Discussion: MPI Point to Point Communication I

Veaceslav Zaloj

Video Introduction
View with modem or broadband connection
Read the text transcript

Table of Contents

1. Overview
2. Blocking Behavior
   2.1 Communication Modes
      2.1.1 Blocking Synchronous Send
      2.1.2 Blocking Ready Send
      2.1.3 Blocking Buffered Send
      2.1.4 Blocking Standard Send
      2.1.5 Blocking Send and Receive
      2.2 Conclusions: Modes
3. Non-Blocking Behavior
   3.1 Syntax of Non-Blocking Calls
   3.2 Example: Non-blocking standard send

http://www.tc.cornell.edu/Services/Edu/Topics/MPI/Pt2pt1/more.asp

9/23/2002
1. Overview

Point to point communication involves transmission of a message between one pair of processes, as opposed to collective communication, which involves a group of tasks. MPI features a broader range of point to point communication calls than most other message passing libraries.

In many message-passing libraries, such as PVM or MPI, the method by which the system handles messages has been chosen by the library developer. The chosen method gives acceptable reliability and performance for all possible communication scenarios. But it may hide possible programming problems or may not give the best performance in specialized circumstances. For MPI, this is equivalent to standard mode communication, which will be introduced in this module.

In MPI, more control over how the system handles the message has been given to the programmer, who selects a communication mode when they select a send routine. In addition to standard mode, MPI provides synchronous, ready, and buffered modes. This module will look at the system behavior for each mode, and discuss their advantages and disadvantages.

In addition to specifying communication mode, the programmer must decide whether send and receive calls will be blocking or non-blocking. A blocking or non-blocking send can be paired to a blocking or non-blocking receive.

Blocking suspends execution until the message buffer is safe to use. In both sending and receiving modes, the buffer used to contain the message can be an off-used resource, and data may be corrupted when it is used before an on-going transaction has completed; blocking communications insure that this never happens -- when control returns from the blocking call, the buffer can safely be modified without any danger of corrupting some other part of the process.

Non-blocking separates communication from computation. A non-blocking call
effectively guarantees that an interrupt will be generated when the transaction is ready to proceed, thus allowing the original thread to get back to computationally-oriented processing.

2. Blocking Behavior

Before moving on to the communication modes, let's review syntax for a blocking send and receive:

MPI_SEND is a blocking send. This means the call does not return control to your program until the data have been copied from the location you specify in the parameter list. Because of this, you can change the data after the call and not affect the original message. (There are non-blocking sends where this is not the case.)

- C:
  ```c
  int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  ```

- Fortran:
  ```fortran
  MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)
  ```

The parameters:
- **buf** is the beginning of the buffer containing the data to be sent. For Fortran, this is often the name of an array in your program. For C, it is an address.
- **count** is the number of elements to be sent (not bytes)
- **datatype** is the type of data
- **dest** is the rank of the process which is the destination for the message
- **tag** is an arbitrary number which can be used to distinguish among messages
- **comm** is the communicator
- **ierr** is a return error code

Like MPI_SEND, MPI_RECV is blocking. This means the call does not return control to your program until all the received data have been stored in the variable(s) you specify in the parameter list. Because of this, you can use the data after the call and be certain it is all there. (There are non-blocking receives where this is not the case.)
C:

    int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
                  int tag, MPI_Comm comm, MPI_Status *status)

Fortran:

    MPI_RECV(buf, count, datatype, source, tag, comm, status, ierror)

The parameters:

- **buf** is the beginning of the buffer where the incoming data are to be stored. For Fortran, this is often the name of an array in your program. For C, it is an address.
- **count** is the number of elements (not bytes) in your receive buffer
- **datatype** is the type of data
- **source** is the rank of the process from which data will be accepted (This can be a wildcard, by specifying the parameter MPI_ANY_SOURCE.)
- **tag** is an arbitrary number which can be used to distinguish among messages (This can be a wildcard, by specifying the parameter MPI_ANY_TAG.)
- **comm** is the communicator
- **status** is an array or structure of information that is returned. For example, if you specify a wildcard for source or tag, status will tell you the actual rank or tag for the message received
- **ierror** is a return error code

### 2.1 Communication Modes

The communication mode is selected with the send routine. There are four blocking send routines and four non-blocking send routines, corresponding to the four communication modes. The receive routine does not specify communication mode -- it is simply blocking or non-blocking.

The table below summarizes the send and receive calls which will be described in this module.

<table>
<thead>
<tr>
<th>Communication Mode</th>
<th>Blocking Routines</th>
<th>Non-Blocking Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>MPI_SSEND</td>
<td>MPI_ISSEND</td>
</tr>
<tr>
<td>Ready</td>
<td>MPI_RSEND</td>
<td>MPI_IRSEND</td>
</tr>
<tr>
<td>Buffered</td>
<td>MPI_BSEND</td>
<td>MPI_IBSEND</td>
</tr>
</tbody>
</table>

We'll start by examining the behavior of blocking communication for the four modes, beginning with synchronous mode. For compactness, we'll delay examination of non-blocking behavior until a later section.

### 2.1.1 Blocking Synchronous Send

In the diagram below, time increases from left to right. The heavy horizontal line marked S represents execution time of the sending task (on one node), and the heavy dashed line marked R represents execution time of the receiving task (on a second node). Breaks in these lines represent interruptions due to the message-passing event.

When the blocking synchronous send MPI_Ssend (S) is executed, the sending task sends the receiving task a "ready to send" message. When the receiver executes the receive call, it sends a "ready to receive" message. The data are then transferred.

There are two sources of overhead in message-passing. **System overhead** is incurred from copying the message data from the sender's message buffer onto the network, and from copying the message data from the network into the

---

<table>
<thead>
<tr>
<th>Standard</th>
<th>MPI_SEND</th>
<th>MPI_ISEND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Try One  
Flash animation

http://www.tc.cornell.edu/Services/Edu/Topics/MPI/Pt2pt1/more.asp  
9/23/2002
receiver's message buffer.

**Synchronization overhead** is the time spent waiting for an event to occur on another task. In the figure above, the sender must wait for the receive to be executed and for the handshake to arrive before the message can be transferred. The receiver also incurs some synchronization overhead in waiting for the handshake to complete. Synchronization overhead can be significant, not surprisingly, in synchronous mode. As we shall see, the other modes try different strategies for reducing this overhead.

Only one relative timing for the MPI_Ssend (S) and MPI_Recv (S) calls is shown, but they can come in either order. If the receive call precedes the send, most of the synchronization overhead will be incurred by the receiver.

One might hope that, if workload is properly **load balanced**, synchronization overhead would be minimal on both the sending and receiving task. This is not always realistic. If nothing else causes lack of synchronization, system services which run at unpredictable times on the various nodes will cause unsynchronized delays. One might respond to this by saying that it would be simpler to just call MPI_Barrier frequently to keep the tasks in sync, but that call itself incurs synchronization overhead and doesn't assure that the tasks will be in sync a few seconds later. Thus, barrier calls are almost always a waste of time. (MPI_Barrier blocks the caller until all group members have called it.)

### 2.1.2 Blocking Ready Send

![Diagram of MPI_RSEND](http://www.tc.cornell.edu/Services/Edu/Topics/MPI/Pt2pt1/more.asp)

**MPI_RSEND** (blocking ready send)

- **data transfer from source complete**
- **wait**
- **MPI_RECV** receiving task waits until buffer is filled

Flash animation

http://www.tc.cornell.edu/Services/Edu/Topics/MPI/Pt2pt1/more.asp

9/23/2002
The ready mode send MPI_Rsend (S) simply sends the message out over the network. It requires that the "ready to receive" notification has arrived, indicating that the receiving task has posted the receive. If the "ready to receive" message hasn't arrived, the ready mode send will incur an error. By default, the code will exit. The programmer can associate a different error handler with a communicator to override this default behavior. The diagram shows the latest posting of the MPI_Recv (S) that would not cause an error.

Ready mode aims to minimize system overhead and synchronization overhead incurred by the sending task. In the blocking case, the only wait on the sending node is until all data have been transferred out of the sending task's message buffer. The receive can still incur substantial synchronization overhead, depending on how much earlier it is executed than the corresponding send.

This mode should not be used unless the user is certain that the corresponding receive has been posted.

### 2.1.3 Blocking Buffered Send

The blocking buffered send MPI_Bsend (S) copies the data from the message buffer to a user-supplied buffer, and then returns. The sending task can then proceed with calculations that modify the original message buffer, knowing that these modifications will not be reflected in the data actually sent. The data will be copied from the user-supplied buffer over the network once the "ready to receive" notification has arrived.

Buffered mode incurs extra system overhead, because of the additional copy
from the message buffer to the user-supplied buffer. Synchronization overhead is eliminated on the sending task -- the timing of the receive is now irrelevant to the sender. Synchronization overhead can still be incurred by the receiving task. Whenever the receive is executed before the send, it must wait for the message to arrive before it can return.

Another benefit for the user is the opportunity to provide the amount of buffer space for outgoing messages that the program needs. On the downside, the user is responsible for managing and attaching this buffer space. A buffered mode send that requires more buffer space than is available will generate an error, and (by default) the program will exit.

Buffer Management

For a buffered mode send, the user must provide the buffer: it can be a statically allocated array, or memory for the buffer can be dynamically allocated with malloc. The amount of memory allocated for the user-supplied buffer should exceed the sum of the message data, as message headers must also be stored.

This space must be identified as the user-supplied buffer by a call to MPI_Buffer_attach (S). When it is no longer needed, it should be detached with MPI_Buffer_detach (S). There can only be one user-supplied message buffer active at a time. It will store multiple messages. The system keeps track of when messages ultimately leave the buffer, and will reuse buffer space. For a program to be safe, it should not depend on this happening.

2.1.4 Blocking Standard Send

For standard mode, the library implementor specifies the system behavior that will work best for most users on the target system. For MPI/Pro, there are two scenarios, depending on whether the message size is greater or smaller than a threshold value, called the cutoff limit.

Cutoff limit for VIA: 8192 bytes

Cutoff limit for TCP: 32768 bytes

Cutoff limit for SMP: 8248 bytes

This value can be tuned at runtime with the following flags. However, manipulating these values can lead to undefined behavior.
   -tcp_long xxx(in bytes) Set the TCP protocol cutoff for long messages
-via_long xxx(in bytes) Set the VIA protocol cutoff for long messages
-smp_long xxx(in bytes) Set the SMP protocol cutoff for long messages

**Message size less than threshold**

The behavior when the message size is less than or equal to the threshold is shown below:

![Diagram showing MPI_Send and MPI_RECV](http://www.tc.cornell.edu/Services/Edu/Topics/MPI/Pt2pt1/more.asp)

In this case, the blocking standard send MPI_Send (S) copies the message over the network into a system buffer on the receiving node. The standard send then returns, and the sending task can continue computation. The system buffer is attached when the program is started -- the user does not need to manage it in any way. There is one system buffer per task that will hold multiple messages. The message will be copied from the system buffer to the receiving task when the receive call is executed.

As with buffered mode, use of a buffer decreases the likelihood of synchronization overhead on the sending task at the price of increased system overhead resulting from the extra copy to the buffer. As always, synchronization overhead can be incurred by the receiving task if a receive is posted first.

Unlike buffered mode, the sending task will not incur an error if the buffer space is exceeded. Instead the sending task will block until the receiving task calls a receive that pulls data out of the system buffer. Thus, synchronization overhead can still be incurred by the sending task.

**Message size greater than threshold**

When the message size is greater than the threshold, the behavior of the
blocking standard send MPI_Send (S) is essentially the same as for synchronous mode.

Why does standard mode behavior differ with message size? Small messages benefit from the decreased chance of synchronization overhead resulting from use of the system buffer. However, as message size increases, the cost of copying to the buffer increases, and it ultimately becomes impossible to provide enough system buffer space. Thus, standard mode tries to provide the best compromise.

You have now seen the system behavior for all four communication modes for blocking sends.

2.1.5 Blocking Send and Receive

The send and receive operations can be combined into one call. MPI_SENDRECV does a blocking send and receive, where the buffers for send and receive must be disjoint. MPI_SENDRECV_REPLACE also does a blocking send and receive, but note that there is only one buffer instead of two, because the received message overwrites the sent one.

- send and receive process can be the same
- SENDRECV can send to a regular RECV
- SENDRECV can receive from a regular SEND
- SENDRECV can be probed by a probe operation
2.2 Conclusions: Modes

Synchronous mode is the "safest", and therefore also the most portable. "Safe" means that if a code runs under one set of conditions (i.e. message sizes, or architecture) it will run under all conditions. Synchronous mode is safe because it does not depend upon the order in which the send and receive are executed (unlike ready mode) or the amount of buffer space (unlike buffered mode and standard mode). Synchronous mode can incur substantial synchronization overhead.

Ready mode has the lowest total overhead. It does not require a handshake between sender and receiver (like synchronous mode) or an extra copy to a buffer (like buffered or standard mode). However, the receive must precede the send. This mode will not be appropriate for all messages.

Buffered mode decouples the sender from the receiver. This eliminates synchronization overhead on the sending task and ensures that the order of execution of the send and receive does not matter (unlike ready mode). An additional advantage is that the programmer can control the size of messages to be buffered, and the total amount of buffer space. There is additional system overhead incurred by the copy to the buffer.

Standard mode behavior is implementation-specific. The library developer chooses a system behavior that provides good performance and reasonable safety.

3. Non-Blocking Behavior

A blocking send or receive call suspends execution of the program until the message buffer being sent/received is safe to use. In the case of a blocking send, this means that the data to be sent have been copied out of the send buffer, but they have not necessarily been received in the receiving task. The contents of the send buffer can be modified without affecting the message that was sent. Completion of a blocking receive implies that the data in the receive buffer are valid.

Non-blocking calls return immediately after initiating the communication. The programmer does not know at this point whether the data to be sent have been
copied out of the send buffer, or whether the data to be received have arrived. So, before using the message buffer, the programmer must check its status. Status will be covered in MPI Point to Point II.

The programmer can choose to block until the message buffer is safe to use, by using a call to MPI_Wait and its variants (S) or to just return the current status of the communication by using MPI_Test and variants (S).

The different variants of the Wait and Test calls allow you to check the status of a specific message, or to check all, any, or some of a list of messages.

It is fairly intuitive why you need to check the status of a non-blocking receive: you do not want to read the message buffer until you are sure that the message has arrived. It is less obvious why you would need to check the status of a non-blocking send. This is most necessary when you have a loop that repeatedly fills the message buffer and sends the message. You can't write anything new into that buffer until you know for sure that the preceding message has been successfully copied out of the buffer. Even if a send buffer is not re-used, it is advantageous to complete the communication, as this releases system resources.

### 3.1 Syntax of Non-Blocking Calls

The nonblocking calls have the same syntax (S) as the blocking ones, with two exceptions:

1. Each call has an "I" immediately following the "."
2. The last argument is a handle to an opaque request object that contains information about the message, i.e., its status.

For example, the standard non-blocking send and a corresponding Wait call look like this:

- **C:**

  ```c
  MPI_Issend (buf,count,dtype,dest,tag,comm,request)
  MPI_Wait (request,status)
  ```

- **Fortran:**

  ```fortran
  MPI_Isend (buf,count,dtype,dest,tag,comm,request,ierror)
  MPI_Wait (request,status,ierror)
  ```
Similarly, the non-blocking receive call is MPI_Irecv.

The Wait and Test calls take one or more request handles as input and return one or more statuses. In addition, Test indicates whether any of the communications to which the request applies have completed. Wait, Test, and status are discussed in detail in Point to Point II.

### 3.2 Example: Non-blocking standard send

We have seen the blocking behavior for each of the communication modes. We will now discuss the non-blocking behavior for standard mode. The behaviors of the other modes can be inferred from this.

The following figure shows use of both a non-blocking standard send MPI_Isend (S) and a non-blocking receive MPI_Irecv (S). As before, the standard mode send will proceed differently depending on the message size. The following figure demonstrates the behavior for message size less than or equal to the threshold.

The sending task posts the non-blocking standard send when the message buffer contents are ready to be transmitted. It returns immediately without waiting for the copy to the remote system buffer to complete. MPI_Wait (S) is called just before the sending task needs to overwrite the message buffer.

The receiving task calls a non-blocking receive as soon as a message buffer is available to hold the message. The non-blocking receive returns without waiting for the message to arrive. The receiving task calls MPI_Wait (S) when it needs to use the incoming message data (i.e. needs to be certain that it has arrived).
The system overhead will not differ substantially from the blocking send and receive calls unless data transfer and computation can occur simultaneously. Since the CPU may need to perform both the data transfer and the computation, computation will be interrupted on both the sending and receiving nodes to pass the message. The point in time when the interruption occurs should not be of any particular consequence to the program that is running. Even for architectures that overlap computation and communication, the fact that this case applies only to small messages means that no great difference in performance would be expected.

The advantage of using the non-blocking send occurs when the remote system buffer is full. In this case, a blocking send would have to wait until the receiving task pulled some message data out of the buffer. If a non-blocking call is used, computation can be done during this interval.

The advantage of a non-blocking receive over a blocking one can be considerable if the receive is posted before the send. The task can continue computing until the Wait is posted, rather than sitting idle. This reduces the amount of synchronization overhead.

Non-blocking calls can ensure that both processors waiting for each other will not result. The Wait must be posted after the calls needed to complete the communication.

### 3.3 Example: Non-blocking standard send, large message

The case of a non-blocking standard send MPI_Isend(S) for a message larger than the threshold is more interesting:
For a blocking send, the synchronization overhead would be the period between the blocking call and the copy over the network. For a non-blocking call, the synchronization overhead is reduced by the amount of time between the non-blocking call and the MPI_Wait (S), in which useful computation is proceeding.

Again, the non-blocking receive MPI_Irecv (S) will reduce synchronization overhead on the receiving task for the case in which the receive is posted first. There is also a benefit to using a non-blocking receive when the send is posted first. Consider how the figure would change if a blocking receive were posted. Typically, blocking receives are posted immediately before the message data must be used (to allow the maximum amount of time for the communication to complete). So, the blocking receive would be posted in place of the MPI_Wait. This would delay the synchronization with the send call until this later point in the program, and thus increase synchronization overhead on the sending task.

### 3.4 Conclusions: Non-blocking Calls

- **Avoid Deadlock**
  
  Avoid having processes waiting for the same resource or for each other. Deadlock is discussed in detail in MPI Point to Point II.

- **Decrease Synchronization Overhead**
  
  Non-blocking calls have the advantage that computation can continue almost immediately, even if the message can’t be sent. This can eliminate deadlock and reduce synchronization overhead.
Some Systems: Reduce Systems Overhead

On some machines, the system overhead can be reduced if the message transport can be handled in the background without having any impact on the computations in progress on both the sender and receiver.

A knowledgeable source has the following comments regarding whether or not non-blocking calls do in fact result in a reduction of system overhead:

...I have the strong suspicion that non-blocking calls can actually incur more overhead than blocking ones that result in immediate message transfer. This extra overhead comes from several sources:
1. the cost of allocating a request object
2. the overhead of doing the interrupt when the data are transferred later
3. the cost of querying to determine whether the transfer has completed.

The first cost can be eliminated by using persistent requests (definitely an advanced topic). All these can be small compared to the synchronization overhead that is avoided if useful computations are available to be done.

Once you've gotten some experience with the basics of MPI, consult the standard or available texts for information on persistent requests; the rest of the comment simply points out that a blocking call carries with it much less systems-baggage than does a non-blocking call, if you can assume that both are going to be satisfied immediately, but if the transaction is not immediately satisfied, then the non-blocking call wins to the extent that "useful computation" can be accomplished prior to its conclusion.

Best to post non-blocking sends and receives as early as possible, and to do waits as late as possible

Some additional programming is required with non-blocking calls, to test for completion of the communication. It is best to post sends and receives as early as possible, and to wait for completion as late as possible. "Early as possible" above means that the data in the buffer to be sent must be valid, and likewise the buffer to be received into must be available.

Must avoid writing to send buffer between MPI_Isend and MPI_Wait and must avoid reading and writing in receive buffer between MPI_Irecv and MPI_Wait
It should be possible to safely read the send buffer after the send is posted, but nothing should be written to that buffer until status has been checked to give assurance that the original message has been sent. Otherwise, the original message contents could be overwritten. NO user reading or writing of the receive buffer should take place between posting a non-blocking receive and determining that the message has been received. The read might give either old data or new (incoming) message data. A write could overwrite the recently arrived message.

4. Programming Recommendations

In general, it is reasonable to start programming with non-blocking calls and standard mode. Non-blocking calls can eliminate the possibility of deadlock and reduce synchronization overhead. Standard mode gives generally good performance.

Blocking calls may be required if the programmer wishes the tasks to synchronize. Also, if the program requires a non-blocking call to be immediately followed by a Wait, it is more efficient to use a blocking call. If using blocking calls, it may be advantageous to start in synchronous mode, and then switch to standard mode. Testing in synchronous mode will ensure that the program does not depend on the presence of sufficient system buffering space.

The next step is to analyze the code and evaluate its performance. If non-blocking receives are posted early, well in advance of the corresponding sends, it might be advantageous to use ready mode. In this case, the receiving task should probably notify the sender after the receives have been posted. After receiving the notification, the sender can proceed with the sends.

If there is too much synchronization overhead on the sending task, especially for large messages, buffered mode may be more efficient.

References


http://www.tc.cornell.edu/Services/Edu/Topics/MPI/Pt2pt1/more.asp
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


Exercise  Tutorial in C
Exercise  Tutorial in Fortran
Exercise  Lab exercise
Quiz  Take a multiple-choice quiz on this material, and submit it for grading.
Evaluation  Please complete this short evaluation form. Thank you!

© 2002 Cornell University. All Rights Reserved.
Read our Copyright guidelines.
Last modified on 09/23/2002 12:54:46