumptions would be very difficult.

The answer is not to artificially increase the stakes for users, but rather to place the burden of job length estimation on the current stakeholder: the scheduler. System administered prediction mechanisms have repeatedly been shown to be rather accurate [11, 24, 47, 38] and their deployment should not be precluded by backfilling's reservation guarantees. Relaxing this termination policy is a viable alternative which has been shown to significantly improve throughput over EASY scheduling when more accurate runtimes are introduced [56].

If users are nevertheless required to describe their job submissions, it would be more productive to submit profiling data which would allow the system to automatically predict the job's runtime using existing performance modelling techniques [36, 48, 49]. Such techniques have been shown to predict runtimes to accuracies within 90%.

Profiling data can be leveraged not only to predict runtimes, but also to help the scheduler perform effective processor allocation. Paired gang scheduling for example, proposes to match compute intensive and I/O intensive jobs on the same processors. Profiling data enables the scheduler to identify these categories.

This concept is highly applicable to space-sharing as well. Current implementations measure a job by the number of processors used and assign other resources such as memory and I/O accordingly. This assumption can lead to an under-utilization of system resources when jobs that require little memory make poor use of their memory allocations and jobs that require much memory spread across more, consequently lightly loaded, processors. If profiling data is provided and job categories are known, techniques such as *symbiotic space-sharing* [60, 59] can be leveraged to boost throughput by an estimated 20%.

It may even be possible to avoid burdening the user with the task of acquiring and supplying this information. In some cases, the system could acquire the data automatically by using partial executions (provided the application's effects are idempotent). As suggested by Perkovic [42], an ideal opportunity for partial executions exists in schedule fragmentation. If no job can be backfilled, idle processors should be used to perform partial executions of queued jobs.

Aside from collecting profiling data, partial executions can be used to provide other benefits. For instance, recent work in performance modelling has shown that the runtimes of highly iterative codes such as those commonly found in the sciences, can be very accurately anticipated by using partial executions [63]. Such techniques yield runtime estimates whose accuracy is very competitive with other existing modelling techniques. Existing schedule fragmentation can be exploited to produce these estimates.

Partial executions also have the potential to eliminate a perennial cause of schedule fragmentation and unpredictability: job crashes. When jobs crash upon starting, as many do because of misconfiguration or programming error, fragmentation and pseudo-delay may result. Eliminating such instances helps to create more efficient, fair, and predictable schedules.

Jobs that can checkpoint may also benefit from partial executions. Fragmentation holes may be used to make headway on certain jobs while they wait in the queue, thereby increasing throughput and system utilization levels.

6 Trends in Parallel Processing

At first glance, scheduling research may appear to be quite stable and mature. There are however, several trends in parallel processing that are reinvigorating the discipline and motivating novel scenarios and studies.

6.1 Parallelism in the Mainstream

Despite reports to the contrary, Moore's law is continuing unabated, doubling transistor densities every eighteen months. It is the accompanying doubling in processor speeds, and particularly in computing speeds, that is hobbled. Chip makers have long used increased densities to make faster serial chips and hid memory latencies with cache hierarchies and clever compilation. Recently, however, these techniques have been unable to keep pace.

In response, system architects are increasingly turning their attention to HPC technologies such as chip multiprocessors where a typical serial processor is replicated several times on a single chip. These designs can be exploited both by instruction level parallelism [53] and by parallel applications as discussed in this work.

Consequently, parallel processing is quickly entering the mainstream. The implication for job scheduling is an expansion in the types of users, scenarios, architectures, and applications that must be supported. For example, industry is relying on these technologies to support heavily loaded web services, application servers, and online transaction processing systems. These systems have users, interfaces, and requirements that are well outside of traditional HPC contexts. The necessity for scheduling policies that can optimize system performance under these conditions and in the face of complex and fluid business priorities, has already given rise to viable businesses.

Mainstream deployment of parallel applications may also drive innovation in programming models. The explicit parallelism of MPI is cumbersome for general purpose software development. Even in the respectively tolerant academic and scientific communities, more dynamic initiatives are taking hold. OpenMP [2], for example, a shared-memory programming interface based on a fork-join model is making inroads deep enough to marginalize purely distributed memory architectures.

OpenMP is not alone. The Department of Defense's High Productivity Computer Systems program [1] constitutes an immense national effort to create the next generation of HPC tools and architectures. All three industry partners (Sun, IBM, and Cray) are developing new programming languages to accompany their architecture proposals. Independently, Microsoft is also developing a new version of C that eases the multi-threaded programming burden through implicit parallelism [52].

The implication of this trend for scheduling is that jobs may no longer be strictly rigid. Work that has assumed malleable jobs [10, 9] could be revisited and completely new scenarios considered.

6.2 Grid Computing

A great source of new requirements for supercomputer scheduling is *grid computing*. Grid computing is the idea that a single community of users can gain access to multiple heterogeneous, physically distributed, and independently administered machines through a common interface.

The purpose of grids is not necessarily to build a single, immensely powerful supercomputer, but rather to increase the utility of the machines involved through better load balancing and by exploiting application affinities. At any given time, more lightly loaded machines can relieve pressure on more heavily loaded ones. Additionally, applications can increase performance by executing on more affine systems. Since increased resource utilization is the central goal, job scheduling is the central problem.

Grid computing does not only create a new field of scheduling (grid scheduling), but also directly impacts the requirements of local schedulers. Because Grid computing systems must respect site autonomy, grid schedulers must be built to interface with each machine's local scheduler. In many respects today's local schedulers inadequately support effective grid scheduling.

The most obvious obstacle is predictability. In order to decide which site is best for a particular job, a grid scheduler would have to determine the full response time of the job for each site. The unpredictability of queue wait times makes this very difficult. What was simply an inconvenience for local scheduling becomes critical for grid scheduling. These circumstances could conceivably cause some shift in priorities for local scheduling algorithms.

Even if queue times were more predictable, local schedulers would still need to provide updated capabilities such as reservations. Grid applications are often represented as *Directed Acyclic Graphs* (DAGs) of tasks, each of which must be scheduled at the most appropriate site. If task B is data dependent on task A, and each is to be executed on a different machine, reservations are likely more efficient than is serially waiting in each machine's queue.

Even more important is the case when tasks need to be *coallocated*, that is, scheduled to run at the same time but on different machines. This is a common requirement in workflow based grid applications and is impossible to guarantee without support from the local scheduler. In response, some have proposed plan-based scheduling schemes in place of queueing approaches [25].

Another feature of grid computing that cannot exist without local scheduler support is service level agreements. Such agreements can be rather arbitrary, for example, "up to 5 requests by user X between Tuesday and Thursday for a maximum of 16 CPU's and 12 hours must be fulfilled within 5 hours". Local scheduling techniques would have to be developed for supporting such high level objectives. If such agreements are dynamically reached, local schedulers would necessarily play an integral role in negotiations.

Lastly, if local schedulers still require runtime estimates for each job, generating them automatically is unavoidable. When a user submits a job to a grid scheduler, he cannot be expected to even know which machines the job *may* run on, let alone provide a reasonably tight runtime estimate for every possibility. The estimate must be generated dynamically by the grid and local schedulers. All of the approaches to automating runtime predictions discussed in Section 5 are applicable; it is therefore reasonable to expect such technologies to mature and take hold.

7 Conclusions

Job scheduling on parallel machines is a well studied research field that has led to widespread de facto standards: queued space-sharing with backfilling. This approach works well but can be improved through many techniques including automated runtime estimates, partial executions, and more intelligent processor allocation schemes.

While single site MPP scheduling is settling down, overall trends in parallel processing are ensuring that scheduling researchers will not bore. Parallel processing is expanding onto emerging architectures that are deployed in new scenarios and in support of disparate users and objectives.

Grid computing has created an entirely new field of scheduling research aimed at the efficient distribution of jobs across heterogeneous and independently administered machines. Concurrently, it is pressuring local scheduling research to provide expanded interfaces and reevaluate scheduling objectives.

References

[1] http://www.highproductivity.org/.

[2] OpenMP Architecture Review Board. OpenMP Specifications. http://www.openmp.org.

[3] S. Agarwal, G. S. Choi, C. R. Das, A. B. Yoo, and S. Nagar. Co-Ordinated Coscheduling in Time-Sharing Clusters through a Generic Framework. In *IEEE International Conference on Cluster Computing (CLUSTER'03)*, page 84, 2003.

[4] J. M. Barton and N. Bitar. A Scalable Multi-Discipline, Multiple-Processor Scheduling Framework for IRIX. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 45–69. Springer-Verlag, 1995.

[5] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and Stretch Metrics for Scheduling Continuous Job Streams. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1998.

[6] J. Brevik, D. Nurmi, and R. Wolski. Predicting Bounds on Queuing Delay for Batch-scheduled Parallel Machines. In *Proceedings of ACM Principles and Practices of Parallel Programming (PPoPP)*, March 2006.

[7] S.-H. Chiang, A. C. Arpaci-Dusseau, and M. K. Vernon. The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance. In *JSSPP '02: Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 103–127, London, UK, 2002. Springer-Verlag.

[8] W. Cirne and F. Berman. Adaptive Selection of Partition Size for Supercomputer Requests. In *IPDPS '00/JSSPP '00: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 187–208, London, UK, 2000. Springer-Verlag.

[9] J. Corbalan, X. Martorell, and J. Labarta. Improving Gang Scheduling through job performance analysis and malleability. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 303–311, New York, NY, USA, 2001. ACM Press.

[10] J. Corbalan, X. Martorell, and J. Labarta. Performance-Driven Processor Allocation. *IEEE Trans. Parallel Distrib. Syst.*, 16(7):599–611, 2005.

[11] A. B. Downey. Predicting Queue Times on Space-Sharing Parallel Computers. In *11th Intl. Parallel Processing Symp.*, pages 209–218, 1997.

[12] A. B. Downey. Using Queue Time Predictions for Processor Allocation. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 35–57. Springer Verlag, 1997.

[13] A. B. Downey and D. G. Feitelson. The elusive goal of workload characterization. *SIGMETRICS Perform. Eval. Rev.*, 26(4):14–29, 1999.

[14] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 1996. ACM Press.

[15] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. On Advantages of Grid Computing for Parallel Job Scheduling. In *Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CC-GRID 2002.*, 2002.

[16] Feitelson and Nitzberg. Job Characteristics of a Production parallel scientific workload on the NASA Ames iPSC/860. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949, pages 337–360. Springer, 1995.

[17] D. G. Feitelson. Distributed Hierarchical Control for Parallel Processing. *Computer*, 23(5):65–77, 1990.

[18] D. G. Feitelson. Packing Schemes for Gang Scheduling. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 89–110, London, UK, 1996. Springer-Verlag.

[19] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel & Distributed Comping*, 16(4):306–318, December 1992.

[20] D. G. Feitelson, L. Rudolph, and et al. Parallel Job Scheduling - A Status Report. In *Lecture Notes in Computer Science*, volume 3277, pages 1–16. Springer-Verlag, 2004.

[21] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer Verlag, 1997.

[22] D. G. Feitelson and L. R. L. Scheduling. Parallel Job Scheduling: Issues and Approaches. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949, pages 1–18. Springer, 1995.

[23] M. J. Flynn. Some computer organisations and their effectiveness. In *IEEE Transactions on Computers*, volume C-21, pages 948–960, September 1972.

[24] R. Gibbons. A Historical Application Profiler for Use by Parallel Schedulers. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 58–77, London, UK, 1997. Springer-Verlag.

[25] M. Hovestadt, O. Kao, A. Keller, and A. Streit. Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In D. G. Feitelson and L. Rudolph, editor, *Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2003.

[26] D. Jackson, Q. Snell, and M. Clement. Core Algorithms of the Maui Scheduler. *Lecture Notes in Computer Science*, 2221:87, 2001.

[27] J. P. Jones and B. Nitzberg. Scheduling for Parallel Supercomputing: A Historical Perspective of Achievable Utilization. In *IPPS/SPDP '99/JSSPP '99: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 1–16, London, UK, 1999. Springer-Verlag.

[28] S. Kannan, P. Mayes, M. Roberts, D. Brelsford, and J. Skovira. *Workload Management with LoadLeveler*. IBM, November 2001.

[29] D. Karger, C. Stein, and J. Wein. *CRC Handbook of Computer Science*, chapter Scheduling Algorithms. 1997.

[30] P. Krueger, T. H. Lai, and V. A. Radiya. Job Scheduling is More Important than Processor Allocation for Hypercube Computers. *IEEE Transactions on Parallel and Distributed Systems*, 5:488 – 497, 1994.

[31] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[32] C. Lee, Y. Schwartzman, J. Hardy, and A. Snavely. Are user runtime estimates inherently inaccurate? In *In 10th Job Scheduling Strategies for Parallel*, June 2004.

[33] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for Gang Scheduled Workloads. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 215–237. Springer Verlag, 1997.

[34] D. A. Lifka. The ANL/IBM SP Scheduling System. In *IPPS 1995 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949, pages 295–303, 1995.

[35] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, B. Liu, L. Johnsson, and J. Mellor-Crummey. Scheduling Strategies for Mapping Application Workflows onto the Grid. In *14th IEEE Symposium on High Performance Distributed Computing (HPDC 2005)*. IEEE Computer Society Press, 2005.

[36] G. Marin and J. Mellor-Crummey. Crossarchitecture Performance Predictions for Scientific Applications Using Parameterized Models. In *SIGMETRICS 2004 /PERFORMANCE 2004: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 2–13, New York, NY, 2004. ACM Press.

[37] R. Motwani, S. Phillips, and E. Torng. Nonclairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.

[38] A. Mu'alem and D. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. In *12th Intl. Parallel Processing Symposium*, pages 542–546, April 1998.

[39] R. R. Muntz and J. E. G. Coffman. Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems. *J. ACM*, 17(2):324–338, 1970.

[40] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. Alternatives to coscheduling a network of workstations. *Journal of Parallel Distributed Computing*, 59(2):302–327, 1999.

[41] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[42] D. Perkovic and P. J. Keleher. Randomization, Speculation, and Adaptation in Batch Schedulers. In *Supercomputing 2000*, pages 48–48, 2000.

[43] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante. The impact of I/O on program behavior and parallel scheduling. volume 26, pages 56–65, New York, NY, USA, 1998. ACM Press.

[44] U. Schwiegelshohn and R. Yahyapour. Analysis of first-come-first-serve parallel job scheduling. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 629–638, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[45] U. Schwiegelshohn and R. Yahyapour. Improving First-Come-First-Serve Job Scheduling by Gang Scheduling. In *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 180–198, London, UK, 1998. Springer-Verlag.

[46] U. Schwiegelshohn and R. Yahyapour. Fairness in Parallel Job Scheduling. *Journal of Scheduling*, 3(5):297–320, 2000.

[47] W. Smith, V. Taylor, and I. Foster. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 202–219. Springer Verlag, 1999.

[48] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A Framework for Application Performance Modeling and Prediction. In *Supercomputing 02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–17, Los Alamitos, CA, 2002. IEEE Computer Society Press.

[49] A. Snavely, N. Wolter, and L. Carrington. Modeling Application Performance by Convolving Machine Signatures with Application Profiles. In *Proceedings of IEEE 4th Annual Workshop on Workload Characterization*, pages 128–137, December 2001.

[50] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic Coscheduling on Workstation Clusters. *Lecture Notes in Computer Science*, 1459:231–256, 1998.

[51] D. P. Spooner, J. Cao, S. A. Jarvis, L. He, and G. R. Nudd. Performance-Aware Workflow Management for Grid Computing. *Comput. J.*, 48(3):347–357, 2005.

[52] H. Sutter and J. Larus. Software and the Concurrency Revolution. *ACM Queue*, 3(7):5462, September 2005.

[53] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *The 36th Annual International Symposium on Microarchitecture (MICRO-36)*, December 2003.

[54] D. Talby and D. Feitelson. Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling, 1999.

[55] S. Thakkar, P. Gifford, and G. Fielland. Balance: : a shared memory multiprocessor system. In *Proceedings of the International Conference on Supercomputing 2 (ICS)*, pages 93–101, 1987.

[56] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling Using Runtime Predictions Rather Than User Estimates. Technical Report 2005-5, Hebrew University, Jerusalem, Israel, February 2005.

[57] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling user runtime estimates. *Job Scheduling Strategies for Parallel Processing*, 3834:1–35, 2005.

[58] G. Utrera, J. Corbal, and J. Labarta. Using moldability to improve the performance of supercomputer jobs Source. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, October 2002.

[59] J. Weinberg and A. Snavely. User-Guided Symbiotic Space-Sharing of Real Workloads. In *The 20th ACM International Conference on Supercomputing (ICS '06)*, June 2006.

[60] J. Weinberg and A. Snavely. When Jobs Play Nice: The Case For Symbiotic Space-Sharing. In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC-15 '06)*, Paris, France, June 2006.

[61] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to higher moments of conditional response time. In *SIGMETRICS ’05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 229–240, New York, NY, USA, 2005. ACM Press.

[62] Y. Wiseman and D. Feitelson. Paired Gang Scheduling. In *IEEE Transactions on Parallel and Distributed Systems*, volume 14, pages 581–592, 2003.

[63] L. T. Yang, X. Ma, and F. Mueller. Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution. In *SC ’05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 40, Washington, DC, USA, 2005. IEEE Computer Society.

[64] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling, and Migration. In *JSSPP ’01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 133–158, London, UK, 2001. Springer-Verlag.

[65] Y. Zhang, A. A. Yang, A. Sivasubramaniam, and J. Moreira. Gang Scheduling Extensions for I/O Intensive Workloads. In *Lecture Notes in Computer Science*, volume 2862, pages 183–207. october 2003.

[66] D. Zotkin and P. J. Keleher. Job-Length Estimation and Performance in Backfilling Schedulers. In *IEEE International Symposium on High Performance Distributed Computing*, 1999.