Advance Access published on May 25, 2006
doi:10.1093/comjnl/bxl020

An Abstract Interface for System Software on Large-Scale Clusters

Juan Fernández[1], Eitan Frachtenberg[2]*, Fabrizio Petrini[2] and José-Carlos Sancho[2]

[1]*Computer Engineering Department, University of Murcia, 30071 Murcia, Spain*
[2]*Modeling, Algorithms, and Informatics Group, Computer and Computational Sciences (CCS) Division, Los Alamos National Laboratory, Los Alamos, NM 87545 USA*
**Corresponding author: eitanf@lanl.gov*

Scalable management of distributed resources is one of the major challenges when building large-scale clusters for high-performance computing. This task includes transparent fault tolerance, efficient deployment of resources and support for all the needs of parallel applications: parallel I/O, deterministic behavior and responsiveness. These challenges may seem 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 and demonstrate an abstract network interface 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, can allow commodity clusters to grow to thousands of nodes, while still retaining the usability and performance of the single-node workstation.

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

Received 12 August 2005; revised 20 March 2006

1. INTRODUCTION

Although workstation clusters are a common platform for high-performance computing (HPC), they remain more difficult to manage than sequential systems or symmetric multiprocessors. Furthermore, as cluster sizes increase, the role of the system software—essentially, all of the code that runs on a cluster other than the applications—becomes increasingly more important. For the scope of this paper, we address HPC clusters and their system software. On the hardware side, this typically implies a homogeneous cluster with high-performance networking. On the system side, such clusters often have very specific needs that set them aside from desktop machines or even computational grids. 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 affects not only application performance, but also the cost of ownership of such machines. One of the main goals of this paper is to explore the relationship between these system software requirements and the specialized hardware it runs on.

Interconnection network and system software designers of high-performance computational clusters traditionally rely on a common abstract machine to separate their domains. This abstract machine sees the network as a medium that can move information from one processing node to another, with a given 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, allowing the exploration of multiple hardware designs. The success of this interface implicitly relies on the assumption that any performance improvement in latency and bandwidth can be directly inherited by the system software.

Abstract interfaces evolve over time, as new factors come into play. For example, in the last decade this basic abstract interface has been augmented with distributed shared memory. In such a global address space, a chunk of data is moved from a source to a destination address. This approach was used successfully by communication layers such as Active Messages [1], that emulated a virtual address space on top of physically addressed network interfaces. This successful experience was able to influence 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 nowadays a common feature in networks as Quadrics [2] or Infiniband [3].

In this paper we try to answer the following question. *What hardware features, and consequently 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 crucial for scalable 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 greatest common denominator of the various components of the cluster software and the backbone to integrate a collection of local operating systems (OS) in a single, global middleware.

It is important to stress again here that we are mainly concerned with supercomputer clusters dedicated to high-performance, tightly-coupled applications. Unlike clusters running more distributed applications, these supercomputers are typically characterized by fast, dedicated interconnects (often with collective communication support), and homogeneous computing architectures. For more distributed workloads or less dedicated architectures there are several global OS solutions in existence and in development, including Mosix,[1] Kerrighed,[2] Plurix[3] and Split-OS.[4] In fact, some of these systems support transparent process migration, which is a useful feature for distributed programs, but largely impractical for fine-grained parallel programs.

This paper provides 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 design system software that is hierarchically based on these primitives, called Buffered Coscheduling (BCS) [4]. One of BCS's goals is to simplify the software design by enforcing global coordination of all the activities in a cluster. Through a series of case studies, we show how important parts of the system software can benefit from these primitives. We provide experimental evidence that resource management and job scheduling can be implemented on thousands of nodes and achieve the same level of responsiveness of a dedicated workstation, without any significant increase in complexity. We also describe how a popular communication library, the Message Passing Interface (MPI) can be implemented on top of these global coordination primitives. The proposed implementation is so simple from a design point of view, that it can run almost entirely on the network interface card (NIC) as efficiently as a the production-level 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 proposal to build scalable system software. In Section 4 we describe several case studies, and report several experimental results obtained on our working software prototype on three different clusters. Finally, we conclude and offer venues for future research in Section 5.

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 OS such as Linux to provide a 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 rarely sufficient for running demanding HPC applications in large clusters.

Common middleware solutions include software extensions on top of the workstation operating system, such as the MPI communication library [5] 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. More importantly, some desired features such as multiprogramming, garbage collection or automatic checkpointing are not supported at all, or are very costly in terms of both development costs and performance hits.

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 code. Table 1 reviews the differences in several basic services required to develop, debug and effectively use computing resources. Let us 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 specific middleware. As shown later in Section 4, job launching times can range anywhere from seconds to hours and are typically far from interactive. Many solutions were

[1]www.mosix.org
[2]www.kerrighed.org
[3]www.plurix.de
[4]discolab.rutgers.edu/split-os

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 non-determinism of data's live state [6]

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 [7], GLUnix [8], Cplant [9], BProc [10] and SLURM [11]. 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, with larger clusters (of thousands of nodes), these systems are expected to take many seconds or minutes to launch parallel jobs, due to their reliance on software mechanisms.

Job scheduling. With workstations, it is taken for granted that several applications can run concurrently using time sharing. This concurrency is rarely the norm with clusters. Most middleware used for parallel job scheduling use simple versions of batch scheduling (or gang scheduling at best). This simple scheduling 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 use relatively high time quanta, to hide the high overhead costs associated with context switching in software a parallel job.

To address this problem, the SCore-D [12] scheduler, for example, uses a combination of software and hardware to perform the global context switch more efficiently than with software alone. 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 time quanta when using a cluster size of more than a few hundreds of nodes. In the SHARE gang scheduler of the IBM SP2 [13] network context is switched by the 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 [14]. 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 [15] also had some hardware support for context-switching. However, these specialized machines cost more than, 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. Although these mechanisms may be rudimentary and provide no high-level abstraction, they are adequate for serial and coarse-grained distributed jobs, due to their low synchronization requirements. Unlike these jobs, HPC applications require a more expressive set of communication tools to keep the software development at manageable levels.

The prevailing communication model for modern HPC applications is message passing, where processes use a communication library to send synchronous and asynchronous messages. Of these libraries, the most popular are MPI [5] and PVM [16]. These libraries offer standards that facilitate portability across various cluster and MPP architectures. On the other hand, much effort is required for the optimization and tuning of the libraries to different platforms in order to improve the latency and bandwidth for single messages. Another problem with these libraries is the low-level of the mechanisms they offer that forces the software developer to focus on implementation details, and makes 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.* [17] focuses on latency and bandwidth in asynchronous message passing systems. A higher-level abstraction is the Bulk-Synchronous Parallel (BSP) computing model introduced by Valiant *et al.* [18]. 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. Although BSP is a computing model for parallel systems rather than a programming model, it has the important advantage that modeling the performance of BSP applications becomes significantly simpler, compared with typical asynchronous applications. Compared to the prior PRAM model [19], the BSP model provides a more realistic performance model with respect to time complexity. Algorithm developers can achieve application performance corresponding to their expectations based on the computing model. The BSP computing model has been implemented as several programming libraries to develop parallel applications [20, 21].

Determinism. Serial applications are much easier to debug compared to their parallel counterparts. This is mainly because of their inherent determinism, rendering most bugs easy to reproduce. Parallel programs are sometimes virtually intractable to trace repeatedly: the independent nature of the many components of the systems—nodes, operating systems, processes and network components—add up to an inherently non-deterministic behavior.

Naturally, users can create determinism by writing tightly synchronous parallel applications, where the global state is often known (and can be checkpointed). Even though some parallel programs do indeed follow this model (e.g. using BSP libraries), many programmers prefer an asynchronous model, mainly for performance reasons. Instead of requiring programs to be rewritten, we suggest that synchronization can be enforced at system level, without compromising performance.

Fault tolerance. The same non-determinism also makes checkpoint-based fault tolerance challenging, since the application is rarely in a known steady state where all processes and in-transit messages are synchronized. Fault tolerance on workstations is not currently 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.* [22] introduced a system called MPICH-V 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 complex runtime environment, partly because of 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 used a checkpoint interval of 130 s.

We believe that with some minimal support from the hardware, a relatively simple fault-tolerant system software can be implemented with significantly smaller overhead and shorter checkpoint frequency. To this end, we rely on global synchronization and scheduling of all system activities. We provide points along the execution of a parallel program in which all the allocated resources are in a steady state. Consequently, it is relatively straightforward to implement an algorithm to checkpoint the job safely.

2.1 Toward a global 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 [23]. System software is required to deal with one or more parallel jobs comprising of 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. This non-deterministic behavior makes both system software and user-level applications much harder to debug and maintain. The lack of synchronization also hampers application performance, e.g. when non-synchronized system dæmons introduce computational holes that can severely skew and impact fine-grained applications [24].

To address these challenges, we propose design principles for a global cluster OS that exploits advanced network resources, just like any other HPC application. Our vision is that of a cluster OS that behaves like any SIMD (single-instruction-multiple-data) application, performing resource coordination in lockstep. We argue that performing this task in a scalable way and at sub-millisecond granularity requires hardware support, represented in a small set of network mechanisms. Our goal in this study is to identify and describe these mechanisms. Using an implemented prototype system, 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 proceed to describe how to use these mechanisms to tackle the challenges raised in the preceeding section.

3.1. Core primitives

The proposed architectural support consists of 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 user-level 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 Compare (using $\geq$, $<$, $=$, or $\neq$) 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 statement 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 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 [25]. TEST-EVENT and COMPARE-AND-WRITE are traditional, blocking operations, while XFER-AND-SIGNAL is non-blocking. In the latter case, the local memory block is required to remain unchanged until completion. 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). The events and signals can occur at a user-level library, without involving the kernel.

3.2. Suggested mechanisms

As posited in Section 2, the lack of global coordination between cluster nodes is one of the major deficiencies of cluster system software. On one hand, it can be detrimental to performance. For example, system dæmons that perform resource management tasks can introduce computational ‘holes’ of several hundreds of milliseconds that severely impact fine-grained parallel applications. On the other hand, this lack of coordination restricts the functionality that the cluster system software can provide. For example, many job-scheduling algorithms, such as gang-scheduling, require that the cluster nodes perform context switches concurrently.

To address this problem, our methodology includes a simple yet powerful synchronization mechanism built on top of the core primitives. In this scheme, a *master node* coordinates all the nodes in the cluster (*slave nodes*). To this end, the master node issues a *heartbeat* on a regular basis. Each heartbeat is received by all the slave nodes and constitutes a global synchronization point.

The heartbeat packet optionally incorporates a data payload. Heartbeat data is used by the master node to broadcast instructions/events to the slave nodes. Events describe specific actions to be carried out by the system software dæmons running on the slave nodes. Events just keep in sync system software dæmons across the cluster. This simple scheme turns out to be flexible enough to accommodate the synchronization needs of most system software components.

Under this scheme, all the cluster nodes move from one known, steady state to the next in lockstep. This operation guarantees that the system evolves under control at any time. In addition, each heartbeat can include one or more *microheartbeats*. This feature allows for a variety of granularity levels in order to implement system software tasks with different synchronization requirements.

3.3. Implementation and portability

The three primitives presented above assume that the network hardware provides global, user-level, virtually addressable shared memory and remote direct memory access (RDMA). Multicast, global queries and programmable NICs are also convenient but not required. If either multicast or global queries are not supported, XFER-AND-SIGNAL and COMPARE-AND-WRITE must be emulated through a thin software layer using point-to-point messages. In this case, atomicity is guaranteed if, and only if, the RDMA operation is atomic. To implement XFER-AND-SIGNAL, the source node separately *puts* the block of data to all of the destination nodes. Only if all of the RDMAs complete successfully, does the source node then proceed to signal destination nodes for event triggering in the same way. To implement COMPARE-AND-WRITE, the source node separately *gets* all of the remote variable values. Only if the comparison is true for all of them, does the source node then proceed to write the global variable and trigger the corresponding events using the same algorithm as that of XFER-AND-SIGNAL. To guarantee sequential consistency, a strict ordering must be imposed on both XFER-AND-SIGNAL and COMPARE-AND-WRITE operations. To this end, algorithms and techniques from the DSM world can be used [26]. Finally, it is worth noting that this approach, while feasible, may prove to be prohibitively slow and complex for clusters of substantial size.

These features are present in several state-of-the-art networks like QsNet and Infiniband and their convenience has been extensively studied [2, 27]. Although the physical

TABLE 2. Measured performance of the core primitives.

Network	Multicast (MB/s)	Comparison (μs)
Myrinet	13 (61 nodes) [28]	14.20 (8 nodes) [29]
Infiniband	450 (4 nodes) [27]	Not available
QsNet	140 (1024 nodes) [30]	10 (1024 nodes) [30]
Qsnet^{II}	825 (512 nodes) [31]	3.5 (512 nodes) [31]
BlueGene/L	240 (512 nodes) [32]	8 (512 nodes) [32]

implementation aspects of these primitives are outside the scope of this paper, we note that some or all of them have already been implemented in several other interconnects, as shown in Table 2. Their design was originally meant 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 convenient in order to guarantee scalability for large-scale systems. Software approaches, while feasible for small clusters, do not scale to thousands of nodes. In our case studies, 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 allowed us to implement COMPARE-AND-WRITE. Unlike the other core primitives, TEST-EVENT does not require special hardware support. However, local OS support may improve event handling by allowing processes to block on events.

Table 2 shows the expected performance of the mechanisms that are already implemented in several contemporary interconnect technologies (references point to the source of data). These commercial networks already support at least some of these mechanisms, attesting to the mechanisms' portability. We argue that the scalable implementation of these primitives should become a standard part of every large-scale interconnect, to better serve the system software.

3.3.1. *Hardware support for communication primitives.*

Several networks support these mechanisms, like the QsNet network that we used. One of the notable features incorporated into the Infiniband standard [3] is the hardware-supported multicast operation. This operation sends a single message to a specific *multicast address*. Then, the message is delivered to multiple processes which may be on different end nodes. In Infiniband, hardware multicast is only available under the Unreliable Datagram (UD) transport service. Thus, multicast messages can be dropped or arrive out of order. Liu *et al.* [33] have added a thin software layer on top of Infiniband to provide reliability, in-order delivery and large-message handling.

Before the advent of QsNet and Infiniband, some MPPs provided specialized hardware support for certain communication and synchronization primitives. Unlike cluster interconnects, MPP interconnects are proprietary in most cases. Some of their unique features, such as barrier synchronization and multicast operations, have been inherited by modern cluster interconnects. The following three examples describe MPP technologies that can be used for our suggested primitives, although not all of them are used anymore.

The Cray T3D multiprocessor [34] is a shared-memory system scalable up to 2048 Alpha 21064 processors interconnected via a three-dimensional (3D) torus network. The T3D integrates a barrier network over the entire machine to provide full machine barrier synchronization in less than 2 μs. In addition, two dedicated FETCH&INC registers, and a dedicated message queue are used to perform atomic memory operations and to support distributed-memory applications.

The Cray T3E [35] is the successor to the Cray T3D. The T3E interconnects up to 2048 Alpha 21164 processors through a bidirectional 3D torus. The T3E augments the memory interface of the 21164 with a large set of explicitly managed external registers, called *E-registers*. Direct E-register access operations are used to store operands into E-registers and load results from E-registers. Global E-register operations transfer data to/from global (remote or local) memory, perform atomic memory operations, such as FETCH&INC, FETCH&ADD and COMPARE&SWAP, and support distributed memory applications by defining message queues. The T3E provides a set of 32 barrier/*eureka* synchronization units (BSUs) at each processor. Eurekas allow a set of processors to determine when any one of the processors has signaled some event. Rather than dedicate physical wires for the BSUs, the T3E embeds logical barrier/eureka networks into the 3D torus.

The Connection Machine CM-5 [36] is a distributed-memory system scalable up to 16384 SPARC processors which are interconnected by three networks: the data network, the control network and the diagnostic network. The control network supports four types of broadcast operations: user broadcast, supervisor broadcast, interrupt broadcast and utility broadcast. User and supervisor broadcasts are essentially identical, except that supervisor broadcasts are privileged. Interrupt broadcasts attract the attention of all processors. Utility broadcasts are used to perform system operations. Additionally, the control network supports four different types of combining operations: reduction, forward scan (parallel prefix), backward scan (parallel suffix) and router done. Reduction operations combine values provided by all processors according to a user-supplied operator (OR, XOR, signed MAX, signed ADD, unsigned ADD). Forward scan operations deliver to

TABLE 3. 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

the *i*-th processor the result of applying one of the operators to the values in the preceding $i - 1$ processors. A backward scan provides similar functionality in the reverse direction.

BlueGene/L (BG/L) [37, 38] contains five different networks, with three of them available to parallel programs. A 3D torus is the main communication network for point-to-point messages. The network hardware guarantees reliable, deadlock-free delivery of variable length packets (up to 256 bytes). It also provides simple broadcast functionality by depositing packets along a route. The tree network supports point-to-point messages of fixed length (256 bytes) used to communicate with I/O nodes. BG/L's network also implements broadcasts and reductions. An ALU integrated in the network logic can combine incoming packets using bitwise and integer operations, and forward a resulting packet along the tree. Floating-point reductions can be performed in two phases (exponent plus mantissa). The global interrupt network provides configurable OR wires to perform full-system hardware barriers. The BG/L communication software architecture [38] is hierarchically organized into three layers: the packet layer is a thin, stateless software library that simplifies access to the torus and tree networks; the message layer implements transport of arbitrary-sized messages between compute nodes using the torus network; and MPI is the user-level communication library built on top of the other communication layers.

3.4. System software requirements and solutions

In the desktop environment, most OS decisions only affect a small number of processes and can be executed immediately by a single OS kernel. With a cluster OS on the other hand, most decisions affecting jobs span many nodes and require tight coordination among the independent nodes' system software. To make a global cluster OS as responsive and usable as a desktop OS, several requirements from the middleware and underlying hardware have to be met. The middleware needs to provide primitives that allow for tightly coupled coordination and execution of OS decisions. The underlying hardware is required to provide the mechanisms for the efficient implementation of these primitives. In this paper, we suggest a set of three middleware primitives that we believe can abstract and facilitate desktop-like capabilities for a cluster OS. An important advantage of our suggested layer is that it maps seamlessly and naturally to contemporary high-performance interconnects.

Let us look into each of the following areas where system software requires this support, and explain how the proposed mechanisms can simplify their design and implementation. Table 3 summarizes the primitives' usage described next.

Job launching. The traditional approach to job launching (including the dissemination of executable and data files to cluster nodes) is a simple extension of single-node job launching: data is disseminated using distributed file systems such as NFS, and jobs are launched with scripts or simple utilities such as rsh or mpirun. These methods obviously do not scale well 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 it impractical. For scalable job launching, a smarter mechanism for data dissemination is called for. Several solutions have been proposed for this problem, all of them focusing on software methods to reduce the dissemination time. For example, Cplant and BProc both use their own tree-based algorithm to disseminate data with latencies that are logarithmic in the number of nodes [10, 39]. Although more portable than relying on hardware support, these solutions are significantly slower and can be hard to implement [40]. By breaking the problem down to simple sub-tasks, we may find that scalable and efficient job launching in fact require only little effort:

- Binary and data dissemination are no more than a multicast of packets from a file server to a set of nodes that can be implemented using Xfer-And-Signal. We can use Compare-And-Write for flow control purposes in order to prevent the multicast packets from overrunning the available buffers.
- Actual launching of a job can again 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. The system software on each node would fork the new processes upon receipt of this message, 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 desktop node. This in turn implies that the distributed system should be able to perform preemptive context switching with the same latencies we have come to expect from single nodes, in the order of magnitude 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 for this is shown in the work on the SCore-D software-only gang scheduler. Hori *et al.* [12] 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 ms 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 at the same time might eventually make the latter go unresponded for a while. If this happens even on a single node and even for 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 alleviate these delays. The ability to maintain multiple communication contexts alive in the network securely and reliably, without kernel intervention, is already implemented in several state-of-the-art interconnects. 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 obvious solution to guarantee quality of service between application and synchronization messages is prioritized messages. The current generation of some networks, including QsNet I, 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 used one rail exclusively for system messages, so that they do not have to compete with application-induced traffic on the same network.

Determinism and fault tolerance. Hori *et al.* [12] proposed a mechanism they called network preemption to facilitate tasks such as maintaining a known state of the cluster and context switching. We believe this mechanism is certainly necessary for an efficient solution to this problem, but not sufficient. Even when a single application is running on the system (so there is only one network context, and no preemption), messages can still be enroute 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 may help to identify the steady 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 basic set of communication primitives, such as 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 an experimental version of the MPI library, called BCS-MPI [41], which is sufficiently large to support real applications. As is shown in the next section, these applications have similar performance whether they are used with our version of MPI, or the one at production-level. However, with our MPI, the applications benefit from the increased determinism and overlapping advantages of BCS-MPI.

TABLE 4. Cluster description.

Component	Feature	*Crescendo* cluster	*Accelerando* cluster	*Wolverine* cluster
Node	Number × PEs	32 × 2	32 × 2	64 × 4
	Memory/node	1 GB	2 GB	8 GB
	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 (1 GHz)	Itanium-II (1 GHz)	Alpha EV68 (833 MHz)
I/O bus	Type	64-bit/66 MHz PCI	64-bit/133 MHz PCI-X	64-bit/33 MHz 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

4. CASE STUDIES

To test the thesis that these mechanisms can be exploited by a scalable global OS, we built a prototype resource-management system and tested it on three architectures. We use the Quadrics third generation network as our interconnect, since it already 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 four tasks: job launching, job scheduling, deterministic communication and fault tolerance.[5]

4.1. Software environment

Our prototype resource-management system, called STORM, is composed of a set of dæmons that run on the compute nodes and management node of a cluster [40]. The main dæmon, called the MM, runs on a dedicated management node and communicates with helper node dæmons called NMs, with one instance per node. STORM contains a network abstraction layer that uses the mechanisms described above for executing tasks such as job launching, process coordination (such as 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 complexity of the code is relatively low, weighing around 10,000 lines of C code for the core functionality.

In addition to their importance for resource management, the core primitives can be used to implement just about any communication protocol while still retaining the performance and determinism advantages. We have implemented our experimental version of MPI, BCS-MPI, using these mechanisms [41]. 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 coalesce several communication calls together whenever possible, thus improving the prospect of interleaving communication and computation.

For the following case studies, we used both synthetic and real HPC applications. The applications, SWEEP3D and SAGE are two hydrodynamics codes of interest to the ASC program.

4.2. Hardware environment

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 employ a 128-port Quadrics Elite switch and Quadrics software library ver. 1.5.0-0. Table 4 summarizes the hardware properties of each cluster.

Quadrics network. The Quadrics network [2] is based on two building blocks, a programmable network interface called Elan [44] and a communication switch called Elite [45]. Elite switches can be interconnected in a fat-tree topology. The network has several layers of communication libraries which provide trade-offs between performance and ease of use. Other important features are hardware support for collective communication patterns and fault-tolerance.

The Elan network interface links the high-performance, multi-stage Quadrics network to a processing node containing one or more CPUs. In addition to generating and accepting packets to and from the network, the Elan also provides substantial local processing power to implement communication protocols.

The other building block of the Quadrics network is the Elite switch. The Elite provides the following features: (1) 8 bidirectional links supporting two virtual channels in each direction, (2) an internal full-crossbar switch, (3) a nominal transmission bandwidth of 400 MB/s on each link direction and a flow-through latency of 35 ns, (4) packet error detection

[5]We have studied in detail other properties of STORM's job scheduling and job launching abilities, and model their scalability [40].

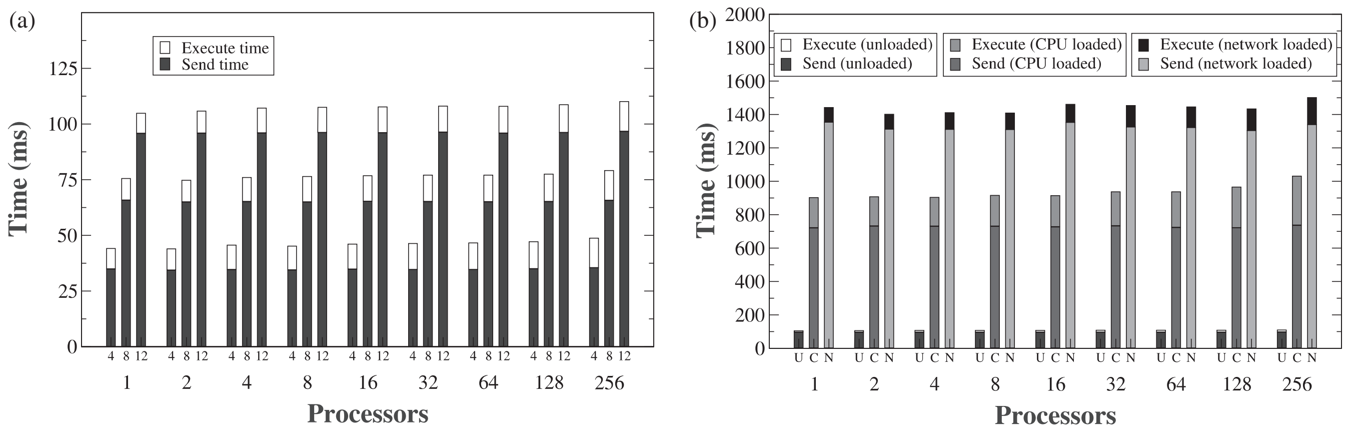

FIGURE 1. Job launching performance. (a) Send and execute times for several file sizes on an unloaded system (Wolverline). (b) Send and execute times for a 12 MB file under various types of load: (U)nloaded (C)ompute-loaded and (N)etwork-loaded.

and recovery, with routing and data transactions CRC protected, (5) two priority levels combined with an aging mechanism to ensure a fair delivery of packets in the same priority level, (6) hardware support for broadcasts, (7) and adaptive routing.

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.* [39] in their study of job launching on Cplant, which is to measure the time it takes to 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 transferal of the binary image and the actual execution, which include sending a job-launch command, forking the job, waiting for its termination, and reporting back to the MM. For the transferal of the files, the MM uses XFER-AND-SIGNAL for multicasting chunks and COMPARE-AND-WRITE for flow control. In order to reduce non-determinism, the MM can issue commands and receive the notification of events only at the beginning of a time slice. Therefore, both the binary transfer and the actual execution will take at least one time slice. 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 1a 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, which is a result of the scalable algorithms and hardware mechanism that are used in 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 the skew that is accumulated by the processes of the job, caused by 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. This latency can be broken down as follows. The average transfer time is 96 ms, and the average execution overhead is 14 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 (limited by the PCI I/O bus in the evaluated architecture), and the actual value of 131 MB/s is caused by unresponsiveness and serialization within the lightweight host process which services TLB misses and performs file accesses on behalf of the NIC. The protocol bandwidth thus reaches 125 MB/s per node, with an aggregate bandwidth of 7.875 GB/s on 63 nodes.

We have also tested the launch times of the 12 MB file under various load conditions. In one experiment, a no-op loop on all the PEs loaded the processors of all nodes. On the second load-inducing experiment we stressed the network by pairing all the processors and continuously sending long ping-pong messages between them. Figure 1b 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, with a network-loaded system, it still takes only 1.5 s to launch a 12 MB file on 256 processors. Still, the true advantage of this approach is revealed when growing to a very large number of processors, as discussed below.

Scalability issues. These job launching results are comparable to other systems in the literature for clusters of up to

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

Software	Hardware	Job-launch time/program size	
RMS	Pentium-III (1 GHz)/QsNet	5.9	12 MB job on 64 nodes [40]
rsh	UltraSPARC (167 MHz)/Myrinet	90	Minimal job on 95 nodes [8]
GLUnix	UltraSPARC (167 MHz)/Myrinet	1.3	Minimal job on 95 nodes [8]
Cplant	Alpha EV6 (466 MHz)/Myrinet	20	12 MB job on 1,010 nodes [39]
BProc	Alpha EV6 (466 MHz)/Myrinet	2.7	12 MB job on 100 nodes [10]
SLURM	*Not available*	4.9	Minimal job on 950 nodes [11]
STORM	Alpha EV68 (833 MHz)/QsNet	0.11	12 MB job on 64 nodes [40]

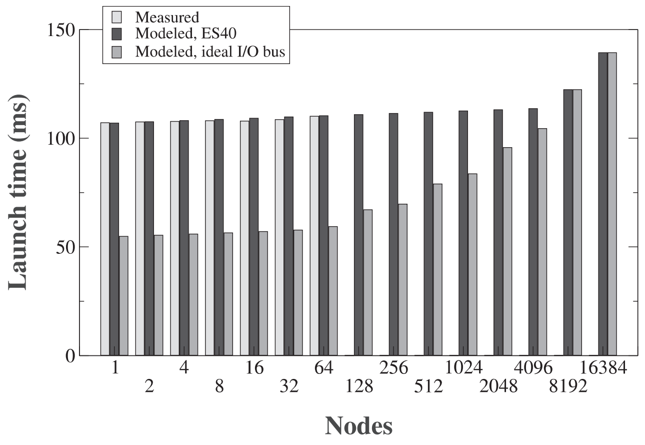

FIGURE 2. Measured and estimated launch times.

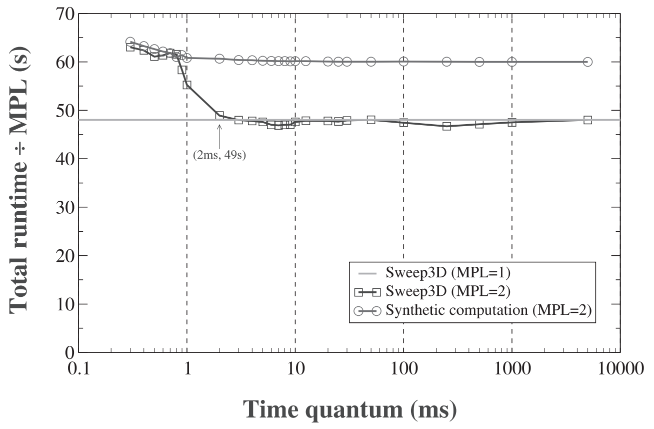

FIGURE 3. Effect of time quantum with a multiprogramming level of 2 on 32 nodes.

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 constructed a detailed model of STORM's job-launching scalability [40]. 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 ASC Q [24]. 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 several 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 chose to focus our evaluation specifically on gang scheduling [46], which is one of the most popular co-scheduling algorithms. In particular we were interested in the effect of the time slice on overhead. Smaller time slices yield better response time at the cost of decreased throughput (because of 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 time slice values. Figure 3 shows the average run time of the two jobs for time slice-values from 300 μs to 8 s, running on the entire Crescendo cluster. The smallest time-slice value that the scheduler can handle gracefully is ∼300 μs, under which the node cannot process the incoming strobe messages at their arrival rate. With a time slice as short as 2 ms, STORM can run multiple concurrent instances of SWEEP3D with negligible performance degradation compared to no time sharing.[6] This time slice is an order of magnitude smaller than the underlying Linux 2.4 scheduler's quanta, and significantly better than the smallest time quanta that conventional gang schedulers can handle with no performance penalties [7]. This low quantum, together with brisk job launching, allows for workstation-class system responsiveness and usage of the parallel system for interactive jobs.

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

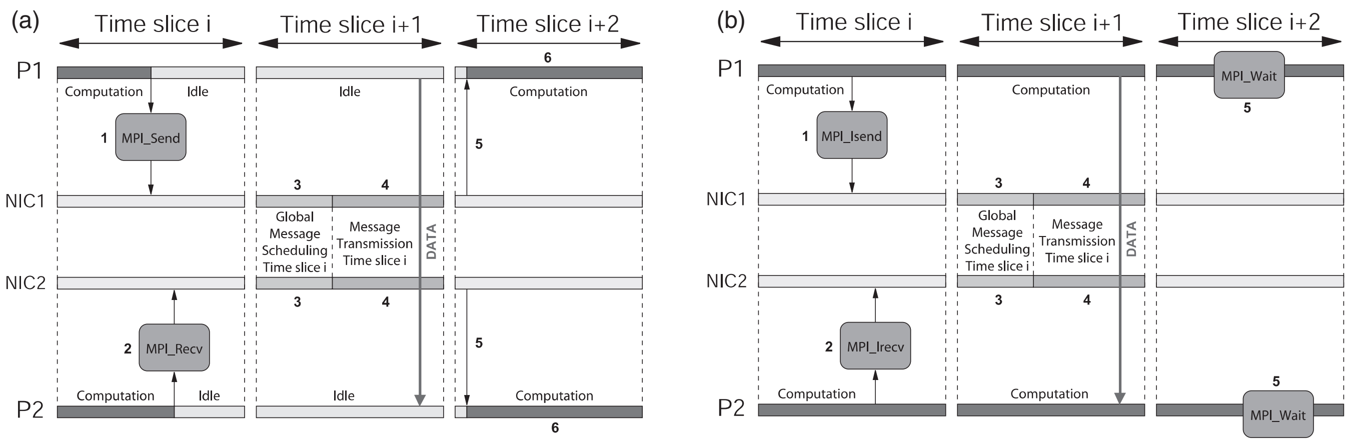

FIGURE 4. Blocking and non-blocking MPL_Send/MPI_Receive seenarious in BCS-MPI. (a) Blocking MPI Send/MPI Receive. (b) Non-blocking MPI Send/MPI Receive.

Note that SWEEP3D tends to have fine-grained communication (in fact, the finest grain of the applications we evaluated for this paper). For this experiment, the communication granularity was in the order of few milliseconds (depending on the phase of the computation). Since fine-grained applications are most adversely affected by a very short time quantum (because of the frequent synchronization interruptions), other applications typically fare even better at these time quanta range. In complementing experiments we performed with extremely fine-grained synthetic benchmarks, a slowdown of 50% or more can be produced, but those scenarios do not use realistic workloads.

4.5. Communication library

In the next experiments, we describe the performance of BCS-MPI, a novel implementation of MPI which globally synchronizes 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-level implementation of MPI.

With BCS-MPI, a global strobe is sent to all the nodes (using XFER-AND-SIGNAL) at regular intervals. Thus, all the system activities are tightly coupled because they occur at the same time on all nodes. Both computation and communication are scheduled and the communication requests are buffered. At the beginning of every time slice 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 time slice. After that, all 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 applications processes directly interact 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 production-level MPI.

The communication protocol is divided into micro-phases within every time slice, which are also globally synchronized. To illustrate how BCS-MPI primitives work, two possible scenarios for blocking and non-blocking MPI primitives are described in Figure 4a and 4b respectively. In Figure 4a, process P_1 sends a message to process P_2 using MPI_Send and process P_2 receives a message from P_1 using MPI_Receive, thus: (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 time slice i are scheduled if possible). If the message cannot be transmitted in a single time slice, then it is chunked and scheduled over multiple time slices. (4) The communication is performed (all the scheduled operations are performed before the end of time slice $i + 1$). (5) P_1 and P_2 are restarted at the beginning of time slice $i + 2$. (6) P_1 and P_2 resume computation. Note that the delay per blocking primitive is 1.5 time slices on average. However, this penalty can be mostly avoided by

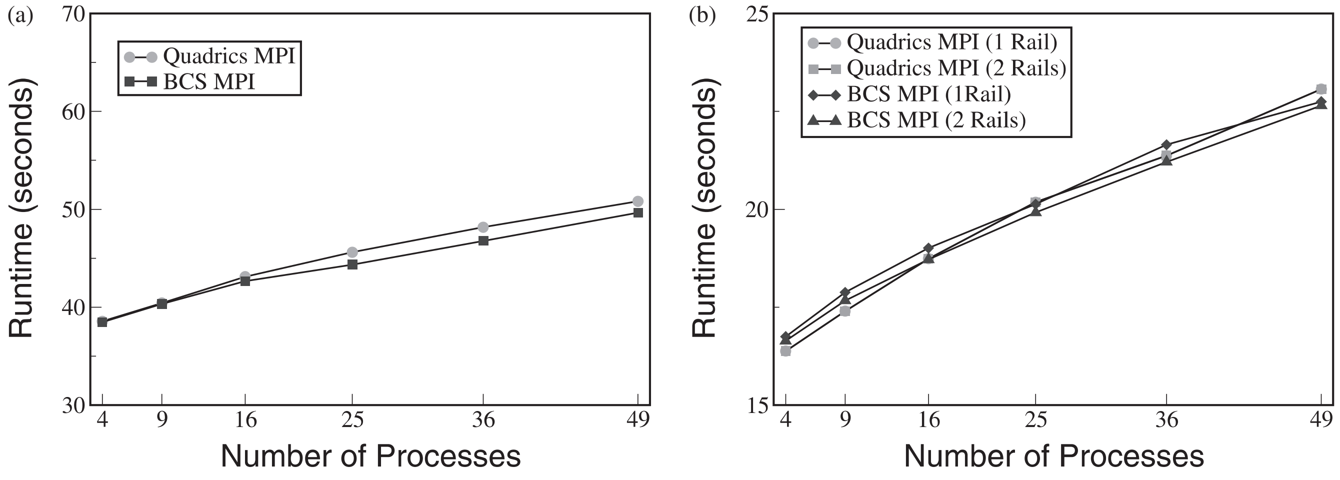

FIGURE 5. SWEEP3D performance. (a) Blocking SWEEP3D (Crescendo). (b) Non-blocking SWEEP3D (Accelerando).

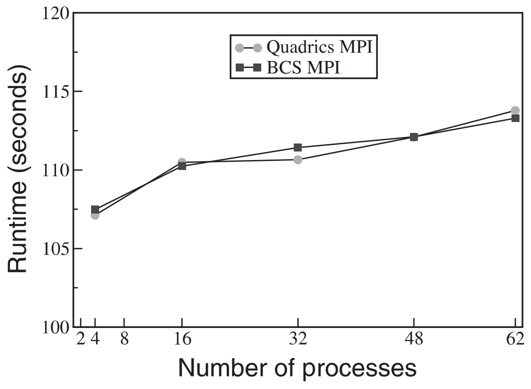

FIGURE 6. SAGE performance (Crescendo).

using non-blocking communications or by scheduling a different job in time slice $i + 1$. Figure 4b 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 5a the runtime of SWEEP3D for both BCS-MPI and Quadrics MPI with varying numbers of processes on the Crescendo cluster is shown. The effective overlap between computation and communication in BCS-MPI along 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 5b 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.

Figure 6 shows SAGE's performance on Crescendo with Quadrics and BCS-MPI. Unlike SWEEP3D, which requires square configurations, SAGE can run on any number of nodes. The figure shows the runtime of SAGE on varying number 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.

4.6. Fault tolerance

We are currently in the process of implementing and evaluating a transparent fault-tolerance layer for cluster operating systems. As part of this effort, we have evaluated the feasibility and overhead of transparent checkpointing of parallel applications with contemporary technology [47]. We found that for most scientific applications, modern interconnects and storage technology provide adequate bandwidth if implemented efficiently. To evaluate the complexity and performance of such an implementation, members of our team have developed a prototype system called TICK (Transparent Incremental Checkpointing at Kernel-level) [48]. In TICK, processes are checkpointed at externally imposed intervals by the kernel, in accordance with our view of a globally coordinated and synchronized OS.

Figure 7 shows the performance hit (or overhead) of Sage and SWEEP3D under the TICK system. Three problem sizes (and memory footprints) are used for each application, as well as several time slice interval values between checkpoints. Figure 7a shows the overhead when checkpointing to main memory (e.g. to protect against software faults),

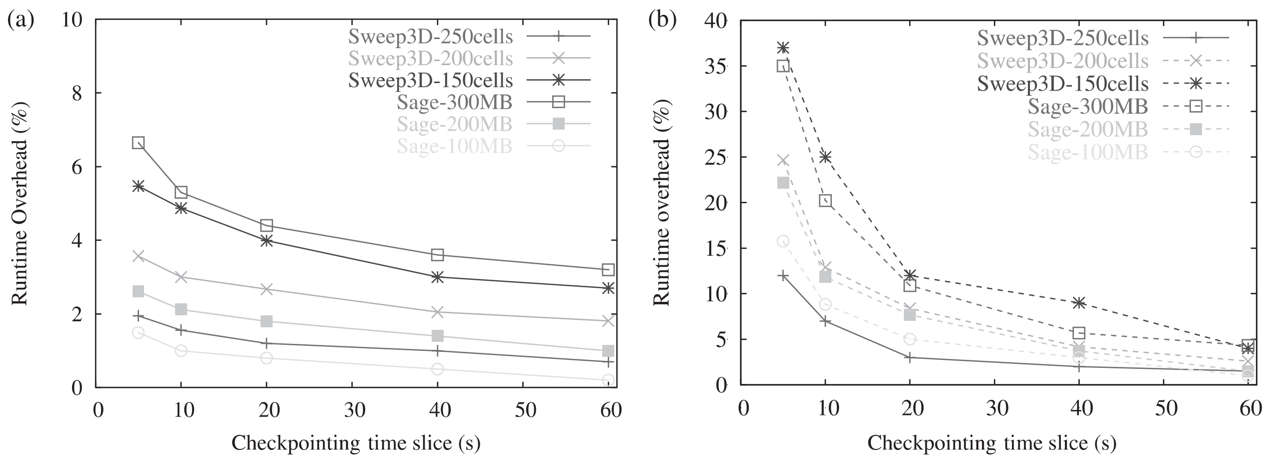

FIGURE 7. Checkpoint performance. (a) Sage and SWEEP3D (memory). (b) Sage and SWEEP3D (disk).

and Figure 7b shows the same for hard-disk checkpointing (protecting against hardware faults). Note that work on parallelizing TICK is still underway, so the applications were run in sequential mode. Also note that TICK can perform both full checkpoints and incremental ones (only saving the pages that were changed since the last time slice).

We chose to show the full-checkpoint overhead performance here as an upper bound for performance degradation. More detailed comparisons and results can be found in the original paper [48]. Note that even at the worst case (disk checkpoint), if we are willing to risk the loss of as little as one minute of computation, the checkpointing adds to the runtime of all programs 5% or less to their checkpoint-free runtime. Combined with our previous results on the network bandwidth requirements for remote checkpointing, we conclude that synchronized transparent checkpointing is not only feasible, but also imposes little overhead on the application. The selectable level of determinism in our cluster OS model is the missing link to complete the checkpointing: it guarantees that that the system can be in a known state at the time slice, allowing the checkpointed state to be consistent across nodes, with no messages in flight.

5. CONCLUSIONS AND FUTURE WORK

In this paper we propose 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 immensely simplify the development of system software for large 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 more efficient and 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 upon this determinism to incorporate transparent fault tolerance into the system software for parallel programs as well. We also plan to explore other possible benefits of a global operating system, such as coordinated parallel I/O and debugging. Additionally, we plan to port our prototypes to more cluster architectures to prove their generality.

ACKNOWLEDGEMENTS

This work is supported by the http://www.energy.gov U.S. Department of Energy through http://www.lanl.gov Los Alamos National Laboratory contract W-7405-ENG-36.

REFERENCES

[1] von Eicken, T., Culler, D. E., Goldstein, S. C. and Schauser, K.E. (1992) Active messages: a mechanism for integrated communication and computation. In *Proc. 19th Int. Symp. on Computer Architecture*, Gold Coast Australia, May 19–21, pp. 256–266. ACM Press, New York, NY.

[2] Petrini, F., chun Feng, W., Hoisie, A., Coll, S. and Frachtenberg, E. (2002) The quadrics network: high-performance clustering technology. *IEEE Micro*, **22**, 46–57.

[3] Infiniband (2004) *www.in.nibandta.org*. Infiniband Trade Association.

[4] Petrini, F. and chun Feng, W. (2001) Improved resource utilization with buffered coscheduling. *J. Parall. Algorithms Appl.*, **16**, 123–144.

[5] Snir, M., Otto, S., Huss-Lederman, S., Walker, D. and Dongarra, J. (1998) *MPI: The Complete Reference* (2nd edn.). The MIT Press, Cambridge, MA.

[6] Kamada, T., Matsuoka, S. and Yonezawa, A. Efficient parallel global garbage collection on massively parallel computers. In *Proc. 1994 IEEE/ACM Conf. on Supercomputing*, Washington, D.C., November 14–18, pp. 79–88. ACM Press, New York, NY.

[7] Frachtenberg, E., Petrini, F., Coll, S. and chun Feng, W. (2001) Gang scheduling with lightweight user-level communication. In *Proc. 30th Int. Conf. on Parallel Processing, Workshop on Scheduling and Resource Management for Cluster Computing*, Valencia, Spain, September 3–7. IEEE Computer Society, Los Alamitos, CA.

[8] Ghormley, D. P., Petrou, D., Rodrigues, S. H., Vahdat, A. M. and Anderson, T. E. (1998) GLUnix: a GLobal Layer Unix for a network of workstations. *Softw. Pract. Exp.*, **28**, 929–961.

[9] Riesen, R., Brightwell, R., Fisk, L. A., Hudson, T., Otto, J. and Maccabe, A. B. (1999) Cplant. login: USENIX Magazine. In *Proc. 1999 USENIX Annual Technical Conf., Second Extreme Linux Workshop*, 24.

[10] Hendriks, E. (2002) BProc: the Beowulf distributed process space. In *Proc. 16th ACM Int. Conf. on Supercomputing*, New York, NY, June 22–26, pp. 129–136. ACM Press, New York, NY.

[11] Jette, M., Yoo, A. B. and Grondona, M. (2003) SLURM: simple utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, Seattle, WA, June 24, pp. 37–51. Springer-Verlag, Berlin.

[12] Hori, A., Tezuka, H. and Ishikawa, Y. (1998) Overhead analysis of preemptive gang scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, Orlando, FL, March, 24, pp. 217–230. Springer-Verlag, Berlin.

[13] Franke, H., Pattnaik, P. and Rudolph, L. (1996) Gang scheduling for highly efficient distributed multiprocessor systems. In *Proc. 6th Symp. on The Frontiers of Massively Parallel Computation*, Annapolis, MD, October 27–31, pp. 4–12. IEEE Computer Society, Los Alamitos, CA.

[14] Thinking Machines Corporation (1992), *NI System Programming*. Version 7.1.

[15] Feitelson, D. G. and Rudolph, L. (1992) Gang scheduling performance benefits for fine-grain synchronization. *J. Parall. Distrib. Comput.*, **16**, 306–318.

[16] Sunderam, V. S. (1990) PVM: a framework for parallel distributed computing. *Concur. Pract. Exp.*, **2**, 315–339.

[17] Culler, D. E., Karp, R. M., Patterson, D. A., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R. and von Eicken, T. (1993) LogP: towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, July 19–22, pp. 1–12. ACM Press, New York, NY.

[18] Valiant, L. G. (1990) A bridging model for parallel computation. *Commun. ACM*, **33**, 103–111.

[19] Fortune, S. and Wyllie, J. (1978) Parallelism in random access machines. In *Proc. 10th ACM Symp. on Theory of Computing*, San Diego, CA, May 1–3. ACM Press, New York, NY.

[20] Hill, J. M. D., McColl, B., Stefanescu, D. C., Goudreau, M. W., Lang, K., Rao, S. B., Suel, T., Tsantilas, T. and Bissel, R. H. (1998) BSPlib: the BSP programming library. *Parall. Comput.*, **24**, 1947–1980.

[21] Kee, Y. and Ha, S. (2002) An efficient implementation of the BSP programming library for VIA. *Parall. Process. Lett.*, **12**, 65–77.

[22] Bosilca, G. *et al.* (2002) MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In *Proc. 2002 IEEE/ACM Conf. on Supercomputing*, Baltimore, MD, April 22–25. IEEE Computer Society, Los Alamitos, CA.

[23] Koch, K. (2002) How does ASCI actually complete multi-month 1000-processor milestone simulations? In *Proc. Conf. on High Speed Computing*, Gleneden Beach, OR, April 22–25. LANL/LLNL/SNL.

[24] Petrini, F., Kerbyson, D. and Pakin, S. (2003) The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the 2003 IEEE/ACM Conf. on Supercomputing*, Phoenix, AZ, November 15–21. IEEE Computer Society, Los Alamitos, CA.

[25] Lamport, L. (1979) How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, **C-28**, 690–691.

[26] Culler, D. E., Singh, J. P. and Gupta, A. (1998) *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kauffman, San Francisco, CA.

[27] Liu, J., Mamidala, A. R., Vishnu, A. and Panda, D. K. (2005) Evaluating infiniBand performance with PCI express. *IEEE Micro*, **25**, 20–29.

[28] Bhoedjang, R. A., Rühl, T. and Bal, H. E. (1998) Efficient multicast on Myrinet using link-level How control. In *Proc. 27th Int. Conf. on Parallel Processing, Workshop on Scheduling and Resource Management for Cluster Computing*, Minneapolis, MN, August 10–14, pp. 381–390. IEEE Computer Society, Los Alamitos, CA.

[29] Yu, W., Buntinas, D., Graham, R. L. and Panda, D. K. (2004) Efficient and scalable barrier over Quadrics and Myrinet with a New NIC-based collective message passing protocol. In *Proc. Int. Parallel and Distributed Processing Symp.*, Santa Fe, NM (USA), April 26–30. IEEE Computer Society, Los Alamitos, CA.

[30] Petrini, F., Fernández, J., Frachtenberg, E. and Coll, S. (2003) Scalable collective communication on the ASCI Q machine. In *Proc. Symp. on High Performance Interconnects*, Stanford, CA (USA), August 20–22, pp. 54–59. IEEE Computer Society, Los Alamitos, CA.

[31] Quadrics (2004) *www.quadrics.com.* Quadrics Supercomputers World Ltd.

[32] Davis, K., Hoisie, A., Johnson, G., Kerbyson, D. J., Lang, M., Pakin, S. and Petrini, F. (2004) A performance and scalability analysis of the BlueGene/L architecture. In *Proc. 2004 IEEE/ACM Conf. on SuperComputing*, Pittsburgh, TA (USA), November 6–12. IEEE Computer Society, Los Alamitos, CA.

[33] Liu, J., Mamidala, A. R. and Panda, D. K. (2004) Fast and scalable MPI-level broadcast using InfiniBand's hardware multicast support. In *Proc. Int. Parallel and Distributed Processing Symp.*, Santa Fe, NM (USA), April 26–30. IEEE Computer Society, Los Alamitos, CA.

[34] Cray Research, Inc. (1993), *Cray T3D. System Architecture Overview*.

[35] Scott, S. L. (1996) Synchronization and communication in the T3E multiprocessor. In *Proc. 7th Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA (USA), October 1–5, pp. 26–36. ACM Press, New York.

[36] Leiserson, C. E. *et al*. (1996) The network architecture of the connection machine CM-5. *J. Parall. Distrib. Comput.*, **33**, 145–158.

[37] Adiga, N. *et al*. (2002) An overview of the BlueGene/L supercomputer. In *Proc. 2002 IEEE/ACM Conf. on Super Computing*, Baltimore, MD (USA), November 16–22. IEEE Computer Society, Los Alamitos, CA.

[38] Almási, G. *et al*. (2003) An overview of the BlueGene/L system software organization. In *Proc. Euro-Par*, Klagenfurt, Austria, August 26–29, pp. 346–353. Springer-Verlag, Berlin.

[39] Brightwell, R. and Fisk, L. A. (2001) Scalable parallel application launch on Cplant. In *Proc. 2001 IEEE/ACM Conf. on Supercomputing*, Denver, CO, November 10–16. IEEE Computer Society, Los Alamitos, CA.

[40] Frachtenberg, E., Petrini, F., Fernández, J., Pakin, S. and Coll, S. (2002) STORM: lightning-fast resource management. In *Proc. 2002 IEEE/ACM Conf. on Supercomputing*, Baltimore, MD, November 16–22. IEEE Computer Society, Los Alamitos, CA.

[41] Fernández, J., Petrini, F. and Frachtenberg, E. (2003) BCS MPI: a new approach in the system software design for large-scale parallel computers. In *Proc. 2003 IEEE/ACM Conf. on Supercomputing*, Phoenix, AZ, November 15–21. IEEE Computer Society, Los Alamitos, CA.

[42] Kerbyson, D., Alme, H., Hoisie, A., Petrini, F., Wasserman, H. and Gittings, M. (2001) Predictive performance and scalability modeling of a large-scale application. In *Proc. 2001 IEEE/ACM Conf. on Supercomputing*, Denver, CO, November 10–16. IEEE Computer Society, Los Alamitos, CA.

[43] Koch, K. R., Baker, R. S. and Alcouffe, R. E. (1992) Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor. *Tran. Amer. Nucl. Soc.*, **65**, 198–199.

[44] Quadrics Supercomputers World Ltd. (1999), *Elan Reference Manual*.

[45] Quadrics Supercomputers World Ltd. (1999), *Elite Reference Manual*.

[46] Feitelson, D. G. and Rudolph, L. (1992) Gang scheduling performance benefits for fine-grain synchronization. *J. Parall. Distrib. Comput.*, **16**, 306–318.

[47] Sancho, J.-C., Petrini, F., Johnson, G., Fernńdez, J. and Frachtenberg, E. (2004) On the feasibility of incremental checkpointing for scientific computing. In *Proc. Int. Parallel and Distributed Processing Symp*, Santa Fe, NM, April 26–30. IEEE Computer Society, Los Alamitos, CA.

[48] Gioiosa, R., Sancho, J.-C., Jiang, S., Petrini, F. and Davis, K. (2005) Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proc. 2005 IEEE/ACM Conf. on SuperComputing*, Seattle, WA, November 12–18. IEEE Computer Society, Los Alamitos, CA.