Monitoring and Debugging Parallel Software with BCS-MPI on Large-Scale Clusters*

Juan Fernández
Departamento de Ingeniería y
Tecnología de Computadores
Universidad de Murcia, 30071 Murcia (SPAIN)
juanf@um.es

Fabrizio Petrini, Eitan Frachtenberg
CCS-3 Modeling, Algorithms & Informatics
Los Alamos National Laboratory
Los Alamos, NM 87545 (USA)
{fabrizio,eitanf}@lanl.gov

Abstract

Buffered CoScheduled (BCS) MPI is a novel implementation of MPI based on global synchronization of all system activities. BCS-MPI imposes a model where all processes and their communication are tightly scheduled at a very fine granularity. Thus, BCS-MPI provides a system that is much more controllable and deterministic. BCS-MPI leverages this regular behavior to provide a simple yet powerful monitoring and debugging subsystem that streamlines the analysis of parallel software. This subsystem, called Monitoring and Debugging System (MDS), provides exhaustive process and communication scheduling statistics. This paper covers in detail the design and implementation of the MDS subsystem, and demonstrates how the MDS can be used to monitor and debug not only parallel MPI applications but also the BCS-MPI runtime system itself. Additionally, we show that this functionality need not come at a significant performance loss.

1. Introduction

Clusters have become a predominant architecture for high-performance computing in the past decade. At the time of this writing, many systems in the Top500 list [15] are clusters, and the ever-increasing demand for computing capability is driving the construction of ever-larger clusters. For those environments, MPI is the *de facto* message-passing standard to build parallel applications. These applications usually comprise as many processes as available processors. In turn, each process may have one or more threads, open files, and pending non-blocking communication. This composite structure results in a complex global state which grows in complexity as cluster sizes increase.

* 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.

Developing, monitoring and debugging parallel MPI applications is far more complicated than sequential programs. This difficulty arises not only from the complex global state of parallel MPI applications but also from the nondeterministic nature of parallel systems. On the one hand, processes exchange messages to satisfy data dependencies, and the sequences of messages may vary between executions (e.g. when using MPI_ANY_SOURCE). On the other hand, local Operating Systems (OS) lack global awareness of parallel applications so that processes are scheduled independently. The combination of these factors may lead to highly varying run times [9] or even different results [7].

To address this challenge, the research community has proposed a number of compile-time and run-time techniques [6, 13], and several tools, such as TotalView [14], that are commercially available. In all these cases, there is an extra software component that somehow interacts with the MPI application to either gather data or perform checks of different nature. In contrast, there is no need of such an additional piece of software with BCS-MPI [1]. The MDS is directly derived from the BCS model. Hence, the MDS is tightly integrated into the BCS-MPI runtime system.

BCS-MPI is based on a methodology for the design of parallel system software in order to reduce complexity [2, 3]. This methodology relies 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. Generally speaking, this approach tries to better integrate all the nodes by leveraging modern interconnection hardware. The core primitives represent a common denominator of most system software components, and thus, constitute the backbone to integrate all nodes with a single, global OS.

BCS-MPI imposes a global communication model where communication is tightly controlled at a fine granularity. In this model, all the user and system-level communication is buffered and controlled. The entire clus-

ter 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. Moreover, this BSP-like execution model not only facilitates monitoring and debugging of parallel jobs but also enables deterministic replay of parallel applications [4] and transparent fault tolerance [12].

This paper provides three primary contributions. First, we present the software structure of BCS-MPI and the MDS, which constitutes a powerful tool for monitoring and debugging parallel MPI applications. BCS-MPI and the MDS can generate process and communication scheduling statistics with negligible performance degradation. Second, we show how to use the MDS to monitor and debug the runtime system itself as well as regular parallel MPI applications. Third, we point out the way to achieve deterministic replay of parallel MPI programs with BCS-MPI.

The rest of this paper is organized as follows. Design and implementation of BCS-MPI are presented in Section 2. Section 3 studies the functionality and implementation of the monitoring and debugging system integrated into BCS-MPI. The overhead incurred by the monitoring and debugging subsystem is characterized in Section 4. Finally, some concluding remarks are given.

2. 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* 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 provides information to schedule the communication requests issued during the previous time slice. Subsequently, all the scheduled communication operations are performed.

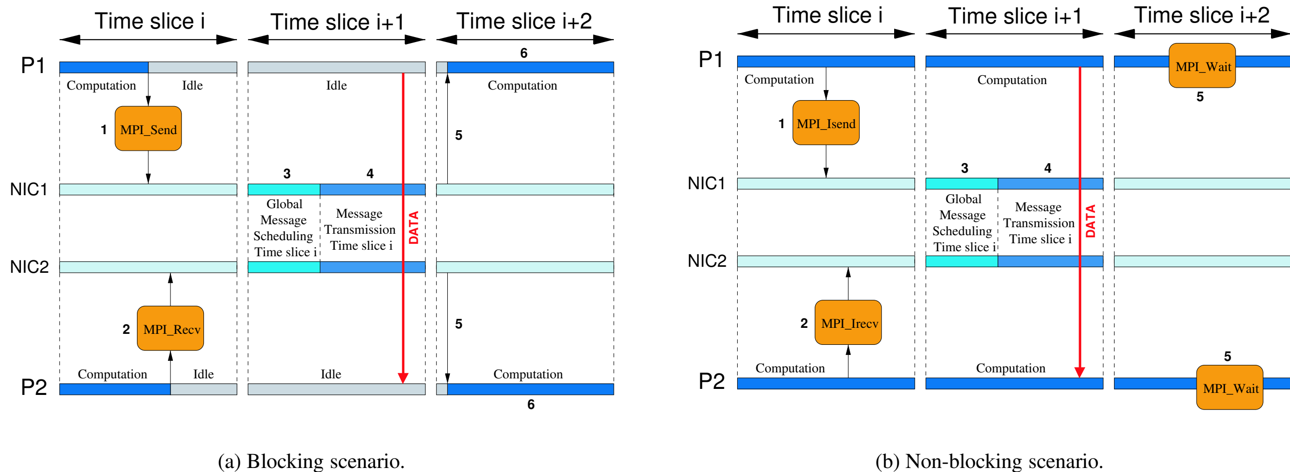
The BCS-MPI communication protocol is executed almost entirely in the network interface card (NIC) [7]. This offloading enables BCS-MPI to overlap communication with the computation executed on the host CPUs. The application processes interact directly with threads running on the NIC. When an application process invokes a communication primitive, it posts a descriptor in a region of NIC memory that is accessible to a NIC thread. Such a descriptor includes all the communication parameters that are required to complete the operation. The actual communication will be performed by a set of cooperating threads running on the NICs involved in the communication protocol. In QsNet, the chosen platform to implement BCS-MPI, 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 2.3. To illustrate how BCS-MPI communication works, two possible scenarios for blocking and non-blocking MPI point-to-point primitives are described below.

2.1. Send/Receive Scenarios

In the blocking scenario depicted in Figure 1(a), 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 as follows. (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 or by scheduling a different parallel job in time slice $i+1$. Finally, the non-blocking scenario shown in Figure 1(b) is similar to the blocking one. However, in this case, the communication is completely overlapped with the computation with no performance penalty.

2.2. BCS-MPI Implementation

For quick prototyping and portability, BCS-MPI was initially implemented for QsNet-based systems 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,



(a) Blocking scenario.

(b) Non-blocking scenario.

Figure 1. Blocking and non-blocking send/receive scenarios.

though more flexible and easier to use. An overview of the software structure of BCS-MPI is provided in Figure 2.

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

2.3. Global Synchronization Protocol

The BCS-MPI runtime system globally schedules all the computation, communication and synchronization activities of the MPI jobs at every time slice. Each time slice is divided into two main phases and several microphases. 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.

3. Monitoring and Debugging Parallel Software

As described in Section 2.2, the BCS-MPI runtime system globally schedules all the computation, communication and synchronization activities of the MPI jobs. In this way, BCS-MPI facilitates monitoring and debugging of parallel software. To this end, BCS-MPI incorporates a monitoring and debugging module, called *Monitoring and Debugging System* (MDS). This module allows monitoring and debugging, using *a posteriori* data analysis not only for MPI applications but also for 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 NIC threads can produce meaningful statistics for both the communication pattern of 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 3.1 and 3.2, 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 4. To better understand these results, Sections 3.1.1 and 3.2.1 give some insight about the way the MDS collects data.

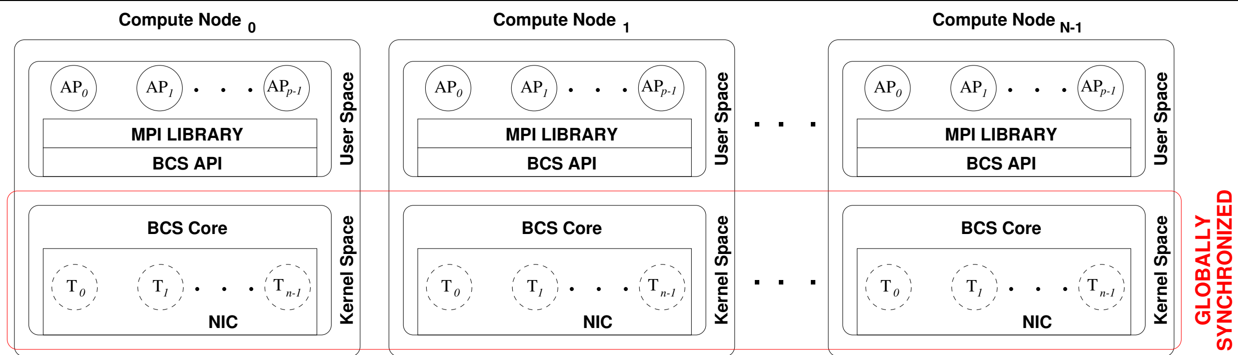


Figure 2. BCS-MPI overview.

3.1. 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 can 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.

3.1.1. Implementation. The MMDS composes part of the BCS-MPI API and as such is executed by the application processes running in the main processor.

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 to store the computation granularity distribution, and a communication overhead array to store the communication overhead distribution. 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 3. 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. 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 4). 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 MMDS is no more than a few megabytes in the worst case which is typically easy to accommodate in contemporary systems.

3.2. Elan MDS (EMDS)

The EMDS monitors the activity of the NIC threads belonging to the BCS-MPI runtime system. This module provides both global statistics on the synchronization protocols and local ones regarding process and communication scheduling on a per-node basis. Table 1 summarizes 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 schedul-

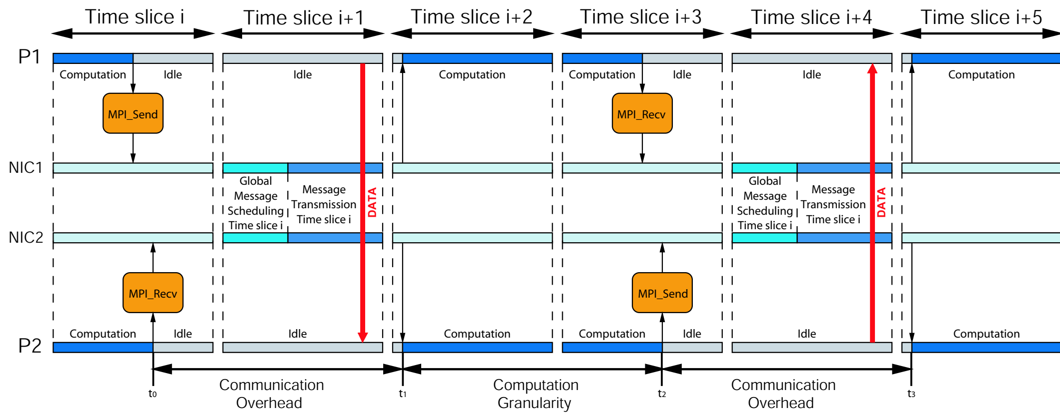


Figure 3. MMDS Time Statistics Implementation

ing 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 enables us to 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 4 illustrates the meaning of both local and global EMDS statistics. The only difference is the NIC thread which gathers the data. A thread running on the management node is in charge of the global EMDS statistics while the local EMDS statistics are obtained by different threads running on each node. For example, GTDEM, gathered by the thread running on the management node, represents the time to complete the DEM microphase in all nodes. In the meantime, LTDEM, gathered by the threads running on each node, corresponds to the time to complete the DEM microphase on every 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 Message Scheduling Microphase prematurely ends because no communications are performed. Moreover, the GETTS metric allows the verification that the master node signals all nodes at regular intervals equal to the chosen time slice value without measurable delays.

3.2.1. Implementation. The EMDS is integrated into the BCS core and as such is executed by the *Elan Thread Processor* [8, 10]. Therefore, all the data structures used by the EMDS need to be stored in Elan3 memory [10], unlike the MMDS. PCI bus transactions could introduce unpredictable delays which negate the BCS-MPI philosophy of implementing 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 $64MB$, the EMDS must be carefully designed to fit into Elan3 memory, along with the thread code, and avoid overflow situations. To this end, both global and 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 before. 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 likely. Moreover, to get the time stamps, the elan3_clock function [11] 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.

3.3. 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 $1.9ms$. Given the BCS-MPI execution model explained in Section 2.3, if the

Global 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 Sync Retries before the previous Time Slice is over
USRDEM	Unsuccessful Sync Retries before the DEM microphase is over
USRMSM	Unsuccessful Sync Retries before the MSN microphase is over

Local 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 P2P descriptors processed in the DEM microphase
NP2PMSN	Number of scheduled P2P operations in the MSN microphase
NCOLLMSN	Number of collectives processed in the DEM microphase
NCOLLMSN	Number of scheduled collectives in the MSN microphase
BPTS	Blocked processes during the current Time Slice
LTRi	Local Time to complete the execution of Routine i

Table 1. EMDS Statistics.

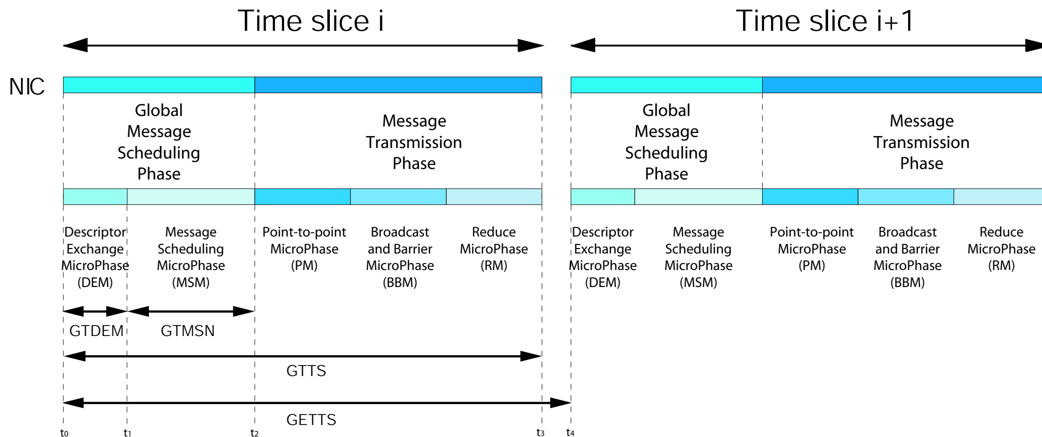
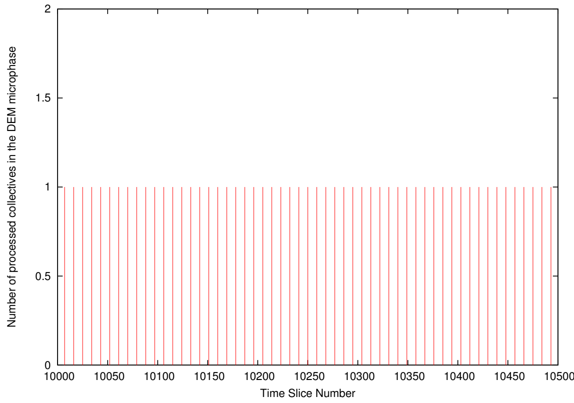


Figure 4. EMDS Time Statistics Implementation

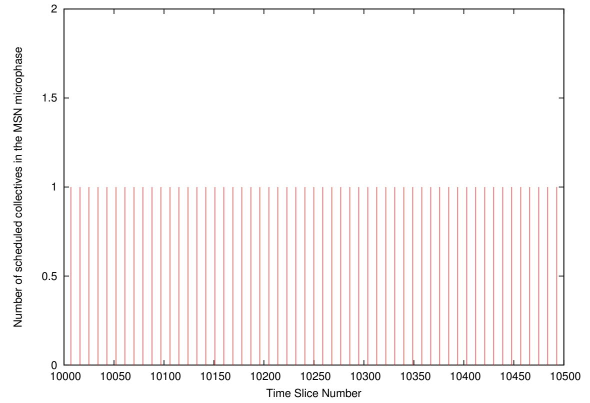
BCS-MPI runtime system globally synchronizes every 250 μs, the correct execution of the benchmark implies that: (1) all the nodes in the system synchronize every 250 μs, (2) every process invokes MPI_Barrier every nine time slices, (3) the BCS-MPI runtime system schedules a barrier every nine time slices. 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 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. Figure 6 shows 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 a 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 simple 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 [9].

3.4. Monitoring and Debugging Parallel MPI Applications

The previous section describes how to take advantage of the MDS to monitor and debug the BCS-MPI runtime system. The same approach is useful for monitoring and debugging actual applications. To demonstrate this, we use SAGE,



(a) Number of processed collectives in the DEM microphase.



(b) Number of scheduled collectives in the MSN microphase.

Figure 6. Collective statistics gathered by the EMDS.

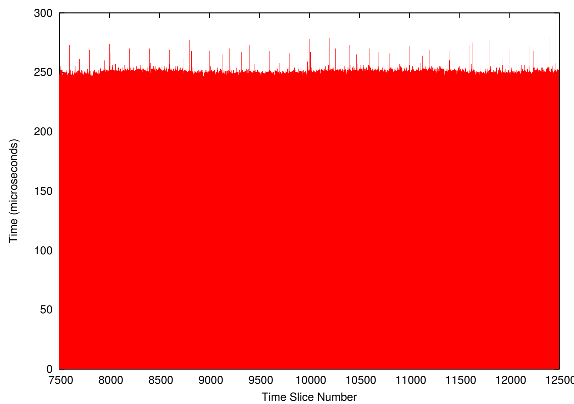


Figure 5. Global Elapsed Time from previous Time Slice

a hydrodynamics code widely used at LANL [5], to illustrate how the MDS can monitor and debug real applications. In Table 2, 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 is 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 [9], must be used until the primitive which causes the functional or performance anomaly 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 the high level of accuracy of the MDS.

4. Performance Evaluation

All experiments were conducted on a 64-node Alpha-based cluster. Each node is an Alphaserver ES40 equipped with four EV68 processors and 8 GB of memory. Nodes are interconnected with a Quadrics QsNet network and QM-400 Elan3 network interface cards [8, 10, 11]. As shown in Section 3, 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 3 shows the runtime of SAGE on 64 processors 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.

5. Concluding Remarks

BCS-MPI strives to achieve scalable performance through global scheduling of communication by logically orchestrating the activities in a large-scale system in deterministically reproducible, global steps. By leveraging the global coordination and the parallel execution of the BCS-MPI runtime system in the network interface card, we were able to develop an innovative monitoring and debugging system (MDS) that can profile

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

Table 2. SAGE statistics with the timing_h input deck.

Input deck	MDS Disabled Runtime	MMDS Runtime	MMDS Overhead	EMDS Runtime	EMDS 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%

Table 3. Overhead incurred by the MMDS and the EMDS while running SAGE.

with extreme accuracy the execution of an MPI program and of the run-time software itself with negligible overhead. We have shown how the MDS profiling capabilities can be used to monitor, debug and optimize both system software and user applications. The BCS-MPI execution model not only facilitates monitoring and debugging of parallel jobs but also paves the way to achieve deterministic replaying of parallel applications and possibly, transparent fault tolerance.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] G. Luecke, H. Chen, J. H. James Coyle, M. Kraeva, and Y. Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(93):93 – 100, 2003.

[7] 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.

[8] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January/February 2002.

[9] 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.

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

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

[12] 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.

[13] J. S. Vetter and B. R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings of IEEE/ACM Conference on SuperComputing*, Dallas, TX (USA), November 2000.

[14] www.etnus.com. TotalView.

[15] www.top500.org. Top500 Supercomputing Sites, 2004.