

Parallel Job Scheduling — A Status Report

Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Larry Rudolph
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

Uwe Schwiegelshohn
Computer Engineering Institute
Universität Dortmund
44221 Dortmund, Germany

1 Introduction

Scheduling parallel jobs has been a popular research topic for many years. A couple of surveys have been written on this topic in the context of parallel supercomputers [17, 20]. The purpose of the present paper is to update this material, and to extend it to include work concerning clusters and the grid.

The first part of the paper deals with algorithmic and research issues. It covers the two main approaches in use today, backfilling and gang scheduling. For each one, recent advances are reviewed, both in terms of how to perform the scheduling and in terms of understanding the performance results.

The second part of the paper covers current usage. It presents a short overview of vendor offerings, and then reviews the scheduling frameworks used by top-ranking parallel systems.

2 Advances in Parallel Job Scheduling Research

There are many different ways to schedule parallel jobs, and the threads which make them up [17]. But only a few mechanisms are used in practice and studied in detail. Two approaches that have dominated the last decade are backfilling and gang scheduling. In this section we review their variants and the connections between them. We then review the special requirements of scheduling parallel jobs on a grid, and algorithms that have been developed to address them.

2.1 Backfilling

The basic batch scheduling algorithm is First-Come-First-Serve (FCFS) [39]. Under this algorithm, jobs are considered in order of arrival. If there are enough processors available to run a job, the processors are allocated and the job is started. But if enough processors are not available, the first job must wait for some currently running job to terminate and free additional processors. All subsequent jobs also wait so as not to violate the FCFS order. This may lead to a waste of processing power as processors sit idle waiting for enough of them to accumulate.

Backfilling is an optimization that tries to balance between the goals of utilization and maintaining FCFS order. It allows small jobs to move ahead and run on processors that would otherwise remain idle. This is done subject to some restrictions, so as to avoid situations in which the FCFS order is completely violated and some jobs are never run (a phenomenon known as “starvation”). In particular, jobs that need to wait are typically given a reservation for some future time.

The use of reservations was included in several early batch schedulers [26, 8]. Backfilling, in which small jobs move forward to utilize the idle resources, was introduced by Lifka [29]. This was done in the context of EASY, the Extensible Argonne Scheduling sYstem, which was developed for the first large IBM SP1 installation at Argonne National Lab.

2.1.1 Improved Algorithms

While the concept of backfilling is quite simple, it nevertheless has several variants with subtle differences. These can be described by the settings of different parameters of the algorithm.

The first parameter is the **number of reservations** that are made. In the original EASY backfilling algorithm,

only the first queued job received a reservation. This means that when the first job cannot run because sufficient processors are not available, we estimate when such processors *will* be available, and reserve them for this job. Other jobs that are backfilled may not violate this reservation. Thus they must either terminate before the time of the reservation (known as the “shadow time”), or use only processors that are not required by the first job [29].

The problem with this is that backfilling may cause delays in the execution of other waiting jobs (which are not the first, and therefore do not get a reservation). The obvious alternative is to make reservations for all jobs. This approach has been named “conservative backfilling” [33]. However, simulation results indicate that delaying other jobs is rarely a problem, and that conservative backfilling tends to achieve reduced performance in comparison with the more aggressive EASY backfilling. The MAUI scheduler includes a tunable parameter that allows system administrators to decide how many reservations will be made [27]. Chiang et al. suggest that making 2–4 reservations is a good compromise [6].

An intriguing idea in this context is to make reservations adaptively as needed. This is realized by noting how much different jobs have been delayed by previous backfilling decisions. If a job is delayed by too much, a reservation is made for this job [45]. This is essentially equivalent to the earlier “flexible backfilling”, in which all jobs have reservations, but backfilling is allowed to violate these reservations up to a certain slack [46]. Setting the slack to the threshold used by adaptive reservations is equivalent to only making a reservation if the delay exceeds this threshold.

The second parameter of the backfilling algorithm is the **order of queued jobs**. The original EASY scheduler, and many other systems and designs, use a FCFS order [29]. A general alternative is to prioritize jobs in some way, and select jobs for scheduling (including as candidates for backfilling) according to this priority order. Flexible backfilling combines three types of priorities: an administrative priority set to favor certain users or projects, a user priority used to differentiate among the jobs of the same user, and a scheduler priority used to guarantee that no job is starved [46]. The Maui scheduler has a priority function that includes even more components [27].

A special type of prioritization depends on job characteristics. In particular, Chiang et al. have proposed a whole set of criteria based on resource consumption, that are generalizations of the well-known Shortest Job First (SJF) scheduling algorithm [6]. These have been shown to improve performance metrics, especially those that are particularly sensitive to the performance of short jobs, such as slowdown.

A third parameter is the amount of **lookahead into the**

queue. All previous backfilling algorithms consider the queued jobs one at a time, and try to schedule them. But the order in which jobs are scheduled may lead to loss of resources to fragmentation. The alternative is to consider the whole queue at once, and try to find the set of jobs that together maximize desired performance metrics. This can be done using dynamic programming, leading to optimal packing and improved performance [42].

2.1.2 Effect of User Runtime Estimates

Backfilling depends on estimates of how long each job will run. These estimates are used to figure out when additional processors will become available, and to verify that backfilled jobs will terminate in time so as not to violate reservations. The source of the estimates is typically the user who runs the job. Jobs that try to run beyond their estimated runtime are usually terminated by the system. Many therefore regard these estimates as upper bounds, rather than as tight estimates.

Initial expectations were that user runtime estimates will nevertheless be tight, as low estimates improve the chance for backfilling. However, comparisons of user estimates with real runtimes show that they tend to be inaccurate, even when users are requested to provide their best possible estimate with no danger of having their job killed if the estimate is too low [18, 33, 28]. Attempts to derive better estimates automatically based on historical information from previous runs have not been successful, as they suffered from too many under-estimations (which in backfilling would lead to killed jobs).

Surprisingly, several studies have demonstrated that inaccurate runtime estimates actually lead to improved average performance [18, 33, 55]. This is not simply the result of more backfilling due to more holes in the schedule, because inflated runtime estimates not only create holes in the schedule, but also enlarge potential backfill jobs, making it harder for them to fit into these holes. Rather, it is the result of a sequence of events where small backfill jobs prevent the holes from closing up, leading to a strong preference for short jobs and the automatic production of an SJF-like schedule [47]. This also motivates the construction of algorithms that explicitly favor short jobs such as those proposed in [6].

All this does not necessarily indicate that more accurate runtime estimates are impossible and useless. First, not all estimates are bad; In most cases, some users provide reasonably accurate estimates while others do not. Some studies indicate that those users who do provide reliable estimates do indeed benefit from this, as their jobs receive better service from the scheduler [6]. Also, while it seems that deriving good estimates automatically is not possible for all jobs, it might be possible to do so for short jobs and

for jobs that have exhibited especially small variability in the past.

Incidentally, inaccurate user runtime estimates have also been shown to have surprising effects on performance evaluations [16, 15]. In a nutshell, it was seen that for workloads with numerous long single-process jobs the inaccurate estimates allow for significant backfilling of these jobs under the aggressive EASY backfilling, but not under conservative backfilling. This in turn was detrimental for the performance of short jobs that were delayed by the long backfilled jobs. But if accurate estimates were used the effect was reversed, leading to a situation where short jobs were favored over long ones. This has more to do with evaluation methodology than will scheduling technology.

2.2 Gang Scheduling

The main alternative to batch scheduling is gang scheduling, where jobs are preempted and re-scheduled as a unit, across all involved processors. The notion was introduced by Ousterhout, using the analogy of a working set of memory pages to argue that a “working set” of processes should be co-scheduled for the application to make efficient progress [34]. Subsequent work emphasized gang scheduling, which is an all-or-nothing affair, i.e. either all of the job’s processes run or none do.

The point of gang scheduling is that it provides an environment similar to a dedicated machine, in which all a job’s threads progress together, and at the same time allows resources to be shared. In particular, preemption is used to improve performance in face of unknown runtimes. This prevents short jobs from being stuck in the queue waiting for long ones, and improves fairness [40].

2.2.1 Flexible Algorithms

One problem with gang scheduling is that the requirement that all a job’s processes always run together causes too much fragmentation. This has led to several proposals for more flexible variants.

One such variant, called “paired gang scheduling” is designed to alleviate inefficiencies caused by I/O activity [50]. In conventional gang scheduling, processor running processes that perform I/O remain idle for the duration of the I/O operation. In paired gang scheduling jobs with complementary characteristics are paired together, so that when the processes of one perform I/O those of the other can compute. Given a good job mix, this can lead to improved resource utilization at little penalty to individual jobs.

A more general approach is to monitor the communication behavior of all applications, and try to determine whether they really benefit for gang scheduling [22].

Gang scheduling is then used for those that need it. Processes belonging to other jobs are used as filler to reduce the fragmentation cause by the gang scheduled jobs.

2.2.2 Dealing with Memory Pressure

Early evaluations of gang scheduling assumed that all arriving jobs can be started immediately. Under high loads this could lead to situations where dozens of jobs share each processor. This is unrealistic as all these jobs would need to be memory resident or else suffer from paging, which would interfere with the synchronization among the job’s threads.

A simple approach for avoiding this problem is to use admission controls, and only allow additional jobs to start if enough memory is available [3]. An alternative is placing an oblivious cap on the multiprogramming level (MPL), usually in the range of 3–5 jobs [31]. While this avoids the need to estimate how much memory a new job will need, it is more vulnerable to situations in which memory becomes overcommitted and paging may occur.

When admission controls are used and jobs wait in the queue the question of queue order presents itself. The simplest option is obviously to use a FCFS order. Improved performance is obtained by using backfilling, and allowing small jobs to move ahead in the queue [53, 52]. In fact, using backfilling fully compensates for the loss of performance due to the limited number of jobs that are actually run concurrently [21].

All the above schemes may suffer from situations in which long jobs are allocated resources while short jobs remain in the queue and await their turn. The solution is to use a preemptive long-range scheduling scheme. With this construction, the long turn scheduler allocates memory to waiting jobs, and then the short turn scheduler decides which jobs will actually run out of those that are memory resident. The long turn scheduler may decide to swap out a job that has been in memory for a long time, to make room for a queued job that has been waiting for a long time. Such a scheme was designed for Tera (Cray) MTA machine [1].

2.2.3 System Integration

The only commercially successful implementation of gang scheduling so far was the one on the Connection Machine CM-5. Other implementations, e.g. on the Intel Paragon, suffered from significant overheads and were not generally used. But recently there have been several advances in the implementation of gang scheduling in experimental systems.

Gang scheduling requires the context switching to be synchronized across the nodes of the machine. This is hard to achieve on large machines, and may suffer from

significant overheads. But modern interconnection networks provide hardware support for global operations, and this can be exploited also in the runtime system. This is done in STORM, where all parallel system activities are expressed in terms of three basic primitives, which in turn are supported by the hardware of the Quadrics network. In particular, this design has resulted in a very scalable implementation of gang scheduling [23].

While high performance networks enable efficient implementation of system primitives, they may cause problems with multiprogramming. The difficulty arises due to the use of user-level communication, in which user processes access the network interface cards (NICs) directly so as to avoid the overheads involved in trapping into the operating system. As a result no protection is available, and only one job can use the NICs. This can be solved by switching communication buffers as part of the gang scheduling's context switch operation [14]. It is also possible that this problem will be reduced in the future, as the memory available on NICs continues to grow.

Even tighter integration between communication and scheduling is used in the "buffered coscheduling" scheme proposed by Petrini and Feng [35, 36]. In this scheme the execution of all jobs is partitioned by the system into phases. In each phase processes are only executed until they try to perform a communication operation, and then they are blocked and the communication is buffered. At the end of the phase all the required communications are scheduled so as to achieve optimal performance, and performed during the next phase. This leads to complete overlap of computation and communication.

Gang scheduling was originally developed in order to support fine-grain synchronization of parallel applications [19]. But an even greater benefit may be its contribution to reducing interference [32, 37]. The problem is that the nodes of parallel machines and clusters typically run a full operating system, with various user-level daemons that are required for various system services. These daemons may wake up at unpredictable times in order to perform their function. Obviously this interferes with the application process running on the node. If such interferences are not synchronized across nodes, the application will be slowed considerably as different processes are delayed. But with gang scheduling it is possible to run all the daemons on the different nodes at the same time, and eliminate their interference when user jobs are running. When this is done, the full capabilities of the hardware are achieved.

2.3 Parallel Job Scheduling and the Grid

More recently, parallel computers are becoming part of a so called computational grid. The name grid has been

chosen in analogy to the electrical power grid where several power plants provide numerous consumers with electrical power without that the consumer is aware of the origin of the power he draws from the net. Similarly, it is the goal of a computational grid or simply Grid to allow users to run their jobs on any suitable computer belonging to the Grid. This way the computational load is balanced across many machines. Clearly, the Grid is mainly of interest for large computational jobs or jobs using a large data set as smaller jobs will usually run locally. However, the Grid is not restricted to this kind of jobs but will cover a wide range of general services. Nevertheless at the moment large computational jobs form the dominant grid application.

Before addressing the scheduling problem in a grid it is necessary to point out some differences between a parallel computer and the grid. A parallel computer has a central resource management system that can control all individual processors. However in a grid, the compute resources typically have different owners and as in most distributed systems there is no central control. Therefore, a compute resource typically has its own local resource management system that implements the policy of its owner. Hence, a grid scheduling architecture must be built on top of those existing local resource management systems. This requires communication between those different layers of the scheduling system in a grid [41, 49]. As in a distributed system the use of a central grid scheduler may result in a performance bottleneck and lead to a failure of the whole system if the scheduler fails. It is therefore appropriate to use a decentralized grid scheduler architecture and distributed algorithms.

Further, grid resources are heterogeneous in hardware and software which imposes constraints on the suitability of a resource for a given job. In addition, not every user may be accepted on every machine due to the implemented owner policy. A grid scheduler must determine which resources can be used for a specific submitted job while such a problem is usually not encountered in a parallel processor or even in a cluster of computers [10, 12]. Moreover, the grid is subject to frequent changes as some compute resources may be temporarily withdrawn from the grid due to maintenance or privileged non-grid use on request of the owner. To obtain these data, the grid scheduler needs a specific grid information service while the necessary up-to-date information is always assumed to be available in a parallel computer.

Today, the main purpose of grid computing is considered to be in the area of cross-domain load balancing. To support this idea the Globus Toolbox provides basic services that allow the construction of a grid scheduler. With the help of those basic services grid schedulers are constructed that run on top of commercial resource manage-

ment systems, like LSF, PBS or Loadleveler. Further, existing Systems, like Condor [30, 38], are adapted to include grid scheduling abilities or allow integration with a grid scheduler.

If a parallel computer is embedded in a grid, a large variety of jobs from different users will be run on this machine. Then it will become increasingly difficult to implement the usage policy of an owner with the help of those simple scheduling criteria that are used today, like utilization and response time. Therefore, it can be assumed that the grid will also change job scheduling strategies for parallel computers. However in practise such an effect has not been observed yet.

Large grid application projects, like LCG, Datagrid, GriPhyn, frequently include the construction of some grid scheduler. Unfortunately, the scope of such a scheduler is usually restricted to the corresponding application project. On the other hand, there are academic projects that specifically address scheduling issues like the generation, distribution and selection of resource offerings. To this end various means are used, for instance economic methods.

In another approach, the job itself is responsible for its scheduling. Then we speak of an application scheduler. This is important for jobs which have a complex workflow and are subject to complex parallelization constraints. For example, this is the approach taken in the AppLes project [7].

As a continuation of some metacomputing ideas it is sometimes considered to use a computational grid as a single parallel processor, where many computational resources, that is parallel computers in the grid, are combined to solve a single very large problem. In this situation, the network performance varies greatly from communication within a parallel computer to communication between two parallel computers. Some models have been derived to evaluate the performance of so called multi site computing [24, 4, 9, 11, 13]. However in practice, such an approach has not been implemented with the possible exception of the preplanned combination of a few specific parallel computers for a specific purpose.

An important component of using the grid as a single parallel resource is co-allocation [5, 4, 2, 43]. This means that resources on several different machines need to be allocated to the same job at the same time. This is hard to accomplish due to the fact that the different resources belong to different owners, and do not have a common resource management infrastructure. The way to circumvent this problem is to try and reserve resources on the different machines, and then to use them only if all required reservations are successful [44].

3 Parallel Job Scheduling Practice

3.1 Vendor Offerings

Commercial scheduling software for parallel jobs comes in two types: portable, standalone systems, and components in a specific system.

There are two main competitors in the market for scheduling software. One is the Platform Computing Load Sharing Facility (LSF), which is based on the Utopia project [54]. The other is the Veridian Portable Batch System (PBS) [25]. Both provide similar functionality. In particular, they provide support for various administrative tasks, which is often lacking from research prototypes.

In addition, vendors of parallel supercomputers typically provide some sort of scheduling support with their systems. This includes schedulers on the IBM SP, the Cray Origin, and HP and Sun systems.

3.2 Actual Usage

In order to determine which job scheduling strategies for parallel processors are actually applied in practice we considered the 50 most powerful parallel computers based on actual Top500 list. Information about the strategies used in each case where mainly retrieved from publicly available information sources like the web. In addition many sites were contacted directly and asked to provide further information.

Those parallel computers can be classified into 3 groups:

Parallel Vector Processors Only 4 of the computers belong to this class which consist of NEC's Earth-Simulator, the leader of the Top500 list, and 3 installations of a Cray X1.

Parallel Processors Almost 40% of the considered computers are true parallel processors. Only 4 of them are not of the types IBM SP Power3 or IBM pSeries 690.

Clusters There is a larger variety of types for clusters although Xeon clusters clearly dominate with more than 50% of all cluster installation among the considered computer systems.

3.2.1 Parallel Vector Processors

The Cray X1 installations all use the same scheduling system consisting of *PBS Pro*, a load balancer and a gang scheduler. We were not able to obtain additional information.

Scheduling is different for the Earth Simulator which is currently the most powerful parallel processor according to the Top500 list. The system uses a queue for small

batch requests (S-queue) and a queue for large batch requests (L-queue) [48]. For the S-queue, ERS-II is used as a scheduling system. Although ERS-II supports gang scheduling this feature is not used for the S-queue. The L-queue has a customized scheduler which does not support gang scheduling. Further, the Earth Simulator scheduling systems support backfilling and checkpointing.

3.2.2 Parallel Processors

Most IBM systems use LoadLeveler which supports backfilling. Although LoadLeveler also allows job prioritization, this is not mentioned as a feature in the description of most installations. As most direct replies confirmed job prioritization, we may assume that it is actually used in most systems but not explicitly mentioned. At least the newer versions of LoadLeveler also support gang scheduling which is also not found in most descriptions. However, at least the Max-Planck-Society in Germany explicitly states that gang scheduling is possible but not used. This shows that at least some installations have decided against gang scheduling.

The Lawrence Livermore National Labs have developed a home grown resource management system called LCRM (Livermore Computing Resource Management System) that supports backfilling, reservation, preemption and gang scheduling. This system is used for the ASCI White installation and for cluster installations at Lawrence Livermore National Labs. The ASCI White system has batch partition and an interactive partition but uses only a single queue with 3 classes of jobs (*expedited, normal* and *stand-by*). However, it does not currently use the preemption feature. The utilization is between 80% and 90%.

Reservation is also used in the installation at ECMWF (European Centre for Medium-Range Weather Forecast). Here, LoadLeveler is enhanced by a special job filter. The system separates serial and parallel jobs by assigning them to different classes (2 classes for serial jobs and 3 classes for parallel jobs). The utilization of this system is between 94% and 97.5%. A similar utilization is achieved on the above mentioned parallel processor of the Max-Planck-Society with a more elaborate scheme of job queues.

We were not able to obtain much information on non-IBM parallel processors except that gang scheduling is supported by the ASCI Red system consisting of Intel Xeon processors and using the Paragon operating system.

3.2.3 Clusters

Various commercial resource management systems can be found in cluster installations, including various form of PBS [25] and LSF [54]. They are frequently combined with the Maui scheduler [27]. As already mentioned

Lawrence Livermore National Labs use LCRM also for their clusters. In many Linux clusters SLURM (Simple Linux Utility for Resource Management) is especially used for low priority jobs [51]. The Pittsburgh Supercomputing Center has developed a custom scheduler called Simon on top of OpenPBS in order to support a variety of advanced scheduling features like advance reservation, backfilling, and checkpointing.

In general, it can be stated that the scheduler of most cluster installations support backfilling and job prioritization. Gang scheduling, preemption, advance reservations and checkpointing are more frequently found than in parallel processor installations. In most installations, almost all computing nodes are in a single partition. There are few exceptions. For instance the Pacific Northwest National Lab has additional partitions for management and user log-in nodes (4 nodes) as well as for the Lustre file system nodes (34). However, these partitions are relatively small in comparison to the total number of nodes in the compute partition (940). The cluster at Los Alamos National Labs also has file serving nodes that allow interactive access via LSF.

Los Alamos National Lab also uses more queues (8-9 active queues and 4-5 special purpose queues) than other installations. In addition queues can be specifically set up for a project. In other clusters users can submit their jobs to at most 3 different queues.

The utilization of the systems depends on the applications and ranges from approximately 55% in 2003 (Los Alamos National Lab) to 95% for the last 30 days (Pittsburgh Supercomputing Center).

References

- [1] G. Alverson, S. Kahan, R. Korry, C. McCann, and B. Smith, “*Scheduling on the Tera MTA*”. In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 19–44, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.
- [2] S. Banen, A. I. D. Bucur, and D. H. J. Epema, “*A measurement-based simulation study of processor co-allocation in multicluster systems*”. In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 105–128, Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.
- [3] A. Batat and D. G. Feitelson, “*Gang scheduling with memory considerations*”. In *14th Intl. Parallel & Distributed Processing Symp.*, pp. 109–114, May 2000.

- [4] A. I. D. Bucur and D. H. J. Epema, "The influence of communication on the performance of co-allocation". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 66–86, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.
- [5] A. I. D. Bucur and D. H. J. Epema, "The influence of the structure and sizes of jobs on the performance of co-allocation". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 154–173, Springer Verlag, 2000. Lect. Notes Comput. Sci. vol. 1911.
- [6] S-H. Chiang, A. Arpaci-Dusseau, and M. K. Vernon, "The impact of more accurate requested runtimes on production job scheduling performance". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 103–127, Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.
- [7] W. Cirne and F. Berman, "Adaptive selection of partition size for supercomputer requests". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 187–207, Springer Verlag, 2000. Lect. Notes Comput. Sci. vol. 1911.
- [8] D. Das Sharma and D. K. Pradhan, "Job scheduling in mesh multicomputers". In *Intl. Conf. Parallel Processing*, vol. II, pp. 251–258, Aug 1994.
- [9] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, "Enhanced Algorithms for Multi-Site Scheduling". In *Proceedings of the 3rd International Workshop on Grid Computing, Baltimore*, Springer-Verlag, Lecture Notes in Computer Science LNCS, 2002.
- [10] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, "On Advantages of Grid Computing for Parallel Job Scheduling". In *Proc. 2nd IEEE/ACM Int'l Symp. on Cluster Computing and the Grid (CCGRID2002)*, IEEE Press, Berlin, May 2002.
- [11] C. Ernemann, V. Hamscher, A. Streit, and R. Yahyapour, "On Effects of Machine Configurations on Parallel Job Scheduling in Computational Grids". In *International Conference on Architecture of Computing Systems, ARCS*, pp. 169–179, VDE, Karlsruhe, April 2002.
- [12] C. Ernemann, V. Hamscher, and R. Yahyapour, "Economic Scheduling in Grid Computing". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 128–152, Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.
- [13] C. Ernemann and R. Yahyapour, "Grid Resource Management - State of the Art and Future Trends", chap. "Applying Economic Scheduling Methods to Grid Environments", pp. 491–506. Kluwer Academic Publishers, 2003.
- [14] Y. Etsion and D. G. Feitelson, "User-level communication in a system with gang scheduling". In *15th Intl. Parallel & Distributed Processing Symp.*, Apr 2001.
- [15] D. G. Feitelson, *Experimental Analysis of the Root Causes of Performance Evaluation Results: A Backfilling Case Study*. Technical Report 2002–4, School of Computer Science and Engineering, Hebrew University, Mar 2002.
- [16] D. G. Feitelson, "Metric and workload effects on computer systems evaluation". *Computer* **36(9)**, pp. 18–25, Sep 2003.
- [17] D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems*. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.
- [18] D. G. Feitelson and A. Mu'alem Weil, "Utilization and predictability in scheduling the IBM SP2 with backfilling". In *12th Intl. Parallel Processing Symp.*, pp. 542–546, Apr 1998.
- [19] 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.
- [20] D. G. Feitelson and L. Rudolph, "Parallel job scheduling: issues and approaches". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–18, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.
- [21] E. Frachtenberg, D. G. Feitelson, J. Fernandez, and F. Petrini, "Parallel job scheduling under dynamic workloads". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 208–227, Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.
- [22] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez, "Flexible coscheduling: mitigating

- load imbalance and improving utilization of heterogeneous resources". In *17th Intl. Parallel & Distributed Processing Symp.*, Apr 2003.
- [23] E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll, "STORM: lightning-fast resource management". In *Supercomputing*, Nov 2002.
- [24] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, "Evaluation of Job-Scheduling Strategies for Grid Computing". In *Proc. 7th Int'l Conf. on High Performance Computing, HiPC-2000*, pp. 191–202, Springer, Berlin, Lecture Notes in Computer Science LNCS 1971, Bangalore, Indien, 2000.
- [25] 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. Lect. Notes Comput. Sci. vol. 949.
- [26] Intel Corp., *iPSC/860 Multi-User Accounting, Control, and Scheduling Utilities Manual*. Order number 312261-002, May 1992.
- [27] D. Jackson, Q. Snell, and M. Clement, "Core algorithms of the Maui scheduler". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 87–102, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.
- [28] C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavey, "Are user runtime estimates inherently inaccurate?". In *10th Job Scheduling Strategies for Parallel Processing*, Jun 2004.
- [29] 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. Lect. Notes Comput. Sci. vol. 949.
- [30] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - a hunter of idle workstations". In *8th Intl. Conf. Distributed Comput. Syst.*, pp. 104–111, Jun 1988.
- [31] J. E. Moreira, W. Chan, L. L. Fong, H. Franke, and M. A. Jette, "An infrastructure for efficient parallel job execution in terascale computing environments". In *Supercomputing '98*, Nov 1998.
- [32] R. Mraz, "Reducing the variance of point-to-point transfers for parallel real-time programs". *IEEE Parallel & Distributed Technology* **2(4)**, pp. 20–31, Winter 1994.
- [33] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling". *IEEE Trans. Parallel & Distributed Syst.* **12(6)**, pp. 529–543, Jun 2001.
- [34] J. K. Ousterhout, "Scheduling techniques for concurrent systems". In *3rd Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.
- [35] F. Petrini and W-c. Feng, "Buffered coscheduling: a new methodology for multitasking parallel jobs on distributed systems". In *14th Intl. Parallel & Distributed Processing Symp.*, pp. 439–444, May 2000.
- [36] F. Petrini and W-c. Feng, "Time-sharing parallel jobs in the presence of multiple resource requirements". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 113–136, Springer Verlag, 2000. Lect. Notes Comput. Sci. vol. 1911.
- [37] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q". In *Supercomputing*, Nov 2003.
- [38] 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. Lect. Notes Comput. Sci. vol. 949.
- [39] U. Schwiegelshohn and R. Yahyapour, "Analysis of First-Come-First-Serve Parallel Job Scheduling". In *Proceedings of the 9th SIAM Symposium on Discrete Algorithms*, pp. 629–638, January 1998.
- [40] U. Schwiegelshohn and R. Yahyapour, "Fairness in Parallel Job Scheduling". *Journal of Scheduling*, **3(5):297-320**. John Wiley, 2000.
- [41] U. Schwiegelshohn and R. Yahyapour, "Grid Resource Management - State of the Art and Future Trends", chap. "Attributes for Communication Between Grid Scheduling Instances", pp. 41–52. Kluwer Academic Publishers, 2003.
- [42] E. Shmueli and D. G. Feitelson, "Backfilling with lookahead to optimize the performance of parallel job scheduling". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 228–251, Springer-Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.

- [43] J. M. P. Sinaga, H. H. Mohammed, and D. H. J. Epema, "A dynamic co-allocation service in multicluster systems". In *10th Job Scheduling Strategies for Parallel Processing*, Jun 2004.
- [44] Q. Snell, M. Clement, D. Jackson, and C. Gregory, "The performance impact of advance reservation meta-scheduling". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 137–153, Springer Verlag, 2000. Lect. Notes Comput. Sci. vol. 1911.
- [45] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Selective reservation strategies for backfill job scheduling". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 55–71, Springer-Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.
- [46] D. Talby and D. G. Feitelson, "Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling". In *13th Intl. Parallel Processing Symp.*, pp. 513–517, Apr 1999.
- [47] D. Tsafirir. in preparation.
- [48] A. Uno, T. Aoyagi, and K. Tani, "Job scheduling on the earth simulator". *NEC Res. & Develop.* **44(1)**, pp. 47–52, Jan 2003.
- [49] U. Schwiegelshohn and R. Yahyapour, "GGF-GFD.6: Attributes for Communication between Scheduling Instances". <http://www.ggf.org/documents/GFD/GFD-I-6.pdf>, Dec 2001.
- [50] Y. Wiseman and D. G. Feitelson, "Paired gang scheduling". *IEEE Trans. Parallel & Distributed Syst.* **14(6)**, pp. 581–592, Jun 2003.
- [51] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: simple Linux utility for resource management". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 44–60, Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.
- [52] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration". *IEEE Trans. Parallel & Distributed Syst.* **14(3)**, pp. 236–247, Mar 2003.
- [53] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam, "Improving parallel job scheduling by combining gang scheduling and backfilling techniques". In *14th Intl. Parallel & Distributed Processing Symp.*, pp. 133–142, May 2000.
- [54] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a load sharing facility for large, heterogeneous distributed computer systems". *Software — Pract. & Exp.* **23(12)**, pp. 1305–1336, Dec 1993.
- [55] D. Zotkin and P. J. Keleher, "Job-length estimation and performance in backfilling schedulers". In *8th Intl. Symp. High Performance Distributed Comput.*, Aug 1999.