MPI Fundamentals—a Quick Overview

Shantanu Dutt
ECE Dept., UIC
Acknowledgements

• MPI: A Message-Passing Interface Standard, Version 2.2, by The Message Passing Interface Forum

• Tutorial on MPI: The Message-Passing Interface (Slides), William D Gropp; see http://www.mcs.anl.gov/research/projects/mpi/tutorial/gropp/talk.html
#include "mpi.h"

int main( int argc, char **argv )
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0)    /* code for process zero */
    {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else if (myrank == 1)    /* code for process one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
}

In this example, process zero (myrank = 0) sends a message to process one using the send operation MPI_SEND. The operation specifies a send buffer in the sender memory from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable message in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition, the send operation associates an envelope with the message. This envelope specifies the message destination and contains distinguishing information that can be used by the receive
Intro by Example (contd.)

#include "mpi.h"

int main( int argc, char **argv )
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0) /* code for process zero */
    {
        strcpy(message,"Hello, there!");
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else if (myrank == 1) /* code for process one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();

operation to select a particular message. The last three parameters of the send operation, along with the rank of the sender, specify the envelope for the message sent. Process one (myrank = 1) receives this message with the receive operation MPI_RECV. The message to be received is selected according to the value of its envelope, and the message data is stored into the receive buffer. In the example above, the receive buffer consists of the storage containing the string message in the memory of process one. The first three parameters of the receive operation specify the location, size and type of the receive buffer. The next three parameters are used for selecting the incoming message. The last parameter is used to return information on the message just received.
Blocking Send’s and Receive’s

The syntax of the blocking send operation is given below.

\[
\text{MPI}_{-}\text{SEND}(\text{buf, count, datatype, dest, tag, comm})
\]

- **IN buf** initial address of send buffer (choice)
- **IN count** number of elements in send buffer (non-negative integer)
- **IN datatype** datatype of each send buffer element (handle)
- **IN dest** rank of destination (integer)
- **IN tag** message tag (integer)
- **IN comm** communicator (handle)

\[
\text{int MPI}_{-}\text{Send( void* buf, int count, MPI\_Datatype datatype, int dest,}
\]

\[
\text{int tag, MPI\_Comm comm})}
\]

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>char</td>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>(treated as printable character)</td>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG_INT</td>
<td>signed long long int</td>
<td>MPI_UNSIGNED_LONG_LONG</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG (as a synonym)</td>
<td>signed long long int</td>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_SIGNED_CHAR</td>
<td>signed char</td>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td></td>
<td>(treated as integral value)</td>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
</tbody>
</table>

MPI_BYTE

MPI_PACKED
The integer-valued message tag is specified by the tag argument. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is 0,...,UB, where the value of UB is implementation dependent. It can be found by querying the value of the attribute MPI_TAG_UB, as described in Chapter 8. MPI requires that UB be no less than 32767.

The comm argument specifies the communicator that is used for the send operation. Communicators are explained in Chapter 6; below is a brief summary of their usage.

A communicator specifies the communication context for a communication operation. Each communication context provides a separate “communication universe:” messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

The communicator also specifies the set of processes that share this communication context. This process group is ordered and processes are identified by their rank within this group. Thus, the range of valid values for dest is 0, ... , n-1, where n is the number of processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 6.)

A predefined communicator MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in the group of MPI_COMM_WORLD.
The syntax of the blocking receive operation is given below.

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>buf</td>
<td>initial address of receive buffer (choice)</td>
</tr>
<tr>
<td>IN</td>
<td>count</td>
<td>number of elements in receive buffer (non-negative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>datatype of each receive buffer element (handle)</td>
</tr>
<tr>
<td>IN</td>
<td>source</td>
<td>rank of source or MPI_ANY_SOURCE (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>tag</td>
<td>message tag or MPI_ANY_TAG (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>status</td>
<td>status object (Status)</td>
</tr>
</tbody>
</table>

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

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.

*Advice to users.* The MPI_PROBE function described in Section 3.8 can be used to receive messages of unknown length. *(End of advice to users.)*
The selection of a message by a receive operation is governed by the value of the message envelope. A message can be received by a receive operation if its envelope matches the source, tag and comm values specified by the receive operation. The receiver may specify a wildcard MPI_ANY_SOURCE value for source, and/or a wildcard MPI_ANY_TAG value for tag, indicating that any source and/or tag are acceptable. It cannot specify a wildcard value for comm. Thus, a message can be received by a receive operation only if it is addressed to the receiving process, has a matching communicator, has matching source unless source=MPI_ANY_SOURCE in the pattern, and has a matching tag unless tag=MPI_ANY_TAG in the pattern.

The message tag is specified by the tag argument of the receive operation. The argument source, if different from MPI_ANY_SOURCE, is specified as a rank within the process group associated with that same communicator (remote process group, for intercommunicators). Thus, the range of valid values for the source argument is \{0,...,n-1\}\cup\{MPI_ANY_SOURCE\}, where n is the number of processes in this group.

Note the asymmetry between send and receive operations: A receive operation may accept messages from an arbitrary sender, on the other hand, a send operation must specify a unique receiver. This matches a “push” communication mechanism, where data transfer is effected by the sender (rather than a “pull” mechanism, where data transfer is effected by the receiver).

Source = destination is allowed, that is, a process can send a message to itself. (However, it is unsafe to do so with the blocking send and receive operations described above, since this may lead to deadlock. See Section 3.5.)
3.2.5 Return Status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function (see Section 3.7.5), a distinct error code may need to be returned for each request. The information is returned by the status argument of MPI_RECV. The type of status is MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, status is a structure that contains three fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR; the structure may contain additional fields. Thus, status.MPI_SOURCE, status.MPI_TAG and status.MPI_ERROR contain the source, tag, and error code, respectively, of the received message.

An example:

```c
MPI_Status status;
MPI_Recv( ..., &status );
... status.MPI_TAG;
... status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &count );
MPI_TAG and MPI_SOURCE primarily of use when MPI_ANY_TAG and/or MPI_ANY_SOURCE in the receive.
MPI_Get_count may be used to determine how much data of a particular type was received.
```
An example:
program main
include 'mpif.h'
integer rank, size, to, from, tag, count, i, ierr
integer src, dest
integer st_source, st_tag, st_coun
integer status(MPI_STATUS_SIZE)
double precision data(100)
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'Process ', rank, ' of ', size, ' is alive'
dest = size - 1
src = 0
if (rank .eq. src) then to = dest
count = 10
tag = 2001
do 10 i=1, 10
10 data(i) = i
call MPI_SEND( data, count, MPI_DOUBLE_PRECISION, to, + tag, MPI_COMM_WORLD, ierr )
else if (rank .eq. dest) then
    tag = MPI_ANY_TAG
count = 10
    from = MPI_ANY_SOURCE
    call MPI_RECV(data, count, MPI_DOUBLE_PRECISION, from, + tag, MPI_COMM_WORLD, status, ierr )
call MPI_GET_COUNT( status, MPI_DOUBLE_PRECISION, + st_count, ierr )
st_source = status(MPI_SOURCE)
st_tag = status(MPI_TAG)
print *, rank, ' received', (data(i),i=1,10)
endif
call MPI_FINALIZE( ierr )
end
The MPI_PROBE and MPI_IPROBE 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 (basically, the information returned by status). In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

The MPICANCEL operation allows pending communications to be canceled. This is required for cleanup. Posting a send or a receive ties up user resources (send or receive buffers), and a cancel may be needed to free these resources gracefully.

**MPI_IPROBE**(source, tag, comm, flag, status)

<table>
<thead>
<tr>
<th>IN</th>
<th>source</th>
<th>rank of source or MPI_ANY_SOURCE (integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>tag</td>
<td>message tag or MPI_ANY_TAG (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>flag</td>
<td>(logical)</td>
</tr>
<tr>
<td>OUT</td>
<td>status</td>
<td>status object (Status)</td>
</tr>
</tbody>
</table>

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
                MPI_Status *status)
MPI_IPROBE(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 status the same value that would have been returned by MPI_RECV(). Otherwise, the call returns flag = false, and leaves status undefined.

If MPI_IPROBE returns flag = true, then the content of the status object can be subsequently accessed as described in Section 3.2.5 to find the source, tag and length of the probed message.

A subsequent receive executed with the same communicator, and the source and tag returned in status by MPI_IPROBE will receive the message that was matched by the probe, if no other intervening receive occurs after the probe, and the send is not successfully cancelled before the receive. If the receiving process is multi-threaded, it is the user’s responsibility to ensure that the last condition holds.
The source argument of MPI_PROBE 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.

It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received.

**MPI_PROBE**(source, tag, comm, status)

<table>
<thead>
<tr>
<th>IN</th>
<th>source</th>
<th>rank of source or MPI_ANY_SOURCE (integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>tag</td>
<td>message tag or MPI_ANY_TAG (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>status</td>
<td>status object (Status)</td>
</tr>
</tbody>
</table>

```c
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

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

The MPI implementation of MPI_PROBE and MPI_IPROBE needs to guarantee progress: if a call to MPI_PROBE has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to MPI_PROBE will return, unless the message is received by another concurrent receive operation (that is executed by another thread at the probing process). Similarly, if a process busy waits with MPI_IPROBE and a matching message has been issued, then the call to MPI_IPROBE will eventually return flag = true unless the message is received by another concurrent receive operation.
Example 3.18 Use blocking probe to wait for an incoming message.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
ELSE IF (rank.EQ.1) THEN
   CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
ELSE IF (rank.EQ.2) THEN
   DO i=1, 2
      CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                      comm, status, ierr)
      IF (status(MPI_SOURCE) .EQ. 0) THEN
         CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
      ELSE
         CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
      END IF
   END DO
END IF
```

Each message is received with the right type.
Example 3.19  A similar program to the previous example, but now it has a problem.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
ELSE IF (rank.EQ.2) THEN
    DO i=1, 2
        CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                      comm, status, ierr)
        IF (status(MPI_SOURCE) .EQ. 0) THEN
            CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,
                          0, comm, status, ierr)
        ELSE
            CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,
                          0, comm, status, ierr)
        END IF
    END DO
END IF
END IF
```

We slightly modified Example 3.18, using MPI_ANY_SOURCE as the source argument in the two receive calls in statements labeled 100 and 200. The program is now incorrect: the receive operation may receive a message that is distinct from the message probed by the preceding call to MPI_PROBE.
Simple Collective Operations

- MPI_Bcast(start, count, datatype, root, comm)
- MPI_Reduce(start, result, count, datatype, operation, root, comm)

The routine MPI_Bcast sends data from one process “root” to all others.
The routine MPI_Reduce combines data from all processes (by adding them in the following example), and returning the result to a single process, “root”.
#include "mpi.h"
#include <math.h>
int main(argc, argv)
int argc; char *argv[];
{
    int done = 0, n, myid, numprocs, i, rc;
double PI25DT = 3.141592653589793238462643;
double mypi, pi, h, sum, x, a;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
while (!done)
{
    if (myid == 0) { printf("Enter the number of intervals: (0 quits) ");
        scanf("%d", &n); }
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n == 0) break;
    h = 1.0 / (double) n; sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs)
{
    x = h * ((double)i - 0.5); sum += 4.0 / (1.0 + x*x);
}
    mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
}
MPI_Finalize();
}

Exercise: Run either program for Pi. Write new versions that replace the calls to MPI_Bcast and
MPI_Reduce with MPI_Send and MPI_Recv. Note that this code needs to be run in interactive mode if
that is available in the parallel system being used. Otherwise, a variation of this code in which “n” is
passed as a command line argument to the MPI code when executing it, needs to be used.