ACHIEVING PREDICTABLE AND SCALABLE PERFORMANCE WITH BCS-MPI[*]

JUAN FERNÁNDEZ[†], FABRIZIO PETRINI[‡], AND EITAN FRACHTENBERG[§]

Abstract. Demand for increasingly-higher computing capability is driving a similar growth in compute cluster sizes, soon to be reaching tens of thousands of processors. This growth is not matched however by system software, which has remained largely unchanged from the advent of clusters. The failure of system software to scale and develop in the same rate as the underlying hardware constrains the productivity of these machines by severely limiting their utilization, reliability, and responsiveness. The traditional approach to system software, namely, the use of loosely-coupled independent daemons on each node, is inadequate for the management of large-scale clusters, a problem which is inherently tightly-coupled and requires a high degree of synchronization.

One model for large-scale system software is Buffered Coscheduling (BCS), wherein synchronization and scalability are obtained by means of global scheduling of all system activities and collective network operations. BCS represents a new methodology for the design of system software as a single, parallel program using traditional parallel constructs. As such, system software can be made orders of magnitude more scalable, simple, and easy to debug than the existing distributed solutions. The most important aspect of the BCS model and the overlying system software is the buffering and scheduling of all communication, resulting in highly controllable and deterministic system behavior. This chapter describes in detail the implementation of BCS-MPI, an MPI library designed after this model, and shows that the benefits of determinism need not come at a significant performance cost. Furthermore, BCS-MPI comes with a sophisticated monitoring and debugging subsystem that simplifies the analysis of system and application performance, and is covered in detail in this chapter.

Key words. Cluster computing, system software, buffered coscheduling, MPI, communication protocol, parallel monitoring and debugging, Quadrics, QsNet.

1. Introduction.

Clusters have become the most successful player in the high-performance computing arena in the last decade. At the time of this writing, eight of the ten fastest systems in the Top500 list [31] are clusters – typically assembled from commodity off-the-shelf (COTS) components – and the ever-increasing demand for computing capability is driving the construction of ever-larger clusters. Experience with large-scale clusters, such as ASCI Q at Los Alamos National Laboratory, has shown that managing such systems is a very time- and resource-consuming task. This difficulty owes to the fact that system software[1] has neither evolved nor scaled accordingly to cluster sizes, inadvertently making these systems more complex, less reliable, and less efficient.

For a workstation or symmetric multiprocessor (SMP) the system software is just a traditional microprocessor operating system (e.g. Linux) but for a cluster there are additional components. These include job launcher/scheduler (e.g. Cplant [2], SLURM [10], RMS [24, 25]), communication library (e.g. OpenMP, MPI), parallel file system (e.g. Lustre [29]), and many others.

The reason for the poor performance of the system software is that it had not been important to make it efficient in the past. On smaller clusters, inefficient or non-

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

[†] Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, 30071 Murcia, SPAIN (juanf@um.es).

[‡] Applied Computer Science Group, MS K7-90, Pacific Northwest National Laboratory, Richland, WA 99352, USA (fabrizio.petrini@pnl.org).

[§] CCS-3 Modeling, Algorithms & Informatics, Los Alamos National Laboratory, Los Alamos, NM 87545, USA (eitanf@lanl.gov)

[1] We refer to *system software* as all software running on a machine other than user applications.

scalable system software has little impact on overall performance, so it is reasonable to focus more attention on application performance. However, on a large-scale cluster, with size in excess of 10,000 processors (e.g. the NASA Columbia supercomputer [31]), system software becomes a major factor to account for. Even a small overhead in a node can cause a major performance problem at scale [20]. The problem is exacerbated when trying to amortize the performance lost to slow or non-scalable system software functions by calling them as infrequently as possible, resulting in degraded response time and hindering interactive jobs.

The root of the problem is the use of largely independent, loosely-coupled compute nodes for the solution of problems that are inherently tightly coupled. On the one hand, the commodity hardware components used to build clusters were conceived for loosely-coupled environments. On the other hand, the local operating system (OS) running on the compute nodes lacks global awareness of parallel software. In this scenario, system software positions between the local OSes and the parallel applications, and acts like *glue* to put all the pieces together.

The current widely-used methodology is to design system software components separately, in order to have a modular design and allow different developers to work concurrently while limiting cross-dependencies. Most system software components have many elements in common which may lead to a redundancy of functionality (e.g. the same communication mechanisms are implemented by different system software components). Furthermore, the lack of a single source of system services is often detrimental to performance (e.g. most clusters cannot guarantee Quality of Service – QoS – for user-level traffic and system-level traffic in the same network).

To deal with system software complexity, we have proposed a new methodology for the design of parallel system software based on two cornerstones: (1) global control and coordination of all system activities, and (2) a very small set of efficient and scalable network-supported primitives [5, 6]. Basically, this approach tries to better integrate all the nodes by leveraging modern interconnects. We use core primitives that represent a least common denominator of most system software components, and thus constitute the backbone to integrate all nodes with a single, global OS.

This methodology has already been applied to demonstrate the efficient implementation of several system software tasks. In particular, the STORM resource manager provides scalable, high-performance, and lightweight job launching and scheduling [7]. In this chapter, we show how this methodology is applied to building a message-passing communication library, BCS-MPI. We describe and analyze the performance of BCS-MPI, which imposes a global communication model where communication is tightly controlled at a fine granularity. The system as a whole behaves as a bulk-synchronous program (BSP), where computation and communication are divided into distinct, timed phases. In this model, called Buffered Coscheduling (BCS) [4, 16], all the user and system-level communication is buffered and controlled. The entire cluster marches to the beat of a global strobe that is issued every few hundreds of microseconds. This is reminiscent of the SIMD model, with the exception that the granularity is expressed in time intervals rather than instructions. In the intervals between strobes, or *time slices*, newly-issued communication calls are buffered until the next time slice. At every strobe, nodes exchange information on pending communication, so that every node has complete knowledge of the required incoming and outgoing communication for the next time slice. The nodes then proceed to globally schedule those communications that will actually be carried out during the time slice, and proceed to execute them. The advantage of this model is that all the communication is controlled and can be maintained in a known state at every given time slice, so that problems arising from congestion or hot spots can be avoided. Moreover, this BSP-like execution model not only facilitates monitoring and debugging of parallel jobs but also paves the way to achieve deterministic replaying of parallel applications [9] and transparent fault tolerance [26]. Finally, these constraints impose little or no overhead for scientific applications, while obtaining the advantages of a more deterministic, controllable machine.

2. Core Primitives and Mechanisms. BCS-MPI relies on an abstract common layer of high-performance, scalable 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 (PUT) 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. By global memory we refer to data at the same virtual address on all nodes. Depending on implementation, global data may reside in main or network-interface memory.

Test-Event 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.

Note that Xfer-And-Signal and Compare-And-Write are both atomic operations. That is, Xfer-And-Signal either *put*s 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 [13]. 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. These 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 (though it may be advantageous to do so).

2.1. Implementation and Scalability. The three primitives presented above were originally designed to improve the communication performance of user applications. We posit that system software should be regarded as a parallel application as well, and as such, make use of these primitives too.

Hardware support for multicast messages sent with Xfer-And-Signal is needed to guarantee scalability for large-scale systems. Software approaches, while feasible for small clusters, do not scale well to thousands of nodes. In our case, QsNet provides Put/Get operations as primitive hardware operations, making the implementation of Xfer-And-Signal 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 provides a global query operation that allows direct implementation of Compare-And-Write. Such functionality is provided by several state-of-the-art networks such as QsNet and Infiniband and has been extensively studied

Table 2.1
Measured/expected performance of the core mechanisms as a function of the number of nodes n.

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

[14, 17]. Moreover, the results presented in [15, 18] show that the implementation of these primitives on QsNet scales to thousands of nodes.

2.2. Portability. Quadrics' QsNet [17], 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. Table 2.1 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 (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 [1]. 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 [7] we demonstrated their utility for resource management tasks, this chapter focuses on their usage as a basis for a user-level communication library, BCS-MPI.

3. BCS-MPI Architecture. 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 Xfer-And-Signal, is sent to all nodes at regular intervals or *time slices*. Consequently, 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 Xfer-And-Signal and Test-Event, provides information to schedule the communication requests issued during the previous time slice. Subsequently, all the scheduled communication operations are performed using Xfer-And-Signal and Test-Event.

The BCS-MPI communication protocol is executed almost entirely in the network interface card (NIC). This offloading enables BCS-MPI to overlap the communication with the computation executed on the host CPUs [3]. 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 QsNet 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 demonstrate how BCS-MPI communication works, 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 P_1 using MPI_Recv (see Figure 3.1):

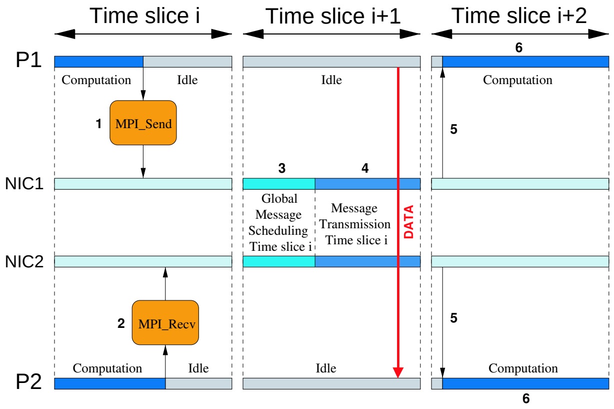

Fig. 3.1. *Blocking MPI_Send/MPI_Receive scenario.*

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 performance penalty can be alleviated by using non-blocking communication (see Section 6.3) 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 3.2):

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 $i + 1$).

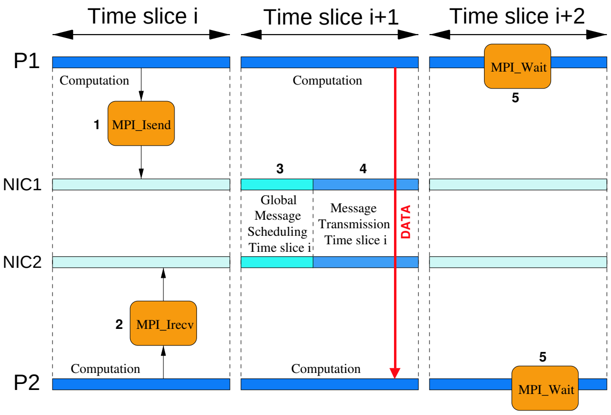

FIG. 3.2. *Non-blocking MPI_Isend/MPI_Ireceive scenario.*

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 was initially implemented as a user-level communication library, and some typical kernel level functionalities such as process scheduling are implemented with the help of dæmons. This user-level implementation is expected to be somewhat 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 4.1

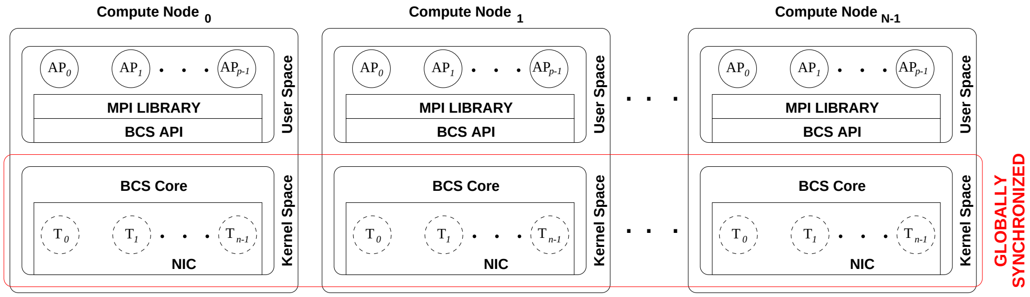

FIG. 4.1. *BCS-MPI overview.*

The communication library is hierarchically designed on top of a small set of communication/synchronization primitives, the BCS core primitives (Figure 4.2), while higher-level primitives (see the BCS API on Table 4.1) 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 Table 4.2). Note that scalability is enhanced by tightly coupling the BCS core primitives with the collective primitives provided at hardware level by the interconnection network.

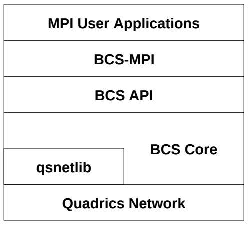

Fig. 4.2. *Library hierarchy.*

Table 4.1
BCS API.

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

BCS-MPI is integrated with STORM [7], a scalable, flexible resource management system for clusters, running on i386-, IA64- and Alpha-based architectures. STORM exploits the BCS core primitives to offer high-performance job launching and resource management. We now turn to describe how the BCS-MPI infrastructure is used with actual MPI applications.

4.1. Processes and Threads. In the current 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 4.3. 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 launched 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.

Table 4.2
MPI BCS-API correspondence.

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_Bcast()	bcs_bcast()
MPI_Reduce()	bcs_reduce(IN non-all)
MPI_Allreduce()	bcs_reduce(IN all)
MPI_Scatter()	bcs_scatter(IN non-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)

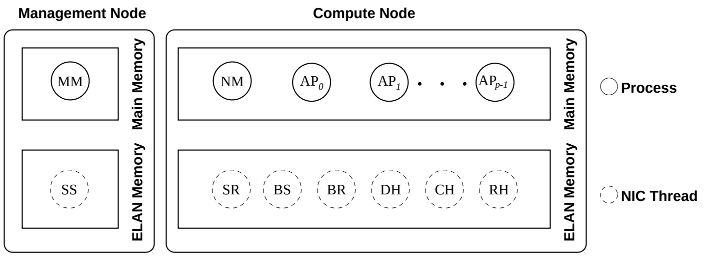

Fig. 4.3. *Processes and threads.*

4.2. Global Synchronization Protocol. The BCS-MPI runtime system globally schedules all the computation, communication and synchronization activities of the MPI jobs every time slice. Each time slice is divided into two main phases and several microphases, as shown in Figure 4.4. 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 BCS-MPI runtime system must globally coordinate the execution of the time slices and their microphases in all the nodes. To this end, the SS and the SR threads synchronize at the beginning of every microphase with a microstrobe implemented using Xfer-And-Signal. The SS verifies that 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 subsequently 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 microphase 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.

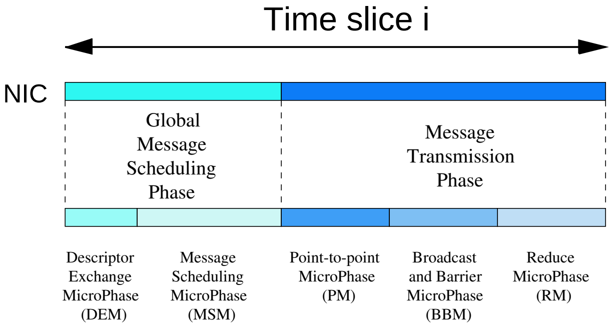
Time slice i
NIC
Global Message Scheduling Phase
Message Transmission Phase
Descriptor Exchange MicroPhase (DEM)
Message Scheduling MicroPhase (MSM)
Point-to-point MicroPhase (PM)
Broadcast and Barrier MicroPhase (BBM)
Reduce MicroPhase (RM)

Fig. 4.4. *Global synchronization protocol.*

4.3. Point-to-point. As shown in Figures 3.1 and 3.2, 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 4.5 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.

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.

Note that this scheme is based on a *rendezvous* protocol implemented in the NIC. This approach eliminates the need for intermediate buffers and minimizes the main CPU overhead due to communications [3].

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-

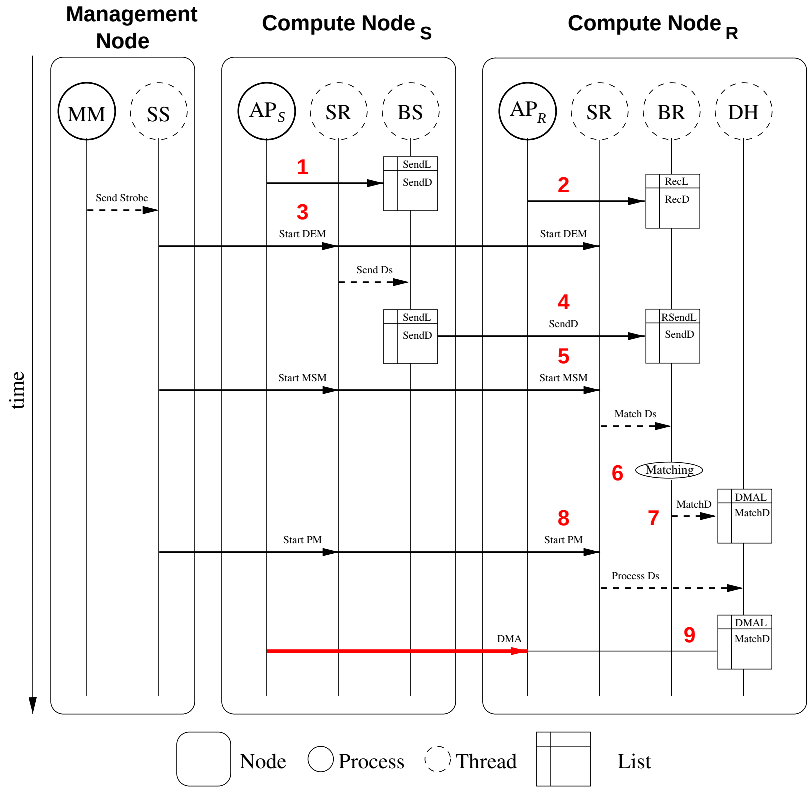

FIG. 4.5. *Send/receive scenario: (1) The sender process posts a descriptor to the BS (2) The receiver process posts a descriptor to the BR (3) SS sends a microstrobe to signal all the SRs the beginning of the Descriptor Exchange Microphase (DEM) (4) BS sends the descriptor to the BR running on the receiving 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).*

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 carried out 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 [28]). Since most applications reduce over a very small number of elements [27], computing the reduce in the NIC is faster than sending the data through the PCI bus to perform the operation in the host [19].

Figure 4.6 illustrates the execution of a broadcast operation. The MPI program in this example is composed of four processes running on two different nodes.

5. Monitoring and Debugging Parallel Software with BCS-MPI. As described in Section 4, the BCS-MPI runtime system globally schedules all the computation, communication and synchronization activities of the MPI jobs in a BSP-like fashion. In this way, BCS-MPI facilitates monitoring and debugging of parallel software. To this end, BCS-MPI incorporates a monitoring and debugging module, namely *Monitoring and Debugging System* (MDS). This module allows to monitor and debug, using *a posteriori* data analysis (see Sections 5.3 and 5.4), not only MPI applications but also the BCS-MPI runtime system itself. This NIC-based monitoring ability has a twofold importance. First, profiling the BCS-MPI API can provide statistics about process scheduling and communication primitives. Second, profiling the threads running in the NIC (see Section 4.1) can produce meaningful statistics for both the communication pattern of parallel applications and the behavior of the runtime system itself.

In this section, we show the functionality and implementation of the MDS. Furthermore, we describe how to use the MDS to monitor and debug the runtime system as well as real applications. The MDS is logically divided into two main components, the *Main MDS* (MMDS) and the *Elan MDS* (EMDS). These modules are described and analyzed in Sections 5.1 and 5.2, respectively. Both modules can be independently enabled and disabled without compiling or linking the code by just setting an environment variable to an specific value. Finally, the performance implications on the use of the MDS are studied in Section 6.4. To better understand these results, Sections 5.1.1 and 5.2.1 give some insight about the way the MDS collects data.

5.1. Main MDS (MMDS). The MMDS' main role is to produce statistics on process scheduling as well as communication primitives usage, for any running MPI parallel application on a per-process basis. To provide this capability, the MMDS can extract distribution data for computation granularity and communication overhead, in addition to a summary of the usage of the BCS-MPI primitives (including the number of invocations, and the minimum, maximum and average latency). Furthermore, this module allows to select specific primitives so that the corresponding latency and size (if applicable) distributions can be generated as well. The user retains selective control over each process and metric to be measured, as well as over the latency and size resolution.

5.1.1. Implementation. The MMDS composes part of the BCS-MPI API (see Section 4) and as such is executed by the application processes running in the main processor (see Section 4.1).

To assemble the computation granularity and communication overhead distributions, every application process uses four data structures: a computation granularity counter, a communication overhead counter, a computation granularity array, and a communication overhead array. Every time a blocking BCS primitive is invoked, the calling process updates the computation granularity array using the computation granularity counter, obtains a new time stamp for the communication overhead counter, and blocks. Once the application process is awoken, the process obtains a new time stamp for the computation granularity counter and updates the communication overhead array using the communication overhead counter. Thus, the sum of all the points belonging to both distributions is approximately equal to the total run time of the application.

A simple scenario, which illustrates how the MMDS works, is shown in Figure 5.1. This scenario comprises two processes, P_1 and P_2, running a ping-pong test for a single iteration. At time t_0, process P_2 invokes MPI_Recv which, in turn, calls bcs_recv (see Table 4.2). This function sets the communication overhead counter (the counter is assigned t_0). Once the process P_2 is awakened, at time t_1, a new time stamp is obtained to compute the current communication overhead point, t_1 - t_0 (the value of the communication overhead counter), and reset the computation granularity counter which is assigned t_1. Next time P_2 invokes a blocking primitive, MPI_Send in this case, a time stamp is again used to compute the current computation granularity point, t_2 - t_1 (the value of the computation granularity counter), and reset the communication overhead as well.

To assemble the latency and size (if applicable) distributions for every single primitive, we follow the very same approach as before. However, in this case, the counters and arrays are always used regardless of whether the primitive is blocking or not.

Every process dumps each individual distribution to a different file at the end of its execution. Consequently, the impact of the MDS on the execution of the MPI parallel job is minimal. On the one hand, the overhead incurred by the MMDS is negligible (as shown in Section 6). On the other hand, the amount of memory required to store the MDS data structures depends on the desired resolution, that is, the finer the resolution, the higher the MDS memory requirements will be. However, the memory required by the MDDS is no more than a few megabytes in the worst case which is typically easy to accommodate in contemporary systems.

5.2. Elan MDS (EMDS). The EMDS monitors the activity of the threads described in Section 4.1. This module provides both global statistics on the global synchronization protocol and local ones regarding process and communication scheduling on a per-node basis. Tables 5.1 and 5.2 summarize the global and local metrics, respectively. LTRi refers to the time to complete the execution of routine i where routine i is some internal routine related to the resource scheduling process performed by the NIC. This data facilitates the profiling of how communication time is spent, allowing the optimization and tuning of the runtime system. Given that the global synchronization protocol splits time into time slices, all of them are expressed as a function of a time slice number. Note that this approach is very powerful since we can track the progress of specific communication operations through different nodes using discrete events. In all cases, nodes and metrics can be selectively enabled and disabled, and it is possible to adjust the measurement resolution.

Figure 5.2 illustrates the meaning of EMDS statistics. The data describe both the global and local EMDS statistics. The only difference is the NIC thread which gathers the data. The Strobe Sender is in charge of the global EMDS statistics while the local EMDS statistics are obtained by the Strobe Receiver running on each node. For example, the GTDEM value, gathered by the Strobe Sender, represents the time to complete the DEM microphase in all nodes. In the meantime, the Strobe Receiver running on every node obtains a similar figure, namely LTDEM, which corresponds

Table 5.1
Global EMDS Statistics.

Metric	Meaning
GETTS	Global Elapsed Time from the previous Time Slice
GTDEM	Global Time to complete the DEM microphase
GTMSM	Global Time to complete the MSM microphase
GTTS	Global Time to complete the Time Slice
USRTS	Unsuccessful Synchronization Retries before the previous Time Slice is over
USRDEM	Unsuccessful Synchronization Retries before the DEM microphase is over
USRMSM	Unsuccessful Synchronization Retries before the MSM microphase is over

Table 5.2
Local EMDS Statistics.

Metric	Meaning
LETTS	Local Elapsed Time from the previous Time Slice
LTDEM	Local Time to complete the DEM microphase
LMSM	Local Time to complete the MSM microphase
LTTS	Local Time to complete the Time Slice
NP2PDEM	Number of point-to-point descriptors processed in the DEM microphase
NP2PMSM	Number of scheduled point-to-point operations in the MSM microphase
NCOLLDEM	Number of collectives processed in the DEM microphase
NCOLLMSM	Number of scheduled collectives in the MSM microphase
BPTS	Blocked processes during the current Time Slice
LTRi	Local Time to complete the execution of Routine i

to the time to complete the DEM microphase on that very same node. Given that all nodes are synchronized between microphases, the largest value for a particular figure at any node constitutes a lower bound for the corresponding global figure. Finally, we note that the GTTS metric may be shorter than the time slice value imposed to the system if the MSM microphase prematurely ends because no communications are performed. Moreover, the GETTS metric allows the verification that the Strobe Sender signals all nodes at regular intervals equal to the chosen time slice value without measurable delays.

5.2.1. Implementation. The EMDS is integrated into the BCS core (see Section 4) and as such is executed by the *Elan Thread Processor* [17, 21]. Therefore, all the data structures used by the EMDS need to be stored in Elan3 memory [21], unlike with the MMDS. PCI bus transactions could introduce unpredictable delays which negate the BCS-MPI philosophy that tries to implement a deterministic system. The *global and local EMDS statistics* are expressed in terms of the time slice number. Therefore, unlike the MMDS, the memory requirements grow linearly with the time slice number. Since the amount of memory in the Elan3 NICs used here is limited to 64 MB, the EMDS must be carefully designed to fit into Elan3 memory, along with the thread code, and avoid overflow situations. To this end, both the global and the local EMDS statistics can only be active during a period of time equivalent to ten thousand time slices, e.g. 5 seconds for a 500 μs time slice. The mechanism to keep the statistics up-to-date is similar to the one explained for the MMDS statistics.

Every node dumps both the local and the global EMDS statistics to a file once the BCS-MPI runtime system is shut down. Even though the overhead incurred while updating the EMDS data structures is quite low, it is higher than in the MMDS case, due to the small TLB and cache sizes in the Elan3. This small size entails that access to the EMDS data structures may pollute either or both tables.

Finally, it is worth noting that time measurements, for both the MMDS and the EMDS, are highly accurate. All the counters and the individual distribution points are 64-bit values so that the overflow of any of them is not possible. Moreover, to get the time stamps, the elan3_clock function [23] is used. This function uses the Elan hardware clock in order to provide the current time, since some arbitrary time in the past, expressed in nanoseconds as a 64-bit value.

5.3. Monitoring and Debugging the BCS-MPI Runtime System. In this section we show how to use the MDS to monitor and debug the behavior of the BCS-MPI runtime system itself. To do that, let's assume a simple MPI benchmark which barrier synchronizes every 1.9 ms. Given the BCS-MPI execution model explained in Section 4.2, if the BCS-MPI runtime system globally synchronizes every 250 μs, the correct execution of the benchmark implies that:

1. all the nodes in the system synchronize every 250 μs,
2. every process invokes MPI_Barrier every nine slices,
3. the BCS-MPI runtime system schedules a barrier every nine time slices.

To verify these assumptions, we enabled the MDS and ran this experiment for 10,000 iterations. We activated the EMDS from time slice 7500 to time slice 12500. Figure 5.3 shows the GETTS as a function of the time slice number. The length of the time slices varies between 250 and 270 μs, guaranteeing that all nodes are synchronized at regular intervals. Figures 5.4 and 5.5 show NDEM and NCOLL as a function of the time slice number respectively for a randomly chosen node. As expected, these results satisfy assumptions 1 through 3 with negligible deviations from the expected values. This suggests that an small set of benchmarks, like the one used in this section, would constitute a powerful tool to test and debug the BCS-MPI runtime system. In these simples cases, the BCS-MPI execution model enables performance prediction in order to validate the dynamic behavior of the runtime system. Moreover, the BCS-MPI implementation reduces non-determinism since most of the tasks are performed by the NIC which is immune to the effect of computational noise [11, 20].

5.4. Monitoring and Debugging Parallel MPI Applications. In the previous section, we showed how to take advantage of the MDS to monitor and debug the BCS-MPI runtime system. The same approach can be used to monitor and debug actual applications. To demonstrate this, we use SAGE, a hydrodynamics code widely used at LANL (for further details see Section 6), to illustrate how to use the MDS to monitor and debug real applications.

In Table 5.3, the summary generated by the MMDS when executing SAGE is shown. This summary provides general information about all the BCS-MPI primitives used by SAGE during its execution. By using these data, it would be possible to identify either bottlenecks or hot-spots in the communication pattern of the application. In such cases, a top-down approach, like the one described in [20], must be used until primitive which causes the functional or performance bug is identified. After that, the MDS can be used to get further details about the problematic primitive by generating a latency distribution and a size distribution, if applicable.

Finally, the actual runtime for this run was 115.023 seconds while the total computation time plus the total communication time is 115.027 seconds. This gap represents an error of less than 0.01%, indicating a high accuracy and a low level of intrusion of the MDS.

Table 5.3
SAGE statistics with the timing_h input deck.

Primitive	Min(ms)	Max(ms)	Total(ms)	Count	Average(ms)
MPI_Isend	0.588	16.576	21026.554	4396	4.783
MPI_Recv	0.415	0.699	10.469	19	0.551
MPI_Irecv	0.736	16.644	24280.771	5617	4.323
MPI_Probe	0.123	0.816	36.923	136	0.271
MPI_Waitall	0.071	13.688	2481.633	2140	1.160
MPI_Barrier	0.097	0.639	0.736	2	0.368
MPI_Bcast	0.355	178.988	348.312	312	1.116
MPI_Allreduce	0.366	24.753	18906.279	7025	2.691
MPI_Allgather	1.796	41.121	500.593	45	11.124
MPI_Alltoall	14.373	27.621	645.445	34	18.984
Comp Granularity	0.001	2515.857	92440.640	9771	9.461
Comm Overhead	0.068	178.953	22587.242	9770	2.312

6. Performance Evaluation. To evaluate and validate our implementation of BCS-MPI, we compare the performance of our user-level implementation of BCS-MPI to that of Quadrics MPI using two scientific applications SWEEP3D [8] and SAGE [12]. Quadrics MPI [30] is a production-level implementation for QsNet-based systems, based on MPICH 1.2.4. Quadrics MPI is used by three of the ten fastest systems in the Top500 list [31], at the time of this writing.

6.1. 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 [17, 22] (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 [17, 21, 23] for the data network, and a $100 Mbit$ 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 [30]. 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.

6.2. Application Performance. SAGE is a multidimensional (1D, 2D and 3D), multi-material, Eulerian, hydrodynamics code with adaptive mesh refinement. 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 runtimes of SAGE for both Quadrics MPI and BCS-MPI are shown in Figure 6.1. 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.

6.3. Blocking vs. Non-blocking Communications. As stated in Section 6.2, bulk-synchronous applications with non-blocking or infrequent blocking communications run efficiently with BCS-MPI. However, fine-grained applications that use blocking communications or applications that group blocking communications are expected

Table 6.1
Overhead incurred by the MMDS and the EMDS while running SAGE.

Input deck	MDS Disabled	MMDS Enabled		EMDS Enabled	
	Runtime	Runtime	Overhead	Runtime	Overhead
timing_h.input	114.604s	115.023s	0.36%	116.102s	1.31%
timing_c.input	193.202s	193.345s	0.07%	193.419s	0.11%

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.

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. SWEEP3D is characterized by a fine granularity (each compute step takes ≈ 3.5 ms) and a nearest-neighbor communication stencil with blocking send/receive operations. Figure 6.2 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 6.3. 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.

6.4. MDS Overhead. As shown in Section 5, the MDS can be a powerful tool for monitoring and debugging MPI applications, as well as the runtime system itself. As with any similar tool, the MDS incurs an operational overhead. In this section, we study the overhead incurred by the MMDS and the EMDS while running scientific applications. Table 6.1 shows the runtime of SAGE for two different input decks, *timing_c.input* and *timing_h.input*. The MMDS overhead is less than 0.5% for both input decks. The EMDS overhead is only slightly higher than in the MMDS case due to the small TLB and cache sizes in the Elan3 (see Section 5.2.1).

7. Concluding Remarks. We have presented BCS-MPI, a user-level communication library designed according to the Buffered Coscheduling methodology. BCS-MPI tries to achieve scalable performance through global scheduling of communication by logically orchestrating the activities in a large-scale system in deterministically reproducible, global steps.

An important contribution of this chapter is the detailed description of the communication protocols that enforce global coordination. These protocols are executed in the network interface card, overlapping computation with communication.

The global coordination and the parallel execution of the BCS-MPI runtime in the network interface card allowed us to develop an innovative monitoring and debugging system (MDS) that can profile with extreme accuracy the execution of an MPI program and of the run-time software itself without any measurable overhead.

The experimental results have shown that the performance of BCS-MPI is comparable to the production-level MPI for most applications but with a much simpler design. The performance of some applications, as SWEEP3D, can be improved by slightly 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 MPI, thanks to BCS-MPI's low overhead in the compute nodes, the overlapped execution of the communication protocols, and the accurate profiling capabilities provided by the MDS that allow sophisticated optimizations of both system software and user applications.

REFERENCES

[1] G. Almási, C. Archer, J. G. Castaños, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, N. Smeds, B. Steinmacher-burow, W. Gropp, and B. Toonen, *Implementing MPI on the BlueGene/L Supercomputer*, in Euro-Par 2004 Parallel Processing, vol. 3149/2004 of Lecture Notes in Computer Science, Pisa, Italy, Sept. 2004, Springer-Verlag.

[2] R. Brightwell and L. A. Fisk, *Scalable Parallel Application Launch on Cplant*, in Proceedings of IEEE/ACM Conference on SuperComputing, Denver, CO (USA), Nov. 2001.

[3] R. Brightwell, W. Lawry, A. B. Maccabe, and C. Wilson, *Improving Processor Availability in the MPI Implementation for the ASCI/Red Supercomputer*, in Proceedings of IEEE Conference on Local Computer Networks, Tampa, FL (USA), Nov. 2002.

[4] J. Fernández, E. Frachtenberg, and F. Petrini, *BCS-MPI: A New Approach in the System Software Design for Large-Scale Parallel Computers*, in Proceedings of IEEE/ACM Conference on SuperComputing, Phoenix, AZ (USA), Nov. 2003.

[5] J. Fernández, E. Frachtenberg, F. Petrini, K. Davis, and J. C. Sancho, *Architectural Support for System Software on Large-Scale Clusters*, in Proceedings of International Conference on Parallel Processing, Montreal, Canada, Aug. 2004.

[6] E. Frachtenberg, K. Davis, F. Petrini, J. Fernández, and J. C. Sancho, *Designing Parallel Operating Systems via Parallel Programming*, in Euro-Par 2004 Parallel Processing, vol. 3149/2004 of Lecture Notes in Computer Science, Pisa, Italy, Sept. 2004, Springer-Verlag.

[7] E. Frachtenberg, F. Petrini, J. Fernández, S. Pakin, and S. Coll, *STORM: Lightning-Fast Resource Management*, in Proceedings of IEEE/ACM Conference on SuperComputing, Baltimore, MD (USA), Nov. 2002.

[8] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme, *A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs*, in Proceedings of International Conference on Parallel Processing, Toronto, Canada, Aug. 2000.

[9] C.-H. Hong, G. On, B.-S. Lee, and D. H. Chi, *Replay for Debugging MPI Parallel Programs*, in Proceedings of the 2^{th} MPI Developer's Conference, Notre Dame, IN (USA), July 1996.

[10] M. Jette and M. Grondona, *SLURM: Simple Utility for Resource Management*, in ClusterWorld Conference and Expo, San José, CA (USA), June 2003.

[11] T. Jones, W. Tuel, and B. Maskell, *Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System*, in Proceedings of IEEE/ACM Conference on SuperComputing, Phoenix, AZ (USA), Nov. 2003.

[12] D. J. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings, *Predictive Performance and Scalability Modeling of Large-Scale Applications*, in Proceedings of ACM/IEEE Conference on SuperComputing, Denver, CO (USA), Nov. 2001.

[13] L. Lamport, *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, C-28 (1979), pp. 690–691.

[14] J. Liu, A. R. Mamidala, and D. K. Panda, *Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support*, in Proceedings of International Conference on Parallel and Distributed Processing, Santa Fe, NM (USA), Apr. 2004.

[15] A. Moody, J. Fernández, F. Petrini, and D. K. Panda, *Scalable NIC-Based Reduction*

on Large-Scale Clusters, in Proceedings of IEEE/ACM Conference on SuperComputing, Phoenix, AZ (USA), Nov. 2003.

[16] F. Petrini and W. chun Feng, *Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems*, in Proceedings of International Conference on Parallel and Distributed Systems, Cancun, Mexico, 2000.

[17] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg, *The Quadrics Network: High-Performance Clustering Technology*, IEEE Micro, 22 (2002), pp. 46–57.

[18] F. Petrini, J. Fernández, E. Frachtenberg, and S. Coll, *Scalable Collective Communication on the ASCI Q Machine*, in Proceedings of Symposium on High Performance Interconnects, Stanford, CA (USA), Aug. 2003.

[19] F. Petrini, J. Fernández, A. Moody, E. Frachtenberg, and D. K. Panda, *NIC-based Reduction Algorithms for Large-Scale Clusters*, International Journal of High-Performance Computing and Networking (to appear), (2005).

[20] F. Petrini, D. J. Kerbyson, and S. Pakin, *The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8192 Processors of ASCI Q*, in Proceedings of ACM/IEEE Conference on SuperComputing, Phoenix, AZ (USA), Nov. 2003.

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

[22] ———, *Elite Reference Manual*, Quadrics Supercomputers World Ltd., 1999.

[23] ———, *Elan Programming Manual*, Quadrics Supercomputers World Ltd., Nov. 2003.

[24] ———, *RMS Reference Manual*, Quadrics Supercomputers World Ltd., Nov. 2003.

[25] ———, *RMS User Manual*, Quadrics Supercomputers World Ltd., Nov. 2003.

[26] J. C. Sancho, F. Petrini, G. Johnson, J. Fernández, and E. Frachtenberg, *On the Feasibility of Incremental Checkpointing for Scientific Computing*, in Proceedings of Parallel and Distributed Processing Symposium, Santa Fe, NM (USA), Apr. 2004.

[27] J. S. Vetter and F. Mueller, *Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures*, in Proceedings of International Parallel and Distributed Processing Symposium, Fort Lauderdale, FL (USA), Apr. 2002.

[28] www.jhauser.us/arithmetic, *SoftFloat: Software Implementation of IEEE-754.*

[29] www.lustre.org, *Cluster File Systems, Inc.*

[30] www.quadrics.com, *Quadrics Supercomputers World Ltd.*

[31] www.top500.org, *Top500 Supercomputing Sites*, 2004.

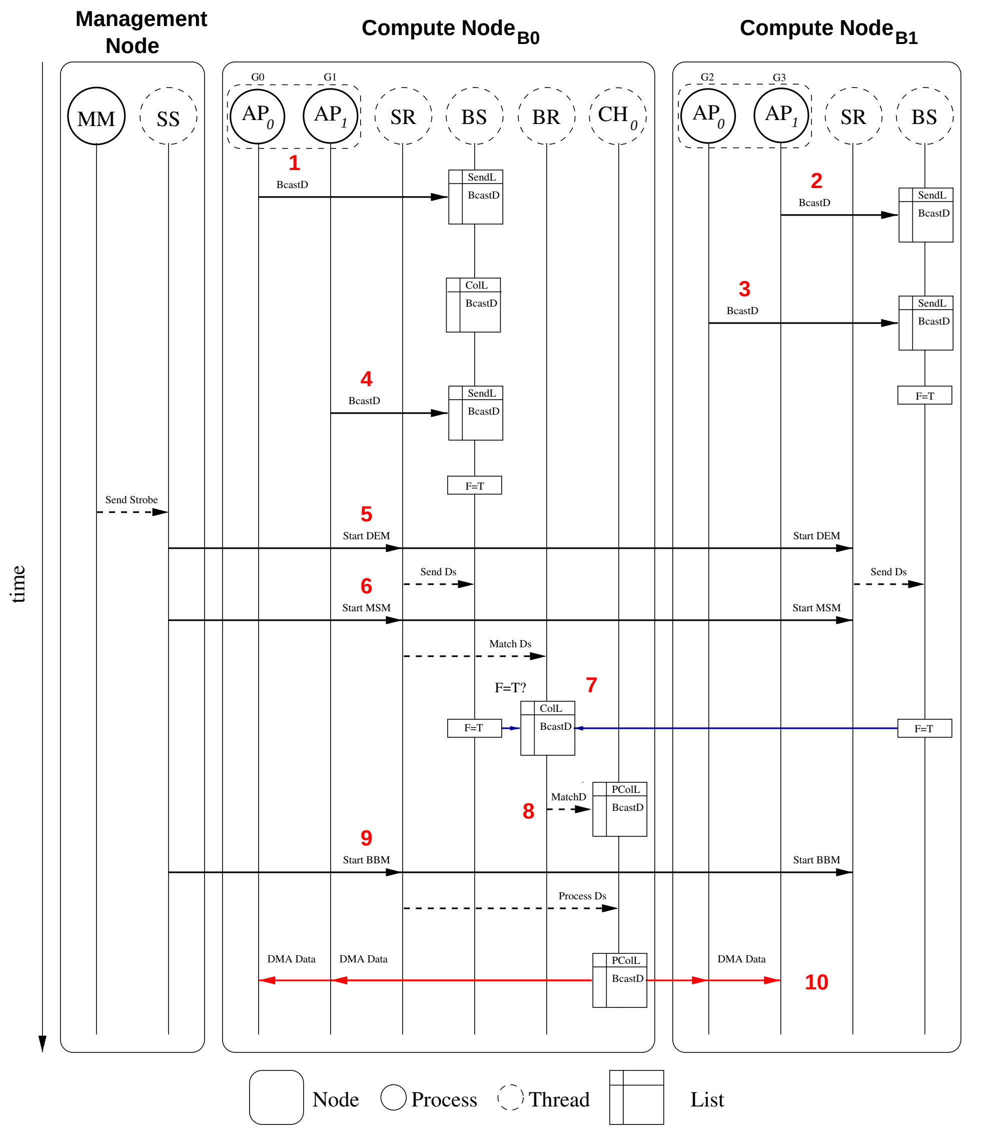

FIG. 4.6. *Broadcast Scenario: (1) Application Process (AP) G_0 posts a descriptor to the local BS. G_0 is the master process and its descriptor is copied to the Collective List (2) G_3 posts a descriptor to the local BS. The descriptor is processed and discarded (3) G_2 posts a descriptor to the local 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.*

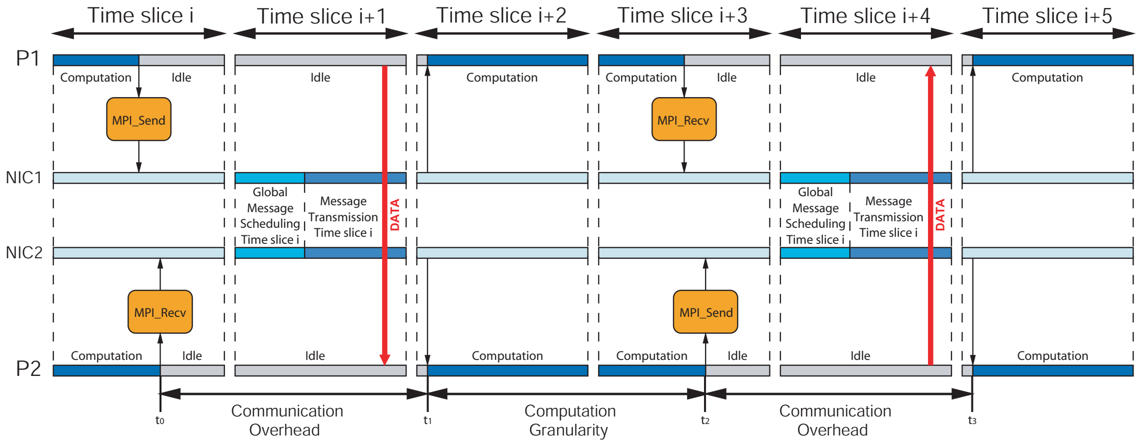

Fig. 5.1. *MMDS Time Statistics Implementation.*

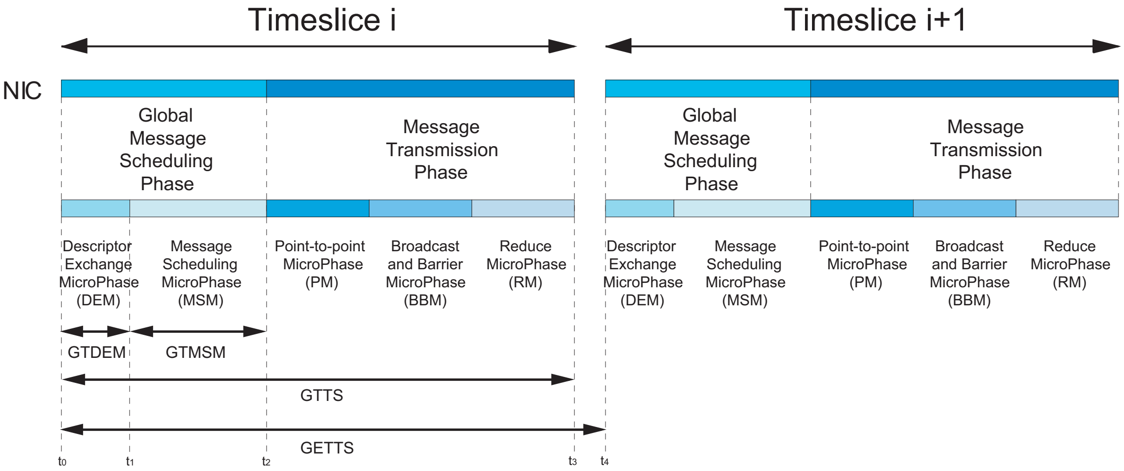
Timeslice i
Timeslice i+1
NIC
Global Message Scheduling Phase
Message Transmission Phase
Descriptor Exchange MicroPhase (DEM)
Message Scheduling MicroPhase (MSM)
Point-to-point MicroPhase (PM)
Broadcast and Barrier MicroPhase (BBM)
Reduce MicroPhase (RM)
GTDEM
GTMSM
GTTS
GETTS
t_0
t_1
t_2
t_3
t_4

Fig. 5.2. *EMDS Time Statistics Implementation.*

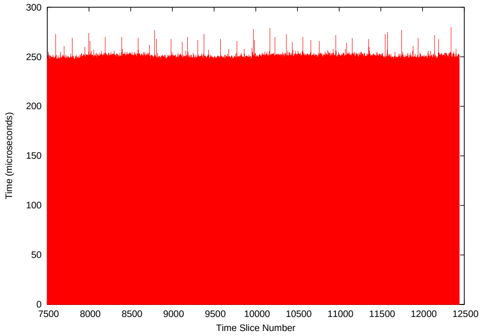

Fig. 5.3. *Global Elapsed Time from previous Time Slice.*

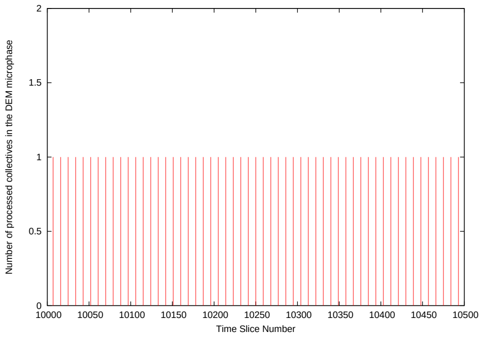

Fig. 5.4. *Number of processed collectives in the DEM microphase.*

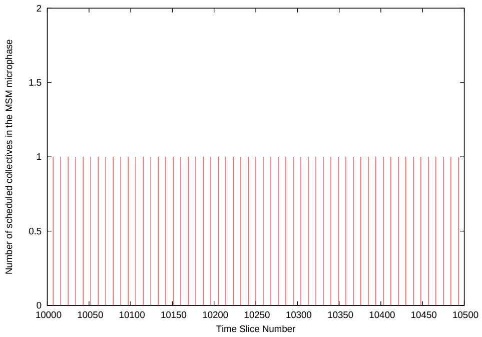

Fig. 5.5. *Number of scheduled collectives in the MSM microphase.*

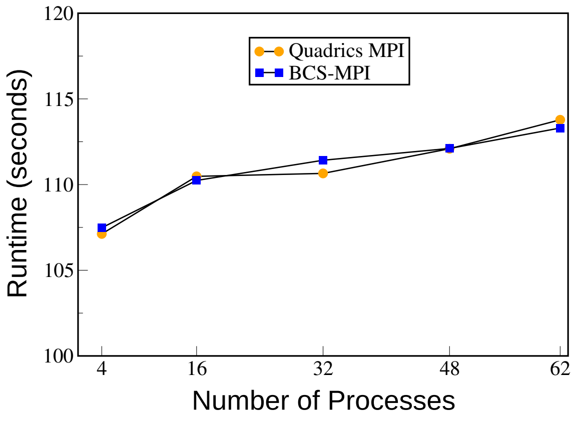

Fig. 6.1. *SAGE Performance.*

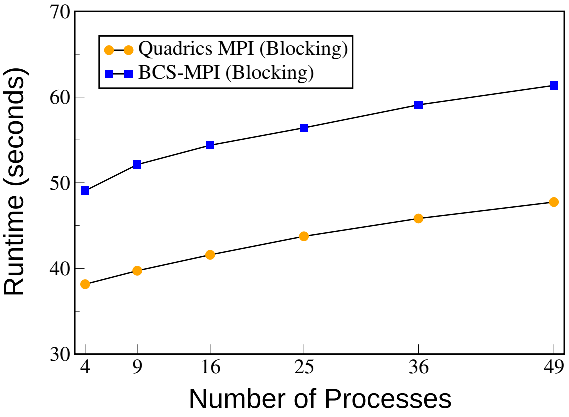

Fig. 6.2. *Blocking SWEEP3D Performance.*

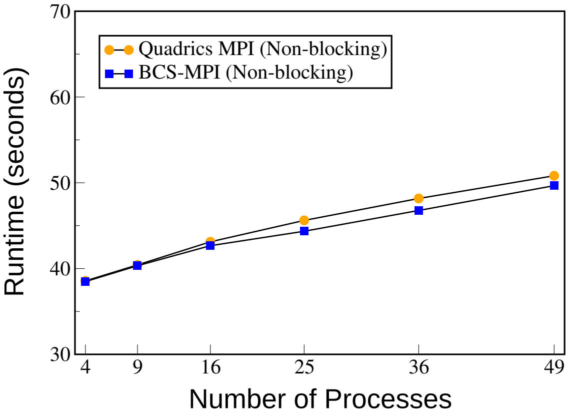

Fig. 6.3. *Non-blocking SWEEP3D Performance.*