A Case for Standard Non-Blocking Collective Operations

Torsten Hoefler[1,4], Prabhanjan Kambadur[1], Richard L. Graham[2], Galen Shipman[3], and Andrew Lumsdaine[1]

[1] Open Systems Lab, Indiana University, Bloomington IN 47405, USA,
{htor,pkambadu,lums}@cs.indiana.edu,
[2] National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge TN 37831, USA,
rlgraham@ornl.gov,
[3] Advanced Computing Laboratory, Los Alamos National Laboratory, Los Alamos, NM 87545, USA,
gshipman@lanl.gov,
LA-UR-07-3159,
[4] Chemnitz University of Technology, 09107 Chemnitz, Germany,
htor@cs.tu-chemnitz.de,

Abstract. In this paper we make the case for adding standard non-blocking collective operations to the MPI standard. The non-blocking point-to-point and blocking collective operations currently defined by MPI provide important performance and abstraction benefits. To allow these benefits to be simultaneously realized, we present an application programming interface for non-blocking collective operations in MPI. Microbenchmark and application-based performance results demonstrate that non-blocking collective operations offer not only improved convenience, but improved performance as well, when compared to manual use of threads with blocking collectives.

1 Introduction

Although non-blocking collective operations are notably absent from the MPI standard, recent work has shown that such operations can be beneficial, both in terms of performance and abstraction. Non-blocking operations allow communication and computation to be overlapped and thus to leverage hardware parallelism for the asynchronous (and/or network-offloaded) message transmission. Several studies have shown that the performance of parallel applications can be significantly enhanced with overlapping techniques (e.g., cf. [1, 2]). Similarly, collective operations offer a high-level interface to the user, insulating the user from implementation details and giving MPI implementers the freedom to optimize their implementations for specific architectures.

In this paper, we advocate for standardizing non-blocking collective operations. As with non-blocking point-to-point operations and blocking collective operations, the performance and abstraction benefits of non-blocking collective operations can only be realized if these operations are represented with a procedural abstraction (i.e., with an API). Although a portable library on top of MPI could be used to provide non-blocking collective functionality to the HPC community, standardization of these operations is essential to enabling their widespread adoption. In general, vendors will only tune operations that are in the standard and users will only use features that are in the standard.

It has been suggested that non-blocking collective functionality is not needed explicitly as part of MPI because a threaded MPI library could be used with collective communication taking place in a separate thread. However, there are several drawbacks to this approach. First, it requires language and operating system support for spawning and managing threads, which is not possible on some operating systems—in particular on operating systems such as Catamount designed for HPC systems. Second, programmers must then explicitly manage thread and synchronization issues for purposes of communication even though these issues could and should be hidden from them (e.g., handled by an MPI library). Third, the required presence of threads and the corresponding synchronization mechanisms imposes the higher cost of thread-safety on all communication operations, whether overlap is obtained or not (cf. [3]). Finally, this approach provides an asymmetric treatment of collective communications with respect to point-to-point communications (which do support asynchronous communications).

Non-blocking collective operations provide some performance benefits that may only be seen at scale. The scalability of large scientific application codes is often dependent on the scalability of the collective operations used. At large scale, system noise affects the performance of collective communications more than it affects the performance of point-to-point operations. because of the ordered communications patterns use by collective communications algorithms. To continue to scale the size of HPC systems to peta-scale and above, we need communication paradigms that will admit effective use of the hardware resources available on modern HPC systems. Implementing collective operations so that they do not depend on the the main CPU is one important means of reducing the effects of system noise on application scalability.

1.1 Related Work

Several efforts have studied the benefits of overlapping computation with communications, with mixed results. Some studies have shown that non-blocking collective operations improve performance, and in other cases a bit of performance degradation was observed. Danalis et.al. [2] obtained performance improvement by replacing calls to MPI blocking collectives with calls to non-blocking MPI point-to-point operations. Kale et al. [4] analyzed the applicability of a non-blocking personalized exchange to a small set of applications. Studies such as [1, 5] mention that non-blocking collective operations would be beneficial but do not quantify these benefits. IBM extended the standard MPI interface to include non-blocking collectives in their parallel environment (PE), but have dropped support for this non-standard functionality in the latest release of this PE. Due to its channel semantics, MPI/RT [6] defines all operations, including collective operations, in a non-blocking manner. Hoefler et. al. [7, 8] have shown that non-blocking collective operations can be used to improve the performance of parallel applications. Finally, several studies of the use of non-blocking collectives to optimize three-dimensional FFTs have been done [5, 9–11]. The results of applying these non-blocking communication algorithms (replacing MPI All-To-All communications) were inconclusive. In some cases the non-blocking collectives improved performance, and in others performance degraded a bit.

The remainder of the paper is structured as follows. Section 2 describes our proposed application programming interface followed by a discussion of different implementation options in Section 3. Microbenchmarks of two fundamentally different implementations are presented in Section 4. Section 5 presents a case study of the applicability of non-blocking collective operations to the problem of a parallel three-dimensional Fourier Transformation.

2 Application Programming Interface

We propose an API for the non-blocking collectives that is very similar to that of the blocking collectives and the former proprietary IBM extension. We use a naming scheme similar to the one used for the non-blocking point-to-point API (e.g., MPI_Ibarrier instead of MPI_Barrier). In addition, request objects (MPI_Request) are used for a completion handle. The proposed interfaces to all collective operations are defined in detail in [12]. For example, a non-blocking barrier would look like:

```
MPI_Ibarrier(comm, request);
...
/* computation, other MPI communications */
...
MPI_Wait(request, status);
```

Our interface relaxes the strict MPI convention that only one collective operation can be active on any given communicator. We extend this so that we can have a huge number (system specific, indicated by MPI_ICOLL_MAX_OUTSTANDING, cf. [12]) of parallel non-blocking collectives and a single blocking collective outstanding at any given communicator. Our interface does not introduce collective tags to stay close to the traditional syntax of collective operations. The order of issuing a given collective operation defines how the collective-communications traffic matches up across communicators. Similar to point-to-point communications, progress for these non-blocking collective operations depends on both underlying system hardware and software capabilities to support asynchronous communications, as well implementation of these collectives by the MPI library. In some cases MPI_Test or MPI_Wait may need to be called to progress these non-blocking collective operations. This may be particularly true for collective operations that transform user data, such as MPI_Allreduce.

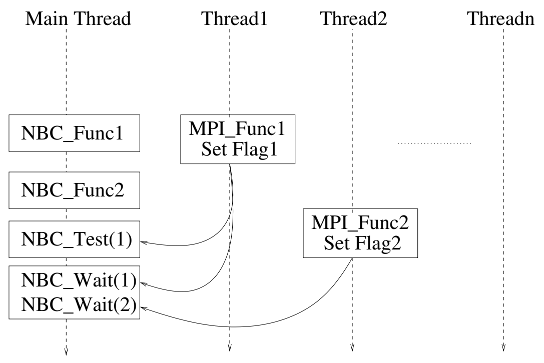

Fig. 1. Execution of NBC_ calls in separate threads

3 Advice to Implementors

There are two different ways to implement support for non-blocking collective operations. The first way is to process the blocking collective operation in a separate thread and the second way is to implement it on top of non-blocking point-to-point operations. We will evaluate both implementations in the following. We use a library approach, e.g., both variants are implemented in a library with a standardized interface which is defined in [12]. This enables us to run identical applications and benchmarks with both versions. The following section discusses our implementation based on threads.

3.1 Implementation with Threads

The threaded implementation, based on the pthread interface, is able to spawn a user-defined number of communication threads to perform blocking collective operations. It operates using a task-queue model, where every thread has its own task queue. Whenever a non-blocking collective function is called, a work packet (containing the function number and all arguments) is placed into the work queue of the next thread in a round robin scheme.

The worker threads could either poll their work queue or use condition signals to be notified. Condition signals may introduce additional latency while constant polling increases the CPU overhead. We will analyze only the condition wait method in the next section because experiments with the polling method showed that it is worse in all regards. Since there must be at least one worker thread per MPI job, at most half of the processing cores is available to compute unless the system is oversubscribed.

Whenever a worker thread finds a work packet in its queue (either during busy waiting or after being signaled), the thread starts the corresponding collective MPI operation and sets a flag after its completion. All asynchronous operations have to be started on separate communicators (mandated by the MPI standard).

Thus, every communicator is duplicated on its first use with any non-blocking collective and cached for later calls. Communicator duplication is a blocking collective operation in itself and causes matching problems when it's run with threads (cf. [3]). The communicator duplication has to be done in the user thread to avoid any race conditions, which makes the first call to a non-blocking collective operation with every communicator block. All subsequent calls are executed truly non-blocking.

When the user calls NBC_Test, the completion flag is simply checked and the appropriate return code generated. A call to NBC_Wait waits on a condition variable.

3.2 Implementation with non-blocking Point-to-Point

The point-to-point message based implementation is available in LibNBC. LibNBC is written in ANSI C using MPI-1 functionality exclusively to ensure highest portability. The full implementation is open source and available for public download on the LibNBC website [13]. The detailed implementation documentation is provided in [14], and the most important issues are discussed in the following.

The central part of LibNBC is the collective schedule. A schedule consists of the operations that have to be performed to accomplish the collective task (e.g., an MPI_Isend, MPI_Irecv). The collective algorithm is implemented like in the blocking case, based on point-to-point messages. But instead of performing all operations immediately, they are saved in the collective schedule together with their dependencies. However, the internal interface to add new algorithms is nearly identical to the MPI interface. A detailed documentation about the addition of new collective algorithms and the internal and external programming interfaces of LibNBC is available in [14]. The current implementation is optimized for InfiniBand$^{\text{TM}}$ and implements different algorithms for most collective operations (cf. [15]).

All communications require an extra communicator to prevent collisions with the user program. This communicator is duplicated from the original one in the first NBC call with a new communicator and cached for subsequent calls. This makes the first call blocking, as in the threaded implementation described in the previous section.

4 Microbenchmarking the Implementations

We developed a micro-benchmark to assess the performance and overlap potential of both implementations of non-blocking collective operations. This benchmark uses the interface described in [12]. For a given collective operation, it measures the time to perform the blocking MPI collective, the non-blocking collective in a blocking way (without overlap) and the non-blocking collective interleaved with busy loops to measure the potential computation and communications overlap. A detailed description of the benchmark is available in [8]. In

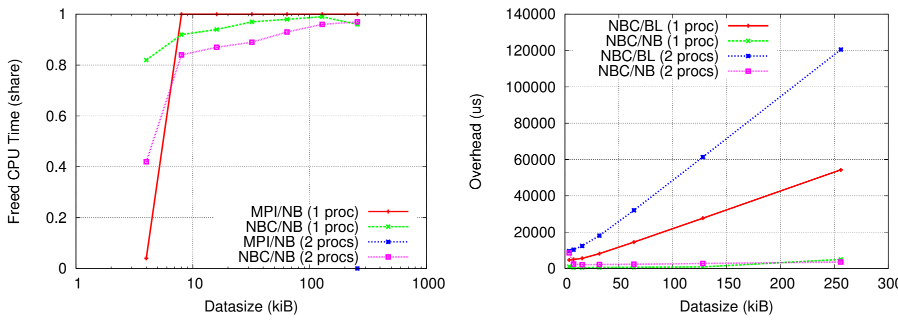

Fig. 2. Left: Share of the freed CPU time with the non-blocking MPI and NBC alltoall operation with regards to the blocking MPI implementation. Right: Blocking and non-blocking NBC_Ialltoall overhead for different CPU configurations. Measured with 128 processes in 64 and 128 nodes respectively.

addition, the benchmark measures the communication overhead of blocking and non-blocking collective operations. Overhead is defined as the time the calling thread spends in communications related routines, i.e., the time the thread can't spend doing other work. The communication overhead of blocking operations is the amount of time to finish the collective operation, as the collective call does not complete until the collective operation had completed locally. Non-blocking operations allow for overlap of the communication latency and the overhead has the potential to be less than the time to complete the given collective, and providing the calling thread compute cycles. The communication overhead of non-blocking operations is highly implementation, network, and communications stack dependent. We could not run these thread-based tests on the Cray XT4, as it does not provide thread support.

Using both implementations and the benchmark results, four different times are measured:

- Blocking MPI collective in the user thread (MPI/BL)
- Blocking MPI collective in a separate communications thread to emulate non-blocking behavior (MPI/NB)
- Non-blocking NBC operation without overlap, i.e., the initiation is directly followed by a wait (NBC/BL)
- Non-blocking NBC operation with maximum overlap, i.e., computing at least as long as an NBC/BL operation takes (NBC/NB)

We benchmarked both implementations with Open MPI 1.2.1 [16] on the Coyote cluster system at Los Alamos National Labs, a 1290 node AMD Opteron cluster with an SDR InfiniBand network. Each node has two single core 2.6 GHz AMD Opteron processors, 8 GBytes of RAM and a single SDR InfiniBand HCA. The cluster is segmented into 4 separate scalable units of 258 nodes. The largest job size that can run on this cluster is therefore 516 processors.

Figure 2 shows the results of the microbenchmark for different CPU configurations of 128 processes running on Coyote. The threaded MPI implementation allows nearly full overlap (frees nearly 100% CPU) as long as the system is not oversubscribed, i.e., every communication thread runs on a separate core. However, this implementation fails to achieve any overlap (it shows even negative impact) if all cores are used for computation. The implementation based on non-blocking point-to-point (LibNBC) allows decent overlap in all cases, even if all cores are used for computation.

5 Case Study: Three-dimensional FFT

Parallel multi-dimensional Fast Fourier Transformations (FFTs) are used as compute kernels in many different applications, such as quantum mechanical or molecular dynamic calculations. In this paper we also study the application of non-blocking collective operations to optimize a three-dimensional FFT to demonstrate the benefit of overlapping computation with communication for this important kernel. This operation is used at least once per application computational step.

The three-dimensional FFT can be split into three one-dimensional FFTs performed along all data points. We use FFTW to perform the 1d-FFTs and distribute the data block-wise (blocks of xy-planes) so that the x and y dimensions can be transformed without redistributing the data between processors. The z transformation requires a data redistribution among all nodes which is efficiently implemented by an MPI_Alltoall function.

A pipeline scheme is used for the communication. As soon as the first data elements (i.e., planes) are ready, the communication of them is started in a non-blocking way with NBC_Ialltoall. This enables the communication of those elements to overlap with the computation of all following elements. As soon as the last element is computed and its communication is started, all outstanding collective operations are completed with NBC_Wait (i.e., the last operation has no overlap potential).

We benchmark the strong scaling of a full transformation of a 1024^3 point FFT box (960^3 for 32 processes due to memory limitations) on the the Cray XT4, Jaguar, at the National Center for Computational Sciences, Oak Ridge National Laboratory. This cluster is made up of a total of 11,508 dual socket 2.6 GHz dual-core AMD Opteron chips, and the network is a 3-D torus with the Cray-designed SeaStar [17] communication processor and network router designed to offload network communication from the main processor. The compute nodes run the Catamount lightweight micro-kernel. All communications use the Portals 3.3 communications interface [18]. The Catamount system does not support threads and can thus not run the threaded implementation. An unreleased development version of Open MPI [16] was used to perform these measurements, as Open MPI 1.2.1 does not provide Portals communications support. However, using the NIC-supported overlap with LibNBC results in a better overall system usage and an up to 14.2% higher parallel efficiency of the FFT on 128 processes.

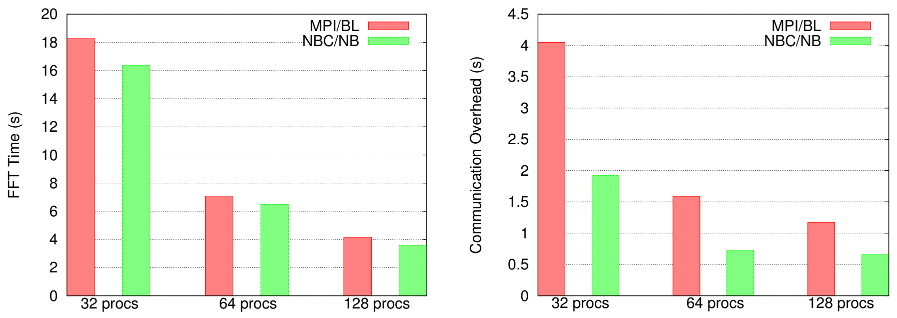

Fig. 3. Left: Blocking and non-blocking FFT times for different process counts on the XT4 system. Right: Communication overhead.

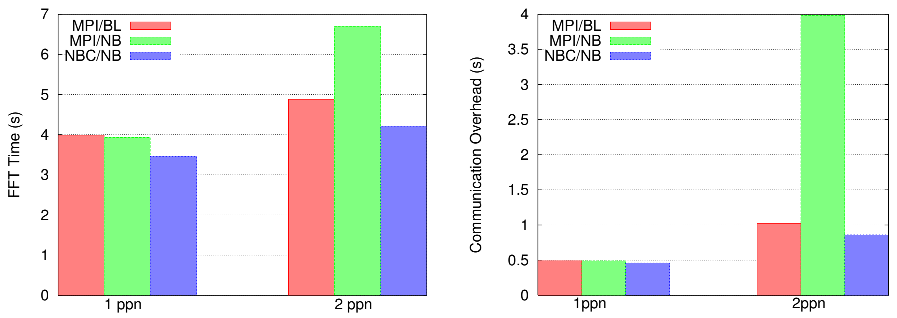

Fig. 4. Left: Blocking and non-blocking FFT times for different node configurations of 128 processes on the Coyote system (using 128 or 64 nodes respectively). Right: Communication overhead.

Another benchmark on the Coyote system (cf. 4), shown on Fig. 4, shows results for runs of the 1024^3 FFT box transformation on 128 processes with either 1 process per node (1ppn) or two processes per node (2ppn). This effectively compares the efficiency of the MPI approach (perform the non-blocking collectives in a separate thread) with the LibNBC approach (use non-blocking point-to-point communication). We clearly see the the LibNBC approach is superior on this system. As soon as all available CPUs are used for computation, the threaded approach even slows the execution down (cf. Section 4). Our conclusion is that with the currently limited number of CPU cores, it does not pay off to invest half of the cores to process asynchronous collectives with the MPI approach; they should rather be used to perform useful computation. Thus, we suggest the usage of non-blocking point-to-point as in LibNBC.

6 Conclusions

As modern computing and communication hardware is becoming more powerful, it is providing more opportunities for delegation of communication operations and hiding of communication costs. MPI has long supported asynchronous point-to-point operations to take advantage of these capabilities. It is clearly time for the standard to support non-blocking functionality for collective operations.

The interface we propose is a straightforward extension of the current MPI collective operations and we have implemented a prototype of these extensions in a library using MPI point-to-point operations. We note however, that implementing non-blocking collective operations as a separate library in this way requires implementing (potentially quite similar) collective algorithms in two different places (the blocking and non-blocking cases). Having those operations standardized in MPI would enable a single shared infrastructure inside the MPI library. In addition, communicator duplication is necessary in both implementations and can not be done in a non-blocking way without user interaction.

Our results with a microbenchmark and an application clearly show the performance advantages of non-blocking collectives. In the case of an all-to-all communication, we are able to overlap up to 99% of the communication with user computation on our systems. The application of a pipelined computation/communication scheme to a 3d-FFT shows application performance gains for a 128 process job of up to 14.2% on a Cray XT4 and 13.7% on an InfiniBand-based cluster system. In particular, we show that using the MPI-2 threaded model for a real-world problem to perform non-blocking collective operations is clearly sub-optimal to an implementation based on non-blocking point-to-point operations.

Acknowledgements

Pro Siobhan. The authors want to thank Laura Hopkins from Indiana University for editorial comments and helpful discussions. This work was supported by a grant from the Lilly Endowment and National Science Foundation grant EIA-0202048. This research was funded in part by a gift from the Silicon Valley Community Foundation, on behalf of the Cisco Collaborative Research Initiative of Cisco Systems. This research was also sponsored in part by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

References

1. Brightwell, R., Riesen, R., Underwood, K.D.: Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. Int. J. High Perform. Comput. Appl. **19**(2) (2005) 103–117
2. Danalis, A., Kim, K.Y., Pollock, L., Swany, M.: Transformations to parallel codes for communication-computation overlap. In: SC '05: Proceedings of the 2005

ACM/IEEE conference on Supercomputing, Washington, DC, USA, IEEE Computer Society (2005) 58

3. Gropp, W.D., Thakur, R.: Issues in developing a thread-safe mpi implementation. In Mohr, B., Träff, J.L., Worringen, J., Dongarra, J., eds.: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Proceedings. Volume 4192 of Lecture Notes in Computer Science., Springer (2006) 12–21
4. Kale, L.V., Kumar, S., Vardarajan, K.: A Framework for Collective Personalized Communication. In: Proceedings of IPDPS'03, Nice, France (April 2003)
5. Dubey, A., Tessera, D.: Redistribution strategies for portable parallel FFT: a case study. Concurrency and Computation: Practice and Experience **13**(3) (2001) 209–220
6. Kanevsky, A., Skjellum, A., Rounbehler, A.: MPI/RT - an emerging standard for high-performance real-time systems. In: HICSS (3). (1998) 157–166
7. Hoefler, T., Gottschling, P., Rehm, W., Lumsdaine, A.: Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. In: Recent Advantages in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI User's Group Meeting, Proceedings, LNCS 4192, Springer (9 2006) 374–382
8. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and performance analysis of non-blocking collective operations for mpi. In: Submitted to Supercomputing'07. (2007)
9. Adelmann, A., A. Bonelli and, W.P.P., Ueberhuber, C.W.: Communication efficiency of parallel 3d ffts. In: High Performance Computing for Computational Science - VECPAR 2004, 6th International Conference, Valencia, Spain, June 28-30, 2004, Revised Selected and Invited Papers. Volume 3402 of Lecture Notes in Computer Science., Springer (2004) 901–907
10. Calvin, C., Desprez, F.: Minimizing communication overhead using pipelining for multidimensional fft on distributed memory machines (1993)
11. Goedecker, S., Boulet, M., Deutsch, T.: An efficient 3-dim FFT for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes. Computer Physics Communications **154** (August 2003) 105–110
12. Hoefler, T., Squyres, J., Bosilca, G., Fagg, G., Lumsdaine, A., Rehm, W.: Non-Blocking Collective Operations for MPI-2. Technical report, Open Systems Lab, Indiana University (08 2006)
13. LibNBC: http://www.unixer.de/NBC (2006)
14. Hoefler, T., Lumsdaine, A.: Design, Implementation, and Usage of LibNBC. Technical report, Open Systems Lab, Indiana University (08 2006)
15. Vadhiyar, S.S., Fagg, G.E., Dongarra, J.: Automatically tuned collective communications. In: Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), Washington, DC, USA, IEEE Computer Society (2000) 3
16. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (September 2004)
17. Alverson, R.: Red storm. In: Invited Talk, Hot Chips 15. (2003)
18. Brightwell, R., Hudson, T., Maccabe, A.B., Riesen, R.: The portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories (1999)