Scalable Resource Management in High Performance Computers *

Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, and Salvador Coll
CCS-3 Modeling, Algorithms, and Informatics Group
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory

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

18th February 2002

Abstract

Clusters of workstations have emerged as an important platform for building cost-effective, scalable and highly-available computers. Although many hardware solutions are available today, the largest challenge in making large-scale clusters usable lies in the system software.

In this paper we present STORM, a resource management tool designed to provide scalability, low overhead and the flexibility necessary to efficiently support and analyze a wide range of job scheduling algorithms. STORM achieves these feats by closely integrating the management daemons with the low-level features that are common in state-of-the-art high-performance system area networks. The architecture of STORM is based on three main technical innovations. First, a sizable part of the scheduler runs in the thread processor located on the network interface. Second, we use hardware collectives that are highly scalable both for implementing control heartbeats and to distribute the binary of a parallel job in near-constant time, irrespective of job and machine sizes. Third, we use an I/O bypass protocol that allows fast data movements from the file system to the communication buffers in the network interface and vice versa.

The experimental results show that STORM can launch a job with a binary of 12MB on a 64 processor/32 node cluster in less than 0.25 sec on an empty network, in less than 0.45 sec when all the processors are busy computing other jobs, and in less than 0.65 sec when the network is flooded with a background traffic. This paper provides experimental and analytical evidence that these results scale to a much larger number of nodes. To the best of our knowledge, STORM is at least two orders of magnitude faster than existing production schedulers in launching jobs, performing resource management tasks and gang scheduling.

Keywords: Resource management, Job Scheduling, Gang Scheduling, Performance Evaluation, Parallel Architectures, Quadrics interconnect, User-level Communication, I/O bypass, Cluster Computing

1 Introduction

Recent improvements in commodity processors and networks, combined with their attractive price/performance ratio, have made clusters of workstations a popular form of high performance computing.

For example, RLX[1], Compaq[2] and HP[3] have recently introduced high-density *blade servers* which incorporate low-power processors. Several hundreds of these processors can be integrated in a single rack and it is foreseeable that in the near future clusters with thousands of processors will quickly move out of the boundaries of research labs and academic research and will become widespread in the commercial world.

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

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

[2] http://www.compaq.com/products/servers/proliant-bl/e-class/index.html

[3] http://www.hp.com/products1/servers/blades

On the networking side, Infiniband [7] is an emerging standard that can provide connectivity to a potentially large number of processing nodes which is likely to become the default interconnection network of future commodity clusters. Important aspects of Infiniband are high-performance, fault-tolerance, hardware support for large-scale system management and close integration with the processing nodes.

Although powerful hardware solutions are already available, the largest challenge to make these clusters usable lies in the system software. The scalability of resource management, job scheduling and job launching are important aspects that are often overlooked.

Many run-time environments use a globally mounted file-system, such as NFS, when they have to move executables, for example when they spawn the processes of a job.[4] This design, where potentially many clients are accessing a single file on a single server at the same time is inherently non-scalable. In such environments, the typical method of launching a job is a shell script that loops over remote shell commands, which use TCP/IP, to start processes on remote nodes. Though that is not a problem on small-scale clusters, this approach can have severe performance and scalability limitations on larger systems with several hundreds (or, maybe, thousands) of nodes.

The ParPar [17] cluster environment addresses the problem of distributions of control messages, from a management node to a set of clients, by implementing a special-purpose multicast protocol. This protocol, called Reliable DataGram Multicast (RDGM), broadcasts UDP datagrams on the network and adds selective multicast and reliability. Each datagram is prepended by a bit string that identifies the set of destinations, and each node in the destination set sends and acknowledgment to the management node after the successful delivery of the broadcast datagram. By using RDGM, a job can be launched in a few tens of seconds on a cluster with 16 nodes, with relatively good scalability.

GLUnix [16] is an operating system middle-ware for clusters of workstations, designed to provide transparent remote execution, load balancing, coscheduling of parallel jobs and fault-detection. In [16] the authors of GLUnix note that the overhead in the master node, when forking a parallel job, increases by a small amount (an average of 220 μsec per node). Also, one-to-many communication patterns scale relatively well, at only 230 μsec per node. When GLUnix launches a job, remote execution messages are sent from the management node to all the daemons that will run the job. Each of these daemons generate a reply message, indicating success or failure. When performing remote execution to many nodes (more than 32, in the experimental results shown in [16]) the replies from earlier daemons in the communication schedule collide with the remote execution requests sent to later daemons on the switched Ethernet, causing a substantial performance degradation. Thus, many-to-one communication patterns using TCP/IP over Ethernet may exhibit poor scalability.

Scalability problems are already evident in ASCI-scale machines, with thousands of nodes. The Computational Plant (Cplant) at Sandia National Laboratories includes several large-scale parallel computers composed of commodity computing and networking components. In order to enhance scalability, Cplant uses a high-performance interconnect, Myrinet [5], and a custom lightweight communication protocol based on Portals [6]. When the run-time environment of Cplant launches a job, it first identifies a group of active worker nodes, organizes them in a logical tree structure and then fans out the executable. The experimental results in [6] show that a parallel job can be launched on a 1010-node cluster in about 15 seconds, depending on the binary size.

Many recent research results show that job scheduling algorithms can substantially improve scalability, responsiveness, resource utilization and usability of a large-scale parallel machine [2] [13]. Unfortunately, the body of work developed in the last few years has not yet led to any practical applications/implementations of such gang scheduling and coscheduling algorithms in parallel clusters. We argue that one of the main problems is the lack of flexible and efficient run-time systems that can support the implementation and evaluation of new scheduling algorithms, which are expected to replace the conventional, space-shared, schedulers.

In this paper we present STORM (Scalable TOol for Resource Management). STORM provides a number of technical breakthroughs that can pave the way to research advances in the area of resource management and job scheduling. STORM achieves these feats by closely integrating a collection of management daemons with a state-of-the-art, high-performance interconnection network, the Quadrics network (QsNET) [21]. Large-scale clusters based on the QsNET are already installed at CEA (France), LLNL, ORNL, Pittsburgh Supercomputer Center (largest unclassified computer in the world), and Los Alamos National Laboratory (expected to be largest computer in the world at the end of the year 2002). STORM is also designed to provide large flexibility, in order to quickly prototype and test new scheduling algorithms.

[4] http://www.openpbs.org

At the core of STORM there are three main technical innovations. First, the management daemons of STORM can exchange control messages using the lowest level interface of the QsNET and the communication overhead is also relieved by user-level threads that run in the thread processor of the network interface. These threads can process incoming messages and perform protocol processing without interrupting the computing node. Second, STORM fully exploits the hardware support provided by the QsNET to broadcast control messages of arbitrary size, and the possibility of combining the acknowledgments in the network switches, thus dramatically reducing the overhead of managing a large number of nodes. Third, the threads in the network interface can use an extremely lightweight I/O by-pass protocol, that allows interactions with the filesystem with almost no measurable overhead in the processing nodes.

The rest of this paper is organized as follows. Section 2 describes the architecture of STORM, the main design goals, the characteristics of QsNET that are relevant to the STORM implementation and its process structure. In Section 3 we evaluate the performance of STORM with a set of micro-benchmarks. We focus our attention on two main aspects: the capability of launching jobs and supporting gang scheduling. Finally, some concluding remarks and future directions are given in Section 4.

2 STORM Architecture

This section describes the architecture of STORM. The most important design goals for STORM were:

1. To provide resource management mechanisms that are scalable, high-performance and lightweight.
2. To support the implementation of most current and future job scheduling algorithms.

To fulfill the first goal, we use a set loosely-coupled daemons that communicate with extremely fast messages. Coordination of the daemons is done through scalable strobes (heartbeat messages) that are implemented by an efficient hardware multicast. For the second goal, the daemons were designed so that modules for different scheduling algorithms can be "plugged" into them. In this paper, we focus on one of the most popular of these algorithms, gang-scheduling (GS)[8, 20]. GS employs both space sharing and time sharing to allocate resources to jobs. All the processes of a given job run in the same allotted time slot, for the duration of the timeslice quantum, and are then context-switched to a different job in a cyclic manner, at the end of each time slot.

2.1 Overview of STORM

Several issues were considered crucial for STORM, and were incorporated in its design and implementation:

1. **Flexibility:** probably the most important feature of STORM is the ability to support many modern and future scheduling algorithms in order to provide a valuable research tool. STORM currently supports local scheduling, First-Come-First-Served (FCFS, batch), FCFS with backfilling, GS, and Spin-Block (implicit coscheduling). Moreover, other scheduling methods can be readily added to the system. In fact, our research directions include the implementation of buffered coscheduling (BCS) [9, 10] and other scheduling algorithms, for in-depth study of their properties.
2. **Scalability:** One of the most important metrics for resource-managers in the advent of large high-performance computing (HPC) machines is the scalability with the number of nodes. STORM is designed so that many of the scheduler operations are decentralized and asynchronous, and the only two global operations, namely job launching and strobes, are implemented by fast and lightweight hardware multicast.
3. **Performance:** To make the experimental results both valid and comparable to current state-of-the-art systems, the design should strive for best scheduler and application performance. This requirement translates to a lightweight, efficient scheduler, with fast user-level internal communication and a relatively low-overhead implementation.
4. **Simplicity:** The scheduler should not be over-complicated, so that maintenance and augmentation of new scheduling algorithms will incur little overhead. This implies that parallel applications should not be changed to accommodate the system, and only need to be re-linked with a modified version of MPI.

5. **Portability:** The scheduler should be designed so that porting it to other hardware platforms, interconnects or even operating systems will be relatively simple, to allow for extended testing and enhancing. To this end, STORM runs entirely in user level with no operating system modifications. Furthermore, the single hardware-dependent module of STORM, the underlying communication layer, is encapsulated in a small, isolated module (the final version actually included two different implementations of this layer, one for the Quadrics network and a generic one for any MPI platform.)

Some other important issues were not implemented in the current current version of STORM, but are high on our agenda for future versions:

1. **Security:** The system takes no special precautions to avoid rogue requests and does not check for access control rights. However, since it is run by the user in user-mode, using her own user-id and group-id, the scope of potential security violations is limited to that of any application the user might run.

2. **Reliability:** STORM contains the infrastructure and design for advanced fault-tolerance mechanisms. This issue will be addressed in the next version of STORM.

3. **Ease-of-Use:** The resource management system has a relatively simple command-line, scripts, and files interfaces, and offers no graphic user interface (GUI).

2.2 The Quadrics Network

The QsNET is based on two building blocks, a programmable network interface called Elan [26] and a low-latency high-bandwidth communication switch called Elite [27]. Elites can be interconnected in a fat-tree topology [18].

The Elan network interface links the high-performance, multi-stage Quadrics network to a processing node containing one or more CPUs. In addition to generating and accepting packets to and from the network, the Elan is equipped with a 32-bit thread processor, which is used to aid the implementation of higher-level messaging libraries without explicit intervention from the main CPU. In order to better support this implementation, the thread processor's instruction set is augmented with extra instructions that construct network packets, manipulate events, efficiently schedule threads, and block save and restore a thread's state when scheduling.

The other building block of the QsNET is the Elite switch. The Elite provides the following features: (1) 8 bidirectional links supporting two virtual channels in each direction, (2) an full crossbar switch, (3) a transmission bandwidth of 320 MB/s per link and a flow through latency of 35 ns, and (4) hardware support for collective communication. The Elite switches are interconnected in a quaternary fat-tree topology, which belongs to the more general class of the k-ary n-trees [23] Quaternary fat trees of dimension 1, 2 and 3 are shown in Figure 1.

2.2.1 Collective Communication

Packets can be sent to multiple destinations using the *hardware* multicast capability of the network. A multicast packet can only take a pre-determined path, in order to avoid deadlocks. In Figure 2 a) it is shown that the top leftmost switch is chosen as the logical root for the collective communication, and every request, in the ascending phase, must pass through one of the dotted paths until it gets to the root switch. In Figure 2 b) we can see how a multicast packet reaches the root node; the multiple branches are then propagated in parallel. All nodes connected to the network are capable of receiving the multicast packet, as long as the multicast set is physically contiguous.

For a multicast packet to be successfully delivered, a positive acknowledgment must be received from all the recipients of the multicast group. The Elite switches combine the acknowledgments, as pioneered by the NYU Ultracomputer [4] [24], returning a single one to the source. Acknowledgments are combined in a way that the "worst" ack wins (a network error wins over an unsuccessful transaction, which on its turn wins over a successful one), returning a positive ack only when all the partners in the collective communication complete the distributed transaction with success.

The main communication primitive of the QsNET is the remote DMA. A DMA operation transfers data between local and remote address spaces (including Elan memory). In addition to providing point-to-point communication, DMAs can also be used to perform group-wide operations such as broadcast. A group of destination processes is defined by specifying a virtual

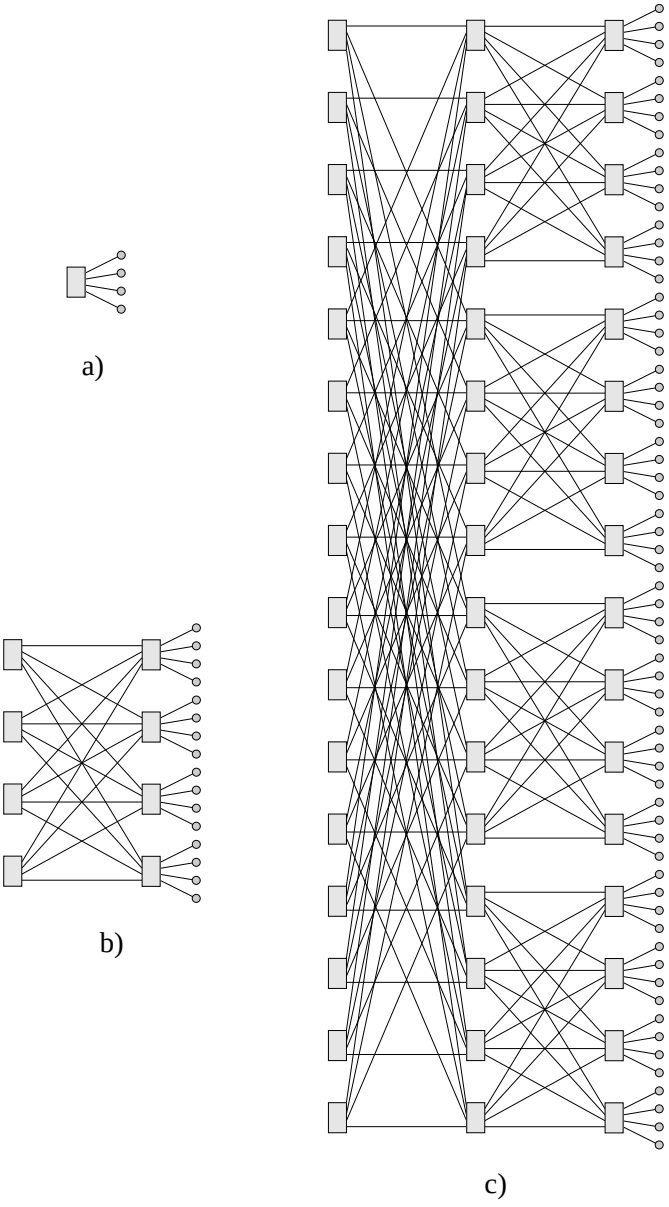

Figure 1: 4-ary n-trees of dimension 1, 2 and 3

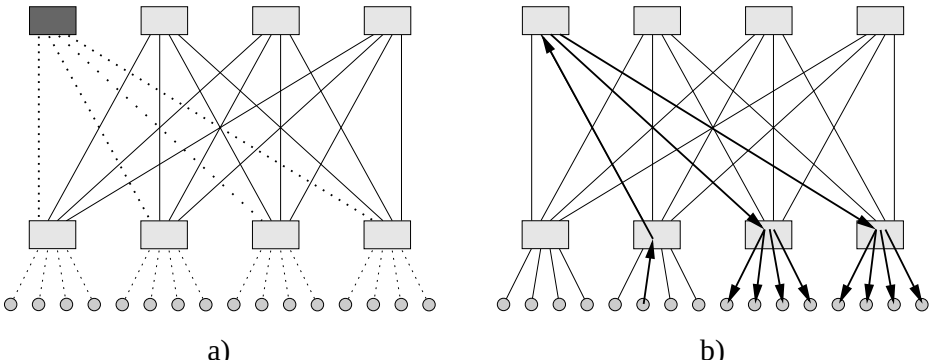

Figure 2: Hardware Multicast

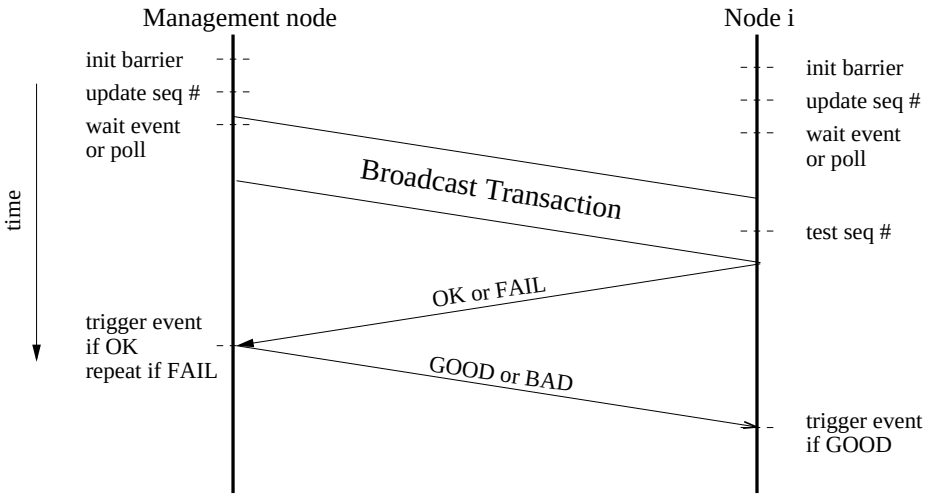

Figure 3: Strobe Implementation

group identifier. The effect of a write broadcast DMA is to copy the data from the source to the destination buffers of all the processes in the group. The implementation of the broadcast DMAs relies on all receiving processes having the destination buffer at the same virtual address, to obtain good performance.

Figure 3 describes how the strobing algorithm, the communication core of STORM, is implemented on top of the QsNET multicast capability. The management node can send a multicast message to all the nodes in the cluster, which opens a tree-shaped circuit between the source node and all the client nodes.

While the circuit is open, the management node can perform a set of atomic transactions on the client nodes. A typical transaction is to perform an atomic fetch-and-add or compare-and-swap on a memory location on all clients, combine the results through the network, and based on the global result, it can perform another transaction (up to 16 in the current implementation) and close the tree of circuits or abort the current transaction. In order to fork a job, the management node can ask whether all the nodes are ready to accept it, and if successful, it can fan out the binary to all the clients with a single message. An in-depth experimental evaluation of the network [22] shows that the broadcast bandwidth scales almost linearly with number of nodes, reaching an aggregate bandwidth that is linear with the number of destination nodes. The same infrastructure can also be used to notify the end of a time-slice in a gang scheduling or coscheduling algorithm.

2.3 Process Structure

STORM consists of three types of daemons that handle job launching, scheduling, and monitoring: the Machine Manager or MM (a single daemon on a management node), the Node Manager or NM (one daemon per compute node) and the Program Launcher or PL (several daemons per compute node).

The MM is in charge of resource allocation for jobs (both in space and time). Whenever a new job arrives, the MM queues it and tries to allocate PEs to it (using a buddy tree algorithm [11, 12]). If the scheduling policy allows for multiprogramming (e.g. GS), the PEs are allocated in any time slot that has enough available resources. After a successful allocation, the MM broadcasts a job-launch message to all the NMs, and those NMs on nodes that are allocated to the job will launch it when its time slot arrives (the handling is done asynchronously). One optimization that is readily implemented allows the MM to send the binary image of the program to the NMs prior to running it (the file is being sent only if it had not been previously sent, to avoid unnecessary communication). This optimization exploits the efficient hardware broadcast mechanism, which can disseminate a file of several megabytes in a fraction of a second to all the nodes, instead of the non-scalable use of NFS for distributing binaries. When a process of the job terminates, the MM receives an event from the appropriate NM, and marks the time/space resource occupied by that process as available for allocation. Note that even though the MM is centralized, in reality it does not create a bottleneck: since all the global operations it performs are done with scalable hardware broadcasts and other operation such as reading a new job, allocating resources to it, and receiving process-termination notifications are rare and lightweight. In fact, the MM sleeps between timeslice intervals, to maximize CPU availability, and only performs its operations when a new time slot is due.

NMs are responsible for managing resources on a single node (which could be an SMP). NMs work asynchronously, by responding to the following types of events:

- Job launch: If the job pertains to the NM's node, the NM finds some available PLs and sends them the job information.
- Job caching: The binary image is read from the communication layer and stored in a file, preferably in a RAM-disk file to avoid unnecessary I/O.
- Heartbeat/strobe: The NM checks in its local data structures, for every PE, whether another process occupies the next time slot. If so, it deschedules the current process (using UNIX's SIGSTOP) and resumes the next one.
- Process termination: upon receipt of such a message from the PL, the NM passes it on to the MM.

At all other times, the NMs block, leaving the PE to the application processes. Note that some scheduling algorithms require that the NM makes its own local scheduling decisions. For example, in local scheduling, the NM ignores context-switch messages, as the UNIX scheduler handles all scheduling decisions. In Algorithms such as BCS or ICS, the NM might deschedule a process that is blocked for communication before the expiration of the time slot, and schedule another process instead.

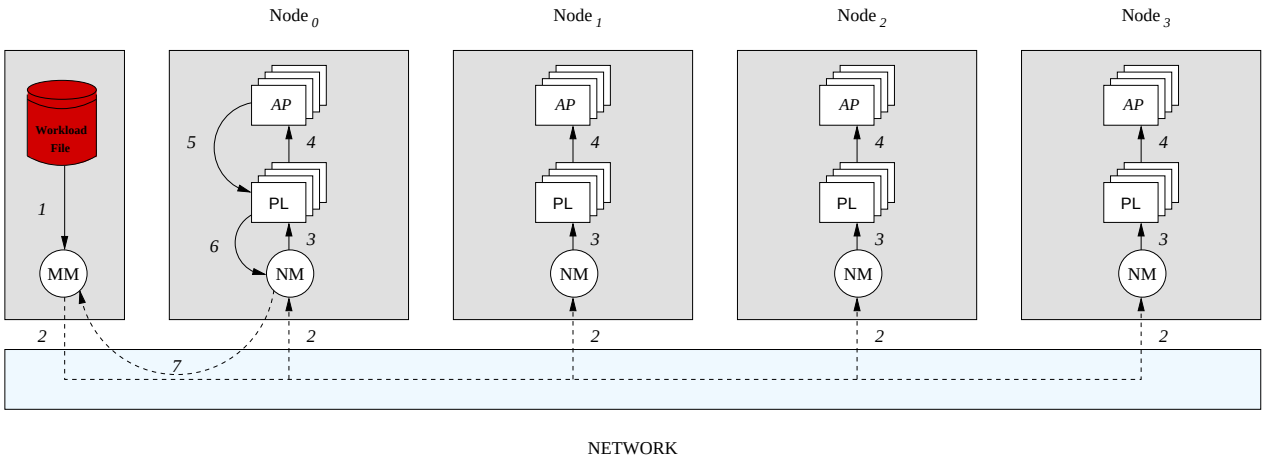

Figure 4: Running a job.

The PLs have the relatively simple task of handling individual application processes for the NM. One copy of the PL runs for each PE and timeslot in a node, and sleeps until it receives a program execution event from the NM. It then proceeds to fork a new process, set up Quadrics communication capabilities for the application process (AP), redirect standard output and error to the console that launched STORM, and execute the AP. It then blocks with the wait() system call until the AP terminates, and notifies the NM when this happens.

2.4 Running a Job

Figure 4 illustrates the process of running a job with STORM. In this example, we have a management node and four SMP nodes with two processing elements (PEs) each. The arrows show the information flow (dashed lines represent network messages), and the numbers on the arrows indicate the order of the events, according to the following key:

1. First, the MM reads the job information from the workload file, and queues it according to its given arrival time and resource availability.
2. When the job's time to run has come, and resources have been allocated, the MM broadcasts the job information (possibly with the binary image) to all the NMs. A future optimization might use the hardware multicast to send the information only to a subset of NMs. It is worth noting that this multicast is performed by a thread in the network interface card, without interrupting the main CPU and using an I/O by-pass mechanism, as described in Section 2.5 below.
3. If the NM needs to fork some processes, it locates the appropriate PLs (according to the job's PE/timeslot allocation). They then communicate the job information to the PL through shared memory.
4. The PLs execute the application processes (APs) as described in Section 2.3.
5. When an AP terminates, the PL receives a notification from the operating system.
6. The PLs then proceed to notify the NM of the termination of the AP.
7. Finally, the NMs send an asynchronous point-to-point message to the MM, which inspects these messages before issuing the next strobe, and deallocates resources.

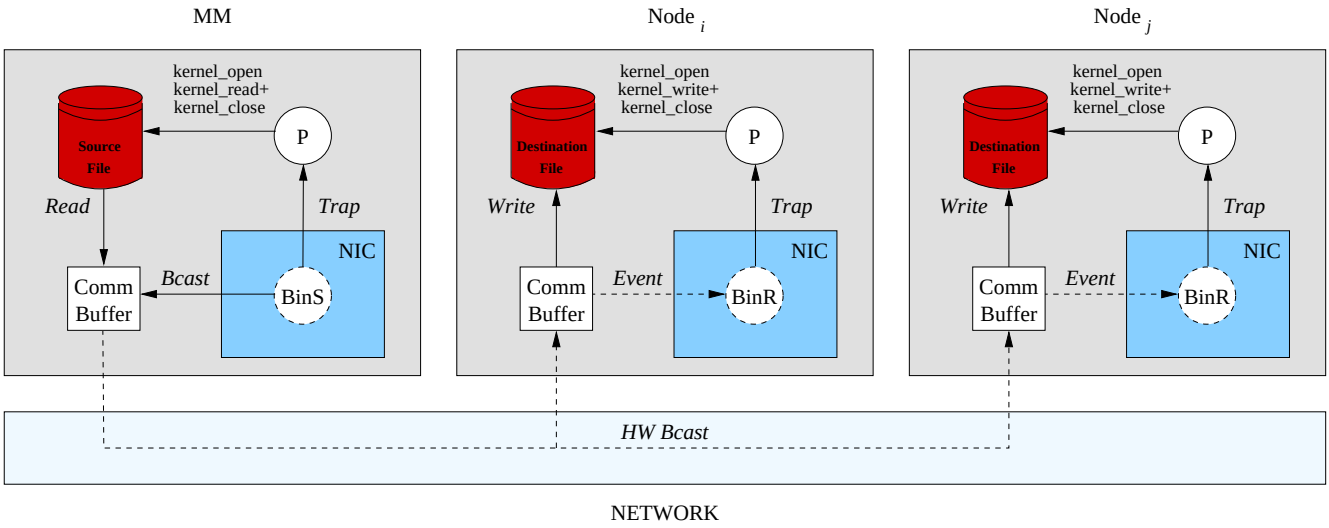

Figure 5: I/O bypass mechanism. kernel_read+ and kernel_write+ indicate sequences of kernel reads and writes. BinS and BinR are the Binary Sender and Receiver threads running on the Elan NIC.

2.5 I/O bypass mechanism

We implemented a mechanism for alleviating one of the major bottlenecks in program launching, the interaction with the I/O subsystem. The threads in the Elan network interface can directly issue system calls that operate on the file system, for example opening, reading, writing, and closing files. The relevant phases of the I/O by-pass protocol during the launch of a job are listed below (see Figure 5).

1. The MM sends a DMA message to a thread in the local Elan NIC with the source file name and a remote destination path.
2. The sender thread uses kernel traps to open and read the source file. These traps go through the kernel, but require very little CPU intervention, so that the processes running on the processing node are not unaffected.
3. The file is read in chunks directly into a communication buffer that can be efficiently accessed by the Elan DMA engine, and then sent to a peer thread on all the compute nodes, using the hardware multicast.
4. The sender thread uses two chunk buffers to pipeline the reading and multicast operations, so that while one buffer is being read, the other is being sent in parallel, as shown in Figure 6.
5. The destination threads on the compute nodes queue the incoming chunks and write them to the destination path, using a flow control protocol to avoid buffer overflows. File system writes and incoming multicasts can proceed in parallel.
6. When all the chunks have been sent and written to their respective local files, or conversely, if an error occurred, the MM is notified.
7. When the MM decides to launch the job (after successfully sending the binary and allocating resources to it), it uses the new remote path name in the job-launch message.

3 Experimental Results

In this section, we analyze the performance of STORM. In particular, we: (1) measure the costs of launching jobs in STORM, and (2) test various aspects of the gang-scheduler (effect of the timeslice quantum, node scalability and multiprogramming level).

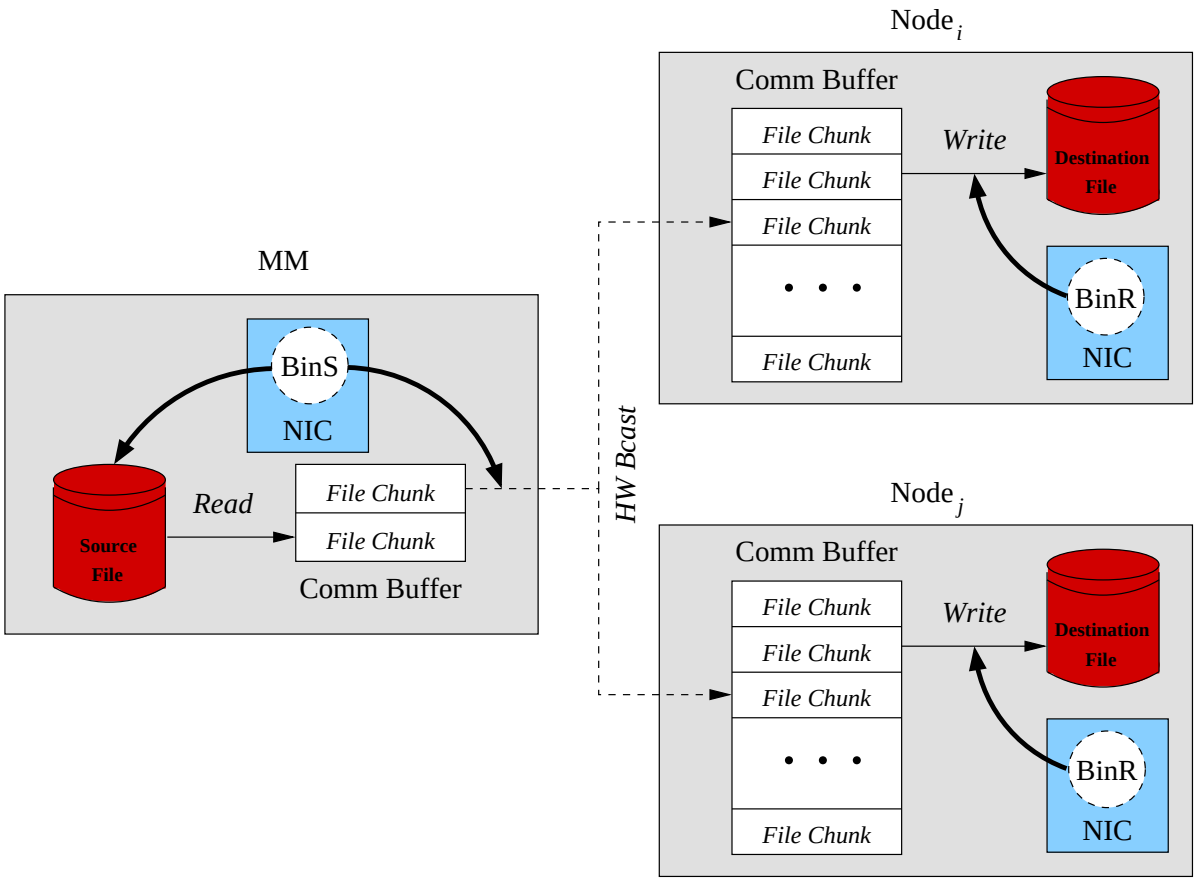

Figure 6: Pipelining of I/O read, hardware multicast and I/O write.

Main Processor			Thread Processor		
NFS	Local	RAM-disk	NFS	Local	RAM-disk
11.22	30.50	506.00	11.43	31.5	120

Table 1: Read bandwidth in MB/s for a $12MB$ binary image on NFS, local-disk, and RAM-disk .

3.1 Experimental Framework

The hardware used for the experimental evaluation was the 'crescendo' cluster at LANL/CCS-3. This cluster consists of 32 compute nodes (Dell 1550), one management node (Dell 2550) and the Quadrics network equipped with a 128-port switch [29] (using only 32 of the 128 ports). Each compute node has two 1.13 GHz Pentium-III with 1 GB of ECC RAM, two independent 66MHz/64-bit PCI buses, a Quadrics QM-400 Elan3 NIC [25, 26, 29] for data network and an Ethernet-100 network adapter for management network. All the nodes run under Red Hat Linux 7.1 with Quadrics kernel modifications and user-level libraries.

For the experiments in Sections 3.3.1-3.3.3, we used a small synthetic application that performs intensive CPU-bound computations for approximately 60 seconds without performing communication or I/O. This micro-benchmark was chosen to expose the performance of the scheduler only, without analyzing application scalability and other performance issues. In the tests that involve a multiprogramming level (MPL) of more than one, we launch all the jobs at the same moment (even though this may not be a realistic scenario), to further stress the scheduler.

3.2 Job Launching Time

In this set of experiments, we study the overhead associated with launching jobs with STORM, and analyze its scalability with the size of the binary size and number of PEs. We use the approach taken by Brightwell et al. in their study of job launching on Cplant [6]: measure the time it takes to run a program that terminates immediately, using different binary sizes: $4MB$, $8MB$, and $12MB$.

3.2.1 Anatomy of a job-launch

The time taken for execution of a parallel job can be broken down into the following four components:

- *Read Time:* the time taken by the management node to read the binary from the file system. This image can be read via a distributed filesystem like NFS, from a local disk, or it can be cached in RAM disk[5].Table 1 shows the read times of a 12 MB binary on a compute node. We distinguish the two cases when a process or an Elan thread try to read the file, in order to expose the performance of the I/O by-pass protocol. There is little difference between main and thread processors in the slow cases, namely NFS and local disk. But processes can take advantage of the RAM disks, getting more than 500 MB/sec, while the thread processor can only get 120 MB/sec. We still have not fully investigated this asymmetry, which is influenced by the fact that the thread processor resides on the slower, PCI bus.

- *Broadcast Time*: the time to broadcast the binary image to all the compute nodes. This collective communication can take place in several ways, thus affecting how the time is measured. For example, if the file is read via a distributed filesystem like NFS, the distribution time and file read time are intermixed. However, if a dedicated mechanism is used to disseminate the file, like ParPar's [17] or our own, this component can be measured separately from the others. QsNET can broadcast messages in a scalable way and there is no significant performance penalty when increasing the number of nodes. The typical performance for a main-memory-to-main-memory broadcast is around 200 MB/sec per node [22]. Figure 7 shows the scalability of the hardware multicast (Section 2.2.1), on the Terascale Computing System Installed at Pittsburgh Supercomputing Center, a cluster with 758 nodes. We can see that the latency grows by a negligible amount, about 2 μsec. This is a reliable indicator that the broadcast, implemented with the same hardware mechanism, will scale efficiently.

[5]The RAM disk is a segment of RAM that has been configured to simulate a Linux ext2 filesystem. RAM disk is expected to be faster than an actual disk drive.

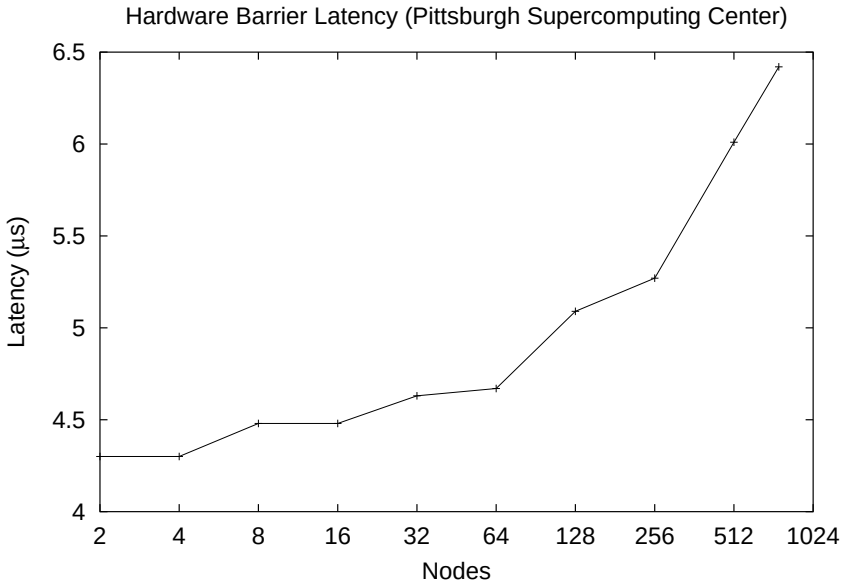

Figure 7: Barrier synchronization latency as a function of the number of nodes, Terascale computing system, Pittsburgh supercomputing center.

- *Write Time* : write time is less critical than read time because the file copy on the client nodes is followed by an *exec()* of the binary. Depending on how the kernel is implemented, part of the binary can reside on the buffer cache in memory at the time of the execution, and it doesn't need to flushed to disk.
- *Execution overhead*: some of the time for launching a job in STORM is spent in allocating resources, waiting for a new time slot to launch it, and possibly for another time slot to run it. Events such as process termination are also collected by the MM at timeslice intervals only, so a delay of up to 2 time-quanta is spent in STORM in MM overhead.

Our implementation tries to pipeline the three delays: read time, broadcast and write time, by dividing the file transmission in chunks of 128 KB. Table 1 shows that the bottleneck is the read time from disk in the management node, which is 118 MB/sec vs 200 MB/sec for the broadcast. Based on the scalability analysis reported in [22] and in Figure 7, we believe that this will be the bottleneck in large-scale (several hundreds of nodes) configurations too.

3.2.2 Launch times in STORM

As described in Section 2, STORM divides the job-launching task into two separate operations: The sending (broadcasting) of the binary image, which can be done before the designated time of running, and the actual execute, which includes sending the job-launch command, forking the job, waiting for its termination and reporting back to the MM. We measure the times of both this tasks on the MM, as well as their sum, which represents the total time it takes to launch a job. Figure 8 shows the time it takes to send each of the binaries, as well as the time to execute them and the total time to launch the job. Observe that the send times are roughly proportional to the binary size, but do not grow significantly with the number of nodes. This can be explained by the highly scalable hardware broadcast that is used for the send operation. On the other hand, the execution times are quite independent of the binary size, but grow slowly with the number of nodes. The reason for this growth is the cumulative time it takes for the MM to receive point-to-point notifications of the process termination from all the NMs. Since the increase in launch time is relatively small[6], we did not address this issue for this version of STORM, but intend to replace this mechanism with a more scalable one, using Quadrics' hardware support for collective communication.

[6] At this growth rate, it would still take to less than $350\,msec$ to launch a $12\,MB$ binary on $16\,K$ nodes.

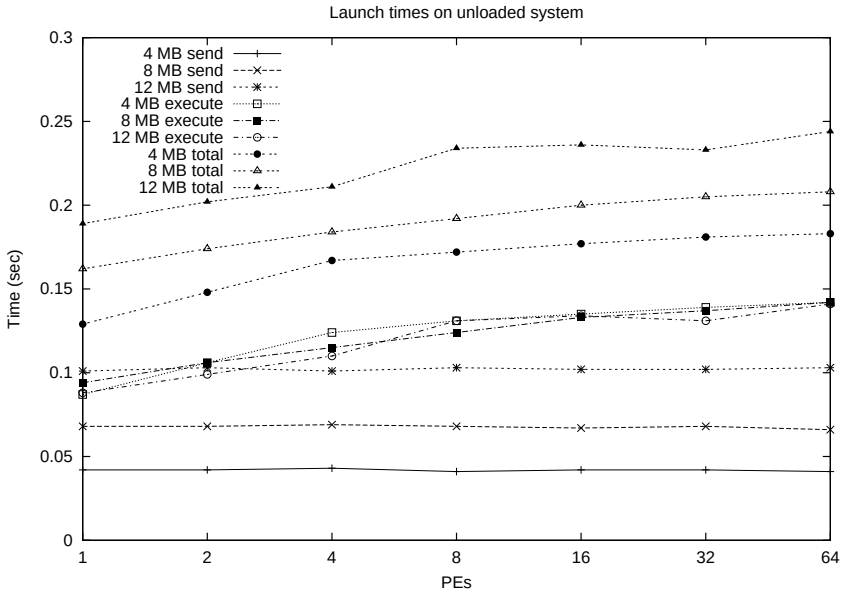

Figure 8: Send, execute and total launch times for 4MB, 8MB and 12MB files

3.2.3 Launching on a loaded system

To test how a heavily-loaded system affects the launch times of jobs, we added two different programs that run in the background on all the cluster's nodes while measuring job-launch times. The first program performs a CPU-intensive computation.

Figure 9 shows the results of launching the same three binaries while the CPU-consuming program is running in the background. We can see that while the execution times remain nearly unaffected by the system's load, the send times are approximately doubled. This large increase is mostly due to the interference of the computation with the I/O activities (reads and writes). However, the total launch time for all programs is still quite small, and it is less than twice the launch time on an unloaded system.

The second program is designed to stress the entire network, by pairing all the processes and continuously sending long messages back and forth. This test is particularly interesting, since a previous study [22] shows that a heavily-loaded network has an adverse effect on collective communications in the Quadrics interconnect. In Figure 10 we can see how running this program in the background affects the launch time of the test binaries. Indeed, there is a small, but noticeable, increase in the execution times, due mostly to the increased delay in the delivery of termination messages from the NMs. However, the send operation is considerably slower than on an unloaded system. This agrees with the previous study, since the communication part in the send operation is implemented by a Quadrics collective.

Figure 11summarizes the difference between the launch times on loaded and unloaded systems. In this figure, the total launch time is shown for the 12 MB file only, under the three loading scenarios. Note that even in the worst scenario, with a network-loaded system, it still takes only ≈ 0.6 seconds to launch a 12 MB file on 64 nodes, and the growth rate of about 3.5% on every doubling of the nodes, suggest it would take less than a second to launch this program on a $16,384$-node machine (assuming the same growth rate).

3.3 Gang Scheduling Performance

3.3.1 Effect of Timeslice

As a first experiment, we analyse the range of usable timeslice values, to better understand the limits of the gang-scheduler. Figure 12 shows the average runtime of the jobs for various timeslice values, up to 1.6 seconds, running on 64 PEs. The small-

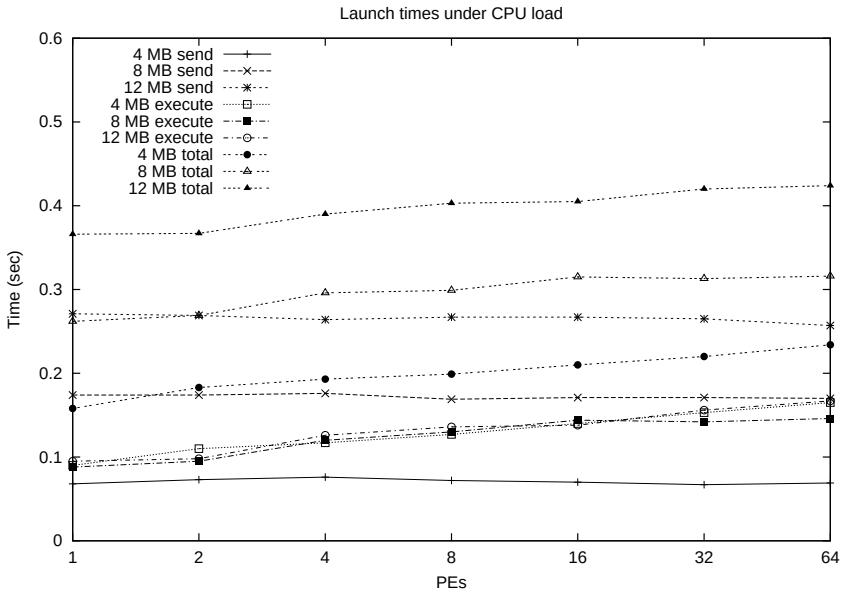

Figure 9: Send, execute and total launch times on a CPU-loaded system.

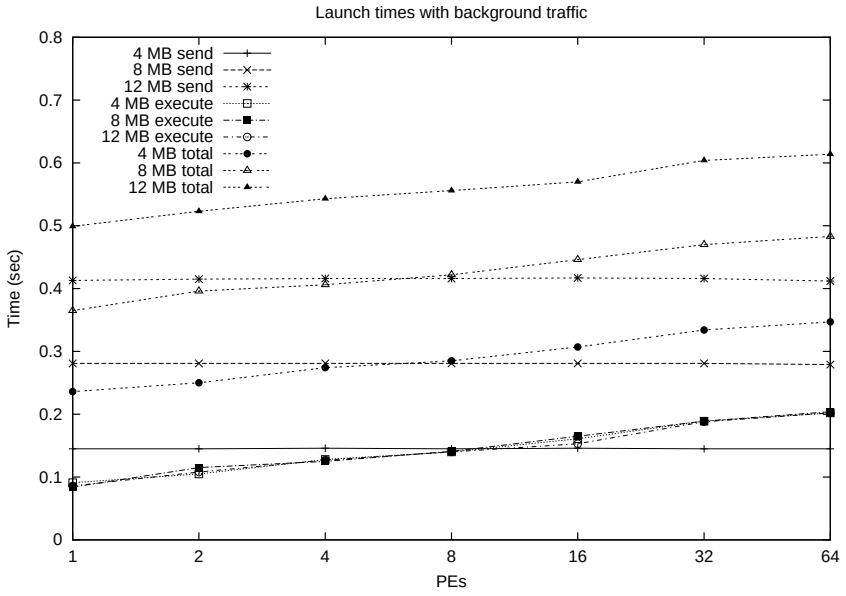

Figure 10: Send, execute and total launch times on a network-loaded system.

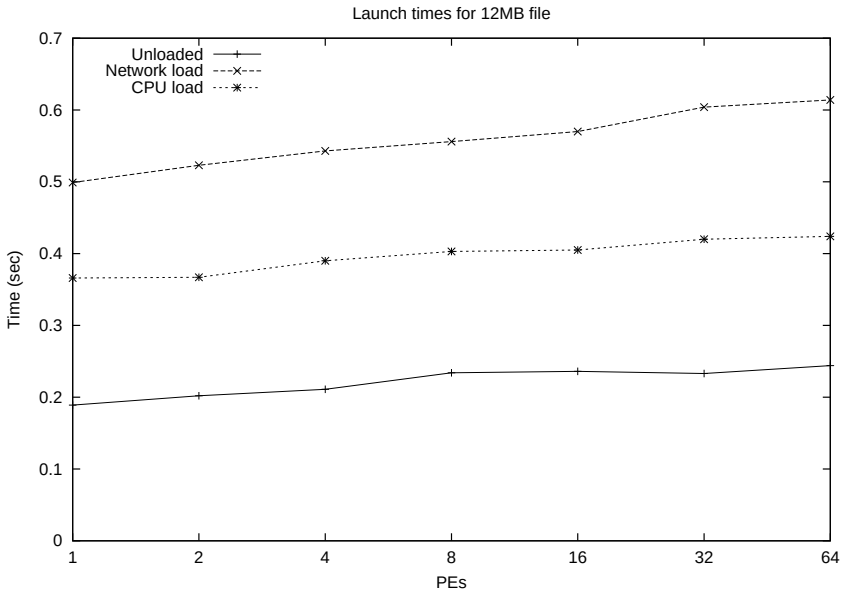

Figure 11: Total launch times for a 12MB binary

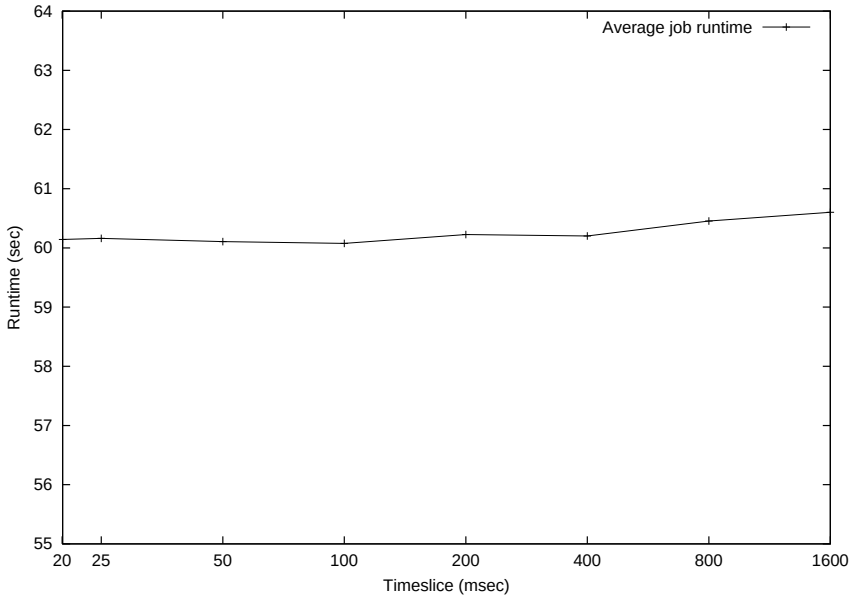

Figure 12: Effect of timeslice quantum with an MPL of 4, on 64 PEs

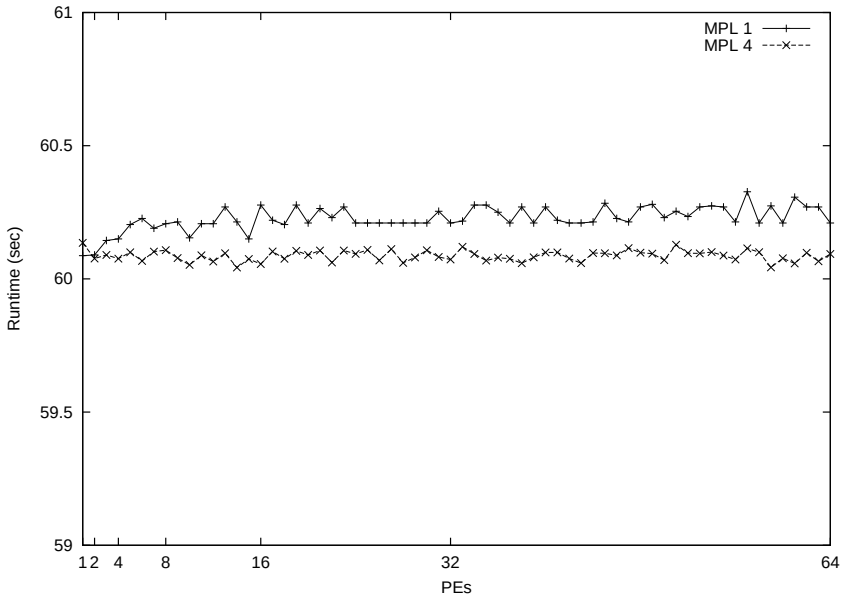

Figure 13: Effect of number of PEs on runtime, for MPL values of 1 and 4

est timeslice value that the scheduler can handle gracefully is $\approx 20msec$, below which the NM cannot process the incoming strobe message at the rate they arrive. Note that this value is in the same order of magnitude of the local Linux scheduler's quantum, and approximately two to three orders of magnitude better than the smallest timeslice quantum conventional gang-schedulers can handle with no performance penalties [15]. This allows for good system responsiveness, and usage of the parallel system for interactive jobs. Furthermore, a short quantum allows the implementation of advanced scheduling algorithms that can benefit greatly from short timeslice quanta, such as buffered coscheduling (BCS) [9, 10], implicit coscheduling (ICS) [1, 3], and periodic boost (PB) [19].

Another interesting feature is that the average runtime of the jobs is practically unchanged by the choice of timeslice quantum. There is a slight increase of less than one second toward the higher values, which is caused by the fact that events, such as process launch and termination reporting, only happen at timeslice intervals. Since the scheduler can handle small values of timeslice quanta with no performance penalty, we chose the value of $50msec$ for the next sets of experiments, providing a fairly responsive system.

3.3.2 Node Scalability

An important metric of a resource manager is the scalability with the number of nodes and PEs it manages. To test this, we measured the effect on the runtime of the program when running on an increasing number of nodes. This test only measures the effect on the runtime of the program, since it is relatively small. The effect on the launch-time of the program is measured in Section 3.2.

Figure 13 shows the results for running the program on varying number of PEs in the range 1-64, for MPL values of 1 and 4 (results for MPL 4 are normalized by dividing the total runtime of all jobs by 4). We can observe that the results are centered around two nearly-horizontal lines with relatively little variance. There is no increase in runtime or overhead with the increase in the number of nodes beyond that caused by the job-launch. This is explained by the fact that the only global operations in the system besides job-launching are strobes. These are implemented with a short hardware broadcast, that does not interrupt the main CPU. Since the Quadrics collective operations take a few microseconds to run even on thousands of nodes [21], we may expect the gang-scheduler to scale well to a large number of nodes.

The results for MPL 4 seem somewhat better than those of for MPL 1. This is a side effect of dividing the runtime by four, which also divides the constant overhead associated with job-launching. Also note that the results for 1 and 2 PEs are slightly

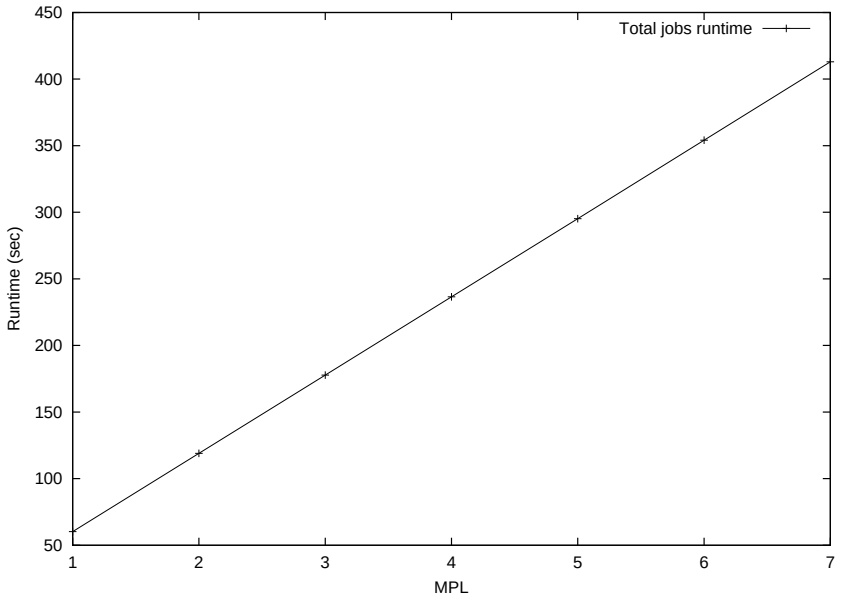

Figure 14: Effect of multiprogramming level on runtime

better with MPL 1. This stems from the fact than when only using one node, communication operations are replaced with inter-node shared-memory operations. Also note that all of the processes in the MPL 4 tests start and finish approximately at the same time, with a variance of less than 0.1%, implying a fair allocation of PEs to multiprogrammed processes.

3.3.3 Effect of MPL

Another important property of a gang-scheduler is the overhead incurred by the context switch operation. This context switch can cause penalties to the application due to loss of cache states, synchronization difficulties across nodes, and the need to change the communication context gracefully, including in-transit messages (in previous work, we have shown that applications can actually benefit from context switches on the Quadrics network, due to the overlapping of communication and computation from different jobs [14, 15]). The context switch operation in STORM is rudimentary, requiring very little computation to determine the next process to run, suspending the current process and resuming the next one. This is actually less work than the UNIX scheduler typically takes for a context switch [28] so we may expect it incurs a small overhead. To measure only the effect of the overhead incurred by the scheduler, we consider a test application that uses very little memory, I/O or bandwidth. In Figure 14 we can see the total runtime of running a varying number of jobs together, from 1 to 7. All jobs were launched concurrently and run on 64 PEs. It can be clearly seen that the curve is almost entirely straight, implying very little overhead resulting from running at higher multiprogramming levels.

4 Conclusions

In this paper we presented STORM, a lightweight, flexible and scalable environment to perform resource management in a large-scale cluster. With STORM we tried to prove the concept that it is possible to perform ultra-fast resource management with latencies well under a second even in the presence of high CPU utilization or network congestion. The paper provided a number of technical guidelines on how to achieve these goals.

This paper also explored in detail the performance of the gang-scheduling algorithm implemented as part of the system. We showed that by using a set of loosely-coupled daemons combined with fast hardware communication mechanisms, we can implement an extremely efficient scheduler. This scheduler can handle timeslice quanta of approximately $20\,msec$, providing

responsiveness similar to that of the local UNIX scheduler. The excellent scalability of the scheduler, both in terms of number of nodes and multiprogramming level, suggests that very little overhead is associated with resource management.

We demonstrated that STORM encompasses major advances in resource management by using three novel techniques: (1) employing NIC threads to relieve the resource-manager from most communication tasks, (2) relying on efficient hardware collectives to perform global operations in a scalable way, and (3) using an I/O bypass mechanism that minimize system overhead. The combination of these methods decreases the job-launch time by two orders of magnitude, thus making the system much more responsive and usable.

It is our hope that the flexibility inherent in the design of STORM will prepare the ground for new research results in the area of resource management and scheduling for large-scale parallel computers.

Future work

Our main venues of research include the implementation and testing of new scheduling algorithms, such as BCS, in order to address critical resource management issues such as reliability, load balancing and system utilization. One important point in this direction is the implementation of user-transparent fault-tolerance, that seamlessly allows applications to resume execution even when nodes fail. Another direction is the implementation of a flexible coscheduling algorithms that can increase system utilization in the presence of load-imbalance. We also plan to validate the scalability of STORM on larger clusters.

Acknowledgments

We thank David Addison and David Hewson for their timely and insightful advice in the design of the I/O bypass protocol. We also thank Duncan Roweth for the barrier scalability analysis on the Terascale computing system Installed at Pittsburgh supercomputing center.

References

[1] Andrea C. Arpaci-Dusseau, David Culler, and Alan M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the 1998 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[2] Andrea Carol Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, 19(3), 2001.

[3] Remzi Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, Amin Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of the 1995 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, Ottawa, Canada, May 1995.

[4] G. Bell. Ultracomputer: a Teraflop before its time. *Communications of the ACM*, 35(8):27–47, 1992.

[5] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawick, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, January 1995.

[6] Ron Brightwell and Lee Ann Fisk. Scalable Parallel Application Launch on Cplant. In *Supercomputing 2001*, Denver, CO, November 2001.

[7] Daniel Cassiday. Infiniband architecture tutorial. Hot Chips 12 Tutorial, August 2000.

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

[9] Fabrizio Petrini and Wu-chun Feng. Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000, IPDPS2000*, volume 16, Cancun, MX, May 2000.

[10] Fabrizio Petrini and Wu-chun Feng. Improved Resource Utilization with Buffered Coscheduling. *Journal of Parallel Algorithms and Applications*, 2000.

[11] Dror G. Feitelson. Packing Schemes for Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – Proceedings of the IPPS'96 Workshop*, volume 1162, pages 89–110. Springer, 1996.

[12] 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.

[13] Dror G. Feitelson and Morris A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261. Springer-Verlag, 1997.

[14] Eitan Frachtenberg and Fabrizio Petrini. Overlapping of Computation and Communication in the Quadrics Network. Technical Report LAUR 01-4695, Los Alamos National Laboratory, August 2001.

[15] Eitan Frachtenberg, Fabrizio Petrini, Salvador Coll, and Wu chun Feng. Gang Scheduling with Lightweigth User-Level Communication. In *2001 International Conference on Parallel Processing (ICPP2001), Workshop on Scheduling and Resource Management for Cluster Computing*, Valencia, Spain, September 2001.

[16] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Amin M. Vadhar. GLUnix: a GLobal Layer Unix for a Network of Workstations. *Software - Practice and Experience*, 28(9), 1998.

[17] Avi Kavas, David Er-El, and Dror G. Feitelson. Using Multicast to Pre-Load Jobs on the ParPar Cluster. *Parallel Computing*, 27:315–327, 2001.

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

[19] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita R. Das. A Closer Look At Coscheduling Approaches for a Network of Workstations. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures, SPAA'99*, Saint-Malo, France, June 1999.

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

[21] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[22] Fabrizio petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware-Based and Software-Based Collective Communication on the Quadrics Network. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, October 2001.

[23] Fabrizio Petrini and Marco Vanneschi. Performance Analysis of Wormhole Routed k-ary n-trees. *International Journal on Foundations of Computer Science*, 9(2):157–177, June 1998.

[24] G. F. Pfister and V. A. Norton. Hot-spot Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[25] Quadrics Supercomputers World Ltd. *Elan Programming Manual*, January 1999.

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

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

[28] Jeffrey H. Straathof, Ashok K. Thareja, and Ashok K. Agrawala. UNIX Scheduling for Large Systems. In *Proceedings of the USENIX Winter Conference*, pages 111–139, Denver, CO, 1986.

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