Fixing Probe for Multi-Threaded MPI Applications (Revision 4)

Douglas Gregor, Torsten Hoefler, Brian Barrett, and Andrew Lumsdaine
{dgregor,htor,brbarret,lums}@osl.iu.edu

January 21, 2009

1 Introduction

MPI's message-probing operations, MPI_Probe and MPI_Iprobe, are useful in MPI applications that do not know *a priori* what messages they will receive or how much data those messages will contain. Such applications often have irregular, data-driven communication patterns or deal with data structures that require serialization for transmission.

Unfortunately, MPI's message-probing operations are unusable in multi-threaded applications where they could be most useful. The fundamental problem with MPI_Probe and MPI_Iprobe functions is that a message found by a probe can still be matched and received by a receive operation in a different thread. Thus, despite the fact that a probe operation returns a source and tag that can be used to receive a message, there is no guarantee that the message will still be available when that receive operation is invoked. For example, the following code can not be executed concurrently by two threads in an MPI process, because a message could be found by the MPI_Probe in both threads, while only one of the threads could successfully receive the message (the other will block):

```
MPI_Status status;
int value;
MPI_Probe(MPI_ANY_SOURCE, /*tag=*/0, MPI_COMM_WORLD, &status);
MPI_Recv(&value, 1, MPI_INT, status.MPI_SOURCE, /*tag=*/0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

The lack of a usable threaded MPI_Probe or MPI_Iprobe causes serious problems for the construction of language bindings for high-level object-oriented and generic languages, where users would like to be able to transmit objects that require serialization, including C++ [3,4], Java [1], Python [5], and C# [2]. MPI provides a mechanism to "pack" (serialize) an object into a buffer that can be transmitted via MPI, then "unpack" (de-serialize) that object at the receiver's end. However, while MPI provides good support for serialization and sending serialized data, it does not provide adequate support for receiving serialized data. The problem is that, in general, the receiver cannot know the length of the serialized data before it posts the receive. MPI_Probe and MPI_Iprobe provide this functionality, but they are unusable in a multi-threaded environment. Thus, bindings for these languages must resort to elaborate and inefficient workarounds [2].

There is no known workaround that addresses all of the problems with MPI_Probe and MPI_Iprobe in multi-threaded MPI applications. Therefore, we propose extensions for MPI that introduce a new kind of probe—a "matched" probe—and a set of corresponding receive operations. The new probe matches a message and returns a handle to that specific message, which cannot be found by any other probe operation or matched by any other receive. The new receive operations allow the receipt of a message based on the message handle returned from this probe. These extensions allow the use of probe in a multi-threaded context, ensuring that the message found by probe is the message received.

In the following example, we illustrate how the new probe operation, MPI_Mprobe, can be used to receive a message of unknown length. Note that this code can be concurrently executed in several threads, each of which will receive different messages.

```
MPI_Message msg;
MPI_Status status;
/* Match a message */
MPI_Mprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &msg, &status);

/* Allocate memory to receive the message */
int count;
MPI_get_count(&status, MPI_BYTE, &count);
char* buffer = malloc(count);

/* Receive this message. */
MPI_Mrecv(buffer, count, MPI_BYTE, &msg, MPI_STATUS_IGNORE);
```

2 Implementation Experience

A prototype implemented in Open MPI took under a week and likely could be considered production ready. Implementations with unexpected queue search exposed in the MPI library will have little difficulty with the implementation of this proposal. The Mprobe follows the same code path through the implementation as a normal probe, but removes the message fragment from the unexpected queue on match and stashes it in the MPI_Message handle. The MPI_Mrecv then restarts the fragment through the progress engine, as a matched fragment would be started during MPI_Recv on an unexpected message. There is one extra **if** statement in the probe matching path of Open MPI, which is negligible when compared to walking the unexpected queue.

Implementations over libraries such as PSM and MX, which hide the unexpected queue in the lower library, are more problematic. They will require exposing a similar semantic to the Mprobe/Mrecv in their API.

3 Proposed Extensions

3.1 Message handles

A message handle returned by a matching probe (MPI_Mprobe or MPI_Improbe) has type MPI_Message. A *null* handle is a handle with value MPI_MESSAGE_NULL.

```
int MPI_Message_cancel(MPI_Message *message)

MPI_MESSAGE_CANCEL(MESSAGE, IERROR)
INTEGER MESSAGE, IERROR

void Message::Cancel()
```

INOUT message the message to be cancelled (Message)

A call to MPI_MESSAGE_CANCEL cancels the receipt of a message matched by a matching probe. A cancelled message cannot be received.

One is allowed to call MPI_MESSAGE_CANCEL with a null message argument. In this case the operation returns immediately.

Advice to implementers. Because no receive buffers have been posted for a receive, cancellation always succeeds even if the underlying interconnect does not permit the cancellation of transmissions after they have been matched. A valid implementation of MPI_Message_cancel that supports such interconnects is:

```
int count; /* set to the size of the message */
char* buffer = malloc(count);
MPI_Mrecv(buffer, count, MPI_BYTE, &msg, MPI_STATUS_IGNORE);
free(buffer);
```
□

3.2 Matching Probe

The MPI_MPROBE and MPI_IMPROBE operations allow incoming messages to be queried without actually receiving them. The user can then decide how to receive them, based on the information returned by the probe. In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

```
int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag, MPI_Message *message,
                MPI_Status *status)

MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)
LOGICAL FLAG
INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR

bool Comm::Improbe(int source, int tag, Message& message) const
bool Comm::Improbe(int source, int tag, Message& message, Status& status) const
```

IN source source rank or MPI_ANY_SOURCE (integer)
IN tag tag value or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT flag (logical)
OUT message message handle (Message)
OUT status status object (Status)

MPI_IMPROBE(source, tag, comm, flag, message, status) returns flag = **true** if there is a message that can be received and that matches the pattern specified by the arguments source, tag, and comm. The call matches the same message that would have been received by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point in the program, and returns in message a handle to that message and in status the same value that would have been returned by MPI_RECV(). Otherwise, the call returns flag = **false**, and leaves message undefined.

A matched receive executed with the message handle will receive the message that was matched by the probe. Unlike MPI_IPROBE, no other probe or receive operation may match the message returned by MPI_IMPROBE. Each message returned by MPI_IMPROBE must either be completed with MPI_MRECV, MPI_IMRECV, or cancelled with MPI_MESSAGE_CANCEL.

The source argument of MPI_IMPROBE can be MPI_ANY_SOURCE, and the tag argument can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source and/or with an arbitrary tag. However, a specific communication context must be provided with the comm argument.

When a matched probe operation matches a synchronous-mode send, the synchronous send can only complete once the corresponding matched receive operation has begun execution.

A matched probe is not a receive. Therefore, a ready send that matches an MPI_IMPROBE operation is erroneous and its outcode is undefined.

A matched probe with MPI_PROC_NULL as source returns flag = **true** and MPI_MESSAGE_NULL.

```
int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message
               MPI_Status *status)

MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)
INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR

void Comm::Mprobe(int source, int tag, Message& message) const
void Comm::Mprobe(int source, int tag, Message& message, Status& status) const
```

IN source source rank or MPI_ANY_SOURCE (integer)
IN tag tag value or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT message message handle (Message)
OUT status status object (Status)

MPI_MPROBE behaves like MPI_IMPROBE except that it is a blocking call that returns only after a matching message has been found.

Advice to users. Unlike the (deprecated) MPI_PROBE and MPI_IPROBE, MPI_MPROBE and MPI_IMPROBE can be safely used in a multi-threaded MPI program. A message returned by MPI_MPROBE or MPI_IMPROBE has already been matched, and can only be received with a matched receive (section 3.3) executed with the corresponding message handle. ☐

The MPI implementation of MPI_MPROBE and MPI_IMPROBE needs to guarantee progress: if a call to MPI_MPROBE has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to MPI_MPROBE will return, unless the message is matched by a concurrent matching probe operation or received by another concurrent receive operation (that is executed by another thread at the probing process). Similarly, if a process busy waits with MPI_IMPROBE and a matching message has been issued, then the call to MPI_IMPROBE will eventually return flag = **true** unless the message is matched by a concurrent matching probe operation or received by another concurrent receive operation.

Editorial note: the definitions of MPI_IPROBE and MPI_Probe should remain the same as they are now, but we deprecate them by adding the following text:

MPI_PROBE and MPI_IPROBE are deprecated.

Rationale. MPI_PROBE and MPI_IPROBE find messages, but do not match them, which makes MPI_PROBE and MPI_IPROBE unusable in multi-threaded MPI programs. MPI_MPROBE and MPI_IMPROBE provide better semantics than MPI_PROBE and MPI_IPROBE for multi-threaded MPI programs. ☐

3.3 Matched receives

Messages that have been matched by a matching probe (section 3.2) can be received by a matched receive.

```
int MPI_Mrecv(void* buf, int count, MPI_Datatype datatype, MPI_Message* message,
              MPI_Status *status)

MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR

void Message::Mrecv(void* buf, int count, const Datatype& datatype, Status& status)
void Message::Mrecv(void* buf, int count, const Datatype& datatype)
```

OUT buf initial address of receive buffer (choice)
IN count number of elements in receive buffer (integer)
IN datatype datatype of each receive buffer element (handle)
INOUT message message to be received (Message)
OUT status status object (Status)

This call receives a message found by a matching probe operation (section 3.2).

The receive buffer consists of the storage containing count consecutive elements of the type specified by datatype, starting at address buf. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer.

If a message that is shorter than the receive buffer arrives, then only those locations corresponding to the (shorter) message are modified.

On return from this function, the message handle is set to MPI_MESSAGE_NULL. All errors that occur during the execution of this operation are handling according to the error handler set for the communicator used in them matched probe call that produced the message handle.

If the message argument MPI_MESSAGE_NULL, the call returns immediately with the status object set to source = MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0, as if a receive from MPI_PROC_NULL was issued.

Example The following example uses a matching probe and a matched receive to receive any message of any size. This code can be executed in multiple threads concurrently.

```
MPI_Message message;
MPI_Status status;
/* Match a message */
MPI_Mprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &message, &status);

/* Allocate memory to receive the message */
int count;
MPI_Get_count(&status, MPI_BYTE, &count);
char* buffer = malloc(count);

/* Receive this message. */
MPI_Mrecv(buffer, count, MPI_BYTE, &message, MPI_STATUS_IGNORE);
```
□

Rationale. MPI_MRECV does not have a communicator parameter because the communicator was part of the matching probe operation. Requiring the communicator to also be passed into MPI_MRECV would involve addition user code and additional error checking in the MPI implementation, with no clear benefit. □

```
int MPI_Imrecv(void* buf, int count, MPI_Datatype datatype, MPI_Message* message,
               MPI_Request *request)

MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR

void Message::Imrecv(void* buf, int count, const Datatype& datatype, Request& request)
```

OUT buf initial address of receive buffer (choice)
IN count number of elements in receive buffer (integer)
IN datatype datatype of each receive buffer element (handle)
INOUT message message to be received (Message)
OUT request request object (Status)

Start a matched, non-blocking receive of a message found by a matching probe operation (section 3.2). The message handle is set to MPI_MESSAGE_NULL.

If the message argument MPI_MESSAGE_NULL, the call returns immediately with the status object set to source = MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0, as if a receive from MPI_PROC_NULL was issued.

3.4 Error Codes and Classes

Add the following to the table of error codes:

MPI_ERR_MESSAGE Invalid message argument

4 Revision History

Revision 4

- added description for the case thet MPI_PROC_NULL is passed as source to MPI_Mprobe.

Revision 3

- Clarify the interaction between a matched probe, its receive, and a synchronous send.
- Added MPI_ERR_MESSAGE, and clarified the behavior of each of the new functions when given MPI_MESSAGE_NULL.
- Clarified that a ready send is erroneous if it matches an MPI_IMPROBE or MPI_MPROBE.

Revision 2

- Added MPI_MESSAGE_NULL.
- Clarified that errors in the receive operations are handled according to the communicator with which the message handle is associated.

Revision 1

- Renamed MPI_Rprobe, MPI_Irprobe, MPI_Rrecv and MPI_Irrecv to MPI_Mprobe, MPI_Improbe, MPI_Mrecv, and MPI_Imrecv, respectively.
- Made MPI_Message an opaque handle; MPI_Mprobe and MPI_Improbe now return an MPI_Status result to provide information about the message (source, tag, count). Removed MPI_Message_get_count.
- Corrected numerous small errors in the C++ bindings.

References

[1] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object serialization for marshalling data in a Java interface to MPI. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 66–71, New York, NY, USA, 1999. ACM Press.

[2] Douglas Gregor and Andrew Lumsdaine. Design and implementation of a high-performance MPI for C# and the common language infrastructure. In *Proceedings ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008. To appear.

[3] Douglas Gregor and Matthias Troyer. Boost.MPI. `http://www.generic-programming.org/~dgregor/boost.mpi/doc/`, November 2006.

[4] Prabhanjan Kambadur, Douglas Gregor, Andrew Lumsdaine, and Amey Dharurkar. Modernizing the C++ interface to mpi. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, LNCS, pages 266–274, Bonn, Germany, September 2006. Springer.

[5] Patrick Miller and Martin Casado. MPI Python. `http://sourceforge.net/projects/-pympi/`.