# STORM: Lightning-Fast Resource Management[*]

Eitan Frachtenberg Fabrizio Petrini Juan Fernandez Scott Pakin
Salvador Coll

CCS-3 Modeling, Algorithms, and Informatics Group
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory

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

July 26, 2002

**Abstract**

Although workstation clusters are a common platform for high-performance computing (HPC), they remain more difficult to manage than sequential systems or even symmetric multiprocessors. Furthermore, as cluster sizes increase, the quality of the resource-management subsystem—essentially, all of the code that runs on a cluster other than the applications—increasingly impacts application efficiency. In this paper, we present STORM, a resource-management framework designed for scalability and performance. The key innovation behind STORM is a software architecture that enables resource management to exploit low-level network features. As a result of this HPC-application-like design, STORM is orders of magnitude faster than the best reported results in the literature on two sample resource-management functions: job launching and process scheduling.

## 1 Introduction

Workstation clusters are steadily increasing in both size and popularity. Although cluster hardware is improving in terms of price and performance, cluster usability remains poor. Ideally, a large cluster should be as easy to develop for, use, and manage as a desktop computer. In practice, this is not the case. Table 1 contrasts desktops and clusters in terms of a few usability characteristics. In all of the cases listed desktop systems are easier to use than clusters. None of the cluster's shortcomings, however, are inherent: they are an artifact of deficient resource management software.

The reason that resource management software tends to perform inefficiently is that it has not previously been important to make it efficient. On a moderate-sized cluster, inefficient, nonscalable resource-management algorithms have little impact on overall cluster performance, so it is reasonable to divert more attention to application performance. However, with cluster sizes soon to reach 10,000 processors, resource management can no longer be ignored. Even a small amount of wasted

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

In Proceedings of the IEEE/ACM SC2002 conference, Baltimore, Maryland, November 16–22, 2002.

Table 1: Desktop vs. cluster usability

| Characteristic | Desktop | Cluster |
|---|---|---|
| Mean time between user-visible failures | Years | Days (for a large cluster) down to hours (for a very large cluster) |
| Scheduling | Timeshared | Batch queued or gang scheduled with large quanta (seconds to minutes) |
| Job-launching speed | < 1 second | Arbitrarily long (batch, including queue waiting time) or many seconds (gang scheduled) |

wall-clock time translates to a significant amount of wasted total CPU time. Furthermore, the performance lost to slow or nonscalable resource-management functions must be amortized by calling them as infrequently as possible. This degrades response time and hinders interactive jobs.

Our grand goal is for clusters to be as convenient to use as an individual workstation. In order to achieve this goal we set out to improve resource-management performance by several orders of magnitude over the current state of the art. Our vehicle for experimentation is a resource-management framework we developed called STORM (Scalable TOol for Resource Management). Section 2 of this paper describes the STORM architecture in detail and shows the division of labor among the various components. Essentially, STORM's uniqueness involves defining all resource-management functions in terms of a small set of mechanisms. By optimizing these mechanisms for a given cluster—in much the same way that one would optimize a high-performance application—STORM can achieve an immense performance and scalability benefit. Section 3 investigates this performance and scalability benefit in terms of two resource-management routines that were implemented within the STORM framework: job launching and gang scheduling. We discuss the implications of STORM's architecture and performance in Section 4. In Section 5, we compare STORM to related resource management systems. Finally, we draw some conclusions in Section 6.

## 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; and, (2) to support the implementation of most current and future job scheduling algorithms.

To fulfill the first goal we defined a small set of mechanisms upon which we based all of the resource-management functionality. Then, by optimizing just those mechanisms, we improved the performance and scalability of the rest of the system. For the second goal we implemented some loosely-coupled dæmons that divide up the tasks of managing a cluster, a node within a cluster, and a process within a node. By structuring these dæmons in a modular fashion, different functionality (e.g., coordinated process-scheduling algorithms, fault tolerance, or usage policies) can be "plugged" into them.

Table 2: STORM dæmons

| Dæmon | Distribution | Location |
|---|---|---|
| MM (Machine Manager) | One per cluster | Management node |
| NM (Node Manager) | One per compute node | Compute nodes |
| PL (Program Launcher) | One per potential process (# of compute nodes × # of processors per node × desired level of multiprogramming) | Compute nodes |

## 2.1 Process Structure

STORM consists of three types of dæmons that handle job launching, scheduling, and monitoring: the Machine Manager (MM), the Node Manager (NM), and the Program Launcher (PL). Table 2 describes the distribution and location of each of these.

The MM is in charge of resource allocation for jobs, including both space and time resources. Whenever a new job arrives, the MM enqueues it and attempts to allocate processors to it using a buddy tree algorithm [11, 12]. Global process-scheduling decisions are made by the MM. NMs are responsible for managing resources on a single node (which is typically an SMP). NMs are involved in finding available PLs for a job launch, receiving files transfered by the MM, scheduling and descheduling local processes, and detecting PL process termination. A PL has the relatively simple task of launching an individual application process. When its application process terminates, the PL notifies its NM.

## 2.2 STORM Mechanisms

For software-engineering purposes we partitioned STORM's code into three layers (Figure 1). The top layer is a set of functions that implement various resource-management functions. These are based on a small set of mechanisms which are portrayed as the middle layer. The idea is that if these middle-layer mechanisms are implemented in a scalable, efficient manner, then the top-layer functions will automatically inherit this scalability and efficiency. The middle-layer mechanisms are implemented directly atop whatever primitives are provided by the underlying network in the bottom layer. (In our initial implementation, this is the Quadrics network, (QsNET) [31].) The performance of the middle-layer mechanisms is a function not only of the bottom-layer primitives' performance, but also of how much "impedance matching" is required. For example, if the bottom layer does not provide hardware multicast, then this must be fabricated from point-to-point messages.

Our goals in designing the middle layer were *simplicity* and *generality*. We therefore defined our entire middle layer in terms of only three operations, which we nevertheless believe encapsulate all of the communication and synchronization mechanisms required by a resource-management system:

**XFER-AND-SIGNAL** Transfer (PUT) a block of data from local memory to the global memory of a set of nodes (possibly a single node). Optionally signal a local and/or a remote event upon completion.

**TEST-EVENT** Poll a local event to see if it has been signaled. Optionally, block until it is.

**COMPARE-AND-WRITE** Compare (using ≥, <, =, or ≠) a global variable on a set of nodes to a local value. If the condition is true on *all* nodes, then (optionally) assign a new value to a—possibly different—global variable.

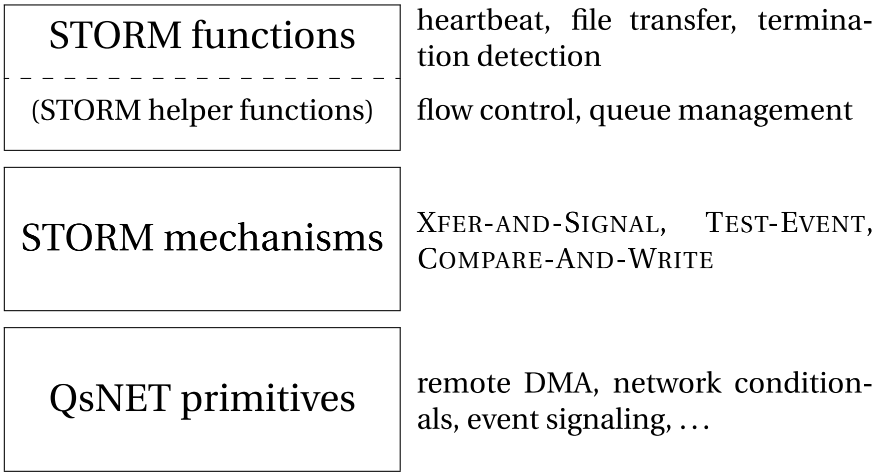
STORM functions
(STORM helper functions)
STORM mechanisms
QsNET primitives
heartbeat, file transfer, termination detection
flow control, queue management
Xfer-and-Signal, Test-Event, Compare-And-Write
remote DMA, network conditionals, event signaling, ...

Figure 1: STORM implementation structure

We call the preceding mechanisms the *STORM mechanisms* because they form the cornerstone of the STORM communication architecture. The following are some important points about the mechanisms' semantics:

1. *Global* data refers to data at the same virtual address on all nodes. Depending on the implementation global data may reside in main memory or network-interface memory.
2. Xfer-And-Signal and Compare-And-Write are both atomic operations. That is, Xfer-And-Signal either Puts data to *all* nodes in the destination set (which could be a single node) or—in case of a network error—*no* nodes. The same condition holds for Compare-And-Write when it writes a value to a global variable. Furthermore, if multiple nodes simultaneously initiate Compare-And-Writes with identical parameters except for the value to write, then, when all of the Compare-And-Writes have completed, all nodes will see the same value in the global variable. In other words, Xfer-And-Signal and Compare-And-Write are sequentially consistent operations [24].
3. Although Test-Event and Compare-And-Write are traditional, blocking operations, Xfer-And-Signal is non-blocking. The only way to check for completion is to Test-Event on a local event that Xfer-And-Signal signals.
4. The semantics do not dictate whether the STORM mechanisms are implemented by the host CPU or by a network coprocessor. Nor do they require that Test-Event yield the CPU (although not yielding the CPU may adversely affect system throughput).

We selected QsNET for our initial implementation because of its raw speed and the wealth of hardware primitives it provides to the bottom layer of our hierarchy: ordered, reliable multicasts; network conditionals (which return True if and only if a condition is True on *all* nodes); and events that can be waited upon and remotely signaled. In Section 4 we will discuss how the results shown in this paper generalize to other high-performance networks.

### 2.3 Applications of STORM

The dæmons described in Section 2.1 and the mechanisms described in Section 2.2 are sufficiently flexible to implement a variety of resource-management functions. For the purposes of this paper, viz. demonstrating STORM's performance and scalability, we limit ourselves to two particular functions: application launch and gang scheduling.

The traditional way to distribute an application from a fileserver to the cluster nodes is to have the cluster nodes demand-page the application binary from a shared filesystem, typically NFS [35]. This design, in which potentially many clients are simultaneously accessing a single file on a single server, is inherently nonscalable. Even worse, file servers are frequently unable to handle extreme loads and tend to fail with timeout errors. An alternative is to multicast the application binary from the fileserver to the cluster nodes. While this is the approach that we took in STORM, the challenges are that the multicast must be implemented in a scalable manner, and must implement global flow control because the cluster nodes may take different lengths of time to write the file. Our solution is to use a tree-based multicast (hidden within XFER-AND-SIGNAL's implementation) and to implement flow control on file fragments using COMPARE-AND-WRITE which can detect if all nodes have processed a fragment. To manage filesystem variability we double-buffer (actually, multi-buffer) the fragments so a node that is slow to write one fragment does not immediately delay the transmission of subsequent fragments. Finally, we utilize RAM disk-based filesystems so that file accesses proceed closer to memory speeds rather than disk speeds.

The uniqueness of our gang scheduler is that an efficient implementation of the STORM mechanisms enables global decisions to be made and enacted almost instantaneously. This lets us utilize much smaller time quanta than are typical in gang schedulers, which, in turn, makes it possible to run interactive parallel applications—something that is not possible with current gang-scheduling implementations.

# 3 Analysis

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 and node scalability).

We evaluated STORM on a 256-processor Alpha cluster with a Quadrics QsNET network and QM-400 Elan3 network interface cards (NICs) [31, 32, 33]. At the time of this writing, this cluster is rated as the world's 83rd fastest supercomputer [26]. Table 3 describes the cluster in more detail.

For most of the tests described in this section, we ran each experiment multiple times (3–20, depending upon the experiment) and took the mean of the measured performance. The variance was fairly small so the choice of mean over other statistics was largely immaterial. Unfortunately, the application experiments in Section 3.2 had—for as-yet unclear reasons—occasional slow runs, which biased the mean. As a result, we used the minimum time in that section; taking the median would have yielded essentially the same result.

## 3.1 Job Launching Time

In this set of experiments, we study the overhead associated with launching jobs with STORM and analyze STORM's scalability with the size of the binary and then number of PEs. We use the approach taken by Brightwell et al. in their study of job launching on Cplant [7]: we measure the time it takes to run a do-nothing program of size 4 MB, 8 MB, or 12 MB that terminates immediately.[1]

[1] The program contains a static array, which pads the binary image to the desired size.

Table 3: Cluster description

| Component | Feature | Type or number |
|---|---|---|
| Node | Number | 64 |
| | CPUs/node | 4 |
| | Memory/node | 8 GB |
| | I/O buses/node | 2 |
| | Operating system | Red Hat Linux 7.1<br>(+ Quadrics mods) |
| | Model | AlphaServer ES40 |
| CPU | Type | Alpha EV68 |
| | Speed | 833 MHz |
| I/O bus | Type | 64-bit PCI |
| | Speed | 33 MHz |
| Network | Type | QsNET |
| | Ports/switch | 128 |
| | NIC model | QM-400 Elan3 |

### 3.1.1 Launch times in STORM

STORM logically divides the job-launching task into two separate operations: The transferring (reading + broadcasting + writing +notifying the MM) of the binary image and the actual execution, which includes sending a job-launch command, forking the job, waiting for its termination, and reporting back to the MM. In order to reduce nondeterminism, the MM can issue commands and receive the notification of events only at the beginning of a timeslice. Therefore, both the binary transfer and the actual execution will take at least one timeslice. To minimize the MM overhead and expose maximal protocol performance, in the following job-launching experiments we use a small timeslice of 1 ms.

Figure 2 shows the time needed for each of these two tasks. The figure shows data for 1–256 processors and binary sizes of 4 MB, 8 MB, and 12 MB. Launch times are divided into the time it takes to transfer the binaries and the time needed to execute the (do-nothing) application. Observe that the send times are proportional to the binary size but grow very slowly with the number of nodes. This can be explained by the highly scalable algorithms and 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 more rapidly with the number of nodes. The reason for this growth is skew caused by local operating system scheduling effects.

In the largest configuration tested, a 12 MB file can be launched in 110 ms, a remarkably low latency. In this case, the average transfer time is 96 ms (a protocol bandwidth of 125 MB/s per node, with an aggregate bandwidth of 7.87 GB/s on 63 nodes[2]).

### 3.1.2 Launching on a loaded system

To test how a heavily-loaded system affects the launch times of jobs, we created a pair of programs that artificially load the system in a controlled manner. The first program performs a tight spin-loop, which introduces CPU contention. The second program repeatedly issues point-to-point messages

[2] The binary transfer does not include the source node.

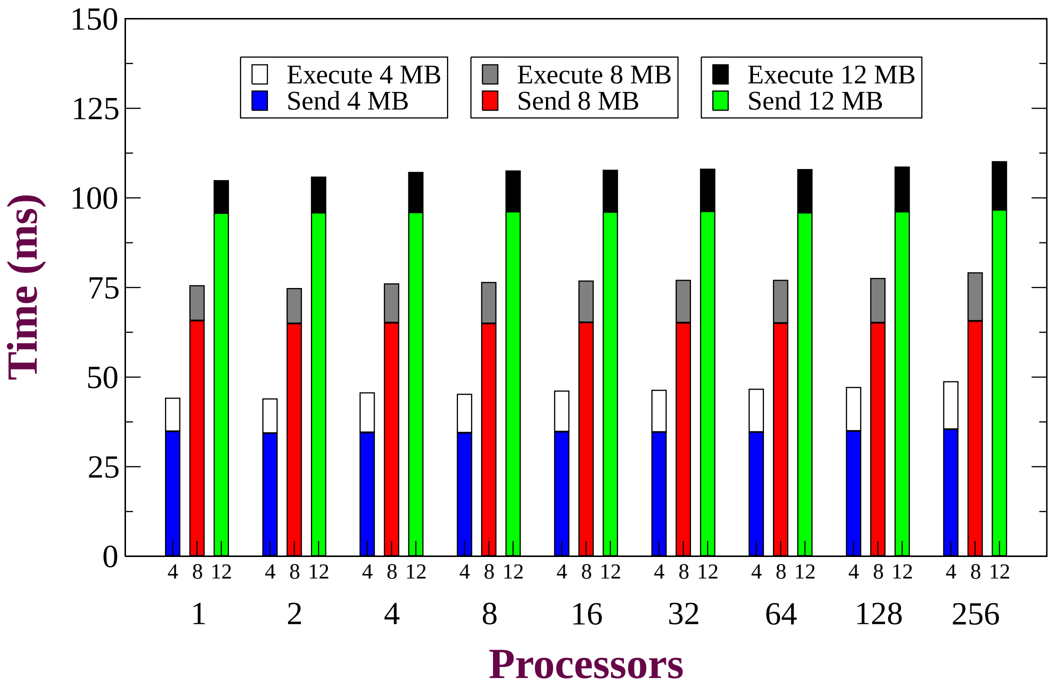

Figure 2: Send and execute times for a 4 MB, 8 MB, and 12 MB file on an unloaded system

between pairs of processes. This introduces network contention. Both programs were run on all 256 processors. The following experiments are the same as those used in Section 3.1.1 but with either the CPU-consuming program or the network-bandwidth-consuming program simultaneously running on all nodes of the cluster.

Figure 3 summarizes the difference among the launch times on loaded and unloaded systems. In this figure, the send and execute times are shown under the three loading scenarios (unloaded, CPU loaded, and network loaded), but only for the 12 MB file. Note that even in the worst-case scenario, with a network-loaded system, it still takes only 1.5 seconds to launch a 12 MB file on 256 processors.

### 3.2 Gang Scheduling Performance

Although STORM currently supports a variety of process scheduling algorithms—with more under development—we have chosen to focus our evaluation specifically on gang scheduling because it is one of the most popular coscheduling algorithms. Gang Scheduling (GS) can be defined to be a scheduling scheme that combines these three features: (1) processes in a job are grouped into gangs;

(2) the processes in each gang execute simultaneously on distinct PEs, using a one-to-one mapping; and,

(3) time slicing is used, with all the processes in a gang being preempted and rescheduled at the same time. Time slicing is obtained using a coordinated multi-context-switch, which occurs at regular intervals of time, called the timeslot quantum. The following are important issues regarding gang scheduling [28]:

**Effect of timeslice on overhead** Smaller timeslices yield better response time at the cost of decreased throughput (due to scheduling overhead that cannot be amortized). In Section 3.2.1, we show that STORM's scheduling overhead is so low that it can support workstation time quanta with virtually no performance penalty.

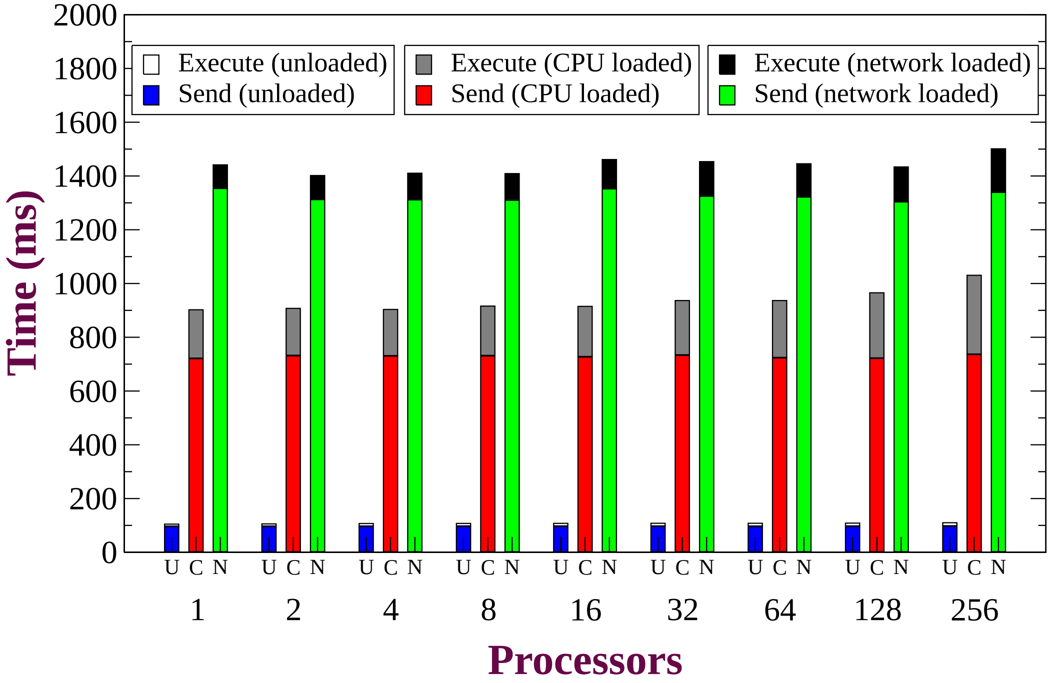

Figure 3: Send and execute times for a 12 MB file under various types of load

**Scalability** Because gang scheduling requires global coordination, the cost of enacting a global decision frequently increases with the number of processors. Section 3.2.2 demonstrates that STORM exhibits such good scalability that applications running on large clusters can be coscheduled almost as rapidly as small clusters.

The applications we use for our experiments in this section are a synthetic CPU-intensive job and SWEEP3D [23], a time-independent, Cartesian-grid, single-group, "discrete coordinates", deterministic, particle-transport code taken from the Accelerated Strategic Computing Initiative (ASCI) workload. SWEEP3D represents the core of a widely utilized method of solving the Boltzmann transport equation. Estimates are that deterministic particle transport accounts for 50–80% of the execution time of many applications that run on current Department of Energy (DOE) systems [20].

In 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. Results for MPL 2 are normalized by dividing the total runtime of all jobs by 2.

### 3.2.1 Effect of Time Quantum

As a first experiment, we analyzed the range of usable timeslice values to better understand the limits of STORM's gang scheduler. Figure 4 shows the average run time of the jobs for various timeslice values, from 300 $\mu$s to 8 seconds, running on 32 nodes/64 PEs. The smallest timeslice value that the scheduler can handle gracefully is $\approx$ 300 $\mu$s, below which the NM cannot process the incoming control messages at the rate they arrive. More importantly, with a timeslice as small as 2 ms, STORM can run multiple concurrent instances of an application with virtually no performance degradation over a single instance of the application.[3] This timeslice is an order of magnitude smaller than typ-

[3] This result is also influenced by the poor memory locality of SWEEP3D, so running multiple processes on the same processor does not pollute their working sets.

ical operating-system (OS) scheduler quanta, and approximately three-to-four orders of magnitude better than the smallest time quanta that 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 time quanta, such as buffered coscheduling (BCS) [29, 30], implicit coscheduling (ICS) [2, 3], and periodic boost (PB) [27].

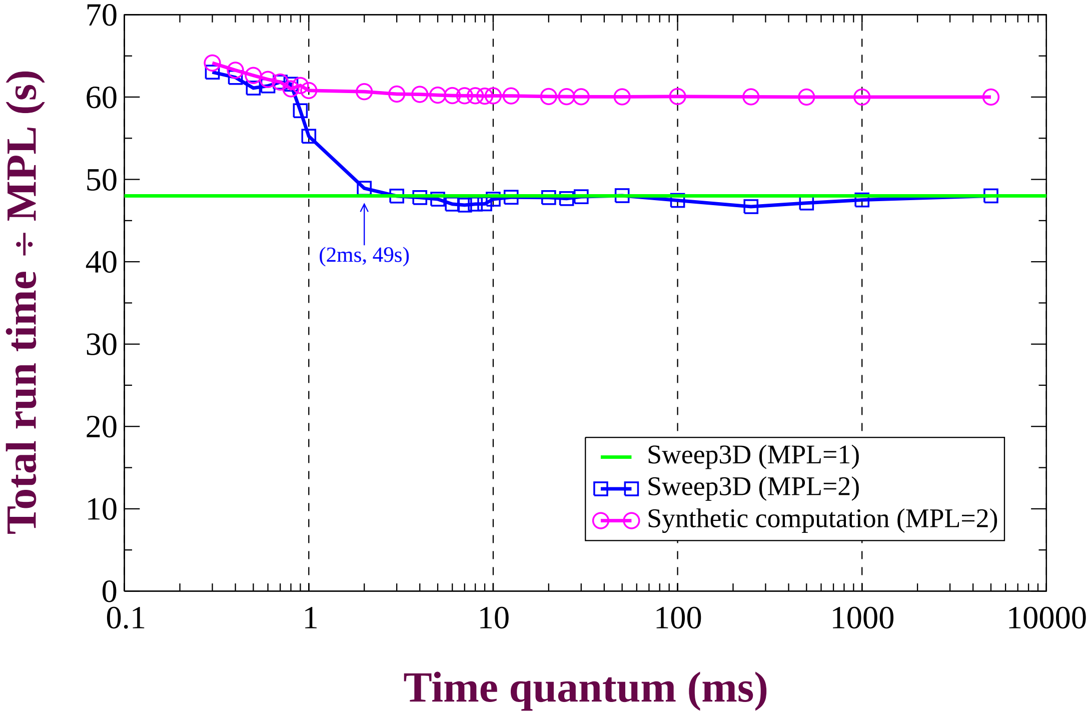

Figure 4: Effect of time quantum with an MPL of 2, on 32 nodes.

Another interesting feature is that the average runtime of the jobs is practically unchanged by the choice of time quantum. There is a slight increase of less than one second out of 50 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 time quanta with no performance penalty, we chose the value of 50 ms for the next sets of experiments, which provides a fairly responsive system.

### 3.2.2 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 program runtime as a function of the number of nodes. Figure 5 shows the results for running the programs on varying number of nodes in the range 1–64 for MPL values of 1 and 2. We can observe that there is no increase in runtime or overhead with the increase in the number of nodes beyond that caused by the job-launch.

## 3.3 Performance and Scalability Analysis

In this section, we analyze all of the components involved in the launching of a job on an unloaded system, and we present an analytical model showing how STORM's performance is expected to scale

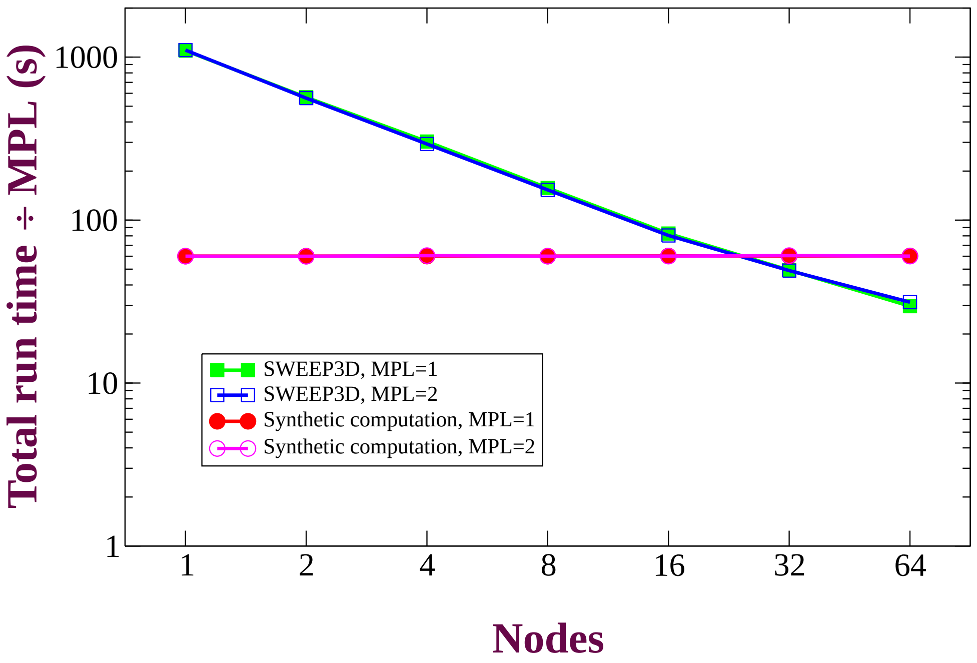

Figure 5: Effect of node scalability, varying the number of nodes in the range 1–64 for MPL values of 1 and 2.

to cluster configurations containing thousands of processing nodes.

### 3.3.1 Performance Analysis

The time needed to launch a parallel job can be broken down into the following components:

**Read time** This is the time taken by the management node to read the application binary from the file system. The image can be read from a distributed filesystem such as NFS [35], from a local hard disk, or from a RAM disk.[4] In our cluster the NIC can read a file directly from the RAM disk with an effective bandwidth of 218 MB/s.

Figure 6 shows the bandwidth achieved when the NIC—with assistance from a lightweight process on the host—reads a 12 MB file from various types of filesystems into either host- or NIC-resident buffers. As the figure shows, in the slow cases, namely NFS and local disk, it makes little difference whether the target buffers reside in main memory or NIC memory. However, when reading from a (fast) RAM disk, the data clearly show that keeping data buffers in main memory gives the better performance.

**Broadcast time** This is the time to broadcast the binary image to all of the compute nodes. If the file is read via a distributed filesystem like NFS, which supports demand paging, the distribution time and the file read time are intermixed. However, if a dedicated mechanism is used to distribute the file, as in ParPar [22] or STORM, broadcast time can be measured separately from the other components of the total launch time. The QsNET's hardware broadcast is both scalable and extremely fast. On the ES40 Alphaserver, the performance for a main-memory-to-main-memory broadcast is therefore limited by the PCI I/O bus. Figure 7 shows the bandwidth

[4] A RAM disk is a segment of RAM that has been configured to simulate a disk filesystem. This provides better performance than mechanical media but at increased cost, as DRAM is more expensive than disk media for a given capacity.

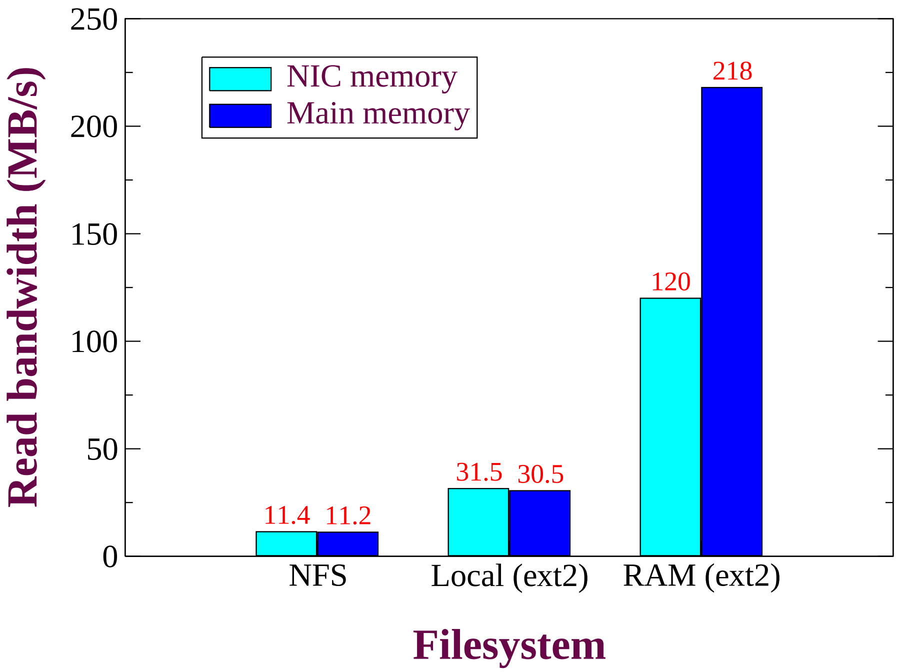

Figure 6: Read bandwidth for a 12 MB binary image from NFS, a local hard disk, and a local RAM disk, with buffers placed in NIC and main memory

delivered by the broadcast on 64 nodes when both source and destination buffers are placed in NIC (thereby bypassing the PCI bus) and main memory, respectively. As the figure shows, the hardware broadcast can deliver 312 MB/s when the buffers are in NIC memory but only 175 MB/s when the buffers are placed in main memory.

**Write time** We are concerned primarily with the overhead component of the write time. It does not matter much if the file resides in the buffer cache or is flushed to the (RAM) disk. A number of experiments—for brevity not reported here—show that the read bandwidth is consistently lower than the write bandwidth. Thus the write bandwidth is not the bottleneck of the file-transfer protocol.

**Launch and execution overhead** Some of the time needed to launch a job in STORM is spent allocating resources, waiting for a new time slot in which to launch the job, and possibly waiting for another time slot in which to run it. In addition, events such as process termination are collected by the MM at heartbeat intervals only, so a delay of up to 2 heartbeat quanta can be spent in MM overhead.

Our implementation tries to pipeline the three components of file-transfer overhead—read time, broadcast time, and write time—by dividing the file transmission into fixed-size chunks and writing these chunks into a remote queue that contains a given number of slots. In order to optimize the overall bandwidth of the pipeline, $BW_{launch}$, we need to maximize the bandwidth of each single stage. $BW_{launch}$ is limited by the bandwidth of the slowest stage of the pipeline, which implies that

$$BW_{launch} \leq \min(BW_{read}, BW_{broadcast}, BW_{write}) = \min(BW_{read}, BW_{broadcast}) \quad . \tag{1}$$

As previously stated, the buffers into which data is read and from which data is broadcast can reside in either main memory or NIC memory. Figure 6 showed that reading into main

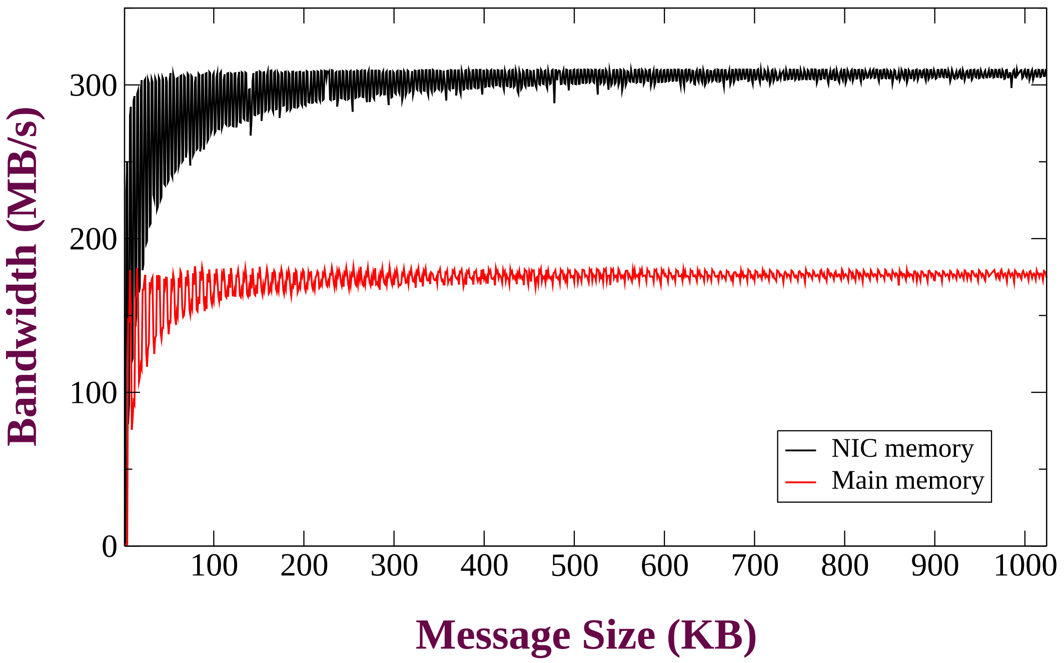

Figure 7: Broadcast bandwidth from NIC- vs. host-resident buffers

memory is faster, while Figure 7 showed that broadcasting from NIC memory is faster. The preceding inequality dictates that the better choice is to place the buffers in main memory, as $\min(BW_{read}, BW_{broadcast}) = \min(218\,\text{MB/s}, 175\,\text{MB/s}) = 175\,\text{MB/s}$ when the buffers reside in main memory, versus $\min(BW_{read}, BW_{broadcast}) = \min(120\,\text{MB/s}, 312\,\text{MB/s}) = 120\,\text{MB/s}$ when they reside in NIC memory.

We empirically determined the optimal chunk size and number of multi-buffering slots (i.e., receive-queue length) for our cluster based on the data plotted in Figure 8. That figure shows the total send time for the cross-product of {2, 4, 8, 16} slots and {32, 64, 128, 256, 512, 1024}-kilobyte chunks. The communication protocol is almost insensitive to the number of slots, and the best performance is obtained with four slots of 512 KB. Increasing the number of slots does not provide any extra performance because doing so generates more TLB misses in the virtual memory hardware of the NIC.

Figure 2 on page 7 showed that the transfer time of a 12 MB binary is about 96 ms. Of those, 4 ms are owed to skew caused by OS overhead and the fact that STORM dæmons act only on heartbeat intervals (1 ms). The remaining 92 ms are determined by a file-transfer-protocol bandwidth of about 131 MB/s. The gap between the previously calculated upper bound, 175 MB/s, and the actual value of 131 MB/s, is due to unresponsiveness and serialization within the lightweight process running on the host, which services TLB misses and performs file accesses on behalf of the NIC.

### 3.3.2 Scalability Analysis

Because all STORM functionality is based on three mechanisms—COMPARE-AND-WRITE, XFER-AND-SIGNAL, and TEST-EVENT—the scalability of these primitives determines the scalability of STORM as a whole. In fact TEST-EVENT is a local operation, so scalability is actually determined only by the remaining two mechanisms.

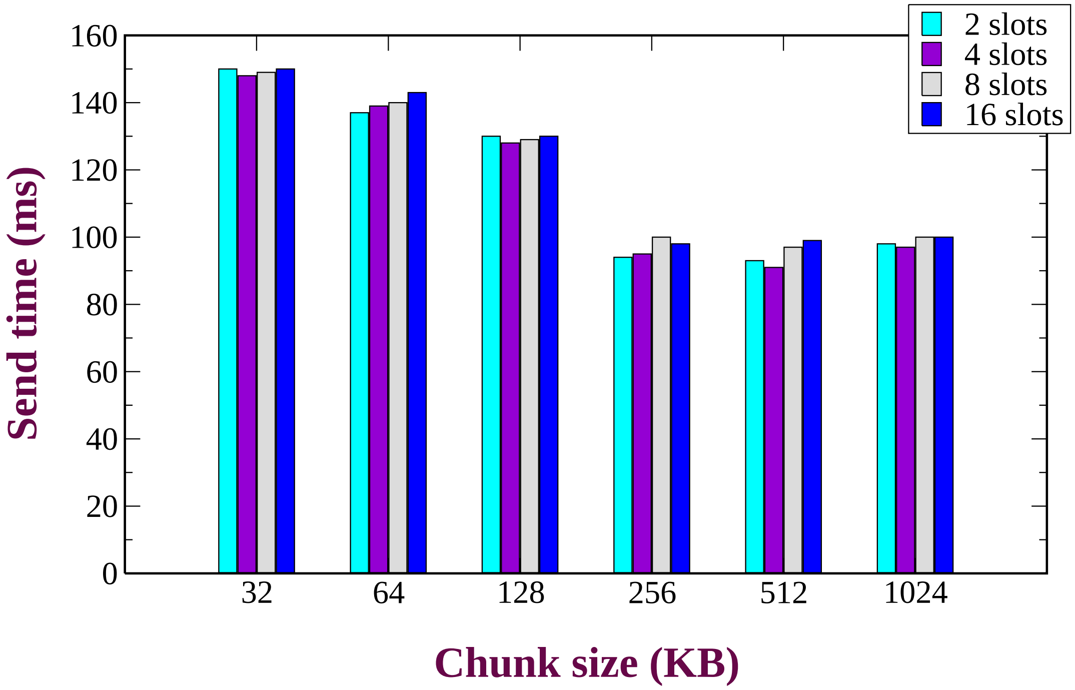

Figure 8: Send time as a function of chunk size and slot count

**Scalability of COMPARE-AND-WRITE** Figure 9 shows the scalability of QsNET's hardware barrier synchronization (on which COMPARE-AND-WRITE is based) on the Terascale Computing System [38] installed at the Pittsburgh Supercomputing Center, a cluster with 768 nodes/3,072 processors but otherwise similar to our cluster. We can see that the latency grows by a negligible amount—about 2 $\mu$s—across a 384X increase in the number of nodes. This is a reliable indicator that COMPARE-AND-WRITE, when implemented with the same hardware mechanism, will scale comparably. In fact, the flow-through latency of each switch of the QsNET is about 35 ns plus the wire delay, so in a network with 4,096 nodes, each broadcast message must cross at most 11 switches in the worst case, leading to a network latency of less than a microsecond.

**Scalability of XFER-AND-SIGNAL** In order to determine the scalability of XFER-AND-SIGNAL to a large number of nodes, we need to carefully evaluate the communication performance of the hardware broadcast, and consider details of the hardware flow control in the network, which take into account the wire and switch delays. The QsNET network transmits packets with circuit-switched flow control. A message is chunked into packets of 320 bytes of data payload, and the packet with sequence number $i$ can be injected into the network only after the successful reception of the acknowledgment token of packet $i-1$. On a broadcast, an acknowledgment is received by the source only when all of the nodes in the destination set have successfully received the packet. The maximum transfer unit of the QsNET network is only 320 bytes.[5] Hence, in the presence of long wires and/or many switches, the propagation delay of the acknowledgment token can introduce a bubble in the communication protocol's pipeline and therefore a reduction of the asymptotic bandwidth.

The following model is currently used in the procurement of the ASCI Q machine under development at Los Alamos National Laboratory, and has been validated on several network configurations, up to 1,024 nodes, with a prediction error of less than 5%. In order to make the $BW_{broadcast}$ dependent

[5] This limitation does not apply to version 4, which allows packets of variable length.

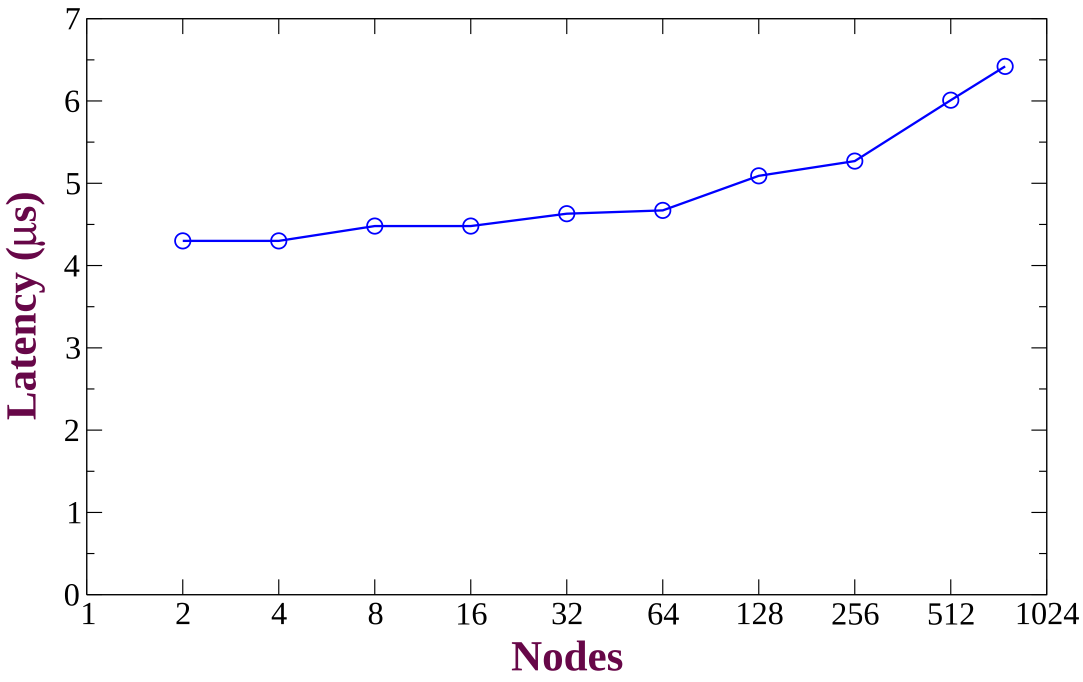

Figure 9: Barrier synchronization latency as a function of the number of nodes, Terascale Computing System, Pittsburgh Supercomputing Center

Table 4: Bandwidth scalability (MB/s)

| | | | | Cable length (m) | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Nodes | Processors | Stages | Switches | 10 | 20 | 30 | 40 | 60 | 80 | 100 |
| 4 | 16 | 1 | 1 | **319** | 319 | 319 | 319 | 284 | 249 | 222 |
| 16 | 64 | 2 | 3 | **319** | 319 | 309 | 287 | 251 | 224 | 202 |
| 64 | 256 | 3 | 5 | **312** | 290 | 270 | 254 | 225 | 203 | 185 |
| 256 | 1,024 | 4 | 7 | 273 | **256** | 241 | 227 | 204 | 186 | 170 |
| 1,024 | 4,096 | 5 | 9 | 243 | 229 | 217 | **206** | 187 | 171 | 158 |
| 4,096 | 16,384 | 6 | 11 | 218 | 207 | 197 | 188 | 172 | **159** | 147 |

on a single parameter—the number of nodes—we compute a conservative estimate of the diameter of the floor plan of the machine, which approximates the maximum cable length between two nodes. We assume that the parallel computer and the network are arranged in a square configuration. Considering that with current technology we can stack between four and six ES40 Alphaserver nodes in a single rack with a footprint of a square meter, we estimate the floor space required by four nodes to be 4 $m^2$ (1 $m^2$ for the rack surrounded by 3 $m^2$ of floor space). Conservatively

$$\text{diameter(nodes)} = \left\lfloor \sqrt{2 \times \text{nodes}} \right\rfloor \quad , \tag{2}$$

where the network diameter is expressed in meters. Table 4 describes how the asymptotic bandwidth, $BW_{broadcast}$, is affected by the network size for networks with up to 4,096 nodes and physical diameters of up to 100 meters. The worst-case bandwidth for each network size is shown in boldface in the table.

**Scalability of the Binary Transfer Protocol** We now consider a model of the launch time for a binary of 12 MB. The model contains two parts. The first part represents the actual transmission time and is inversely proportional to the available bandwidth for the given configuration. The second part is the local execution time of the job, followed by the notification to the MM, and the time wasted in OS overhead and waiting for the end of the STORM timeslices. Combining these two parts, the model states that

$$T_{launch}(nodes) = \frac{12}{BW_{transfer}(nodes)} + T_{exec} \quad . \tag{3}$$

We now apply this model to two node configurations. The first one, in which

$$BW^{ES40}_{transfer}(nodes) = \min(131, BW_{broadcast}(nodes)) \quad , \tag{4}$$

represents our current cluster, based on ES40 Alphaservers that can deliver at most 131 MB/s over the I/O bus. The second configuration, in which

$$BW^{ideal}_{transfer}(nodes) = BW_{broadcast}(nodes) \quad , \tag{5}$$

represents an idealized Alphaserver cluster that is limited by the network broadcast bandwidth (i.e., the I/O bus bandwidth is greater than the network broadcast bandwidth).

Figure 10 shows measured launch times for network configurations up to 64 nodes and estimated launch times for network configurations up to 16,384 nodes. The model shows that in an ES40-based Alphaserver the launch time is scalable and only slightly sensitive to the machine size. A 12 MB binary can be launched in 135 ms on 16,384 nodes. The graph also shows the expected launch times on an ideal machine in which the I/O bus is not the bottleneck (and in which a lightweight processes on the host can responsively handle the requests of the NIC). Both models converge with networks larger than 4,096 nodes because for such configurations they share the same bottleneck—the network broadcast bandwidth.

## 4 Discussion

Although much attention is paid to the performance of *applications* running on high-performance computer systems, the performance of resource-management software has largely been neglected. Even though resource-management performance is not critical on today's moderate-sized clusters, larger clusters and new usage models will require resource-management environments that are faster, more scalable, and easier to apply to a variety of tasks. With STORM, we are approaching this challenge proactively by basing all resource-management functionality on a sufficiently small set of functions that hardware-specific optimizations are feasible to implement—and reimplement when computers or networks are upgraded or replaced.

**Job launching** Although job-launching time seldom dominates the total run time of a real application, long launching times can be frustrating to programmers who want quick turnaround times while debugging their applications. This quick turnaround, however, requires efficient, scalable resource management. By providing the requisite performance– an order of magnitude faster than the best reported result [19] on a comparably sized cluster—STORM makes it feasible to perform interactive development on a large-scale cluster. Without a system like STORM, interactive development is practical only on small clusters, which may be insufficient for exposing race conditions or handling real data sets.

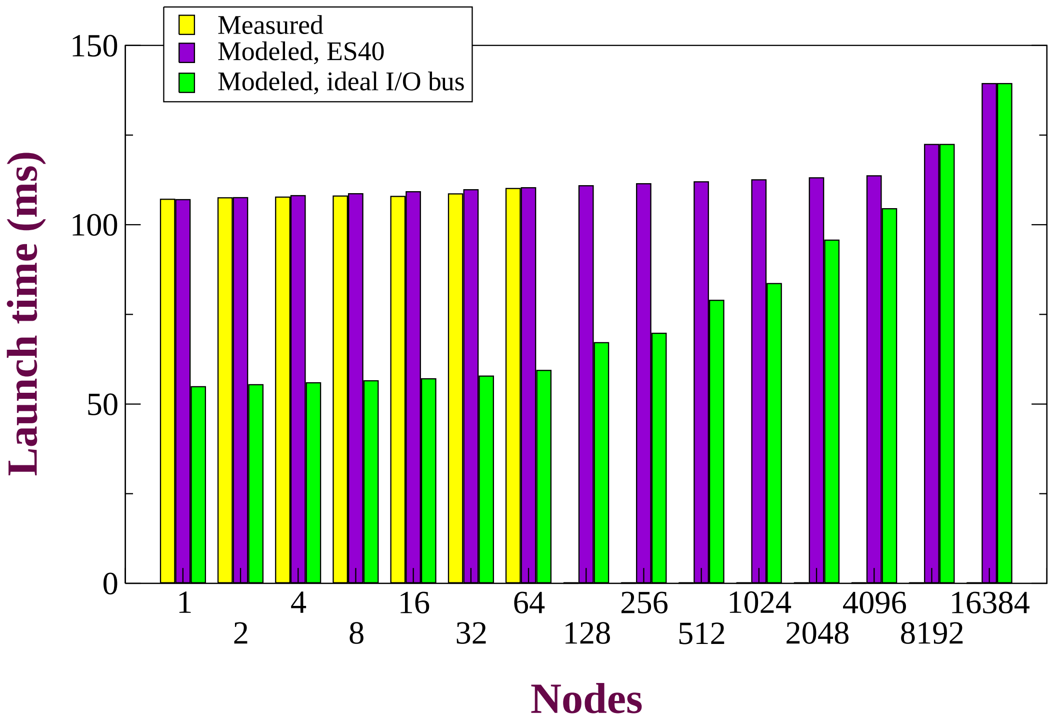

Figure 10: Measured and estimated launch times

**Process scheduling** Although most large cluster installations use batch queues to schedule applications, gang schedulers have the advantage that jobs do not have to wait minutes—or even hours—to start running [37]. However, because of the overhead incurred by enacting scheduling decisions, gang schedulers are usually run with large scheduling quanta—on the order of seconds or even minutes [16]—to amortize this overhead. Unfortunately, such long quanta hinder interactive jobs. In order to open up clusters to a wider breadth of applications and usage models, STORM seeks to minimize, rather than amortize, scheduling overhead. STORM can schedule a job on an entire cluster at about the same speed that a single node OS can schedule a process on an individual computer. As a result of this work clusters may soon be able to run parallel versions of applications that are currently limited to sequential workstations.

**Collective communication** There are two important observations about collective communication that led to the development of STORM: (1) resource management in a cluster environment is *inherently* collective, and (2) collective operations can be made fast and efficient by taking advantage of support provided by the network hardware. We therefore designed the STORM mechanisms to operate on *sets* of nodes. The collective-communication routines—primarily multicast and reduce—can thereby be implemented as efficiently as possible for a given platform.

**Portability of the STORM Mechanisms** Table 5 shows the expected performance of the STORM mechanisms as a function of the number of nodes $n$ on four high performance networks other than QsNET, namely Gigabit Ethernet, Myrinet, Infiniband and BlueGene/L, based on the best performance reported in the literature. The performance of COMPARE-AND-WRITE is expressed as the latency to check a global condition and to write a single word to all of the destinations. With XFER-AND-SIGNAL we consider the asymptotic bandwidth delivered to all nodes which in the optimal case should scale linearly with the number of nodes. In some of these networks (Ethernet, Myrinet and Infiniband) the STORM mechanisms need to be emulated through a thin software layer, while in the other networks there is a one-to-one mapping with existing hardware mechanisms. For the networks that need emulations we can use logarithmic-time (in the number of nodes) algorithms that organize the processing nodes in a tree. The broadcast implementations on Myrinet, in order to be general purpose, need to solve complex problems like buffer management, potential deadlocks, and congestion-free mapping of the logical tree in the presence of multiple concurrent broadcasts [5, 8, 9]. These problems are greatly simplified in STORM because these mechanisms are issued by a single node, so we think that the implementation of COMPARE-AND-WRITE can substantially improve the results. The Infiniband standard includes hardware support for multicast [10], but at the time of this writing there are neither functional descriptions of how it will be implemented nor experimental results available in the public domain. BlueGene/L provides a dedicated "global tree" network that implements one-to-all broadcast and arithmetic reduce operations in the tree itself, so the STORM mechanisms can be implemented very efficiently without any extra software layer [18].

We argue that in both cases—with or without hardware support—the STORM mechanism represent an ideal abstract machine that on the one hand can export the raw performance of the network, and on the other hand can provide a general-purpose basis for designing simple and efficient resource managers.

Table 5: Measured/expected performance of the STORM mechanisms

| Network | COMPARE-AND-WRITE ($\mu$s) | XFER-AND-SIGNAL (MB/s) |
|---|---|---|
| Gigabit Ethernet [36] | $46 \log n$ | Not available |
| Myrinet [5, 8, 9] | $20 \log n$ | $\sim 15n$ |
| Infiniband [25] | $20 \log n$ | Not available |
| QsNET (see Section 3.3) | $< 10$ | $> 150n$ |
| BlueGene/L [18] | $< 2$ | $700n$ |

**Generality of Mechanisms** Currently, STORM supports batch scheduling with and without backfilling, gang scheduling [28], and implicit coscheduling [2]. However, we believe that STORM's mechanisms are sufficiently general as to be used for an efficient implementation of a variety of schemes. We initially plan to implement a number of additional coordinated-scheduling algorithms (such as buffered coscheduling (BCS) [29, 30]) using the STORM mechanisms. This will enable us to perform a fair comparison of multiple algorithms, and to research how each behaves on a common set of workloads.

Fault detection is a rather different application from process scheduling but it relies on the same set of mechanisms. A master process periodically multicasts a heartbeat message (with XFER-AND-SIGNAL) and queries the slaves for receipt (with COMPARE-AND-WRITE). If the query returns FALSE, indicating that a slave missed a heartbeat, the master can gather status information to isolate the failed slave. Another possible use of the STORM mechanisms is to implement a graphical interface for cluster monitoring. As before, the master can multicast a request for status information and gather the results from all of the slaves. In short, we argue that STORM's mechanisms are sufficiently general for a variety of uses and sufficiently fast to make their use worthwhile.

# 5 Related Work

Although powerful hardware solutions for high-performance computing are already available, the largest challenge in making large-scale clusters usable lies in the system software. In this paper, we presented our solution, STORM, which currently focuses on increasing the scalability and performance of job launching and process scheduling.

## 5.1 Job launching

Many run-time environments, such as the Portable Batch System (PBS) [4], distribute executable files to all nodes via a globally mounted filesystem, typically NFS [35]. The advantages of this approach are its simplicity and portability. However, because many clients are simultaneously accessing a single file on a single server, the shared-filesystem approach is inherently nonscalable and tends to lead to launch failures under load because of communication timeouts. A common workaround is to employ a simple shell script that iteratively starts processes on each node of the cluster. While this approach reduces contention on the file server it still has severe performance and scalability limitations on large-scale clusters. In contrast, by implementing the STORM mechanisms in terms of a tree-based multicast, STORM overhead grows logarithmically, not linearly, in the number of nodes.

GLUnix [17] is a piece of operating system middleware for clusters of workstations, designed to provide transparent remote execution, load balancing, coscheduling of parallel jobs, and fault-detection. While GLUnix launches jobs quickly on small clusters, a substantial performance degradation emerges on larger clusters ($> 32$ nodes) because reply messages from the slaves collide with subsequent request messages from the master [17]. STORM, however, can benefit from QsNET's network conditionals [31] which utilize a combining tree to reduce network contention and improve performance and scalability.

The Computational Plant (Cplant) project [34] at Sandia National Laboratories is the closest project in spirit to ours in that it identifies poor resource-management performance as a problem worth studying and approaches the problem by replacing a traditionally nonscalable algorithm with a scalable one. Given the same Myrinet [6]-based platform the STORM mechanisms would likely be implemented similarly to Cplant's. However, on a platform such as QsNET, which boasts hardware collectives, STORM is able to exploit the underlying hardware to improve job-launching performance by a hundredfold.

BProc [19], the Beowulf Distributed Process Space, takes a fairly different approach to job launching from STORM and the other works described above. Rather than copy a binary file from a disk on the master to a disk on each of the slaves and then launching the file from disk, BProc replicates a *running* process into each slave's memory—the equivalent of Unix's `fork()` and `exec()` plus an efficient migration step. The advantage of BProc's approach is that no filesystem activity is required to launch a parallel application once it is loaded into memory on the master. Even though STORM utilizes a RAM-disk-based filesystem, the extra costs of reading and writing that filesystem add a significant amount of overhead relative to BProc's remote process spawning. STORM's advantage over BProc is that the same mechanisms that STORM uses to transmit executable files can also be used to transmit data files. BProc has no equivalent mechanism though a system could certainly use BProc for its single-system-image features and STORM for the underlying communication protocols.

Table 6 presents a selection of job-launching performance results found in the literature, and Table 7 extrapolates each of these out to 4,096 nodes. In addition, Figure 11 graphs both the measured and extrapolated (to 16,384 nodes) job-launching data. While this is admittedly not an apples-to-apples comparison, the point remains that STORM is so much faster than state-of-the-art resource-management systems that we expect that the data shown in those tables would be qualitatively the same in a fairer comparison.

Table 6: A selection of job-launch times found in the literature

| Resource manager | Job-launch time | |
|---|---|---|
| `rsh` | 90 | seconds to launch a minimal job on 95 nodes [17] |
| RMS | 5.9 | seconds to launch a 12 MB job on 64 nodes [14] |
| GLUnix | 1.3 | seconds to launch a minimal job on 95 nodes [17] |
| Cplant | 20 | seconds to launch a 12 MB job on 1,010 nodes [7] |
| BProc | 2.7 | seconds to launch a 12 MB job on 100 nodes [19] |
| **STORM** | 0.11 | seconds to launch a 12 MB job on 64 nodes |

Table 7: Extrapolated job-launch times

| Resource manager | Job-launch time extrapolated to 4,096 nodes | | |
|---|---|---|---|
| `rsh` | 3,827.10 | seconds for 0 MB | ($t = 0.934n + 1.266$) |
| RMS | 317.67 | seconds for 12 MB | ($t = 0.077n + 1.092$) |
| GLUnix | 49.38 | seconds for 0 MB | ($t = 0.012n + 0.228$) |
| Cplant | 22.73 | seconds for 12 MB | ($t = 1.379 \lg n + 6.177$) |
| BProc | 4.88 | seconds for 12 MB | ($t = 0.413 \lg n - 0.084$) |
| **STORM** | 0.11 | seconds for 12 MB | (see Section 3.3) |

To clarify the performance improvement provided by STORM, Figure 12 renormalizes the extrapolated Cplant [7] and BProc [19] launch-time data to the extrapolated STORM data, which is defined as 1.0. Cplant and BProc are the two pieces of related work that, like STORM, scale logarithmically, not linearly, in the number of nodes.

## 5.2 Process scheduling

Many recent research results show that good job scheduling algorithms can substantially improve scalability, responsiveness, resource utilization, and usability of large-scale parallel machines [1, 13]. Unfortunately, the body of work developed in the last few years has not yet led to many practical implementations of such coscheduling algorithms on production 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 needed to convincingly demonstrate their superiority over today's entrenched, space-shared schedulers. STORM's flexibility positions STORM as a suitable vessel for *in vivo* experimentation with alternate scheduling algorithms, so researchers and cluster administrators can determine the best way to manage cluster resources.

Regarding traditional gang-schedulers, the SCore-D scheduler [21] is one of the fastest. By employing help from the messaging layer, PM [39], SCore-D is able to force communication into a quiescent state, save the entire global state of the computation, and restore another application's global state with only ~2% overhead when using a 100 ms time quantum. While this is admirable performance, STORM is able to do significantly better. Because the STORM mechanisms can be written to exploit QsNET's process-to-process communication (versus PM/Myrinet's node-to-node communication), STORM does not need to force the network into a quiescent state before freezing one appli-

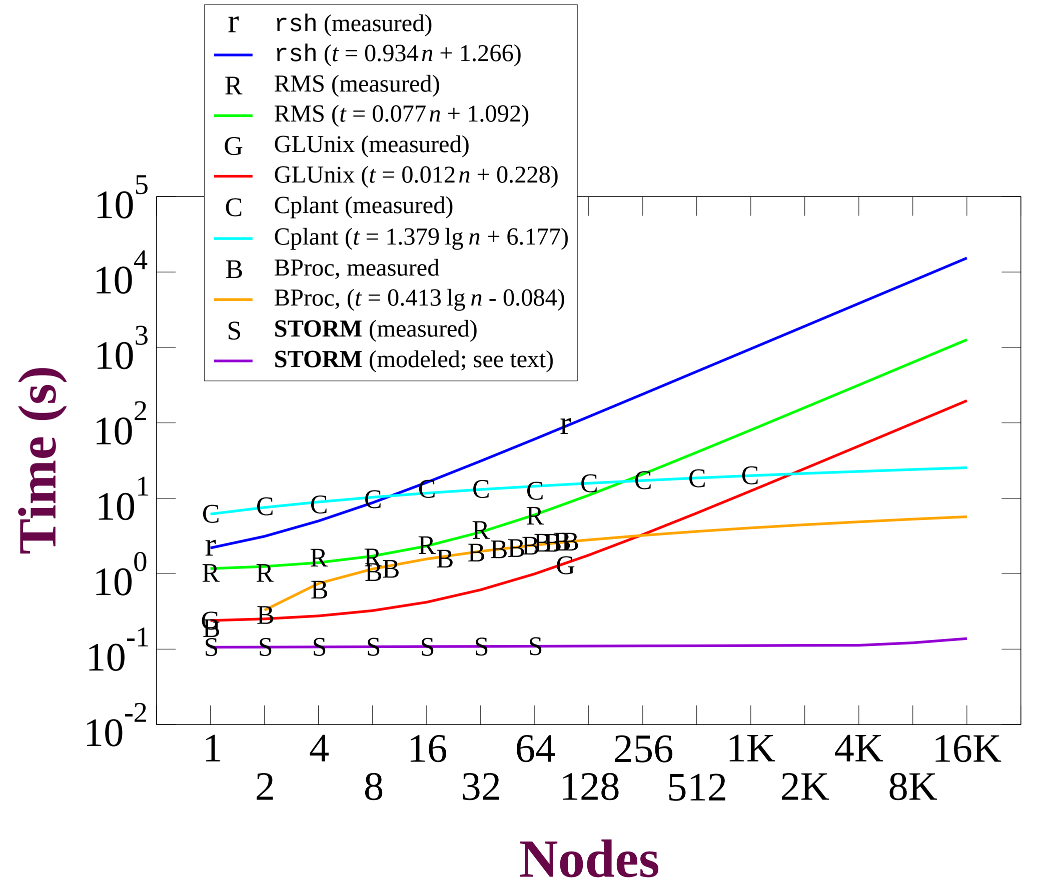

Figure 11: Measured and predicted performance of various job launchers

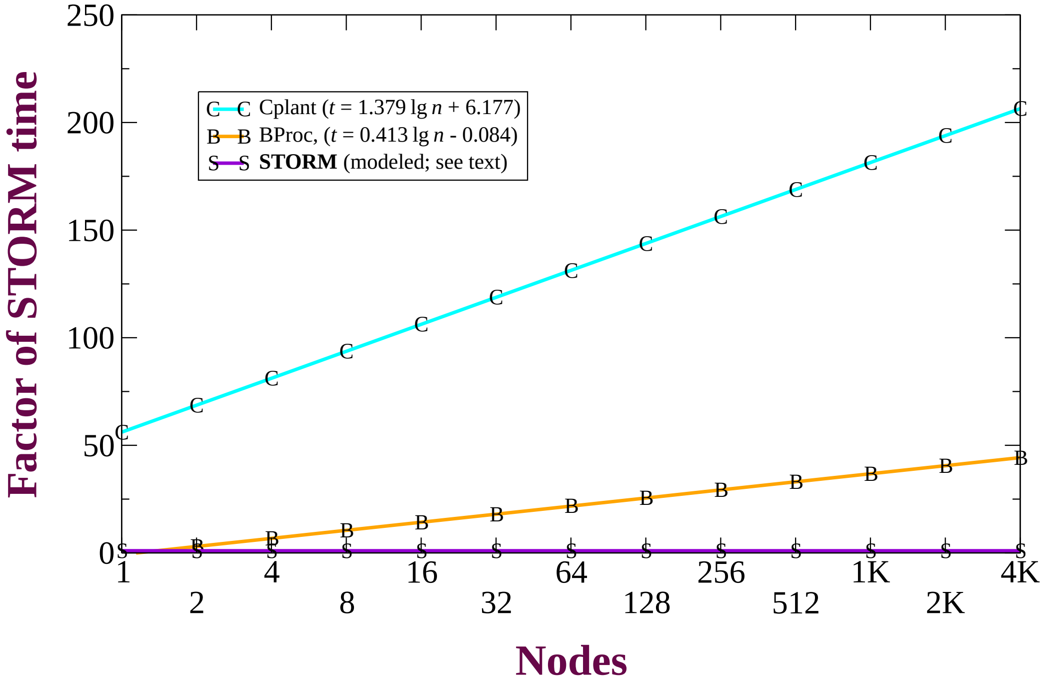

Figure 12: Relative performance of Cplant, BProc, and STORM

cation and thawing another. As a result STORM can gang-schedule applications with no noticeable overhead when using quanta as small as 2 ms.

Table 8 lists the minimal feasible scheduling quantum supported by STORM and previous job schedulers. That is, the table does not show the shortest possible quantum, but rather the shortest quantum that leads to an application slowdown of 2% or less. Again, this is not an entirely fair comparison but it does indicate that STORM is at least two orders of magnitude better than the best reported numbers from the literature.

Table 8: A selection of scheduling quanta found in the literature

| Resource manager | Minimal feasible scheduling quantum | |
|---|---:|---|
| RMS | 30, 000 | milliseconds on 15 nodes (1.8% slowdown) [15] |
| SCore-D | 100 | milliseconds on 64 nodes (2% slowdown) [21] |
| **STORM** | 2 | milliseconds on 64 nodes (no observable slowdown) |

## 6 Conclusions

An increasingly important factor in large scale computing is the performance of the resource-management system. While the purpose of a cluster is to run applications, it is the goal of the resource manager to ensure that these applications load quickly, make efficient use of cluster resources (CPU, network, etc.), and interact to user input with a small response time. Current resource-management systems require many seconds to launch a large application. They either batch-schedule jobs—precluding interactivity—or gang-schedule them with such large quanta as to be effectively non-interactive. Furthermore, they make poor use of resources because large jobs frequently suffer from internal load imbalance or imperfect overlap of communication and computation, yet scheduling decisions are too costly to warrant lending unused resources to alternate jobs.

To address these problems we developed STORM, a lightweight, flexible, and scalable environment for performing resource management in large-scale clusters. In terms of job launching STORM is an order of magnitude faster than the best reported results in the literature [19], and in terms of process scheduling STORM is two orders of magnitude faster than the best reported results in the literature [21]. The key to STORM's performance lies in its design methodology. Rather than implement heartbeat issuance, job launching, process scheduling, fault detection, and other resource-management routines as separate entities, we designed those functions in terms of a small, portable set of data-transfer and synchronization mechanisms: XFER-AND-SIGNAL, TEST-EVENT, and COMPARE-AND-WRITE. If each of these mechanisms is fast and scalable on a given platform, then STORM as a whole is fast and scalable as well. We validated STORM's performance on a 256-processor Alpha cluster interconnected with a QsNET network and demonstrated that STORM performs well on that cluster, and presented evidence that it should perform comparably well on significantly larger clusters.

An important conclusion of our work is that it is indeed possible to scale up a cluster without sacrificing fast job-launching times, machine efficiency, or interactive response time. STORM can launch parallel jobs on a large-scale cluster almost as fast as a node OS can launch a sequential application on an individual workstation. In addition, STORM can schedule all of the processes in a large, parallel job with the same granularity and with almost the same low overhead at which a sequential OS can schedule a single process.

A second conclusion, drawn from our experience with implementing the STORM mechanisms on QsNET, is that there are a number of features that a NIC/network can support that greatly help to improve the performance—and simplify the code—of a resource-management system. The most useful of these include (1) ordered, reliable communication, (2) programmable NICs, (3) direct NIC access to host virtual memory, (4) global network conditions, and (5) host process-to-host process communication. In addition, we found QsNET's hardware multicasts and remote hardware queues quite convenient for implementing resource-management functions.

In short, by improving the performance of various resource-management functions by orders of magnitude, STORM represents an important step towards making large-scale clusters as efficient and easy to use as a workstation. While STORM is still a research prototype, we foresee STORM or a tool based on our resource-management research as being the driving force behind making large-scalable clusters usable and efficient.

## Acknowledgments

The authors would like to thank Dror Feitelson for providing the ParPar scheduler which STORM uses as its global-allocation scheduler, Jon Beecroft and Moray McLaren of Quadrics Supercomputers World for providing the network bandwidth data presented in Table 4 on page 14, Ron Brightwell of Sandia National Laboratories for providing the Cplant data used in Figure 11 on page 20, Morris "Moe" Jette of Lawrence Livermore National Laboratory for his insightful comments, and Erik Hendriks of Los Alamos National Laboratory for providing the BProc data also used in Figure 11 on page 20.

## References

[1] Andrea C. Arpaci-Dusseau. Implicit coscheduling: Coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3), August 2001. Available from `http://www.cs.wisc.edu/~dusseau/Papers/tocs01.ps`.

[2] Andrea C. Arpaci-Dusseau, David Culler, and Alan M. Mainwaring. Scheduling with implicit information in distributed systems. In *Proceedings of the SIGMETRICS '98/PERFORMANCE '98 Joint International Conference on Measurement and Modeling of Computer Systems.* Also published in *Performance Evaluation Review 26(1)*, pages 233–243, Madison, Wisconsin, June 22–26, 1998. ACM, ACM Press. ISSN 0163-5999. Available from `http://now.cs.berkeley.edu/Implicit/sig98.ps`.

[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 ACM SIGMETRICS '95/PERFORMANCE '95 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, Ottawa, Canada, May 15–19, 1995. Available from `http://now.cs.berkeley.edu/Implicit/sig95.ps`.

[4] Albeaus Bayucan, Robert L. Henderson, Casimir Lesiak, Bhroam Mann, Tom Proett, and Dave Tweten. Portable Batch System: External reference specification, release 1.1.12. Technical report, Numerical Aerospace Simulation Systems Division, NASA Ames Research Center, August 10, 1998. Available from `http://www.nas.nasa.gov/Software/PBS/pbsdocs/ers.ps`.

[5] Raoul A.F. Bhoedjang, Tim Rühl, and Henri E. Bal. Efficient multicast on Myrinet using link-level flow control. In *27th International Conference on Parallel Processing (ICPP98)*, pages 381–390, Minneapolis, MN, August 1998. Available from `ftp://ftp.cs.vu.nl/pub/raoul/papers/multicast98.ps.gz`.

[6] 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, February 1995. Available from `http://www.myri.com/research/publications/Hot.ps`.

[7] Ron Brightwell and Lee Ann Fisk. Scalable parallel application launch on Cplant. In *Proceedings of SC2001*, Denver, Colorado, November 10–16, 2001. Available from `http://www.sc2001.org/papers/pap.pap263.pdf`.

[8] Darius Buntinas, Dhabaleswar Panda, José Duato, and P. Sadayappan. Broadcast/multicast over Myrinet using NIC-assisted multidestination messages. In *Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC '00), High Performance Computer Architecture (HPCA-6) Conference*, Toulouse, France, January 2000. Available from `ftp://ftp.cis.ohio-state.edu/pub/communication/papers/canpc00-nic-multicast.pdf`.

[9] Darius Buntinas, Dhabaleswar Panda, and William Gropp. NIC-based atomic operations on Myrinet/GM. In *SAN-1 Workshop, High Performance Computer Architecture (HPCA-8) Conference*, Boston, MA, February 2002. Available from `ftp://ftp.cis.ohio-state.edu/pub/communication/papers/san-1-atomic_operations.pdf`.

[10] Chris Eddington. InfiniBridge: An InfiniBand channel adapter with integrated switch. *IEEE Micro*, 22(2):48–56, March/April 2002.

[11] Dror G. Feitelson. Packing schemes for gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *2nd Workshop on Job Scheduling Strategies for Parallel Processing (in IPPS '96)*, pages 89–110, Honolulu, Hawaii, April 16, 1996. Springer-Verlag. Published in Lecture Notes in Computer Science, volume 1162. ISBN 3-540-61864-3. Available from `http://www.cs.huji.ac.il/~feit/parsched/p-96-6.ps.gz`.

[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 758–774. Prentice-Hall, 1999. ISBN 0-13-013784-7. Available from `http://www.cs.huji.ac.il/labs/parallel/chap.ps.gz`.

[13] Dror G. Feitelson and Morris A. Jette. Improved utilization and responsiveness with gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *3rd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261, Geneva, Switzerland, April 5, 1997. Springer-Verlag. Available from `http://www.cs.huji.ac.il/~feit/parsched/p-97-11.ps.gz`.

[14] Eitan Frachtenberg, Juan Fernandez, Fabrizio Petrini, and Scott Pakin. STORM: A Scalable TOol for Resource Management (student poster). In *The Conference on High Speed Computing*, page 27, Gleneden Beach, Oregon, April 22–25, 2002. LANL/LLNL/SNL.

[15] Eitan Frachtenberg, Fabrizio Petrini, Salvador Coll, and Wu-chun Feng. Gang scheduling with lightweight user-level communication. In *2001 International Conference on Parallel Processing (ICPP 2001), Workshop on Scheduling and Resource Management for Cluster Computing*, Valencia, Spain, September 3–7, 2001. Available from `http://www.c3.lanl.gov/~fabrizio/papers/icpp01.pdf`.

[16] Hubertus Franke, Joefon Jann, José E. Moreira, Pratap Pattnaik, and Morris A. Jette. An evaluation of parallel job scheduling for ASCI Blue-Pacific. In *Proceedings of SC99*, Portland, Oregon, November 13–19, 1999. ACM Press and IEEE Computer Society Press. ISBN 1-58113-091-0. Available from `http://www.supercomp.org/sc99/proceedings/papers/moreira2.pdf`.

[17] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix: a global layer Unix for a network of workstations. *Software—Practice and Experience*, 28(9):929–961, July 25, 1998. Available from `http://www-2.cs.cmu.edu/~dpetrou/papers/glunix98.ps.gz`.

[18] Manish Gupta. Challenges in developing scalable scalable software for bluegene/l. In *Scaling to New Heights Workshop*, Pittsburgh, PA, May 2002. Available from `http://www.psc.edu/training/scaling/gupta.ps`.

[19] Erik Hendriks. BProc: The Beowulf distributed process space. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing (ICS '02)*, New York, New York, June 22–26, 2002. Available from `http://www.acl.lanl.gov/cluster/papers/hendriks-ics02/hendriks-ics02.pdf`.

[20] Adolfy Hoisie, Olaf Lubeck, Harvey Wasserman, Fabrizio Petrini, and Hank Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *2000 International Conference on Parallel Processing,(ICPP2000)*, Toronto, Canada, August 2000. Available from `http://www.c3.lanl.gov/~fabrizio/papers/icpp00.pdf`.

[21] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Highly efficient gang scheduling implementation. In *Proceedings of Supercomputing '98*, Orlando, Florida, November 7–13, 1998. IEEE Computer Society and ACM SIGARCH. Available from `http://www.supercomp.org/sc98/TechPapers/sc98_FullAbstracts/Hori698/INDEX.HTM`.

[22] Avi Kavas, David Er-El, and Dror G. Feitelson. Using multicast to pre-load jobs on the ParPar cluster. *Parallel Computing*, 27(3):315–327, February 2001. Preliminary version is available from `http://www.cs.huji.ac.il/labs/parallel/rdgm.ps.gz`.

[23] K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65(108):198–199, 1992.

[24] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[25] Jiuxing Liu, Jiesheng Wu, Dhabaleswar K. Panda, and Chanan Shamir. Designing clusters with Infiniband: Early experience with Mellanox technology. Submitted for publication.

[26] Hans W. Meuer, Erich Strohmaier, Jack J. Dongarra, and Horst D. Simon. Top500 supercomputer sites. In *Proceedings of SC2001*, Denver, Colorado, November 10–16, 2001. Available from `http://www.top500.org/lists/2001/11/top500_0111.pdf`.

[27] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita R. Das. A closer look at coscheduling approaches for a network of workstations. In *Proceedings of the Eleventh ACM Symposium on Parallel Algorithms and Architectures, (SPAA '99)*, pages 96–105, Saint-Malo, France, June 27–30, 1999. ACM Press. Available from `http://www.cse.psu.edu/~anand/csl/papers/spaa99.pdf`.

[28] J. K. Ousterhout. Scheduling techniques for concurrent systems. *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 22–30, October 1982.

[29] 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 (IPDPS 2000)*, volume 16, Cancun, Mexico, May 1–5, 2000. Available from `http://ipdps.eece.unm.edu/2000/papers/petrini.pdf`.

[30] Fabrizio Petrini and Wu-chun Feng. Improved resource utilization with buffered coscheduling. *Journal of Parallel Algorithms and Applications*, 16:123–144, 2001. Available from `http://www.c3.lanl.gov/~fabrizio/papers/paa00.ps.gz`.

[31] 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. ISSN 0272-1732. Available from `http://www.computer.org/micro/mi2002/pdf/m1046.pdf`.

[32] Quadrics Supercomputers World Ltd. *Elan Programming Manual*, 2nd edition, December 1999.

[33] Quadrics Supercomputers World Ltd. *Elan Reference Manual*, 1st edition, January 1999.

[34] Rolf Riesen, Ron Brightwell, Lee Ann Fisk, Tramm Hudson, Jim Otto, and Arthur B. Maccabe. Cplant. In *Proceedings of the 1999 USENIX Annual Technical Conference, Second Extreme Linux Workshop*, Monterey, California, June 6–11, 1999. Available from `http://www.cs.sandia.gov/~rolf/papers/extreme/cplant.ps.gz`.

[35] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3010, Internet Engineering Task Force, Network Working Group, December 2000. Available from `http://www.rfc-editor.org/rfc/rfc3010.txt`.

[36] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. EMP: Zero-copy os-bypass nic-driven gigabit ethernet message passing. In *Proceedings of SC2001*, Denver, Colorado, November 10–16, 2001. Available from `http://www.sc2001.org/papers/pap.pap315.pdf`.

[37] Warren Smith, Valerie Taylor, and Ian Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the IPPS/SPDP 1999 Workshop on Job Scheduling Strategies for Parallel Processing,* in *Lecture Notes on Computer Science*, volume 1659, pages 202–219, San Juan, Puerto Rico, April 12–16, 1999. Springer-Verlag. Available from `http://www.globus.org/documentation/incoming/p.ps`.

[38] Byron Spice. Pittsburgh supercomputer is complete, and scientists are champing at the bit to use it. *The Pittsburgh Post-Gazette*, October 1, 2001. Available from `http://www.postgazette.com/healthscience/20011001terascale1001p3.asp`.

[39] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An operating system coordinated high performance communication library. In Bob Hertzberger and Peter M. A. Sloot, editors, *High-Performance Computing and Networking: International Conference and Exhibition (HPCN Europe)*, volume 1225 of *Lecture Notes in Computer Science*, pages 708–717, Vienna, Austria, April 28–30, 1997. Springer-Verlag. ISBN 3-540-62898-3. Available from `http://citeseer.nj.nec.com/tezuka97pm.html`.