STORM: Scalable Resource Management for Large-Scale Parallel Computers[*]

Eitan Frachtenberg[†] Fabrizio Petrini[†,‡] Juan Fernández[§] Scott Pakin[†]

Abstract

Although clusters are a popular form of high-performance computing, they remain more difficult to manage than sequential systems—or even symmetric multiprocessors. In this paper, we identify a small set of primitive mechanisms that are sufficiently general to be used as building blocks to solve a variety of resource-management problems. We then present STORM, a resource-management environment that embodies these mechanisms in a scalable, low-overhead, and efficient implementation. The key innovation behind STORM is a modular software architecture that reduces all resource management functionality to a small number of highly scalable mechanisms. These mechanisms simplify the integration of resource management with low-level network features. As a result of this design, STORM can launch large, parallel applications an order of magnitude faster than the best time reported in the literature and can gang-schedule a parallel application as fast as the node OS can schedule a sequential application. This paper describes the mechanisms and algorithms behind STORM and presents a detailed performance model that shows that STORM's performance can scale to thousands of nodes.

Keywords: C.0.b Hardware/software interface; C.0.e System architectures, integration, and modeling; C.2.4.d Network operating systems; C.5.1.a Supercomputers

1 Introduction

Although much attention is paid to the performance of *applications* running on high-performance computer systems, the performance of resource-management (RM) software has largely been neglected. This neglect is not unfounded, for the following reasons:

1. Even the least efficient, least scalable RM tools occupy a small fraction of total time on today's small-to-medium-sized clusters.

[*]This work was supported by the U.S. Department of Energy through Los Alamos National Laboratory contract W-7405-ENG-36 and the Spanish MCYT under grant TIC2003-08154-C06-03.

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

[‡]Fabrizio Petrini is now affiliated with the Applied Computer Science Group, Computational Sciences and Mathematics Division, Pacific Northwest National Laboratory.

[§]Dpto. Ingeniería y Tecnología de Computadores, Universidad de Murcia (Spain)

2. Today's users are willing to tolerate noninteractive execution of their jobs.

3. Application programmers are still able to move RM functionality into their applications to make up for whatever functionality is not provided by the RM system or for whatever functionality is unusably slow.

With cluster sizes of tens of thousands of processors coming online [34], resource management can no longer be ignored. Even a small amount of wasted wall-clock time translates to a significant amount of aggregate wasted CPU time. Furthermore, nonscalable RM functions must be amortized by calling them as infrequently as possible. This degrades responsiveness and hinders the usage of interactive jobs and noninteractive jobs being debugged. Clusters—even the largest ones in existence—should ideally be as usable as a desktop system, with split-second job-launch times and timeshared job execution. We claim that clusters can approach the usability of a desktop if functions such as job launching and scheduling can be implemented in a manner that is both fast and scalable to large numbers of nodes. As a proof of concept of the potential for improving system performance by optimizing RM software we have developed a system called STORM (Scalable TOol for Resource Management). STORM implements RM primitives in an efficient manner by delegating as much of the work as possible to scalable network collective operations.

The rest of this paper is organized as follows. Section 2 and 3 detail STORM's architecture and implementation. We analyze STORM's performance in Section 4. Section 5 compares STORM to prior resource managers. Finally, we draw some conclusions from this work in Section 6.

2 Architecture

This section describes the architecture of STORM. The main design goals for STORM were: (1) to provide RM 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 use a set of loosely-coupled dæmons that communicate with extremely fast messages. Dæmons are coordinated by multicasting strobes (a.k.a. heartbeat) messages with a scalable multicast algorithm. For the second goal, the dæmons are designed so that modules for different scheduling algorithms can be "plugged" into them. In this paper, we focus

on one successful algorithm, gang-scheduling (GS) [8].[1] We did however implement four other scheduling algorithms, which are detailed in various scheduling-specific studies [9, 10]. GS allocates both space (processors) and time resources to incoming jobs. All of the processes of a job are coscheduled within their time slot for the duration of a timeslice and are then context-switched to a different job in a cyclic manner.

Relative to batch scheduling, GS allows jobs to start immediately but at the cost of longer execution times caused by resource contention with other running jobs. To amortize overhead, gang schedulers are usually run with large scheduling quanta, on the order of seconds or even minutes [13]. While large quanta increase throughput they also increase response time, which hinders interactive jobs. Section 3 explains how STORM was architected to minimize rather than amortize scheduling overhead.

2.1 Overview of STORM

Several issues were considered crucial for STORM and were incorporated into its design:

Flexibility STORM supports various scheduling algorithms, including first-come first-served (FCFS), GS, Flexible-, Buffered-, and Implicit-Coscheduling [1, 8, 10, 24], and backfilling.

Usability STORM is designed so that parallel applications need not be changed to accommodate the system, and need only be relinked with a slightly modified version of the Message-Passing Interface (MPI) library.

Portability STORM runs entirely in user mode and with no OS modifications. STORM was successfully ported to three CPU architectures (x86, Alpha, and Itanium), and two interconnects (QsNet and generic MPI).

Scalability STORM is designed so that most of the scheduler operations are decentralized and asynchronous and the rest are implemented on top of scalable collective operations.

Performance STORM can take full advantage of the underlying network hardware and is designed to provide significantly superior performance to any existing parallel job scheduler.

[1] By gang-scheduling we refer to explicit coscheduling with global synchronization.

Table 1: STORM dæmons

Dæmon	Number	Role	Location
MM	One per cluster	Global scheduling, strobing, and resource accounting	Management node
NM	One per compute node	Local scheduling and resource monitoring	Compute nodes
PL	One per application process	Fork and execute application process	Compute nodes

2.2 Process Structure

For ease of initial implementation, the STORM scheduler was written as a user-level scheduler which consists of a number of communicating dæmons. The primary advantage of user-level scheduling is that is is easier to implement and modify, which facilitates experimentation. The primary disadvantage is that the STORM scheduler is susceptible to variability induced by the underlying OS scheduler. If the scheduling dæmons do not each have a dedicated CPU then they must compete with applications for CPU access, which increases scheduling time. While a longer-term solution is to integrate the STORM scheduler with the OS kernel scheduler, the experiments shown in this paper represent the case in which each dæmon has its own CPU (except where noted).

STORM comprises three types of dæmons (Table 1): the Machine Manager (MM), the Node Manager (NM), and the Program Launcher (PL). These dæmons do not require dedicated CPUs because they run only briefly and only at timeslice intervals. 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 [5, 6]. If the scheduling policy allows for multiprogramming (as does GS), the processors are allocated in any time slot that has enough resources available. After a successful allocation, the MM broadcasts a job-launch message to the relevant NMs, which launch the job when its time slot arrives.

When launching a parallel application, the MM first transfers the binary image of the program to each node's local file system (via each node's NM) and then instructs the PLs (again, via the NMs) to launch the application locally. This procedure exploits an efficient broadcast mechanism which can disseminate a file of several megabytes to all of the nodes in a fraction of a second. (A more common but poorer-performing alternative is to disseminate files via a—less-scalable—shared filesystem such as NFS [30]. When a process terminates, the appropriate PL notifies the

NM, which in turn notifies the MM. The MM then marks the time/space resources occupied by that process as available for allocation. Section 2.4 describes the termination algorithm in more detail. Note that even though the MM is centralized, in reality it does not create a bottleneck because all of its global operations utilize scalable hardware broadcasts, and all of its local operations—reading a new job, allocating resources to it, and receiving process-termination notifications—are both rare and lightweight.

2.3 Running a Job

Jobs are launched in STORM according to the following procedure. (1) The MM reads the job information from the workload file and enqueues it. (2) After the designated job launch time, as soon as the requested resources become available, the MM multicasts the job information (possibly with the binary image) to all of the NMs. (3) If the NM needs to fork processes, it locates the appropriate PLs (according to the job's PE/timeslot allocation) and sends the job information to it using shared memory. (4) The PLs execute the APs (Section 2.2). (5) When an AP terminates, its PL receives a notification from the operating system. (6) The PLs notify the NM of the APs' termination. Finally, (7) the NMs send a message to the MM, which deallocates the processes' resources.

For the job launching mechanism, which involves the broadcast of binaries and data, we implemented a specialized protocol. This protocol alleviates one of the major bottlenecks in program launching: the interaction with the I/O subsystem. The MM multicasts file chunks to the NMs who write the files locally in a symmetric fashion. File chunks are pipelined and flow-controlled so that a chunk can be sent while the next chunk is read at the source and the preceding chunk is written at the destination. The performance of this file-distribution protocol is discussed in Section 4.2.1.

2.4 Algorithms

The key insight behind STORM's architecture is that by reducing all RM functions to a small set of primitives we increase STORM's portability and maintainability. We therefore partitioned the code into a higher-level abstraction layer which implements all of STORM's RM functions and a lower-level abstraction layer which maps network functionality into three basic primitives (Figure 1). Our thesis is that if these three primitives are implemented efficiently on a network then all of STORM's

STORM RM functions	heartbeat, file transfer, termination detection
(STORM helper functions)	flow control, queue management
STORM primitives	XFER-AND-SIGNAL, TEST-EVENT, COMPARE-AND-WRITE
network primitives	remote DMA, network conditionals, event signaling, . . .

Figure 1: STORM implementation structure

RM functions will be efficient as well. Furthermore, porting STORM to a new network architecture requires only a suitable mapping of the three STORM primitives to that architecture. In this section we describe first the lower-level abstraction layer and then the higher-level abstraction layer.

2.4.1 Lower-level Abstraction Layer

Collective communication is central to STORM's lower-level abstraction layer. Not only is resource management in a cluster environment *inherently* collective but collective operations can be made efficient by taking advantage of network support. There are only three primitives upon which all of STORM is based—one for global data transfer, one for local control, and one for global control:

XFER-AND-SIGNAL Transfer (PUT) a block of data from local memory to the global memory of a subset of nodes. 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 $\geq$, $<$, $=$, or $\neq$) a local value to a global variable on a subset of the nodes and report whether the condition holds true everywhere. Also, optionally assign a new value to a—possibly different—global variable.

The following are some important points about the STORM primitives' semantics:

1. *Global* data refers to data that lies at the same virtual address on all nodes. Depending on the implementation, global data may reside in either main memory or network-interface memory. Global cache coherency is neither assumed nor required.

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 or—in case of a

network error—*no* nodes (and an error status is returned). 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 [19].

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 invoke TEST-EVENT on a local event which XFER-AND-SIGNAL signals.

4. The semantics do not dictate whether the STORM primitives 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 [1]).

2.4.2 Higher-level Abstraction Layer

The three STORM primitives described in Section 2.4.1 are general enough to be used for a wide variety of RM functions. Continuing our bottom-up exposition we now examine how to construct the following functions out of those primitives: (1) issue a heartbeat (data transfer + notification) to all nodes; (2) transfer a large file to a set of nodes; and, (3) detect termination of an application running on a set of nodes. STORM's implementations of job launching and its various process-scheduling algorithms follow in a straightforward manner from heartbeats, file transfers, and termination detection.

Both heartbeat issuance and termination detection require a remote queueing mechanism by which a master node can multicast data into a queue read by a set of slave nodes. To ensure that no data is lost, STORM uses a *global* flow-control algorithm (AWAIT-SPACE, Algorithm 1). Algorithm 1 is implemented with a simple application of COMPARE-AND-WRITE. If a designated queue is full on *any* node (i.e., $\exists p \in P$ such that $Q^{(p)}.enqueued - Q^{(p)}.length = Q^{(p)}.dequeued$) the algorithm waits and checks again. If no node's queue is full (i.e., $Q^{(p)}.enqueued - Q^{(p)}.length < Q^{(p)}.dequeued \quad \forall p \in P$), the algorithm returns. Algorithms 2 and 3 show how remote queueing is

implemented in terms of the flow-control algorithm and the STORM primitives. The master uses AWAIT-SPACE to wait until none of the nodes has a full queue. It then uses XFER-AND-SIGNAL to transfer the data and signal the slaves. The slave threads merely block on the data-transfer event and process the new data when the event occurs. Note that in Algorithms 1 and 2, *Q.dequeued* represents global data, as defined on page 6.

Algorithm 1 Implementation of STORM flow-control mechanism

```
function AWAIT-SPACE (P,        ▷ Set of processes to write to
                      Q)        ▷ Queue in which to check for space

  while COMPARE-AND-WRITE (P, Q.enqueued − Q.length, "<", Q.dequeued, NULL, NULL) = TRUE do
    {∃p ∈ P such that p's queue is full. Wait and try again later.}
    SLEEP (exponential backoff)
```

Algorithm 2 Remote queueing (master side: enqueue)

```
function ENQUEUE (P,                ▷ Set of processes to write to
                  Q,                ▷ Queue on which to push data
                  data,             ▷ Data to write to P's queue_tail
                  remote_event,     ▷ Remote event to signal upon completion
                  local_event)      ▷ Local event to signal upon completion

  AWAIT-SPACE (P, queue)
  XFER-AND-SIGNAL (P, Q.queue_{Q.tail}, data, ||data||, remote_event, local_event)
  Q.tail ← (Q.tail + 1) mod Q.length
  Q.enqueued ← Q.enqueued + 1
```

Algorithm 3 Remote queueing (slave side: blocking dequeue)

```
function DEQUEUE (Q,                ▷ Queue from which to pop data
                  new_data_event)   ▷ Local event signaled whenever data is enqueued

  TEST-EVENT (new_data_event, TRUE)
  current_head ← Q.head
  Q.head ← (Q.head + 1) mod Q.length
  Q.dequeued ← Q.dequeued + 1
  return Q.queue_{current_head}
```

Heartbeat issuance The STORM master process (the MM) communicates to its slaves (the NMs) only at regular heartbeat intervals. Using a heartbeat minimizes nondeterminism in the cluster. STORM heartbeats have both a data transfer and a notification component. Data are first multicast to all nodes and then the nodes are awakened and notified that new data exist. Because the

remote queueing shown in Algorithms 2 and 3 already supports both data transfer and notification, the heartbeat functions (Algorithms 4 and 5) are trivial applications of remote queueing.

Algorithm 4 Heartbeat (master side: issue)

function ISSUE-HEARTBEAT (*P*) ▷ Set of processes to write to

{Trivial use of Algorithm 2}
ENQUEUE (*P*, *heartbeat_queue*, *heartbeat_data*, *heartbeat_event*, NULL) {No need to wait for completion.}

Algorithm 5 Heartbeat (slave side: receive)

function ISSUE-HEARTBEAT ()

heartbeat_data ← DEQUEUE (*heartbeat_queue*, *heartbeat_event*)
if *heartbeat_data.type* = RECEIVE-FILE-INFO **then**
{We were given a filename and file size; prepare to receive a file.}
else if *heartbeat_data.type* = LAUNCH-JOB **then**
{We were given the filename of an executable program; launch it.}
else if *heartbeat_data.type* = SCHEDULE-PROCESS **then**
{We were given a (STORM) process ID; schedule the corresponding process.}
else if {other RM commands, to be added later} **then**
⋮

Data transfer STORM uses Algorithms 6 and 7 to distribute executable files across a cluster. However, the same mechanisms can be used to distribute data files, as well. To increase throughput STORM uses a double-buffering scheme on the master (i.e., send side). That is, the master divides the file into fixed-sized chunks and overlaps the multicasting of one chunk with the reading from disk of the next chunk. The chunks are maintained in a queue on each slave. Using a queue enables data transfer to continue briefly even when a slave is temporarily unresponsive (e.g., because of contention on the I/O bus from the host). At the end of the transfer, the master uses STORM's COMPARE-AND-WRITE mechanism to ensure that all of the slaves have finished writing the file. This avoids a race condition with the master's instructing the slaves to launch the executable.

Termination detection The MM must detect when all of the processes in an application have terminated so it can reclaim the application's resources. Algorithms 8 and 9 illustrate how STORM implements termination detection. The procedure is identical regardless of whether the application terminated normally, crashed, or was killed. The interesting aspect of STORM's termination detection is that the MM blocks until a designated NM detects that all of the processes under its control

Algorithm 6 Data transfer (master side: send)

function TRANSFER-FILE (*P*, ▷ Set of processes to write to
file, ▷ Handle to a file opened for reading
num_chunks, ▷ Number of fixed-size chunks of data in *file*
chunk_event) ▷ Remote event to signal after each chunk

Prerequisite: We have already opened the file for reading and issued a heartbeat of type RECEIVE-FILE-INFO that contains the filename and the size of the file in chunks.

{Perform a double-buffered transmission.}
$buf_num \leftarrow 0$
FILE-READ (*file*, $chunk_{buf_num}$) {Elan threads can read the host's filesystems directly via the I/O bypass protocol.}
for $[0,\ldots,num_chunks-2]$ **do** {Both bounds are inclusive.}
  ENQUEUE (*P*, *chunk_queue*, $chunk_{buf_num}$, *chunk_event*, *enqueue_event*) {Asynchronous}
  $buf_num \leftarrow (buf_num+1)$ **mod** 2
  FILE-READ (*file*, $chunk_{buf_num}$)
  TEST-EVENT (*enqueue_event*, TRUE) {Wait for the ENQUEUE to finish.}
{No need to FILE-READ on the final iteration.}
ENQUEUE (*P*, *chunk_queue*, $chunk_{num_chunks-1}$, *chunk_event*, *enqueue_event*)
TEST-EVENT (*enqueue_event*, TRUE) {Wait for the final ENQUEUE to finish.}

Algorithm 7 Data transfer (slave side: receive)

function RECEIVE-FILE (*P*, ▷ Set of processes to write to
file, ▷ Handle to an file opened for writing
num_chunks, ▷ Number of fixed-size chunks of data in *file*
chunk_event) ▷ Local event announcing chunk reception

Prerequisite: We have already received a heartbeat of type RECEIVE-FILE-INFO that contained the filename and the size of the file in chunks and opened the file for writing.

{We repeatedly dequeue a chunk and write it to disk.}
for $chunk_num \in [0,\ldots,num_chunks-1]$ **do**
  chunk_data ← DEQUEUE (*chunk_queue*, *chunk_event*)
  FILE-WRITE (*file*, *chunk_data*)

have terminated and notifies the MM. Only then does the MM begin polling for termination of the application's remaining processes. The insight is that in an SPMD programming model all of an application's processes tend to terminate at approximately the same time. Many MPI [31] implementations exacerbate this effect by barrier-synchronizing as part of their `MPI_Finalize()` routine. Blocking on the first node's termination ensures that no network traffic related to termination detection will occur in the common case, while the application is running. Polling thereafter minimizes network traffic during termination detection.

Algorithm 8 Termination detection (master side: wait)

```
function DETECT-TERMINATION (P,            ▷ Set of nodes that are running jobs
                             Q,            ▷ Unprocessed termination announcements
                             terminated,   ▷ Global array of termination flags
                             term_event)   ▷ Local event signaled on new termination

  {On heartbeat intervals, poll the job-termination queue. If another job has begun to terminate, then add it to the set
  of terminating jobs.}
  if TEST-EVENT (term_event, FALSE) = TRUE then
    job_info ← DEQUEUE (Q, term_event)
    terminating_jobs ← terminating_jobs ∪ {job_info.ID}

  {Check each job that is in the process of terminating and see if it has finished terminating on all nodes. If so, then
  reset all of the terminated flags.}
  for all j ∈ terminating_jobs do
    if COMPARE-AND-WRITE (P, terminated_j, "=", TRUE, terminated_j, FALSE) = TRUE then
      {Job has finished on all nodes.}
      terminating_jobs ← terminating_jobs − {j}
```

Algorithm 9 Termination detection (slave side: notify)

```
function ANNOUNCE-TERMINATION (P,            ▷ Set of nodes that are running jobs
                               Q,            ▷ Unprocessed termination announcements
                               terminated,   ▷ Global array of termination flags
                               term_event)   ▷ Local event signaled on new terminations

  {On heartbeat intervals, poll shared memory for application process completion.}
  if ∄p such that proc_terminated_p = TRUE then
    num_procs_terminated ← num_procs_terminated + 1
    if num_procs_terminated = APP-PROCS-PER-NODE then
      terminated_self.node_ID ← TRUE
      {Last NM notifies the MM when all of its application processes have terminated.}
      if self.node_ID = last_node_ID then
        ENQUEUE (P, Q, self.job_ID, term_event, NULL) {No need to wait for completion.}
```

2.4.3 Generality of Mechanisms

Although the algorithms presented in Section 2.4.2 have, to date, been used by STORM to implement job launching and process scheduling (local scheduling, batch scheduling with and without backfilling, gang scheduling, and implicit coscheduling), the mechanisms are sufficiently general as to be used for the efficient implementation of a variety of RM functions.

For example, fault tolerance 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 and—using COMPARE-AND-WRITE—queries the slaves for receipt. If COMPARE-AND-WRITE returns FALSE, indicating that a slave missed a heartbeat, the master can isolate the failed slave and

commence repairs. Another proposed use of the STORM mechanisms is to implement a graphical interface for cluster monitoring. As before, master can multicast a request for status information and gather the results from all of the slaves. The point is that STORM's mechanisms are general enough for a variety of uses and fast enough to make their use worthwhile.

3 Implementation

This section details a specific implementation of the STORM primitives which were introduced in Section 2.4.1. Algorithms 2–9 demonstrate that the STORM primitives are sufficiently flexible to implement a variety of RM functions: heartbeat issuance, data transfer, termination detection, and—we believe—many others. Furthermore, porting STORM to a new network architecture requires primarily that the STORM primitives be retargeted for that architecture.

A prior publication [12] shows the expected performance—based on the best performance reported in the literature—of the STORM primitives on Gigabit Ethernet, Myrinet, InfiniBand, QsNet, and BlueGene/L. The data indicate that with or without hardware support, the STORM primitives 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.

We developed our initial implementation of STORM on Quadrics's QsNet network [25] because (a) we have convenient access to a QsNet cluster and (b) QsNet is a convenient platform for implementing the STORM primitives. There is a smaller semantic gap between the STORM and QsNet primitives than there is between the STORM primitives and those offered by some of the other networks to which we have ready access. In particular, QsNet has hardware support for atomic transactions, PUT operations, and local and remote events. Algorithms 10–12 present pseudocode for the QsNet implementation of the STORM mechanisms. XFER-AND-SIGNAL (Algorithm 10) follows directly from QsNet's multicast PUT operation and TEST-EVENT (Algorithm 11) is a trivial application of QsNet's event-waiting primitive. Algorithm 12, COMPARE-AND-WRITE, uses a QsNet network conditional for the comparison operation and chains this in the same transaction to a call to XFER-AND-SIGNAL.

Algorithm 10 QsNet implementation of the STORM data-transfer primitive

```
function XFER-AND-SIGNAL ( P,            ▷ Set of processes to write to
                         global_addr,    ▷ Address in global address space to write to
                         local_addr,     ▷ Address in sender's address space to read from
                         size,           ▷ Number of bytes to transfer
                         remote_event,   ▷ Remote event to signal upon completion
                         local_event)    ▷ Local event to signal upon completion

  {Note that XFER-AND-SIGNAL is asynchronous. Status is returned by signaling a local event.}
  try
    {Multicast data using a trivial application of QsNet's DMA primitives. Important: The following PUT and
    SIGNAL calls (if any) must be executed as a single, atomic operation.}
    PUT (global_addr, local_addr, size)    ∀p ∈ P
    if remote_event ≠ NULL then
      SIGNAL (remote_event, SUCCESS)    ∀p ∈ P
    if local_event ≠ NULL then
      SIGNAL (local_event, SUCCESS)
  except
    SIGNAL (local_event, FAILURE)
```

Algorithm 11 QsNet implementation of the STORM event-synchronization primitive

```
function TEST-EVENT ( event,      ▷ Local event to wait for
                      blocking)   ▷ Block if TRUE; poll if FALSE

  if blocking = TRUE then
    {Blocking maps trivially to QsNet's event-blocking primitive.}
    return BLOCK-ON-EVENT (event)
  else
    {Polling is implemented by directly examining the a hardware-set field of QsNet's event data structure.}
    if event.num_pending_signals ≥ 1 then
      return TRUE
    else
      return FALSE
```

Algorithms 10–12 demonstrate that STORM's primitives are straightforward to implement. Even on networks with no collectives support, XFER-AND-SIGNAL and COMPARE-AND-WRITE can be implemented with scalable, logarithmic-time broadcast and reduction algorithms [12].

4 Analysis

In this section, we analyze STORM's performance. 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).

Algorithm 12 QsNet implementation of the STORM global-comparison primitive

```
function COMPARE-AND-WRITE ( P,                    ▷ Set of processes to write to
                             global_var_R,         ▷ Variable to compare
                             relation,             ▷ One of {≥, <, =, ≠}
                             value_R,              ▷ Value to compare to
                             global_var_W,         ▷ Variable to write if relation is TRUE
                             value_W)              ▷ Value to write to global_var_W

  try
    if global_var_W = NULL then
      {Multicast a QsNet network conditional.}
      return (global_var_R ⟨relation⟩ value_R   ∀p ∈ P)
    else
      {Multicast a QsNet network conditional with a chained PUT transfer. This results in a single, atomic, QsNet
      network operation.}
      if global_var_R ⟨relation⟩ value_R   ∀p ∈ P then
        XFER-AND-SIGNAL (P, global_var_W, value, ||integer||, NULL, xfer_finished)
        return TEST-EVENT (xfer_finished, TRUE)
      else
        return FALSE
  except
    return FAILURE
```

4.1 Experimental Framework

We evaluated STORM on a 256-processor Alpha cluster with a QsNet network [25]. The cluster comprises 64 ES40 nodes, each having four 667 MHz, Alpha EV67 processors, 8 GB of RAM, two QM-400 Elan 3 NICs on independent 66 MHz, 64-bit PCI buses, running Red Hat Linux 7.1. At the time of the evaluation, this cluster was rated as the world's 83rd fastest supercomputer [34]. The experiments we review here relate to job-launching and multiprogramming mechanisms.

4.2 Job Launching Time

We first study the overhead associated with launching jobs with STORM and analyze STORM's scalability with the size of the binary and the number of PEs. We use the approach taken by Brightwell et al. in their study of job launching on Cplant [3], viz., we measure the time it takes to run a do-nothing program of size 4 MB, 8 MB, or 12 MB that terminates immediately.[2]

[2]The program contains a static array, which pads the binary image to the desired size, although the total dynamic memory usage may exceed that. The target sizes were selected as representative of large, scientific applications at LANL.

4.2.1 Launch times in STORM

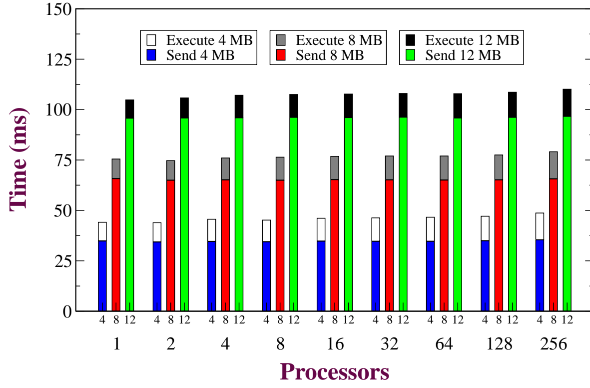
STORM logically divides the job-launching task into two separate operations: The transferal (reading + broadcasting + writing + notifying the MM) of the binary image and the actual execution, which includes sending a job-launch command, forking the job, waiting for its termination, and reporting back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

Figure 2(a) shows the time needed to transfer and execute a do-nothing application of sizes 4 MB, 8 MB, and 12 MB on 1–256 processors. Observe that the send times are proportional to the binary size but grow only slowly with the number of nodes. This is explained by the highly scalable algorithms and hardware broadcast that are used for the send operation. On the other hand, the execution times are independent of the binary size but grow more rapidly with the number of nodes. The reason for this growth is performance skew which is caused by OS overhead and accumulated by the processes in the job [27]. In the largest configuration tested, a 12 MB file can be launched on all 64 nodes within 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[3]) and an average execution time of 14 ms. In Section 4.4 we analyze in depth the launch-time scalability.

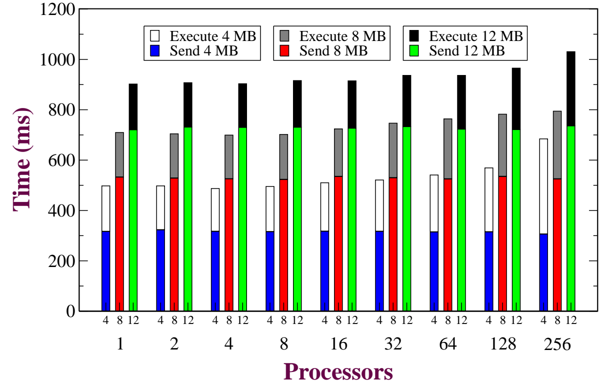
4.2.2 Launching on a loaded system

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 to introduce CPU contention. The second program repeatedly issues point-to-point messages between pairs of processes to introduce network contention. Both programs are run on all 256 processors. The following experiments are the same as those used in Section 4.2.1 but with one of the load-inducing programs simultaneously running on all nodes of the cluster.

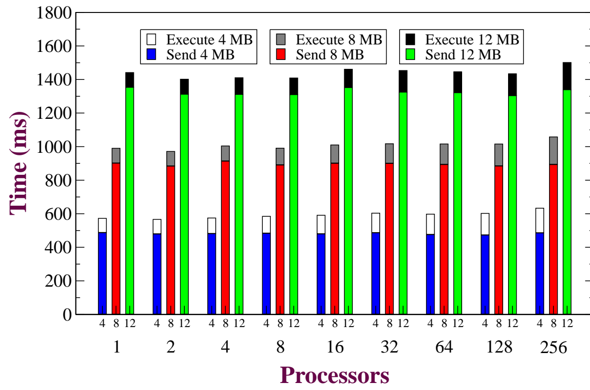
[3] The binary transfer does not include the source node.



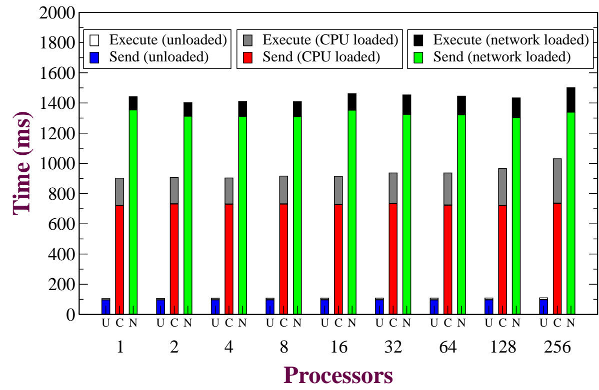
(a) Unloaded system



(b) CPU-loaded system



(c) Network-loaded system



(d) Launch summary 12 MB

Figure 2: Send and execute times for a 4 MB, 8 MB, and 12 MB binary

Figure 2(b) shows the results of launching the same three binaries while the CPU-consuming program is running in the background. In this case, STORM must make a scheduling decision every timeslice and notify the MM when the application terminates. (Note the different scale on the y axis from that in Figure 2(a).) Observe that CPU load exacts a large increase in both send time and execution time. The launch and execution time is now a second in the largest configuration and with a 12 MB binary. This increase in time is due to the interference of the computation with the I/O activities (reads and writes). Because the STORM NM and PL dæmons plus the application plus the CPU-consuming program utilize all four CPUs, there are no CPUs remaining to service the NIC's I/O-handling process. The result is the sensitivity to CPU load depicted in Figure 2(b).

The second test is particularly relevant to STORM because, as a previous networking study [23] shows, a heavily-loaded network can have an adverse effect on collective-communication perfor-

mance. In Figure 2(c), we can see how running the network-loading program in the background affects the launch time of the test binaries. (Again, note the different *y*-axis scale.) The execution time does increase to 160 ms in the worst case, caused mostly by the increased delays in the collection of the termination info. However, it increases less than in the previous experiment. In contrast, the send operation is considerably slower than on a CPU-loaded or unloaded system. This agrees with the networking study, as the the send operation is implemented with a collective operation.

Figure 2(d) summarizes the differences 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. Even in the worst case, with a network-loaded system, it still takes only 1½ seconds to launch a 12 MB file on 256 processors.

4.3 Gang Scheduling Performance

Although STORM supports a variety of process scheduling algorithms—with more under development—we have chosen to focus our evaluation specifically on GS [8], which is one of the most popular coscheduling algorithms. The following are the important issues regarding GS:

Effect of timeslice on overhead Smaller timeslices yield better response time at the cost of decreased throughput (caused by scheduling overhead which cannot be amortized). In Section 4.3.1 we show that STORM's scheduling overhead is so low that STORM can support workstation time quanta with virtually no performance penalty.

Node scalability Because GS requires global coordination, the cost of enacting a global decision frequently increases with the number of processors. Section 4.3.2 demonstrates that STORM exhibits such low overhead that applications running on large clusters can be coscheduled almost as rapidly as small clusters.

Effect of MPL The multiprogramming level (MPL) is the amount of oversubscription of processors to processes. Ideally, if there are P processes per processor (i.e., MPL $= P$), the turnaround time will be P times what it would be with a single process per processor (i.e., MPL $= 1$). In practice, schedulers require a certain amount of time to switch processes, which causes performance degradation. Also, the process context switch can destroy

the working set that resides in the cache memory. Section 4.3.3 provides data showing that application performance under STORM is not harmed by increased MPL; an MPL of P will cause applications to complete only P times slower than with an MPL of 1.

The application we use for our experiments in this section is Sweep3D [18], a time-independent, Cartesian-grid, single-group, "discrete ordinates", deterministic, particle-transport code which is representative of the DOE Advanced Simulation and Computing (ASC) 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 realistic simulations on current DOE systems.

In tests that involve an MPL of more than one, we further stress the scheduler by—somewhat unrealistically—launching all of the jobs simultaneously.

4.3.1 Effect of Time Quantum

As a first gang-scheduling experiment, we analyze the range of usable timeslice values to better understand the limits of STORM's gang scheduler. Figure 3 shows the average run time of the jobs for various timeslice values, from 300 µs to 8 s, running on 32 nodes/64 PEs. The smallest timeslice value that the scheduler can handle gracefully is ∼300 µs, below which the NM cannot process the incoming strobe messages at the rate they arrive. More importantly, even with a timeslice as small as 2 ms, STORM can still run multiple concurrent instances of an application with virtually no performance degradation relative to a single instance of the application.[4] This timeslice is an order of magnitude smaller than the local Linux scheduler's quantum and multiple orders of magnitude better than the smallest time quanta that conventional gang schedulers can handle with no performance penalties [11]. 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) [24], implicit coscheduling (ICS) [1], and periodic boost (PB) [22]. Because STORM can handle small time quanta with no performance penalty we chose the value of 50 ms for the next

[4] This result is also influenced by the poor memory locality of Sweep3D; running multiple processes on the same processor does not further pollute their caches.

sets of experiments. This value provides a fairly responsive system yet with minimal overhead.

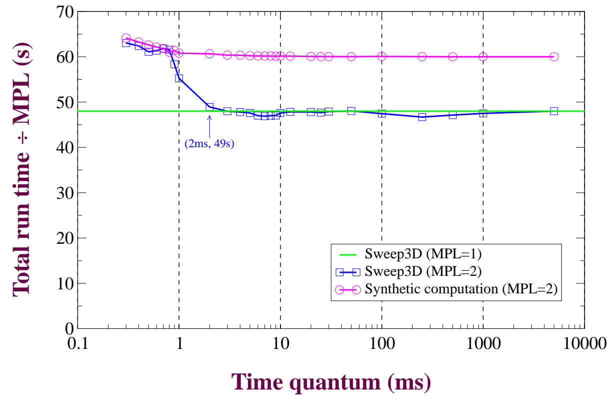


Figure 3: Effect of time quantum with an MPL of 2, on 32 nodes

4.3.2 Node Scalability

An important metric of a resource manager is its scalability with the number of nodes. To test this, we measured the effect on application run time when running on an increasing number of nodes.

Figure 4 shows the results for running the programs on varying number of nodes in the range 1–64 for MPL values of 1 and 2 and a timeslice of 50 ms. (Results for MPL 2 are normalized by dividing the total runtime of all jobs by 2.) We can observe that there is no visible increase in either the application run time or overhead with the increase in the number of nodes.

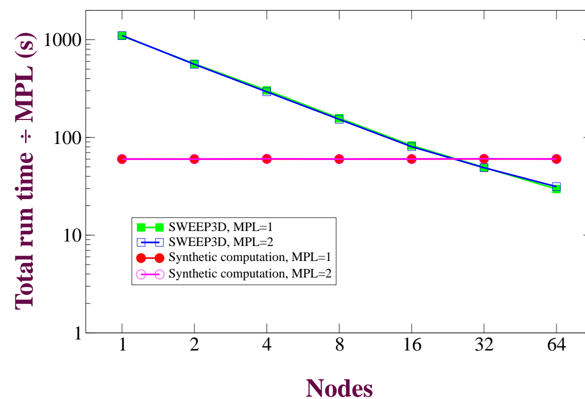


Figure 4: Effect of node scalability, varying the number of nodes in the range 1–64

4.3.3 Effect of MPL

Another important property of a gang scheduler is the overhead incurred by a context-switch operation. Context switches can cause performance degradation due to loss of cache state, synchronization difficulties across nodes, and the need to change the communication context gracefully, including the handling of in-transit messages. The context switch in STORM requires very little computation to determine the next process to run, suspend the current process, and resume the next one. This is actually less work than the UNIX scheduler typically takes for a context switch [32], so we can hypothesize that it incurs little overhead. To verify this hypothesis, we measure the effect of the overhead incurred by the scheduler on Sweep3D. Figure 5 shows the results of running 1, 2, 4, or 8 jobs together, with a timeslice of 50 ms. All jobs were launched concurrently and run on 32 nodes. It can clearly be seen that the scheduling overhead is minimal.

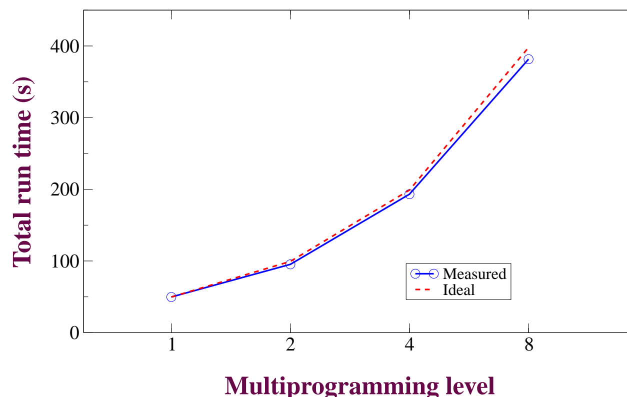


Figure 5: Effect of multiprogramming level on run time

4.4 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 to cluster configurations containing thousands of processing nodes.

4.4.1 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 [30], from a local hard disk, or from a RAM disk.[5] In our cluster, the NIC can read a file directly from the RAM disk at 218 MB/s. In previous work [12] we measured the bandwidth achieved when the NIC—with assistance from a lightweight process on the host—reads a 12 MB file from NFS, local disk, and RAM disk into either a host- or NIC-resident buffer. We found that it makes little difference whether the target buffers reside in main memory (11.4 MB/s and 30.5 MB/s respectively) or NIC memory (11.2 MB/s and 31.5 MB/s respectively). However, when reading from a (fast) RAM disk, keeping data buffers in main memory gives a bandwidth of 218 MB/s as opposed to only 120 MB/s in Elan memory.

Broadcast time This is the time to broadcast the binary image to all of the compute nodes. If the file is read from a networked 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 disseminate the file, as in ParPar [17], Cplant [3], BProc [15] or STORM, broadcast time can be measured separately from the other components of the total launch time. QsNet's 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. The hardware broadcast on 64 nodes 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 [12].

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.

Execution overhead Some of the time needed to launch a job in STORM is spent waiting for a time slot in which to run the job and collect the termination information in the management node. In our experiments the execution overhead is about 10 ms.

[5]A RAM disk is a segment of RAM that mimics a disk-based filesystem. RAM disks provide better performance than mechanical media but make the corresponding amount of RAM unavailable to applications.

Timeslice overhead In addition, events such as process termination are collected by the MM at heartbeat intervals only, so a delay of few heartbeat quanta can be spent in MM overhead.

The overall launch time T_{launch} can be expressed by the following equation,

$$T_{launch} = T_{transfer} + T_{exec} + T_{timeslice} \quad (1)$$

where $T_{transfer}$ represents the binary transfer delay, T_{exec} the execution overhead and $T_{timeslice}$ the overhead induced by STORM's scheduling policy.

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. To optimize the overall bandwidth of the pipeline, $BW_{transfer}$, we need to maximize the bandwidth of each single stage. $BW_{transfer}$ is bounded above by the bandwidth of the slowest stage of the pipeline:

$$BW_{launch} \leq \min(BW_{read}, BW_{broadcast}, BW_{write}) = \min(BW_{read}, BW_{broadcast}) \quad (2)$$

The buffers into which data is read and from which data is broadcast can reside in main memory or NIC memory. We have seen that reading into main memory is faster, while 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 determined empirically the optimal chunk size and number of buffer slots (i.e., the receive-queue length) for our cluster in a prior publication [12]. The communication protocol is largely insensitive to the number of slots, and the best performance is obtained with two slots of 512 KB. Increasing the number of slots does not provide any extra performance because doing so generates more TLB misses in the NIC's virtual memory hardware.

Figure 2(a) showed that the transfer time of a 12 MB binary is about 96 ms. Of those 96 ms, 4 ms are owed to skew caused by the OS overhead and the way that STORM dæmons act only on

heartbeat intervals (1 ms). The remaining 92 ms is 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 host process which services TLB misses and performs file accesses on behalf of the NIC.

Figure 6 illustrates all of the steps involved in the file-transfer protocol and indicates the performance of each stage of the pipeline. The file transfer protocol is initiated by the master node, which broadcasts a descriptor containing information about the binary: size, destination filename and directory, access rights, etc. The master opens the source file in read mode and each slave open the destination file in write mode ("Open file" in Figure 6). In the main loop, the master reads a file chunk from the filesystem ("Read chunk"), waits until *all* of the slaves are ready to accept it ("Await space"), multicasts the chunk to all of the slaves ("Send chunk"), and waits for an acknowledgment from the network ("Await sent"). Note that the master overlaps the sending of one chunk with the reading of the subsequent chunk. The slaves perform the complementary operations from the master; they repeatedly wait for a chunk from the master ("Await received") and write it to disk ("Write chunk"). The filesystem is the bottleneck in the file transfer. All of the network operations (communication and flow control) take microseconds to complete, while most of the filesystem operations have latencies measured in milliseconds.

4.4.2 Scalability Analysis

Because all STORM functionality is based on three mechanisms the scalability of these primitives determines the scalability of STORM as a whole. In fact, AWAIT-SIGNAL is a local operation, so scalability is actually determined only by the remaining two mechanisms.

Scalability of COMPARE-AND-WRITE We analyzed the scalability of QsNet's barrier synchronization (on which COMPARE-AND-WRITE is based) on the ASCI Q machine [26], a cluster with 1024 nodes/4,096 processors but otherwise identical to our cluster. As shown in Figure 7, latency grows by a negligible amount—about 6 μs—across a range of 1024 nodes. This is a reliable indicator that COMPARE-AND-WRITE, when implemented with the same hardware mechanism, will scale as efficiently.

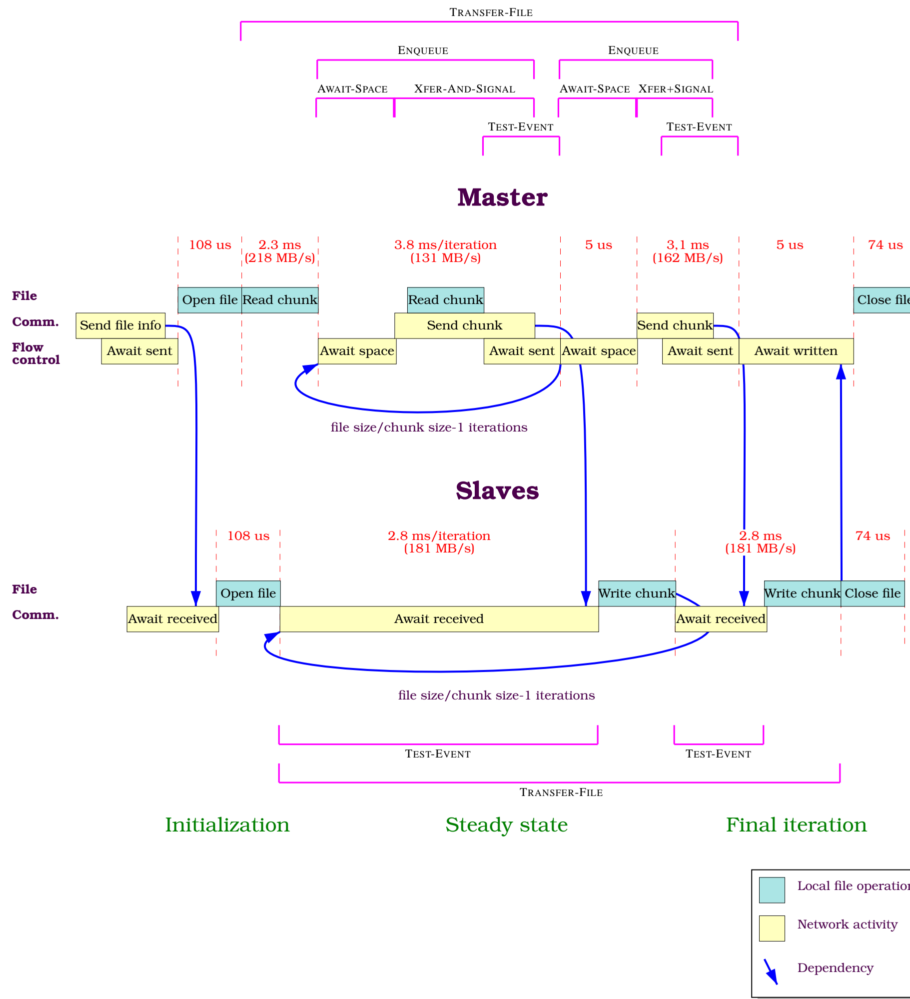


Figure 6: Transmission pipeline

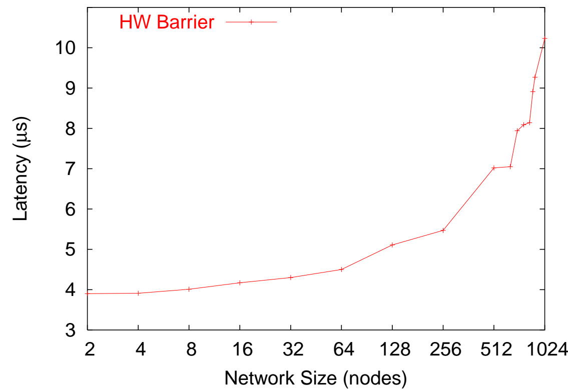


Figure 7: Barrier synchronization latency as a function of the number of nodes, ASCI Q [26]

Scalability of XFER-AND-SIGNAL To determine the scalability of XFER-AND-SIGNAL to a large number of nodes, we need to evaluate carefully the communication performance of the hardware broadcast, consider details of the hardware flow control in the network, and take into account the wire and switch delays. QsNet 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 packet $i-1$. Given that the maximum transmission unit of the QsNet network is only 320 bytes, 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 hence a reduction of the asymptotic bandwidth.

QsNet's end-to-end flow-control algorithm is based in an acknowledgment token sent by the destination NIC. The Elan's DMA engine uses overlapped packet ACK mode to maximize link utilization [28]. In this case, the destination NIC sends the ACK token immediately after receiving the packet header. If the token arrives at the source NIC while the packet body is still being transmitted, the next packet in the sequence can proceed without delay. Otherwise, the protocol introduces a transmission gap (i.e., delay) before injecting the following packet.

Equation 3 describes the asymptotic bandwidth of QsNet as a function of the maximum cable length and the number of switches that a packet traverses in the worst case. The equation distinguishes the case where packets can be pipelined without interruptions and the other case, in which the combination of wire and the switch delays introduces communication gaps.

$$BW_{QsNet}(\text{Cable}, \text{Switches}) = \frac{\text{Packet size}}{\max(T_{packet}, T_{base} + 2 \times \text{Cable} \times T_{cable} + \text{Switches} \times T_{switch})} \quad (3)$$

Table 2 describes the components of Equation 3 and provides an estimate of their values. This analytical model was used in the procurement of the ASCI Q machine [4] at Los Alamos National Laboratory and has been validated on several network configurations with a prediction error of less

Table 2: Legend of terms used in the STORM scalability model

Component	Description	Value
Cable	Maximum cable length between any pairs of nodes	*input parameter*
Switches	Maximum number of switches crossed by a packet	*input parameter*
Packet size	Maximum packet size	320 bytes
T_{packet}	Minimum packet delay at peak bandwidth	953 ns
T_{base}	Base delay for packet processing	580 ns
T_{cable}	Cable delay per meter	3.94 ns/m
T_{switch}	Sum of the forward data delay and ack. delay	73 ns

Table 3: Bandwidth scalability

				Cable length (meters)								
Nodes	Processors	Stages	Switches	10	20	30	40	50	60	70	80	90
4	16	1	1	**320**	320	320	315	291	271	253	238	224
16	64	2	3	**320**	319	295	274	256	240	226	213	202
64	256	3	5	**298**	277	258	242	228	215	204	194	184
256	1024	4	7	261	**244**	230	217	206	195	186	177	170
1024	4096	5	9	232	219	207	197	**187**	178	171	163	157
4096	16384	6	11	209	198	188	180	172	164	158	152	**146**

than 5%. Table 3 shows the asymptotic bandwidth BW_{QsNet} for networks with up to 4,096 nodes and physical diameters of up to 90 meters.

To make BW_{QsNet}—and, by consequence, $BW_{broadcast}$—dependent upon only 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 computers in the cluster are arranged in a square. 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,[6] 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). The following equation therefore provides a conservative estimate of the diameter in meters as a function of the number of nodes:

$$\text{Diameter}(nodes) = \sqrt{2 \times nodes} \tag{4}$$

In a quaternary fat tree, the maximum number of switches traversed by a packet can be ex-

[6] See, for example, the photograph of ASCI Q at `http://www.lanl.gov/asci/`.

pressed as a function of the number of nodes:

$$\text{Switches}(nodes) = (2 \times \log_4(nodes)) - 1 \tag{5}$$

By replacing the cable length and the number of switches in Equation 3, we obtain the asymptotic bandwidth BW_{QsNet} as a function of the number of nodes:

$$BW_{QsNet}(nodes) = \frac{\text{Packet size}}{\max(T_{packet}, T_{base} + 2 \times \sqrt{2 \times nodes} \times T_{cable} + [(2 \times \log_4(nodes)) - 1] \times T_{switch})} \tag{6}$$

Scalability of the Binary Transfer Protocol We now consider a model of the launch time for a 12 MB executable. The model contains three 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, which is about 10 ms. The third part is the timeslice overhead, the time that is wasted in OS overhead and waiting for the end of the STORM timeslices. The launch-time model indicates that

$$T_{launch}(nodes) = \frac{12}{BW_{transfer}(nodes)} + T_{exec} + T_{timeslice} \quad . \tag{7}$$

We now apply this model to two node configurations. The first one, represented by Equation 8, represents our current cluster which is based on ES40 AlphaServers that can deliver at most 131 MB/s over the I/O bus. The second configuration, Equation 9, 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).

$$BW^{ES40}_{transfer}(nodes) = \min(131, BW_{broadcast}(nodes)) \tag{8}$$

$$BW^{ideal}_{transfer}(nodes) = BW_{broadcast}(nodes) \tag{9}$$

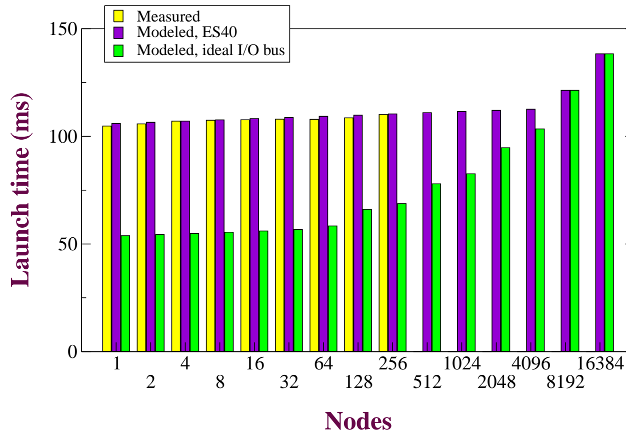


Figure 8: Measured and estimated launch times

Figure 8 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 executable can be launched in 135 ms on 16,384 nodes. The graph also shows the expected launch times in 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, which is the network broadcast bandwidth.

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. The CM-5's system software relied upon a custom-designed control network for RM tasks [20]. In contrast, STORM provides scalable job launching and process scheduling on commodity hardware. We now examine others' attempts at improving these two RM functions.

5.1 Job launching

The ParPar cluster environment [17] addresses the problem of the distribution of control messages from a management node to a set of clients. ParPar utilizes a special-purpose multicast protocol, *reliable datagram multicast* (RDGM), which 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 an 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 and with relatively good scalability. Nevertheless, this is still significantly slower than the launch time of a sequential job on an individual workstation and enough to annoy users who are waiting for an interactive job to launch.

GLUnix [14] 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. The creators of GLUnix note that when forking a parallel job, the overhead in the master node increases by a small, but linear-time, amount: an average of 220 μs per client node. Extrapolating, this implies just over 50 seconds to launch a job on 4,096 nodes (16,384 processors).

When GLUnix launches a job, remote execution messages are sent from the management node to all of the dæmons that will run the job. Each of these dæmons generates a reply message, indicating success or failure. When performing remote execution to more than 32 nodes over switched Ethernet, the replies from earlier dæmons in the communication schedule collide with the remote execution requests sent to later dæmons [14]. This causes a substantial performance degradation. STORM, however, uses network conditionals [25], which utilize a combining tree to reduce network contention and improve performance and scalability.

Scalability problems are already evident in ASC-scale machines (thousands of nodes). The Computational Plant (Cplant) project [29] utilizes several large-scale commodity-based clusters. To enhance scalability, Cplant uses a high-performance interconnect, Myrinet [2], and a custom, communication protocol based on Portals [3]. When Cplant's RM system launches a job, it first identifies a group of active worker nodes, organizes them into a logical tree structure, and then fans out the executable to the nodes. Experimental results show that a large, parallel application can be launched on a 1010-node cluster in about 20 seconds [3]. Cplant is the closest project in spirit to ours, in that it identifies poor RM performance as a problem worth studying and approaches the problem by replacing a traditionally nonscalable algorithm with a scalable one.

BProc [15], 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 functions STORM uses to transmit executable files (Algorithms 6 and 7 in Section 2.4) can also be used to transmit data files. BProc has no equivalent mechanism, although a cluster could certainly use BProc for its single-system-image features and STORM for the underlying communication protocols.

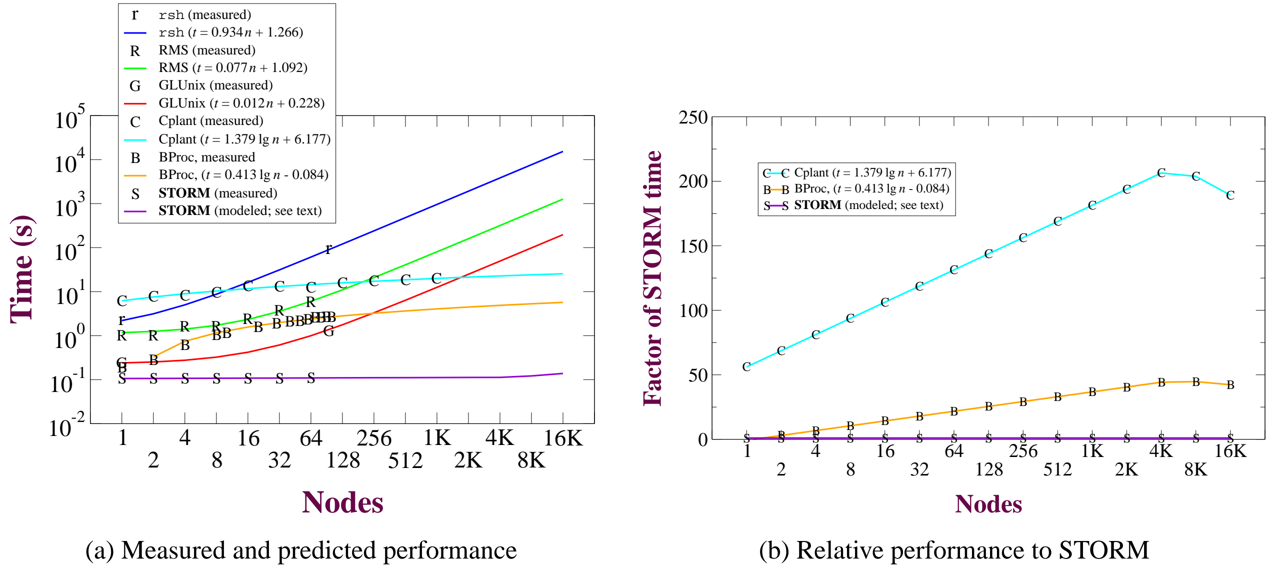
Table 4 shows a sampling of job-launch times found in the literature; Table 5 presents the same data extrapolated out to 4,096 nodes (twice the size of ASCI Q [4]); and Figure 9(a) graphs both the measured and extrapolated (to 16,384 nodes) data. Although the different cluster types and sizes make the comparison imprecise, these tables and figures give at least a general indication that STORM does, in fact, provide a significant performance improvement over previous works.

Table 4: 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 [14]
RMS	5.9	seconds to launch a 12 MB job on 64 nodes [12]
GLUnix	1.3	seconds to launch a minimal job on 95 nodes [14]
Cplant	20	seconds to launch a 12 MB job on 1,010 nodes [3]
BProc	2.7	seconds to launch a 12 MB job on 100 nodes [15]
STORM	0.11	seconds to launch a 12 MB job on 64 nodes

Table 5: Extrapolated job-launch times

Resource manager	Job-launch time extrapolated to 4,096 nodes		
`rsh`	3827.10	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 4.4)



(a) Measured and predicted performance

(b) Relative performance to STORM

Figure 9: Job-launching performance comparison

To clarify the performance improvement provided by STORM, Figure 9(b) renormalizes the extrapolated Cplant [3] and BProc [15] 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. The figure shows a decrease in the Cplant and BProc slowdown at 4,096 nodes. This is an artifact of the conservative performance model we used for STORM in Section 4.4, which indicates decreased network bandwidth as cluster sizes—and hence, cable lengths—increase. We extrapolated the performance of Cplant, BProc, and all of the other job-launchers presented in Table 4, Table 5, and Figure 9(a) under the unrealistic assumption that network performance scales indefinitely. Nevertheless, even though the STORM model is more conservative than the other models, the crossover point between BProc and STORM is expected to be on a system containing approximately 1 billion nodes, and the crossover point between Cplant and STORM is expected to be on a system containing approximately 17 billion 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, 7]. 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, in order 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.

As far as traditional gang-schedulers are concerned, the SCore-D scheduler [16] is one of the fastest. By employing help from the messaging layer, PM [33], 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 $\sim$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 application and thawing another. As a result, STORM can gang-schedule applications with no noticeable overhead when using quanta as small as 2 ms.

Prior schedulers for large-scale computer systems sometimes required time quanta on the order of minutes to amortize scheduler overhead [7, 21]. Table 6 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 6: 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) [11]
SCore-D	100 milliseconds on 64 nodes (2% slowdown) [16]
STORM	2 milliseconds on 64 nodes (no observable slowdown)

6 Conclusions

While the purpose of a cluster is to run applications, it is the goal of the resource-management system to ensure that these applications load quickly, make efficient use of cluster resources, and interact to user input with small response times. While resource management is comparatively simple to do well on a small-scale cluster, it is more challenging on a large-scale cluster. 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; and, 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 presented STORM, a lightweight, flexible, and scalable environment for performing resource management in large-scale clusters. In terms of both job launching and process scheduling, STORM is 1–2 orders of magnitude faster than the best reported results in the literature [15, 16]. The key to STORM's performance lies in its design methodology. Rather than implement heartbeat issuance, job launching, process scheduling, and other routines as separate entities, we designed those functions in terms of a small, common set of data-transfer and synchronization mechanisms: XFER-AND-SIGNAL, AWAIT-SIGNAL, 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 Quadrics network and demonstrated that STORM performs well on that cluster and is expected to 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. And 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.

By improving the performance of various resource-management functions by two 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-scale clusters usable and efficient.

References

[1] A. C. Arpaci-Dusseau. Implicit coscheduling: Coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, Aug. 2001.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawick, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[3] R. Brightwell and L. A. Fisk. Scalable parallel application launch on Cplant. In *Proceedings of IEEE/ACM Conference on Supercomputing (SC'01)*, Nov. 2001.

[4] Compaq High Performance Technical Computing Group. U.S. DOE selects Compaq to build ASCI Q. *HPTC News*, 17, Sept./Oct. 2000.

[5] D. G. Feitelson. Packing schemes for gang scheduling. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of the International Parallel Processing Symposium (IPPS'96), 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 89–110, Apr. 1996.

[6] D. G. Feitelson, A. Batat, G. Benhanokh, D. Er-El, Y. Etsion, A. Kavas, T. Klainer, U. Lublin, and M. Volovic. The ParPar system: A software MPP. In R. Buyya, editor, *High Performance Cluster Computing*, volume 1: Architectures and Systems, pages 758–774, 1999.

[7] D. G. Feitelson and M. A. Jette. Improved utilization and responsiveness with gang scheduling. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of the International Parallel Processing Symposium (IPPS'97), 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261, Apr. 1997.

[8] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, Dec. 1992.

[9] E. Frachtenberg, D. G. Feitelson, J. Fernandez-Peinador, and F. Petrini. Parallel job scheduling under dynamic workloads. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 208–227. Springer-Verlag, 2003.

[10] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez. Adaptive parallel job scheduling with flexible coscheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(11):1066–1077, Nov. 2005.

[11] E. Frachtenberg, F. Petrini, S. Coll, and W. Feng. Gang scheduling with lightweight user-level communication. In *Proceedings of the International Conference on Parallel Processing (ICPP'01), Workshop on Scheduling and Resource Management for Cluster Computing*, Sept. 2001.

[12] E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll. STORM: Lightning-fast resource management. In *Proceedings of the IEEE/ACM Conference on Supercomputing (SC'02)*, Nov. 16–22, 2002.

[13] H. Franke, J. Jann, J. E. Moreira, P. Pattnaik, and M. A. Jette. An evaluation of parallel job scheduling for ASCI Blue-Pacific. In *Proceedings of IEEE/ACM Conference on Supercomputing (SC'99)*, Nov. 1999.

[14] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson. GLUnix: a global layer Unix for a network of workstations. *Software—Practice and Experience*, 28(9):929–961, July 1998.

[15] E. Hendriks. BProc: The Beowulf distributed process space. In *Proceedings of ACM International Conference on Supercomputing (ICS'02)*, June 2002.

[16] A. Hori, H. Tezuka, and Y. Ishikawa. Highly efficient gang scheduling implementation. In *Proceedings of IEEE/ACM Conference on Supercomputing (SC'98)*, Nov. 1998.

[17] A. Kavas, D. Er-El, and D. G. Feitelson. Using multicast to pre-load jobs on the ParPar cluster. *Parallel Computing*, 27(3):315–327, Feb. 2001.

[18] 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.

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

[20] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, Mar. 1996.

[21] J. E. Moreira, H. Franke, W. Chan, L. L. Fong, M. A. Jette, and A. B. Yoo. A gang-scheduling system for ASCI Blue-Pacific. In *HPCN Europe*, pages 831–840, Apr. 1999.

[22] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. A closer look at coscheduling approaches for a network of workstations. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures, (SPAA'99)*, June 1999.

[23] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. Hardware- and software-based collective communication on the Quadrics network. In *Proceedings of the International Symposium on Network Computing and Applications (NCA'01)*, Oct. 2001.

[24] F. Petrini and W. Feng. Improved resource utilization with Buffered Coscheduling. *Journal of Parallel Algorithms and Applications*, 16:123–144, 2001.

[25] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, Jan./Feb. 2002.

[26] F. Petrini, J. Fernández, E. Frachtenberg, and S. Coll. Scalable collective communication on the ASCI Q machine. In *Proceedings of the Symposium on High Performance Interconnects (HotI'03)*, Aug. 2003.

[27] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of IEEE/ACM Conference on Supercomputing (SC'03)*, Nov. 2003.

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

[29] R. Riesen, R. Brightwell, L. A. Fisk, T. Hudson, J. Otto, and A. B. Maccabe. Cplant. In *Proceedings of the USENIX Annual Technical Conference, Second Extreme Linux Workshop*, June 1999.

[30] 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, Dec. 2000.

[31] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*, volume 1, The MPI Core. The MIT Press, Cambridge, Massachusetts, 2nd edition, Sept. 1998.

[32] J. H. Straathof, A. K. Thareja, and A. K. Agrawala. UNIX scheduling for large systems. In *Proceedings of the USENIX 1986 Winter Conference*, Jan. 1986.

[33] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An operating system coordinated high performance communication library. In B. Hertzberger and P. 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, Apr. 1997.

[34] Top 500 supercomputers. `http://www.top500.org/`.