The Impact of Migration on Parallel Job Scheduling for Distributed Systems

Yanyong Zhang[1], Hubertus Franke[2], Jose E. Moreira[2], and Anand Sivasubramaniam[1]

[1] Department of Computer Science & Engineering
The Pennsylvania State University
University Park PA 16802
{yyzhang, anand}@cse.psu.edu
[2] IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights NY 10598-0218
{frankeh, jmoreira}@us.ibm.com

Abstract. This paper evaluates the impact of task migration on gang-scheduling of parallel jobs for distributed systems. With migration, it is possible to move tasks of a job from their originally assigned set of nodes to another set of nodes, during execution of the job. This additional flexibility creates more opportunities for filling holes in the scheduling matrix. We conduct a simulation-based study of the effect of migration on average job slowdown and wait times for a large distributed system under a variety of loads. We find that migration can significantly improve these performance metrics over an important range of operating points. We also analyze the effect of the cost of migrating tasks on overall system performance.

1 Introduction

Scheduling strategies can have a significant impact on the performance characteristics of large parallel systems [3, 5, 6, 7, 12, 13, 17]. When jobs are submitted for execution in a parallel system they are typically first organized in a job queue. From there, they are selected for execution by the scheduler. Various priority ordering policies (FCFS, best fit, worst fit, shortest job first) have been used for the job queue. Early scheduling strategies for distributed systems just used a space-sharing approach, wherein jobs can run side by side on different nodes of the machine at the same time, but each node is exclusively assigned to a job. When there are not enough nodes, the jobs in the queue simply wait. Space sharing in isolation can result in poor utilization, as nodes remain empty despite a queue of waiting jobs. Furthermore, the wait and response times for jobs with an exclusively space-sharing strategy are relatively high [8].

Among the several approaches used to alleviate these problems with space sharing scheduling, two have been most commonly studied. The first is a technique called backfilling, which attempts to assign unutilized nodes to jobs that are behind in the priority queue (of waiting jobs), rather than keep them idle. A lower priority job can be scheduled before a higher priority job as long as it does not delay the start time of that job. This requirement of not delaying higher priority jobs imposes the need for an estimate of job execution times. It has already been shown [4, 13, 18] that a FCFS queueing policy combined with backfilling results in efficient and fair space sharing scheduling. Furthermore, [4, 14, 18] have shown that overestimating the job execution time does not significantly change the final result.

The second approach is to add a time-sharing dimension to space sharing using a technique called gang-scheduling or coscheduling [9]. This technique virtualizes the physical machine by slicing the time axis into multiple space-shared virtual machines [3, 15], limited by the maximum multiprogramming level (MPL) allowed in the system. The schedule is represented as a cyclical Ousterhout matrix that defines the tasks executing on each processor and each time-slice. Tasks of a parallel job are coscheduled to run in the same time-slices (same virtual machines). A cycle through all the rows of the Ousterhout matrix defines a *scheduling cycle*.

Gang-scheduling and backfilling are two optimization techniques that operate on orthogonal axes, space for backfilling and time for gang-scheduling. The two can be combined by treating each of the virtual machines created by gang scheduling as a target for backfilling. We have demonstrated the efficacy of this approach in [18].

The approaches we described so far adopt a static model for space assignment. That is, once a job is assigned to nodes of a parallel system it cannot be moved. We want to examine whether a more dynamic model can be beneficial. In particular, we look into the issue of *migration*, which allows jobs to be moved from one set of nodes to another, possibly overlapping, set [16]. Implementing migration requires additional infrastructure in many parallel systems, with an associated cost. Migration requires significant library and operating system support and consumes resources (memory, disk, network) at the time of migration [2].

This paper addresses the following issues which help us understand the impact of migration. First, we determine if there is an improvement in the system performance metrics from applying migration, and we quantify this improvement. We also quantify the impact of the cost of migration (*i.e.*, how much time it takes to move tasks from one set of nodes to another) on system performance. Finally, we compare improvements in system performance that come from better scheduling techniques, backfilling in this case, and improvements that come from better execution infrastructure, as represented by migration. We also show the benefits from combining both enhancements.

The rest of this paper is organized as follows. Section 2 describes the migration algorithm we use. Section 3 presents our evaluation methodology for determining the quantitative impact of migration. In Section 4, we show the results from our evaluation and discuss the implications. Finally, Section 5 presents our conclusions.

2 The Migration Algorithm

Our scheduling strategy is designed for a distributed system, in which each node runs its own operating system image. Therefore, once tasks are started in a node, it is preferable to keep them there. Our basic (nonmigration) gang-scheduling algorithm, both with and without backfilling, works as follows. At every scheduling event (*i.e.*, job arrival or departure) a new scheduling matrix is derived:

– We schedule the already executing jobs such that each job appears in only one row (*i.e.*, into a single virtual machine). Jobs are scheduled on the same set of nodes they were running before. That is, no migration is attempted.
– We *compact* the matrix as much as possible, by scheduling multiple jobs in the same row. Without migration, only nonconflicting jobs can share the same row. Care must be taken in this phase to ensure forward progress. Each job must run at least once during a scheduling cycle.
– We then attempt to schedule as many jobs from the waiting queue as possible, using a FCFS traversal of that queue. If backfilling is enabled, we can look past the first job that cannot be scheduled.
– Finally, we perform an *expansion phase*, in which we attempt to fill empty holes left in the matrix by replicating job execution on a different row (virtual machine). Without migration, this can only be done if the entire set of nodes used by the job is free in that row.

The process of migration embodies moving a job to any row in which there are enough free processors. There are basically two options each time we attempt to migrate a job A from a source row r to a target row p (in either case, row p must have enough nodes free):

– *Option 1*: We migrate the jobs which occupy the nodes of job A at row p, and then we simply replicate job A, in its same set of nodes, in row p.
– *Option 2*: We migrate job A to the set of nodes in row p that are free. The other jobs at row p remain undisturbed.

We can quantify the cost of each of these two options based on the following model. For the distributed system we target, namely the IBM RS/6000 SP, migration can be accomplished with a checkpoint/restart operation. (Although it is possible to take a more efficient approach of directly migrating processes across nodes [1, 10, 11], we choose not to take this route.) Let $S(A)$ be the set of jobs in target row p that overlap with the nodes of job A in source row r. Let C be the total cost of migrating one job, including the checkpoint and restart operations. We consider the case in which (i) checkpoint and restart have the same cost $C/2$, (ii) the cost C is independent of the job size, and (iii) checkpoint and restart are dependent operations (*i.e.*, you have to finish checkpoint before you can restart). During the migration process, nodes participating in the migration cannot make progress in executing a job. We call the total amount of resources (processor $\times$ time) wasted during this process *capacity loss*. The capacity loss for option 1 is

$$(\frac{C}{2} \times |A| + C \times \sum_{J \in S(A)} |J|), \tag{1}$$

where $|A|$ and $|J|$ denote the number of tasks in jobs A and J, respectively. The loss of capacity for option 2 is estimated by

$$(C \times |A| + \frac{C}{2} \times \sum_{J \in S(A)} |J|). \tag{2}$$

The first use of migration is during the compact phase, in which we consider migrating a job when moving it to a different row. The goal is to maximize the number of empty slots in some rows, thus facilitating the scheduling of large jobs. The order of traversal of jobs during compact is from least populated row to most populated row, wherein each row the traversal continues from smallest job (least number of processors) to largest job. During the compact phase, both migration options discussed above are considered, and we choose the one with smaller cost.

We also apply migration during the expansion phase. If we cannot replicate a job in a different row because its set of processors are busy with another job, we attempt to move the blocking job to a different set of processors. A job can appear in multiple rows of the matrix, but it must occupy the same set of processors in all the rows. This rule prevents the ping-pong of jobs. For the expansion phase, jobs are traversed in first-come first-serve order. During expansion phase, only migration option 1 is considered.

As discussed, migration in the IBM RS/6000 SP requires a checkpoint/restart operation. Although all tasks can perform a checkpoint in parallel, resulting in a C that is independent of job size, there is a limit to the capacity and bandwidth that the file system can accept. Therefore we introduce a parameter Q that controls the maximum number of tasks that can be migrated in any time-slice.

3 Methodology

Before we present the results from our studies we first need to describe our methodology. We conduct a simulation based study of our scheduling algorithms using synthetic workloads. The synthetic workloads are generated from stochastic models that fit actual workloads at the ASCI Blue-Pacific system in Lawrence Livermore National Laboratory (a 320-node RS/6000 SP). We first obtain one set of parameters that characterizes a specific workload. We then vary this set of parameters to generate a set of nine different workloads, which impose an increasing load on the system. This approach, described in more detail in [5, 18], allows us to do a sensitivity analysis of the impact of the scheduling algorithms over a wide range of operating points.

Using event driven simulation of the various scheduling strategies, we monitor the following set of parameters: (1) t_i^a: arrival time for job i, (2) t_i^s: start time for job i, (3) t_i^e: execution time for job i (on a dedicated setting), (4) t_i^f: finish time for job i, (5) n_i: number of nodes used by job i. From these we compute: (6) $t_i^r = t_i^f - t_i^a$: response time for job i, (7) $t_i^w = t_i^s - t_i^a$: wait time for job i, and (8) $s_i = \frac{\max(t_i^r, T)}{\max(t_i^e, T)}$: the slowdown for job i, where T is the time-slice for gang-scheduling. To reduce the statistical impact of very short jobs, it is common practice [4] to adopt a minimum execution time. We adopt a minimum of one time slice. That is the reason for the $\max(\cdot, T)$ terms in the definition of slowdown.

To report quality of service figures from a user's perspective we use the average job slowdown and average job wait time. Job slowdown measures how much slower than a dedicated machine the system appears to the users, which is relevant to both interactive and batch jobs. Job wait time measures how long a job takes to start execution and therefore it is an important measure for interactive jobs.

We measure quality of service from the system's perspective with utilization. Utilization is the fraction of total system resources that are actually used for the execution of a workload. It does not include the overhead from migration. Let the system have N nodes and execute m jobs, where job m is the last job to finish execution. Also, let the first job arrive at time $t = 0$. Utilization is then defined as

$$\rho = \frac{\sum_{i=1}^{m} n_i t_i^e}{N \times t_m^f \times \mathrm{MPL}}. \tag{3}$$

For the simulations, we adopt a time slice of $T = 200$ seconds, multiprogramming levels of 2, 3, and 5, and consider four different values of the migration cost C: 0, 10, 20, and 30 seconds. The cost of 0 is useful as a limiting case, and represents what can be accomplished in more tightly coupled single address space systems, for which migration is a very cheap operation. Costs of 10, 20, and 30 seconds represent 5, 10, and 15% of a time slice, respectively.

To determine what are feasible values of the migration cost, we consider the situation that we are likely to encounter in the next generation of large machines, such as the IBM ASCI White. We expect to have nodes with 8 GB of main memory. If the entire node is used to execute two tasks (MPL of 2) that averages to 4 GB/task. Accomplishing a migration cost of 30 seconds requires transferring 4 GB in 15 seconds, resulting in a per node bandwidth of 250 MB/s. This is half the bandwidth of the high-speed switch link in those machines. Another consideration is the amount of storage necessary. To migrate 64 tasks, for example, requires saving 256 GB of task image. Such amount of fast storage is feasible in a parallel file system for machines like ASCI White.

4 Experimental Results

Table 1 summarizes some of the results from migration applied to gang-scheduling and backfilling gang-scheduling. For each of the nine workloads (numbered from 0 to 8) we present achieved utilization (ρ) and average job slowdown (s) for four different scheduling policies: (i) backfilling gang-scheduling without migration (BGS), (ii) backfilling gang-scheduling with migration (BGS+M), (iii) gang-scheduling without migration (GS), and (iv) gang-scheduling with migration (GS+M). We also show the percentage improvement in job slowdown from applying migration to gang-scheduling and backfilling gang-scheduling. Those results are from the best case for each policy: 0 cost and unrestricted number of migrated tasks, with an MPL of 5.

We can see an improvement from the use of migration throughout the range of workloads, for both gang-scheduling and backfilling gang-scheduling. We also note that the improvement is larger for mid-to-high utilizations between 70 and 90%. Improvements for low utilization are less because the system is not fully stressed, and the matrix is relatively empty. Therefore, there are not enough jobs to fill all the time-slices, and expanding without migration is easy. At very high loads, the matrix is already very full and migration accomplishes less than at mid-range utilizations. Improvements for backfilling gang-scheduling are not as impressive as for gang-scheduling. Backfilling gang-scheduling already does a better job of filling holes in the matrix, and therefore the potential benefit from migration is less. With backfilling gang-scheduling the best improvement is 45% at a utilization of 94%, whereas with gang-scheduling we observe benefits as high as 90%, at utilization of 88%.

We note that the maximum utilization with gang-scheduling increases from 85% without migration to 94% with migration. Maximum utilization for backfilling gang-scheduling increases from 95% to 97% with migration. Migration is a mechanism that significantly improves the performance of gang-scheduling without the need for job execution time estimates. However, it is not as effective as backfilling in improving plain gang-scheduling. The combination of backfilling and migration results in the best overall gang-scheduling system.

work load	backfilling gang scheduling					gang scheduling				
	BGS		BGS+M		% s better	GS		GS+M		% s better
	ρ	s	ρ	s		ρ	s	ρ	s	
0	0.55	2.5	0.55	2.4	5.3%	0.55	2.8	0.55	2.5	11.7%
1	0.61	2.8	0.61	2.6	9.3%	0.61	4.4	0.61	2.9	34.5%
2	0.66	3.4	0.66	2.9	15.2%	0.66	6.8	0.66	4.3	37.1%
3	0.72	4.4	0.72	3.4	23.2%	0.72	16.3	0.72	8.0	50.9%
4	0.77	5.7	0.77	4.1	27.7%	0.77	44.1	0.77	12.6	71.3%
5	0.83	9.0	0.83	5.4	40.3%	0.83	172.6	0.83	25.7	85.1%
6	0.88	13.7	0.88	7.6	44.5%	0.84	650.8	0.88	66.7	89.7%
7	0.94	24.5	0.94	13.5	44.7%	0.84	1169.5	0.94	257.9	77.9%
8	0.95	48.7	0.97	42.7	12.3%	0.85	1693.3	0.94	718.6	57.6%

Table 1. Percentage improvements from migration.

Figure 1 shows average job slowdown and average job wait time as a function of the parameter Q, the maximum number of task that can be migrated in any time slice. We consider two representative workloads, 2 and 5, since they define the bounds of the operating range of interest. Beyond workload 5, the system reaches unacceptable slowdowns for gang-scheduling, and below workload 2 there is little benefit from migration. We note that migration can significantly improve the performance of gang-scheduling even with as little as 64 tasks migrated. (Note that the case without migration is represented by the parameter $Q = 0$ for number of migrated tasks.) We also observe a monotonic improvement in slowdown and wait time with the number of migrated tasks, for both gang-scheduling and backfilling gang-scheduling. Even with migration costs as high as 30 seconds, or 15% of the time slice, we still observe benefit from migration. Most of the benefit of migration is accomplished at $Q = 64$ migrated tasks, and we choose that value for further comparisons. Finally, we note that the behaviors of wait time and slowdown follow approximately the same trends. Thus, for the next analysis we focus on slowdown.

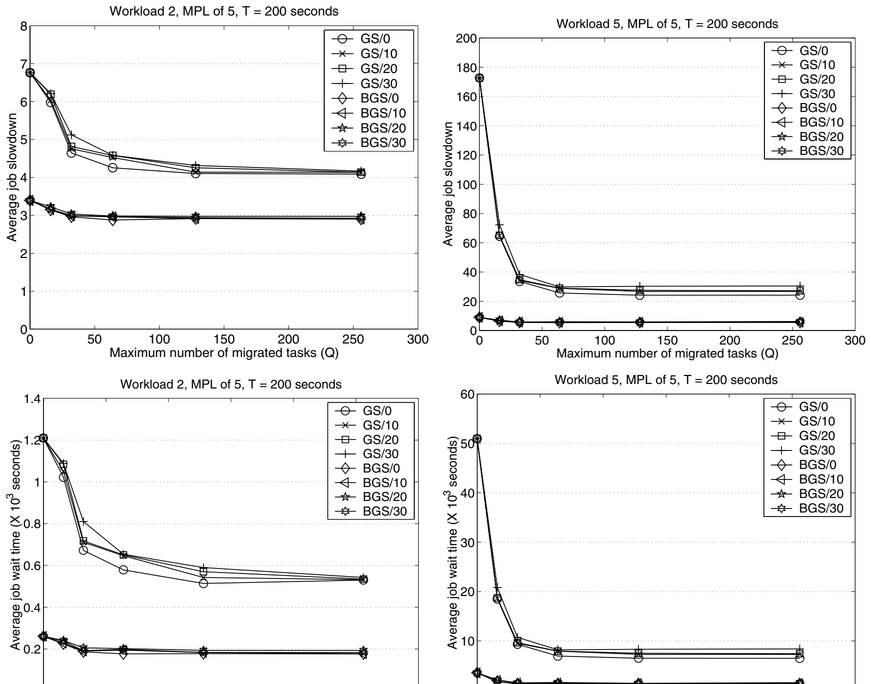

Fig. 1. Slowdown and wait time as a function of number of migrated tasks. Each line is for a combination of scheduling policy and migration cost.

Figure 2 shows average job slowdown as a function of utilization for gang-scheduling and backfilling gang-scheduling with different multiprogramming levels. The upper left plot is for the case with no migration ($Q = 0$), while the other plots are for a maximum of 64 migrated tasks ($Q = 64$), and three different migration costs, $C = 0$, $C = 20$, and $C = 30$ seconds, corresponding to 0, 10, and 15% of time slice, respectively. We observe that, in agreement with Figure 1, the benefits from migration are essentially invariant with the cost in the range we considered (from 0 to 15% of the time slice).

From a user perspective, it is important to determine the maximum utilization that still leads to an acceptable average job slowdown (we adopt $s \leq 20$ as an acceptable value). Migration can improve the maximum utilization of gang-scheduling by approximately 8%. (From 61% to 68% for MPL 2, from 67% to 74% for MPL 3, and from 73% to 81% for MPL 5.) For backfilling gang-scheduling, migration improves the maximum acceptable utilization from from 91% to 95%, independent of the multiprogramming level.

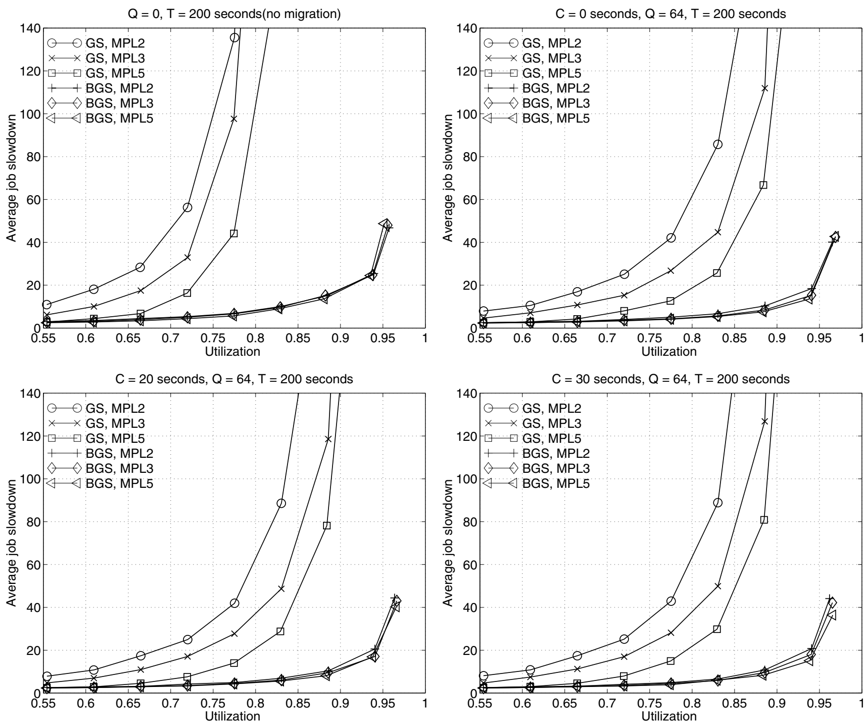

Fig. 2. Slowdown as function of utilization. Each line is for a combination of scheduling policy and multiprogramming level.

5 Conclusions

In this paper we have evaluated the impact of migration as an additional feature in job scheduling mechanisms for distributed systems. Typical job scheduling for distributed systems uses a static assignment of tasks to nodes. With migration we have the additional ability to move some or all tasks of a job to different nodes during execution of the job. This flexibility facilitates filling holes in the schedule that would otherwise remain empty. The mechanism for migration we consider is checkpoint/restart, in which tasks have to be first vacated from one set of nodes and then reinstantiated in the target set.

Our results show that there is a definite benefit from migration, for both gang-scheduling and backfilling gang-scheduling. Migration can lead to higher acceptable utilizations and to smaller slowdowns and wait times for a fixed utilization. The benefit is essentially invariant with the cost of migration for the range considered (0 to 15% of a time-slice). Gang-scheduling benefits more than backfilling gang-scheduling, as the latter already does a more efficient job of filling holes in the schedule. Although we do not observe much improvement from a system perspective with backfilling scheduling (the maximum utilization does not change much), the user parameters for slowdown and wait time with a given utilization can be up to 45% better. For both gang-scheduling and backfilling gang-scheduling, the benefit is larger in the mid-to-high range of utilization, as there is not much opportunity for improvements at either the low end (not enough jobs) or very high end (not enough holes). Migration can lead to a better scheduling without the need for job execution time estimates, but by itself it is not as useful as backfilling. Migration shows the best results when combined with backfilling.

References

[1] J. Casas, D. L. Clark, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole. **MPVM: A Migration Transparent Version of PVM**. *Usenix Computing Systems*, 8(2):171–216, 1995.

[2] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. **A world-wide flock of Condors: Load sharing among workstation clusters**. *Future Generation Computer Systems*, 12(1):53–65, May 1996.

[3] 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.

[4] 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.

[5] H. Franke, J. Jann, J. E. Moreira, and P. Pattnaik. **An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific**. In *Proceedings of SC99, Portland, OR*, November 1999. IBM Research Report RC21559.

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

[7] H. D. Karatza. **A Simulation-Based Performance Analysis of Gang Scheduling in a Distributed System**. In *Proceedings 32nd Annual Simulation Symposium*, pages 26–33, San Diego, CA, April 11-15 1999.

[8] 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.

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

[10] S. Petri and H. Langendörfer. **Load Balancing and Fault Tolerance in Workstation Clusters – Migrating Groups of Communicating Processes**. *Operating Systems Review*, 29(4):25–36, October 1995.

[11] J. Pruyne and M. Livny. **Managing Checkpoints for Parallel Programs**. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, IPPS'96 Workshop*, volume 1162 of *Lecture Notes in Computer Science*, pages 140–154. Springer, April 1996.

[12] 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.

[13] J. Skovira, W. Chan, H. Zhou, and D. Lifka. **The EASY-LoadLeveler API project**. In *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47. Springer-Verlag, April 1996.

[14] W. Smith, V. Taylor, and I. Foster. **Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance**. In *Proceedings of the 5th Annual Workshop on Job Scheduling Strategies for Parallel Processing*, April 1999. In conjunction with IPPS/SPDP'99, Condado Plaza Hotel & Casino, San Juan, Puerto Rico.

[15] K. Suzaki and D. Walsh. **Implementation of the Combination of Time Sharing and Space Sharing on AP/Linux**. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.

[16] C. Z. Xu and F. C. M. Lau. **Load Balancing in Parallel Computers: Theory and Practice**. Kluwer Academic Publishers, Boston, MA, 1996.

[17] K. K. Yue and D. J. Lilja. **Comparing Processor Allocation Strategies in Multiprogrammed Shared-Memory Multiprocessors**. *Journal of Parallel and Distributed Computing*, 49(2):245–258, March 1998.

[18] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramanian. **Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques**. In *Proceedings of IPDPS 2000*, Cancun, Mexico, May 2000.