Workshop on Scheduling and Resource Management for Cluster Computing (in conjunction with the ICPP01). LA-UR 01-3025

Gang Scheduling with Lightweight User-Level Communication[*]

Eitan Frachtenberg, Fabrizio Petrini, Salvador Coll, Wu chun Feng

CCS-3 Modeling, Algorithms, & Informatics Group
Computer & Computational Sciences Division
Los Alamos National Laboratory
{eitanf,fabrizio,scoll}@lanl.gov

Abstract

In this paper we explore the performance of gang scheduling on a cluster using the Quadrics interconnection network. On such a cluster, the scheduler can take advantage of the unique capabilities of this network, including a NIC based processor and memory and efficient user level communication libraries. We developped 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 level. Our experiments show that the gang scheduler preforms relatively well under most workload conditions, is largely insensitive to the amount 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, CoScheduling, Performance Evaluation, Parallel Architectures, Quadrics interconnect.

1 Introduction

Gang scheduling has been proposed as an efficient means to multiprogramming of frequently communicating processes on parallel supercomputers [11, 1]. Gang scheduling offers many advantages for job and system efficiency which are similar to those of time-sharing in uniprocessor systems. The system can be better utilized by the scheduler's ability to preempt jobs in several ways:

- The responsiveness of the system to interactive and high priority jobs can be very high, even when the system is highly utilized.
- Jobs requiring a high number of processors do not have to wait for other jobs to terminate until the resource requirement is met before being launched. Furthermore, once such a job is running, it does not monopolize resources, and other jobs can still be executed.
- Unused resources can be utilized by low priority jobs and reallocated to higher priority jobs when these become available.
- 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 due to the effect of the context switch on the computing nodes.

This overhead is caused by the resource sharing between multiple jobs and spans several dimensions. A first dimension is the cache memory and the Translation Look-aside Buffer (TLB): a context switch between processes causes a "cold start", which initiates the load of the working set of the new process and the eviction of the old working set.

A severe performance penalty is usually caused by the overflow of the physical memory into the virtual memory. 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 main memory.

Another important dimension is the interface between the processing node and the network. A common architectural solution to deliver high-performance over system area networks (SANs) is the close integration between processors and network interface. This is usually obtained using user-level messaging protocols, that minimize the communication delay removing the operating system from the communication

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

protocols [8, 6, 18, 7, 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 are "pinned" in main memory, that are reserved by the user-level protocol to perform the inter-node communication. With gang scheduling, all these buffers must be properly managed between context-switches. Finally, the context switch between 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/restored at every context switch, in order to minimize the amount of pinned memory. The coordination between processing nodes is achieved through synchronized clocks. The nodes do not interact through explicit synchronization and do not receive a coordination message from 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.

Network flushing was pioneered by the CM-5 Connection Machine [10]. During a context switch, all packets are "dropped down" to the closest node in the fat-tree network and stored in a temporary node. When the job is re-scheduled, these packets are re-injected to complete their trip.

In the ParPar scheduler [3] the context switch is coordinated by a master daemon, which sends a collection of point-to-point messages to the worker nodes. Upon receiving the synchronization message, a partner daemon in the worker node stops 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 FIFO order and the network delivers packets using a single determinstic path between each pair of nodes, this protocol guarantees that no packets belonging to the previous timeslice will be received after the halt message. In [2] it is shown that the buffers in the network interface are often underutilized during the context switch, so the context switch overhead can be reduced by saving only the packets that are in the buffer rather than the whole set of buffers.

Network flushing is also used in the SCore-D cluster [5]. SCore-D doesn't need to send special control messages because each single packet is explicitly acked or nacked by the destination. So 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 since it is to be used as the interconnect for the

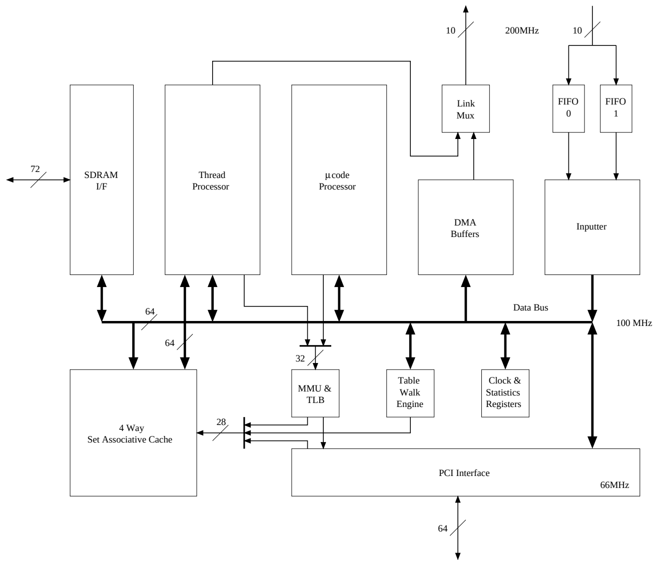

Figure 1: Elan functional units

30Tops ASCI-Q machine, currently developed 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 start by describing the hardware and software features of a Quadrics-based cluster in section 2. In section 3 we provide the details of the experimental methodology we used in this study. We provide experimental results obtained with our cluster and analysis in section 4. Finally, section 5 concludes the paper and outlines future research work.

2 Overview of QsNET and RMS

2.1 Hardware

QsNET [12] is based on 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 which 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 contain-

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

[2]http://www5.compaq.com/alphaserver/news/archive/2000/supercomputer_0822.html

ing one or more CPUs. In addition to generating and accepting packets to and from the network, the Elan also provides substantial local processing power to implement high-level message-passing protocols such as MPI. The internal functional structure of the Elan is shown in Figure 1.

In the Elan there are four independent microcode threads: (1) Inputter thread; handles input transactions from the network. (2) 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) Processor-scheduling thread; prioritizes and controls the scheduling and descheduling of the thread processor. (4) Command-processor thread; handles operations requested by the host (i.e., "command") processor at user level. The Elan is also equipped with 64 MB of SDRAM.

Processes in a parallel job can communicate with each other through an abstraction of distributed virtual shared memory. 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 a distributed virtual shared memory. 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. Messages are chunked by the DMA engine in packets of 320 bytes which are delivered in-order.

2.2 Software

The various compnents of the Quadrics network are integrated by a software environment called Resource Management System (RMS)[15, 16].

An RMS system is a set of computers that are all connected to a management network, and a Quadrics data network to provide high performance user-space communication. Some nodes can be connected to an external LAN to provide access to the RMS system. 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 different set of properties per partition (for example, different configurations can exist for day and night operation, to allow 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.

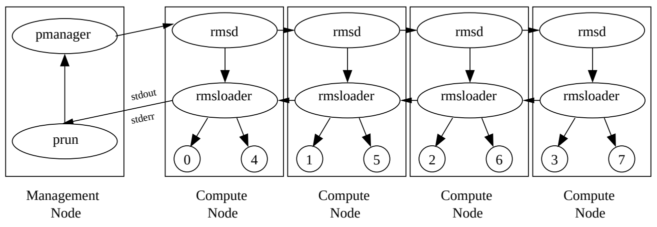

Figure 2: Loading and Running a parallel program

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 8-process job on 4 t-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 CPUs for the job and start it on them. `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 CPUs per node) to jobs, so that they may take advantage of the hardware support of the QsNet for broadcast and barrier operations which operate over a contiguous range of network addresses.

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

1. Gang scheduling of parallel programs, where all the pro-

cesses in a program are scheduled and descheduled together.

2. Regular UNIX scheduling with the addition of simple load balancing.
3. Batch scheduling, where use of resources is controlled by a batch system.

When `pmanager` decides to suspend a running program or run another (either due to 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 amount 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 (GS):

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 in the cluster increases.
3. How different memory requirements and memory access patterns of the applications affect the overall performance.
4. How the GS handle different communication granularities when applications are gang-scheduled. 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. What the usable range of timeslices is and 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 some 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, also known as personalized all-to-all communication, each process sends a distinct message 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. Note that if n processes are having a total exchange of b bytes, each process sends and receives $\frac{b}{n}$ to and from every other process.
- Number of processes per job.
- MPL (the number of jobs that are launched concurrently).
- Size of memory array for read/write operations.

We normalize the run time of each job with the MPL. Job slowdown is compared to the basic case, where a program uses an array size of one byte, does not communicate 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 Quadrics and Ethernet connections. The first node is used as a management node for RMS.

3.3 Workload

Several assumptions on the workload were made for this study. First, when the multiprogramming level 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. This granularity was found to be large enough to make the experimental sample relatively stable, and small enough to make large experiments practical. Still, it should be noted that there is some variability in the results. 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 like varying load of Linux and RMS daemons, and local scheduling decisions

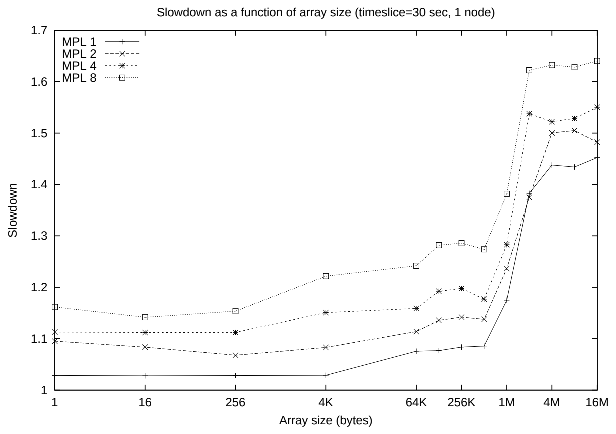

Figure 3: Effect of array size and multiprogramming level on runtime

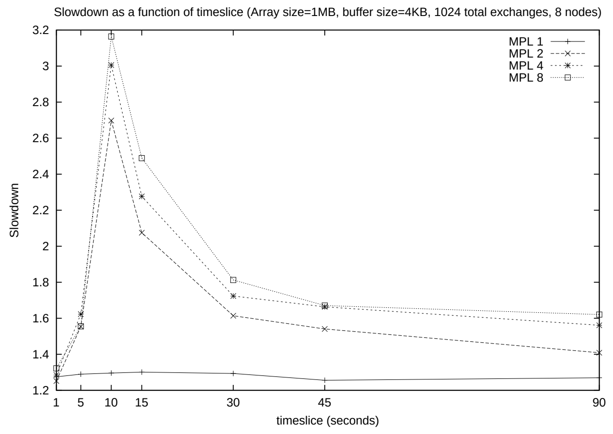

Figure 4: Run time slowdown as a function of timeslice length

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 on a per-experiment basis):

1. Read/write array size of 1 MB, with no separation between read and write locations.
2. 1024 total exchanges, with 4096-bytes of total buffer size. This represents a granularity of a total-exchange every 50 ms.
3. 16 processes running on 8 nodes.
4. Timeslice of 30 seconds.

These default parameters represent an application that uses enough memory, so that memory bandwidth becomes a relevant performance factor, but not enough that swapping becomes one. It communicates small messages frequently and synchronously, as is the worst case behavior for many parallel applications. A multiprogramming level of 4 is supposed to represent a moderate-to-high workload.

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 mode. This indicates that for workloads that do no incur swapping the performance penalty from gang scheduling is contained. It is worth noting that our benchmark has a deliberate 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 main memory. For example, running two processes of 512 MB each on one machine (thus exhausting the machine's physical memory) exhibits 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 runtime. 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. Counterintuitively, run time is actually better for timeslice values smaller than 10 seconds. This occurs because the `pamanger` 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 runtime is 131 seconds. This in turn also affects adversly 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.

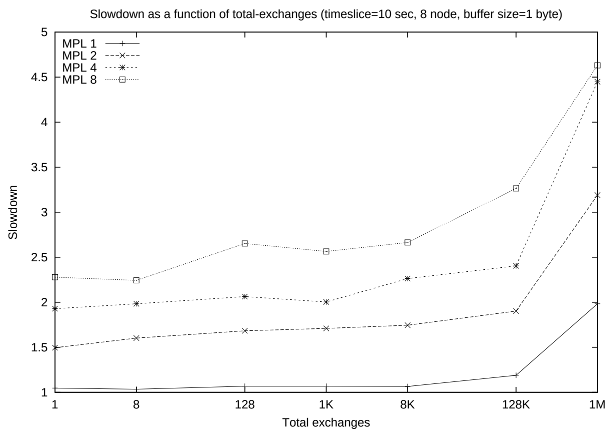

Figure 5: Effect of communication amount on runtime on one node (2 PEs). The communication buffer size is 1 byte

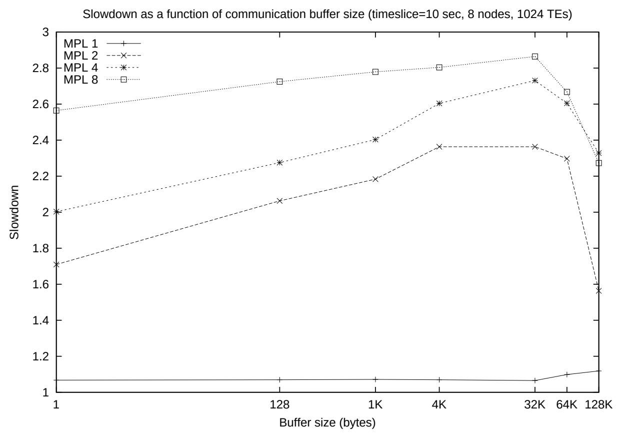

Figure 6: Effect of different communication buffer sizes and multiprogramming levels on runtime.

4.3 Effect of Communication

We measure the effect of communication on the gang scheduler in two dimensions, using latency-bound and bandwidth-bound communication patterns. For the former, the communication buffer is only one byte, and the number of total exchanges varies. For the latter, we have 1024 total exchanges and we increase the buffer size upto 128 KB. In both cases, we try to stress the scheduler further by using the relatively low timeslice value of 10 seconds. As discussed in the previous section, this value represents the smallest timeslice that still guarantee fairness. Figure 5 shows the result of the first experiment. It can be seen that the difference between the slowdown curves are nearly flat for each MPL, upto the point of 1M exchanges (which represents one total exchange for every 50 μsec). Even though the slowdown values are relatively high, because the gang scheduler is operating in a saturated mode, they are almost insensitive to changes in the number of total exchages.

Figure 6 describes the result of the second experiment. Again, we see that the gap between the slowdown curves for different MPLs remains nearly constant, suggesting that the gang scheduler is insensitive to the bandwidth requirements of the benchmark. This is due to the fact that the Elan network interface card can store multiple network contexts, as outlined in section 2.1. This can support a lightweight context switch and eliminates the need for a full network cleanup. It is also worth noting that for larger buffers, the Elan offers some degree of overlapping between computation and communication of distinct jobs. This explains the run time improvement for buffers larger than 32 KB.

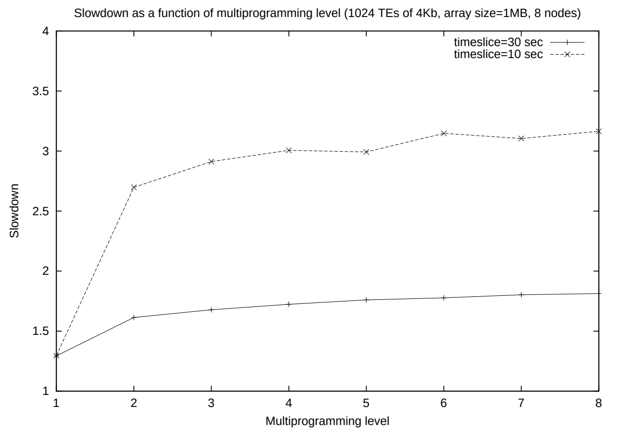

Figure 7: Effect of multiprogramming level on run time

4.4 Effect of Multiprogramming Level

An important property of gang scheduling is that the amount of control information exchanged between the resource manager and the workers is insensitive to the number of concurrent jobs. The determining factor for the amount of information exchanged is actually the timeslice value, because a constant amount of information is exchanged on every timeslice, irrespective of the number of jobs. We would therefore expect the cost of adding additional jobs to a gang scheduling system to be relatively low. This can be clearly seen in Figure 7. The scheduling overhead for adding more jobs after the second is relatively small, and furthermore, is only determinded by the timeslice quantum.

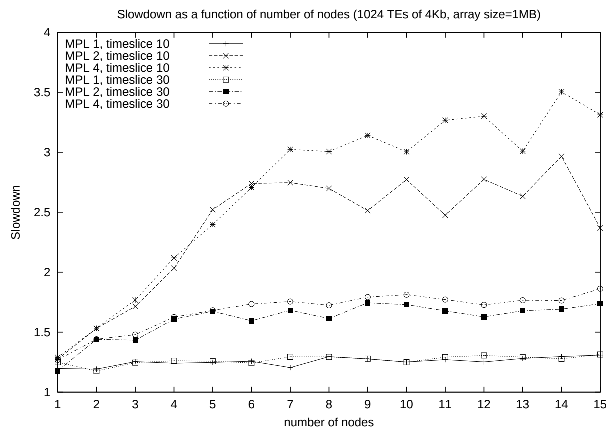

Figure 8: Node scalabilty

4.5 Node Scalability

To conclude our experimental results, we would like to measure how the gang scheduler scales as the number of nodes is increased. To this end, we run and analyse six test sets over a continuous range of nodes from 1 to 15 (with each node using both PEs). Figure 8 shows the slowdown results of these runs. One apparent property is the instability of the test-sets that were run with a timeslice quantum of 10 seconds. As was shown in section 4.2, for lower timeslice values the scheduler performs erratically, where actual timeslice values can vary widely, and serious fariness problems occur. These properties are displayed in this figure for MPL values of 2 and 4. For an MPL value of 1 the timeslice has no effect. It can be seen from the graph that both curves for MPL 1 are nearly identical, differing only because of the inherent variablity of the test system (see section 3.3). These two curves show a relatively small, gradual growth in the slowdown from 1.2 to 1.3 (approx. 8%), which is explained by the increasing cost of the MPI_Alltoall operation as the number of nodes increases. Both the MPL 2 and MPL 4 results with a timeslice of 30 seconds exhibits a higher growth rate for the first 7 nodes (43% and 38% respectively). However, for more than 7 nodes the growth rate is rather moderate, reaching 3% and 9% respectively. These results indicate that for a larger number of nodes, the overhead associated with context switches scales gracefully, and is probably due to the fact that context switches do not penalize the NIC, so the increase in traffic does not aggravate the context switch overhead. Both MPL 2 and 4 results indicate a similar growth pattern when the timeslice is reduced to 10 seconds: a steep growth from 1 to 7 nodes, and a moderate growth afterwards. However, The picture is quite different in terms of slowdown - both reach slowdown values of 250% and upwards, and make the choice of such a timeslice value unattractive.

5 Conclusions

This paper describes the tests that were conducted to explore the properties of the Quadrics gang scheduler. We have shown that it is relatively insensitive to the amount of communication granularity in terms of latency and bandwidth, and may actually improve the overall runtime of bandwidth-hungry programs that are co-scheduled, by overlapping some of the computation and communication times. Further, we have shown that the scheduler is also insensitive to the amount of memory applications use, as long as the physical memory of the machine is not exhausted. Another important issue is the scheduler's scalability in terms of number of nodes and number of coscheduled programs. It was shown in both cases to be quite good for upto 30 PEs and 8 coscheduled programs respectively.

On the negative side, the scheduler is very sensitve to the timeslice quantum, and can perform poorly if a low value is chosen. For timeslice values of under 30 seconds, performance degradation of upto approximately 90% can be observed in some cases; for values of 5-10 seconds, severe fairness and starvation problems occur, which have an adverse effect on system responsiveness. On the other hand, using higher values for the timeslice quantum also has implications on responsiveness, especially for short or interactive programs.

Future Work

One future direction that can ascertain the suitability of the Quadrics gang scheduler for real world systems is the usage of actual applications for the measurements. Another interesting venue for research is to use realistic workloads, or simulations thereof. The simple workload described in this paper does not launch jobs according to a workload model or real workload trace, taking into account issues such as day/night/weekend periods, development issues, etc. Another obvious, but nevertheless important extension of this paper would be to measure the scheduler's performance on large-scale machines, so that a substantial understanding of its scalability properties can be obtained.

References

[1] D.G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefi ts 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 15th International Parallel and Distributed Processing Symposium*, 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.