An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific

H. Franke J. Jann J. E. Moreira P. Pattnaik

IBM T. J. Watson Research Center
Yorktown Heights NY 10598-0218
{*frankeh,joefon,jmoreira,pratap*}*@us.ibm.com*

M. A. Jette

Lawrence Livermore National Laboratory
Livermore CA 94550
jette@llnl.gov

Abstract

In this paper we analyze the behavior of a gang-scheduling system that we are developing for the ASCI Blue-Pacific machines. Starting with a real workload obtained from job logs of one of the ASCI machines, we generate a statistical model of this workload using Hyper Erlang distributions. We then vary the parameters of those distributions to generate various workloads, representative of different operating points of the machine. Through simulation we obtain performance characteristics for three different scheduling strategies: (i) first-come first-serve, (ii) gang-scheduling, and (iii) backfilling. Our results show that both backfilling and gang-scheduling with moderate multiprogramming levels are much more effective than simple first-come first-serve scheduling. In addition, we show that gang-scheduling can display better performance characteristics than backfilling, particularly for large production jobs.

1 Introduction

Parallel job scheduling in large systems with hundreds or thousands of processors can be very challenging. A good job scheduling system works to maximize such objective measures as average job response time and system utilization. It also attempts to maximize the more subjective measure of user happiness. It has been shown in the literature that the scheduling strategy can have significant impact on the performance characteristics of a large parallel system [1].

In this paper we analyze the behavior of a gang-scheduling strategy that we are developing for the ASCI Blue-Pacific machines, at the Lawrence Livermore National Laboratory (LLNL) [11]. The larger of these machines (the SST machine, used for classified research) has approximately 1500 nodes, or 6000 processors. (Each node is a 4-processor SMP.) Even the "smaller" machine (the CTR machine, open to general research) has 320 nodes and more than 1200 processors. Academic and industrial experience in job scheduling for such large systems is practically nonexistent.

The workload of the CTR machine is very diverse, with many small jobs that only use one or a few nodes but also some very large jobs that use up to 256 nodes. One of the main goals of job scheduling for the CTR machine is to provide good average response time while efficiently servicing these very large jobs. Typical scheduling strategies for large parallel systems include first-come first-serve (FCFS), backfilling, and gang-scheduling. (Gang-scheduling with multiprogramming level of 1 reverts to FCFS.) FCFS and backfilling are already provided through current implementations of LoadLeveler. We have extended LoadLeveler with gang-scheduling, in a system called GangLL, to provide a more flexible scheduling strategy.

Our goal in this paper is to conduct a workload sensitivity analysis of the performance of three common scheduling strategies (FCFS, backfilling, and gang-scheduling) for the CTR machine. It is important to check the range of acceptable operability of a new scheduling system before it is deployed so that unpleasant surprises can be avoided. Although we do have job logs of the current use of the CTR machine, an increase in demand is expected. (An increase in demand is both a cause for and an effect of a better scheduling system.) To evaluate the performance of the schedulers at increasing loads, we use the following approach: We start with a real workload obtained from job logs of this machine. We then generate a statistical model of this workload using Hyper Erlang distributions [9, 13]. We vary the parameters of these distributions to generate several workloads with different behaviors. Finally, we obtain performance characteristics for each of these workloads through simulation of the various scheduling strategies.

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

Our study shows that good space-sharing strategies, in particular backfilling, can deliver good response time averaged over all jobs. We also show that gang-scheduling with modest multiprogramming levels likewise delivers good average response time, but with the following added benefits: (i) an estimate of the job termination time is unnecessary, (ii) the response time for large production jobs is reduced, and (iii) the availability of the system for large jobs is increased. It is important to emphasize the relevance of large jobs. One can easily reduce average job response time by always favoring small jobs at the expense of large jobs. However, large jobs, although not numerous, typically represent important production runs and improving their response time is an important step in achieving happiness of the user community. With our sensitivity analysis, we show that gang-scheduling continues to deliver good response time even when utilization is driven to much higher levels than currently observed.

The rest of this paper is organized as follows. Section 2 gives some details of the ASCI Blue-Pacific machines. Section 3 discusses the scheduling strategies analyzed in this paper. Section 4 describes the organization of our GangLL system. Section 5 describes our methodology for evaluating the behavior of the different strategies. Section 6 presents and discusses the results of our evaluation. Finally, Section 7 presents our conclusions and discusses future work.

2 The ASCI Blue-Pacific machines

The main machine in the ASCI Blue-Pacific program is the *Sustained Stewardship TeraOPS* (SST) "Hyper-Cluster". Figure 1 shows the high-level organization of the SST machine. It consists of three 488-node *sectors* that are connected via high performance gateway links. Each node is comprised of a 4-way SMP with 332 MHz PowerPC 604e processors and up to 2.5 GB of main memory. Each node executes its own operating system (AIX) image. The nodes in each sector are interconnected via a TBMX high-performance switch that delivers a bidirectional bandwidth of 150 MB/s per node.

While the SST machine is for use on classified applications, the ASCI Blue-Pacific program also provides computing resources to academic research through the ASCI Strategic Alliance Program. In this program the *Combined Technology Refresh* (CTR) machine provides 320 nodes of similar characteristics as the SST nodes, organized in a single sector. Features of both machines are summarized in Table 1. For the remaining of this paper we discuss job scheduling for the CTR machine. Techniques developed for the CTR machine may eventually be incorporated into the SST machine.

Table 1: Characteristics of the ASCI Blue-Pacific machines.

characteristic	SST	CTR
Peak speed	3.9 teraOPS	850 gigaOPS
Memory	2.6 terabytes	480 gigabytes
Local Disk	17.3 terabytes	3.0 terabytes
Min/Max memory/node	1.5-2.5 gigabytes	1.5 gigabytes
Number of compute nodes (4-way SMPs)	1,464	320
Node-to-node bandwidth (TBMX switch, bidirectional)	150 MB/s	150 MB/s
Total number of processors	5,856	1,280
Processor-to-memory bandwidth	2.1 TB/s	460 GB/s
Compute node peak performance	2.656 gigaOPS	2.656 gigaOPS
Delivered RAID I/O bandwidth	6.4 GB/s	320 MB/s
Delivered I/O bandwidth to local disk	10.5 GB/s	4.7 GB/s
RAID storage	62.5 terabytes	10.0 terabytes

3 Scheduling strategies for ASCI Blue-Pacific

Despite the size of the CTR machine, we know from LLNL's experience [2, 5, 8] with this and other large systems that the demand for resources to execute jobs is often larger than the total number of processors available. The traditional

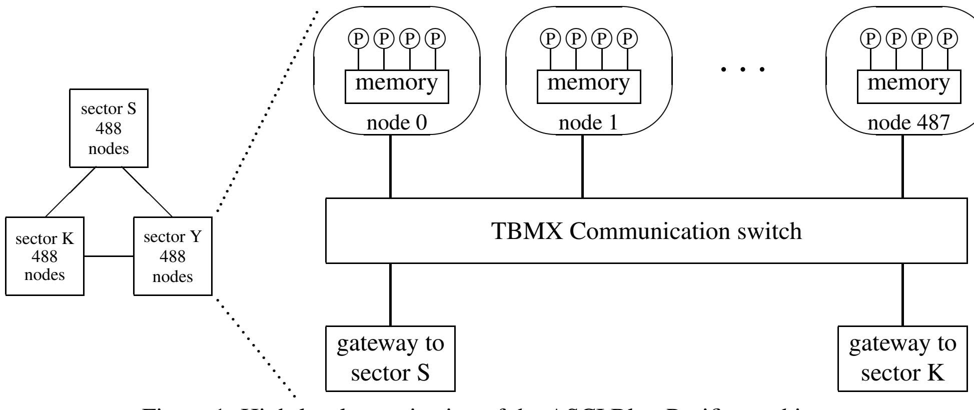


Figure 1: High-level organization of the ASCI Blue-Pacific machine.

solution to this problem in a space-sharing environment is to schedule jobs for execution strictly in the order they arrive (FCFS) and to queue the excess jobs: They wait to start execution until some running jobs have finished. It is well known in the literature that this can be highly detrimental to system and job performance. Simply queueing jobs until enough resources are available leads to low system utilization and high job response time. The problem can be alleviated in part by clever space-sharing scheduling techniques, such as backfilling [10]. With backfilling, jobs are also scheduled for execution in the order they arrive, but the scheduler looks past jobs that cannot be immediately started. Jobs are scheduled to run as soon as possible, so long as they do not delay the execution of a job that arrived earlier. This is also called *conservative backfilling*. Another variation of backfilling, called *aggressive backfilling*, schedules jobs to run as soon as possible, so long as they do not delay the *first* waiting job. The literature shows that both approaches result in comparable performance in real systems [3]. Therefore, in this paper we consider conservative backfilling only. We note that, to perform backfilling, the scheduler must have an estimate of the termination time for each job.

Backfilling can be very effective in increasing machine utilization and average job response time. However, it still presents some problems to users of large jobs. On one hand, execution of large jobs are postponed until enough processors are available. On the other hand, once large jobs start, they monopolize the resources for long periods of time, which in turn make the machine inaccessible to other users. Consequently, large computing centers typically restrict running large jobs to evenings and weekends.

A more flexible solution to the problem of executing a large number of jobs of varied characteristics is to share the machine's resources not only spatially but also temporally [2, 4, 6, 15]. The examples in the literature illustrate the benefit of adding this other axis of partitioning for parallel systems. It can result in better system utilization and reduced response time. In addition to time-slicing resources among several jobs, systems that support time-sharing typically also support the important feature of *preemption*. Preemption allows a lower priority job to be suspended so that a high priority job can run. Both time-slicing and preemption are supported in GangLL. When time-slicing multiple jobs in the same nodes, processes from all active jobs have to coexist in the same system image. This puts pressure on both the virtual and physical memory systems and effectively limits the degree of multiprogramming (MPL) that can be supported efficiently. For the CTR machine, an MPL of 2 or 3 is likely to be feasible. This degree of multiprogramming is expected to increase in future systems (*e.g.*, ASCI White) which will have larger memory.

Time-sharing on a large scale distributed parallel machine is complicated by two issues: (i) the native schedulers of the operating systems executing on each node are not coordinated, and (ii) tasks of distributed jobs interact tightly through the communication switch. The tasks of a parallel job must be coscheduled [14] (*i.e.*, must run concurrently on all nodes) or inefficient communication behavior results. Without coscheduling, receivers may not be ready when senders are and vice-versa. As a result, we adopt a coarse grain time-sharing strategy for GangLL that guarantees tasks of the same job execute simultaneously despite operating system images not being coordinated. We accomplish this by (i) partitioning the time axis into large time slices (order of seconds to minutes), (ii) populating the time slices with tasks from parallel jobs so that all tasks of a job occupy the same time-slice, and (iii) implementing the schedule at the individual nodes. The schedule is represented by an Ousterhout matrix [14], where the columns correspond to processors and the rows correspond to revolving time slices. We note that, during a particular time-slice, the processor is dedicated to running a single task. If that task is blocked (*e.g.*, I/O or paging) the processor remains idle.

4 Organization of GangLL

The high-level organization of GangLL is shown in Figure 2. At LLNL, users submit jobs to a meta batch job system called Distributed Production Control System (DPCS). DPCS forwards the jobs for execution to the appropriate system. In our particular case, DPCS interacts with *Schedd*, which is the job submission agent for GangLL. Submitted jobs are placed in the queue of jobs and are scheduled for execution by the *Scheduler* module of the *Central Manager*. Some degree of control of job scheduling and execution can be performed by a site specific external scheduler [11].

A parallel job consists of a set of tasks. Each task runs on a single node of the machine, using one or more processors on that node. For conventional MPI programs one task is initiated for each processor, but at LLNL about half the programs initiate one task per node and multithread to make use of the multiple CPUs in the node.

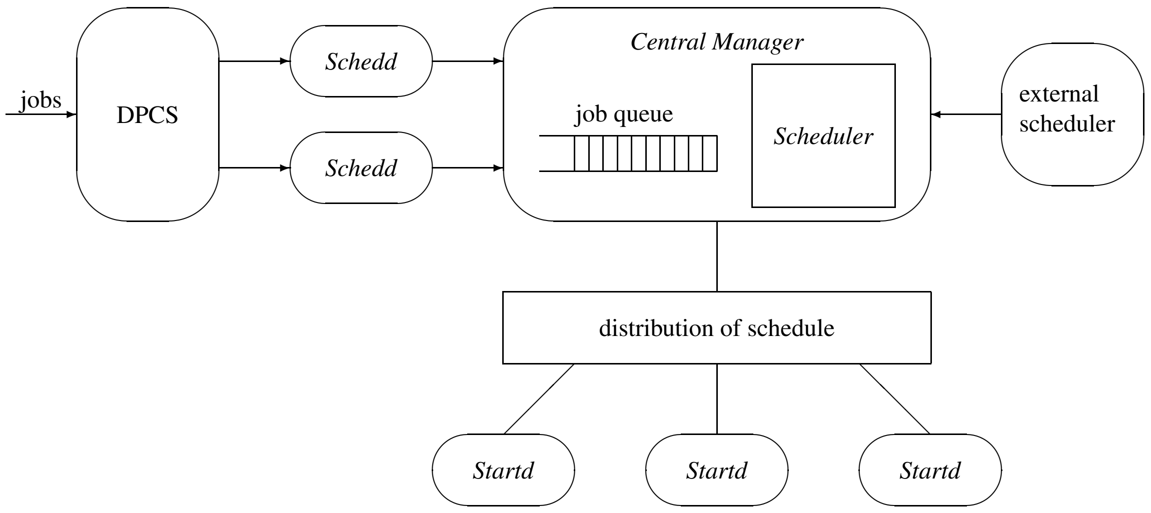


Figure 2: General organization of the GangLL prototype.

The ultimate goal of the *Scheduler* is to build the Ousterhout matrix that precisely defines a schedule for the system. The *Scheduler* has to modify and/or rebuild the matrix to accommodate new jobs, remove jobs that have terminated, and handle reconfigurations in system and job parameters. When scheduling jobs, the *Scheduler* starts by selecting which jobs to execute. Job execution can be constrained by the multiprogramming level allowed and also by rules defined by the system administrators. A typical use of these rules is to constrain which classes of jobs can time-share a node with other jobs. Once the selection is done, the *Scheduler* performs the temporal and spatial allocation of tasks to processors, thus building an Ousterhout matrix.

The Ousterhout matrix built by the *Scheduler* is a wall-clock matrix in the sense that it defines the actual wall-clock time at which each time-slice begins and ends. Each node has a *Startd* LoadLeveler daemon that is responsible for implementing the schedule defined by the columns of the matrix corresponding to processors in that node. That is, the *Startd* acts as a *node-level scheduler*. For the *Startd* daemons to perform coordinated context-switches at the time-slices boundaries, it suffices that they have some form of synchronized wall-clocks (*e.g.*, by running NTP). No explicit synchronization among daemons is necessary. Note that the synchronization does not have to be strict. The wall-clocks only need to be synchronized within a very small fraction of the time-slice (*e.g.*, a part in a thousand). Each *Startd* has to perform the context-switch for all the tasks running in its node.

The columns of the Ousterhout matrix have to be distributed from the *Central Manager* to the corresponding *Startd* daemons in a scalable and consistent manner. Although the matrix itself is not too big (a few kilobytes), it has to be distributed efficiently to hundreds of nodes. Furthermore, we do want to avoid the undesirable situation in which some nodes start to use a newly distributed version of the matrix while other nodes are still using an old version. We solve this problem through a two-phase, hierarchical distribution scheme [11, 12].

5 Evaluation methodology

When selecting and developing job schedulers for use in large parallel system installations, it is important to understand their expected performance. The first stage is to have a characterization and a procedure to synthetically generate the expected workloads. Our methodology for generating these workloads involves the following steps:

1. Fit a typical workload with mathematical models.
2. Generate synthetic workloads based on the derived mathematical models.
3. Determine the waiting time, response time and other parameters of interest for different scheduling policies using these synthetic workloads and a simulation of the scheduler.

When fitting a parallel load, it is very useful to be able to find a compact mathematical representation that is expressible by a few parameters, is reasonably easy to use for the generation of synthetic workloads, and is also suitable for theoretical queuing analysis of scheduling algorithms.

Parallel workloads often are over-dispersive. That is, job interarrival time distribution and job service time (execution time on a dedicated system) distribution each has a coefficient of variation that is greater than one. Distributions with coefficient of variation greater than one are also referred to as long-tailed distributions, and can be fitted adequately with Hyper Erlang Distributions of Common Order. In an earlier work [7] we developed such a model, and demonstrated its efficacy by using it to fit a typical workload from the Cornell University Theory Center. Here we use this model to fit a typical workload from the ASCI Blue-Pacific System.

The Hyper Erlang Distribution of Common Order is a special form of a more general type of distribution named Phase Type Distribution. The Phase Type Distribution is a distribution (often used in queuing theory) which is capable of representing any stochastic process whose associated probability density function has a Laplace transform that is a rational function. Practically all the relevant systems one encounters in the stochastic modeling of computer workloads can be modeled by a Phase Type Distribution.

The Hyper Erlang Distribution of Common Order in turn is a generalization of three more commonly known distributions – namely exponential, hyper exponential, and Erlang distributions. The Hyper Erlang Distribution of Common Order has a Laplace transform of the form

$$f^*(s) = \sum_{i=1}^{2} p_i \Big(\frac{\lambda_i}{s+\lambda_i}\Big)^n \tag{1}$$

where n, a positive integer, is called the order of the distribution, and $0 \leq p_i \leq 1$ with $p_1 + p_2 = 1$. The Erlang distribution is a special case of the Hyper Erlang Distribution of Common Order with one of the p_is equal to 1 (*e.g.*, $p_1 = 1$). The hyper exponential distribution is a Hyper Erlang Distribution of Common Order with $n = 1$. The exponential distribution is also a special case with $p_1 = 1$ and $n = 1$. The k^{th} noncentral moment of a distribution, for all integers $k \geq 1$, can be obtained from the Laplace transform of the distribution by

$$\mu_k = E[t^k] = (-1)^k \Big[\frac{d^k f^*(s)}{ds^k}\Big]_{s=0} \tag{2}$$

which, for Hyper Erlang Distribution of Common Order, is

$$\mu_k = \sum_{i=1}^{2} p_i \frac{n(n+1)...(n+k-1)}{\lambda_i^k} \tag{3}$$

The moments for the Erlang distribution are obtained by setting $p_1 = 1$ and $p_2 = 0$ in equation 3, yielding

$$\mu_k = \frac{n(n+1)...(n+k-1)}{\lambda^k}. \tag{4}$$

The moments for the hyper exponential distribution are obtained by setting $n = 1$ in equation 3, yielding

$$\mu_k = \sum_{i=1}^{2} p_i \frac{k!}{\lambda_i^k} \tag{5}$$

The moments for the exponential distribution is obtain by letting $n = 1$ in equation 4, giving

$$\mu_k = \frac{k!}{\lambda^k} \tag{6}$$

There are a number of interrelationships among the moments to ensure physical consistency. In particular, we must have nonnegative p_is and λ_is. The procedure for the extraction of the model parameters need to preserve these constraints. The algebra involved is detailed in [7]. Basically the procedure automatically selects the simplest distribution (exponential, hyper exponential, Erlang, or Hyper Erlang) that is commensurate with the first three moments, and the related constraints in the observed data. More precisely, our modeling procedure involves the following steps:

1. First we group the jobs into classes, based on the number of processors they require to execute on. Each class is a bin in which the upper boundary is a power of 2.
2. Then we model the interarrival time distribution for each class, and the service time distribution for each class as follows:
   (a) From the job traces, we compute the first 3 moments, $\mu_1^o,\ \mu_2^o,\ \mu_3^o$, of the observed interarrival time and those of the observed service time.
   (b) Then we select the Hyper Erlang Distribution of Common Order that fits these 3 observed moments. We chose to fit the moments of the model against those of the actual data because the first 3 moments usually capture the generic features of the workload and are more robust to the effect of outliers. These three moments carry the information on the mean, variance, and skewness of the random variable respectively.

Next we generate various synthetic workloads from the observed workload by varying the interarrival rate and service time used. The Hyper Erlang parameters for these synthetic workloads are obtained by multiplying the interarrival rate and the service time each by a separate multiplicative factor, and by specifying the number of jobs to generate. From these model parameters the actual job trace is obtained using the procedure described in [7]. Finally we simulate the effects of these synthetic workloads with a simulator and observe the results.

6 Results

In this section we discuss the performance characteristics of the three scheduling strategies available for the CTR machine: FCFS, backfilling, and gang-scheduling. We consider three different configurations of gang-scheduling, with maximum multiprogramming levels (MPL) of 2, 3, and 5. Note that, during simulation, these are the highest MPLs allowed for that particular configuration. The actual multiprogramming level at any particular time and any particular node is determined by the number of jobs allocated to that node and can be less than or equal to the maximum MPL for the system. As discussed in Section 1, backfilling requires an estimate of job execution time. As a limiting performance reference we consider what we call *perfect backfilling*, in which the estimate of the job execution time, as used by the scheduler, is exactly equal to the actual job execution time.

Each job in the system is characterized by the number of nodes it uses and its submission, start, and termination times. The submission time of a job is the time it was submitted for execution in the system. Its start time is the time it actually starts running on nodes, after possibly waiting in the queue. Finally, the termination time is the time the job completes execution and leaves the system. We describe the performance characteristics of our scheduling systems through plots of average response time and average wait time as functions of utilization. These system parameters are defined as follows:

1. *utilization:* This is a measure of how busy the nodes in the machine are during the simulated period. Total CPU time for one job is computed by multiplying its number of nodes by its execution time *on a dedicated system*. (This is the same as the service time discussed in Section 5.) We obtain utilization by summing CPU time for all jobs and dividing by the total number of nodes times the length of the simulated interval. The utilization is always a number between 0 and 1.

2. *average job response time:* The response time of a job is computed as its termination time minus its submission time. We then compute an average for all jobs as well as an average for *large* jobs. A large job is defined as any job that uses more than 32 nodes (or 10% of the CTR machine). This parameter is reported in seconds.

3. *average job wait time:* The wait time of a job is computed as its start time minus its submission time. We again compute an average for all jobs as well as an average for large jobs. This parameter is also reported in seconds.

The parameters for the Hyper Erlang distributions that model the actual LLNL workload are shown in Table 2 and Table 3. There is one model for each job class. Columns $N_{\min}$ and $N_{\max}$ define the job class through the minimum and maximum number of nodes for jobs in that class. Columns λ_1, λ_2, n, and p_1 are the parameters for the Hyper Erlang distribution, as described in Section 5. Column E_4 is the relative discrepancy (in percentage) between the noncentral fourth moment of the data and that of the model. It gives an estimate of the accuracy of the fit. Finally, the last column of the tables gives the percentage of jobs in the workload that belong to that particular class.

The baseline workload consists of 10000 jobs generated by the Hyper Erlang model for the actual LLNL workload. The simulated period spans approximately 60 days. Some characteristics of this workload are shown in Figure 3 and Figure 4. Figure 3 reports the distribution of job sizes (number of nodes). For each job size, between 1 and 256, Figure 3(a) shows the number of jobs of that size, while Figure 3(b) plots the number of jobs with *at most* that size. (In other words, Figure 3(b) is the integral of Figure 3(a).) Figure 4 reports the distribution of CPU time. For each job size, Figure 4(a) shows the sum of the CPU times for all jobs of that size, while Figure 4(b) is a plot of the sum of the CPU times for all jobs of *at most* that size. (In other words, Figure 4(b) is the integral of Figure 4(a).) From Figure 3 and Figure 4 we observe that, although large jobs (those with more than 32 nodes), represent only 30% of the number of jobs, they constitute more than 80% of the total work performed in the system. This baseline workload corresponds to an utilization of 0.55.

Table 2: Hyper Erlang parameters for job interarrival time (seconds).

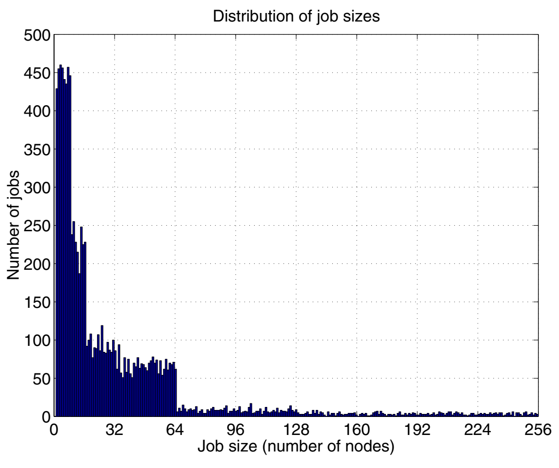
$N_{\min}$	$N_{\max}$	λ_1	λ_2	n	p_1	E_4	% of jobs
1	8	1.02e-04	2.06e-03	1	1.07e-01	11%	37.18
9	16	1.69e-04	1.40e-03	1	4.10e-01	11%	19.63
17	32	1.94e-04	3.02e-03	2	3.09e-01	7%	13.86
33	64	3.17e-04	2.15e-03	1	7.71e-01	7%	21.48
65	128	8.94e-05	4.56e-03	3	3.07e-01	15%	3.93
129	256	7.94e-05	1.71e-03	2	4.26e-01	10%	3.93

Table 3: Hyper Erlang parameters for job service (execution) time (seconds).

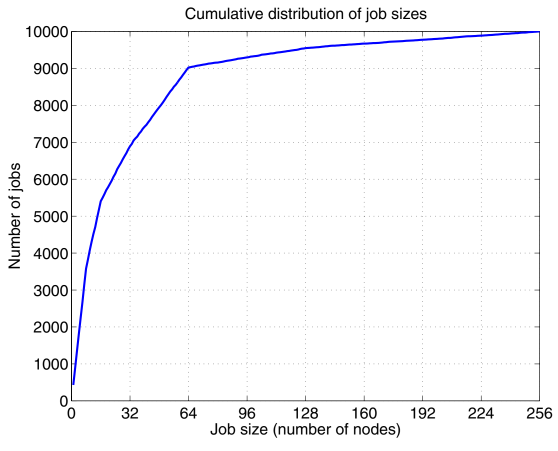
$N_{\min}$	$N_{\max}$	λ_1	λ_2	n	p_1	E_4	% of jobs
1	8	5.20e-04	4.65e-03	1	2.85e-01	14%	37.18
9	16	1.15e-03	6.08e-02	5	5.83e-01	7%	19.63
17	32	1.32e-04	1.50e-01	1	3.03e-01	21%	13.86
33	64	4.85e-04	1.09e-02	2	9.00e-01	3%	21.48
65	128	8.25e-04	4.71e-02	3	5.38e-01	11%	3.93
129	256	4.52e-04	8.61e-03	3	4.37e-01	9%	3.93

In addition to the baseline workload of Figure 3 and Figure 4 we generate 80 additional workloads, of 10000 jobs each, by varying the model parameters so as to increase average job execution time and to decrease average job interarrival time. For a fixed interarrival time, increasing job execution time typically increases utilization, until the system saturates. The same is true for decreasing job interarrival time for a fixed job execution time.

Results for response time and wait time as a function of utilization, averaged over all jobs, are shown in Figure 5 and Figure 6 respectively. Each plot is for a particular average job interarrival time and has results for each of the

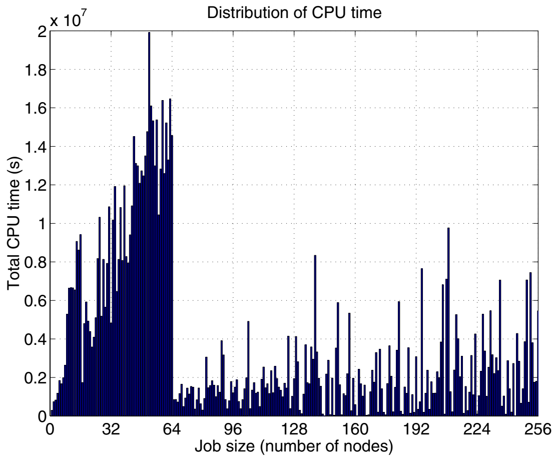


(a)

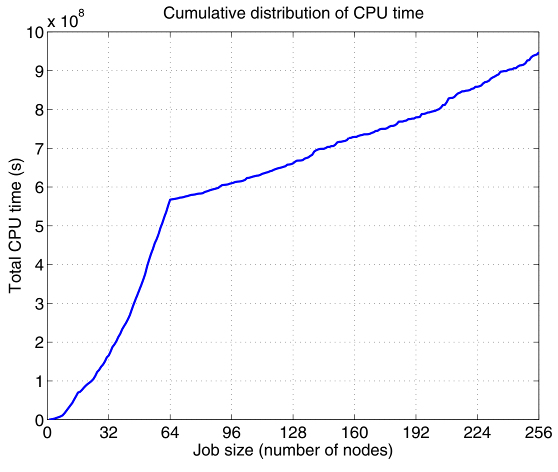


(b)

Figure 3: Workload characteristics: distribution of job sizes.



(a)



(b)

Figure 4: Workload characteristics: distribution of cpu time.

scheduling strategies considered. The same results, averaged only over large jobs, are shown in Figure 7 and Figure 8. Results for gang-scheduling (indicated by the MPL 2, MPL 3, and MPL 5 lines) are for a negligible context switch time.

We first note that backfilling can be very effective in improving system performance, when compared to FCFS. We observe that the performance of gang-scheduling, measured by average job response time and average job wait time, improves as the maximum multiprogramming level increases. This behavior has been previously described in [11]. Even gang-scheduling with a low MPL of 2 or 3 offers a significant improvement over FCFS. Over a wide range of utilizations, perfect backfilling performs slightly worse than gang-scheduling with MPL of 5 with respect to average job response time. However, in terms of average job response time for large jobs, perfect backfilling performs only as well as gang-scheduling with MPL of 3. In terms of average wait time for large jobs, gang-scheduling with an MPL of 3 or 5 is far superior to backfilling.

As shown in Figure 7 and Figure 8, the impact of the scheduling strategy on the performance of large jobs can be

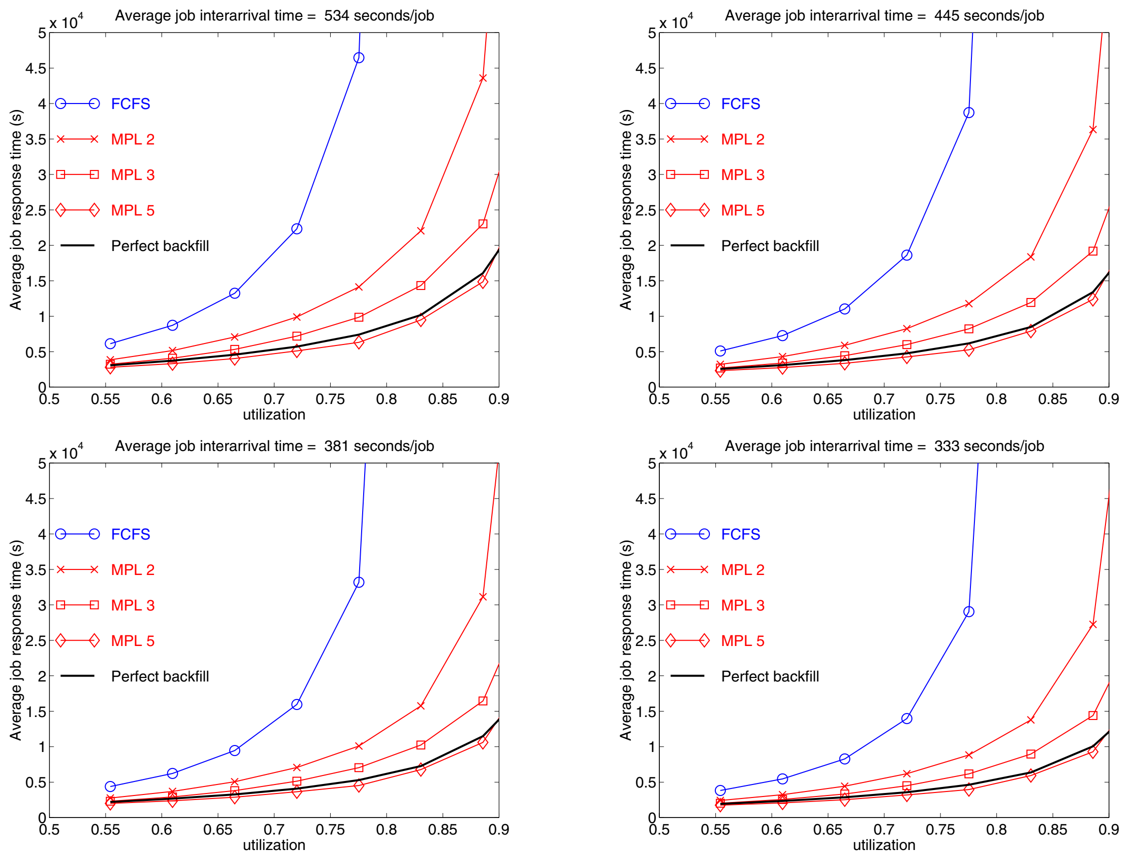


Figure 5: Response times as a function of utilization, averaged over all jobs.

significant. Gang-scheduling is effective in both reducing the average response time for large jobs and in increasing machine availability for these jobs, as indicated by the reduced wait time. As the workload in the CTR machine matures, it is expected that large jobs will be more common and their response time and wait time behavior will be more important. Even at the present mix of small and large jobs, gang-scheduling with low MPLs of 2 and 3 allows the system to operate with utilizations of up to 0.90.

So far we have only considered the situation in which there is negligible context-switch time. This is typically the case unless memory requirements of the jobs are so large that paging will be induced by gang-scheduling context switches. The mean resident set size of the current workload is 133 MB per node, or less than ten percent of the CTR machine's main memory per node. Therefore, paging is not expected to significantly impact context switch times with the gang-scheduling multiprogramming levels being modeled. Memory demands are higher on the SST machine and are expected to increase on both machines as the workload matures. At this point we do not know the precise distribution of memory usage for future production jobs at LLNL. Neither does GangLL incorporate any memory-usage conscious scheduling policies. Nevertheless, we can show that there is a potential problem which warrants further work.

Figure 9 shows the behavior of the sequential (single task) NAS benchmark LU class A under control of GangLL. Each instance of this job has a 45 MB memory footprint. In both the 128 MB node (dark bars) and the 256 MB node (light bars), time-sharing 2 instances of LU class A (multiprogramming level of 2) behaves close to the ideal case (twice the time for a dedicated execution). However, paging effects are severe under a multiprogramming level of three for the 128 MB node. The average execution time for the three jobs is 3.5 times greater than in the 256 MB node under the same conditions.

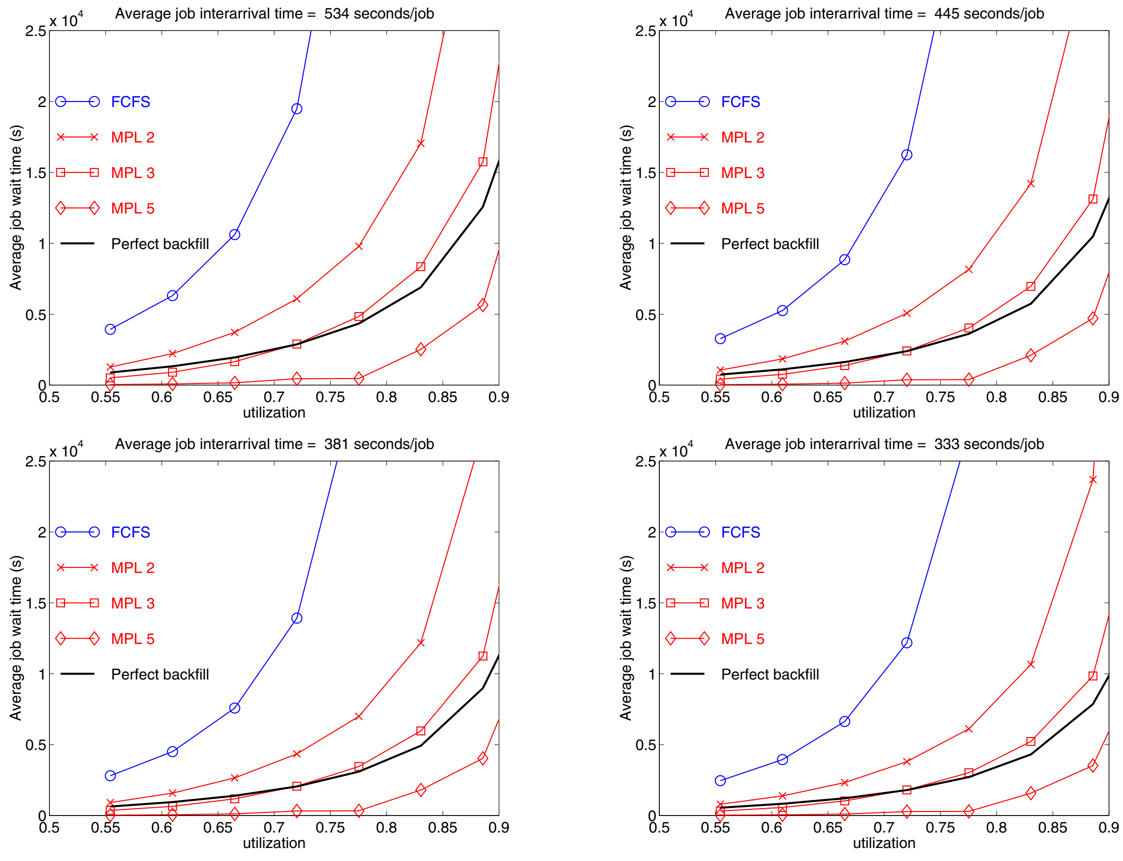


Figure 6: Wait times as a function of utilization, averaged over all jobs.

7 Conclusions

The ASCI Blue-Pacific machines, because of their size and diverse workloads, present new challenges to parallel job scheduling systems. Strategies that rely exclusive on space-sharing, such as backfilling, can be effective in improving system utilization. However, to accomplish the other scheduling objectives mandated by LLNL, particularly with respect to large jobs, it is necessary to use time-sharing as well.

We have developed a gang-scheduling system, GangLL, that performs both space- and time-sharing of resources in a parallel system. Our performance analysis shows that, even at modest multiprogramming levels of 2 and 3, gang-scheduling can accommodate very high utilization rates of up to 90%.

Gang-scheduling is expected to be deployed in the ASCI CTR machine sometime in the near future. We will then be able to assess the behavior of GangLL in an actual production environment. Feedback from our observations will allows us to improve our performance models and will provide guidance for future development of gang-scheduling strategies.

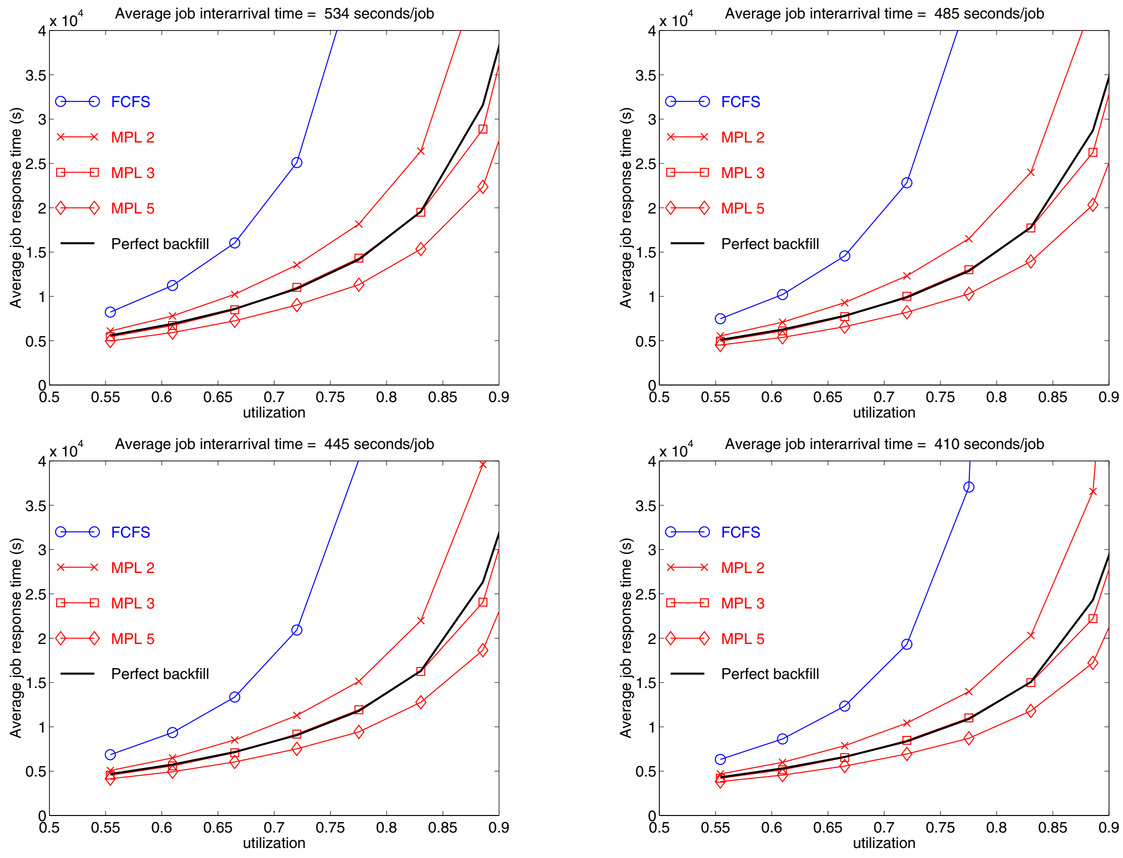


Figure 7: Response times as a function of utilization, averaged over large jobs (nodes > 32).

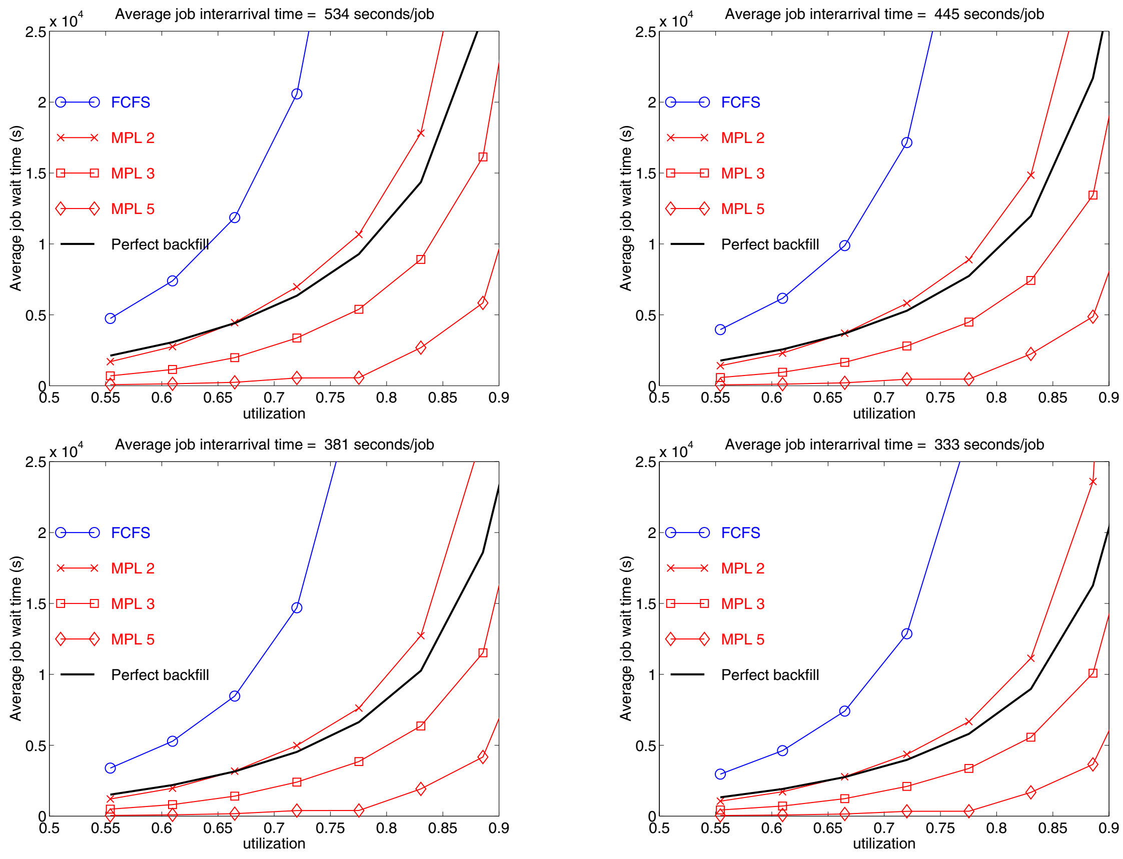


Figure 8: Wait times as a function of utilization, averaged over large jobs (nodes > 32).

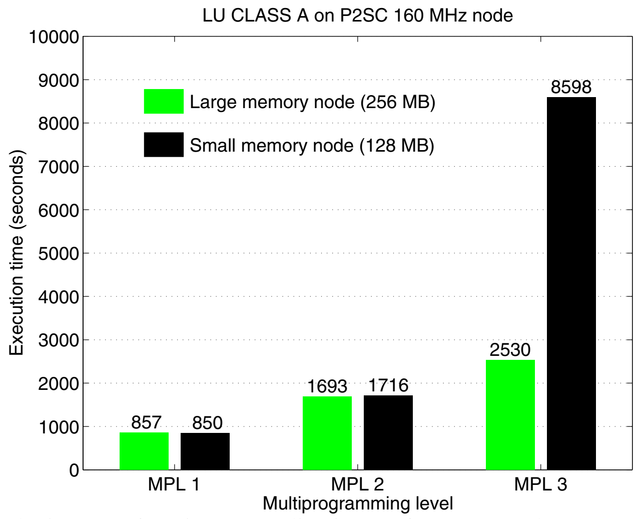


Figure 9: The impact of paging on application performance under gang-scheduling.

References

[1] D. G. Feitelson. **A Survey of Scheduling in Multiprogrammed Parallel Systems**. Technical Report RC 19790 (87657), IBM T. J. Watson Research Center, October 1994.

[2] D. G. Feitelson and M. A. Jette. **Improved Utilization and Responsiveness with Gang Scheduling**. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261. Springer-Verlag, April 1997.

[3] D. G. Feitelson and A.M. Weil. **Utilization and predictability in scheduling the IBM SP2 with backfilling**. In *12th International Parallel Processing Symposium*, pages 542–546, April 1998.

[4] H. Franke, P. Pattnaik, and L. Rudolph. **Gang Scheduling for Highly Efficient Multiprocessors**. In *Sixth Symposium on the Frontiers of Massively Parallel Computation, Annapolis, Maryland*, 1996.

[5] B. Gorda and R. Wolski. **Time Sharing Massively Parallel Machines**. In *International Conference on Parallel Processing*, volume II, pages 214–217, August 1995.

[6] N. Islam, A. L. Prodromidis, M. S. Squillante, L. L. Fong, and A. S. Gopal. **Extensible Resource Management for Cluster Computing**. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 561–568, 1997.

[7] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riordan. **Modeling of Workload in MPPs**. In *Proceedings of the 3rd Annual Workshop on Job Scheduling Strategies for Parallel Processing*, pages 95–116, April 1997. In Conjunction with IPPS'97, Geneva, Switzerland.

[8] M. Jette, D. Storch, and E. Yim. **Timesharing the Cray T3D**. *Cray User Group*, pages 247–252, March 1996.

[9] M. A. Johnson and M. R. Taaffe. **Matching Moments to Phase Distributions: Mixtures of Erlang Distributions of Common Order**. *Communications in Statistics – Stochastic Models*, 5(4):711–743, 1989.

[10] D. Lifka. **The ANL/IBM SP scheduling system**. In *IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer-Verlag, April 1995.

[11] 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 *Proceedings of SC98, Orlando, FL*, November 1998.

[12] J. E. Moreira, H. Franke, W. Chan, L. L. Fong, M. A. Jette, and A. Yoo. **A Gang-Scheduling System for ASCI Blue-Pacific**. In *Proceedings of the 7th International Conference on High-Performance Computing and Networking (HPCN'99)*, volume 1593 of *Lecture Notes in Computer Science*, pages 831–840. Springer, April 1999.

[13] R. Nelson. **Probability, Stochastic Processes, and Queueing Theory**. Springer-Verlag, 1995.

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

[15] U. Schwiegelshohn and R. Yahyapour. **Improving First-Come-First-Serve Job Scheduling by Gang Scheduling**. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.