Gang Scheduling with Lightweight User-Level Communication*

Eitan Frachtenberg,⋆ Fabrizio Petrini,⋆ Salvador Coll,⋆ and Wu-chun Feng†
⋆ CCS-3 Modeling, Algorithms, and Informatics Group
† CCS-1 Advanced Computing
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory

{eitanf,fabrizio,scoll,feng}@lanl.gov

Abstract

In this paper, we explore the performance of gang scheduling on a cluster using the Quadrics interconnection network. In such a cluster, the scheduler can take advantage of this network's unique capabilities, including a network interface card-based processor and memory and efficient user-level communication libraries. We developed a micro-benchmark to test the scheduler's performance under various aspects of parallel job workloads: memory usage, bandwidth and latency-bound communication, number of processes, timeslice quantum, and multiprogramming levels. Our experiments show that the gang scheduler performs relatively well under most workload conditions, is largely insensitive to the number of concurrent jobs in the system and scales almost linearly with number of nodes. On the other hand, the scheduler is very sensitive to the timeslice quantum, and values under 30 seconds can incur large overheads and fairness problems.

***Keywords**: Gang Scheduling, Performance Evaluation, Parallel Architectures, Quadrics interconnect.*

1 Introduction

Gang scheduling is an efficient way to multiprogram frequently communicating processes on parallel supercomputers [1, 11]. Similar to time-sharing in uniprocessor systems, gang scheduling offers many advantages for job and system efficiency. The scheduler's ability to preempt jobs allows higher-utilization of the system in the following ways:

- The system's responsiveness interactive and high-priority jobs can be very high, even when the system is being heavily used.
- Jobs requiring a large number of processors do not have to wait for the completion of other jobs before being launched. Furthermore, once such a job is running, it does not monopolize resources; other jobs can still be executed.
- Unused resources can be used by low-priority jobs and reallocated to higher-priority jobs when needed.
- The system can maintain a high utilization rate under varying workloads.

On the other hand, it has been argued that gang scheduling can incur a relatively high overhead because of the effect the context switch on the computing nodes.

This overhead is caused by resource sharing between multiple jobs and spans several dimensions. One dimension is the cache memory and translation look-aside buffer (TLB): context switch between processes causes a *cold start* which initiates the loading of the new process's working set and the eviction of the old working set.

The overflow of the physical memory into the virtual memory usually causes a severe performance penalty. In fact, the access time of a page swapped to disk can be orders of magnitude slower than the access time of the same page in the main memory.

Another important dimension is the interface between the processing node and the network. The advent of high-performance network interface cards (NICs) that include processors and memory enable user-level messaging protocols, that minimize the communication delay removing the operating system from the communication protocols [6, 7, 8, 18, 19]. These protocols require dedicated buffers in the network interface that are usually mapped in the virtual address space of the user processes. They may also require communication buffers that must be "pinned" in the main memory to perform the inter-node communication. With gang scheduling, all these buffers must be properly managed between context switches. Finally, the context switch be-

*The work was supported by the U.S. Department of Energy through Los Alamos National Laboratory contract W-7405-ENG-36.

tween jobs must take care of the packets in transit inside the network.

In the SHARE scheduler of the IBM SP2[4], the communication buffers are saved and restored at every context switch to minimize the amount of pinned memory. Processing nodes achieve coordination with synchronized clocks. The nodes do not interact through explicit synchronization and are not coordinated by a central controller. In particular, the network is not flushed during a context switch; therefore, a node may receive a packet that is addressed to a process that is no longer running. To address this problem, the CM-5[10], ParPar[3], and SCore-D[5] propose network flushing mechanisms.

In the CM-5, during a context switch, all packets are "dropped down" to the closest node in the fat-tree network and stored temporarily. When the job is rescheduled, these packets are reinjected to complete their trip.

In the ParPar scheduler, the master daemon coordinates the context switch by sending synchronization messages to the worker nodes. Upon receiving the message, a partner daemon in the worker node suspends the running process and schedules the new process. In order to flush the network, each network interface broadcasts a halt message to all other network interfaces. Given that the communication queues in the network interface are managed in first in-first out order and the network delivers packets using a single deterministic path between each pair of nodes, this protocol guarantees that no packets belonging to the previous timeslice will be received after the halt message. Ref. [2] shows that the buffers in the network interface are often under utilized during the context switch, so the context switch overhead can be reduced by saving only the packets that are in the buffer rather than in the whole set of buffers.

The SCore-D cluster does not need to send special control messages because each single packet is explicitly ACKed or NACKed by the destination. Each node simply stops transmitting and waits until all its outstanding packets are acknowledged.

In this paper we analyze the overhead associated with the gang scheduler of the Quadrics network (QsNET).[1] The QsNET is of particular importance to the Los Alamos National Laboratory because it is to be used as the interconnect for the 30-teraops ASCI-Q machine by Compaq.[2] This paper's contribution lies in a systematic study of various properties of the Quadrics gang scheduler that can be used to compare it to other schedulers in terms of performance, overhead, and scalability.

The rest of this paper is organized as follows. We first describe the features of a Quadrics-based cluster in Section 2. In Section 3, we provide the details of the experimental methodology. Section 4 presents our experimental results and analysis. Finally, we conclude and outline future work in Section 5.

[1] http://www.quadrics.com

[2] http://www.compaq.com/hpc

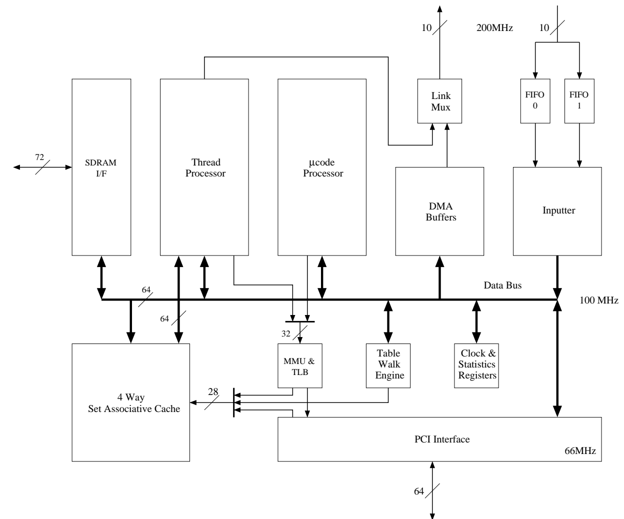

Figure 1. Elan functional units

2 Overview of QsNET and RMS

2.1 Hardware

QsNET [12] consists of two building blocks: a programmable network interface called Elan [13] and a low-latency, high-bandwidth communication switch called Elite [14]. Elites can be interconnected in a fat-tree topology [9]. The network has several layers of communication libraries that provide trade-offs between performance and ease of use. Other important features are hardware support for collective communication patterns and fault tolerance.

The Elan network interface links the high-performance, multi stage Quadrics network to a processing node containing one or more processing elements (PEs). In addition to generating and accepting packets to and from the network, the Elan provides substantial local processing power as well as 64 MB of SDRAM to implement high-level message-passing protocols such as MPI[17]. Messages are chunked by the DMA engine in packets of 320 bytes which are delivered in-order. The internal functional structure of the Elan is shown in Figure 1.

The Elan supports four independent microcode threads: (1) Inputter thread; handles input transactions from the network. (2) Direct memory access (DMA) thread; generates DMA packets to be written to the network, prioritizes outstanding DMAs, and time-slices large DMAs so that small DMAs are not adversely blocked. (3) The processor-scheduling thread prioritizes and controls the scheduling and descheduling of the thread processor. (4) The command-processor thread handles operations requested by the host (i.e., "command") processor at user level.

Processes in a parallel job can communicate with each other through an abstraction of distributed virtual shared memory (DVSM). Each process is allocated a virtual process ID (VPID) and can map a portion of its address space into the Elan. These address spaces, taken in combination, constitute an DVSM. Remote memory (i.e., memory on another processing node) can be addressed by a combination of a VPID and a virtual address. The SDRAM in the Elan can be used to keep the virtual-to-physical translation and routing tables of several jobs. Thus, in the presence of a context switch, there is no need to flush the Elan communication buffers and the system data structures.

2.2 Software

The resource management system (RMS) integrates various components of the QsNET [15, 16]. An RMS system connects a set of computers to a management network and to a Quadrics data network to provide high-performance user-space communication. To provide access to the RMS system, nodes can be connected to an external LAN. Computing nodes that are used for the parallel applications are accessed via RMS and can optionally have user logins disabled. Nodes can be divided into mutually exclusive *partitions* so that each partition can have different properties and policies for resource allocation, and several *configurations* can be defined and switched to allow a different set of properties per partition (e.g., different configurations can exist for day and night operations, allowing larger programs to run at night). One node (which can be separate from the computing nodes) is designated as a management node and holds the RMS database, which enables interfacing to the system using standard SQL queries.

The RMS provides a single point of interface to the system for resource management. It includes facilities for gathering information on resources (monitoring, auditing, accounting, fault diagnosis, and statistical data collection) and for resource handling (CPU allocation, access control, parallel jobs support, and execution, and scheduling). RMS is implemented as a set of UNIX commands and daemons that communicate using socket daemons and access the database for storing or retrieving all the system details. Of the set of daemons provided by RMS, two are concerned primarily with parallel job launching and scheduling. The *Partition Manager* (`pmanager`) is a per-partition daemon that runs on the management node. It handles requests for job launching and termination, checks the privileges and priorities allowed for each job, manages and allocates resources within its partition, and schedules the jobs. The *RMS Daemon* (`rmsd`) runs on each computing node in the system. It loads and runs user processes (using the application loader `rmsloader`), creates communication contexts for the application, delivers signals, and monitors resource usage and system performance.

Figure 2 shows how the system runs an eight-process job

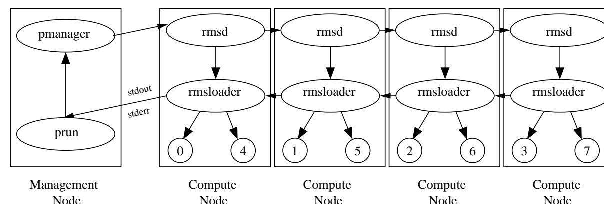

Figure 2. An eight-process program on four nodes

on four two-way SMP nodes. First, a user invokes a program called `prun` on the management node to launch her program, which in turn asks pmanager to allocate PEs and start the job on them. The `pmanager` notifies the `rmsd` processes on the allocated nodes to invoke an `rmsloader` process with the user's program; `rmsloader` also directs the `stdout` and `stderr` streams of the program to `prun`, which forwards it to the controlling terminal or output files.

The RMS scheduler allocates *boxes* (N nodes with a fixed number of PEs per node) to jobs so that they may take advantage of the hardware support of the QsNet for broadcast and barrier operations that operate over a contiguous range of network addresses.

Each partition can have its own scheduling policy and parameters (such as timeslice interval, time limit, etc.). The scheduling algorithm used can be one of the following:

1. Gang scheduling of parallel programs, in which all the processes in a program are scheduled and descheduled together.

2. Regular UNIX scheduling with the addition of simple load balancing.

3. Batch scheduling, in which the use of resources is controlled by a batch system.

When `pmanager` decides to suspend a running program or run another (either because of timeslice expiration, insertion of a higher-priority job to the system, or user command), it sends an appropriate command to the `rmsd` processes on the affected nodes through their sockets channel. Thus, the traffic density of the control messages is not determined by the number of jobs but rather by the timeslice value.

3 Experimental Methodology

3.1 Goals

We are primarily concerned with the following properties of the RMS gang scheduler:

1. How it scales as the multiprogramming level (MPL) increases. We would like to quantify the overhead that is introduced by the scheduler.

2. How it scales as the number of nodes increases.
3. How different memory requirements of the applications affect coscheduling performance.
4. How the scheduler handles different communication granularities of applications. What effect, if any, a context switch has on the network and the communication buffers of the network interface.
5. What the effect of the timeslice length is. Which values offer a good trade-off between response time and scheduling overhead.

3.2 Experimental Framework

We have designed a micro benchmark to test several aspects of the gang scheduler. Our benchmark is structured as a program that loops over an array, reads a value from one entry, performs a simple floating-point calculation, writes the result in another entry of the same array, and copies a subset of these results on another array which serves as the communication buffer. The stride for traversing the array is a constant large prime. At a specified frequency the program performs a total exchange with its peer processes (using MPI_Alltoall [17]). In a total exchange of b bytes (also known as personalized all-to-all communication), each of the n processes sends a distinct message of b/n bytes to every other process. An external script launches this program with different parameters according to a predefined sequence and with several instances to create the desired MPL. In the experiments we varied the following parameters:

- Total computation cycles, number of total exchanges and communication buffer size. These three factors determine the computation/communication granularity.
- Number of processes per job.
- multiprogramming level.
- Size of memory array for read/write operations.

We divide the run time of each job by the MPL. Job slowdown is compared with the basic case, in which a program, does not communicate, uses a single-byte array and runs on one PE only, with no other jobs.

For our measurements we used a cluster of 16 dual-processor nodes running Linux 2.4.0. Each node is equipped with two 733-MHz Pentium-III processors, a 66-MHz PCI bus, 1 GB of ECC memory, and QsNET and Ethernet connections. The first node is used as a management node for RMS.

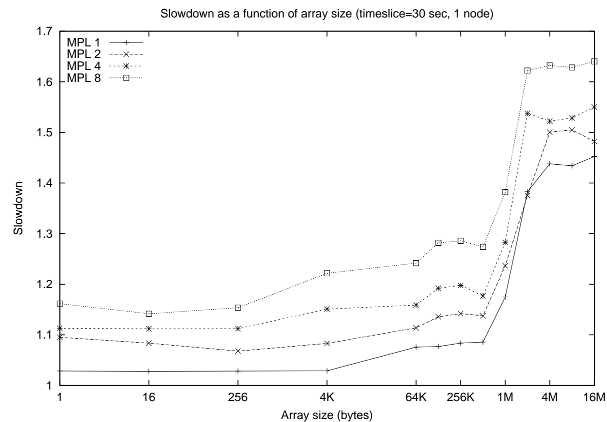

Figure 3. Effect of array size

3.3 Workload

We made several assumptions on the workload for this study. First, when the MPL is greater than 1, we launch all the jobs together. The amount of computation of each job is 100 million read/modify/write cycles, which is approximately 50 seconds of run time. We found this granularity to be large enough to make the experimental sample relatively stable and small enough to make large experiments practical. Still, some variability in the results exists. This variability stems from various system parameters that are difficult to control and add noise to the experiments. Such parameters include small architectural differences between the nodes, temporal effects such as varying load of Linux and RMS daemons, and local scheduling decisions that are done by Linux on each SMP and affect cache affinity and synchronization issues. We used the following default values for all other parameters (unless otherwise indicated for individual experiment):

1. Read/write array size of 1 MB, with no separation between read and write locations.
2. 1,024 total exchanges, with 4,096-bytes of total buffer size. This represents a granularity of a total exchange every 50 ms of computation.
3. Sixteen processes running on 8 nodes.
4. Timeslices of 10 and 30 seconds.

4 Experimental Results

4.1 Effect of Memory Usage

Figure 3 shows the slowdown of gang scheduling multiple jobs on a single processing node with a timeslice of 30 seconds. We can see that the overhead is approximately 10% more than the basic case when a single job is run in dedicated

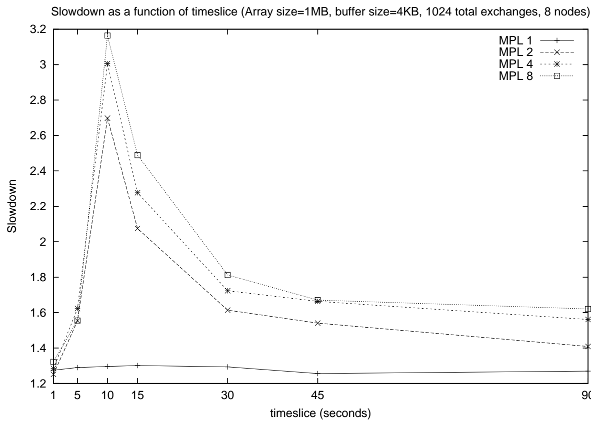

Figure 4. Effect of timeslice length.

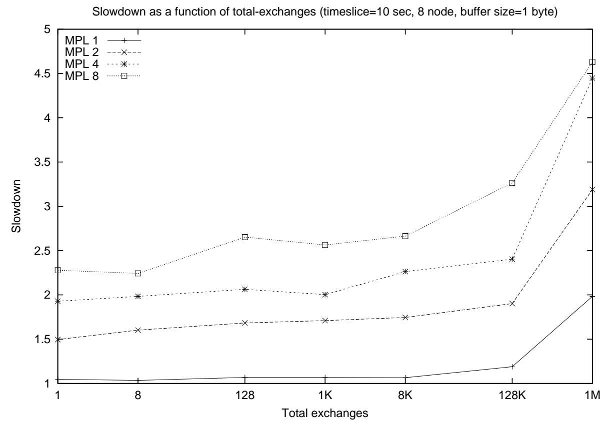

Figure 5. Effect of latency-bound communication.

mode. This indicates that process context switch penalty is contained. It is worth noting that our benchmark has a deliberately poor data reuse (because we use a large stride for scanning the array) so that cache concerns have no real effect on the results. The picture changes for workloads that do not fit in the main memory. For example, running two processes with a memory footprint of 512 MB each on one machine (thus exhausting the machine's physical memory) results in a slowdown of 30. The same applications using 620 MB each yields a slowdown of more than 200.

4.2 Effect of Timeslice Quantum

Figure 4 shows the effect of the timeslice on the run time. We would expect a decrease in run time as the timeslice increases, due to a lower amount of context switches and associated overhead. This can be seen in the graph for timeslice values larger than 10 seconds, although the responsiveness of small and interactive jobs can be low for such timeslices. Counter intuitively, run time is actually better when the timeslice is smaller than 10 seconds. This occurs because the `pmanager` process cannot handle this rate of control messages, and skips several context switches. This results in poor fairness and starvation, which is demonstrated in a very high variation of the jobs' run time. In the case of 4 jobs and a timeslice of 1 second, we measured a standard deviation of 65 when the average run time is 131 seconds. This in turn also affects adversely the responsiveness of starved jobs.

From our measurements it can be seen that the gang scheduler requires a timeslice larger than 30 seconds to operate effectively.

4.3 Effect of Communication

To measure the effect of communication on the gang scheduler, we use both latency-bound and bandwidth-bound communication patterns. For the former, we set the size of the communication buffer to one byte (the time for a single-

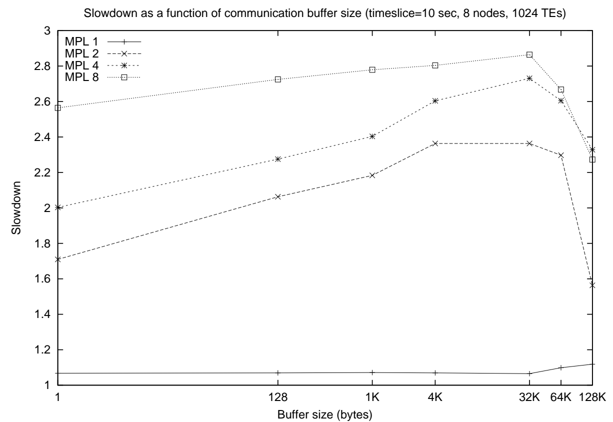

Figure 6. Effect of bandwidth-bound communication.

byte total exchange is dominated by the communication latency) and vary the number of total exchanges to the point of system saturation. For the latter, we fix the number of total exchanges to 1,024 and increase the buffer size up to 128 KB, thus fixing the latency and increasing the bandwidth requirement. In both cases, we stress the scheduler further by using a relatively small timeslice value of 10 seconds[3].

Figure 5 shows our results with the latency-bound communication patterns. The difference between the slowdown curves for each MPL is fairly constant, thus implying that the gang scheduler is relatively insensitive to the number of total exchanges. In absolute terms, the slowdown curves are relatively flat up to 1M total exchanges (which represents one total exchange for every 50 μsec of computation). This sudden and severe slowdown is primarily due to the fact that the total communication time is now the same order of magni-

[3]As discussed in Section 4.2, this value represents the smallest timeslice that still guarantees fairness.

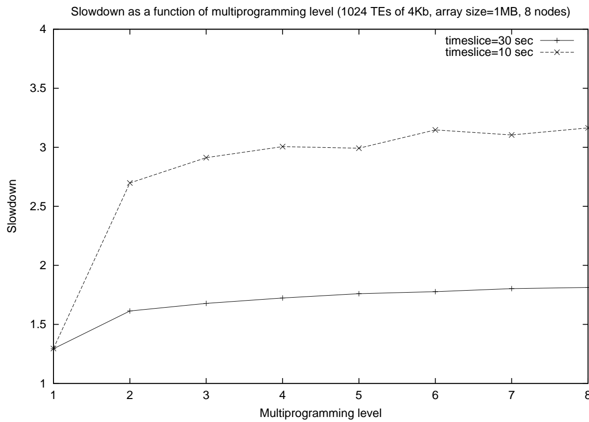

Figure 7. Effect of MPL.

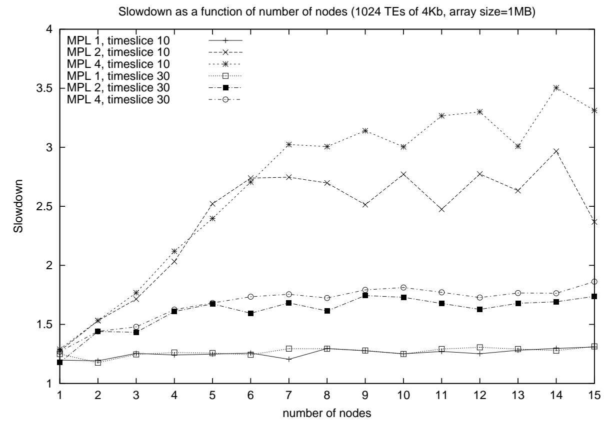

Figure 8. Node scalabilty.

tude as the timeslice.

Figure 6 presents the results with the bandwidth-bound access patterns. As in Figure 5, the relative differences between the curves remains nearly constant, indicating the gang scheduler's insensitivity to the bandwidth requirements of the benchmark. This behavior is due to the ability of the Elan NIC to store multiple network contexts, as outlined in Section 2.1, thus allowing lightweight context switches and eliminating the need for a full network clean-up. For buffer sizes larger than 32 KB, the Elan offers a degree of overlap between the computation and communication of distinct jobs.

4.4 Effect of Multiprogramming Level

Gang scheduling provides the advantage that the amount of control information exchanged between the resource manager and the workers is unaffected by the number of concurrent jobs. The determining factor is actually the timeslice value as a constant amount of information is exchanged every timeslice, irrespective of the number of jobs. Therefore, we expect that the cost of adding more jobs to the gang scheduler would be relatively low. Figure 7 verifies our intuition by showing that adding more jobs after the second incurs little additional scheduling overhead. Furthermore, the figure shows that the scheduling overhead, and hence slowdown, depends on the timeslice quantum.

4.5 Node Scalability

In this section, we study the scalability of the gang scheduler with respect to the number of nodes in the system. Figure 8 shows the results of running six different tests of a continuous range of nodes up to 15 (with each node consisting of two PEs). As expected from the results in Section 4, the scheduler performs erratically when the timeslice quantum is small (i.e., 10 seconds) and the MPL is two or four, thus creating fairness problems. These curves also exhibit steep growth from one to seven nodes with moderate growth thereafter. However, the overall slowdown reaches values over 250% higher than the base case, thus making the choice of such a timeslice unattractive.

When the timeslice is 30 seconds for MPL two and MPL four, the slowdown initially grows rapidly (~40% clip) but at a more moderate rate than when the timeslice is 10. Beyond seven nodes, the growth rate moderates further, 3% for MPL 2 and 9% for MPL 4. These results indicate that for a larger number of nodes, the overhead associated with context switches scales gracefully and is likely due to the fact that context switches are not penalized by the NIC, so the increase in traffic does not aggravate the context-switch overhead.

For an MPL of one, fairness is not an issue, and the timeslice has no effect. In fact, Figure 8 shows that the curves for an MPL of one are virtually identical, differing only because of the inherent variability of the test system (Section 3.3). These curves show a very small but gradual growth in the slowdown from 1.2 to 1.3 due to the increasing cost of the MPI_AlltoAll() operation as the number of nodes increases.

5 Conclusions

This paper described an experimental study of the Quadrics gang scheduler. We demonstrated that the scheduler is relatively insensitive to the communication granularity in terms of latency and bandwidth and may actually improve the overall run time of bandwidth-hungry programs that are coscheduled by overlapping computation and communication. Further, we showed that the scheduler is also insensitive to the amount of memory used but only as long as the physical memory of the machine is not exhausted. With respect to the scalability of the scheduler, it performed quite well for up to 15 nodes (or 30 PEs) and an MPL of 8.

On the negative side, the scheduler is very sensitive to the timeslice quantum and can perform poorly if a small value is chosen. For time slices under 30 seconds, performance

degradation of up to 90% can be observed in some cases; for values of 5 to 10 seconds, severe fairness and starvation problems occur, which have an adverse effect on system responsiveness. On the other hand, using larger values for the timeslice has implications on responsiveness, particularly for short or interactive jobs.

6 Future Work

Our future work encompasses three directions. First, we will ascertain the suitability of the Quadrics gang scheduler for real-world applications. Second, we will examine the performance of the gang scheduler using more realistic workloads (both in simulation and on our Quadrics cluster). The simple workload described in this paper does not launch jobs according to a workload model or a real workload trace, taking into account issues such as day/night/weekend periods, development issues, etc. Lastly, we plan to measure the scheduler's performance on larger-scale machines so that a better understanding of its scalability properties can be obtained.

References

[1] D.G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16(4), 1992.

[2] Yoav Etsion and Dror G. Feitelson. User-Level Communication in a System with Gang Scheduling. In *Proceedings of the International Parallel and Distributed Processing Symposium 2001, IPDPS2001*, San Francisco, CA, April 2001.

[3] Dror G. Feitelson, Anat Batat, Gabriel Benhanokh, David Er-El, Yoav Etsion, Avi Kavas, Tomer Klainer, Uri Lublin, and Marc Volovic. The ParPar System: a Software MPP. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1: Architectures and systems, pages 754–770. Prentice-Hall, 1999.

[4] Hubertus Franke, Pratap Pattnaik, and Larry Rudolph. Gang Scheduling for Highly Efficient Distributed Multiprocessor Syetems. In *6th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS '96)*, pages 1–9, Annapolis, MD, October 1996.

[5] Atsushi Hori, Hiroshi Tezuka, Yutaka Ishikawa, NoriYuki Soda, Hiroki Konaka, and Muneori Maeda. Overhead Analysis of Preemptive Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 217–230. Springer-Verlag, 1998.

[6] http://www.hippi.org/cST.html. Scheduled Transfer Protocol (ST), (ST is also being commercially promoted as part of GSN), 1996–present.

[7] http://www.viarch.org. VI Architecture, 1998–1999.

[8] Mario Lauria and Andrew Chien. High-Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, November 1995.

[9] Charles E. Leiserson. Fat-Trees: Universal Networks for Hardware Efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[10] Charles E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[11] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proceedings of Third International Conference on Distributed Computing Systems*, 1982.

[12] Fabrizio Petrini, Adolfy Hoisie, Wu chun Feng, and Richard Graham. Performance Evaluation of the Quadrics Interconnection Network. In *Workshop on Communication Architecture for Clusters (CAC '01)*, San Francisco, CA, April 2001.

[13] Quadrics Supercomputers World Ltd. *Elan Reference Manual*, January 1999.

[14] Quadrics Supercomputers World Ltd. *Elite Reference Manual*, November 1999.

[15] Quadrics Supercomputers World Ltd. *RMS Reference Manual*, June 2000.

[16] Quadrics Supercomputers World Ltd. *RMS User Manual*, April 2000.

[17] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI - The Complete Reference*, volume 1, The MPI Core. The MIT Press, 1998.

[18] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An Operating System Coordinated High-Performance Communication Library. In *Proceedings of High-Performance Computing and Networking '97*, pages 708–717, April 1997.

[19] Werner Vogels, David Follett, Jenwi Hsieh, David Lifka, and David Stern. Tree-Saturation Control in the AC^3 Velocity Cluster. In *Hot Interconnects 8*, Stanford University, Palo Alto CA, August 2000.