Title: BCS MPI: A NEW APPROACH IN THE SYSTEM SOFTWARE DESIGN FOR LARGE-SCALE PARALLEL COMPUTERS

Author(s): Juan Fernandez, 183357, CCS-3
Fabrizio Petrini, 154601, CCS-3
Eitan Frachtenberg, 177896, CCS-3

Submitted to: Supercomputing 2003
Phoenix, AZ
November 2003
Abstract

Buffered CoScheduled (BCS) MPI proposes a new approach to design the communication libraries for large-scale parallel machines. The emphasis of BCS MPI is on the global coordination of a large number of processes rather than in the traditional optimization of the local performance of a pair of communicating processes. BCS MPI delays the interprocessor 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. The experimental results, executed on a set of scientific applications representative of the ASCI workload, show that BCS MPI is only marginally slower than the production-level MPI, but much simpler to implement, debug and analyze.

Keywords: MPI, communication protocols, cluster computing, large-scale parallel machines, STORM, system software

1 Introduction

The design and the implementation of scalable system software for a large-scale parallel computer is a challenging task. The experience in the development of ASCI-class machines [15] shows that it takes several years to design, implement, debug and optimize all the software stack before these machines become reliable and efficient production-level systems.

In a sense, the complexity of such machines has risen to a level that is comparable -if not higher- to that of the scientific simulations that are supposed to run. Such scientific simulations routinely use thousands of processors and each processor can have multiple outstanding messages at any given time of its execution, resulting in a very large and complicated global state.

Important aspects of the system software, other than the communication libraries, are the resource management -the software infrastructure that keeps track of the resources available and their allocation- the parallel file system and the fault tolerance. The current state of the art is to design all these components separately, in order to have a modular design and allow different groups to work in parallel without many dependencies. But all these elements of the system software have many parts in common, for example the communication protocols, that are implemented and re-implemented many times. In many cases the lack of coordination is also detrimental: most parallel systems cannot guarantee Quality of Service (QoS) between user-level traffic and system-level traffic in the high-performance interconnection network. Low-priority, best-effort traffic generated by the parallel file system can interfere with higher-priority, latency-sensitive traffic generated at user level. Also, system daemons that perform resource management can introduce computational holes of several hundreds of milliseconds that can severely impact fine-grained scientific simulations [12].

Buffered Coscheduling (BCS) [19] is a new design methodology for the system software that tries to attack these two problems -the complexity of a large scale parallel machine and the redundancy of the software. The vision behind BCS is that both size and complexity of the system software can be substantially reduced.

BCS tries to globally coordinate all the activities of
such machines at a fine granularity, in the order of a few hundreds of µseconds. The scheduling actions of all processing nodes and the communication pattern are strictly scheduled at regular intervals. All the operating systems in the processing nodes are integrated in a single system, and tick altogether at regular, fine-grained intervals. To provide some insight, BCS implements a global SIMD operating system that runs MIMD applications. But the computational granularity of BCS MPI is much coarser (hundreds of µseconds) than the SIMD machines of the eighties, that were synchronizing all the processors at every instruction.

The second innovative aspect is the vision that all the various parts of the system software could be implemented on top of a small set—just three—primitives, which we call the BCS core primitives [9]. Figure 1 shows that parallel file system, resource management and communication libraries are built on top of such primitives. We argue that the BCS core primitives are general enough to cover most needs in parallel software development and close enough to the hardware to expose the highest level of performance.

In [9] we proved that it is possible to implement a scalable resource manager, called STORM, that is orders of magnitude faster than existing production-level software, by using the BCS core primitives.

In this paper we try to generalize our research to the MPI communication library. 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 last decade has been to minimize the point-to-point latency by removing kernel overhead and moving the data communication into the user level [1, 3, 8, 10, 18, 22, 27, 29]. BCS MPI follows a different, at first sight backward, path: rather than optimizing each single point-to-point communication in isolation, we try to optimize globally the whole communication pattern. Communication is performed at kernel level [7], and scheduled only at the begin of a timeslice. The smallest latency that a message will experience will be at least one time slice, which is in the order of few hundreds of µseconds or more.

The primary contribution of this paper is to show that a constrained communication library such as BCS MPI provides almost 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 in the network interface, 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—for example replacing blocking communication with non-blocking communication—can substantially improve the 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. Another contribution is to demonstrate the great potential of having hardware mechanisms to perform global coordination in the interconnection network in a primitive form.

The rest of this paper is organized as follows. Section 2 describes the BCS core mechanisms. Section 3 presents BCS MPI. In Section 4, the implementation of BCS MPI is discussed. The performance evaluation results are shown and analyzed in Section 5. Finally, we present our concluding remarks and feasible ways of future work.

2 The BCS Core Primitives

Our goals in identifying the BCS core primitives were simplicity and generality. We, therefore, defined our layer in terms of only three operations, which we nevertheless believe encapsulate all of the communication and synchronization mechanisms required by most system software:

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

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

Compare-And-Write Compare (using ≥, <, =, or ≠) a global variable on a set of nodes to a local value. If the condition is true on all nodes, then (optionally) assign a new value to a—possibly different—global variable.
The following are some important points about the mechanisms' semantics:

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

2. Xfer-And-Signal and Compare-And-Write are both atomic operations. That is, Xfer-And-Signal either puts data to all nodes in the destination set (which could be a single node) or—in case of a network error—no nodes. The same condition holds for Compare-And-Write when it writes a value to a global variable. Furthermore, if multiple nodes simultaneously initiate Compare-And-Writes with identical parameters except for the value to write, then, when all of the Compare-And-Writes have completed, all nodes will see the same value in the global variable. In other words, Xfer-And-Signal and Compare-And-Write are sequentially consistent operations [16].

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).

We selected Quadrics' QsNET network [21] for our initial implementation because of its raw speed and the wealth of hardware primitives it provides to the bottom layer of our hierarchy: ordered, reliable multicasts; network conditionals (which return True if and only if a condition is True on all nodes); and events that can be waited upon and remotely signaled.

Table 1 shows the expected performance of the mechanisms as a function of the number of nodes n on four high performance networks other than QsNET, namely Gigabit Ethernet, Myrinet, Infiniband and BlueGene/L, based on the best performance reported in the literature. In some of these networks (Ethernet, Myrinet and Infiniband) the BCS mechanisms need to be emulated through a thin software layer, while in the other networks there is a one-to-one mapping with existing hardware mechanisms.

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

### 3 BCS MPI Design

BCS MPI is a novel implementation of MPI that globally schedules the system activities in all the nodes. To do that, a synchronization broadcast message or **global strobe**—implemented with the BCS core primitive Xfer-And-Signal—is sent to all the nodes at regular intervals or **time slices**. In this way, all the system activities are tightly coupled because they occur at the same time in 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 control information schedules the communication requests issued during the previous time slice. After that, all the scheduled communication operations are performed.

The BCS MPI communication protocol is implemented almost entirely in the network interface card (NIC). By doing this, 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 posts a descriptor in a piece of NIC memory that is accessible to a NIC thread. Such a descriptor includes all the communication parameters which are need to complete the operation. The actual communication will be performed by a set of cooperating threads running
In 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 micro-phases within every time slice and its progress is also globally synchronized as described in Section 4.2.

In order to explain how BCS MPI communication primitives work, two possible scenarios for blocking and non-blocking MPI point-to-point primitives are described in Section 3.1 and Section 3.2, respectively.

### 3.1 Blocking Send/Receive Scenario

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_Recv (see Figure 3(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

Process $P_1$ sends a message to process $P_2$ using MPI_Isend and process $P_2$ receives a message from $P_1$ using MPI_Irecv (see Figure 3(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 case, the communication is completely overlapped with the computation with no performance penalty.

### 4 BCS MPI Implementation

In order to validate the framework proposed in Section 3, we developed a fully functional version of BCS MPI for QsNet-based systems which allow us to run complex MPI applications. For quick prototyping and portability, BCS MPI is initially implemented as a user-level communication library, and some typical kernel level functionalities —as process scheduling— are implemented with the help of daemons. As a general rule, we expect this user level implementation to be slower than a kernel level one, though more flexible and easier to use.

The communication library is hierarchically designed on top of a small set of communication/synchronization primitives, the BCS core, and higher level primitives are implemented on top of the BCS core (the BCS API, described in Appendix A). 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 the MPI calls to BCS calls (see Appendix A). Note that scalability is enhanced by tightly coupling the BCS core with the collective primitives provided by the hardware.

BCS MPI is integrated in STORM [9], a scalable, flexible resource management system for clusters, implemented on top of various 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.

In the rest of this section we will describe 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 protocol 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 daemons and a set of threads running in the NIC. The processes and NIC threads that constitute the BCS MPI runtime system are described in Figure 4(b). The Machine Manager (MM), one for the whole machine, resides in the management node. This daemon coordinates the resources of the system issuing regular heartbeats and controls the execution of the parallel jobs. The Strobe Sender (SS) is a NIC thread forked by the MM that implements the global synchronization protocol as described later. The Node Manager (NM) daemon resides in every compute node. This process executes all the commands received from 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 NIC threads forked by the NM in all compute nodes. The SR is the counterpart of the SS in all the compute nodes and coordinates the execution of all the local threads as well. The BS and the BR process the descriptors posted by the application processes every time 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 in regular intervals or time slices. Within every time slice we can identify two main phases and several micro-phases 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 micro-phase. The point-to-point and collective communication operations are scheduled in the message scheduling micro-phase using the information gathered during the previous micro-phase. The message transmission phase performs the point-to-point operations, the barrier and broadcast operations, and the reduce operations, respectively, during its three micro-phases.

In order to implement the global synchronization mechanism, the SS and the SR threads synchronize at the beginning of every micro-phase with a microstrobe implemented with a Xfer-And-Signal. The SS checks whether all the nodes have completed the current micro-phase and, if so, sends a micro-strobe to all the SRs. Then, the SR running on every node wakes up the local NIC thread(s) that must be active in the new micro-phase. The BS and the BR run during the descriptor exchange micro-phase to process the descriptors and during the message scheduling micro-phase to schedule the messages. The DH, the CH and the RH run during the point-to-point micro-phase, the broadcast and barrier micro-phase, and the reduce micro-phase, respectively, to perform all the operations scheduled for execution in the global message scheduling phase.

4.3 Point-to-point

As shown in Figure 3, 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, in time slice $i$ as follows (see Figure 6 for further details).

**Descriptor Exchange Micro-phase** The BS sends each send descriptor posted in time slice $i - 1$ to the BR running on the destination node.

**Message Scheduling Micro-phase** The BR matches the remote send descriptor list against the local receive descriptor list. For each matching pair, the BR builds a match 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 big and cannot be scheduled within a single time slice, the BR splits them 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 about 125 $\mu$seconds.

**Point-to-point Micro-phase** For each match descriptor built in the previous micro-phase by the BR, the DH performs the real data transmission. Note that the intervention of the two application processes involved is not required.
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 micro-strobe to signal all the SRs the beginning of the Descriptor Exchange Micro-phase (DEM) (4) BS sends the descriptor to the BR running on the receiving end (5) SS sends a micro-strobe to signal the beginning of the Message Scheduling Micro-phase (MSM) (6) BR matches the remote send and the local receive descriptors (7) SS sends a micro-strobe to signal the beginning of the Point-to-point Micro-phase (PM) (8) BR schedules the operation for execution (9) DH performs the get (one-sided communication)
4.4 Collective Communication

Every time a user process invokes a collective MPI primitive such as MPI_Barrier, MPI_Broadcast, MPI_Reduce or MPI_Allreduce, it posts a descriptor to the BR thread, which initiates the operation on its behalf, and blocks. The BR preprocesses all the collective descriptors. If all the local processes of an MPI parallel job have invoked the collective primitive, a local flag for that job is set to true. After that, all the collective descriptors, but the ones corresponding to the job master processes, are discarded. All these descriptors posted during time slice $i-1$ will be scheduled, if possible, in time slice $i$ as follows:

Message Scheduling Micro-phase For each collective descriptor corresponding to a job master process, the BR tests if all the application processes of that MPI parallel job have invoked the collective primitive in all nodes. To do that, the BR issues a query broadcast (using Xfer-And-Signal) message that checks the flag for that job in all the nodes. If the flag is equal to true in all nodes, the collective operation is scheduled for execution.

Broadcast and Barrier/Reduce Micro-phase 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, a IEEE compliant library for binary floating-point arithmetic has been used to compute the reduce in the NIC (SoftFloat [31]). Since most applications reduce over a very small number of elements [28], computing the reduce in the NIC is faster than sending the data through the PCI bus to perform the operation in the host.

Figure 7 illustrates the execution of a broadcast operation. The MPI program in the 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 implemented yet.
- A daemon, that belongs to the BCS MPI runtime system schedules the user processes at every time slice.

Since we have little control over the OS scheduler at user level, the NM daemon may not be scheduled on time. This anomaly introduces noise in the system that causes a considerable performance degradation [12]. To eliminate this problem, we are developing a kernel-based implementation of BCS MPI.

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. We are going to evaluate the potential benefits of a kernel-level implementation by executing every experiment with a single application process per node. This reserves a CPU to the NM daemon, and eliminates the problem as we will see in Section 5.

5 Experimental Results

This section presents and analyzes the experimental results. We compare the performance of our user-level implementation of BCS MPI to that of Quadrics MPI for several benchmarks and applications. Quadrics MPI [32] is a production-level implementation for QsNet-based systems, based on MPICH 1.2.4, which is currently used on three systems among the top 5 in the Top500 list [33] at the time of this writing. To validate the implementation of BCS MPI we use a set of synthetic benchmarks, the NAS suite of benchmarks, and several real applications which are representative of the ASCI workload at LANL. In all the experiments, we use the largest configuration which allow us to get the results for both one and two processes per node. In this way, we want to evaluate the potential of a kernel-level implementation of BCS MPI.

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 [20, 24] (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 [20, 23, 25] for the data network, and a 100 Mbit Ethernet NIC for the management network. All the nodes run Red Hat Linux 7.3 with Quadrics kernel modifications and qs-netlibs v1.5.0-0 [32]. All the benchmarks and the applications used throughout this section are compiled with the Intel C/Fortran Compiler v5.0.1 for IA32 using -O3 as optimization option. Finally, a 500µs time slice is used in BCS MPI for all the experiments in this paper.
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 micro-strobe to signal all the SRs the beginning of the Descriptor Exchange Micro-phase (DEM) (6) SS sends a micro-strobe to signal the beginning of the Message Scheduling Micro-phase (MSM) (7) BR checks if all the processes are ready (8) BR schedules the broadcast operation for execution (9) SS sends a micro-strobe to signal the beginning of the Broadcast and Barrier Micro-phase (BBM) (10) CH performs the broadcast.
5.2 Synthetic Benchmarks

Many scientific codes display a bulk-synchronous behavior and can be characterized by a nearest-neighbor communication stencil optionally followed by a global synchronization operation such as barrier, broadcast or reduce [13, 14]. 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. In Figure 8(a) the slowdown of BCS MPI versus Quadrics MPI for different computational granularities is shown. As expected, the slowdown decreases when we increase the computational granularity because the effect of the delay introduced by the barrier synchronization is amortized. The Figure shows that with a single processor the slowdown is less than 5%, with a computational granularity of 10 ms. As expected, with two processors we have a larger slowdown, about 10%, due to the interference of the NM daemon with the user processes. Figure 8(b) shows the slowdown of BCS MPI against Quadrics MPI for different numbers of processes. In this case, the results prove 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 wait for the completion of all the communication operations in a loop. Figure 8(c) shows the slowdown for different computational granularities. As in the previous case, the slowdown decreases when we increase the computational granularity and is below 7.5% and 15%, respectively with one and two processors per node. Figure 8(d) shows that BCS MPI scales well for point-to-point operations too.

5.3 NAS Benchmarks and Applications

The benchmarks and applications used in this section are the NAS Parallel Benchmarks (NPB 2.4) [2, 30, 34] and SAGE (SAIC's Adaptive Grid Eulerian hydrocode) [14]. The NAS Parallel Benchmarks are a set of eight programs designed to help evaluate the performance of parallel supercomputers. The benchmarks, which are derived from computational fluid dynamics (CFD) applications, consist of five kernels and three applications. Since BCS MPI doesn’t support MPI groups yet, we were only able to use four kernels and one application: Integer Sort (IS), Embarrassingly Parallel (EP), Multigrid (MG), Conjugate Gradient (CG) and LU solver (LU). The NPB, but IS which is coded in C, are written in Fortran 77 and use MPI for interprocessor communications. All the benchmarks were compiled for class C workload. SAGE is a multi-dimensional (1D, 2D and 3D), multimaterial, Eulerian, hydrodynamics code with adaptive mesh refinement. SAGE represents a large class of production ASCI applications at Los Alamos. SAGE comes from the Los Alamos National Laboratory Crestone project, whose goal is the investigation of continuous adaptive Eulerian techniques to stockpile stewardship problems. SAGE is characterized by a medium granularity with a nearest-neighbor communication pattern using non-blocking communication operations followed by a reduce operation for each compute step. The code is written in Fortran90 and uses MPI for inter-processor communications. The timing.h input data set, which is considered representative of real workloads, was used in all the experiments. For each case, we compare the runtime of BCS MPI to that of Quadrics MPI, and analyze the results. All the benchmarks and applications were run several times. The final runtime was computed as the average runtime for all the executions. The runtime of NPB and SAGE for both Quadrics MPI and BCS MPI is shown in Figure 9 while the slowdown of BCS MPI related to Quadrics MPI is computed in Table 2. All NPB benchmarks (but IS) and SAGE run efficiently with BCS MPI with a slowdown of less than 5% using one process per node and less than 13% using two processes per node. The NPB are coarse-grained bulk-synchronous applications that, as discussed in Section 5.2, will report under 10% slowdown. IS runs only for ≈ 12s and, consequently, pays the overhead of initializing the BCS MPI runtime sys-
Figure 8: Synthetic Benchmarks

Table 2: Benchmark and Application Slowdown

<table>
<thead>
<tr>
<th>Application</th>
<th>1P/Node</th>
<th>2P/Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>10.10%</td>
<td>10.47%</td>
</tr>
<tr>
<td>EP</td>
<td>0.91%</td>
<td>8.28%</td>
</tr>
<tr>
<td>MG</td>
<td>4.29%</td>
<td>7.51%</td>
</tr>
<tr>
<td>CG</td>
<td>2.43%</td>
<td>8.09%</td>
</tr>
<tr>
<td>LU</td>
<td>3.75%</td>
<td>10.58%</td>
</tr>
<tr>
<td>SAGE</td>
<td>5.02%</td>
<td>12.64%</td>
</tr>
</tbody>
</table>

5.4 Blocking vs. Non-blocking Communications

As explained 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 communications are expected to perform poorly with BCS MPI. The delays introduced by the blocking communications will considerably increase their run time. To alleviate this problem, it is possible to transform the blocking operations into non-blocking ones with minor modifications in the source code in most cases. To illustrate this technique, we have chosen SWEEP3D [13]. SWEEP3D is a time-independent, Cartesian-grid, single-group, discrete ordinates, deterministic, particle transport code taken from the Accelerated Strategic Computing Initiative (ASCI) workload. 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 $\approx 3.5$ ms) and a nearest-neighbor communication stencil with blocking send/receive operations.

In Figure 10(a) we show the run time for SWEEP3D for both Quadrics MPI and BCS MPI for different numbers of processes. The slowdown is more than 400% for one process per node and more than 300% for two processes per node. Each process exchanges four messages with its nearest neighbors every compute step using blocking send/receive operations. This communication pattern along with the fine granularity incurs a very high overhead. Every compute step, the process will block for 1.5 time slices on average for every blocking operation. To minimize the delay, we replaced every 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 performance as shown in Figure 10(b). In this case, the slowdown is under 7% using either one or two processes per node.

6 Conclusions and Future Work

This paper presented an alternative approach to the design of the MPI communication library. Rather than following the beaten track of the user-level communication protocols and optimize the point to point performance, BCS MPI tried to optimize the global state of the machine. 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 there is only a minimal performance penalty in following this route. In one case we had to slightly change the communication pattern of the application, and replace blocking communication with non-blocking one. This optimization led to a manifold improvement of the performance of SWEEP3D.

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 as the one proposed by BCS MPI, it is possible to substantially simplify the implementation of the parallel file system and provide a solid infrastructure to implement system-level fault tolerance.

Our future plans include the implementation of BCS MPI as a Linux kernel module and the design a fault-tolerant version of BCS MPI.

References


Figure 10: SWEEP3D


[34] http://www.nas.nasa.gov/Software/NPB/

7 Appendix A

The BCS communication primitives are listed in Fig. 11. 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 Fig. 12.
<table>
<thead>
<tr>
<th>BCS Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcs_send()</td>
<td>Blocking/non-blocking send</td>
</tr>
<tr>
<td>bcs_recv()</td>
<td>Blocking/non-blocking receive</td>
</tr>
<tr>
<td>bcs_probe()</td>
<td>Blocking/non-blocking test for a matching receive</td>
</tr>
<tr>
<td>bcs_test()</td>
<td>Blocking/non-blocking test for send/receive completion</td>
</tr>
<tr>
<td>bcs_testall()</td>
<td>Blocking/non-blocking test for multiple send/receive completions</td>
</tr>
<tr>
<td>bcs_barrier()</td>
<td>Barrier synchronization</td>
</tr>
<tr>
<td>bcs_bcast()</td>
<td>Broadcast</td>
</tr>
<tr>
<td>bcs_reduce()</td>
<td>Reduce and allreduce</td>
</tr>
<tr>
<td>bcs_scatter()</td>
<td>Vectorial/non-vectorial scatter</td>
</tr>
<tr>
<td>bcs_gather()</td>
<td>Vectorial/non-vectorial gather</td>
</tr>
<tr>
<td>bcs_allgather()</td>
<td>Vectorial/non-vectorial allgather</td>
</tr>
<tr>
<td>bcs_alltoall()</td>
<td>Vectorial/non-vectorial all-to-all</td>
</tr>
</tbody>
</table>

Figure 11: BCS API

<table>
<thead>
<tr>
<th>MPI Primitive</th>
<th>BCS API Primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI.Send()</td>
<td>bcs.send(IN blocking)</td>
</tr>
<tr>
<td>MPI.Send()</td>
<td>bcs.send(IN non-blocking, OUT BCS.Request)</td>
</tr>
<tr>
<td>MPI.Recv()</td>
<td>bcs.recv(IN blocking)</td>
</tr>
<tr>
<td>MPI.Recv()</td>
<td>bcs.recv(IN non-blocking, OUT BCS.Request)</td>
</tr>
<tr>
<td>MPI.Probe()</td>
<td>bcs.probe(IN blocking, IN BCS.Request)</td>
</tr>
<tr>
<td>MPI.Probe()</td>
<td>bcs.probe(IN non-blocking, IN BCS.Request)</td>
</tr>
<tr>
<td>MPI.Test()</td>
<td>bcs.test(IN non-blocking, IN BCS.Request)</td>
</tr>
<tr>
<td>MPI.Wait()</td>
<td>bcs.test(IN blocking, IN BCS.Request)</td>
</tr>
<tr>
<td>MPI.Testall()</td>
<td>bcs.testall(IN non-blocking, IN BCS.Request +)</td>
</tr>
<tr>
<td>MPI.Waitall()</td>
<td>bcs.testall(IN blocking, IN BCS.Request +)</td>
</tr>
<tr>
<td>MPI.Barrier()</td>
<td>bcs.barrier()</td>
</tr>
<tr>
<td>MPI.Reduce()</td>
<td>bcs.reduce(IN non-all)</td>
</tr>
<tr>
<td>MPI.Allreduce()</td>
<td>bcs.reduce(IN all)</td>
</tr>
<tr>
<td>MPI.Scatter()</td>
<td>bcs.scatter(IN non-vectorial)</td>
</tr>
<tr>
<td>MPI.Scatterv()</td>
<td>bcs.scatter(IN vectorial)</td>
</tr>
<tr>
<td>MPI.Gather()</td>
<td>bcs.gather(IN non-vectorial)</td>
</tr>
<tr>
<td>MPI.Gatherv()</td>
<td>bcs.gather(IN vectorial)</td>
</tr>
<tr>
<td>MPI.Allgather()</td>
<td>bcs.allgather(IN non-vectorial)</td>
</tr>
<tr>
<td>MPI.Allgatherv()</td>
<td>bcs.allgather(IN vectorial)</td>
</tr>
<tr>
<td>MPI.Alltoall()</td>
<td>bcs.alltoall(IN non-vectorial)</td>
</tr>
<tr>
<td>MPI.Alltoallv()</td>
<td>bcs.alltoall(IN vectorial)</td>
</tr>
</tbody>
</table>

Figure 12: MPI-BCS Correspondence