Fixing Probe for Multi-Threaded MPI Applications

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

April 28, 2008

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);
```

There is no known workaround that addresses all of the problems with MPI_Probe and MPI_Iprobe in multi-threaded MPI applications. For certain aspects of the problem (e.g., receiving data of unknown size), there are workarounds, but they are inefficient and verbose. Therefore, we propose extensions for MPI-3 that introduce a new kind of 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_Rprobe, 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;
/* Match a message */
MPI_Rprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &msg);

/* Allocate memory to receive the message */
MPI_Aint count;
MPI_Message_get_count(&msg, MPI_BYTE, &count);
char* buffer = malloc(count);

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

2 Proposed Extensions

2.1 Message handles

A message handle returned by a matching probe (MPI_Rprobe or MPI_Irprobe) has type MPI_Message.

In C, the message is a structure that contains two fields named MPI_SOURCE and MPI_TAG; the structure may contain additional fields. Thus, message.MPI_SOURCE and message.MPI_TAG contain, respectively, the source and tag of the matched message.

In Fortran, the message is an array of INTEGERs of size MPI_MESSAGE_SIZE. The constants MPI_SOURCE and MPI_TAG are the indices of the entries that store the source and tag fields. Thus, message(MPI_SOURCE) and message(MPI_TAG) contain, respectively, the source and tag of the matched message.

In C++, the message is a class MPI::Message with member functions used to access the source and tag of the matched messages.

```
int Message::Get_source() const
int Message::Get_tag() const
```

The message argument also returns information on the length of the message matched. However, this information is not directly available as a field of the message variable and a call to MPI_MESSAGE_GET_COUNT is required to "decode" this information.

```
int MPI_Message_get_count(MPI_Message *message, MPI_Datatype datatype, MPI_Aint *count)

MPI_MESSAGE_GET_COUNT(MESSAGE, DATATYPE, COUNT, IERROR)
INTEGER MESSAGE(MPI_MESSAGE_SIZE), DATATYPE, COUNT, IERROR

MPI_Aint Message::Get_count(const Datatype& type) const
```

IN message matched message handle (Message)
IN datatype datatype of each buffer entry (handle)
OUT count number of entries in the message (integer)

Returns the number of entries in the message. (Again, we count entries, each of type datatype, not bytes.)

```
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.

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:

```
MPI_Aint count;
MPI_Message_get_count(&message, MPI_BYTE, &count);
char* buffer = malloc(count);
MPI_Rrecv(buffer, count, MPI_BYTE, &msg, MPI_STATUS_IGNORE);
free(buffer); □
```

Rationale. Although a message handle has similar fields and usage to the status structure, the differences between the two are significant enough to warrant having different names. The message handle does not require error information (MPI_ERROR), since errors will never be reported in the message; however, it does require extra hidden information that status does not require, e.g., an internal handle to a message that has been matched but not yet received. More importantly, message handles are used in different ways from status structures: status structures describe just the characteristics of a communication (source, tag, size, whether it was cancelled, etc.), whereas message handles represent and manipulate the communication itself. □

2.2 Matching Probe

The MPI_RPROBE and MPI_IRPROBE operations allow incoming messages to be checked for, 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_Irprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Message *message)

MPI_IRPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, IERROR)
LOGICAL FLAG
INTEGER SOURCE, TAG, COMM, MESSAGE(MPI_MESSAGE_SIZE), IERROR

bool Comm::Irprobe(int source, int tag, Message& message) 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)

MPI_IRPROBE(source, tag, comm, flag, 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. Otherwise, the call returns flag = **false**, and leaves message undefined.

If MPI_IRPROBE returns flag = **true**, then the content of the message object can be subsequently accessed as described in section 2.1 to find the source, tag and length of the matched message.

A ready 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_IRPROBE. Each message returned by MPI_IRPROBE must either be completed with a a ready receive or cancelled with MPI_MESSAGE_CANCEL.

The source argument of MPI_IRPROBE 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.

int MPI_Rprobe(**int** source, **int** tag, MPI_Comm comm, MPI_Message *message)

MPI_RPROBE(SOURCE, TAG, COMM, MESSAGE, IERROR)
INTEGER SOURCE, TAG, COMM, MESSAGE(MPI_MESSAGE_SIZE), IERROR

void Comm::Rprobe(**int** source, **int** tag, Message& message) **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)

MPI_RPROBE behaves like MPI_IRPROBE 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_RPROBE and MPI_IRPROBE can be safely used in a multi-threaded MPI program. A message returned by MPI_RPROBE or MPI_IRPROBE has already been matched, and can only be received with a ready-receive operation (section 2.3) executed with the corresponding message handle. □

The MPI implementation of MPI_RPROBE and MPI_IRPROBE needs to guarantee progress: if a call to MPI_RPROBE has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to MPI_RPROBE 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_IRPROBE and a matching message has been issued, then the call to MPI_IRPROBE 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_RPROBE and MPI_IRPROBE provide better semantics than MPI_PROBE and MPI_IPROBE for multi-threaded MPI programs. □

2.3 Ready receives

Messages that have been matched by a matching probe (section 2.2) can be received by a ready receive.

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

MPI_RRECV(BUF, COUNT, DATATYPE, MESSAGE, COMM, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, MESSAGE(MPI_MESSAGE_SIZE), STATUS(MPI_STATUS_SIZE),
        IERROR

void Message::Recv(void* buf, MPI_Aint count, const MPI_Datatype& datatype,
                   MPI_Status& status)
void Message::Recv(void* buf, MPI_Aint count, const MPI_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 2.2).

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.

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

```
MPI_Message message;
/* Match a message */
MPI_Rprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &message);

/* Allocate memory to receive the message */
MPI_Aint count;
MPI_Message_get_count(&message, MPI_BYTE, &count);
char* buffer = malloc(count);

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

Rationale. MPI_RRECV 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_RRECV would involve addition user code and additional error checking in the MPI implementation, with no clear benefit. □

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

MPI_IRRECV(BUF, COUNT, DATATYPE, MESSAGE, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, MESSAGE(MPI_MESSAGE_SIZE), REQUEST, IERROR

void Message::Irrecv(void* buf, MPI_Aint count, const MPI_Datatype& datatype,
                     MPI_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 ready, non-blocking receive of a message found by a matching probe operation (section 2.2).