Discussion: MPI Point to Point Communication II

Table of Contents

1. Overview
2. Basic Deadlock
3. Determining Information about Messages
   3.1 Wait, Test, and Probe
   3.2 Status
4. Special Parameters
5. MPI Pro Implementation
6. Programming Recommendations

References □ Lab Exercise □ Quiz □ Evaluation □ Navigation Guide
1. Overview

We will start with an introduction to the programming options covered in this module:

Deadlock

*Deadlock* is an often-encountered situation in parallel processing, resulting when two or more processes are in contention for the same set of resources. In communications, a typical scenario involves two processes wishing to exchange messages, but both trying to give theirs to the other while neither is yet ready to accept one. A number of strategies will be described to help insure against this sort of thing occurring in the application.

Checking and acting on communications-"state"

When involved in non-blocking transactions, the calls *wait*, *test*, and *probe* make it possible for the application to query the status of particular messages, or to check on any that meet a certain set of characteristics, without necessarily taking any completed transactions out of the queue.

Checking the information returned from a transaction (status) allows the application to, for example, take corrective action if an error has occurred.

Using special parameters for special cases

**Wildcards - MPI_ANY_SOURCE and MPI_ANY_TAG**

Use of these calls can allow the receiving process to specify receipt of a message using a wildcard for either the sender or for the kind of message.

**Null Processes and Requests**

Special null parameters can simplify applications with regular data structures.

2. Basic Deadlock

Deadlock is a phenomenon most common with blocking communication. It occurs when all tasks are waiting for events that haven’t been initiated yet.

The following diagram represents two SPMD tasks: both are calling blocking standard sends at the same point of the program. Each task’s matching receive occurs later in the other task’s program.

```
MPI_SEND

A ----> ----> ----> ----> ----> ----> ----> ---->

B ----> ----> ----> ----> ----> ----> ----> ---->
```

A simple example of deadlock is one in which the two sends are each waiting on their corresponding receives in order to complete, but those receives are executed **after** the sends, so if the sends do not complete and return, the receives can never be executed, and both sets of communications will stall indefinitely.

A more complex example of deadlock can occur if the message size is greater than the threshold; deadlock will occur because neither task can synchronize with its matching receive. If the message size is less than or equal to the threshold, deadlock can still occur if insufficient system buffer space is available. Both tasks will be waiting for a receive to draw message data out of the system buffer, but these receives cannot be executed because both tasks are blocked at the send.

**Solutions for avoiding deadlock:**

There are four ways to avoid deadlock:

1. **different ordering of calls between tasks**

   Arrange for one task to post its receive first and for the other to post its send first. That clearly establishes that the message in one direction will precede the other.
2. **non-blocking calls**

   Have each task post a non-blocking receive before it does any other communication. This allows each message to be received, no matter what the task is working on when the message arrives or in what order the sends are posted.

3. **MPI_Sendrecv**
   **MPI_Sendrecv_replace**

   Use MPI_Sendrecv (S). This is an elegant solution. In the _replace (S) version, the system allocates some buffer space (not subject to the threshold limit) to deal with the exchange of messages.

4. **buffered mode**

   Use buffered sends so that computation can proceed after copying the message to the user-supplied buffer. This will allow the receives to be executed. Buffered sends were discussed earlier, in MPI Point to Point Communication I.

---

### 3. Determining Information about Messages

Applications can use specific calls to control how they deal with incomplete transactions, or transactions whose state is not known, without necessarily having to complete the operation (this is analogous to wanting to know whether or not your aunt has written you, but not particularly caring what she had to say). The application can be programmed to be aware of the state of its communications, and thereby act intelligently in different situations:

#### 3.1 Wait, Test, and Probe

Having successfully decoupled your communications from your main computational thread, you will nonetheless need to be able to both obtain information about the status of the communication transaction, and arrange to take certain actions regarding the conditions you find. The three calls described here are among the most commonly useful for dealing with these kinds of situations.
MPI _Wait

Operation of "MPI Wait"

- **Useful for both sender and receiver of non-blocking communications.**

Blocking communications involve an automatic *wait*, so you'll never see a non-trivial call to it when such operations are used. In both the send and receive cases for non-blocking operations, the calling process suspends its operation until the operation referenced by the *wait* has completed, at which time execution resumes in the calling process.

- **Receiving process blocks until message is received, under programmer control**

On the receive side, the process has already posted a non-blocking receive, which will be completed regardless of what the calling process does. The programmer is therefore able to determine whether or not any useful computation can be accomplished before the information in the not-yet-received message is required ... if there is useful work available, the application has clearly gained efficiency by doing that work while the message is still in transit. At some point, of course, the message will be needed, and a *wait* will be issued.

- **Sending process blocks until send operation completes, at which time the message buffer is available for re-use**

On the sending side, the process is also freed from the transaction,
except for the fact that it is constrained from doing any more writing to that particular buffer until its current use is completed ... if the sender attempts to put some other message into that buffer before the preceding transmission completes, the results are indeterminate and based solely on what part of the process was in train when the overwriting occurred. Doing a wait on the message guarantees that re-use will not be destructive.

- **MPI_Test**

  ![Operation of "MPI Test"

<table>
<thead>
<tr>
<th>Compute</th>
<th>&quot;MPI Test&quot;</th>
<th>Compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;MPI Isend&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute</td>
<td></td>
<td>Compute</td>
</tr>
<tr>
<td>Receive</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  - **Used for both sender and receiver of non-blocking communication.**

    Where wait suspends execution until an operation completes, test returns immediately with information about its current status.

  - **Receiver checks to see if a specific sender has sent a message that is waiting to be delivered ... messages from all other senders are ignored.**

    The test call will return true only in the case that the sender specified in the object has sent a message which is currently in the queue for delivery; traffic from all other sources is ignored.

  - **Sender can find out if the message-buffer can be re-used ... have to wait until operation is complete before doing so.**
On the sender side, *test* is the non-blocking analog to *wait*, giving the application knowledge of the current state without requiring it to block until completion, thus allowing the application to do other work, if any exists.

- **MPI_Probe**

Receiver is notified when (i.e., this is a blocking call) messages from potentially *any* sender arrive and are ready to be processed.

The previous calls all targeted specific messages and senders; the *probe* call can be tailored to return "deliverable" information regarding messages from any sender, as well as from specific ones.

### 3.2 Status

**status** returns source, tag, error (standard)

*Status* is the object at which to look to determine information on the message source and tag, and any error incurred on the communication call. In Fortran these are returned as status(MPI_SOURCE), status(MPI_TAG), and status(MPI_ERROR) respectively. In C they are returned as status.MPI_SOURCE,
status.MPI_TAG, and status.MPI_ERROR. In the case of sends, the status information is probably not different from what had been in the request, so is of little use.

- **MPI_Get_count** returns number of elements

  This can be useful if you've allocated a receive buffer that may be larger than the incoming message, or if you want to learn the length of a message identified with MPI_Probe (S).

- **Circumstances in which it makes sense to check the status**

  The information made available by status comes in very handy when dealing with the following situations, offered without discussion as the various particulars have already been covered elsewhere (except for MPI_ANY_TAG and MPI_ANY_SOURCE, which will be covered shortly):

  - blocking receive or wait, when MPI_ANY_TAG or MPI_ANY_SOURCE has been used
    - MPI_ANY_TAG, accept a message with any tag value
    - MPI_ANY_SOURCE, accept a message from any source
  - MPI_Probe or MPI_Iprobe to obtain information on incoming messages
    - MPI_Probe - Blocking test for a message
    - MPI_Iprobe - Nonblocking test for a message
  - MPI_Test to learn if the communication has completed

---

### 4. Special Parameters

**Wildcards - MPI_ANY_SOURCE and MPI_ANY_TAG**
A generic term meaning "anything meeting a very general set of characteristics." MPI_ANY_SOURCE allows the receiver to get messages from any sender, and MPI_ANY_TAG allows the receiver to get any kind of message from a sender.

**Null Processes and Requests**

Applications often deal with *regular* data structures, like rectangular hyper-arrays, and perform the same kind of communication everywhere within them, except for at the edges, where special code has to be written in order not to communicate where there are no valid "neighbors" to receive, or from whom to receive; special null parameters move the logic for this out of user-code and into the system, simplifying the application.

5. **MPI Pro Implementation**

In the MPI Pro implementation, the standard, synchronous and buffered modes work as specified by the standard.

However, in the MPI Pro implementation, the ready mode always succeeds. In effect, the ready mode is equivalent to the standard send. The reason for this is that there is virtually no way of improving the performance based on knowing that the receiver has already posted the receive request by the time the send message arrives.

Therefore the only specific thing in MPI/Pro is that no matter how hard the programmer may try to cause an error using MPI_Rsend by making sure that the receive request is posted late, they will not succeed.

6. **Programming Recommendations**

Avoid deadlock by intelligent arrangement of sends/receives, or by posting non-blocking receives early.

If you choose to use *blocking* transactions, try to guarantee that deadlock
will be avoided by carefully tailoring your communication strategy so that sends and receives are properly paired and in the necessary order; alternatively, post non-blocking receives as early as possible, so that the sends will stay in the system for as little time as is necessary.

**Use the appropriate "operation-status" call ("wait", "test", or "probe") to control the operation of non-blocking communications calls.**

Correctly knowing the state of communications transactions allows the application to intelligently steer itself to better efficiency in its use of available cycles. Ultimately pending traffic must be accepted, but that action can be long in coming and much can possibly be accomplished while it is incomplete. The *wait, test* and *probe* calls allow the application to match the appropriate activity with the particular situation.

**Check the value of the "status" fields for problem reports.**

Don't just assume that things are running smoothly -- make it a general rule that every transaction is checked for success, and that failures are promptly reported and as much related information as possible is developed and made available for debugging.

**Intelligent use of wildcards can greatly simplify logic and code.**

Using general receives, receives capable of handling more than one kind of message traffic (in terms of either sender, or message-type, or both), can greatly simplify the structure of your application, and can potentially save on system resources (if you are in the habit of using a unique message buffer for each transaction).

**Null processes and requests move tests out of user code.**

Use of *MPI_PROC_NULL* and *MPI_REQUEST_NULL* does not get rid of boundary tests, it simply allows the programmer to use a call that will be ignored by the system.

**References**

Gropp, W., Lusk, E. and Skjellum, A. *Using MPI. Portable Parallel Programming with the Message-Passing Interface*. The MIT Press.
Cambridge, Massachusetts.

MPI Home Page at Argonne National Labs http://www.mcs.anl.gov/mpi


---

**Quiz**

Take a multiple-choice quiz on this material, and submit it for grading.

**Exercise**

Lab exercises for MPI Point to Point Communication II

**Evaluation**

Please complete this short evaluation form. Thank you!