Non-Blocking Collective Operations for MPI-3

Torsten Hoefler[1,2] and Andrew Lumsdaine[1]

[1]Open Systems Laboratory
Indiana University
501 N. Morton Street
Bloomington, IN 47404 USA
{htor,lums}@cs.indiana.edu

[2]Dept. of Computer Science
Technical University of Chemnitz
Strasse der Nationen 62
Chemnitz, 09107 GERMANY
htor@cs.tu-chemnitz.de

11th January 2008

Version 1.1

Abstract

We propose new non-blocking interfaces for the collective group communication functions defined in MPI-1 and MPI-2. This document is meant as a standard extension and written in the same way as the MPI standards. It covers the MPI-API as well as the semantics of the new operations.

1 Introduction

This document is designed to serve as a discussion gound for the inclusion of non-blocking collective operations into the MPI standard. It was shown by several groups that non-blocking collectives are a viable addition to the MPI standard.

1.1 Contributors

This section alphabetically lists people who have contributed helpful comments or suggestions to the design of this document and support the standardization of non-blocking collective operations.

Christian Bell, QLogic
Brad Benton, IBM
Gil Bloch, Mellanox Technologies
Ron Brightwell, Sandia National Labs
George Bosilca, University of Tennessee
Andrew Friedley, Indiana University
Peter Gottschling, Indiana University

Rich Graham, Oak Ridge National Lab
Florian Lorenzen, Freie Universität Berlin
Frank Mietke, Technical University of Chemnitz
Torsten Mehlan, Technical University of Chemnitz
Wolfgang Rehm, Technical University of Chemnitz
Jeffrey M. Squyres, Cisco Systems

1.2 Background

Non-blocking collective operations are not included in the current Message Passing Interface (MPI, [8]) standard. The Journal of Development (JoD, [9]), a compilation of ideas that were considered but ultimately not included in the standard, documents "split collectives". Split collectives offer some of the benefits of the non-blocking collective operations proposed here, but are somewhat limited in their applicability. For example, while they enable overlapping of computation and communication for collective operations, they do not allow multiple outstanding collective operations on the same communicator or matching with blocking collective operations. These limitations were recognized by IBM and in response they designed a more fully-functional interface for their Parallel Environment (PE). Unfortunately, this interface was only implemented in the PE and applications using this interface were not portable. MPI users value portability and so the IBM implementation was discontinued in the latest version of the PE due to low usage. The MPI/RT standard [6] offers non-blocking semantics for collective operations but the state of the project is unclear. In this document we define a new interface that has the same advantages of the IBM interface and we provide a reference implementation to ensure portability. Details about our implementation and results gathered with several applications can be found in [3, 2, 1, 5, 4].

1.3 A new Approach

Our new interface fits the programmer's needs much more naturally to the existing MPI standard, even if the MPI implementation gets more complicated (e.g. has to handle proper nesting). Our API design is derived from the current MPI API design. We use the same MPI_REQUEST objects in our interface as are used in the MPI standard for non-blocking point-to-point operations and we offer similar semantics like the blocking collective operations.

We relax the semantics of the currently defined blocking collectives such that more than one collective operation can be active on a given communicator. This introduces ordering and matching issues similar to point-to-point communication. We decided against the use of tags to remain close to the existing collective operations. The matching of those operations is ruled by the order in which the calls are issued. The same will be true for their non-blocking counterparts where operations match only with similar operations (i.e., MPI_Gather matches only with other MPI_Gather calls and not others, e.g., MPI_Barrier).

1.4 Reference Implementation

We tested our interface design with a reference implementation called LibNBC (`http://www.unixer.de/NBC`) that is publicly available [3]. We optimized several codes [1, 5] for collective communication computation overlap and compared different implementation options [2].

1.5 Organization of the Document

The following section defines special terms used throughout the document. Section 2 introduces the newly proposed interface for non-blocking collectives.

1.6 Terms

A basic differentiation has to be made between non-blocking collectives, which define a non-blocking interface, and the different progress types. We define two progress types for collective operations in general:

Synchronous Progress Progress that is only made when the user thread enters the MPI library (e.g. with calls to MPI_WAIT, MPI_TEST).

Asynchronous Progress Progress that is made independently of the user program (e.g., a separate communication thread is used or the hardware supports collective communication offload).

2 Non Blocking Collective Operations

Many applications benefit from overlapping communication and computation using non-blocking MPI point-to-point operations. The same mechanism can be applied to collective operations which are defined in a blocking manner in the MPI standard. For example a parallel 3D Fast Fourier Transformation could overlap the often-used and scalability limiting MPI_ALLTOALL operation with local calculation to utilize the architecture more efficiently.

Additionally, these applications benefit from avoiding a phenomenon that we call pseudo-synchronization, which is introduced with most blocking collective operations. A collective operation is finished on a given process as soon as its part of the overall communication is done and the communication buffer can be accessed. This does not indicate that other processes have completed, or for that matter even started the collective operation. However, most algorithms introduce a synchronization due to data dependencies (it is obvious that every process has to wait for the root process in a MPI_BCAST). The application waiting time in blocking collective calls results from the pseudo-synchronization and it limits the scalability of highly parallel MPI codes. Non-blocking collective operations allow to perform the pseudo-synchronizing collective operation in the background and so would allow some limited asynchronism and load imbalance between processes.

We define a new interface, similar to the non-blocking point-to-point interface. We do not use a tag because all collective operations must follow the ordering rules for collective calls. This means that the user has to ensure proper ordering (especially in threaded environments).

A call to a non-blocking barrier would look like:

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

The MPI_IBARRIER call returns a request (similar to non-blocking point-to-point communication) that can be used as any MPI_REQUEST with MPI_WAIT and MPI_TEST. The user might need to call MPI_TEST to progress the collective operation in the background (especially in non-threaded environments), otherwise the whole collective might be performed blocking in the according MPI_WAIT without any possibility of overlapping.

2.1 General Rules for Non-Blocking Collective Communication

This section defines common rules for all non-blocking collective operations:

- Non-blocking collective communications can be nested on a single communicator. However, the MPI implementation may limit the number of outstanding non-blocking collectives to some arbitrary number. If a new non-blocking communication gets started, and the MPI library has no free resources, it fails and raises an exception.
- The send buffer must not be changed for an outstanding non-blocking collective operation, and the receive buffer must not be read until the operation is finished (e.g, after MPI_WAIT).
- All request administration functions (MPI_CANCEL, MPI_TESTALL, MPI_TESTANY, MPI_Request_free ...) described in Section 3.7 of the MPI-1.1 [7] standard are supported for non-blocking collective communications.
- The order of issued non-blocking collective operations defines the matching of them (compare the ordering rules for collective operations in the MPI-1.1 standard).
- Non-blocking collective operations and blocking collective operations can match each other.
- progress is defined similar as for non-blocking point-to-point in the MPI-2 standard

2.2 Example Routines

This section describes some routines in the style of the MPI standard. Not all routines are explained explicitely due to the similarity to the MPI-standardized ones. The new features are summarized in "Other Collective Routines".

2.2.1 Barrier Synchronization

MPI_IBARRIER(comm, request)

IN	`comm`	communicator (handle)
OUT	`request`	request (handle)

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request* request)
```

```
void MPI::Comm::Ibarrier(MPI::Request *request) const = 0
```

MPI_IBARRIER(COMM, REQUEST, IERR)
INTEGER COMM, IERROR, REQUEST

MPI_IBARRIER initializes a barrier on a communicator. MPI_WAIT may be used to block until it is finished.

> *Advice to users.* A non-blocking barrier sounds unusable because MPI_BARRIER is defined in a blocking manner to protect critical regions. However, there are codes that may move independent computations between the MPI_IBARRIER and the subsequent Wait/Test call to overlap the barrier latency.
>
> *Advice to implementers.* A non-blocking barrier can be used to hide the latency of the MPI_BARRIER operation. This means that the implementation of this operation should incur only a low overhead (CPU usage) in order to allow the user process to take advantage of the overlap.

2.2.2 Broadcast

MPI_IBCAST(buffer, count, datatype, root, comm, request)

INOUT	`buffer`	starting address of buffer (choice)
IN	`count`	number of elements in buffer (integer)
IN	`datatype`	data type of elements of buffer (handle)
IN	`root`	rank of the broadcast root (integer)
IN	`comm`	communicator (handle)
OUT	`request`	request (handle)

```
int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root,
MPI_Comm comm, MPI_Request* request)
```

```
void MPI::Comm::Ibcast(void* buffer, int count, const MPI::Datatype& datatype,
int root, MPI::Request *request) const = 0
```

MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERR)
<type> BUFFER(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR, REQUEST

Advice to users. A non-blocking broadcast can efficiently be used with a technique called "double buffering". This means that a usual buffer in which a calculation is performed will be doubled in a communication and a computation buffer. Each time step has two independent operations - communication in the communication buffer and computation in the computation buffer. The buffers will be swapped (e.g. with simple pointer operations) after both operations have finished and the program can enter the next round. Valiant's BSP model [10] can be easily changed to support non-blocking collective operations in this manner.

2.2.3 Gather

MPI_IGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, request)

IN	`sendbuf`	starting address of send buffer (choice)
IN	`sendcount`	number of elements in send buffer (integer)
IN	`sendtype`	data type of sendbuffer elements (handle)
OUT	`recvbuf`	starting address of receive buffer (choice, significant only at root)
IN	`recvcount`	number of elements for any single receive (integer, significant only at root)
IN	`recvtype`	data type recv buffer elements (handle, significant only at root)
IN	`root`	rank of receiving process (integer)
IN	`comm`	communicator (handle)
OUT	`request`	request (handle)

```
int MPI_Igather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
MPI_Request* request)
```

```
void MPI::Comm::Igather(const void* sendbuf, int sendcount,
const MPI::Datatype& sendtype, void* recvbuf, int recvcount,
const MPI::Datatype& recvtype, int root, MPI::Request *request) const = 0
```

```
MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,RECVTYPE,
ROOT, COMM, REQUEST, IERR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR,
REQUEST
```

2.2.4 Other Collective Routines

All other defined collective routines can be executed in a non-blocking manner as shown above. The operation MPI_<OPERATION> is renamed to MPI_I<OPERATION> and a request-reference is added as last element to the argument list. All collective routines are shown in Table 2.2.4.

General advice to users. Non-blocking collective operations can be used to avoid explicit application synchronization and to overlap communication and computation in programs. A common scheme for this would be "double buffering" (explained in Section 2.2.2) which can easily be used to optimize programs written in the BSP model.

MPI_IBARRIER	MPI_IBCAST
MPI_IGATHER	MPI_IGATHERV
MPI_ISCATTER	MPI_ISCATTERV
MPI_IALLGATHER	MPI_IALLGATHERV
MPI_IALLTOALL	MPI_IALLTOALLV
MPI_IALLTOALLW	MPI_IREDUCE
MPI_IALLREDUCE	MPI_IREDUCE_SCATTER
MPI_ISCAN	MPI_IEXSCAN

Table 1: Proposed non-blocking collective functions

General advice to implementers. Most non-blocking operations will be used to overlap communication with computation. The implementation of these operations should cause as low CPU overhead as possible to free the CPU for the user process.

2.3 Environment and Limits

The number of outstanding (nested) non-blocking collective operations may be limited, especially on hardware supported implementations. A new attribute, called MPI_ICOLL_MAX_OUTSTANDING is attached to MPI_COMM_WORLD. The user can access this attribute with MPI_COMM_GET_ATTR, described in the MPI-2 Standard Chapter 8.8. MPI_ICOLL_MAX_OUTSTANDING must have the same value on all processes in MPI_COMM_WORLD.

However, the implementation should support at least 32767 outstanding operations. A software implementation could use non-blocking send-receive to enable non-blocking collective operations, where each outstanding operation uses exactly one tag value. A hardware implementation can fall back to this software implementation if its capabilities are exhausted.

Acknowledgements

The authors want to thank Laura Hopkins from Indiana University for editorial comments and extensive writing support.

References

[1] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, 9 2007.

[2] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine. A Case for Standard Non-Blocking Collective Operations. 10 2007. In proceedings of EuroPVM/MPI 2007.

[3] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *In proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, 11 2007.

[4] T. Hoefler, J. Squyres, W. Rehm, and A. Lumsdaine. A Case for Non-Blocking Collective Operations. In *Frontiers of High Performance Computing and Networking - ISPA 2006 Workshops*, volume 4331/2006, pages 155–164. Springer Berlin / Heidelberg, 12 2006.

[5] T. Hoefler and G. Zerah. Transforming the high-performance 3d-FFT in ABINIT to enable the use of non-blocking collective operations. Technical report, Commissariat a l'Energie Atomique - Direction des applications militaires (CEA-DAM), 2 2007.

[6] Arkady Kanevsky, Anthony Skjellum, and Anna Rounbehler. MPI/RT - an emerging standard for high-performance real-time systems. In *HICSS (3)*, pages 157–166, 1998.

[7] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. 1995.

[8] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1997.

[9] Message Passing Interface Forum. MPI-2 Journal of Development, July 1997.

[10] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.