MPI Point-to-Point Communication

CS 524 – High-Performance Computing

Definitions
- “Completion” means that memory locations used in the message transfer can be safely accessed
  - send: variable sent can be reused after completion
  - receive: variable received can now be used
- MPI communication modes differ in what conditions on the receiving end are needed for completion
- Communication modes can be blocking or non-blocking
  - Blocking: return from function call implies completion
  - Non-blocking: routine returns immediately, completion tested for

Communication Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Completion Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous send</td>
<td>Only completes when the receive has completed</td>
</tr>
<tr>
<td>Buffered send</td>
<td>Always completes (unless an error occurs), irrespective of receiver</td>
</tr>
<tr>
<td>Standard send</td>
<td>Message sent (receive state unknown)</td>
</tr>
<tr>
<td>Ready send</td>
<td>Always completes (unless an error occurs), irrespective of whether the receive has completed</td>
</tr>
<tr>
<td>Receive</td>
<td>Completes when a message has arrived</td>
</tr>
</tbody>
</table>

Blocking Communication Functions

<table>
<thead>
<tr>
<th>Mode</th>
<th>MPI Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard send</td>
<td>MPI_Send</td>
</tr>
<tr>
<td>Synchronous send</td>
<td>MPI_Ssend</td>
</tr>
<tr>
<td>Buffered send</td>
<td>MPI_Bsend</td>
</tr>
<tr>
<td>Ready send</td>
<td>MPI_Rsend</td>
</tr>
<tr>
<td>Receive</td>
<td>MPI_Recv</td>
</tr>
</tbody>
</table>

Sending a Message

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

- buf: starting address of the data to be sent
- count: number of elements to be sent (not bytes)
- datatype: MPI datatype of each element
- dest: rank of destination process
- tag: message identifier (set by user)
- comm: MPI communicator of processors involved

MPI_Send(data, 500, MPI_FLOAT, 5, 25, MPI_COMM_WORLD)
Receiving a Message

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

- **buf**: starting address of buffer where the data is to be stored
- **count**: number of elements to be received (not bytes)
- **datatype**: MPI datatype of each element
- **source**: rank of source process
- **tag**: message identifier (set by user)
- **comm**: MPI communicator of processors involved
- **status**: structure of information about the message that is returned

Example:

```c
MPI_Recv(buffer, 500, MPI_FLOAT, 3, 25, MPI_COMM_WORLD, status)
```

Standard and Synchronous Send

- **Standard send**
  - Completes once message has been sent
  - May or may not imply that message arrived
  - Don’t make any assumptions (implementation dependent)
- **Synchronous send**
  - Use if need to know that message has been received
  - Sending and receiving process synchronize regardless of who is faster. Thus, processor idle time is possible
  - Large synchronization overhead
  - Safest communication method

Ready and Buffered Send

- **Ready send**
  - Ready to receive notification must be posted; otherwise it exits with an error
  - Should not be used unless user is certain that corresponding receive is posted before the send
  - Lower synchronization overhead for sender as compared to synchronous send
- **Buffered send**
  - Data to be sent is copied to a user-specified buffer
  - Higher system overhead of copying data to and from buffer
  - Lower synchronization overhead for sender

Non-blocking Communications

- **Separates communication into three phases**:
  - Initiate non-blocking transfer
  - Do some other work not involving the data in transfer, i.e., overlap communication with calculation (latency hiding)
  - Wait for non-blocking communication to complete
- **Syntax of functions**
  - Similar to blocking functions’ syntax
  - Each function has an “I” immediately following the “_”. The rest of the name is the same
  - The last argument is a handle to an opaque request object that contains information about the message, i.e., its completion status

Non-blocking Communication Functions

<table>
<thead>
<tr>
<th>Mode</th>
<th>MPI Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard send</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td>Synchronous send</td>
<td>MPI_Issend</td>
</tr>
<tr>
<td>Buffered send</td>
<td>MPI_Ibsend</td>
</tr>
<tr>
<td>Ready send</td>
<td>MPI_Irsend</td>
</tr>
<tr>
<td>Receive</td>
<td>MPI_Irecv</td>
</tr>
</tbody>
</table>

Sending and Receiving a Message

```c
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request request)
```

- **request**: a request handle is allocated when a communication is initiated. Used to test if communication has completed.
- **Other parameters** have the same definitions as for blocking functions
Blocking and Non-blocking

- Send and receive can be blocking or non-blocking
- A blocking send can be used with a non-blocking receive, and vice-versa
- Non-blocking sends can use any mode – synchronous, buffered, standard, or ready
  - No advantage for buffered or ready
- Characteristics of non-blocking communications
  - No possibility of deadlocks
  - Decrease in synchronization overhead
  - Increase or decrease in system overhead
  - Extra computation and code to test and wait for completion
  - Must not access buffer before completion

For a Communication to Succeed

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Receiver’s buffer must be large enough
- User-specified buffer should be large enough (buffered send only)
- Receive posted before send (ready send only)

Completion Tests

- Waiting and Testing for completion
  - Wait: function does not return until completion finished
  - Test: function returns a TRUE or FALSE value depending on whether or not the communication has completed

  ```c
  int MPI_Wait(MPI_Request *request, MPI_Status *status)
  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
  ```

Testing Multiple Communications

- Test or wait for completion of one (and only one) message
  - MPI_Waitany & MPI_Testany
- Test or wait for completion of all messages
  - MPI_Waitall & MPI_Testall
- Test or wait for completion of as many messages as possible
  - MPI_Waitsome & MPI_Testsome

Wildcarding

- Receiver can wildcard
- To receive from any source specify MPI_ANY_SOURCE as rank of source
- To receive with any tag specify MPI_ANY_TAG as tag
- Actual source and tag are returned in the receiver’s status parameter

Receive Information

- Information of data is returned from MPI_Recv (or MPI_Irecv) as status
- Information includes:
  - Source: status.MPI_SOURCE
  - Tag: status.MPI_TAG
  - Error: status.MPI_ERROR
  - Count: message received may not fill receive buffer. Use following function to find number of elements actually received:
    - `int MPI_Get_count(MPI_Status status, MPI_Datatype datatype, int *count)`
- Message order preservation: messages do not overtake each other. Messages are received in the order sent.
Timers

```c
double MPI_Wtime(void)
```

- Time is measured in seconds
- Time to perform a task is measured by consulting the timer before and after

Deadlocks

- A deadlock occurs when two or more processors try to access the same set of resources
- Deadlocks are possible in blocking communication
  - Example: Two processors initiate a blocking send to each other without posting a receive

```
... ...
MPI_Send(P1)
MPI_Recv(P1)
...

Process 0
```
```
... ...
MPI_Send(P0)
MPI_Recv(P0)
...

Process 1
```

Avoiding Deadlocks

- Different ordering of send and receive: one processor post the send while the other posts the receive
- Use non-blocking functions: Post non-blocking receives early and test for completion
- Use MPI_Sendrecv and MPI_Sendrecv_replace: Both processors post the same single function
- Use buffered mode: Use buffered sends so that execution continues after copying to user-specified buffer