MPI_Send & MPI_Receive

MPI_Send

- MPI Send, whose syntax is:

```
int MPI_Send(
        void*            msg_buf p        /* in*/,
        int              msg_size         /* in*/,
        MPI_Datatype     msg_type         /* in*/,
        int              dest             /* in*/,
        int              tag              /* in*/,
        MPI_Comm         communicator     /* in*/, );
```

- The first three arguments, ***msg_buf_p***, ***msg_size***, and ***msg_type***, determine the contents of the message. The remaining arguments, dest, tag, and communicator, determine the destination of the message.

- The first argument, msg_buf_p, is a pointer to the block of memory containing the contents of the message. In our program, this is just the string containing the
message, greeting.

- The second and third arguments, msg_size and msg_type, determine the amount of data to be sent. In our program, the msg_size argument is the number of characters in the message plus one character for the '\0' character that terminates C strings. The msg type argument is MPI CHAR. These two arguments together tell the system that the message contains strlen(greeting)+1 chars.
- Since C types (int, char, and so on.) can't be passed as arguments to functions,
MPI defines a special type, MPI_Datatype.

Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

- The fourth argument, ***dest***, specifies the rank of the process that should receive
  the message.

- The fifth argument, ***tag***, is a nonnegative int. It can be used to distinguish messages that are otherwise identical. For example, suppose process 1 is sending floats to process 0. Some of the floats should be printed, while others should be used in a computation. Then the first four arguments to MPI Send provide no information regarding which floats should be printed and which should be used in a computation. So process 1 can use, say, a tag of 0 for the messages that should be printed and a tag of 1 for the messages that should be used in a computation.

- The final argument to MPI Send is a communicator. All MPI functions that involve
  communication have a communicator argument.

7) MPI Recv

- The first six arguments to MPI Recv correspond to the first six arguments of MPI Send:

```
int MPI Recv(
        void *          msg_buf_p       /*out*/,
        int             buf_size        /*in*/,
        MPI_Datatype    buf_ type       /*in*/,
        Int             source          /*in*/,
        int             tag             /*in*/,
        MPI_Comm        communicator    /*in*/,
        MPI_Status*     status p        /*out*/);
```

- Thus, the first three arguments specify the memory available for receiving the message: ***msg_buf_p*** points to the block of memory
- ***buf_size*** determines the number of objects that can be stored in the block
- ***buf type*** indicates the type of the objects.
- The next three arguments identify the message.
- The ***source*** argument specifies the process from which the message should be received.
- The ***tag*** argument should match the tag argument of the message being sent.
- The communicator argument must match the communicator used by the sending process.

8) Message matching

- Suppose process q calls MPI_Send with
  MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send _tag, send_comm);

- Also suppose that process r calls MPI_Recv with

  MPI_Recv (recv_buf_p, recv_buf_sz, recv_type, src, recv_ tag,recv_ comm, &status);

- Then the message sent by q with the above call to MPI_Send can be received by r with the call to MPI_Recv if
- recv_comm = send_comm,
- recv_tag = send_tag,
- dest = r,
- src = q.

- These conditions aren't quite enough for the message to be ***successfully*** received, however. The parameters specified by the first three pairs of arguments, send_buf_p/recv_buf_p, send_buf_sz/recv_buf_sz, and send_ type/recv_type, must specify compatible buffers.

- Most of the time, the following rule will suffice:

- If recv_type = send_type and recv_buf_sz>= send_ buf_ sz, then the message
  sent by q can be successfully received by r.

9) The status_p argument

- The MPI type MPI_Status is a struct with at least the three members MPI_SOURCE, MPI_TAG, and MPI_ERROR. Suppose our program contains the definition

  MPI_Status status;

- Then, after a call to MPI_Recv in which &status is passed as the last argument, we can determine the sender and tag by examining the two members:

  status.MPI_SOURCE
  status.MPI_TAG

- The amount of data that's been received isn't stored in a field that's directly accessible to the application program. However, it can be retrieved with a call to MPI_Get_count.
- For example, suppose that in our call to MPI_Recv, the type of the receive buffer is recv_type and, once again, we passed in &status. Then the call

  ```
  MPI_Get_count(&status, recv type, &count)
  ```

  will return the number of elements received in the count argument.
- In general, the syntax of MPI Get count is

  ```
  int MPI_Get_count(
          MPI_Status*       status_p        /*in*/,
          MPI_Datatype      type            /*in*/,
          Int*              count_ p        /*out*/);
  ```

- Note that the count isn't directly accessible as a member of the MPI_ Status variable simply because it depends on the type of the received data, and, consequently, determining it would probably require a calculation (e.g. (number of bytes received)=(bytes per object)). If this information isn't needed, we shouldn't waste a calculation determining it.

10) Semantics of MPI Send and MPI Recv

- What exactly happens when we send a message from one process to another? Many of the details depend on the particular system, but we can make a few generalizations.

- The sending process will assemble the message. For example, it will add the "envelope" information to the actual data being transmitted—the destination process rank, the sending process rank, the tag, the communicator, and some information on the size of the message.

- Once the message has been assembled, there are essentially two possibilities: the sending process can buffer the message or it can block.

- If it buffers the message, the MPI system will place the message (data and envelope) into its own internal storage, and the call to MPI_Send will return.

- Alternatively, if the system blocks, it will wait until it can begin transmitting
- the message, and the call to MPI_Send may not return immediately. Thus, if we use MPI_Send, when the function returns, we don't actually know whether the message has been transmitted.

- We only know that the storage we used for the message, the send buffer, is available for reuse by our program. If we need to know that the message has been transmitted, or if we need for our call to MPI Send to return immediately—regardless of whether the message has been sent—MPI provides alternative functions for sending.

- The exact behavior of MPI_Send is determined by the MPI implementation. However,typical implementations have a default "cutoff" message size. If the size of a message is less than the cutoff, it will be buffered.
- If the size of the message is greater than the cutoff, MPI Send will block.
  Unlike MPI_Send, MPI_Recv always blocks until a matching message has been received. Thus, when a call to MPI_Recv returns, we know that there is a message stored in the receive buffer (unless there's been an error).