RC 21262 (94656) 19MAY98 Computer Science/Mathematics

Research Report

An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments

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

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

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

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center [Publications 16-220 ykt] P. O. Box 218, Yorktown Heights, NY 10598. email: reports@us.ibm.com
Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home

This page intentionally left blank.

An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments

J. E. Moreira W. Chan L. L. Fong H. Franke

International Business Machines Corporation
Armonk, NY 10504
{ *jmoreira,waimanc,llfong,frankeh* }@*us.ibm.com*

M. A. Jette

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

Abstract

Recent Terascale computing environments, such as those in the Department of Energy Accelerated Strategic Computing Initiative, present a new challenge to job scheduling and execution systems. The traditional way to concurrently execute multiple jobs in such large machines is through space-sharing: each job is given dedicated use of a pool of processors.

Previous work in this area has demonstrated the benefits of sharing the parallel machine's resources not only spatially but also temporally. Time-sharing creates virtual processors for the execution of jobs. The scheduling is typically performed cyclically and each time-slice of the cycle can be considered an independent virtual machine. When all tasks of a parallel job are scheduled to run on the same time-slice (same virtual machine), gang-scheduling is accomplished. Research has shown that gang-scheduling can greatly improve system utilization and job response time in large parallel systems.

We are developing GangLL, a research prototype system for performing gang-scheduling on the ASCI Blue-Pacific machine, an IBM RS/6000 SP to be installed at Lawrence Livermore National Laboratory. This machine consists of several hundred nodes, interconnected by a high-speed communication switch. GangLL is organized as a centralized scheduler that performs global decision-making, and a local daemon in each node that controls job execution according to those decisions.

The centralized scheduler builds an Ousterhout matrix that precisely defines the temporal and spatial allocation of tasks in the system. Once the matrix is built, it is distributed to each of the local daemons using a scalable hierarchical distributions scheme. A two-phase commit is used in the distribution scheme to guarantee that all local daemons have consistent information. The local daemons enforce the schedule dedicated by the Ousterhout matrix in their corresponding nodes. This requires suspending and resuming execution of tasks and multiplexing access to the communication switch.

Large supercomputing centers tend to have their own job scheduling systems, to handle site specific conditions. Therefore, we are designing GangLL so that it can interact with an external site scheduler. The goal is to let the site scheduler control spatial allocation of jobs, if so desired, and to decide when jobs run. GangLL then performs the detailed temporal allocation and controls the actual execution of jobs. The site scheduler can control the fraction of a shared processor that a job receives through an execution factor parameter.

To quantify the benefits of our gang-scheduling system to job execution in a large parallel system, we simulate the system with a realistic workload. We measure performance parameters under various degrees of time-sharing, characterized by the multiprogramming level. Our results show that higher multiprogramming levels lead to higher system utilization and lower job response times. We also report some results from the initial deployment of GangLL on a small multiprocessor system.

1 Introduction

We are witnessing an emergence of Terascale computing environments: those with processing power in the order of 1 Teraflops and physical memory size in the order of 1 Terabyte. The most prominent example of such environments is the Department of Energy (DOE) Accelerated Strategic Computing Initiative (ASCI) project. The ASCI Red machine currently stands in the first place of the TOP500 list [1] with an achieved Linpack performance of 1.3 Teraflops. The ASCI Blue and ASCI White machines, currently under development, will deliver peak performance in the order of 3 and 10 Teraflops, respectively. All these systems achieve this level of performance through massive parallelism, typically thousands of processor.

Parallel machines of this size present a new challenge to job scheduling and execution systems. In this paper we focus on a research prototype environment for job scheduling and execution that we are developing for the ASCI Blue-Pacific machine, an IBM RS/6000 SP to be installed at Lawrence Livermore National Laboratory (LLNL). This machine consists of a large set (several hundred) of nodes, interconnected by a high-speed communication switch. Each node is a shared-memory multiprocessor running its own independent image of the AIX operating system. This organization is shown in Figure 1.

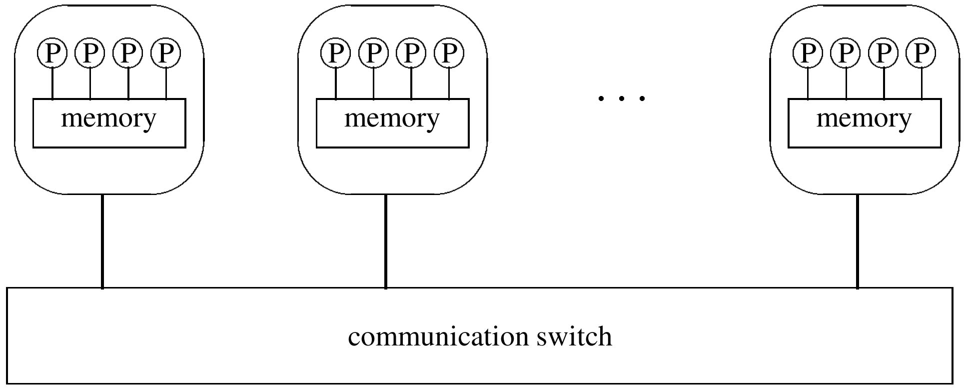

Figure 1: High-level organization of the ASCI Blue-Pacific machine.

The rest of this paper is organized as follows. In Section 2 we describe our goals in terms of job execution for ASCI Blue-Pacific. In summary, we want to to accomplish space-sharing as well as time-sharing of computing resources. In Section 3 we discuss the organization of GangLL, the system we are developing to accomplish these job execution goals. In Section 4 we explain the support that we are building into GangLL so that it can cooperate with site-specific schedulers. These schedulers are common in large supercomputing centers. In Section 5 we evaluate the performance of GangLL, both through simulation as well as direct measurements on our first deployed version. Finally, we discuss related work in Section 6 and present our conclusions in Section 7.

2 Job execution on ASCI Blue-Pacific

The most natural way to concurrently execute multiple jobs in such large machines, with thousands of processors, is *space-sharing*. In this mode, each job is given dedicated use of a pool of processors. This is shown in Figure 2 for a small system of 8 processors. In that figure, P_j represents a particular processor, and J_i^k represents task k of Job J_i. Job J_1 runs on a four-processor pool while jobs J_2 and J_3 run on two two-processor pools. Space-sharing partitions a large machine into smaller, job-dedicated machines. Many parallel programs are written with this dedicated-machine concept in mind. It guarantees that messages and synchronizations can be used between tasks of the parallel programs without the risk of blocking for a task that is not running. In the particular case of the IBM RS/6000 SP, the dedicated mode also allows tasks to use an optimized connection to the communication switch.

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
J_1^0	J_1^1	J_1^2	J_1^3	J_2^0	J_2^1	J_3^0	J_3^1

Figure 2: Space-sharing gives to each job an independent pool of processors.

Despite the size of the ASCI Blue-Pacific machine, we know, from LLNL's experience [3, 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. We know from the literature that this can be highly detrimental to system and job performance: system utilization is low and job response time is high. The problem can be alleviated in part by clever scheduling techniques, such as backfilling [12].

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 [2, 3, 6, 8, 16]. This *time-sharing* is so natural on uniprocessor systems that most of the time we do not even think about it. Time-sharing an 8-processor system with a multiprogramming level of four is shown in Figure 3. (In this paper we always consider time-sharing combined with space-sharing.) The figure shows the Ousterhout matrix [13] that defines the tasks executing on each processor and each time-slice. The matrix is cyclic in that time-slice 3 is followed by time-slice 0. Each row of the matrix defines an 8-processor virtual machine, which runs at $1/4$th of the speed of the physical machine. (We are considering the case of equal time-slices and ignoring the cost of context-switches.) We use these four virtual machines to run two 8-way parallel jobs and several smaller jobs. Note that some tasks can utilize multiple processors (such as J_3^0 and J_3^1) and that some jobs can appear in multiple rows (such as jobs J_4 and J_5).

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
time-slice 0	J_1^0	J_1^1	J_1^2	J_1^3	J_1^4	J_1^5	J_1^6	J_1^7
time-slice 1	J_2^0	J_2^1	J_2^2	J_2^3	J_2^4	J_2^5	J_2^6	J_2^7
time-slice 2	J_3^0	J_3^0	J_3^1	J_3^1	J_4^0	J_4^1	J_5^0	J_5^1
time-slice 3	J_6^0	J_6^1	J_6^2	J_6^3	J_4^0	J_4^1	J_5^0	J_5^1

Figure 3: The Ousterhout matrix defines spatial and time allocation.

Through time-sharing we can create virtual processors for the execution of jobs. Creating one more virtual machine for the execution of a new 8-way job in the case of Figure 3 requires, in principle, only adding one more row to the Ousterhout matrix. Obviously, things are not so simple. 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. A time-sharing system with multiprogramming level of 1 reverts back to a space-sharing system.

In building the Ousterhout matrix of Figure 3 we schedule all tasks of a job to run in the same time-slice.

This is necessary to give each job the impression that it is still running on a dedicated, albeit slower, machine. This type of coscheduling of tasks of a parallel job is often called *gang-scheduling* [2]. Furthermore, in the case of the RS/6000 SP, we also have to give to each job the optimized connection to the communication switch that jobs in dedicated physical machines have. This requires a context-switch of this connection in much the same way that processors are context-switched [6].

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 machine.

3 Organization of GangLL

We build our research prototype system as an extension to the IBM product LoadLeveler, and we call it GangLL. The high-level organization of GangLL is shown in Figure 4. The user submits a job to the *Central Manager*. 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 we expect that to change as programs that make use of both MPI and multithreading start to be more and more common. Submitted jobs are placed in the queue of jobs and are scheduled for execution by the *Scheduler* module of the *Central Manager*.

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 rebuilt 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. 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. A new task always starts as a single process created by *Startd* itself. This process can then create new children processes and so on. The task is defined as the forest of processes that descend from that first process. Note that this set of descendant processes is not always a tree, since intermediate processes may have terminated. To properly perform its context-switches, *Startd* has to keep track of this forest of processes that constitute a task. This is implemented in GangLL through AIX kernel extensions that track each process creation and termination. When *Startd* creates the first process of a new task, that task and its process are registered with the kernel extensions. After that, the kernel extensions maintain information on all the processes that constitute a task. The kernel extensions are also responsible for suspending and resuming task execution atomically. Hence, there is no risk of a process creation/termination event being lost. Due to the behavior of signal handlers and AIX process scheduling, the kernel extensions perform a top-down traversal of the forest when suspending tasks. When resuming tasks, the traversal is bottom-up.

The last topic to discuss is the distribution of the columns of the Ousterhout matrix from the *Central Manager* to the corresponding *Startd* daemons. Although the matrix itself is not too big (a few kilobytes),

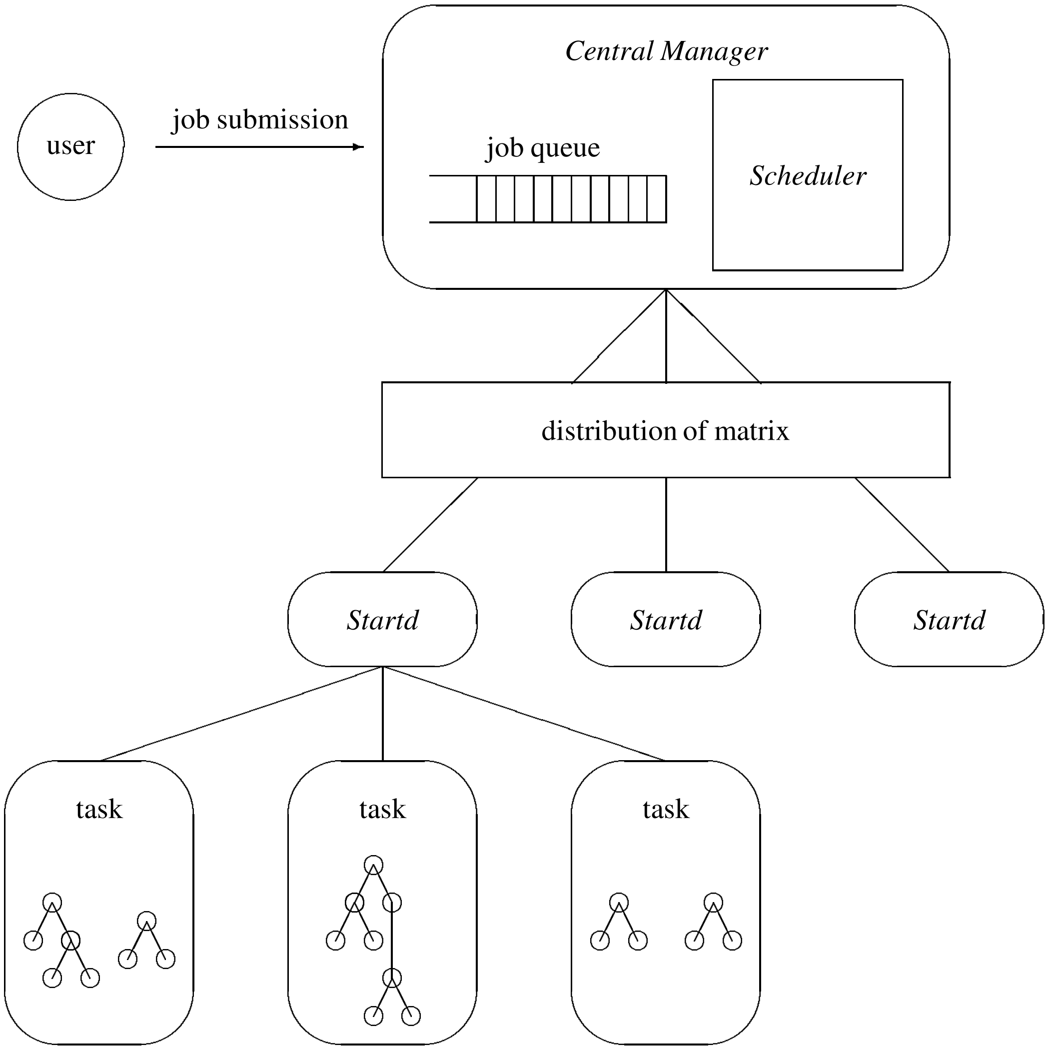

Figure 4: General organization of the GangLL prototype.

it has to be distributed efficiently to hundreds of node. 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 hierarchical nature of the scheme is illustrated in Figure 5. The *Central Manager*, instead of contacting each *Startd* daemon individually, sends the entire matrix to one *Startd* (in this case, *Startd* 0). That *Startd* keeps its corresponding columns and then splits the remaining matrix into subparts, that it forwards to other *Startd*s. These *Startd*s recursively continue the distribution. In particular, we are working with the binary tree format shown in Figure 5. The boxes in that figure represent the columns of the matrix that are sent in each message. Even though the total number of bytes transferred in this scheme is more than if the *Central Manager* contacted each *Startd* directly, there is no bottleneck and the distribution itself is accomplished much faster.

The distribution scheme is two-phase: in the first phase a new Ousterhout matrix is distributed to all *Startd* daemons. Only after all *Startd*s have received this new matrix (the distribution itself may fail) the new matrix is committed, replacing the old one. The replacement actually happens at a specified wall-clock

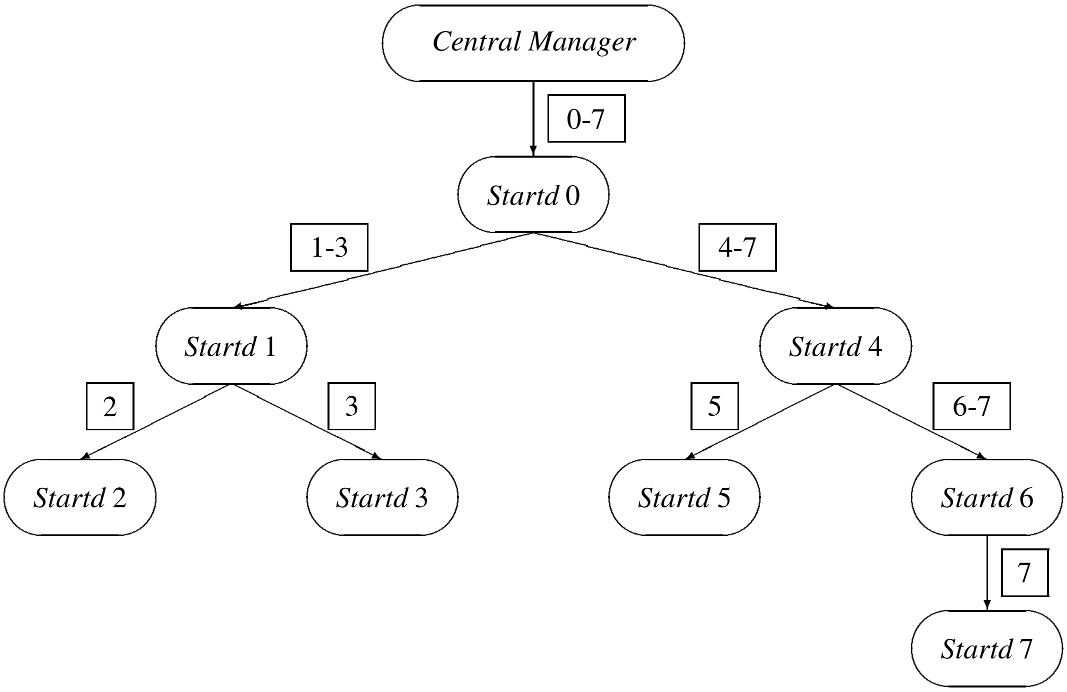

Figure 5: Hierarchical distribution of the Ousterhout matrix.

time, thus guaranteeing that all *Startd*s start using the new matrix at the same time.

4 Interface with site schedulers

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. Examples of large centers that develop and use their own job scheduling systems include LLNL, Cornell Theory Center, Maui Supercomputing Center, Argonne National Laboratory, and Pacific Northwest National Laboratory.

We are designing an API for GangLL that lets site schedulers select jobs for execution and perform spatial allocation. GangLL then takes care of the more detailed time scheduling (building the Ousterhout matrix) and actual execution of the jobs according to the schedule. The approach is analogous to that of the EASY-LoadLeveler API project [15], designed for external control of space-sharing.

It is desirable to give the site scheduler a means to control the distribution of processor time among jobs. For instance, when time-sharing a set of processors between two jobs, it may be important to give a larger fraction of the time to one of the jobs. 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 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.

Each job has a default execution factor, given by its class. (System administrators define a finite set of job classes for the system. One of the parameters that define a job is its class.) The site scheduler can change the execution factor of a job dynamically during its execution, using another API function. The GangLL *Scheduler* recomputes the Ousterhout matrix to satisfy the new execution factor. Note that the execution factors of multiple competing jobs are always self-consistent. This is different, for example, from specifying percentages, since there is no risk of overcommitting a processor. We can always compute an Ousterhout matrix that satisfies the execution factor parameters for any set of jobs, although it may be necessary to have empty (idle) entries in the matrix sometimes. Strict adherence to the execution factor parameters may optionally be sacrificed to minimize system idle time.

The external interface of GangLL allows a site scheduler to control job execution by specifying a time-space-utilization surface for each job. A job J_i executes between times t_i^s and t_i^f, defined by the site scheduler using the GangLL API. The set of processors in which a job executes can also be defined through the API. (Optionally, the site scheduler may let the GangLL *Scheduler* select the processors.) Finally, during the execution of a job, its utilization of processors can be controlled by the execution factor parameter.

5 Evaluation

The benefit of time-sharing (in the form of gang-scheduling) for parallel systems has been extensively quantified in the literature [3, 5, 6, 14, 18]. In this section we assess that benefit in our particular system. We divide the section into three parts. First, we present a qualitative discussion of the impacts of distribution of the Ousterhout matrix and context switch overheads on system performance. Then, we show through simulation that our system can indeed provide significant improvements in system performance when overheads for gang-scheduling are small relative to the time slice chosen. Finally, we report results from experiments with our first deployed prototype. These results confirm that, in certain situations, the overhead for gang-scheduling is indeed small even for time slices of a few seconds.

5.1 Sources of overhead in GangLL

Whenever a new job is to start execution or an existing job terminates execution, GangLL has to respond by computing and distributing a new Ousterhout matrix. Note that these events are unrelated to the operation of actually implementing the schedule in each node. This node-level scheduling is a purely local operation of each *Startd* daemon. The time it takes to update schedule information in the nodes does not affect the context switch time between tasks. It does affect, however, how fast nodes can change their schedules. A new schedule is necessary to start executing a new job, or to reclaim a time-slice that was assigned to a terminated job. The overhead of distributing the Ousterhout matrix prevents highly parallel jobs from experiencing the same level of responsiveness available on uniprocessor systems.

We described in Section 3 the two-phase, hierarchical distribution scheme used to update the schedules in each node. For a system of N nodes, this scheme has a critical path of $4\log_2 N$ transmitted messages (2 in each phase: a command and an acknowledgement). Transmitting a message of a few kilobytes between nodes takes in the order of a few milliseconds (1-20 ms), depending on the particularities of the system. Therefore, even for a system with a thousand nodes, a complete distribution of a new Ousterhout matrix can be accomplished in less than a second. Data from actual workloads at LLNL (see Section 5.2) indicates that job arrival and termination events occur, on average, once every few hundred seconds. Furthermore, except for very large jobs, only a subset of the machine is affected by a new job that starts execution or a job that terminates.

The time that it takes to context switch between two tasks directly influences the utilization of each processor and the choice of time slice [17]. During context switch, no useful work is performed by the tasks. Therefore, it is desirable that the context switch time be a small fraction of the time slice. On the other hand, making the time slice too long reduces responsiveness of the system, and an appropriate compromise has to be reached. Ultimately, the choice of time slice depends on the workload characteristics of the system.

Previous work on gang-scheduling for the IBM RS/6000 SP [6] has shown that the context switch time for parallel programs in this system is approximately 30 ms. Therefore, a time-slice of 3 seconds is enough to reduce this overhead to only 1%. However, there is another hidden cost in context switch. Unless the system has enough resources to hold all active jobs in main memory, at least some of the address space of a task will be paged out to disk while that task is suspended. Those pages will have to be brought back to main memory when the task resumes execution. Because paging is a relatively slow operation, the task can remain idle for long periods during its time slice. In this case, the time slice has to be adjusted properly. It has to be made long enough so that even the paging overhead is not significant. We show some of the negative effects of paging on performance in Section 5.3.

5.2 Analysis of gang-scheduling in a large system

The goal of this section is to quantify the performance benefit delivered by gang-scheduling in a large multiprocessor system. We examine the behavior of three performance parameters in the presence of varying degrees of time-sharing:

1. *utilization*: This is a measure of how well we can utilize the machine by keeping its processor busy with work. A good execution environment leaves few idle cycles and therefore achieves high machine utilization. This parameter is reported as a percentage of total processor-time available, and it is always between 0 and 100%.

2. *slowdown*: This is a measure of machine performance as perceived by the users. For each job, its response time is the elapsed time between job submission and the time the job completes execution. Its processor time is the time it takes to execute on a dedicated system. We compute the system slowdown as the ratio of two sums: the sum of all job response times and the sum of all job processor times. Note, that this is different from the average job slowdown [3].

3. *queue time*: This is defined as the elapsed time between job submission and the time the job starts execution. The smaller this time, the better the impression of a dedicated, though slower, machine. This parameter is reported in absolute seconds, averaged for all jobs.

We tabulate this performance parameters for different multiprogramming levels. A system with multiprogramming level of one corresponds to a pure space-sharing system, while a multiprogramming level of ∞ corresponds to an ideal time-sharing system.

Since we are currently in the development/early-deployment stage of GangLL, we report some results obtained through trace-driven simulation. 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. Figure 6 shows some characteristics of this workload: (a) the number of jobs of each size, and (b) the number of jobs with processor time less than or equal to a certain value.

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 four different values of an *overhead* parameter: 0%, 1%, 5%, and 10%. An overhead of $x\%$ means that during $x\%$ of the time slice no work is accomplished (due to both context-switch time and paging time).

The results are shown in Figure 7, for multiprogramming levels from 1 to 8 and infinity. Analyzing first the case of zero overhead, we observe an increase in system utilization with the multiprogramming level.

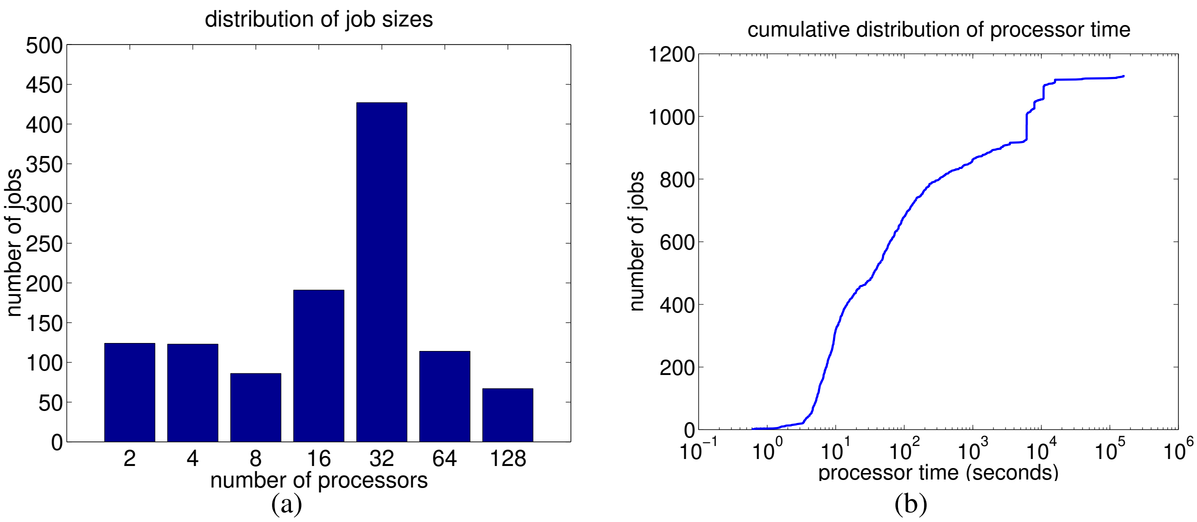

Figure 6: Workload characteristics: (a) distribution of job sizes, (b) distribution of processor times.

The improvement is modest and saturates easily, from 82% with a multiprogramming level of one to 86% with a multiprogramming level of three or more. The improvements in slowdown and queue time with the multiprogramming level are more significant. Even small multiprogramming levels significantly reduce the values of these parameters. 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 4. With infinite multiprogramming, the average time a job waits in the queue drops to zero, as expected. (The maximum multiprogramming level actually necessary in that case was 10.)

overhead	parameter	multiprogramming level								
		1	2	3	4	5	6	7	8	∞
0 %	utilization (%)	82	85	86	86	86	86	86	86	86
	slowdown	26.7	4.7	2.7	2.3	2.3	2.2	2.2	2.2	2.2
	queue time (s)	80842	9474	1978	484	215	23	5	1	0
1 %	utilization (%)	82	84	85	85	85	85	85	85	85
	slowdown	26.7	5.3	2.8	2.4	2.3	2.3	2.3	2.3	2.3
	queue time (s)	80842	11121	2387	462	110	23	6	1	0
5 %	utilization (%)	82	81	82	82	81	82	82	82	81
	slowdown	26.7	8.6	3.9	3.0	2.7	2.7	2.7	2.7	2.7
	queue time (s)	80842	21145	5063	1528	329	76	15	4	0
10 %	utilization (%)	82	78	79	79	78	78	78	78	78
	slowdown	26.7	13.2	5.2	3.7	3.3	3.2	3.2	3.2	3.2
	queue time (s)	80842	35528	8121	2306	592	136	42	16	0

Figure 7: Performance results for systems with time-sharing.

For an overhead value of 1%, results are still very good. This represents what can be accomplished with a time slice of a few seconds and enough man memory to hold all jobs resident. Note that most of the benefit is achieved for multiprogramming levels of 2 or 3. When the overhead climbs to 5% or 10%, the situation starts to deteriorate. There is no improvement in utilization from time-sharing with an overhead of 5%, although we still see benefits in the other performance parameters. Finally, at an overhead of 10% we run into a highly undesirable situation: the utilization of the system starts to drop with time-sharing. To keep overheads low ($\leq 5\%$) in the presence of paging, the time slice has to be increased appropriately. Increasing the time slice has the bad side effect of reducing the system responsiveness. This hurts, in particular, small jobs [17].

5.3 Experimental results

We now report some experimental results that allow some insight into the context switch overhead and paging effects under GangLL. In our experiments, we measure the execution time for a benchmark under different levels of multiprogramming. For a given multiprogramming level l, we run l instances of the benchmark concurrently. The various instances are time-multiplexed by GangLL. Our experiments are performed on two different nodes. Both have a single 160 MHz P2SC processor, but one has 256 MB of main memory while the other has only 128 MB.

Figure 8 shows results for runs of the NAS benchmark BT, class W, which has a memory footprint of 17 MB when compiled for the P2SC. Results for the 128 MB node are shown in Figure 8(a) while for the 256 MB node are in Figure 8(b). For each multiprogramming level l, the marker represents the average execution time for all l jobs, while the vertical bar represents the range of execution times observed for those jobs. The solid line in each plot represents the *ideal* execution time under time-sharing. The ideal execution time for a multiprogramming level l is l times the execution for a single job in a dedicated system ($l = 1$).

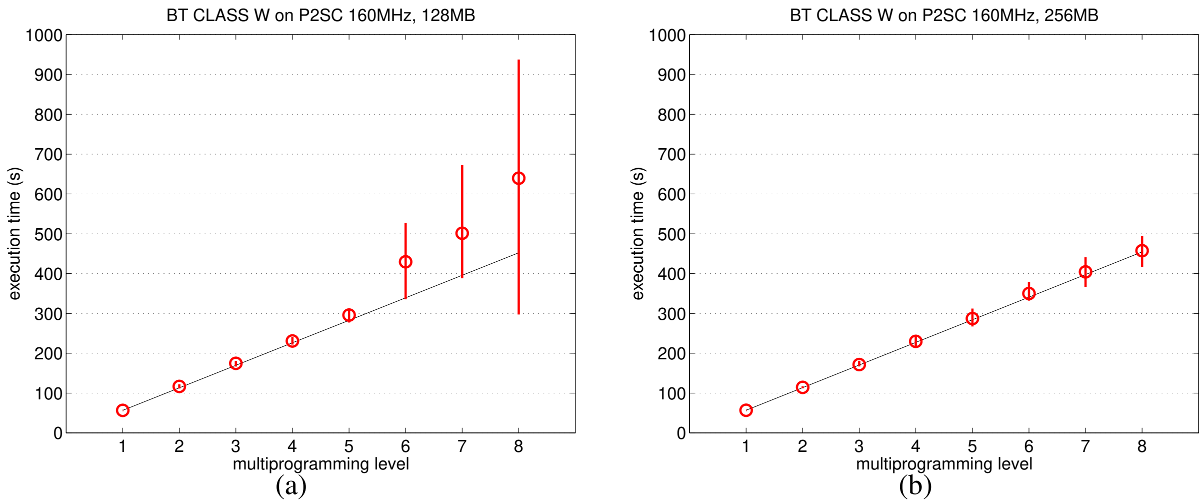

Figure 8: NAS benchmark BT class W performance under gang-scheduling: (a) node with 128 MB of memory, (b) node with 256 MB of memory. A time slice of 5 seconds was used.

For multiprogramming levels up to 5 (85 MB of job footprint), the behavior is close to ideal in both nodes. This indicates small context switch overhead and low paging activity. For multiprogramming levels of 6 and greater, paging activity starts to affect the behavior of jobs in the small memory (Figure 8(a)) node. (With a multiprogramming level of 8, the footprint of the jobs alone is 136 MB. In addition, there are several system daemons active.) The average execution time for all jobs increases, reflecting the time to access paged out regions of the address space. Also, the variance between execution times for the various jobs at a given multiprogramming level increases. Some jobs are more favored than others by the paging algorithms in the operating system. The node with 256 MB of memory (Figure 8(b)), large enough to comfortably hold 8 jobs, continues to exhibit behavior close to ideal, even at high multiprogramming levels.

Paging effects can also hurt performance at low multiprogramming levels, if the jobs have large memory footprint. Figure 9 shows the behavior of NAS benchmark LU class A under control of GangLL. Each instance of this job has a 45 MB memory footprint. Again, the solid line represents ideal time-sharing behavior. In both the 128 MB node (Figure 9(a)) and the 256 MB node (Figure 9(b)), time-sharing 2 instances of LU class A (multiprogramming level of 2) behaves close to the ideal case. 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 expected in the 256 MB node under the same conditions.

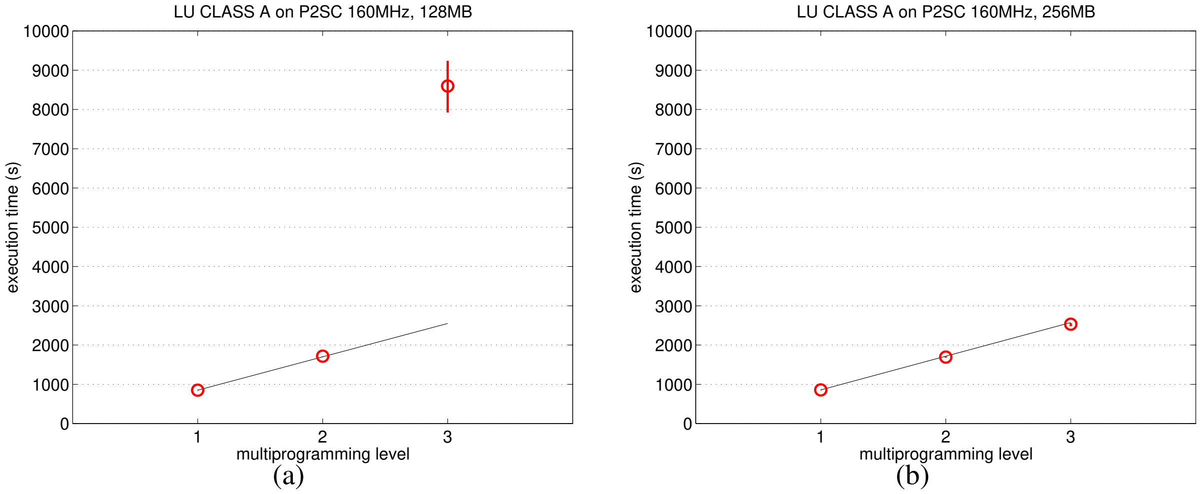

Figure 9: NAS benchmark LU class A performance under gang-scheduling: (a) node with 128 MB of memory, (b) node with 256 MB of memory. A time slice of 50 seconds was used.

GangLL is going through its first stages of deployment at LLNL. Figure 10 shows results from measurements in a 16-processor system. The system consists of 4 nodes, each node a 4-way SMP with 332 MHz PowerPC 604e processors. (However, only three of the nodes were available for benchmarking at the time of our experiments.) The benchmark we use for measurements in this system is a 9-way parallel NAS benchmark BT class A. Each task of this benchmark has a memory footprint of 35 MB, and three tasks of a job run in each of three nodes. Paging is not an issue in our experiments, since the nodes have 512 MB of memory each and we only use multiprogramming level up to 3. (For a maximum of 9 active tasks in any given node.) In the plot, the ideal time-sharing behavior is indicated by the solid line. With a time slice of 5 seconds, we observe a behavior very close to the ideal case. This shows that gang-scheduling can work efficiently across machines even with a short time slice, provided that the jobs fit in physical memory.

6 Related work

The concept of coscheduling (gang-scheduling) tasks for multiprocessor systems was introduced in [13]. The performance advantages of gang-scheduling as a job scheduling mechanism for large systems has been demonstrated in practice before [3]. In fact, it was the previous experience of LLNL with gang-scheduling [3, 7] that motivated us to develop this research prototype for the ASCI Blue-Pacific machine. The gang-scheduling system for a 256-processor Cray T3D described in [3] led to utilization improvements from 33%, with the original space-sharing system, to 96%. We do not expect to see such large improvements in the ASCI Blue-Pacific machine, since space-sharing on the RS/6000 SP is more flexible to begin with, thus leading to larger utilizations.

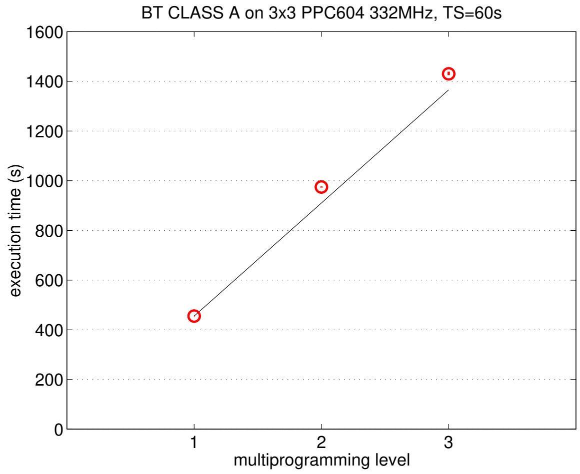

Figure 10: NAS benchmark BT class A performance under gang-scheduling. 9-way parallel job on 3 nodes with 4 processors and 512 MB of memory each. A time slice of 5 seconds was used.

Our time-sharing mechanism for GangLL differs from that reported in [3] in that it keeps all jobs resident in the operating system images of the participating nodes. The system described in [3] actually saves the entire state of a job to disk, at the end of a time slice. It then restores from disk the entire state of the next job. Maintaining the jobs alive in the operating system allows for much smaller time slices (and thus response time) and less context switch overhead, *provided that the jobs fit in main memory*. When job images overflow the main memory, saving and restoring the entire job state to and from disk in one operation typically performs much better than paging. Saving and restoring the entire state of jobs to and from disk also provide the means for the scheduler to relocate jobs during their execution. We note that LLNL is currently developing other gang-scheduling systems that also keep jobs resident [10].

A straightforward implementation of gang-scheduling requires both global coordination and centralized scheduling. GangLL avoids global coordination by having each node implement its schedule independently. Coordination is achieved implicitly, through synchronized wall-clocks. However, GangLL still relies on centralized scheduling. Alternatively, approaches based on Distributed Hierarchical Control (DHC) [4, 6, 17] can avoid both centralized scheduling and global coordination. DHC creates multiple virtual machines, each dedicated to the execution of jobs of a different size. Scheduling decisions can be made locally and coordination can be achieved the same way as GangLL. The disadvantage of DHC over a centralized scheduler approach such as GangLL is that it can lead to fragmentation of processor pools, which are only available in certain sizes (*e.g.*, powers of two).

The interactions between gang-scheduling and I/O-intensive jobs are discussed in [11]. That paper shows that the CPU can remain severely underutilized during time-slices dedicated to I/O-intensive tasks. The same happens in GangLL when the footprints of the active jobs become too large for the physical memory. Processors stay idle for much of the time-slice, waiting for pages to be brought back from disk. The solution discussed in [11], in the form of a *relaxed* gang-scheduling, for efficient support of I/O operations should be considered for GangLL.

Finally, another solution to the paging overhead problem we observed in our experiments is *block paging*, as discussed in [18]. In block paging, the system identifies sets of pages that tend to be referenced together and pages each of these sets in and out of memory as a unit. When the set size is one, block paging degenerates to demand paging, currently used in our environment.

7 Conclusion

It is shown throughout the literature that time-sharing, particularly in the form of gang-scheduling, significantly improves performance characteristics of parallel systems. Our own simulations, using actual workloads, have shown the larger multiprogramming levels can significantly decrease wait time and slowdown while increasing system utilization.

We have described a gang-scheduling environment for a very large scale distributed system, such as the ASCI Blue-Pacific machine. The organization of this environment is decentralized: the global decision making is performed by a single scheduling module, while enforcement of those decisions in job execution are performed locally. The distribution of decisions from the central scheduler to the local daemons is done through a hierarchical distribution scheme, using a two-phase commit for reliability.

The local daemons encapsulate all issues of controlling task execution on each node. Some of these issues, such as task suspension and resumption, can be handled through standard UNIX facilities. Other issues are specific to the RS/6000 SP, such as the multiplexing of the communication switch, and have to be handled accordingly.

We have deployed our first version of GangLL in a small multiprocessor system at LLNL. Initial measurements in this system indicate efficient behavior of gang-scheduling when paging is not an issue. Our future work includes enhancing GangLL so that it incorporates better scheduling and job control techniques to handle less favorable conditions, and operates well in larger systems.

References

[1] J. Dongarra, H.-W. Meuer, and E. Strohmaier. **TOP500 Supercomputer Sites**. Available in Web page at: `http://www.netlib.org/benchmark/top500.html`.

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

[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 L. Rudolph. **Distributed Hierarchical Control for Parallel Processing**. *IEEE Computer*, May 1990.

[5] D. G. Feitelson and L. Rudolph. **Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control**. *Journal of Parallel and Distributed Computing*, 35:18–34, 1996.

[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] M. A. Jette. **Expanding Symetric Multiprocessor Capability Through Gang Scheduling**. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.

[11] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. **Implications of I/O for Gang Scheduled Workloads**. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 215–237. Springer-Verlag, April 1997.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] F. Wang, H. Franke, M. Papaefthymiou, P. Pattnaik, L. Rudolph, and M. Squillante. **A Gang Scheduling Design for Multiprogrammed Parallel Computing Environments**. In *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, April 1996.

[18] 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.