Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems

Gábor Dózsa[1], Sameer Kumar[1], Pavan Balaji[2], Darius Buntinas[2], David Goodell[2], William Gropp[3], Joe Ratterman[4], and Rajeev Thakur[2]

[1] IBM T. J. Watson Research Center, Yorktown Heights, NY 10598
[2] Argonne National Laboratory, Argonne, IL 64039
[3] University of Illinois, Urbana, IL 61801
[4] IBM Systems and Technology Group, Rochester, MN 55901

Abstract. With the ever-increasing numbers of cores per node on HPC systems, applications are increasingly using threads to exploit the shared memory within a node, combined with MPI across nodes. Achieving high performance when a large number of concurrent threads make MPI calls is a challenging task for an MPI implementation. We describe the design and implementation of our solution in MPICH2 to achieve high-performance multithreaded communication on the IBM Blue Gene/P. We use a combination of a multichannel-enabled network interface, fine-grained locks, lock-free atomic operations, and specially designed queues to provide a high degree of concurrent access while still maintaining MPI's message-ordering semantics. We present performance results that demonstrate that our new design improves the multithreaded message rate by a factor of 3.6 compared with the existing implementation on the BG/P. Our solutions are also applicable to other high-end systems that have parallel network access capabilities.

1 Introduction

Because of power constraints and limitations in instruction-level parallelism, computer architects are unable to build faster processors by increasing the clock frequency or by architectural enhancements. Instead, they are building more and more processing cores on a single chip and leaving it up to the application programmer to exploit the parallelism provided by the increasing number of cores. MPI is the most widely used programming model on HPC systems, and many production scientific applications use an MPI-only model. Such a model, however, does not make the most efficient use of the shared resources within the node of an HPC system. For example, having several MPI processes on a multicore node forces node resources (such as memory, network FIFOs) to be partitioned among the processes. To overcome this limitation, application programmers are increasingly looking at using hybrid programming models comprising a mixture of processes and threads, which allow resources on a node to be shared among the different threads of a process.

With hybrid programming models, several threads may concurrently call MPI functions, requiring the MPI implementation to be thread safe. In order

to achieve thread safety, the implementation must serialize access to some parts of the code by using either locks or advanced lock-free methods. Using such techniques and at the same time achieving high concurrent multithreaded performance is a challenging task [2, 3, 10].

In this paper, we describe the solutions we have designed and implemented in MPICH2 to achieve high multithreaded communication performance on the IBM Blue Gene/P (BG/P) system [4]. We use a combination of a multichannel-enabled network interface, fine-grained locks, lock-free atomic operations, and message queues specially designed for concurrent multithreaded access. We evaluate the performance of our approach with a slightly modified version of the SQMR message-rate benchmark from the Sequoia benchmark suite [8]. Although implemented on the BG/P, our techniques and optimizations are also applicable to other high-end systems that have parallel network access capabilities.

The rest of this paper is organized as follows. Section 2 provides a brief background of thread safety in MPI and MPICH2 and the architecture of the Blue Gene/P system. Section 3 describes the design and implementation of our solutions in detail. Performance results are presented in Section 4, followed by conclusions in Section 5.

2 Background

We provide a brief overview of the semantics of multithreaded MPI communication, the internal framework for supporting thread safety in MPICH2, and the hardware and software architecture of the Blue Gene/P system.

2.1 MPI Semantics for Multithreading

The MPI standard defines four levels of thread safety: single, funneled, serialized, and multiple [6]. We discuss only the most general level, `MPI_THREAD_MULTIPLE`, in which multiple threads can concurrently make MPI calls.

MPI specifies that when multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some order. Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions. As a result, multiple threads may access and modify internal structures in the MPI implementation simultaneously, thus requiring serialization within the MPI library to avoid race conditions. Logically global resources, such as allocation/deallocation of objects, context ids, communication state, and message queues, must be updated atomically.

Implementing thread safety efficiently in an MPI implementation is a challenging task. The most straightforward approach is to use a single global lock, which is acquired on entry to an MPI function and held until the function returns, unless the function is going to block on a network operation. In that case, the lock is released before blocking and then reacquired after the network operation returns. The main drawback of this approach is that it permits little concurrency in operations.

Optimizations for accessing queues, such as lock-free methods, often require single-reader/single-writer access, which can be a limitation. Since message queues are on the critical path, using simpler (classical) approaches with locks can add significant overhead. Also, locks or lock-free atomic updates themselves are expensive, even in the absence of contention, because of memory-consistency requirements (typically, some data must be flushed to main memory, an action that costs hundreds of cycles in latency).

A further complication is introduced by the feature in MPI that allows "wildcard" (`MPI_ANY_SOURCE`) receives that can match incoming messages from any sender. For any receive (or for matching any incoming message), this feature requires two logical queues to be searched atomically—receives expecting a specific sender and receives permitting any sender—in a manner that maintains MPI's message-ordering semantics. This requirement makes it difficult to allow for concurrency even in programs written to match receives with specific senders, which in the absence of `MPI_ANY_SOURCE` could be implemented efficiently with separate queues for separate senders. `MPI_ANY_SOURCE` implies a shared queue that all threads must check and atomically update, thereby limiting concurrency.

2.2 Framework for Supporting Thread Safety in MPICH2

Thread safety in MPICH2 is implemented by identifying regions of code where concurrent threads may access shared objects and marking them with macros that provide an appropriate thread-safe abstraction, such as a named critical section. For example, updates to message queues are protected by the `MSGQUEUE` critical section. Most MPI routines, other than a few that are intrinsically thread safe and require no special care, also establish a function-level critical section (`ALLFUNC`). Different granularities of thread safety (coarse-grained versus fine-grained locks or critical sections) are enabled by simply changing the definitions of these macros in a header file. For example, the simple global lock is implemented by defining the `ALLFUNC` critical section to acquire and release a global lock and defining the other macros as no-ops. Finer-grained locking is enabled by reversing these definitions, that is, defining the `ALLFUNC` critical section as a no-op and defining the other named critical sections appropriately.

MPI objects are reference counted internally. This task must be done atomically in a multithreaded environment. The reference-count updates are handled by a macro that can be defined to use a simple update (in the case of the single global lock), processor-specific atomic-update instructions, or a reference-count critical section.

In the few instances where using these macros is not convenient, we use C-preprocessor `#ifdef`s directly. Since this approach makes the code harder to maintain, however, we try to avoid it and instead rely as much as possible on a careful choice of abstractions with a common set of definitions. Using carefully chosen abstractions makes it easier to switch from a coarse-grained, single-lock approach to a finer-grained approach that permits greater concurrency.

2.3 Blue Gene/P Hardware and Software Overview

The IBM Blue Gene/P is a massively parallel system that can scale up to 3 PF/s of peak performance. Each node of the BG/P has four 850 MHz embedded PowerPC 450 cache-coherent cores on a single ASIC and can achieve a peak floating-point throughput of 13.6 GF/s per node. The nodes are connected with three networks that the application may use: a 3D torus network, which is deadlock free and provides reliable delivery of packets; a collective network, which implements global broadcast and global integer arithmetic operations; and a global interrupt network for fast barrier-synchronization operations. Each node has a direct memory access (DMA) engine to facilitate injecting and receiving packets to and from the torus network. This feature allows the cores to offload packet management and enables better overlap of communication and computation.

The MPI implementation on the BG/P is based on MPICH2 [7] and is layered on top of a lower-level messaging library called the Deep Computing Messaging Framework (DCMF) [5]. DCMF provides basic message-passing services that include point-to-point operations, nonblocking one-sided get and put operations, and an optional set of nonblocking collective calls. MPICH2 is implemented on the BG/P via an implementation of the internal MPID abstract device interface on top of DCMF, called `dcmfd`. The currently released version supports the `MPI_THREAD_MULTIPLE` level of thread safety by using the simple, unoptimized approach of a single global lock.

The DMA engine on each node supports 32 injection FIFOs and 8 reception FIFOs per core, an important feature for the work described in this paper. The operating system on the node supports a maximum of four threads, in other words, at most 1 thread per core. Although this limit is much smaller than what is allowed on commodity multicore/SMP platforms, those platforms typically do not offer sufficient parallelism at the network-hardware level that concurrently communicating threads could exploit. On such systems, the MPI implementation would need to serialize accesses to network hardware and operating-system resources and thus would not result in scalable multithreaded communication performance. A commodity-cluster node would need at least four NICs to provide a level of network parallelism comparable to a BG/P node. Therefore, despite the relatively modest number of threads allowed on a BG/P node, the parallelism in the network hardware makes it an interesting platform for studying how to optimize multithreaded MPI communication.

3 Enabling Concurrent Multithreaded MPI Communication on BG/P

To achieve high-performance multithreaded MPI communication on the BG/P, we redesigned multiple layers of the communication-software stack. We enhanced both DCMF and MPICH2 to support multiple communication *channels* between pairs of processes, such that communication from multiple threads on different channels can take place concurrently. We also modified the data structures and

algorithms used to implement message queues in MPICH2 in order to enable message-queue manipulations in parallel on a channel basis. In the following subsections, we describe all these optimizations.

3.1 Multichannel Extensions to DCMF

In the existing design of DCMF, only one abstract DMA device is instantiated per MPI process. This single DMA device allocates one injection/reception FIFO group and provides a single access point for the underlying DMA hardware resources. We extended DCMF to have multiple DMA devices that allocate multiple injection/reception groups for each MPI process. For example, in BG/P's SMP mode, where a program runs with one process and up to four threads on each node, four DMA devices are instantiated that allocate four injection and reception groups. In BG/P's dual mode, with two MPI processes with two threads each per node, each process instantiates two DMA devices. Multiple threads of an MPI process can access these DMA software devices independently and in parallel. DCMF encapsulates DMA devices into software abstractions called channels. A channel assigns a mutex to control access to a particular DMA device.

API Changes To allow threads to lock channels, we added two new calls to the DCMF API: `DCMF_Channel_acquire` and `DCMF_Channel_release`. The rest of the API remained unchanged. In particular, we did not add new arguments for `DCMF_Send` to specify a send channel. Instead, the new `DCMF_Channel_acquire` call saves the ID of the locked channel in thread-private memory. Subsequent calls to send functions use this thread-private information to post messages to the DMA device currently locked by the executing thread. Send function calls specify the same DMA group ID for both injection and reception of a message; that is, by locking a channel, the sender thread implicitly also defines the reception channel at the destination for outgoing messages.

This approach to extending the API has the advantage that it requires minimal changes in upper levels of software that call DCMF. It has the drawback of limited flexibility, however; for example, it cannot specify different send and receive channels for a particular message. We plan to explore a more full-featured API that provides greater flexibility.

Progress Engine The generic DMA progress engine in DCMF ensures that pending outgoing and incoming messages are processed. We extended the progress engine to support multiple channels. Ideally, each channel is advanced by a separate thread, which results in fully parallel progress of the DMA devices. For instance, in the SMP mode, four MPI threads can run on the four cores and make progress on only their corresponding channels. This scheme, however, assumes that all four MPI threads are always active; that is, all of them issue DCMF advance calls eventually, so that pending messages are processed at some point on every channel. If all the threads are not active, this fully parallel progress

approach may fail. For instance, a multithreaded MPI application may enter a global barrier by issuing `MPI_Barrier` calls from threads running on different cores on the different nodes. Only one thread will call the barrier function on each node, and the other threads may be simply blocked (or not even started yet) until the global barrier completes. This situation can lead to a deadlock if a barrier message arrives at a node on a channel that is not advanced by the thread executing the barrier call.

In order to comply with MPI progress semantics, each thread must eventually make progress on every channel. For thread safety, we also need to prevent multiple threads from accessing the same channel simultaneously. A call to the DMA progress engine causes progress by attempting to lock a channel; if the lock succeeds, the DMA device of the channel is advanced and the channel is unlocked. Making progress on multiple channels instead of just one channel implies higher overhead, which can hurt message latency on the low-frequency BG/P cores. However, it must be done at least occasionally in order to satisfy MPI progress semantics. For this purpose, we use an internal parameter (say, n) to decide how often a thread will attempt to advance other channels. A thread will normally advance only its own channel; but on every nth call to the progress function, it will also try to advance other channels. Thus, all threads can make independent parallel progress most of the time, while still guaranteeing MPI progress semantics.

3.2 Exploiting Multiple Channels in MPICH2

The current MPI 2.2 standard does not directly translate the notion of multiple communication channels into a user-visible concept. For the upcoming MPI-3 standard, Marc Snir has proposed extending MPI to support multiple "endpoints" per process [9], which would map cleanly to our definition of channels. Until such explicit support becomes part of the standard, however, MPI can take advantage of multiple channels only in an application-transparent fashion, that is, by using multiple channels internally without exposing them to the user.

We modified the `dcmfd` device in MPICH2 to select and acquire an appropriate channel by calling `DCMF_Channel_acquire(channel)` immediately before issuing a DCMF send call. After the send call completes, the channel is released. We calculate the channel for a particular message by means of a simple hash function: $channel = (source + dest) \bmod num\_channels$. Selecting the channel based on the source and destination ranks in this manner has the following desirable implications:

- Messages sent on a given communicator from a particular source to the same destination process travel over the same channel. This feature makes it easy for us to support MPI's non-overtaking message-ordering semantics, which require that messages from the same source to the same destination appear in the order in which they were sent.
- Messages sent to a particular destination node from different sources are distributed among the available reception channels. This feature enables incoming messages to be received in parallel.

Truly parallel processing of incoming messages at the MPI level also requires support for parallel message matching, as described below.

3.3 Parallel Receive Queues

MPICH2 has two receive queues implemented via linked lists: a queue of receives posted by the application and a queue of unexpected messages, namely, messages that were received before the application posted the matching receive. When an application posts a receive, the unexpected-message queue is first searched for a matching message. If none is found, the receive is enqueued on the posted-receive queue. Similarly, when a message is received from the network, the posted-receive queue is first searched for a matching receive. If none is found, the message is enqueued on the unexpected queue.

We parallelized the receive queues by providing a separate pair of posted- and unexpected-receive queues for each source rank. In this case, an additional queue is needed to hold posted wildcard receives (source=`MPI_ANY_SOURCE`). When a message is received from the network and a matching receive is not found in the posted-receive queue for the corresponding channel, the progress engine checks the wildcard queue. If the wildcard queue is not empty, the progress engine acquires the wildcard-queue lock and searches the queue for a match. If a match is not found, the message is enqueued on the channel's unexpected queue.

A complication is introduced when a posted wildcard receive is followed by a non-wildcard receive with a matching tag. Since MPI's message-ordering semantics require that the wildcard receive be matched first and since MPICH2's progress engine first searches the channel-receive queues, we queue the non-wildcard receive in the wildcard queue. When the wildcard receive is matched and removed from the wildcard queue, we move the non-wildcard receive into its channel queue.

4 Performance Results

Currently, there is no canonical benchmark suite or application to measure multithreaded messaging efficiency of MPI. Also, the `MPI_THREAD_MULTIPLE` mode is often not efficiently supported by existing MPI implementations, which in turn deters applications from using it. The commonly used NAS Parallel Benchmarks (NPB) [1] are not multithreaded. The multi-zone variants of the NAS Parallel Benchmarks (NPB-MZ) [11] do use MPI+threads via OpenMP, but they use only the `MPI_THREAD_FUNNELED` level of thread safety.

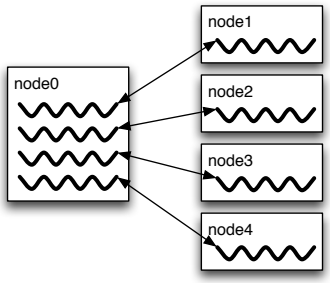

Fig. 1. Communication pattern of the Neighbor Message Rate Benchmark

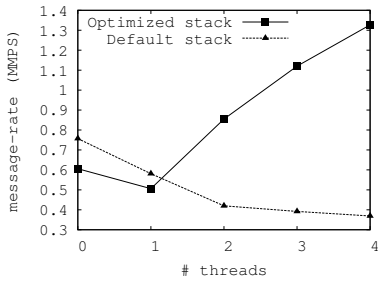

Fig. 2. Message rate performance of default and optimized software stacks on BG/P

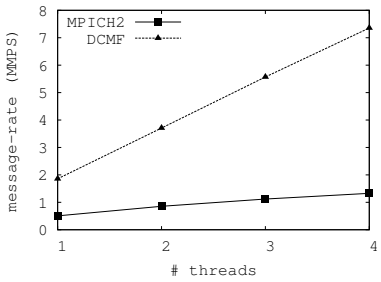

Fig. 3. Message rates with the optimized stack when using MPI versus direct DCMF

We chose message rate as a metric to measure messaging performance. We used a slightly modified version of the SQMR Phloem microbenchmark from the Sequoia benchmark suite [8]. Specifically, the original SQMR code runs single-threaded MPI processes; we adapted it for multithreaded processes running in `MPI_THREAD_MULTIPLE` mode.

The modified benchmark, which we call the Neighbor Message Rate Benchmark, measures the aggregate message rate for N threads in a single MPI process, each sending to and receiving from a corresponding peer process on a separate node, as shown in Figure 1. Each iteration of the benchmark involves each thread posting 12 nonblocking receives and 12 nonblocking sends from/to the peer thread, followed by a call to `MPI_Waitall` to complete the requests. Each thread executes 10 warm-up iterations before timing 10,000 more iterations. We used zero-byte messages in order to minimize the impact of data-transfer times on the measurements. The benchmark reports the total number of messages sent (in millions) divided by the elapsed time in seconds. We evaluate our solution based on the overall message rate of the "root" process and the scaling of the message rate with the number of threads.

We ran the benchmark on the BG/P and varied the number of threads from 1 to 4. We also ran it in the `MPI_THREAD_SINGLE` mode to determine the best achievable performance for a single thread without any locking overhead. Figure 2 shows the results with the default production BG/P software stack and with our optimized DCMF and MPICH2. (The case with 0 threads represents the `MPI_THREAD_SINGLE` mode.) The performance with the optimized stack is much better than with the default stack where the message rate actually decreases with the number of threads. With 4 threads, the message rate with optimized stack is 3.6 times higher than with the default stack. Scaling is not perfect though; we observe an average 10% degradation per thread from linear scaling.

To locate the source of this scaling degradation and to measure the MPI overhead in general, we also implemented a DCMF version of the Neighbor Message Rate Benchmark, which directly makes DCMF calls. Figure 3 shows the results of running both the MPI and DCMF versions of the benchmark with our opti-

mized stack. The performance with direct DCMF is much higher than with MPI. We believe this difference is because DCMF is a much simpler, lower-level API. MPI's message-ordering and matching semantics as well as the notion of communicators, etc., make it more difficult to optimize for multithreading. Nonetheless, the magnitude of the difference suggests room for further optimization, which we plan to investigate. Locking overhead for dynamic channel selection and receive-queue management for message matching are two areas that we specifically plan to optimize further. We also expect that the new proposal for multiple endpoints in MPI-3 [9] will help alleviate some of the bottlenecks at the MPI level.

5 Conclusions

Running MPI applications in fully multithreaded mode is becoming a significant issue as a result of the increasing importance of hybrid programming models for multicore high-end systems. We have presented a solution to achieve high messaging performance in MPICH2 when multiple threads make MPI calls concurrently. We use a combination of a multichannel-enabled network interface, fine-grained locks, lock-free atomic operations, and message queues specifically designed for concurrent multithreaded access. We introduce the "channel" abstraction as the unit of parallelism at the network-interface level and show how MPICH2 can take advantage of channels in a user-transparent way. Applying our optimizations on the Blue Gene/P messaging stack, we demonstrate a factor of 3.6 improvement in multithreaded MPI message rate. Furthermore, the message rate scales reasonably with the number of MPI threads in our optimized stack, as opposed to the default stack where the aggregate message rate decreases with multiple threads. We plan to investigate further optimizations to improve MPI performance compared with native DCMF performance.

The proposed solutions and optimizations for defining and managing multiple network "channels" are also applicable to other high-end systems with parallel network access capabilities. Implementation details will, of course, differ as they depend on the particular messaging software stack, but the techniques we have described for providing access to multiple network channels from concurrent MPI threads and managing progress on multiple channels in parallel should be directly applicable.

Acknowledgments

This work was supported in part by the U.S. Government contract No. B554331; the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract DE-AC02-06CH11357 and DE-FG02-08ER25835; and by the National Science Foundation under grant #0702182.

References

1. Bailey, D., Harris, T., Saphir, W., Wijngaart, R.V.D., Woo, A., Yarrow, M.: The NAS parallel benchmarks 2.0. NAS Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA (1995)

2. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Thakur, R.: Fine-grained multithreading support for hybrid threaded MPI programming. International Journal of High Performance Computing Applications 24(1), 49–57 (2010)
3. Gropp, W., Thakur, R.: Thread safety in an MPI implementation: Requirements and analysis. Parallel Computing 33(9), 595–604 (September 2007)
4. IBM System Blue Gene solution: Blue Gene/P application development. http://www.redbooks.ibm.com/redbooks/pdfs/sg247287.pdf
5. Kumar, S., Dozsa, G., Almasi, G., Heidelberger, P., Chen, D., Giampapa, M.E., Blocksome, M., Faraj, A., Parker, J., Ratterman, J., Smith, B., Archer, C.J.: The Deep Computing Messaging Framework: Generalized scalable message passing on the Blue Gene/P supercomputer. In: Proceedings of the 22nd International Conference on Supercomputing. pp. 94–103. ACM, New York (2008)
6. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.2 (September 2009), http://www.mpi-forum.org
7. MPICH2. http://www.mcs.anl.gov/mpi/mpich2
8. Sequoia benchmark codes. https://asc.llnl.gov/sequoia/benchmarks/
9. Snir, M.: MPI-3 hybrid programming proposal, version 7. http://meetings.mpi-forum.org/mpi3.0_hybrid.php
10. Thakur, R., Gropp, W.: Test suite for evaluating performance of multithreaded MPI communication. Parallel Computing 35(12), 608–617 (December 2009)
11. Wijngaart, R.V.D., Jin, H.: NAS parallel benchmarks, multi-zone versions. NAS Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA (2003)