Point-to-Point Communication

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Introduction

- Point-to-point communication is the fundamental communication facility provided by the MPI library.
- Point-to-point communication is conceptually simple: one process sends a message and another process receives it.
- ***MPI_Send*** and ***MPI_Recv*** work together to complete a transfer of data from one process to another.

Point-to-point communication

- However, it is less simple in practice. For example, a process may have many messages waiting to be received. In that case, a crucial issue is how MPI and the receiving process determine what message to receive.
- Another issue is whether send and receive routines initiate communication operations and return immediately, or wait for the initiated communication operation to complete before returning. The underlying communication operations are the same in both cases, but the programming interface is very different.

Topics

- Fundamentals of point-to-point communication
- Blocking send and receive
- Nonblocking send and receive
- Send modes

Fundamentals

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Fundamentals

- The following issues are fundamental to point-to-point communication in MPI. These apply to all versions of send and receive, both blocking and nonblocking, and to all send modes.
  - Source and Destination
  - Messages
  - Sending and Receiving Messages

Source and Destination

- The point-to-point communication discussed here are two-sided and require active participation from the processes on both sides. One process (the source) sends, and another process (the destination) receives.
- In general, the source and destination processes operate asynchronously.
  - Even the sending and receiving of a single message is typically not synchronized. The source process may complete sending a message long before the destination process gets around to receiving it, and the destination process may initiate receiving a message that has not yet been sent.

Source and Destination

- Because sending and receiving are typically not synchronized, processes often have one or more messages that have been sent but not yet received.
- These sent, but not yet received messages are called **pending** messages.
- It is an important feature of MPI that pending messages are *not* maintained in a simple FIFO queue. Instead, each pending message has several attributes and the destination process (the receiving process) can use the attributes to determine which message to receive.

Messages

- Messages consist of 2 parts: the **envelope** and the **message body**.

$$\text{MPI\_Send(}\underbrace{\text{sendbuf,cnt,datatype}}_{\textbf{message body}}\text{,}\underbrace{\text{dest,tag,comm}}_{\textbf{envelope}}\text{);}$$

$$\text{MPI\_Recv(}\overbrace{\text{recvbuf,cnt,datatype}}\text{,}\overbrace{\text{source,tag,comm}}\text{,status);}$$

MPI Message	**Letter**
Send/receive buffer	letter content
Count/size	Letter weight
Source (receive)	Return address
Destination (send)	Destination address
Communicator	Country

Message body

- The **message body** has 3 parts:
  - **Buffer**

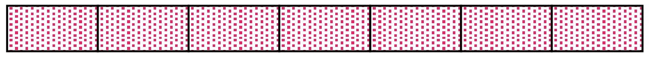
    It is the space in the computer's memory where the MPI message data are to be sent from or stored to
  - **Datatype**

    The type of the message data to be transmitted(e.g.floating point). The datatype should be the same for the send and receive call.
  - **count**

    The number of items of data in the message.

Message body

- Think of the buffer as an array; the dimension is given by count, and the type of the array elements is given by datatype.



(count=7)

An MPI message is an array of elements of a particular MPI *datatype*.

- Using datatypes and counts, rather than bytes and bytecounts, allows structured data and noncontiguous data to be handled smoothly.

Message Envelope

- Message envelope of an MPI message provides information on how to match sends to receives. it consists of 3 parts:
  - **Source or destination**

    This argument is set to a rank in a communicator. Destination is specified by the sending process and source is specified by the receiving process. Only messages coming from that source can be accepted by the receive call, but the receive can set source to ***MPI_ANY_SOURCE*** to indicate that any source is acceptable.
  - **Communicator**

    The communicator specifies a group of processes to which both source and destination belong.
  - **Tag**

    The tag is an arbitrary number to help distinguish among messages. The tags specified by the sender and receiver must match, but the receiver can specify ***MPI_ANY_TAG*** to indicate that any tag is acceptable. For example, one tag value can be used for messages containing data and another tag value for messages containing status information.

Sending and Receiving Messages

- Sending messages is straightforward. The source (the identity of the sender) is determined implicitly, but the rest of the message (envelope and body) is given explicitly by the sending process.
- Receiving messages is not quite so simple. As a process may have several pending messages.
- To receive a message, a process specifies a message envelope that MPI compares to the envelopes of pending messages. If there is a match, a message is received. Otherwise, the receive operation cannot be completed until a matching message is sent.
- In addition, the process receiving a message must provide storage into which the body of the message can be copied. The receiving process must be careful to provide enough storage for the entire message.

Blocking Send and Receive

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Blocking Send and Receive

- The two functions, *MPI_Send and MPI_Recv*, are the basic point-to-point communication routines in MPI. Their calling sequences are presented and discussed in the following sections. Both functions block the calling process until the communication operation is completed.
- Blocking creates the possibility of deadlock, a key issue that is explored by way of simple examples. In addition, the meaning of completion is discussed.
- The nonblocking analogues of MPI_Send and MPI_Recv are presented in next section.

Sending a Message: MPI_SEND

- MPI_Send takes the following arguments:
  - Message Body
    - Buffer
    - Count
    - Datatype
  - Message Envelope (source – the sending process – is defined implicitly)
    - Destination
    - Tag
    - Communicator
- The message body contains the data to be sent: *count* items of type *datatype*. The message envelope tells where to send it. In addition, an error code is returned.

Sending a Message: MPI_SEND

- C binding is shown below.

  int MPI_Send(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);

  - All arguments are input arguments.
  - An error code is returned by the function.

Example:

```
MPI_Send(a,10,MPI_INT,0,10,MPI_COMM_WORLD);
MPI_Send(&b,1,MPI_DOUBLE,2,19,Comm1);
```

Receiving a Message: MPI_RECV

- MPI_Recv takes a set of arguments similar to MPI_Send, but several of the arguments are used in a different way.
  - Message Body
    - Buffer
    - Count
    - Datatype
  - Message Envelope (receiving process – is defined implicitly)
    - Source
    - Tag
    - Communicator
  - Status – information on the message that was received

Receiving a Message: MPI_RECV

- The message envelope arguments determine what messages can be received by the call. The source, tag, and communicator arguments must match those of a pending message in order for the message to be received.
- Wildcard values may be used for the source (*MPI_ANY_SOURCE*, accept a message from any process) and the tag (*MPI_ANY_TAG,* accept a message with any tag value). If wildcards are not used, the call can accept messages from only the specified sending process, and with only the specified tag value. Communicator wildcards are not available.
- The message body arguments specify where the arriving data are to be stored, what type it is assumed to be, and how much of it the receiving process is prepared to accept. If the received message has more data than the receiving process is prepared to accept, it is an error.

Receiving a Message: MPI_RECV

- In general, the sender and receiver must agree about the message datatype, and it is the programmer's responsibility to guarantee that agreement. If the sender and receiver use incompatible message datatypes, the results are undefined.
- The status argument returns information about the message that was received. The source and tag of the received message are available this way (needed if wildcards were used); also available is the actual count of data received.
- In addition, an error code is returned.

Receiving a Message: MPI_RECV

- C binding is shown below.

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

  – buf and status are output arguments; the rest are inputs.

  – An error code is returned by the function.

Example:

```
MPI_Recv(c,10,MPI_INT,1,10,MPI_COMM_WORLD,&stat);
MPI_Recv(&d,1,MPI_DOUBLE,0,19,Comm1,&stat);
```

Notes:

- A maximum of COUNT items of type DTYPE are accepted; if the message contains more, it is an error.
- The sending and receiving processes must agree on the datatype; if they disagree, results are undefined (MPI does not check).
- When this routine returns, the received message data have been copied into the buffer; and the tag, source, and actual count of data received are available via the status argument.

Example: Send and Receive

- In this program, process 0 sends a message to process 1, and process 1 receives it. Note the use of myrank in a conditional to limit execution of code to a particular process.

```
/* simple send and receive */
#include <stdio.h>
#include <mpi.h>

void main (int argc, char **argv) {
   int myrank;
   MPI_Status status;
   double a[100];

   MPI_Init(&argc, &argv);  /* Initialize MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
   if( myrank == 0 )        /* Send a message */
       MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
   else if( myrank == 1 )   /* Receive a message */
       MPI_Recv( a, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );


    MPI_Finalize();          /* Terminate MPI */
}
```

What Happens at Runtime

- It is useful to keep in mind the following model for the runtime behavior of MPI_Send. According to the model, when a message is sent using MPI_Send one of two things may happen:

  Option 1:

  The message may be copied into an MPI internal buffer and transferred to its destination later, in the background, or

  Option 2:

  The message may be left where it is, in the program's variables, until the destination process is ready to receive it. At that time, the message is transferred to its destination.

What Happens at Runtime

- The first option allows the sending process to move on to other things after the copy is completed.
- The second option minimizes copying and memory use, but may result in extra delay to the sending process. The delay can be significant.

- Surprisingly, in 1., a call to MPI_Send may return before any non-local action has been taken or even begun, i.e., before anything has happened that might naively be associated with sending a message. In 2., a synchronization between sender and receiver is implied.
- To summarize, according to the model sketched above, when a message is sent using MPI_Send, the message is either
  - buffered immediately and delivered later asynchronously, or
  - the sending and receiving processes synchronize.

Blocking and Completion

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Blocking and Completion

- Both MPI_Send and MPI_Recv block the calling process. Neither returns until the communication operation it invoked is completed.
- The meaning of completion for a call to MPI_Recv is simple and intuitive
  - A matching message has arrived, and the message’s data ,which have been copied into the output arguments of the call, are ready to be used.
- For MPI_Send, the meaning of completion is simple but not as intuitive.
  - A call to MPI_Send is completed when the variables passed to MPI_Send can now be overwritten and reused.
  - Recall from the previous section that one of two things may have happened: either MPI copied the message into an internal buffer for later, asynchronous delivery; or else MPI waited for the destination process to receive the message. Note that if MPI copied the message into an internal buffer, then the call to MPI_Send may be officially completed, even though the message has not yet left the sending process.

Blocking and Completion

- If a message passed to MPI_Send is larger than MPI's available internal buffer, then buffering cannot be used. In this case, the sending process must block until the destination process begins to receive the message, or until more buffer is available. In general, messages that are copied into MPI internal buffer will occupy buffer space until the destination process begins to receive the message.
- Note that a call to MPI_Recv matches a pending message if it matches the pending message's envelope (source, tag, communicator). Datatype matching is also required for correct execution but MPI does not check for it. Instead, it is the obligation of the programmer to guarantee datatype matching.

Deadlock

- **Deadlock** occurs when 2 (or more) processes are blocked and each is waiting for the other to make progress. Neither process makes progress because each depends on the other to make progress first.

- The program shown below is an example
  - it fails to run to completion because processes 0 and 1 deadlock.

Deadlock

```
/* simple deadlock */
#include <stdio.h>
#include <mpi.h>

void main (int argc, char **argv) {
   int myrank;
   MPI_Status status;
   double a[100], b[100];

   MPI_Init(&argc, &argv);  /* Initialize MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */

   if( myrank == 0 ) {
        /* Receive, then send a message */
        MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD,
   &status );
        MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
   } else if( myrank == 1 ) {
        /* Receive, then send a message */
        MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
   &status );
        MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
   }
   MPI_Finalize();            /* Terminate MPI */

}
```

Deadlock

- In the program, process 0 attempts to exchange messages with process 1.

Process 0		Process 1
MPI_Recv (from P1)	**deadlock**	MPI_Recv (from P0)
MPI_Send (to P1)		MPI_Send (to P0)

- Process 0 cannot proceed until process 1 sends a message; process 1 cannot proceed until process 0 sends a message.
- The program is erroneous and deadlocks. No messages are ever sent, and no messages are ever received.

Avoiding Deadlock

- In general, avoiding deadlock requires careful organization of the communication in a program. The programmer should be able to explain why the program does not (or does) deadlock.
- The program shown below is similar to the program in the preceding section, but its communication is better organized and the program does not deadlock.

Process 0		Process 1
MPI_Recv (from P1)	←	MPI_Send (to P0)
MPI_Send (to P1)	→	MPI_Recv (from P0)

- Once again, process 0 attempts to exchange messages with process 1. The protocol is safe. Except system failures, this program always runs to completion.

Avoiding Deadlock

- Note that increasing array dimensions and message sizes have no effect on the safety of the protocol. The program still runs to completion.
- This is a useful property for application programs
  - when the problem size is increased, the program still runs to completion.

Avoiding Deadlock

```
/* safe exchange */
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {

   int myrank;
   MPI_Status status;
   double a[100], b[100];

   MPI_Init(&argc, &argv);  /* Initialize MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */

   if( myrank == 0 ) {
        /* Receive a message, then send one */
        MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
        MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
   }
    else if( myrank == 1 ) {
        /* Send a message, then receive one */
        MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
        MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );

   }
   MPI_Finalize();           /* Terminate MPI */
}
```

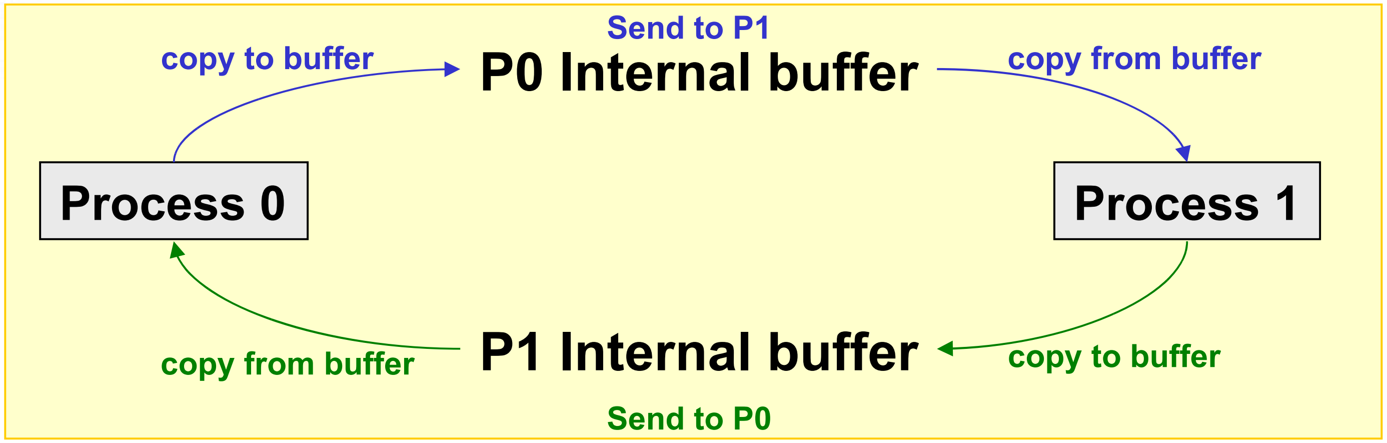
Avoiding Deadlock (Sometimes but Not Always)

- The program shown below is similar to preceding examples. This time, both processes send first, then receive.

Process 0	Process 1
MPI_Send (to P1)	MPI_Send (to P0)
MPI_Recv (from P1)	MPI_Recv (from P0)

buffer size		status
Large enough	→	Completion
Not enough	→	Deadlock

- Success depends on the availability of buffering in MPI. There must be enough MPI internal buffer available to hold at least one of the messages in its entirety.



Avoiding Deadlock (Sometimes but Not Always)

```
/* depends on buffering */
#include <stdio.h>
#include <mpi.h>

void main (int argc, char **argv) {

   int myrank;
   MPI_Status status;
   double a[100], b[100];

   MPI_Init(&argc, &argv);  /* Initialize MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
   if( myrank == 0 ) {
        /* Send a message, then receive one */
        MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
        MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
   }
   else if( myrank == 1 ) {
        /* Send a message, then receive one */
        MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
        MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
   }
   MPI_Finalize();             /* Terminate MPI */
}
```

Avoiding Deadlock (Sometimes but Not Always)

- Under most MPI implementations, the program shown will run to completion. However, if the message sizes are increased, sooner or later the program will deadlock.
- This behavior is sometimes seen in computational codes
  - a code will run to completion when given a small problem, but deadlock when given a large problem.
- This is inconvenient and undesirable. The inconvenience is increased when the original authors of the code are no longer available to maintain it.
- In general, depending on MPI internal buffer to avoid deadlock makes a program less portable and less scalable. The best practice is to write programs that run to completion regardless of the availability of MPI internal buffer.

Probable Deadlock

- The only significant difference between the program shown below and the preceding one is the size of the messages. This program will deadlock under the default configuration of nearly all available MPI implementations.

Probable Deadlock

```
/* probable deadlock */
#include <stdio.h>
#include <mpi.h>

void main (int argc, char **argv) {
   int myrank;
   MPI_Status status;
   #define N 100000000
   double a[N], b[N];

   MPI_Init(&argc, &argv);  /* Initialize MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
   if( myrank == 0 ) {
        /* Send a message, then receive one */
        MPI_Send( a, N, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
        MPI_Recv( b, N, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
   }
   else if( myrank == 1 ) {
        /* Send a message, then receive one */
        MPI_Send( a, N, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
        MPI_Recv( b, N, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
   }
   MPI_Finalize();          /* Terminate MPI */
}
```

Nonblocking Sends and Receives

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Nonblocking Sends and Receives

- MPI provides another way to invoke send and receive operations. It is possible to separate the initiation of a send or receive operation from its completion by making two separate calls to MPI. The first call initiates the operation, and the second call completes it. Between the two calls, the program is free to do other things.
- The nonblocking interface to send and receive requires two calls per communication operation: one call to initiate the operation, and a second call to complete it. Initiating a send operation is called *posting a send*. Initiating a receive operation is called *posting a receive*.
- The communication operations are the same, but the interface to the library is different.

Posting, Completion, and Request Handles

- Once a send or receive operation has been posted, MPI provides two distinct ways of completing it.
  - A process can test to see if the operation has completed, without blocking on the completion.
  - Alternately, a process can wait for the operation to complete.

Posting, Completion, and Request Handles

- After posting a send or receive with a call to a nonblocking routine, the posting process needs some way to refer to the posted operation. MPI uses request handles for this purpose (See Chapter 3 - MPI Handles). Nonblocking send and receive routines all return request handles, which are used to identify the operation posted by the call.
- In summary, sends and receives may be posted (initiated) by calling nonblocking routines. Posted operations are identified by request handles. Using request handles, processes can check the status of posted operations or wait for their completion.

Posting Sends without Blocking

- A process calls the routine MPI_Isend to post (initiate) a send without blocking on completion of the send operation. The calling sequence is similar to the calling sequence for the blocking routine MPI_Send but includes an additional output argument, a request handle.

```
MPI_Send(buf,cnt,datatype,dest,tag,comm);
MPI_Isend(buf,cnt,datatype,dest,tag,comm,request);
```

- The request handle identifies the send operation that was posted. The request handle can be used to check the status of the posted send or to wait for its completion.

Posting Sends without Blocking

- Nonblocking C version of the standard mode send is given below.

  int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request);

  – An error code is returned by the function.

Example:

```
MPI_Isend(a,10,MPI_INT,1,10,MPI_COMM_WORLD,&req);
MPI_Isend(&b,1,MPI_DOUBLE,2,19,Comm1,&req);
```

Notes:

- The source of the message, the sending process, is determined implicitly.
- When this routine returns, a send has been posted (but not yet completed).
- Another call to MPI is required to complete the send operation posted by this routine.
- None of the arguments passed to MPI_Isend should be read or written until the send operation is completed.

Posting Receives without Blocking

- A process calls the routine MPI_Irecv to post (initiate) a receive without blocking on its completion. The calling sequence is similar to the calling sequence for the blocking routine MPI_Recv, but the status argument is replaced by a request handle; both are output arguments.

  ```
  MPI_Recv(buf,cnt,datatype,source,tag,comm,status);
  MPI_Irecv(buf,cnt,datatype,source,tag,comm,request);
  ```

- The request handle identifies the receive operation that was posted and can be used to check the status of the posted receive or to wait for its completion.

Posting Receives without Blocking

- Nonblocking C version of the standard mode send is given below.

  int MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Request *request);

  – An error code is returned by the function.

Notes:

- A maximum of count items of type DTYPE is accepted; if the message contains more, it is an error.
- The sending and receiving processes must agree on the datatype; if they disagree, it is an error.
- When this routine returns, the receive has been posted (initiated) but not yet completed.
- Another call to MPI is required to complete the receive operation posted by this routine.
- None of the arguments passed to MPI_Irecv should be read or written until the receive operation is completed.

Completion: Waiting and Testing

- Posted sends and receives must be completed.
- If a send or receive is posted by a nonblocking routine, then its completion status can be checked by calling one of a family of completion routines.
- MPI provides both blocking and nonblocking completion routines.
  - The blocking routines are *MPI_Wait* and its variants.
  - The nonblocking routines are *MPI_Test* and its variants. These routines are discussed in the following two sections.

Completion: Waiting

- A process that has posted a send or receive by calling a nonblocking routine (for instance, MPI_Isend or MPI_Irecv) can subsequently wait for the posted operation to complete by calling MPI_Wait. The posted send or receive is identified by passing a request handle.
- The arguments for the MPI_Wait routine are:
  - Request
    - a request handle (returned when the send or receive was posted)
  - Status
    - for receive, information on the message received; for send, may contain an error code
- In addition, an error code is returned.

Completion: Waiting

- C version of MPI_Wait is given below.

```
int MPI_Wait( MPI_Request *request, MPI_Status *status );
```

  - An error code is returned.

Notes:

- The request argument is expected to identify a previously posted send or receive.
- MPI_Wait returns when the send or receive identified by the request argument is complete.
- If the posted operation was a receive, then the source, tag, and actual count of data received are available via the status argument.
- If the posted operation was a send, the status argument may contain an error code for the send operation (different from the error code for the call to MPI_Wait).

Completion: Testing

- A process that has posted a send or receive by calling a nonblocking routine can subsequently test for the posted operation's completion by calling MPI_Test. The posted send or receive is identified by passing a request handle.
- The arguments for the MPI_Test routine are:
  - Request
    - a request handle (returned when the send or receive was posted).
  - Flag
    - **true** if the send or receive has completed.
  - Status
    - undefined if flag equals **false**. Otherwise, like MPI_Wait.
- In addition, an error code is returned.

Completion: Testing

- C versions of MPI_Test is given below.

  int MPI_Test(MPI_Request *request, int *flag,
      MPI_Status *status);

  – An error code is returned.

Notes:

- The request argument is expected to identify a previously posted send or receive.
- MPI_Test returns immediately.
- If the flag argument is **true**, then the posted operation is complete.
- If the flag argument is **true** and the posted operation was a receive,
then the source, tag, and actual count of data received are available via the status argument.
- If the flag argument is **true** and the posted operation was a send, then the status argument may contain an error code for the send operation (not for MPI_Test).

Nonblocking Sends and Receives: Advantages and Disadvantages

- Selective use of nonblocking routines makes it much easier to write deadlock-free code. This is a big advantage because it is easy to unintentionally write deadlock into programs.
- On systems where latencies are large, posting receives early is often an effective, simple strategy for masking communication overhead.
  - Latencies tend to be large on physically distributed collections of hosts (for example, clusters of workstations) and relatively small on shared memory multiprocessors. In general, masking communication overhead requires careful attention to algorithms and code structure.
- On the downside, using nonblocking send and receive routines may increase code complexity, which can make code harder to debug and harder to maintain.

Send/Receive Example

- This program is a revision of the earlier example given in previous section. This version runs to completion.
- Process 0 attempts to exchange messages with process 1. Each process begins by posting a receive for a message from the other. Then, each process blocks on a send. Finally, each process waits for its previously posted receive to complete.
- Each process completes its send because the other process has posted a matching receive. Each process completes its receive because the other process sends a message that matches. Except system failure, the program runs to completion.

Send/Receive Example

```
/* deadlock avoided */
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
   int myrank;
   MPI_Request request;
   MPI_Status status;
   double a[100], b[100];

   MPI_Init(&argc, &argv);  /* Initialize MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
   if( myrank == 0 ) {
        /* Post a receive, send a message, then wait */
        MPI_Irecv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &request );
        MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
        MPI_Wait( &request, &status );
   } else if( myrank == 1 ) {
        /* Post a receive, send a message, then wait */
        MPI_Irecv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &request );

        MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
        MPI_Wait( &request, &status );
   }
   MPI_Finalize();             /* Terminate MPI */
}
```

Send Modes

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Send Modes

- MPI provides the following four send modes:
  - Standard Mode Send
  - Synchronous Mode Send
  - Ready Mode Send
  - Buffered Mode Send
- This section describes these send modes and briefly indicates when they are useful. Standard mode, used in all example code so far in this chapter, is the most widely used.
- Although there are four send modes, there is only one receive mode. A receiving process can use the same call to MPI_Recv or MPI_Irecv, regardless of the send mode used to send the message.
- Both blocking and nonblocking calls are available for each of the four send modes.

Standard Mode Send

- **Standard mode send** is MPI’s general-purpose send mode. The other three send modes are useful in special circumstances, but none have the general utility of standard mode.
- Recall the discussion of Sections *What Happens at Runtime* and *Blocking and Completion*.
  - When MPI executes a standard mode send, one of two things happens.
  - Either the message is copied into an MPI internal buffer and transferred asynchronously to the destination process, or the source and destination processes synchronize on the message.
  - The MPI implementation is free to choose (on a case-by-case basis) between buffering and synchronizing, depending on message size, resource availability, etc.

Standard Mode Send

- If the message is copied into an MPI internal buffer, then the send operation is formally completed as soon as the copy is done.
- If the two processes synchronize, then the send operation is formally completed only when the receiving process has posted a matching receive and actually begun to receive the message.

Standard Mode Send

- The preceding comments apply to both blocking and nonblocking calls, i.e., to both MPI_SEND and MPI_ISEND.
- MPI_SEND does not return until the send operation it invoked has completed.
  - Completion can mean the message was copied into an MPI internal buffer, or it can mean the sending and receiving processes synchronized on the message.
- In contrast, MPI_ISEND initiates a send operation and then returns immediately, without waiting for the send operation to complete.
  - Completion has the same meaning as before: either the message was copied into an MPI internal buffer or the sending and receiving processes synchronized on the message.

Standard Mode Send

- Note: the variables passed to MPI_ISEND cannot be used (should not even be read) until the send operation invoked by the call has completed. A call to MPI_TEST, MPI_WAIT or one of their variants is needed to determine completion status.
- One of the advantages of standard mode send is that the choice between buffering and synchronizing is left to MPI on a case-by-case basis. In general, MPI has a clearer view of the tradeoffs, especially since low-level resources and resources internal to MPI are involved.

Synchronous, Ready Mode, and Buffered Send

- **Synchronous mode send** requires MPI to synchronize the sending and receiving processes.
- When a synchronous mode send operation is completed, the sending process may assume the destination process has **begun receiving the message**. The destination process need not be done receiving the message, but it must have begun receiving the message.
- The nonblocking call has the same advantages the nonblocking standard mode send has: the sending process can avoid blocking on a potentially lengthy operation.

Synchronous, Ready Mode, and Buffered Send

- **Ready mode send** requires that a matching receive has already been posted at the destination process before ready mode send is called. If a matching receive has not been posted at the destination, the result is undefined. It is your responsibility to make sure the requirement is met.
- In some cases, knowledge of the state of the destination process is available without doing extra work. Communication overhead may be reduced because shorter protocols can be used internally by MPI when it is known that a receive has already been posted.
- The nonblocking call has advantages similar to the nonblocking standard mode send: the sending process can avoid blocking on a potentially lengthy operation.

Synchronous, Ready Mode, and Buffered Send

- **Buffered mode send** requires MPI to use buffering. The downside is that you must assume responsibility for managing the buffer. If at any point, insufficient buffer is available to complete a call, the results are undefined. The functions MPI_BUFFER_ATTACH and MPI_BUFFER_DETACH allow a program to make buffer available to MPI.

Naming Conventions and Calling Sequences

- There are eight send functions in MPI: four send modes, each available in both blocking and nonblocking forms.
- The blocking send functions take the same arguments (in the same order) as MPI_SEND. The nonblocking send functions take the same arguments (in the same order) as MPI_ISEND.
- Synchronous, buffered, and ready mode sends are indicated by adding the letters S, B, and R, respectively, to the function name. Nonblocking calls are indicated by adding an I to the function name. The table below shows the eight function names.

Naming Conventions and Calling Sequences

Send Mode	Blocking Function	Nonblocking Function
Standard	MPI_SEND	MPI_ISEND
Synchronous	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffered	MPI_BSEND	MPI_IBSEND

Definition Review

- MPI_SEND
  - Used to perform a standard-mode, blocking send of data referenced by message to the process with rank dest.
- MPI_ISEND
  - Used to post a nonblocking send in standard mode, allocating a request object and returning a handle to it.
- MPI_SSEND
  - Used for a synchronous mode for a blocking send. It won't return until a matching receive has been posted and data reception has begun.
- MPI_ISSEND
  - Used to post a nonblocking send in synchronous mode.

Definition Review

- MPI_RSEND
  - Used for a ready send, ready mode for blocking send. The matching receive must be posted before the call to MPI_RSEND.
- MPI_IRSEND
  - Used to post a nonblocking send in ready mode.
- MPI_BSEND
  - Used for a buffered, blocking send. The buffer is user-allocated.
- MPI_IBSEND
  - Used to start a nonblocking, buffered-mode send.

END

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/