Architectural Support for System Software on Large-Scale Clusters[*]

Juan Fernández Eitan Frachtenberg Fabrizio Petrini Kei Davis

Jose Carlos Sancho

CCS-3 Modeling, Algorithms, and Informatics Group

Computer and Computational Sciences (CCS) Division

Los Alamos National Laboratory, Los Alamos, NM 87545 USA

{juanf,eitanf,fabrizio,kei,jcsancho}@lanl.gov

Abstract

Scalable management of distributed resources is one of the major challenges in deployment of large-scale clusters. Management includes transparent fault tolerance, efficient allocation of resources, and support for all the needs of parallel applications: parallel I/O, deterministic behavior, and responsiveness. These requirements are daunting with commodity hardware and operating systems since they were not designed to support a global, single management view of a large-scale system. In this paper we propose a small set of hardware mechanisms in the cluster interconnect to facilitate the implementation of a simple yet powerful global operating system. This system, which can be thought of as a coarse-grain SIMD operating system, allows commodity clusters to grow to thousands of nodes while still retaining the usability and responsiveness of the single-node workstation. Our results on a software prototype show that it is possible to implement efficient and scalable system software using the proposed set of mechanisms.

***Keywords**: Cluster computing, cluster operating system, fault tolerance, network hardware, debuggability, resource management.*

1 Introduction

Although workstation clusters are a common platform for high-performance computing (HPC), they remain more difficult to manage than single-node systems or symmetric multiprocessors. Furthermore, as cluster size increases, the role of the

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

system software—essentially all of the code that runs on a cluster other than the applications—becomes increasingly more important. The system software's main components include the communication library, the resource manager, the parallel file system, the cluster monitoring software, and the software infrastructure to implement fault tolerance. The quality of the system software not only affects application performance but also the cost of ownership of such machines.

Interconnection network and system software design for high-performance computational clusters traditionally rely on a common abstract machine that clearly separates their roles. This abstract machine sees the network simply as a mechanism for moving information from one processing node to another with a performance expressed by latency and bandwidth. This functional interface is simple and general enough to develop most system software, and can be implemented in several different ways. The success of this interface also relies on the implicit assumption that any performance improvement in both latency and bandwidth can be directly inherited by the system software.

Abstract interfaces may change to exploit new hardware capabilities. For example, in the last decade this basic abstract interface has been augmented to exploit distributed shared memory. This approach was pioneered by communication layers such as Active Messages [33] that emulated a virtual address space by using physically addressed network interfaces. Active Messages proved that the use of a global shared memory could greatly simplify the communication library and increase its performance. This successful experience influenced the design of the Cray T3D and the Meiko CS-2, that provided remote direct memory access (RDMA). A global, virtually addressed shared memory is now a common feature in networks as Quadrics [25] or Infiniband [23].

In this paper we try to answer the following question. *What hardware features, and thus which abstract interface, should the interconnection network provide to the system software designers?*

We argue that the efficient and scalable hardware implementation of a small set of network primitives that perform global queries and distribution of data is essential to support most system software and user applications. These primitives can be easily implemented in hardware with current technology and can greatly reduce the complexity of most system software. In a sense they represent the least common denominator of the various components of the cluster software, and the backbone to integrate a collection of local operating systems (OS) into a single, global OS.

This paper makes the following contributions. First, it makes the case for the importance and the potential of having these primitives for global coordination implemented in hardware. Second, it outlines a new approach to system management, *buffered coscheduling* (BCS) [24], that is based on these primitives. One of BCS's goals is to simplify system software design by enforcing global coordination of all the activities in a cluster. Third, a series of case studies shows how important parts of the system software can benefit from these primitives. We provide experimental evidence that resource management and job scheduling can can be implemented on thousands of nodes and achieve the same level of responsiveness as a dedicated workstation, without any significant increase in complexity. Finally, we describe how a popular communication library, the Message Passing Interface (MPI), can be implemented with these global coordination primitives. The proposed implementation is so simple that it can run almost entirely on the network interface card (NIC) as fast as the production-quality MPI.

The rest of the paper is organized as follows. The next section describes some of the system tasks required on clusters and the problems that need to be addressed to achieve responsive and scalable environments. Section 3 details the core primitives and mechanisms that constitute the building blocks of our proposed scalable system software. Section 4 presents several case studies and reports several experimental results obtained on our working software prototype on three different clusters. Section 5 concludes and offers directions for future research.

2. Challenges in the Design of System Software

Many of today's fastest supercomputers are composed of commercial-off-the-shelf (COTS) workstations connected by a fast interconnect. These nodes typically use commodity operating systems such as Linux to provide an hardware abstraction layer to programmers and users. These OSes are quite adequate for the development, debugging, and running of applications on independent workstations and small clusters. However, such a solution is often insufficient for running demanding HPC applications in large clusters.

Common cluster solutions include middleware extensions on top of the workstation operating system, such as the MPI communication library [29] to provide some of the functionality required by these applications. These components tend to have many dependencies and their modular design may lead to redundancy of functionality. For example, both the communication library and the parallel file system used by the HPC applications implement their own communication protocols. Even worse, some desired features such as multiprogramming, garbage collection, or automatic checkpointing are either not supported at all or are very costly in terms of both development costs and overall performance degradation. Consequently, there is a growing gap between the services enjoyed on a workstation and those provided to HPC users, forcing many application developers to complement these services in their application. Table 1 overviews several of these gaps in terms of the basic functionality required to develop, debug, and effectively use parallel applications. Next we discuss some of the gaps in detail.

Job launching. Virtually all modern workstations allow simple and quick launching of jobs, thus enabling interactive tasks such as debugging sessions or visual applications. In contrast, clusters offer no standard mechanism for launching parallel jobs. Typical workarounds rely on shell scripts or particular middleware. Job launching times can range anywhere from seconds to hours and are usually far from interactive. Many solutions were suggested in the past to this problem, ranging from the use of generic tools such as rsh and NFS, to sophisticated programs such as RMS [10], GLUnix [13], Cplant [27], BProc [15], and SLURM [17]. Some of these systems use tree-based algorithms to disseminate binary images and data to compute nodes, which can shorten job-launch times significantly. However, because of their reliance on software mechanisms, with larger clusters (thousands of nodes) these systems may be expected to take many seconds or minutes to

Table 1. System tasks in workstations and clusters

Characteristic	Workstation	Cluster
Job Launching	Operating system (OS)	Scripts, middleware on top of OS
Job Scheduling	Timeshared by OS	Batch queued or gang scheduled with large quanta (seconds to minutes) using middleware
Communication	OS-supported standard IPC mechanisms and shared memory	Message Passing Library (MPI) or Data-Parallel Programming (e.g. HPF)
Storage	Standard file system	Custom parallel file system
Debuggability	Standard tools (reproducibility)	Parallel debugging tools (non-determinism)
Fault Tolerance	Little or none	Application / application-assisted checkpointing
Garbage collection (GC)	Run-time environment such as Java or Lisp	Global GC very difficult due to nondeterminism of data's live state [18]

launch parallel jobs.

Job scheduling. In the workstation world it is taken for granted that several applications can be run concurrently using time sharing, but this is rarely the case with clusters. Most middleware used for parallel job scheduling use simple versions of batch scheduling (or gang-scheduling at best). This affects both the user's experience of the machine, which is less responsive and interactive, and the system's utilization of available resources. Even systems that support gang scheduling typically revert to relatively high time quanta to hide the high overhead costs associated with context switching a parallel job in software.

The SCore-D [16] scheduler uses a combination of software and hardware to perform the global context switch relatively efficiently. A software multicast is used to synchronize the nodes and force them to flush the network state to allow each job the exclusive use of the network for the duration of its time slice. The flushing of the network context and the use of software multicast can have a detrimental effect on the time quanta when using a cluster size of more than a few hundreds of nodes. In the SHARE gang scheduler of the IBM SP2 [12], network context is switched by software, where messages that reach the wrong process are simply discarded. This incurs significant communication overhead as processes need to recover lost messages. The CM-5 had a gang-scheduling operating system (CMOST) and a hardware support mechanism for network preemption called All-Fall-Down [31]. In this system, all pending messages at the time of a context switch fall down to the nearest node regardless of destination. This creates noticeable delays when the messages need to be re-injected to the system. Even more significantly, this implies that message order and arrival time are completely unpredictable, making the system hard to debug and control. Other machines such as the Makbilan [7] and CMOST [31] also had some hardware support for context-switching. However, these specialized machines cost more and do not scale as well as contemporary COTS clusters.

Communication. User processes running in a workstation communicate with each other using standard interprocess communication mechanisms provided by the OS. While these may be rudimentary mechanisms that provide no high-level abstraction, because of their low synchronization requirements they are adequate for serial and coarse-grained distributed jobs. Unlike these jobs, HPC applications require a more expressive set of communication tools to keep the software development effort manageable.

The prevailing communication model for modern HPC applications is message passing, where processes use a communication library to send synchronous and asynchronous messages to each other. Of these libraries, the most commonly used are MPI [29] and PVM [30]. These libraries offer standards that facilitate portability across various cluster and MPP architectures. However, in order to improve the latency and bandwidth for single messages, much effort is required to tune these libraries to different platforms. Another problem is that these libraries offer low-level mechanisms that force the software developer to focus on implementation details, and make modeling application performance difficult. In order to simplify and abstract the communication performance of applications, several models have been suggested.

The well-known LogP model developed by Culler *et al.* [6] focuses on latency and bandwidth in asynchronous message passing systems. A higher level abstraction is the Bulk-Synchronous Parallel (BSP) model introduced by Valiant *et al.* [32]. Computation is divided into *supersteps* so that all messages sent in one superstep are delivered to the destination process at the beginning of the the next superstep. All the processes synchronize between two consecutive supersteps. This model constitutes the first attempt to optimize the communication pattern as a whole rather than optimizing latency and bandwidth for individual messages.

Determinism. Serial applications are much easier to debug compared to their parallel counterparts: their inherent determinism makes many problems easy to reproduce. In contrast, for a large parallel program the trace of just message passing may have a practically unbounded number of correct orderings; the difficulty of debugging an inherently non-deterministic, asynchronous system is exacerbated by interference by the debugging tools itself as it imposes constraints on execution (reduces non-determinism).

Fault tolerance. Non-determinism also makes fault tolerance using checkpointing challenging because the application is rarely known to be in a state wherein all processes and in-transit messages are synchronized. Fault tolerance on workstations is not considered a major problem and thus rarely addressed by the OS. On large clusters, however, where the high number of components results in a low mean time between failures and the amount of computation cycles invested in the program is significant, fault tolerance becomes one of the most critical issues. Still, there is no standard solution available, and many of the existing solutions rely on some application modifications.

Bosilca *et al.* introduced a system called MPICH-V [2] to address some of these problems. Their implementation of MPI uses uncoordinated checkpoint/rollback and distributed message logging to convalesce in case of a network fault. MPICH-V requires a rather complex runtime environment, partly due to messages in transit that need to be accounted for. The performance of MPICH-V varies with the application characteristics, sustaining a slowdown of up to 200% or more in some cases. To amortize some of this overhead, the authors use a checkpoint interval of $130s$.

We believe that with some minimal support from the hardware a relatively simple fault-tolerant system software can be implemented with much less overhead and much higher checkpoint frequency. To do that, we rely on global synchronization and scheduling of all system activities. In that case there are points along the execution of a parallel program in which all the allocated resources are in a known state, making it relatively straightforward to implement an algorithm to checkpoint the job in a safe way.

2.1 Designing a Parallel Operating System

The design, implementation, debugging, and optimization of system middleware for large-scale clusters is far from trivial, and potentially very time- and resource consuming [20]. System software is required to deal with one or more parallel jobs comprising thousands of processes each. Furthermore, each process may have several threads, open files, and outstanding messages at any given time. All these elements result in a large and complicated global machine state which in turn increases the complexity of the system software. The lack of global coordination is a major cause of the non-deterministic nature of parallel systems. The lack of synchronization also diminishes application performance, for example, when non-synchronized system dæmons introduce computational holes that can severely skew and impact fine-grained applications [26].

To address these issues, we promote the idea of a simple, global cluster OS that makes use of advanced network resources, just like any other HPC application. Our vision is that a cluster OS should behave like a SIMD (single-instruction-multiple-data) application, performing resource coordination in lockstep. We argue that performing this task scalably and at sub-millisecond granularity requires hardware support realizable by a small set of network mechanisms. Our goal in this study is to identify and describe these mechanisms. Using a prototype system on a network that supports most of these features, we present experimental results that indicate that a cluster OS can be scalable, powerful, and relatively simple to implement. We also discuss the gaps between our proposed mechanisms and the available hardware, suggesting ways to overcome these limitations.

3 Core Primitives and Mechanisms

In this section, we characterize the primitives and mechanisms that we consider essential in the development of system software for large-scale clusters. We then explain how to use these mechanisms to overcome the challenges raised in the previous section.

3.1 Suggested Mechanisms

The proposed architectural support consists of just three hardware-supported network primitives:

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. By global memory we refer to data at the same virtual address on all nodes. Depending on implementation, global data may reside in main or network-interface memory.

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

COMPARE-AND-WRITE Arithmetically compare a global variable on a node set to a local value. If the condition is true on *all* nodes, then (optionally) assign a new value to a (possibly different) global variable.

Note that 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 [22]. TEST-EVENT and COMPARE-AND-WRITE are traditional blocking operations, while 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. These semantics do not dictate whether the mechanisms are implemented by the host CPU or by a network co-processor. Nor do they require that TEST-EVENT yield the CPU (although not yielding the CPU may adversely affect system throughput).

3.2 Implementation and Portability

The three primitives presented above assume that the network hardware provides global, virtually addressable shared memory and RDMA. These features are present in several state-of-the-art networks like QsNet and Infiniband and their functionality has been extensively studied [23, 25]. While the physical implementation aspects of these primitives are outside the scope of this paper, we note that some or all of them have have already been implemented in several other interconnects, as shown in Table 3. They were originally designed to improve the communication performance of user applications. To the best of our knowledge their usage as an infrastructure for system software was not explored before this work.

Hardware support for multicast messages sent with XFER-AND-SIGNAL is needed to guarantee scalability for large-scale systems. Software approaches, while feasible for small clusters, do not scale to thousands of nodes. In our case, QsNet provides hardware-supported PUT/GET operations and events so that the implementation of XFER-AND-SIGNAL is straightforward.

COMPARE-AND-WRITE assumes that the network is able to return a single value to the calling process regardless of the number of queried nodes. Again, QsNet includes a hardware-supported global query operation that allows the implementation of COMPARE-AND-WRITE.

Table 2. Network mechanisms usage

Characteristic	Requirement	Solution
Job Launching	Data dissemination	XFER-AND-SIGNAL
	Flow control	COMPARE-AND-WRITE
	Termination detection	COMPARE-AND-WRITE
Job Scheduling	Heartbeat	XFER-AND-SIGNAL
	Context switch responsiveness	Prioritized messages / Multiple rails
Communication	PUT	XFER-AND-SIGNAL
	GET	XFER-AND-SIGNAL
	Barrier	COMPARE-AND-WRITE
	Broadcast	COMPARE-AND-WRITE + XFER-AND-SIGNAL
Storage	Metadata / file data transfer	XFER-AND-SIGNAL
Debuggability	Debug data transfer	XFER-AND-SIGNAL
	Debug synchronization	COMPARE-AND-WRITE
Fault Tolerance	Fault detection	COMPARE-AND-WRITE
	Checkpointing synchronization	COMPARE-AND-WRITE
	Checkpointing data transfer	XFER-AND-SIGNAL
Garbage Collection	Live state synchronization	Determinism and COMPARE-AND-WRITE

Table 3 demonstrates the expected performance of the mechanisms that are already implemented by several interconnect technologies. While several networks already support at least some of these mechanisms (which attests to their portability), we argue that they should become a standard part of every large-scale interconnect. We also stress that their implementation must exhibit scalability and high performance (in terms of bandwidth and latency) for them to be useful to the system software.

Table 3. Measured/expected performance of the core mechanisms for n nodes

Network	Comparison (μs)	Multicast (MB/s)
Gigabit Ethernet [28]	$46 \log n$	Not available
Myrinet [1, 4, 5]	$20 \log n$	$\sim 15n$
Infiniband [23]	$20 \log n$	Not available
QsNET ([25])	< 10	$> 150n$
BlueGene/L [14]	< 2	$700n$

3.3 System Software Requirements and Solutions

Next we examine the areas where current system software is lacking and explain how the proposed mechanisms can simplify the design and implementation of practical solutions. Table 2 summarizes these arguments.

Job Launching The traditional approach to job launching, including the distribution of executable and data files to cluster nodes, is a simple extension of single-node job launching: data is transmitted using network file systems such as NFS, and jobs are launched with scripts or simple utilities such as rsh or mpirun. These methods do not scale to large machines where the load on the network file system, and the time it would take to serially execute a binary on many nodes, make them inefficient and impractical. Several solutions have been proposed for this problem, all focusing on software tricks to reduce the distribution time. For example, Cplant and BProc both use their own tree-based algorithm to distribute data with latencies that are logarithmic in the number of nodes [3, 15]. While more portable than relying on hardware support, these solutions are significantly slower and not always simple to implement [11].

Decomposing job launching into simpler sub-tasks makes more clear that it need only require modest effort to make the process efficient and scalable:

- Executable and data distribution are no more than a multicast of packets from a file server to a set of nodes, and can be implemented using XFER-AND-SIGNAL. We may use COMPARE-AND-WRITE for flow control to prevent the multicast packets from overrunning the available buffers.

- Actual launching of a job can be achieved simply and efficiently by multicasting a control message to all the nodes that are allocated to the job by using XFER-AND-SIGNAL. In response the system software on each node would then fork the new processes and wait for their termination.

- The reporting of job termination can incur much overhead if each node sends a single message for every process that terminates. This problem can be solved by ensuring that all the processes of a job reach a common synchronization point upon termination (using COMPARE-AND-WRITE) before delivering a single message to the resource manager (using XFER-AND-SIGNAL).

Job Scheduling. Interactive response times from a scheduler are required to make a parallel machine as usable as a workstation. This in turn implies that the system must be able to perform preemptive context switching with the same latencies we have come to expect from single processor systems, that is, on the order of a few milliseconds. Such latencies however are virtually impossible to achieve without hardware support: the time required to coordinate a context switch over thousands of nodes can be prohibitively large in a software-only solution. A good example of this is shown by the work on the SCore-D software-only gang scheduler of Hori *et al.* [16]. They report that the time for switching the network context on a relatively small Myrinet cluster is more than two thirds of the total context switch time. Furthermore, the context switch message is propagated to the nodes using a software-based multicast tree, increasing in latency as the cluster grows. SCore-D has four separate, synchronized phases for each context switch, requiring about 200 $msec$ context-switch granularity to hide most of the overhead in a 64-node cluster. Finally, even though the system is able to efficiently context switch between different jobs, the coexistence of application traffic and synchronization messages in the network could unacceptably delay response to the latter. If this occurs even on a single node for even just a few milliseconds it will have a detrimental effect on the responsiveness of the entire system.

To overcome these problems the network should offer some capabilities to the software scheduler to prevent these delays. The ability to maintain multiple communication contexts alive in the network securely and reliably, without kernel intervention, is already implemented in some state-of-the-art networks like QsNet. Job context switching can be easily achieved by simply multicasting a control message or *heartbeat* to all the nodes in the network using XFER-AND-SIGNAL. One method of guaranteeing quality of service for synchronization messages is to have support for message prioritization. The current generation of many networks, including QsNet, does not yet support prioritized messages in hardware, so a workaround must be found to keep the system messages' latencies low. In our case, we exploit the fact that some of our clusters have dual networks (two rails), and use one rail exclusively for system messages so that they do not compete with application-induced traffic.

Determinism and fault tolerance. Hori *et al.* proposed a mechanism they called *network preemption* to facilitate such task as maintaining a known state of the cluster and context switching. We believe this mechanism is certainly necessary, but not sufficient, for an efficient solution to this problem. Even when a single application is running on the system (so that there is only one network context, and no preemption), messages can still be en route at different times and the system's state as a whole is not deterministic.

When the system globally coordinates all the application processes, parallel jobs can be led to evolve in a controlled manner. Global coordination can be easily implemented with XFER-AND-SIGNAL, and can be used to perform global scheduling of all the system resources. Determinism can be enforced by taking the same scheduling decisions between different executions. At the same time, the global coordination of all the system activities help to identify the states along the program execution in which it is safe to checkpoint the status.

Communication. Most of MPI's, TCP/IP's, and other communication protocols' services can be reduced to a rather basic set of communication primitives, e.g. point-to-point synchronous and asynchronous messages and multicasts. If the underlying primitives and the protocol reductions are implemented efficiently, scalably, and reliably by the hardware and cluster OS, respectively, the higher level protocol can also inherit the same benefits of scalability, performance, and reliability. In many cases, this reduction is very simple and can eliminate the need for many of the implementation quirks of protocols that need to run on a variety of network hardware.

To illustrate this strategy we have implemented a small subset of the MPI library, called BCS-MPI [9], which has sufficient functionality to support real applications. As is shown in the next section these applications have similar performance using BCS-MPI as using production-quality versions of MPI, but have the potential to benefit from control BCS-MPI.

4. Case Studies

To demonstrate our thesis that these mechanisms can be exploited by a scalable global OS we built a prototype resource-management system, called STORM, and tested it on three architectures. In all cases we used the Quadrics *Elan3* network as our interconnect because it supports most of the mechanisms described in Section 3. In this section we review the performance and scalability that can be obtained with these mechanisms on three tasks: job launching, job scheduling, and deterministic communication.[1]

4.1. Software Environment

Our prototype resource-management system is composed of a set of dæmons that run on the compute nodes and management node of a cluster [11]. It contains a network abstraction layer that uses the described mechanisms for executing tasks such as job launching, process coordination (e.g. gang-scheduling), and resource accounting. Although currently implemented as user-mode dæmons, we plan to fully incorporate the core functionality of STORM with the Linux kernel to obtain optimal performance and latencies. The code is relatively small at around 10,000 lines of C code for the core functionality.

In addition to resource management, the core primitives can be used to implement almost any communication protocol while still retaining the advantages of performance and determinism. Here we have implemented the previously mentioned BCS-MPI.. To use BCS-MPI applications simply need to be re-linked against the new libraries without any code modification. However, to achieve the best performance of BCS-MPI it can be beneficial to replace blocking communication calls such as MPI_Send() and MPI_Recv() with their non-blocking counterparts. This allows BCS-MPI to aggregate several communication calls together whenever possible, so improving the possibility of interleaving communication and computation.

In the following case studies we used both synthetic and real HPC applications. The applications SWEEP3D and SAGE are representative of two hydrodynamics codes from the ASCI workload [19, 21].

4.2. Hardware Environment

For the experimental evaluation we used three different clusters at LANL/CCS-3 to test our mechanisms on different processor architectures. The clusters are called *Crescendo*, *Accelerando*, and *Wolverine*. All clusters used a 128-port Quadrics Elite switch and Quadrics software library version 1.5.0-0. Table 4 summarizes the hardware comprising each cluster.

4.3. Job Launching

In this set of experiments we study the cost associated with launching jobs with STORM and analyze STORM's scalability with the size of the binary and the number of PEs on Wolverine. We use the approach taken by Brightwell *et al.* in their study

[1]In [11] we study in detail other properties of STORM's job scheduling and job launching abilities, and model their scalability.

Table 4. Cluster Description

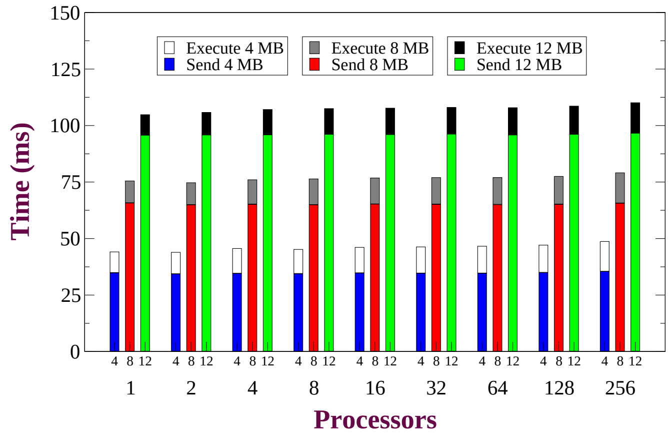
Component	Feature	*Crescendo* cluster	*Accelerando* cluster	*Wolverine* cluster
Node	Number×PEs	32×2	32×2	64×4
	Memory/node	1GB	2GB	8GB
	I/O buses/node	2	2	2
	Model	Dell PowerEdge 1550	HP Server rx2600	AlphaServer ES40
	OS	Red Hat Linux 7.3	Red Hat Linux 7.2	Red Hat Linux 7.1
CPU	Type (speed)	Pentium-III (1GHz)	Itanium-II (1GHz)	Alpha EV68 (833MHz)
I/O bus	Type	64-bit/66MHz PCI	64-bit/133MHz PCI-X	64-bit/33MHz PCI
Network	NIC model	1×QM-400 Elan3	2×QM-400 Elan3	2×QM-400 Elan3
Software	Compiler	Intel C/Fortran v5.0.1	Intel C/Fortran v7.1.17	Compaq's C Compiler

of job launching on Cplant [3], which is to measure the time it takes to launch run a program of size 4 MB, 8 MB, or 12 MB that terminates immediately.

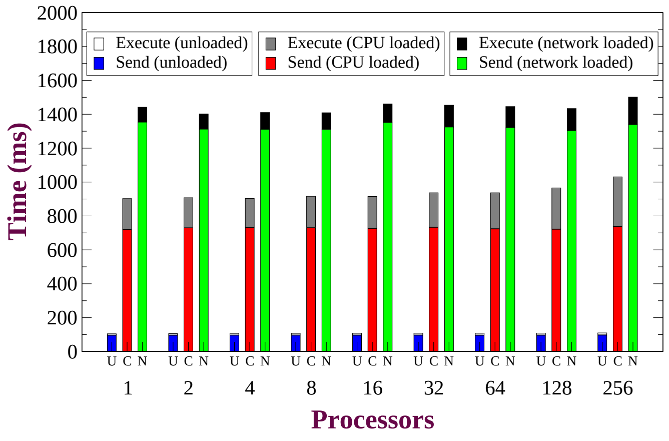
STORM logically divides the job-launching task into two separate operations: the transmission of the binary image, and the actual execution, which includes sending a job-launch command, forking the job, waiting for its termination, and reporting back to the machine manager (MM). For the transmission of the binary images the MM uses XFER-AND-SIGNAL for multicasting chunks and COMPARE-AND-WRITE for flow control. To reduce non-determinism 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 time quantum of 1 ms.

Figure 1(a) shows the time needed to transfer and execute a do-nothing program 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 scalable algorithms and hardware mechanism that are 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 the skew that is accumulated by the processes of the job. The main cause of this skew is the overhead of the operating system. In the largest configuration tested a 12 MB file can be launched in 110 ms, a remarkably low latency.

We have also tested the launch times of the 12 MB file under various load conditions. In one experiment, a do-nothing loop on all the PEs loaded the compute resources of all nodes. On the second load-inducing experiment we stressed the network by pairing all of the processors and continuously sending long messages back and forth between them. Figure 1(b) 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), for the 12 MB file. Note that even in the worst scenario, a network-loaded system, it still takes only 1.5 seconds to launch a 12 MB file on 256 processors.



(a) Send and execute times for several file sizes on an unloaded system (Wolverine)



(b) Send and execute times for a 12 MB file under various types of load: (U)nloaded, (C)ompute-loaded and (N)etwork-loaded

Figure 1. Job Launching Performance

Table 5. A selection of job-launch times (in seconds) found in the literature

Software	Job-launch time / program size	
`rsh`	90	Minimal job on 95 nodes [13]
RMS	5.9	12 MB job on 64 nodes [11]
GLUnix	1.3	Minimal job on 95 nodes [13]
Cplant	20	12 MB job on 1,010 nodes [3]
BProc	2.7	12 MB job on 100 nodes [15]
SLURM	4.9	Minimal job on 950 nodes [17]
STORM	0.11	12 MB job on 64 nodes [11]

Scalability Issues These job launching results are comparable to other systems in the literature for clusters of up to a few hundreds of nodes (see Table 5). Our premise is that one of the main advantages of using hardware mechanisms is that the resource manager can inherit the scalability features of the hardware layer. To verify this property, we presented a detailed model of STORM's job-launching scalability in [11]. Figure 2 shows the predicted launch times of the 12 MB program on clusters of up to 16,384 processors on an Alpha ES40 architecture, similar to that of ASCI Q [26]. In that work we have also extrapolated the expected job-launching performance of the software-based methods found in the literature. Not surprisingly, the hardware-supported mechanisms of STORM provide at least an order of magnitude better performance on very large clusters. In fact, it is the only system that is expected to deliver sub-second performance on thousands of nodes.

4.4. Job Scheduling

STORM supports a variety of job scheduling algorithms including various batch and time-sharing methods. Some of the time sharing methods require a global synchronization message (strobe), which STORM implements using XFER-AND-SIGNAL. we have chosen to focus our evaluation specifically on gang scheduling [8], which is one of the most popular

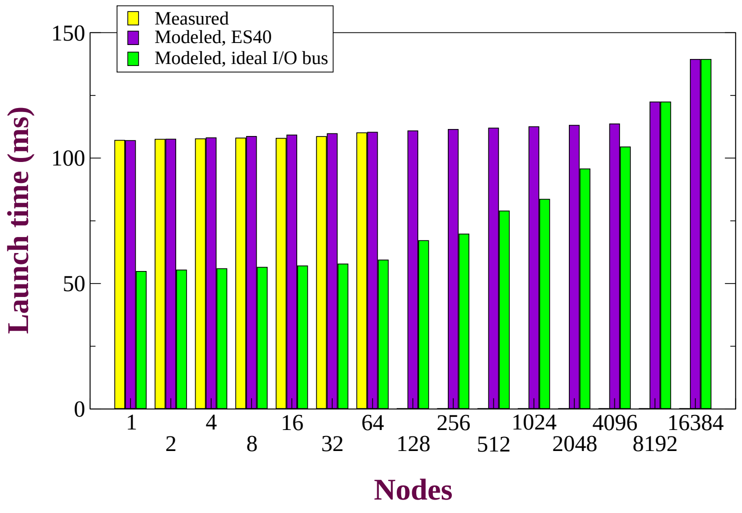


Figure 2. Measured and estimated launch times

coscheduling algorithms. In particular we were interested in the 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). To measure this overhead, we use SWEEP3D and a do-nothing synthetic program, and run two copies of each concurrently, with different timeslice values. Figure 3 shows the average run time of the two jobs for timeslice values from 300 μs to 8 seconds, running on the entire Crescendo cluster. The smallest timeslice value that the scheduler can handle gracefully is $\sim$300 μs, any less than which the node cannot process the incoming strobe messages at the rate they arrive. With a timeslice as short as 2 ms STORM can run multiple concurrent instances of SWEEP3D with virtually no performance degradation over a single instance of the application.[2] This timeslice is an order of magnitude smaller than the local Linux scheduler's quanta, and is significantly smaller than the smallest time quanta that conventional gang schedulers can handle without significant performance penalties [10]. This, together with brisk job launching, allows for workstation-class system responsiveness for interactive jobs on a large parallel system.

4.5. Communication Library

In the following experiments we demonstrate the performance of BCS-MPI. Of interest here is the impact of BCS-MPI's global synchronization of all the nodes in order to schedule communication requests issued by the application processes. We also provide and analyze some results comparing the performance of BCS-MPI to that of Quadrics MPI, a production-quality implementation of MPI.

With BCS-MPI a global strobe is sent to all the nodes (using XFER-AND-SIGNAL) at regular intervals. This tightly couples all the system activities by requiring that they occur at the same time on all nodes. Both computation and communication

[2]This result is also influenced by the poor memory locality of SWEEP3D—the lack of a small memory working set implies minimal extra penalty for a context switch.

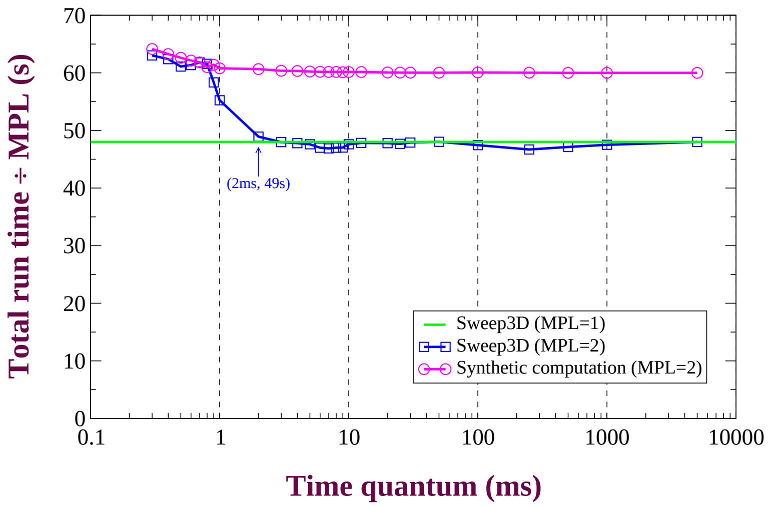
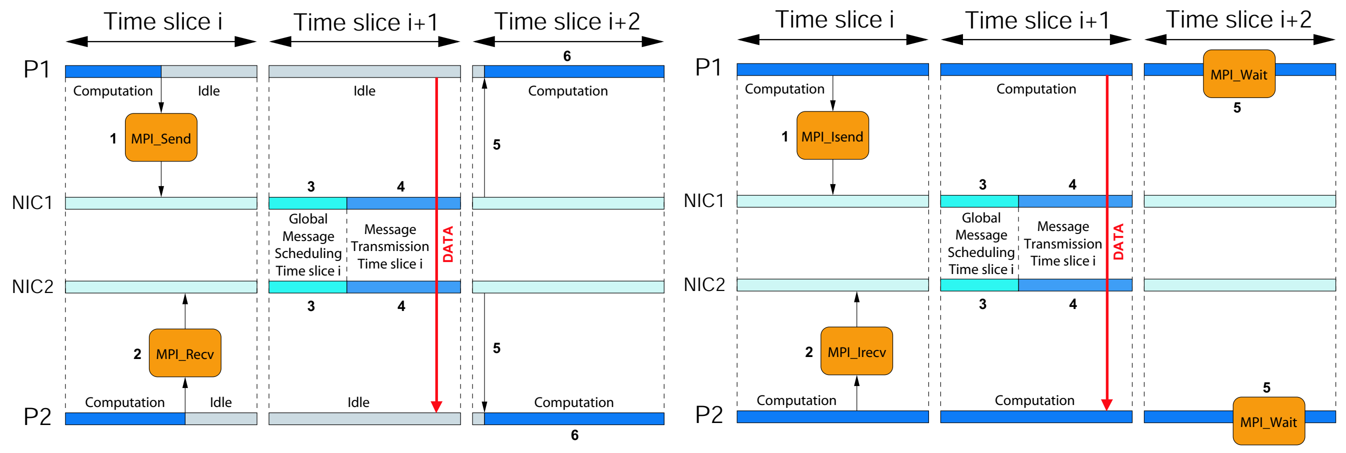


Figure 3. Effect of time quantum with an MPL of 2 on 32 nodes

are scheduled and the communication requests are buffered. At the beginning of every timeslice a partial exchange of communication requirements, implemented with XFER-AND-SIGNAL and TEST-EVENT, provides the information needed for scheduling the communication requests issued during the previous timeslice. After that all of the scheduled communication operations are performed by using XFER-AND-SIGNAL and TEST-EVENT.

The BCS-MPI communication protocol is implemented almost entirely in the network interface card (NIC). By running on the NIC's processor, BCS-MPI is able to overlap the communication with the ongoing computation. The application's processes directly interact (transparently via the BCS-MPI library) with threads running in the NIC. When an application process invokes a communication primitive, it simply posts a descriptor in a region of NIC memory that is accessible to a NIC thread. This descriptor includes all the communication parameters which are needed to complete the operation. The actual communication is performed by a set of cooperating threads running in the NICs involved in the communication protocol (using XFER-AND-SIGNAL). In the QsNet network these threads can directly read/write from/to the application process memory space so that no copies to intermediate buffers are required. Moreover, the posting of the descriptor is a lightweight operation, making the entire latency of the BCS-MPI call even lower than that of the Quadrics MPI.

The communication protocol is divided into micro-phases within every timeslice and its progress is also globally synchronized. To illustrate how BCS-MPI primitives work, two possible scenarios for blocking and non-blocking MPI primitives are described in Figure 4(a) and Figure 4(b), respectively. In Figure 4(a), process P_1 sends a message to process P_2 using MPI_Send and process P_2 receives a message from P1 using MPI_Receive: (1) P_1 posts a send descriptor to the NIC and blocks. (2) P_2 posts a receive descriptor to the NIC and blocks. (3) The transmission of data from P_1 to P_2 is scheduled since both processes are ready (all the pending communication operations posted before timeslice i are scheduled if possible). If the message cannot be transmitted in a single timeslice then it is chunked and scheduled over multiple timeslices. (4) The communication is performed (all the scheduled operations are performed before the end of timeslice $i + 1$). (5) P_1 and P_2



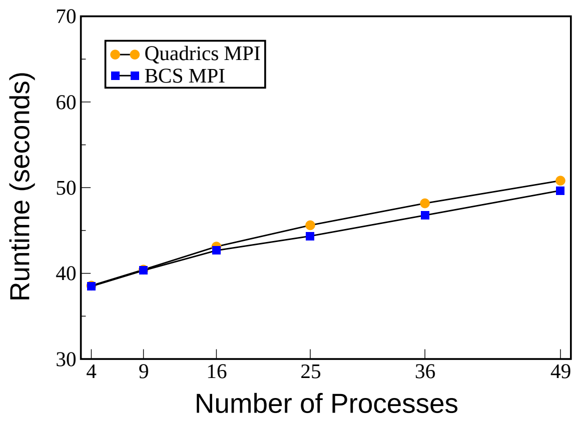
(a) Blocking MPI_Send/MPI_Receive (b) Non-Blocking MPI_Send/MPI_Receive

Figure 4. Blocking and Non-Blocking MPI_Send/MPI_Receive Scenarios in BCS-MPI

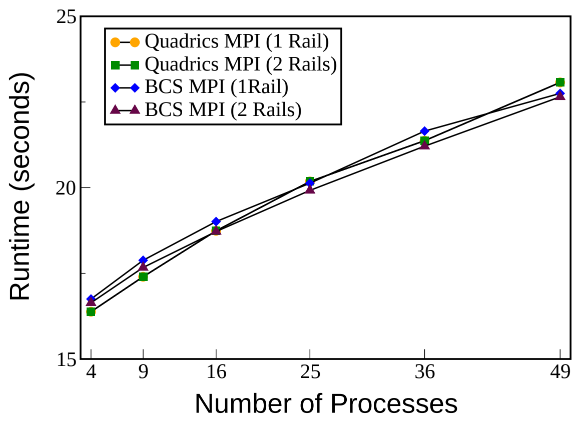
are restarted at the beginning of timeslice i. (6) P_1 and P_2 resume computation. Note that the delay per blocking primitive is 1.5 timeslices on average. However, this penalty can be usually be avoided by using non-blocking communications or by scheduling a different job in timeslice $i+1$. Figure 4(b) shows the same situation for non-blocking MPI primitives. In this case, the communication is completely overlapped with the computation with no performance penalty.

In Figure 5(a) the runtime of SWEEP3D for both BCS-MPI and Quadrics MPI is shown for various numbers of processes on the Crescendo cluster. The effective overlap between computation and communication using BCS-MPI together with the low latency of the BCS-MPI calls allow BCS-MPI to slightly outperform Quadrics MPI, with speedups of up to 2.28%. Figure 5(b) shows the same experiment on the Accelerando cluster. Both BCS-MPI and Quadrics MPI can make use of the second rail available in this cluster. To exploit it, BCS-MPI transmits application point-to-point messages on the second rail while Quadrics MPI statically allocates rails to processes. We observe a small speedup of BCS-MPI over Quadrics MPI, of 1% with one rail and 2% with two rails for the largest configuration.

Scalability Issues To complete the application study and to gain a better understanding of BCS-MPI's scalability, we show SAGE's performance on Crescendo with Quadrics and BCS-MPI. Unlike SWEEP3D, which requires square configurations, SAGE can run on any number of nodes. Figure 6 shows the run time of SAGE on varying numbers of nodes, up to 62 (one node is reserved for the management software). Both versions perform similarly due to the fact that SAGE uses mostly non-blocking point-to-point communication with a relatively large number of neighbors. Most notably, BCS-MPI performs slightly better than Quadrics MPI for the largest configuration, which indicates that the scalability of SAGE is not an issue with BCS-MPI and this cluster size.



(a) Non-Blocking SWEEP3D (Crescendo)



(b) Non-Blocking SWEEP3D (Accelerando)

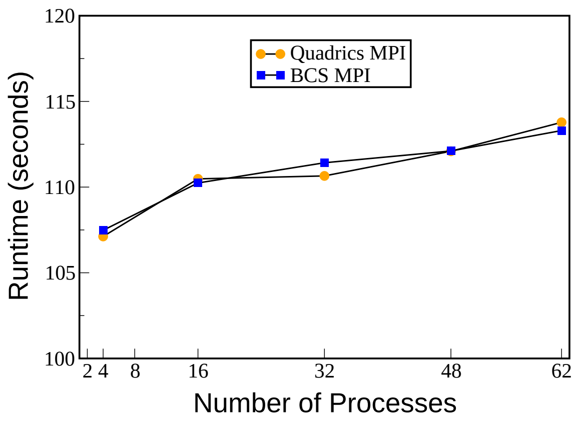


Figure 6. SAGE Performance (Crescendo)

5. Conclusions and Future Work

In this paper we proposed a new abstraction layer for large-scale clusters. This layer, which can be implemented by as few as three communication primitives in the network hardware, can greatly simplify the development of system software for these clusters. In our model the system software is a tightly-coupled parallel application that operates in lockstep on all nodes. If the hardware support for this layer is both scalable and efficient the system software inherits these properties. Such software is not only relatively simple to implement but can also provide parallel programs with most of the services they require to make their development and usage efficient and more manageable. In particular, we discuss how this abstraction layer and the system software can be used for the implementation of efficient, deterministic communication libraries, workstation-class responsiveness, and transparent fault tolerance. We have presented initial experimental results using a prototype system software and advanced interconnection hardware. Our results demonstrate that scalable resource management and application communication are indeed feasible while making the system behave deterministically. Our future work will expand on the use of this determinism to incorporate transparent fault tolerance into the system software. We also plan to explore other possible benefits of a global operating system, such as coordinated parallel I/O and debugging. Lastly, since we envision a simple, global operating system for the cluster, we plan to migrate our code into the Linux kernel. Such an integration should also improve further the performance of the cluster operating system.

References

[1] R. A. Bhoedjang, T. Rühl, and H. E. Bal. Efficient multicast on Myrinet using link-level flow control. In *Proceedings of the* 27^{th} *International Conference on Parallel Processing (ICPP'98)*, pages 381–390, Minneapolis, MN, August 1998.

[2] G. Bosilca, A. Bouteiller, F. Cappello, S. Djailali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Proceedings of IEEE/ACM Supercomputing 2002 (SC'02)*, Baltimore, MD, November 2002.

[3] R. Brightwell and L. A. Fisk. Scalable parallel application launch on Cplant. In *Proceedings of IEEE/ACM Supercomputing 2001 (SC'01)*, Denver, CO, November 10–16, 2001.

[4] D. Buntinas, D. Panda, J. 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.

[5] D. Buntinas, D. Panda, and W. Gropp. NIC-based atomic operations on Myrinet/GM. In *SAN-1 Workshop, High Performance Computer Architecture (HPCA-8) Conference*, Boston, MA, February 2002.

[6] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the* 4^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

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

[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, December 1992.

[9] J. Fernandez, F. Petrini, and E. Frachtenberg. BCS MPI: A New Approach in the System Software Design for Large-Scale Parallel Computers. In *Proceedings of IEEE/ACM Supercomputing 2003 (SC'03)*, Phoenix, AZ, November 2003.

[10] E. Frachtenberg, F. Petrini, S. Coll, and W. chun Feng. Gang scheduling with lightweight user-level communication. In *Proceedings of the* 30^{th} *International Conference on Parallel Processing (ICPP'01), Workshop on Scheduling and Resource Management for Cluster Computing*, Valencia, Spain, September 2001.

[11] E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll. STORM: Lightning-Fast Resource Management. In *Proceedings of IEEE/ACM Supercomputing 2002 (SC'02)*, Baltimore, MD, November 2002.

[12] H. Franke, P. Pattnaik, and L. Rudolph. Gang Scheduling for Highly Efficient Distributed Multiprocessor Syetems. In *Proceedings of the* 6^{th} *Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS '96)*, pages 1–9, Annapolis, MD, October 1996.

[13] 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 25, 1998.

[14] M. Gupta. Challenges in developing scalable scalable software for bluegene/l. In *Scaling to New Heights Workshop*, Pittsburgh, PA, May 2002.

[15] E. Hendriks. BProc: The Beowulf distributed process space. In *Proceedings of the* 16^{th} *Annual ACM International Conference on Supercomputing (ICS '02)*, New York, NY, June 22–26, 2002.

[16] A. Hori, H. Tezuka, and Y. Ishikawa. Overhead analysis of preemptive gang scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 217–230. Springer Verlag, 1998.

[17] M. A. Jette, A. B. Yoo, and M. Grondona. SLURM: Simple linux utility for resource management. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 37–51. Springer-Verlag, 2003.

[18] T. Kamada, S. Matsuoka, and A. Yonezawa. Effi cient parallel global garbage collection on massively parallel computers. In G. M. Johnson, editor, *Proceedings of IEEE/ACM Supercomputing 1994 (SC'94)*, pages 79–88, 1994.

[19] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proceedings of IEEE/ACM Supercomputing 2001 (SC'01)*, Denver, CO, November 2001.

[20] K. Koch. How does ASCI actually complete multi-month 1000-processor milestone simulations? In *Proceedings of the Conference on High Speed Computing*, Gleneden Beach, Oregon, April 22–25, 2002.

[21] K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the fi rst-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.

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

[23] J. Liu, J. Wu, D. K. Panda, and C. Shamir. Designing clusters with Infi niband: Early experience with Mellanox technology. Submitted for publication.

[24] F. Petrini and W. chun 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, January/February 2002.

[26] 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 Supercomputing 2003 (SC'03)*, Phoenix, AZ, November 2003.

[27] R. Riesen, R. Brightwell, L. A. Fisk, T. Hudson, J. Otto, and A. B. Maccabe. Cplant. In *Proceedings of the 1999 USENIX Annual Technical Conference, Second Extreme Linux Workshop*, Monterey, CA, June 6–11, 1999.

[28] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy os-bypass nic-driven gigabit ethernet message passing. In *Proceedings of IEEE/ACM Supercomputing 2001 (SC'01)*, Denver, CO, November 10–16, 2001.

[29] 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, September 1998.

[30] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[31] Thinking Machines Corporation. *NI Systems Programming*, 1992. Version 7.1.

[32] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[33] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the* 19^{th} *International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.