BCS-MPI: A New Approach in the System Software Design for Large-Scale Parallel Computers*

Juan Fernández[1,2] Eitan Frachtenberg[1] Fabrizio Petrini[1]

[1]Performance and Architecture Laboratory (PAL)
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory, NM 87545, USA
{juanf,eitanf,fabrizio}@lanl.gov

[2]Dpto. Ingeniería y Tecnología de Computadores
Universidad de Murcia, 30071 Murcia (SPAIN)

Abstract

Buffered CoScheduled MPI (BCS-MPI) introduces a new approach to design the communication layer for large-scale parallel machines. The emphasis of BCS-MPI is on the global coordination of a large number of communicating processes rather than on the traditional optimization of the point-to-point performance. BCS-MPI delays the inter-processor communication in order to schedule globally the communication pattern and it is designed on top of a minimal set of collective communication primitives. In this paper we describe a prototype implementation of BCS-MPI and its communication protocols. Several experimental results, executed on a set of scientific applications, show that BCS-MPI can compete with a production-level MPI implementation, but is much simpler to implement, debug and model.

Keywords: MPI, buffered coscheduling, STORM, Quadrics, system software, communication protocols, cluster computing, large-scale parallel computers.

1 Introduction

One of the oft-ignored and yet vitally important aspects of large-scale parallel computers is system software. This software, which consists essentially of everything that runs on the computer other than user applications, is required to make the hardware usable and responsive. However, experience in the development of large-scale machines, and in particular of the ASCI[1] ones, shows that it can take several years to design, implement, debug and optimize the entire software stack before these machines become reliable and efficient production-level systems [14].

The complexity of these machines has risen to a level that is comparable to that of the scientific simulations for which they are used. Such high-performance-computing (HPC) applications routinely use thousands of processes and each process can have a large memory image and multiple outstanding messages, resulting in a very large and complicated global state.

System software consists of various aspects, including communication libraries, resource management (the software infrastructure in charge of resource allocation and accounting), services for parallel file system, and fault tolerance. The current state of the art is to design these components separately, in order to have a modular design and allow different developers to work concurrently while limiting cross-dependencies. Many of these components have many elements in common, such as communication mechanisms, that are implemented and re-implemented several times. In some cases the lack of a single source of system services is also detrimental to performance: most parallel systems cannot guarantee quality of service (QoS) for user-level traffic and system-level traffic in the same interconnection network. Low-priority, best-effort traffic generated by the parallel

[1]http://www.lanl.gov/projects/asci/

file system can interfere with higher-priority, latency-sensitive traffic generate at user level. As another example, system dæmons that perform resource management can introduce computational "holes" of several hundreds of ms that can severely impact fine-grained scientific simulations, since they are not coordinated with the communication library's activities [20].

Buffered Coscheduling (BCS) [18] is a new design methodology for the system software that attempts to tackle both problems: the complexity of a large-scale parallel machine and the redundancy of its software components. The vision behind BCS is that both size and complexity of the system software can be substantially reduced by using a common and coordinated view of the system. BCS tries to globally organize all the activities of such machines at a fine granularity, (in the order of a few hundreds of μs). In a sense, BCS represents an effort to implement a SIMD global operating system (OS) at a granularity coarser than the single instruction, and yet fine enough so as not to harm application performance. Both computation and communication are globally scheduled at regular intervals, and the scheduling decisions are taken after exchanging all the required information. The separate operating systems of each node are coalesced into a single system view, without incurring any significant performance penalty.

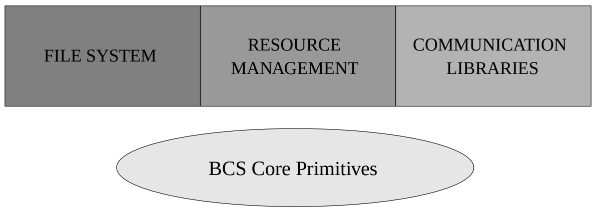


Figure 1: Hierarchical organization of the system software. Parallel file system, resource management, and communication libraries rely on the BCS core primitives for their interactions.

A major innovative aspect of BCS is that most, if not all, of the various elements of the system software can be implemented on top of a small set of primitives (Figure 1). We call this set of only three functions the BCS core primitives [8]. We argue that the BCS core primitives are on the one hand general enough to cover many of the requirements of system software, and yet on the other hand, close enough to the hardware layer to exploit the highest level of performance. In [8] we demonstrated that it is possible to implement a scalable resource management system, called STORM, that is orders of magnitude faster than existing production-level software, by using the BCS core primitives.

In this paper we extend and generalize our research to another aspect of system software, the communication library. We have chosen to implement a variant of the popular MPI library, called BCS-MPI. BCS-MPI is designed following the BCS methodology. It is hierarchically built on top of the BCS core primitives and its scheduling decisions are globally coordinated.

The main research trend in the design of communication libraries over the past decade has been to minimize the point-to-point latency by removing kernel overhead and moving the data communication into the user level [2, 7, 9, 12, 17, 21, 26, 29]. BCS-MPI follows a different path, which may seem counterintuitive at first sight. Rather than optimizing the single point-to-point communication in isolation, it tries to optimize the entire communication pattern. Communication is scheduled globally by dividing time into short slices, and using a distributed algorithm to schedule the point-to-point communication that will occur at each time slice. Communication is scheduled only at the beginning of a time slice and performed at kernel level [6]. The shortest latency that a message will experience will be at least one time slice, which is in the order of few hundreds of μs with current technology. On the other hand, we gain total ordering and determinism of the communication behavior, which can have significant benefits. For example, the fact that the communication state of all processes is known at the beginning of every time slice facilitates the implementation of checkpointing and debugging mechanisms.

The primary contribution of this paper is in demonstrating that a constrained communication library such as BCS-MPI provides approximately the same performance of a production-level version of MPI on a large set of scientific applications, but with a much simpler software design. In fact, BCS-MPI is so small that it runs almost entirely on the network interface processor, and its activity is completely overlapped with the computation of the processing node. In the final part of the paper we also discuss the importance of the non-blocking communication and how minor changes in the communication pattern (e.g. replacing blocking communication with non-blocking communication) can substantially improve the application performance.

Secondary contributions include a detailed description of the innovative software design and the global coordination mechanisms, and an extensive performance evaluation with synthetic benchmarks and scientific applications. We also demonstrate the potential of using hardware mechanisms in the interconnection network to perform global coordination.

The rest of this paper is organized as follows. In Section 2 we describe the BCS core mechanisms that are the basis of BCS-MPI. Section 3 presents various design issues of BCS-MPI. In Section 4, the implementation of BCS-MPI is discussed. Performance evaluation results are shown and analyzed in Section 5. Finally, we present our concluding remarks and directions for future work in Section 6.

2 The BCS Core Primitives

Our goals in identifying the BCS core primitives were *simplicity* and *generality*. We therefore defined our abstraction layer in terms of only three operations. Nevertheless, we believe this layer encapsulates most communication and synchronization mechanisms required by the system software components. The primitives we use are as follows:

Xfer-And-Signal Transfers a block of data from local memory to the global memory of a set of nodes (possibly a single node). Optionally signals a local and/or a remote event upon completion.

Test-Event Polls a local event to see if it has been signaled. Optionally, blocks until it is.

Compare-And-Write Compares (using $\geq$, $<$, $=$, or $\neq$) a global variable on a set of nodes to a local value. If the condition is true on *all* nodes, then (optionally) assigns a new value to a – possibly different – global variable.

The following are some important points about the mechanisms' semantics:

1. *Global data* refers to data at the same virtual address on all nodes. Depending on the implementation, global data may reside in main memory or network-interface memory.

2. Xfer-And-Signal and Compare-And-Write are both atomic operations. That is, Xfer-And-Signal either *puts* data to *all* nodes in the destination set (which could be a single node) or – in case of a network error – *no* nodes. The same condition holds for Compare-And-Write when it writes a value to a global variable. Furthermore, if multiple nodes simultaneously initiate Compare-And-Writes with overlapping destination sets 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 [15].

3. Although Test-Event and Compare-And-Write are traditional, blocking operations, Xfer-And-Signal is non-blocking. The only way to check for completion is to Test-Event on a local event that Xfer-And-Signal signals.

4. The semantics do not dictate whether 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).

Quadrics' QsNet network [19], which we chose for our initial implementation, provides these primitives at hardware level: ordered, reliable multicasts; network conditionals (which return *True* if and only if a condition is *True* on *all* nodes); and events that can be waited upon and remotely signaled. We also quote some expected performance numbers from the literature about other networks, for the two global operations. In some of these networks (Gigabit Ethernet, Myrinet and Infiniband) the BCS primitives need to be emulated through a thin software layer, while in the other networks there is a one-to-one mapping with native hardware mechanisms.

We argue that in both cases – with or without hardware support – the BCS primitives represent an ideal abstract machine that on the one hand can export the raw performance of the network, and on the other hand can provide a general-purpose basis for designing simple and efficient system software. While in [8] we demonstrated their utility for resource management tasks, this paper focuses on their usage as a basis for a user-level communication library, BCS-MPI.

3 BCS-MPI Design

BCS-MPI is a novel implementation of MPI that globally schedules the system activities on all the nodes: a synchronization broadcast message or *global strobe* – implemented with the BCS core primitive Xfer-And-Signal – is sent to all nodes at regular intervals or *time slices*. Thus, all the system activities are tightly coupled since they occur concurrently on all the nodes. Both computation and communication are scheduled and the communication requests generated by each application process are buffered. At the beginning of every time slice a partial exchange of communication requests – implemented with the BCS core primitives Xfer-And-Signal and Test-Event – provides information to schedule the communication requests issued during the previous time slice. Consequently, all the scheduled communication operations are performed using the BCS core primitives Xfer-And-Signal and Test-Event.

TABLE 1: Measured/expected performance of the BCS core mechanisms as a function of the number of nodes n

Network	Compare-and-Write (μs)	Xfer-and-Signal (MB/s)
Gigabit Ethernet [25]	$46 \log n$	Not available
Myrinet [3, 4, 5]	$20 \log n$	$\sim 15n$
Infiniband [12]	$20 \log n$	Not available
QsNet ([19])	< 10	$> 150n$
BlueGene/L [10]	< 2	$700n$

The BCS-MPI communication protocol is implemented almost entirely in the network interface card (NIC). This enables BCS-MPI to overlap the communication with the computation executed on the host CPUs. The application processes interact directly with threads running on the NIC. When an application process invokes a communication primitive, it posts a descriptor in a region of NIC memory that is accessible to a NIC thread. Such a descriptor includes all the communication parameters that are required to complete the operation. The actual communication will be performed by a set of cooperating threads running on the NICs involved in the communication protocol. In the Quadrics network these threads can directly read/write from/to the application process memory space so that no copies to intermediate buffers are needed. The communication protocol is divided into microphases within every time slice and its progress is also globally synchronized, as described in Section 4.2.

To better explain how BCS-MPI communication primitives work, two possible scenarios for blocking and non-blocking MPI point-to-point primitives are described below.

3.1 Blocking Send/Receive Scenario

In this scenario, 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_Recv (see Figure 2(a)):

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.
6. P_1 and P_2 resume computation.

Note that the delay per blocking primitive is 1.5 time slices on average. However, this performance penalty can be alleviated by using non-blocking communication (see Section 5.4) or by scheduling a different parallel job in time slice $i + 1$.

3.2 Non-Blocking Send/Receive Scenario

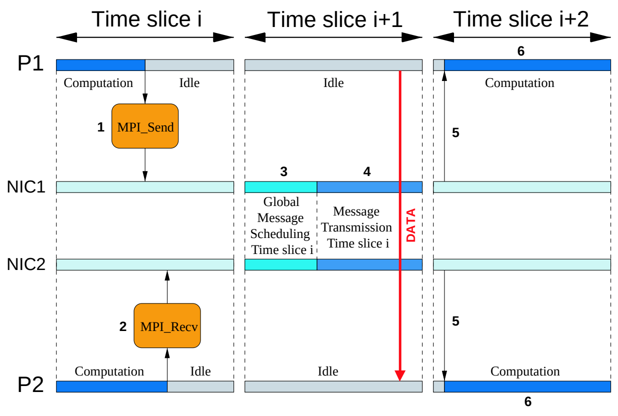
In this scenario, a process P_1 sends a message to process P_2 using MPI_Isend and process P_2 receives a message from P1 using MPI_Irecv (see Figure 2(b)):

1. P_1 posts a send descriptor to the NIC.
2. P_2 posts a receive descriptor to the NIC.
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).
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 verify that the communication has been performed and continue their computation.

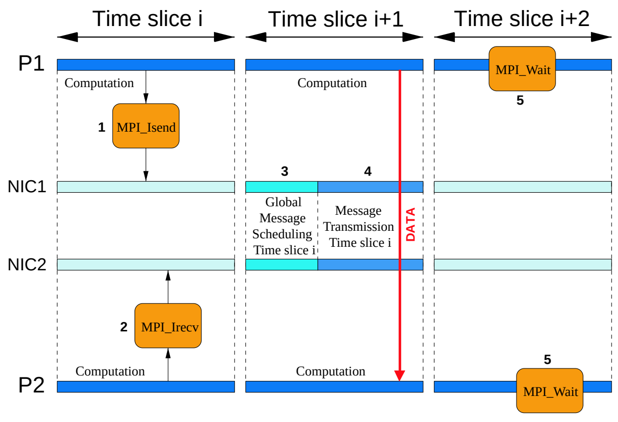
In this scenario, the communication is completely overlapped with the computation with no performance penalty.

4 BCS-MPI Implementation

To evaluate and validate the framework proposed in the previous section, we developed a fully functional version of BCS-MPI for QsNet-based systems. For quick prototyping and portability, BCS-MPI is initially implemented as a user-level communication library, and



(a) Blocking MPI_Send/MPI_Recv



(b) Non-Blocking MPI_Send/MPI_Recv

Figure 2: Blocking and Non-Blocking MPI_Send/MPI_Recv Scenarios

some typical kernel level functionalities such as process scheduling are implemented with the help of dæmons. This user-level implementation is expected to be slower than a kernel-level one, though more flexible and easier to use. An overview of the software structure of BCS-MPI is provided in Figure 3.

The communication library is hierarchically designed on top of a small set of communication/synchronization primitives, the BCS core primitives (Figure 4(a)), while higher-level primitives (the BCS API, described in Appendix A) are implemented on top of the BCS core. This approach greatly simplifies the design and implementation of BCS-MPI in terms of complexity, maintainability and extensibility. BCS-MPI is built on top of the BCS API by simply mapping MPI calls to BCS calls (see Appendix A). Note that scalability is enhanced by tightly coupling the BCS core primitives with the collective primitives provided at hardware level by the interconnection network.

BCS-MPI is integrated in STORM [8], a scalable, flexible resource management system for clusters, running on Pentium-, Itanium2- and Alpha-based architectures. STORM exploits low-level collective communication mechanisms to offer high-performance job launching and resource management. In this way, we provide the necessary infrastructure to run MPI parallel jobs using BCS-MPI.

The rest of this section describes the architecture of BCS-MPI in terms of the processes and NIC threads that compose the BCS-MPI runtime system, the global synchronization protocol, and the communication protocols for the point-to-point and collective primitives.

4.1 Processes and Threads

With the current user-level implementation, the BCS-MPI runtime system consists of a set of dæmons and a set of threads running on the NIC. The processes and NIC threads that constitute the BCS-MPI runtime system are shown in Figure 4(b). The Machine Manager (MM), runs on the management node. This dæmon coordinates the use of system resources issuing regular heartbeats and controls the execution of parallel jobs. The Strobe Sender (SS) is a NIC thread forked by the MM that implements the global synchronization protocol as described in Section 4.2. The Node Manager (NM) dæmons run on every compute node. This process executes all the commands issued by the MM, manages the local resources, and schedules the execution of the local processes. The Strobe Receiver (SR), the Buffer Sender (BS), the Buffer Receiver (BR), the DMA Helper (DH), the Collective Helper (CH) and the Reduce Helper (RH) are all NIC threads forked by the NM in each compute node. The SR is the counterpart of the SS in the compute nodes and coordinates the execution of all the local threads. The BS and the BR handle the descriptors posted by the application processes whenever a communication primitive is invoked, and schedule the point-to-point and collective communication operations. The DH carries out the actual data transmission for the point-to-point operations. Finally, the CH and the RH perform the barrier and broadcast operations, and the reduce operations, respectively.

4.2 Global Synchronization Protocol

The BCS-MPI runtime system globally schedules all the computation, communication and synchronization activities of the MPI jobs at regular intervals. Each time

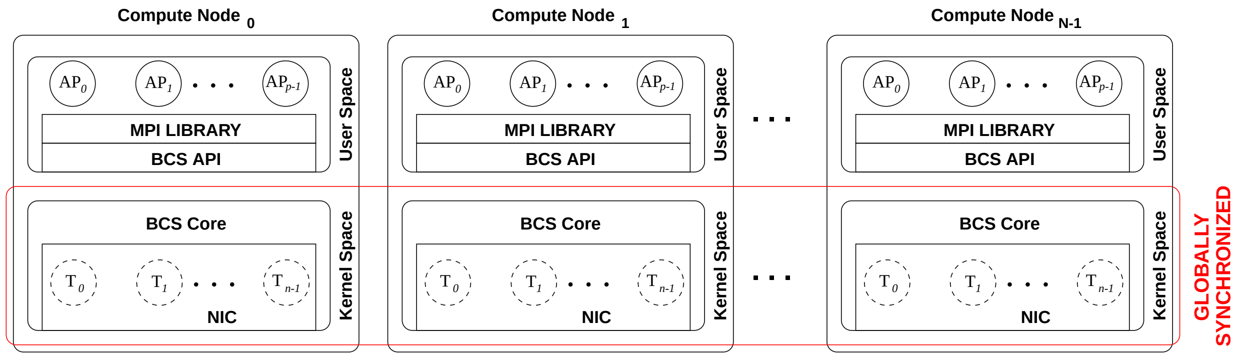
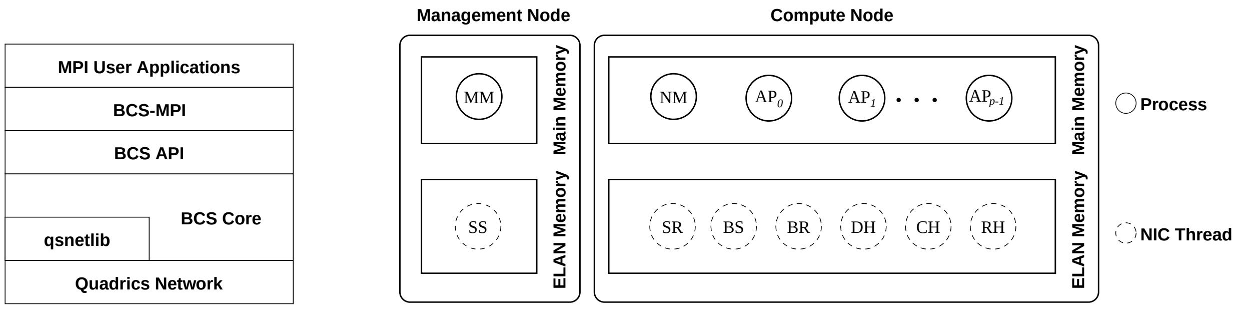


Figure 3: BCS-MPI Overview



(a) Library Hierarchy

(b) Processes and Threads

Figure 4: BCS-MPI Architecture

slice is divided into two main phases and several microphases, as shown in Figure 5. The two phases are the *global message scheduling* and the *message transmission*. The global message scheduling phase schedules all the descriptors posted to the NIC during the previous time slice. A partial exchange of control information is performed during the *descriptor exchange microphase* (DEM). The point-to-point and collective communication operations are scheduled in the *message scheduling microphase* (MSM) using the information gathered during the previous microphase. The *message transmission phase* performs point-to-point operations, barrier and broadcast collectives, and the reduce operations, respectively, during its three microphases.

In order to implement the global synchronization mechanism, the SS and the SR threads synchronize at the beginning of every microphase with a microstrobe implemented using Xfer-And-Signal. The SS checks whether all the nodes have completed the current microphase (using Compare-And-Write) and, if so, sends a microstrobe to all the SRs. The SR running on every node consequently wakes up the local NIC thread(s) that must be active in the new microphase. The BS and the BR run during the descriptor exchange mi-

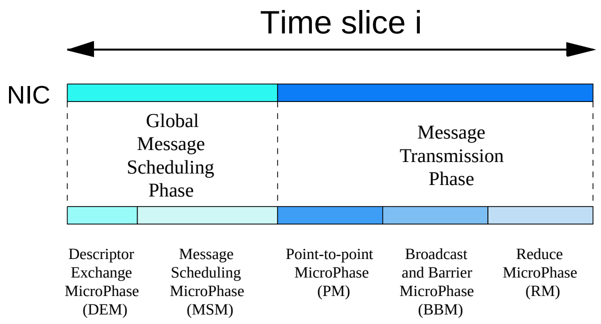

Time slice i
NIC
Global Message Scheduling Phase
Message Transmission Phase
Descriptor Exchange MicroPhase (DEM)
Message Scheduling MicroPhase (MSM)
Point-to-point MicroPhase (PM)
Broadcast and Barrier MicroPhase (BBM)
Reduce MicroPhase (RM)

Figure 5: Global synchronization protocol

crophase to process the descriptors and during the message scheduling microphase to schedule the messages. The DH, the CH and the RH run during the point-to-point microphase, the broadcast and barrier microphase, and the reduce microphase, respectively, to perform all the operations scheduled for execution in the global message scheduling phase.

4.3 Point-to-point

As shown in Figure 2, every time a user process invokes a point-to-point MPI primitive, it initializes a descriptor in a region of memory accessible to the NIC threads which will initiate the operation on its behalf. All the descriptors for either blocking or non-blocking send operations are posted to the BS thread while all the descriptors for either blocking or non-blocking receive operations are posted to the BR thread. Each application process involved in the communication protocol is suspended only if the invoked primitive is blocking. All the descriptors posted during time slice $i - 1$ will be scheduled for execution, if possible, at time slice i as follows (see Figure 6 for further details).

Descriptor Exchange Microphase The BS delivers each send descriptor posted in time slice $i - 1$ to the BR running on the destination node.

Message Scheduling Microphase The BR matches the remote send descriptor list against the local receive descriptor list. For each matching pair, the BR builds a matching descriptor with all the information required to complete the data transfer, and schedules the point-to-point operation for execution. If the message is too large and cannot be scheduled within a single time slice, the BR splits it into smaller chunks. The first chunk of the message is scheduled during the current time slice and the remaining chunks in the following time slices. In the current implementation, these two phases take approximately 125 μs.

Point-to-point Microphase For each matching descriptor created in the previous microphase by the BR, the DH performs the real data transmission. Note that no intervention from the two application processes involved is required.

4.4 Collective Communication

Every time a user process calls a collective MPI function such as MPI_Barrier, MPI_Broadcast, MPI_Reduce or MPI_Allreduce, BCS-MPI posts a descriptor to the BR thread, which in turn initiates the operation on its behalf, and blocks. The BR pre-processes all the collective descriptors. If all the local processes of a parallel job have invoked the collective primitive, a local flag for that job is set. Following that, all the collective descriptors, except for those corresponding to the job master processes, are discarded. All the descriptors posted during time slice $i - 1$ will be scheduled, if possible, in time slice i as follows:

Message Scheduling Microphase For each collective descriptor corresponding to a job master process, the BR tests if all the application processes of that MPI parallel job had invoked the collective primitive in all nodes. In order to accomplish this, the BR issues a query broadcast (using Compare-And-Write) message that checks the flag for that job in all the nodes. If the flag is set on all nodes, the collective operation is scheduled for execution.

Broadcast and Barrier/Reduce Microphase The scheduled broadcast operations are performed by the CH broadcasting the data to all the processes of the MPI parallel job. The barrier operation is a special case of a broadcast operation with no data. The scheduled reduce operations are carried out by the RH on the NIC by using a binomial tree to gather the partial reduce results. The QsNet NIC has no floating-point unit. Hence, an IEEE compliant library for binary floating-point arithmetic has been used to compute the reduce in the NIC (SoftFloat [30]). Since most applications reduce over a very small number of elements [28, 16], computing the reduce in the NIC is faster than sending the data through the PCI bus to perform the operation in the host [16].

Figure 7 illustrates the execution of a broadcast operation. The MPI program in this example is composed of four processes running on two different nodes.

4.5 Features and Limitations

This section discusses some important features of the current user-level implementation of BCS-MPI.

- MPI groups are not fully implemented yet.
- The NM dæmon that belongs to the BCS-MPI runtime system schedules the user processes at every time slice, instead of the kernel.

Since we have little control over the OS scheduler at user level, the NM dæmon may not always be scheduled on time. This anomaly introduces noise in the system that can potentially cause a considerable performance degradation [20]. To eliminate this problem, we are developing a kernel-based implementation of BCS-MPI.

In order to avoid the overhead of a system call to post the descriptors, we use a FIFO queue in a shared memory region accessible by both the application process and the kernel.

5 Experimental Results

In this section we compare the performance of our user-level implementation of BCS-MPI to that of Quadrics MPI using several benchmarks and applications. Quadrics MPI [31] is a production-level implementation for QsNet-based systems, based on MPICH

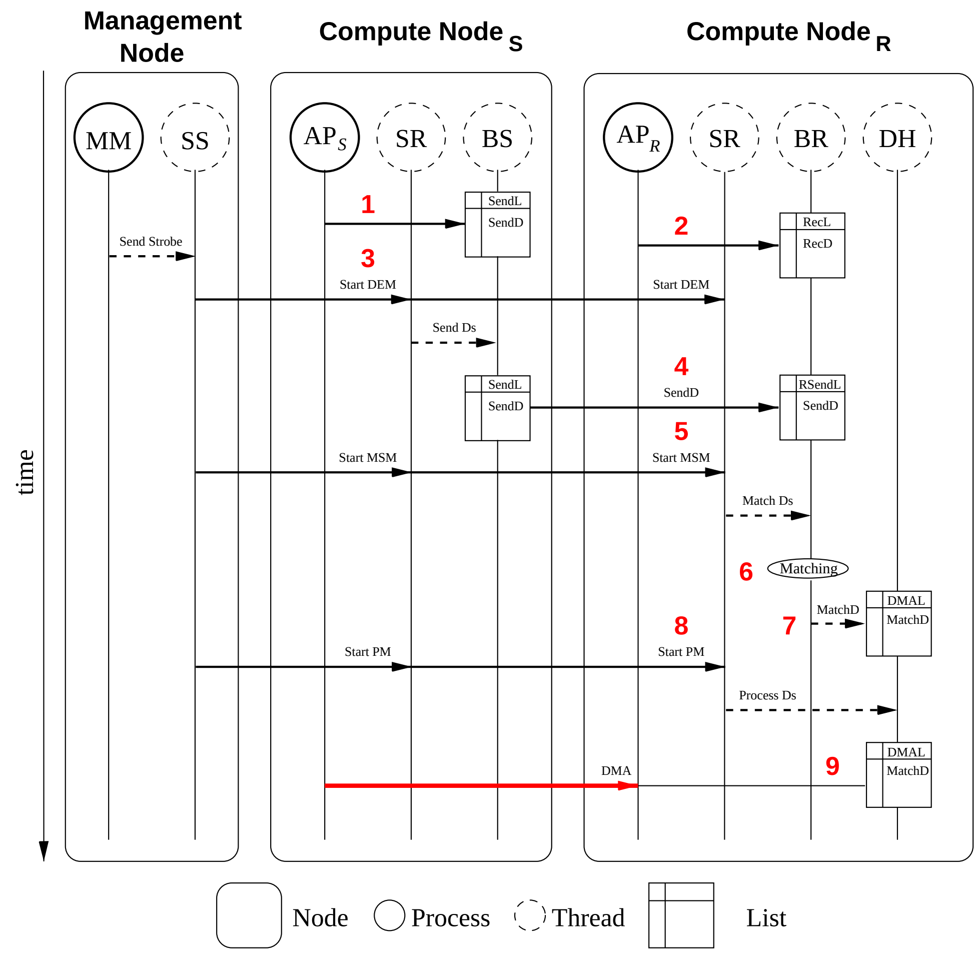


Figure 6: Send/Receive Scenario: (1) The sender process posts a descriptor to the BS (2) The receiver process posts a descriptor to the BR (3) SS sends a microstrobe to signal all the SRs the beginning of the Descriptor Exchange Microphase (DEM) (4) BS sends the descriptor to the BR running on the receiving end (5) SS sends a microstrobe to signal the beginning of the Message Scheduling Microphase (MSM) (6) BR matches the remote send and the local receive descriptors (7) SS sends a microstrobe to signal the beginning of the Point-to-point Microphase (PM) (8) BR schedules the operation for execution (9) DH performs the get (one-sided communication).

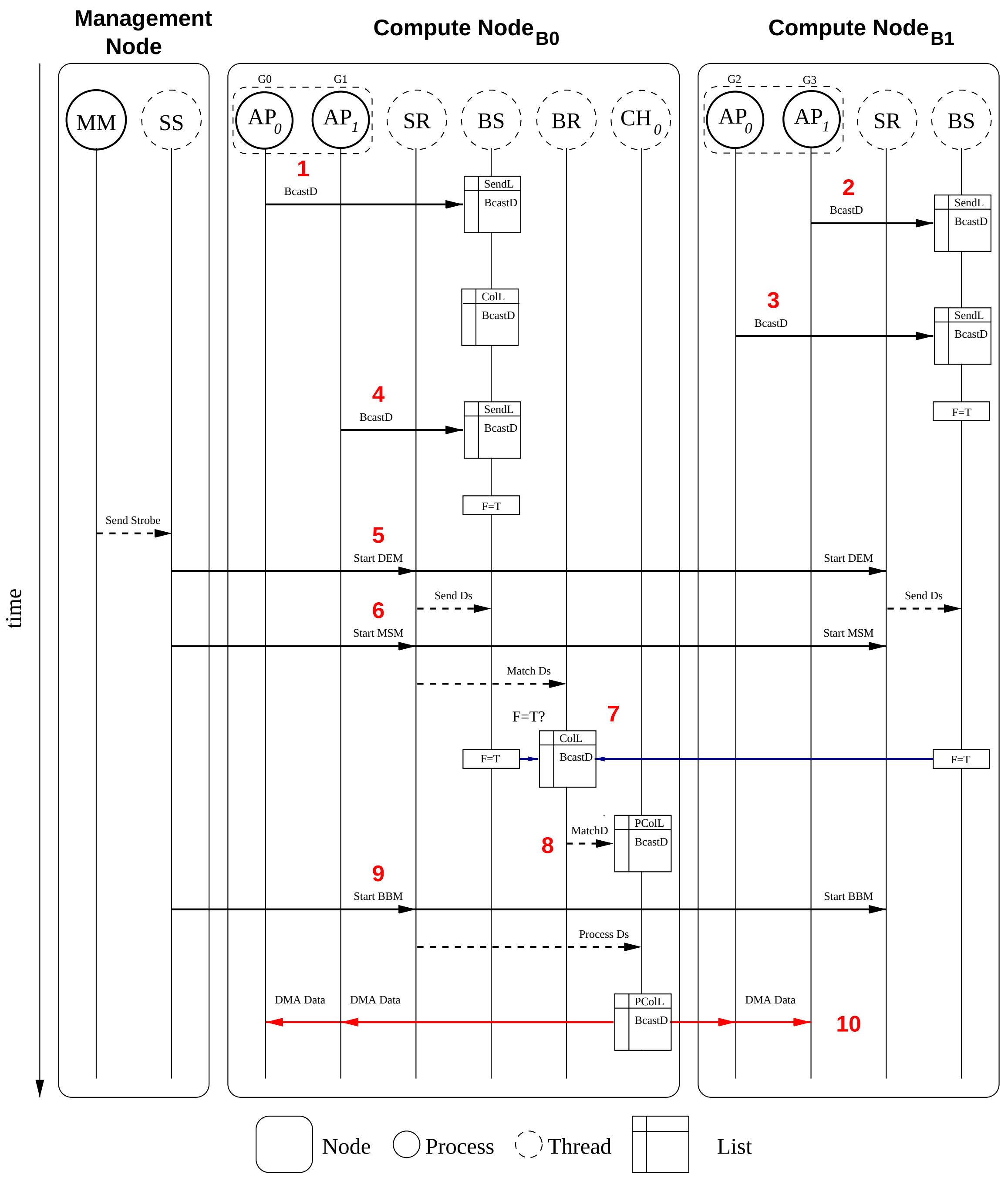


Figure 7: Broadcast Scenario (1) Application Process (AP) G_0 posts a descriptor to the local BS. G_0 is the master process and its descriptor is copied to the Collective List (2) G_3 posts a descriptor to the local BS. The descriptor is processed and discarded (3) G_2 posts a descriptor to the local BS. The descriptor is processed: all the local processes have reached the barrier and Flag *F* is set to True. Descriptor is discarded (4) G_4 posts a descriptor to the local BS. The descriptor is processed: all the local processes have reached the barrier and Flag *F* is set to True. The descriptor is discarded (5) SS sends a microstrobe to signal all the SRs the beginning of the Descriptor Exchange Microphase (DEM) (6) SS sends a microstrobe to signal the beginning of the Message Scheduling Microphase (MSM) (7) BR checks whether all the processes are ready (8) BR schedules the broadcast operation for execution (9) SS sends a microstrobe to signal the beginning of the Broadcast and Barrier Microphase (BBM) (10) CH performs the broadcast.

1.2.4. Quadrics MPI is currently used by six of the ten fastest systems in the Top500 list [32], at the time of this writing. To evaluate and validate our implementation of BCS-MPI we use a set of synthetic benchmarks, the NAS suite of benchmarks, and two real applications which are representative of the ASCI workload at LANL.

5.1 Experimental Setup

The hardware used for the experimental evaluation is the "crescendo" cluster at LANL/CCS-3. This cluster consists of 32 compute nodes (Dell 1550), one management node (Dell 2550), and a 128-port Quadrics switch [19, 23] (using only 32 of the 128 ports). Each compute node has two 1 GHz Pentium-III processors, 1 GB of ECC RAM, two independent 66MHz/64-bit PCI buses, a Quadrics QM-400 Elan3 NIC [19, 22, 24] for the data network, and a 100Mbit Ethernet NIC for the management network. All the nodes run Red Hat Linux 7.3, and use kernel modules provided by Quadrics and the low-level communication library qsnetlibs v1.5.0-0 [31]. All the benchmarks and the applications analyzed in this section are compiled with the Intel C/Fortran Compiler v5.0.1 for IA32 using the -O3 optimization flag. Finally, a $500\mu s$ time slice is used by BCS-MPI for all the experiments in this paper.

5.2 Synthetic Benchmarks

Many scientific codes display a bulk-synchronous behavior [27] and can be characterized by a nearest-neighbor communication stencil, optionally followed by a global synchronization operation such as barrier, broadcast or reduce [11, 13]. Therefore, we designed two synthetic benchmarks that represent this pattern to compare our experimental BCS-MPI with the production-level Quadrics MPI.

In the first synthetic benchmark, every process computes for a parametric amount of time and globally synchronizes with all the other processes in a loop. Figure 8(a) shows the slowdown of BCS-MPI when compared to Quadrics MPI for different computational granularities. As expected, the slowdown decreases as we increase the computational granularity since the effect of the delay introduced by the barrier synchronization is amortized. The figure shows that the slowdown is less than 7.5% with a computation granularity of 10 ms when we run this benchmark on the entire machine. Figure 8(b) shows the slowdown of BCS-MPI versus Quadrics MPI as a function of the number of processes. In this case, the results indicate that BCS-MPI scales well for barrier synchronization operations, and it is almost insensitive to the number of processors.

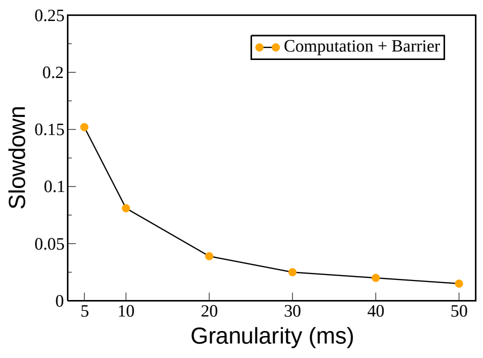
In the second synthetic benchmark, every process computes for a parametric amount of time, exchanges a fixed number of non-blocking point-to-point messages with a set of neighbors, and waits for the completion of all the communication operations in a loop. The slowdown for different computational granularities is shown in Figure 8(c). Like in the previous case, the slowdown decreases as the computational granularity increases, remaining below 8% for granularities larger than 10 ms. Finally, from Figure 8(d) we can observe that BCS-MPI scales well with point-to-point operations too.

5.3 NAS Benchmarks and Applications

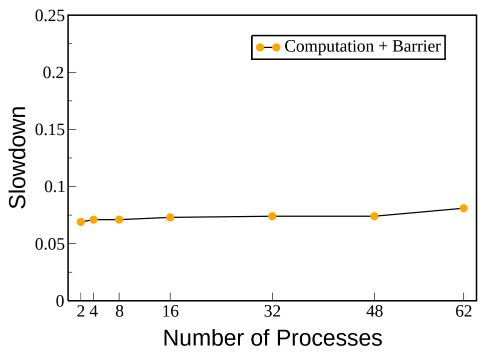
In this section we use the NAS Parallel Benchmarks (NPB 2.4) [1] and SAGE (SAIC's Adaptive Grid Eulerian hydrocode) [13]. The NAS Parallel Benchmarks are a set of eight programs designed to help in evaluating the performance of parallel supercomputers. The suite, which is derived from computational fluid dynamics (CFD) applications, consists of five kernels and three applications. Since BCS-MPI does not support MPI groups yet, we were only able to use four kernels and one application: Integer Sort (IS), Embarrassingly Parallel (EP), Conjugate Gradient (CG), Multigrid (MG) and LU solver (LU). All programs are written in Fortran 77 (except for IS which is written in C) and use MPI for inter-processor communications. All the benchmarks were compiled for the *class C* workload.

SAGE is a multidimensional (1D, 2D and 3D), multimaterial, Eulerian, hydrodynamics code with adaptive mesh refinement. SAGE represents a large class of production ASCI applications at LANL. SAGE comes from LANL's Crestone project, whose goal is the investigation of continuous adaptive Eulerian techniques to stockpile stewardship problems. It is characterized by a nearest-neighbor communication pattern that uses non-blocking communication operations followed by a reduce operation at the end of each compute step. The code is written in Fortran 90 and uses MPI for inter-process communications. The *timing.input* data set was used in all the experiments. In each case, we compare the runtime of BCS-MPI to that of Quadrics MPI, and analyze the results. The final runtime was computed as the average of five executions.

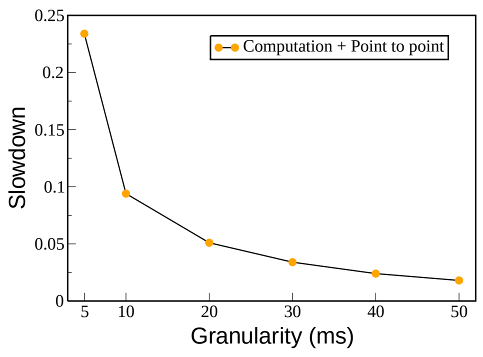
The run times of NPB and SAGE for both Quadrics MPI and BCS-MPI are shown in Figure 9. The slowdown of BCS-MPI in comparison to Quadrics MPI is computed in Table 2. All NPB benchmarks (except LU) and SAGE perform reasonably well with BCS-MPI. The



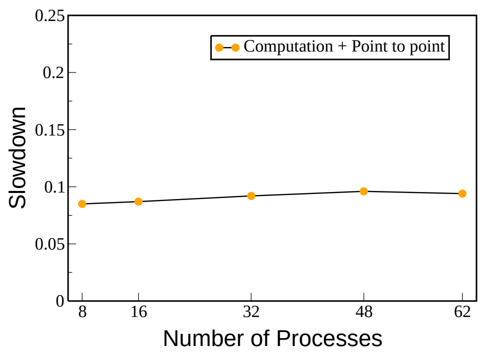
(a) Computation and Barrier: 62 processes



(b) Computation and Barrier: 10 ms granularity



(c) Computation and Nearest-neighbor Communication: 62 processes, 4 neighbors and 4KB messages



(d) Computation and Nearest-neighbor Communication: 10 ms granularity, 4 neighbors and 4KB messages

Figure 8: Synthetic Benchmarks

NPB are coarse-grained bulk-synchronous applications that, as discussed in Section 5.2, show an expected moderate slowdown of up to 8%. However, three programs do not meet the expectations. IS takes approximately 12s to run in this configuration and consequently pays a relatively high price for the overhead of initializing the BCS-MPI runtime system. CG and LU use several consecutive blocking calls inside a loop which introduce a considerable delay, since no overlap between computation and communication is possible for several time slices. This problem can be mitigated by using non-blocking communication, as described in the context of SWEEP3D in Subsection 5.4 below.

SAGE is a medium-grained application and the non-blocking communications mitigate the performance penalty of the global synchronization operation performed at the end of each compute step. The slight performance improvement is obtained thanks to the negligible overhead of the non-blocking calls, that only initialize a communication descriptor.

TABLE 2: Benchmark and Application Slowdown

Application	**Slowdown**
SAGE	-0.42%
SWEEP3D	-2.23%
IS	10.14%
EP	5.35%
MG	4.37%
CG	10.83%
LU	15.04%

5.4 Blocking vs. Non-blocking Communications

As stated in Section 5.3, bulk-synchronous applications with non-blocking or infrequent blocking communications run efficiently with BCS-MPI. However, fine-grained applications that use blocking communi-

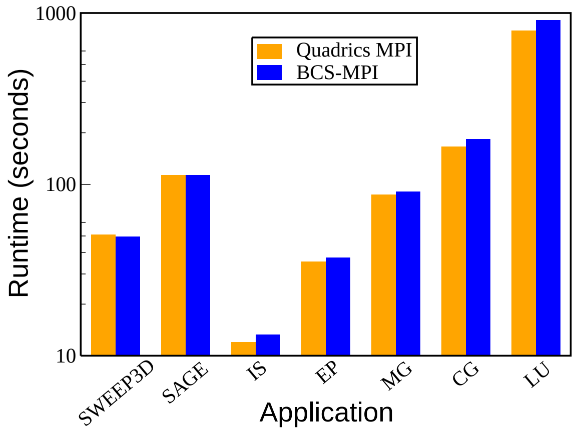


Figure 9: Benchmarks and Applications

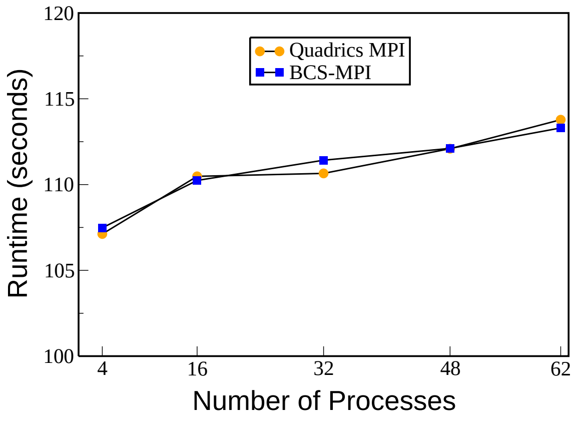


Figure 10: SAGE

cations or applications that group blocking communications are expected to perform poorly with BCS-MPI. The delays introduced by the blocking communications can considerably increase the applications' run time. Two approaches can alleviate this problem. The simplest option is to schedule a different parallel job whenever the application blocks for communication, thus making use of the CPU. This addresses the problem without requiring any code modification, but is not always practical due to memory and performance considerations. Alternatively, we have empirically seen that in such cases it is often possible to transform the blocking communication operations into non-blocking ones, with a few simple code modifications.

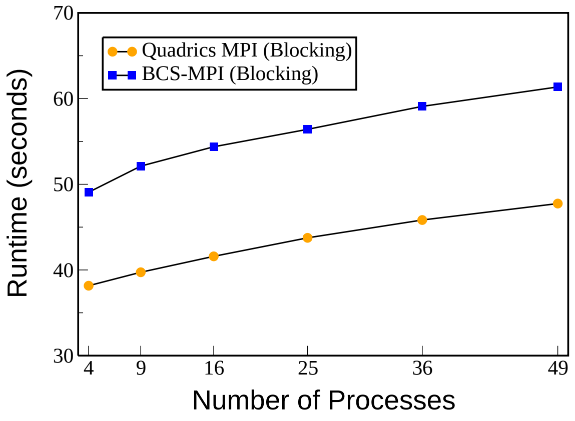
To illustrate the second technique, we look at the SWEEP3D application [11]. SWEEP3D is a time-independent, Cartesian-grid, single-group, discrete ordinates, deterministic, particle transport code. SWEEP3D represents the core of a widely used method of solving the Boltzmann transport equation. Estimates are that deterministic particle transport accounts for 50–80% of the execution time of many realistic simulations on current DOE ASCI systems. SWEEP3D is characterized by a fine granularity (each compute step takes $\approx$ 3.5ms) and a nearest-neighbor communication stencil with blocking send/receive operations.

Figure 11(a) shows the run time of SWEEP3D for both Quadrics MPI and BCS-MPI as a function of the numbers of processes. The slowdown is approximately 30% in all configurations. Each process exchanges four messages with its nearest neighbors on every compute step using blocking send/receive operations. This communication pattern together with the fine granularity incurs a very high overhead. On every compute step, the process will block for 1.5 time slices on average for every blocking operation. To eliminate this delay, we replaced every matching pairs of MPI_Send/MPI_Recv with MPI_Isend/MPI_Irecv and added MPI_Waitall at the end. That involved changing less than fifty lines of source code and improved dramatically the application performance, as shown in Figure 11(b). In this case, the overlapping of computation and communication along with the minimal overhead of the MPI calls allow BCS-MPI to slightly outperform Quadrics MPI.

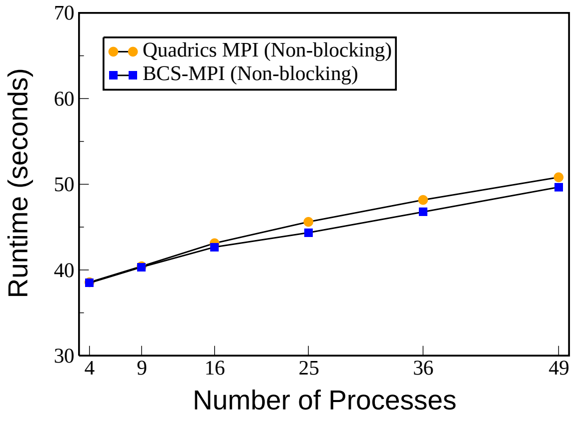
6 Conclusions and Future Work

This paper presented an alternative approach to the design of communication libraries for large-scale parallel computers. Rather than following the beaten track of most communication protocols which focus on optimizing latency and bandwidth of pairs of communicating processes, BCS-MPI tries to optimize the *global* state of the machine in order to reduce the system software complexity. We have provided insight on the global coordination protocols used by BCS-MPI and described a prototype implementation running almost entirely on the network interface of the Quadrics network.

The experimental results have shown that the performance of BCS-MPI is comparable to the production-level MPI for most applications. The performance of some applications, as SWEEP3D, can be improved by modifying their communication pattern from a blocking one to a non-blocking one (typically with minimal changes). Such applications can actually improve their performance when compared to the production level



(a) Blocking Version



(b) Non-Blocking Version

Figure 11: SWEEP3D

MPI, thanks to BCS-MPI's low overhead in the compute nodes.

These results pave the way to future advances in the design of the system software for large-scale parallel machines. We argue that with a globally constrained system such as the one put forth with BCS-MPI, it is possible to substantially simplify the implementation of the resource management software, communication libraries, and parallel file system. Moreover, a scheduled, deterministic communication behavior at system level could provide a solid infrastructure for implementing transparent fault tolerance.

System-level fault tolerance is our main path for future research. However, we also plan to study the advantages of this simplified model for implementing system wide parallel I/O, scheduling, and kernel-level system management.

Acknowledgments

The authors would like to thank David Addison for his help in debugging the Elan thread code, which is at the base of BCS-MPI.

This work was supported by the U.S. Department of Energy through the project LDRD-ER 2001034ER ”Resource Utilization and Parallel Program Development with Buffered Coscheduling” and Los Alamos National Laboratory contract W-7405-ENG-36.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991. Available from `http://www.nersc.gov/~dhb/dhbpapers/benijsa.ps`.

[2] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, Copper Mountain, CO, December 1995. Available from `http://www.cs.cornell.edu/tve/u-net/papers/sosp.pdf`.

[3] Raoul A.F. Bhoedjang, Tim Rühl, and Henri E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *27th International Conference on Parallel Processing (ICPP98)*, pages 381–390, Minneapolis, MN, August 1998. Available from `http://www.cs.cornell.edu/raoul/papers/multicast98.pdf`.

[4] Darius Buntinas, Dhabaleswar Panda, José Duato, and P. Sadayappan. Broadcast/Multicast over Myrinet using NIC-Assisted Multidestination Messages. In *Workshop on Communication, Architecture, and Applications for*

Network-Based Parallel Computing (CANPC '00), High Performance Computer Architecture (HPCA-6) Conference, Toulouse, France, January 2000. Available from `ftp://ftp.cis.ohio-state.edu/pub/communication/papers/canpc00-nic-multicast.pdf`.

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

[6] Giovanni Chiola and Giuseppe Ciaccio. GAMMA: a Low-cost Network of Workstations Based on Active Messages. In *Proceedings of 5th EUROMICRO workshop on Parallel and Distributed Processing (PDP'97)*, London, UK, January 1997. Available from `ftp://ftp.disi.unige.it/pub/project/GAMMA/pdp97.ps.gz`.

[7] Compaq, Intel, and Microsoft. The Virtual Interface Architecture (VIA) Specification. Available Available from `http://www.viarch.org`.

[8] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. STORM: Lightning-Fast Resource Management. In *Proceedings of SC2002*, Baltimore, Maryland, November 16–22 2002. Available from `http://sc-2002.org/paperpdfs/pap.pap297.pdf`.

[9] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI - The Complete Reference*, volume 2, The MPI Extensions. The MIT Press, 1998.

[10] Manish Gupta. Challenges in Developing Scalable Scalable Software for BlueGene/L. In *Scaling to New Heights Workshop*, Pittsburgh, PA, May 2002. Available from `http://www.psc.edu/training/scaling/gupta.ps`.

[11] Adolfy Hoisie, Olaf Lubeck, Harvey Wasserman, Fabrizio Petrini, and Hank Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of the 2000 International Conference on Parallel Processing (ICPP-2000)*, Toronto, Canada, August 21–24, 2000. Available from `http://www.c3.lanl.gov/~fabrizio/papers/icpp00.pdf`.

[12] Infiniband Trade Association. Infiniband Specification 1.0a, June 2001. Available from `http://www.infinibandta.org`.

[13] Darren J. Kerbyson, Hank J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman, and Michael Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proceedings of SC2001*, Denver, Colorado, November 10–16, 2001. Available from `http://www.sc2001.org/papers/pap.pap255.pdf`.

[14] Ken 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. Available from `http://www.ccs.lanl.gov/salishan02/koch.pdf`.

[15] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[16] Adam Moody, Juan Fernández, Fabrizio Petrini, and Dhabaleswar Panda. Scalable NIC-based Reduction on Large-scale Clusters. In *Proceedings of SC2003*, Phoenix, Arizona, November 10–16, 2003. Available from `http://www.c3.lanl.gov/~fabrizio/papers/sc03_reduce.pdf`.

[17] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of IEEE/ACM Supercomputing 1995 (SC'95)*, San Diego, CA, December 1995. Available from `http://www.supercomp.org/sc95/proceedings/567_SPAK/SC95.PDF`.

[18] Fabrizio Petrini and Wu-chun Feng. Improved Resource Utilization with Buffered Coscheduling. *Journal of Parallel Algorithms and Applications*, 16:123–144, 2001. Available from `http://www.c3.lanl.gov/~fabrizio/papers/paa00.ps`.

[19] Fabrizio Petrini, Wu-chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January/February 2002. ISSN 0272-1732. Available from `http://www.computer.org/micro/mi2002/pdf/m1046.pdf`.

[20] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance

on the 8,192 Processors of ASCI Q. In *Proceedings of SC2003*, Phoenix, Arizona, November 10–16, 2003. Available from `http://www.c3.lanl.gov/~fabrizio/papers/sc03_noise.pdf`.

[21] Loïc Prylli and Bernard Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. In *Proceedings of IPPS/SPDP'98 Workshop on Personal Computer Based Networks of Workstations*, Orlando, FL, April 1998. Available from `http://ipdps.eece.unm.edu/1998/pc-now/prylli.pdf`.

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

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

[24] Quadrics Supercomputers World Ltd. *Elan Programming Manual*, May 2002.

[25] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceedings of SuperComputing 2001 (SC'01)*, Denver, Colorado, November 10–16, 2001. Available from `http://www.sc2001.org/papers/pap.pap315.pdf`.

[26] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI - The Complete Reference*, volume 1, The MPI Core. The MIT Press, 1998.

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

[28] Jeffrey S. Vetter and Frank Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. In *Proceedings of the International Parallel and Distributed Processing Symposium 2002 (IPDPS'02)*, Fort Lauderdale, FL, April 2002. Available from `http://www.csc.ncsu.edu/faculty/mueller/ftp/pub/mueller/papers/jpdc02.pdf`.

[29] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of 19th International Conference on Computer Architecture (ISCA'92)*, Gold Coast, Australia, May 1992. Available from `http://www.cs.cmu.edu/~seth/papers/isca92.pdf`.

[30] `http://www.jhauser.us/arithmetic/SoftFloat.html`.

[31] `http://www.quadrics.com`.

[32] `http://www.top500.org`.

7 Appendix A

The BCS communication primitives are listed in Figure 12. The point-to-point primitives and the basic collective primitives, that is, barrier, broadcast and reduce, are implemented in the NIC while the rest of them are built on top of those. The MPI communication primitives currently available and the corresponding BCS-MPI primitives are listed in Figure 13.

BCS Primitive	Description
bcs_send()	Blocking/non-blocking send
bcs_recv()	Blocking/non-blocking receive
bcs_probe()	Blocking/non-blocking test for a matching receive
bcs_test()	Blocking/non-blocking test for send/receive completion
bcs_testall()	Blocking/non-blocking test for multiple send/receive completions
bcs_barrier()	Barrier synchronization
bcs_bcast()	Broadcast
bcs_reduce()	Reduce and allreduce
bcs_scatter()	Vectorial/non-vectorial scatter
bcs_gather()	Vectorial/non-vectorial gather
bcs_allgather()	Vectorial/non-vectorial allgather
bcs_alltoall()	Vectorial/non-vectorial all-to-all

Figure 12: BCS API

MPI Primitive	BCS API Primitive
MPI_Send()	bcs_send(IN blocking)
MPI_Isend()	bcs_send(IN non-blocking, OUT BCS_Request)
MPI_Recv()	bcs_recv(IN blocking)
MPI_IRecv()	bcs_recv(IN non-blocking, OUT BCS_Request)
MPI_Probe()	bcs_probe(IN blocking, IN BCS_Request)
MPI_Iprobe()	bcs_probe(IN non-blocking, IN BCS_Request)
MPI_Test()	bcs_test(IN non-blocking, IN BCS_Request)
MPI_Wait()	bcs_test(IN blocking, IN BCS_Request)
MPI_Testall()	bcs_testall(IN non-blocking, IN BCS_Request+)
MPI_Waitall()	bcs_testall(IN blocking, IN BCS_Request+)
MPI_Barrier()	bcs_barrier()
MPI_Reduce()	bcs_reduce(IN non-all)
MPI_Allreduce()	bcs_reduce(IN all)
MPI_Scatter()	bcs_scatter(IN non-vectorial)
MPI_Scatterv()	bcs_scatter(IN vectorial)
MPI_Gather()	bcs_gather(IN non-vectorial)
MPI_Gatherv()	bcs_gather(IN vectorial)
MPI_Allgather()	bcs_allgather(IN non-vectorial)
MPI_Allgatherv()	bcs_allgather(IN vectorial)
MPI_Alltoall()	bcs_alltoall(IN non-vectorial)
MPI_Alltoallv()	bcs_alltoall(IN vectorial)

Figure 13: MPI-BCS Correspondence