RC 21359 (96204) 09DEC98 Computer Science/Mathematics

Research Report

A Gang-Scheduling System for ASCI Blue-Pacific

J. E. Moreira, H.Franke, W. Chan, L. L. Fong
International Business Machines Corporation
Armonk, NY 10504

M. A. Jette, A. Yoo
Lawrence Livermore National Laboratory
Livermore, CA 94550

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

This page intentionally left blank.

A Gang-Scheduling System for ASCI Blue-Pacific

José E. Moreira[1], Hubertus Franke[1], Waiman Chan[1], Liana L. Fong[1],
Morris A. Jette[2], and Andy Yoo[2]

[1] International Business Machines Corporation
Armonk, NY 10504, USA
{jmoreira,frankeh,waimanc,llfong}@us.ibm.com
[2] Lawrence Livermore National Laboratory
Livermore, CA 94550, USA
{jette,ayoo}@llnl.gov

Abstract. The ASCI Blue-Pacific machines are large parallel systems comprised of thousands of processors. We are currently developing and testing a gang-scheduling job control system for these machines that exploits space- and time-sharing in the presence of dedicated communication devices. Our initial experience with this system indicates that, though applications pay a small overhead, overall system performance as measured by queue time improves significantly. This gang-scheduling system is planned for deployment into production mode during 1999 at Lawrence Livermore National Laboratory.

1 Introduction

The Accelerated Strategic Computing Initiative (ASCI) program is creating a new state-of-the-art in large scale computations. Through this program, three national laboratories in the United States (Lawrence Livermore, Los Alamos, and Sandia) have each received machines capable of Teraflops performance. Combined with new simulation techniques these machines will help guarantee the safety of the United States nuclear stockpile while preventing the need for actual tests. In addition, machines in the ASCI program will be available to University research in a variety of areas.

Effective and efficient job scheduling in such large environments, which can easily have hundreds of simultaneous users, represents a new challenge: How to provide service for production codes that execute for tens of hours on hundreds of processors while at the same time catering to the needs of numerous scientists debugging and developing new codes? In this paper we describe a new job scheduling system that we are developing for the ASCI Blue-Pacific machines, in the Lawrence Livermore National Laboratory (LLNL). These machines are based on the RS/6000 SP, but they have their own peculiarities. Our system is called GangLL and it supports both time- and space-sharing of parallel jobs using gang-scheduling. We have already developed and deployed the first generation of GangLL. Although it is currently running only on a small test system, we will show some encouraging results, both from actual runs as well as simulations. It is important to acknowledge that the development of GangLL has been motivated by excellent performance results obtained by LLNL with gang-scheduling in other large systems [4, 7].

The rest of this paper is organized as follows. Section 2 gives some details of the ASCI Blue-Pacific machines, and describes the gang-scheduling approach to space- and time-sharing. Section 3 describes the architecture and organization of GangLL. Section 4 presents the results from our experiments on an actual machine and simulations. Finally, Section 5 presents our conclusions and discusses some 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 TB3MX 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 access to University research through the ASCI Strategic Alliance Program. In this program the *Combined Technology Refresh* (CTR) machine provides 320 nodes of similar characteristic as the SST nodes, organized in a single sector. Features of both machines are summarized in Table 1.

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

characteristic	SST	CTR
Peak speed	3.9 teraOPS	892 gigaOPS
Memory	2.6 terabytes	504 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 (TB3 switch, bidirectional)	150 MB/s	150 MB/s
Total number of processors	5,856	1,344
Processor-to-memory bandwidth	2.1 TB/s	2.1 TB/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

Despite the size of the ASCI Blue-Pacific machine, we know from LLNL's experience [4, 9] with large systems that the demand for processors to execute jobs will at times be larger than the total number of processors available. The traditional solution to this problem in a space-sharing environment is simply 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 scheduling techniques, such as backfilling [10]. These scheduling technique cannot, however, handle the also important issue of *preemption*: Some times it is desirable to suspend the execution of a lower priority job so that a high-priority job can run.

A more flexible solution to the problem of executing a large number of jobs is to share the machine's resources not only spatially but also temporally [3, 4, 6, 8, 12, 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.

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 [11] (*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 that allows us to guarantee that tasks of the same job execute simultaneously despite the fact that the operating system images are not 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 [11], where the columns correspond to processors and the rows correspond to revolving time slices.

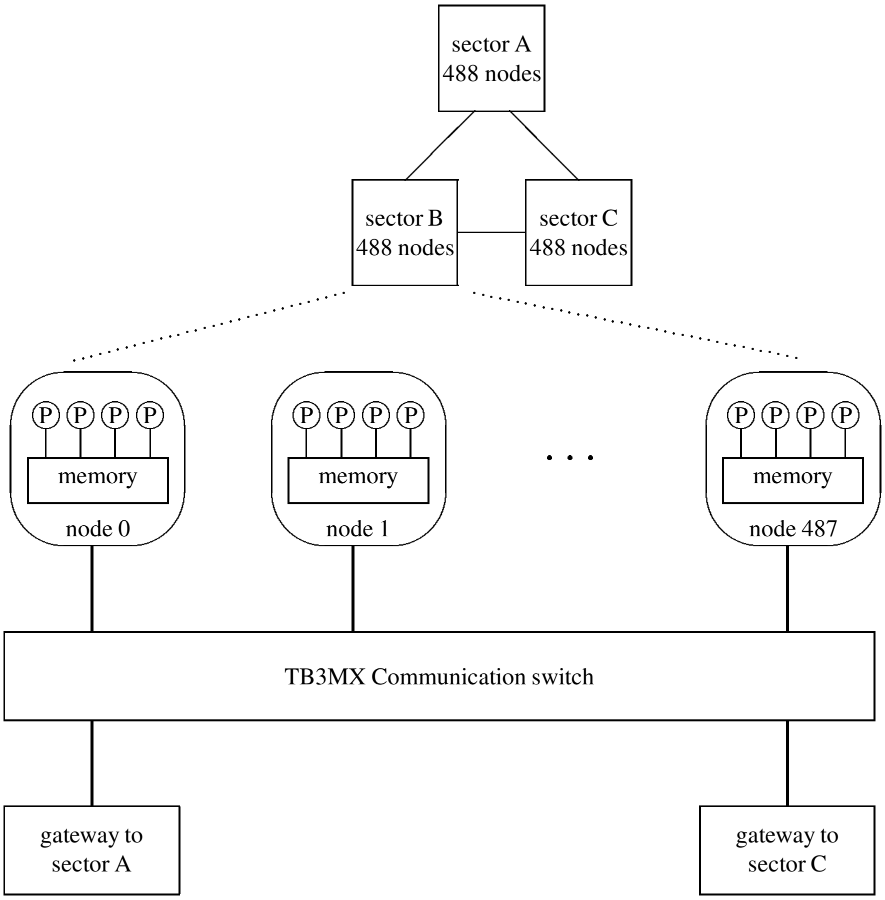

Fig. 1. High-level organization of the ASCI Blue-Pacific machine.

In the next section we describe the organization of GangLL, which is designed to provide the job execution environment described here for the ASCI Blue-Pacific machines. At first, we are designing GangLL to operate on each sector independently. Section 3.3 discusses some issues related to multi-sector operation.

3 Architecture and Organization of GangLL

We build our GangLL research prototype system as an extension to IBM LoadLeveler. The high-level organization of GangLL is shown in Figure 2. We follow a hierarchical scheduling scheme. A *Central Manager* performs resource allocation and overall job scheduling. Using a hierarchical distribution scheme, schedules are distributed to local autonomous node-level schedulers, which are responsible for implementing the columns of global schedules that pertain to their designated node.

3.1 Central Manager

The *Central Manager* is responsible for receiving all job submissions from users and it enqueues the jobs into a central job queue. The *Central Manager* then selects jobs from the queue for execution and performs spatial and temporal scheduling. The schedule is represented by an Ousterhout matrix that precisely identifies the assignment of tasks to processors both in space and time. The matrix is then distributed to the individual node-level schedulers. Note, that each node-level scheduler only needs the columns of the matrix that correspond to the processors in its node. It is also the responsibility of the *Central Manager* to interact with a site-specific external scheduler. This external scheduler can perform some of the scheduling decisions, thus implementing policies that are specific to a particular site.

Jobs are ordered in the queue according to their priority. A job's priority (also known as *sysprio*) is computed by the central manager based on rules specified by the system administrator. Each job has a set of attributes: submission time, owner, processor requirement, memory requirement, estimated execution time, etc. The system administrator defines a formula that computes the priority of a job as a function of those attributes. For example, the following formula would give higher priority to older jobs that need more processors:

$$sysprio : NumProcs * 1000000 - QDate \tag{1}$$

where *NumProcs* is the number of processors requested and *QDate* is the time stamp (seconds since epoch) of the job submission. Higher values of *sysprio* correspond to higher priorities.

After the Ousterhout matrix is built, it has to be distributed from the *Central Manager* to the corresponding node-level schedulers. Although the matrix itself is relatively small (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*.

The *Central Manager*, instead of contacting each node-level scheduler individually, uses a binary-tree scheme to distribute the matrix. It sends the whole matrix to one node-level scheduler, which extracts its own columns and forwards the remaining columns to two other node-level schedulers. This operation continues recursively until all node-level schedulers have received their columns. With this approach, we accomplish the distribution of the matrix in $O(\log n)$ time, as opposed to the $O(n)$ in the naive approach.

The distribution scheme is two-phase: in the first phase a new Ousterhout matrix is distributed to all node-level schedulers. Only after all node-level schedulers have received this new matrix the new matrix is committed, replacing the old one. If the distribution fails, the matrix is not committed, and the nodes continue to use the old one. The replacement actually happens at a specified wall-clock time, thus guaranteeing that all node-level schedulers start using the new matrix at the same time.

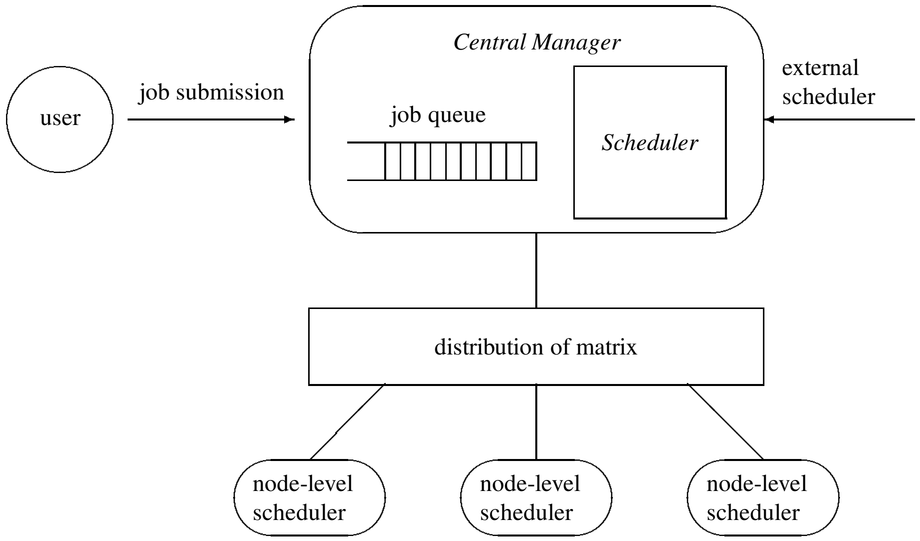

Fig. 2. General organization of the GangLL prototype.

3.2 Node-Level Scheduler

Each node in the system has a node-level scheduler, responsible for implementing the schedule as defined by the columns of the matrix that correspond to the processors in that node. For the node-level scheduler 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 node-level schedulers 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 node-level scheduler has to perform the context-switch for all the tasks running in its node. There are two components of a context-switch: (i) context switching of the processors and (ii) context switching of the communication device. Context switching of the processors is achieved through suspending and resuming execution of user processes. The context-switch of the communication device is necessary because on the RS/6000 SP applications have user-level direct access to the communication device. These connections are a scarce resource: In the ASCI Blue-Pacific machine, each node can have four connections, one for each processor.

The node-level scheduler interacts with two other system services to accomplish proper context switching of tasks: (i) the communication subsystem (CSS) and (ii) the process tracking service (PTS).

Communication Device Control: In the RS/6000 SP communication protocols such as MPI write data packets to a non-pageable communication buffer and activate a DMA engine (on the TB3MX adapter) to transfer the packets to the network. Due to the memory requirements of the communication buffer as well as of those in the adapter, there is a limited number of user-level processes that can be granted direct connection to the communication device at any given time. Each connection is called a *window*. Thus, before suspending a user-level process, we force it to release its window, which then can be reassigned in the next time-slice to another process. Similarly, before resuming a user-level process, access to a particular window must be explicitly granted by the node-level scheduler.

The interaction between kernel-level communication subsystem (CSS) and user-level application is through a *call-back* mechanism: CSS sends an event to the application requesting it to release its window in a timely fashion. This allows the user-level communication protocol to handle disconnections at well defined points. As a result, costly synchronization mechanism (e.g., semaphores, locks) between application and kernel can be avoided, thus reducing the cost of communication operations.

Process Tracking: A new task always starts as a single process created by the node-level scheduler itself. This process can then create new children processes and so on. This set of descendant processes is not always a tree, since intermediate processes may have terminated. The relationship between processes and tasks is defined by *process genealogy*: all processes that descend from the initial, or root, process are in the same task [5]. This concept is not provided in UNIX. Individual processes can explicitly cut their relationship to *groups* and *sessions* [13]. In addition, a process becomes orphaned when its parent terminates, thus severing the creation chain. Hence, the resulting forest of processes cannot be adequately identified by the initial process alone.

To solve these deficiencies, we have implemented a new kernel extension that utilizes process-change events (in particular, process creation and process termination) to maintain proper genealogy of each task. When the node-level scheduler creates the first process of a new task, that task and its process are registered with the kernel extension. After that, the kernel extension maintains information on all the processes that constitute a task. The kernel extension is also responsible for suspending and resuming task execution atomically. Hence, there is no risk of a process creation/termination event being lost. Task suspension and resumption is accomplished from the kernel extension through standard UNIX signal mechanisms, using SIGSTOP to suspend execution and SIGCONT to resume it.

3.3 External Interfaces

Large supercomputing centers tend to have their own job scheduling systems, either to handle site specific conditions or to organize multiple machines into a larger, metacomputing system. The goal is usually to increase system throughput and/or decrease response time for the particular workload in those centers.

To accomplish some level of control by the site scheduler, we introduce the concept of an *execution factor* associated with each job. Let e_i be the execution factor of job J_i. Let $J_1, J_2, \ldots, J_n$ be the set of jobs sharing a particular processor p. Job J_i will receive a fraction of the non-idle processor time of p equal to $e_i / \sum_{j=1}^{n} e_j$. That is, the fraction of processor time that each job receives is proportional to its execution factor.

By manipulating the execution factors of jobs, a site scheduler can perform its own job control, albeit at a coarser granularity. By setting $e_i = 1$ and $e_j = 0, j = 1, \ldots, n, j \neq i$, the site scheduler enables job J_i for execution while suspending all other jobs. Note this approach can be used even across sectors with a single *meta-job* represented as three different jobs, one in each sector.

4 Experiments

There is a cost associated with time-sharing, due mostly to: (i) the cost of the context-switches themselves, (ii) additional memory pressure created by multiple jobs sharing nodes, and (iii) additional disk space pressure caused by more jobs executing concurrently. For that reason, the degree of time-sharing is usually limited by a parameter that we call, in analogy to uniprocessor systems, the multiprogramming level (MPL). A time-sharing system with multiprogramming level of 1 reverts back to a space-sharing system.

At the time of this writing, GangLL is running on a 4-node, 16-processor development system at LLNL. Using this small testbed, and also a simulator for a larger system, we have performed experiments to assess the efficacy of our gang scheduling approach. First, we measure how GangLL impacts the performance of parallel applications. Then, we analyze the performance characteristics of the entire system under GangLL. Finally, we study the negative impact of paging, when the memory footprint of the active jobs exceeds the physical memory size of the nodes.

4.1 Application performance under GangLL

To measure application performance under GangLL, we use four different benchmarks from the NAS Parallel Benchmark suite, version 2.3 [1]. Each benchmark is written in Fortran with calls to the MPI message-passing library. The benchmarks are explicitly compiled to run on 16 tasks, thus using all the processors in the development machine. Each task consists of exactly one process. The name, a brief description, and the memory footprint (per task) of each benchmark are listed in Table 2.

Table 2. Four NAS parallel benchmarks used in our experiments.

benchmark	description	footprint (MBytes)
BT, class A	CFD block tridiagonal solver	22.1
CG, class B	conjugate gradient	34.4
LU, class B	CFD LU solver	14.7
MG, class B	multigrid	29.4

We first run each benchmark on a dedicated machine (multiprogramming level of 1) and measure its execution time. We then run two instances of the same benchmark, each requiring 16 processors, which time-share the machine through our gang-scheduling system (multiprogramming level of 2) with a fine-grained time-slice of 5 seconds. Results from these experiments are shown in Figure 3. The number on top of the bars are the actual execution times in seconds.

In an ideal case, running two instances of a benchmark should slow their execution by a factor of two. We note that the execution overhead created by the time-sharing is between 0 and 10% of this ideal execution time. Overall, our gang-scheduling system does an effective job of providing the illusion of a slower virtual machine for the execution of each job.

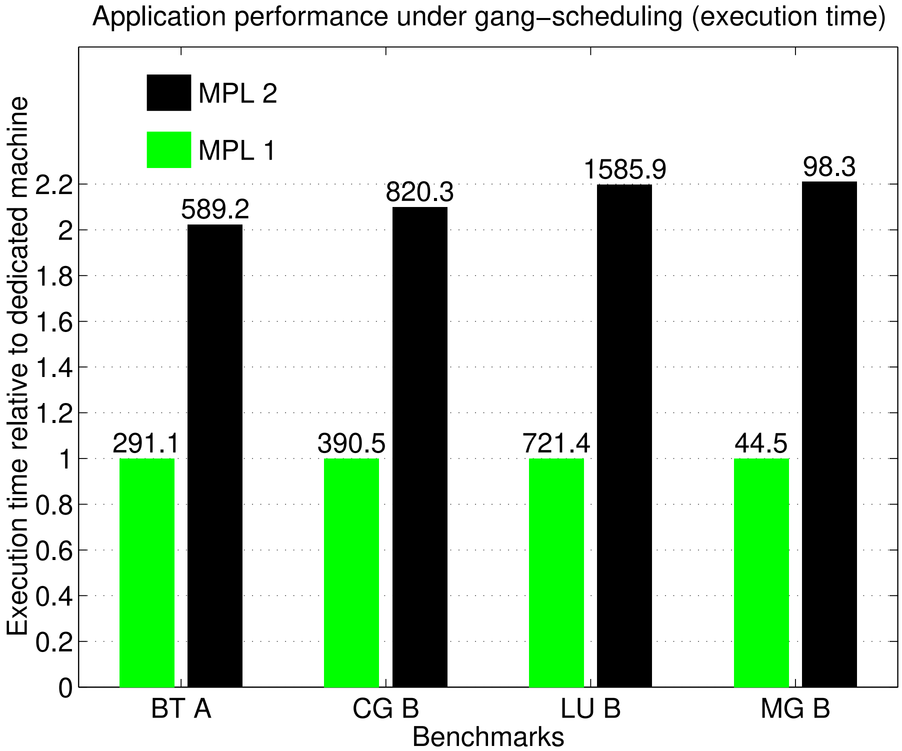

Fig. 3. Application performance under gang-scheduling for four different NAS benchmarks.

4.2 System performance

The goal of this section is to quantify the performance benefits delivered by gang-scheduling in a large multiprocessor system. We examine the behavior of the average job *queue time* in the presence of varying degrees of multiprogramming. The job queue time is defined as the elapsed time between job submission and the time the job starts execution. We show this performance parameter for different multiprogramming levels.

We use a week-long trace from a production 256-processor system. The trace logs submission time and actual processor time for over 1000 jobs. We simulate the execution of these jobs on a 256-processor system under the control of GangLL. Jobs are scheduled on a FCFS basis. All processors are homogeneous: any arbitrary set of processors (of the appropriate size) can be combined in a pool to run a parallel job, and the performance of any such pool is the same. This is typical of RS/6000 SP systems with homogeneous nodes. We consider two different values of an *overhead* parameter: 0% and 5%. An overhead of 5% means that during 5% of the time slice no work is accomplished (due to both context-switch time and paging time).

The results are shown in Figure 4, for multiprogramming levels from 1 to 5. Analyzing first the case of zero overhead, the average time a job waits in the queue goes from almost a day to a few minutes when the multiprogramming level goes from 1 to 5. When the overhead climbs to 5% we still observe substantial reduction in average job queue time. The positive impact of gang-scheduling on FCFS scheduling has also been reported in [12].

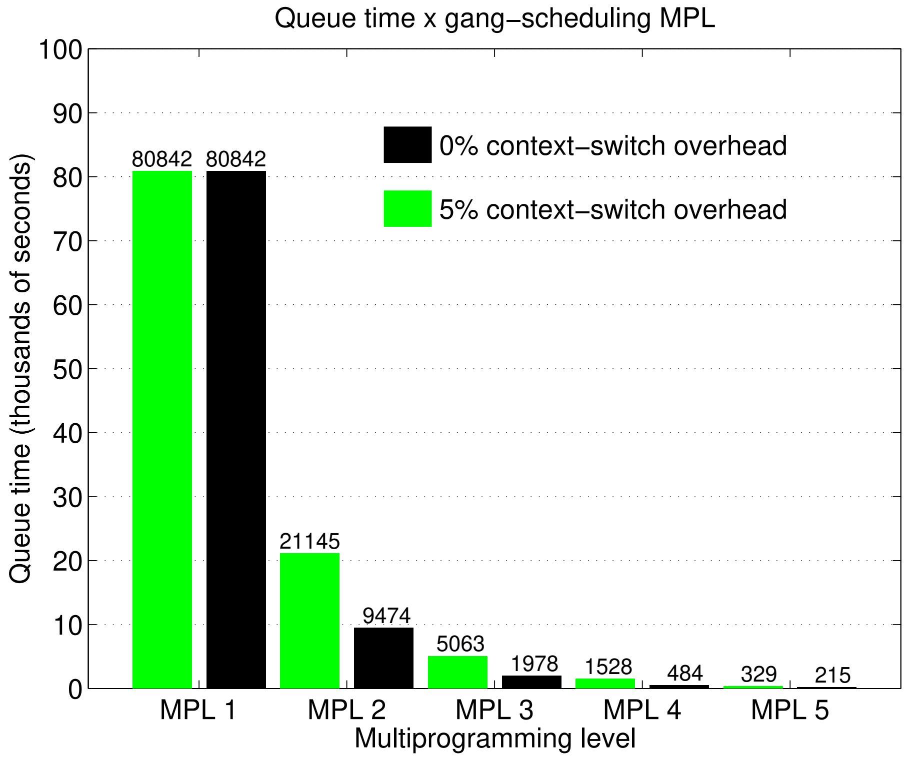

Fig. 4. Reduction in average job queue time through gang-scheduling.

4.3 Impact of paging

Paging effects can significantly hurt performance if jobs have large memory footprint. Figure 5 shows the behavior of 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. Strategies to reduce the impact of paging on gang-scheduling are discussed in [14].

5 Conclusion and Future Work

We have designed and implemented the first generation of GangLL, a gang scheduling system for the RS/6000 SP intended for use in the ASCI-Blue Pacific machine. Initial results form actual runs and simulation show definite benefits. We still have some performance work to do as we would like to reduce

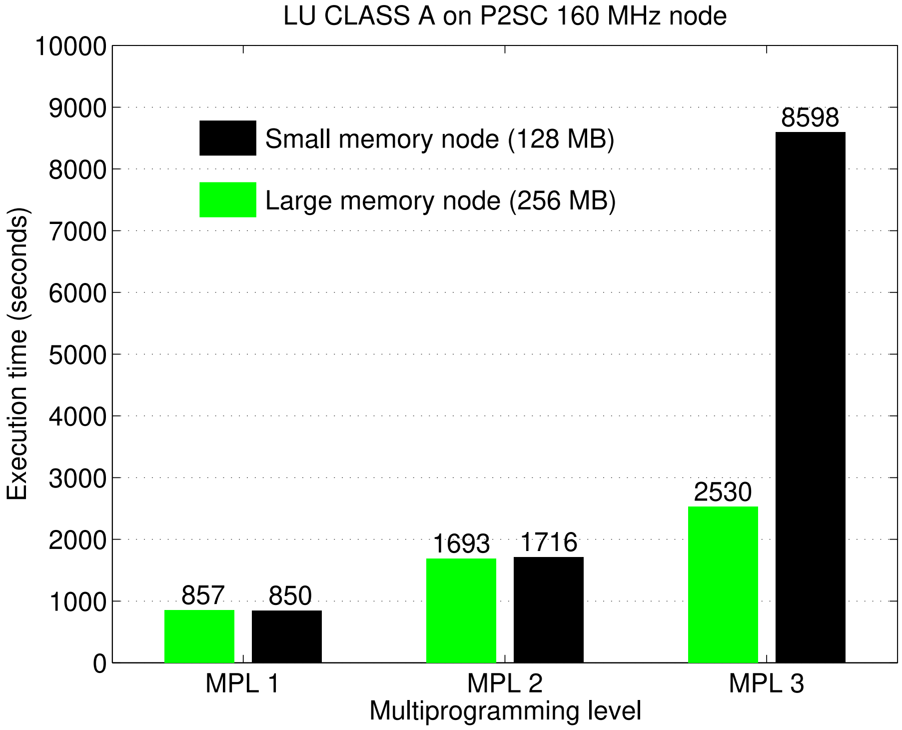

Fig. 5. The impact of paging on application performance under gang-scheduling.

scheduling overhead for parallel applications to only 5% of a 5-second time slice. (This is probably the shortest time-slice we will use.) We already have plans to test the system, which today is only running on a 4-node machine, on 32 and 128 nodes. We also want to investigate the use of *flocking* [2] for inter-sector operation. We expect to introduce the system to production on large ASCI machines during 1999. Previous LLNL experience with gang scheduling indicates that GangLL will be a major factor in improving usability and utilization of these systems.

References

1. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. **The NAS Parallel Benchmarks 2.0**. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.
2. D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. **A worldwide flock of Condors: Load sharing among workstation clusters**. *Future Generation Computer Systems*, 12(1):53–65, May 1996.
3. D. G. Feitelson. **A Survey of Scheduling in Multiprogrammed Parallel Systems**. Technical Report RC 19790 (87657), IBM T. J. Watson Research Center, October 1994.
4. 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.
5. H. Franke, J. E. Moreira, and P. Pattnaik. **Process Tracking for Parallel Job Control**. Technical Report 21290, IBM Research Division, September 1998. Submitted for publication.
6. 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.
7. B. Gorda and R. Wolski. **Time Sharing Massively Parallel Machines**. In *International Conference on Parallel Processing*, volume II, pages 214–217, August 1995.
8. 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.
9. M. Jette, D. Storch, and E. Yim. **Timesharing the Cray T3D**. *Cray User Group*, pages 247–252, March 1996.
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. K. Ousterhout. **Scheduling Techniques for Concurrent Systems**. In *Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.
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. U. Vahalia. **UNIX Internals: The New Frontiers**. Prentice-Hall, Inc, 1996.
14. F. Wang, M. Papaefthymiou, and M. Squillante. **Performance Evaluation of Gang Scheduling for Parallel and Distributed Multiprogramming**. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 277–298. Springer-Verlag, April 1997.
15. B. S. Yoo and C. R. Das. **Processor Management Techniques for Mesh-Connected Multiprocessors**. In *Proceedings of the International Conference on Parallel Processing (ICPP'95)*, volume 2, pages 105–112, August 1995.