MPI Point-to-Point Communication I

Michael Hanke

Summer School on High Performance Computing

Outline

What You Already Know

Basic ideas:

- The 6 basic calls
- MPI messages
- Communicators

What Is Covered in This Module?

- Point to point is sending a message from one process to another
- Most message passing libraries define one system behavior
- MPI defines four communication modes
    - synchronous mode ("safest")
    - ready mode (lowest system overhead)
    - buffered mode (decouples sender from receiver)
    - standard mode (compromise)
- Communication mode is selected with send routine
- Calls are also blocking or nonblocking.
    - Blocking stops the program until the message buffer is safe to use
    - Non-blocking separates communication from computation

Blocking Send

C

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

Fortran

```
MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)
```

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
ierror	is a return error code

Blocking Receive

C

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

Fortran

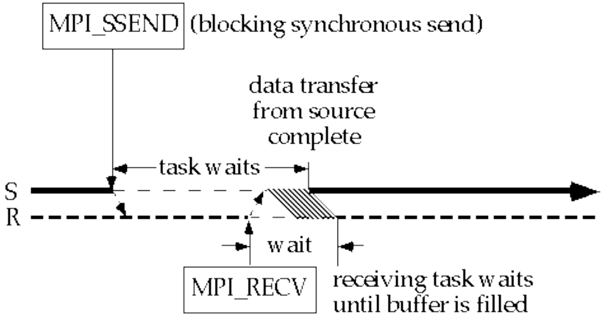
```
MPI_RECV(buf, count, datatype, source, tag, comm, status, ierror)
```

`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.
`ierror`	is a return error code

Communication Modes

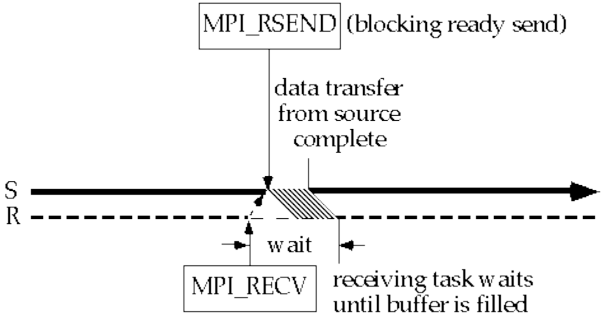
Communication Mode	Blocking Routines	Non-Blocking Routines
synchronous	MPI_SSEND	MPI_ISSEND
ready	MPI_RSEND	MPI_IRSEND
buffered	MPI_BSEND	MPI_IBSEND
standard	MPI_SEND	MPI_ISEND
	MPI_RECV	MPI_IRECV
	MPI_SENDRECV	
	MPI_SENDRECV_REPLACE	

Blocking Synchronous Send


MPI_SSEND
(blocking synchronous send)
data transfer from source complete
task waits
S
R
wait
MPI_RECV
receiving task waits until buffer is filled

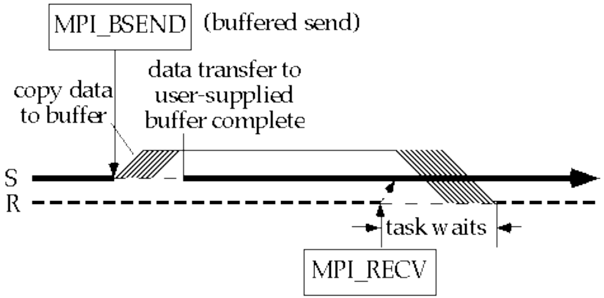
Most of wait on sending end is eliminated if `MPI_Recv` comes before `MPI_Ssend`.

Blocking Ready Send


MPI_RSEND
(blocking ready send)
data transfer from source complete
S
R
wait
MPI_RECV
receiving task waits until buffer is filled

By default, the program exits if `MPI_Rsend` is called before notification arrives

Blocking Buffered Send



Timing of `MPI_Recv` is irrelevant. `MPI_Bsend` returns as soon as data are copied from source to a buffer.

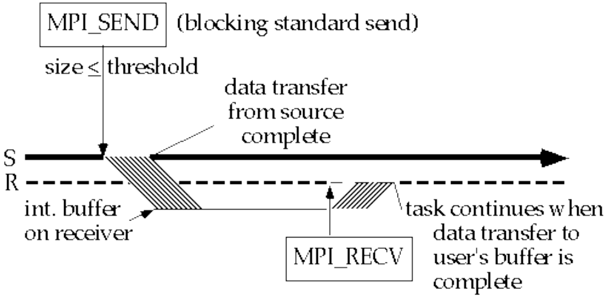
Blocking Standard Send

Behavior depends on message size:

Number of tasks	Eager Limit (bytes) =Threshold
1-16	4096
17-32	2048
33-64	1024
65-128	512

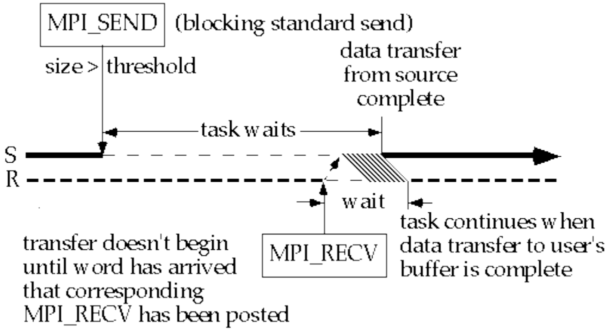
Note: These values are examples for an old MPI implementation by IBM.

Blocking Standard Send (Short Messages)


MPI_SEND
(blocking standard send)
size ≤ threshold
data transfer from source complete
S
R
int. buffer on receiver
MPI_RECV
task continues when data transfer to user's buffer is complete

Exceeding system buffer space causes communication to stall until space is freed.

Standard Blocking Send (Long Messages)


MPI_SEND
(blocking standard send)
size > threshold
data transfer from source complete
task waits
S
R
wait
task continues when data transfer to user's buffer is complete
transfer doesn't begin until word has arrived that corresponding MPI_RECV has been posted
MPI_RECV

Blocking Send And Receive

Send and Receive can be combined into one call.

MPI_SENDRECV

- blocking send and receive
- different buffers for send and receive

MPI_SENDRECV_REPLACE

- blocking send and receive
- only one buffer; the received message overwrites the sent one

Conclusions: Synchronous Mode

Advantages

- Safest, and therefore most portable
- SEND/RECV order not critical
- Amount of buffer space irrelevant

Disadvantages

- Can incur substantial syncronization overhead

Conclusions: Ready Mode

Advantages

- Lowest total overhead
- SEND/RECV handshake not required

Disadvantages

- RECV must precede SEND

Conclusions: Buffered Mode

Advantages

- Decouples SEND from RECV
- No sync overhead on SEND
- Order of SEND/RECV irrelevant
- Programmer can control size of buffer space

Disadvantages

- Additional system overhead incurred by copy to buffer

Conclusions: Standard Mode

Advantages

- Good for many cases

Disadvantages

- Your program may not be suitable

Syntax of Non-Blocking Calls

C

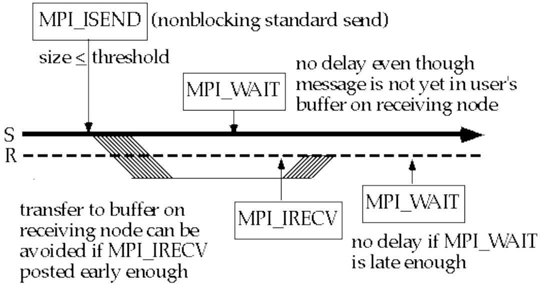
```
MPI_Isend(buf, count, dtype, dest, tag, comm,
          request)
MPI_Wait(request, status)
```

Fortran

```
MPI_ISEND(buf, count, dtype, dest, tag, comm,
          request, ierror)
MPI_WAIT(request, status, ierror)
```

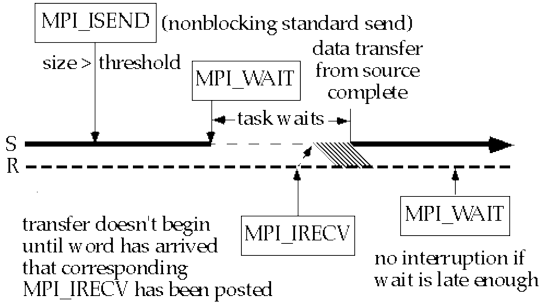
Example: Non-Blocking Standard Send

- Non-blocking standard send, message size <= threshold
- Non-blocking receive


MPI_ISEND
(nonblocking standard send)
size ≤ threshold
MPI_WAIT
no delay even though message is not yet in user's buffer on receiving node
S
R
transfer to buffer on receiving node can be avoided if MPI_IRECV posted early enough
MPI_IRECV
MPI_WAIT
no delay if MPI_WAIT is late enough

Example: Non-Blocking Standard Send, Large Message

- Non-blocking standard send, message size > threshold
- Non-blocking receive


MPI_ISEND
(nonblocking standard send)
size > threshold
MPI_WAIT
data transfer from source complete
task waits
S
R
transfer doesn't begin until word has arrived that corresponding MPI_IRECV has been posted
MPI_IRECV
MPI_WAIT
no interruption if wait is late enough

Conclusions: Non-Blocking Calls

- Gains
  - Avoid Deadlock
  - Decrease Synchronization Overhead
  - Some Systems: Reduce Systems Overhead
- Best to post non-blocking sends and receives as early as possible, and to do waits as late as possible
- 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`

Programming Recommendations

In general: start with non-blocking calls, standard mode

Blocking calls

- use if you want tasks to synchronize
- use if wait immediately follows communication call
- start with synchronous, then switch to standard mode

Evaluate performance and analyze code

- if non-blocking receives are posted early, might consider ready mode
- if there is too much synchronization overhead on sends, could switch to buffered mode

Summary

- MPI defines four communication modes: synchronous, ready, buffered, standard
- Communication mode is selected with send routine
- Calls are also blocking or nonblocking.

A careful selection of the communication strategy is crucial for the efficiency of a parallel programme.

- What comes next?
    - How to control the point-to-point communication?
    - Programming recommendations