# Basic MPI
# Point-to-Point Communication

Erwin Laure
*Director PDC*

## What we know already

- MPI program structure
- Communicators and ranks
- Syntax of MPI commands
- 6 basic MPI commands

## Contents

- Sending data from A to B
  - Message format
  - Buffers and semantics
  - Communication modes

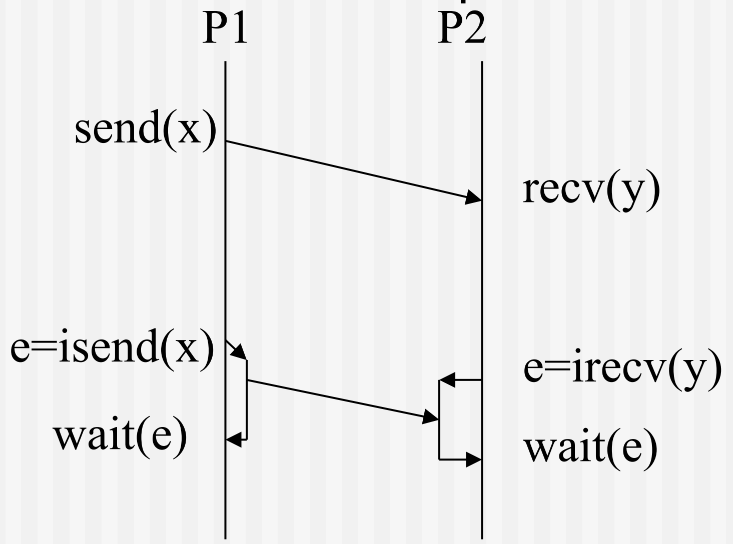
-------------------- TUESDAY -----------------------

- Deadlocks

- Blocking and non-blocking communication

## Sending Data from A to B …

- The basic function of any message passing library
  - Typically a SEND/RECEIVE pair
- Needed when process X needs data from process Y
- Two main incarnations
  - Blocking: stops the program until it is safe to continue
  - Non-blocking: separates communication from computation



# Basic MPI Message Syntax

- An MPI message consists of an **envelope** and **message body** – think of it like a letter in the mail:

- The **envelope** of an MPI message has four parts:
  - **Source** — the sending process
  - **Destination** — the receiving process
  - **Communicator** — specifies a group of processes to which both source and destination belong
  - **Tag** — used to classify messages

- The **message body** has three parts:
  - **Buffer** — the message data
  - **Datatype** — the type of the message data
  - **Count** — the number of items of type datatype in buffer

# Basic Send/Receive Commands

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

```
MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, IERR)
```

$$\left.\begin{array}{l}\text{Buffer}\\\text{Count}\\\text{Datatype}\end{array}\right\}\text{Body}\qquad\left.\begin{array}{l}\text{Destination}\\\text{Tag}\\\text{Communicator}\end{array}\right\}\text{Envelope}$$

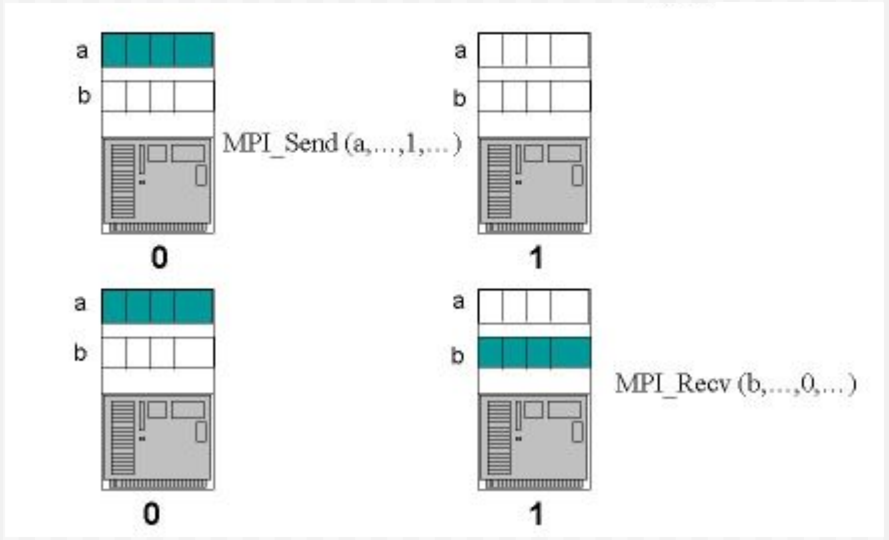
```
int MPI_Recv(void *buf, int count, MPI_Datatype
dtype, int source, int tag, MPI_Comm comm, MPI_Status
*status);
```

```
MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM,
STATUS, IERR)
```

## Example

```
double a[100],b[100];

if( myrank == 0 )          /* Send a message */
{
  for (i=0;i<100;++i)
     a[i]=sqrt(i);
  MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
}
else if( myrank == 1 )     /* Receive a message */
  MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
```

What happens if `b` is replaced with `a`?



## Wildcards

- Instead of specifying everything in the envelope explicitly, wildcards can be used for sender and tag:

  `MPI_ANY_SOURCE` and `MPI_ANY_TAG`

- Actual source and tag are stored in `STATUS` variable

C:

```
MPI_Status status;
MPI_Recv(b, 100, MPI_DOUBLE,
         MPI_ANY_SOURCE, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status );

source = status.MPI_SOURCE;
tag = status.MPI_TAG;
```

## Wildcards cont'd

- Fortran:

```
integer status(MPI_STATUS_SIZE)
call MPI_RECV(b, 100, MPI_DOUBLE_PRECISON,
         MPI_ANY_SOURCE, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status );

tag = status(MPI_TAG)
source = status(MPI_SOURCE)
```

## Message Size

- Semantics of receiving buffer is that it has to be at least as large as the message to be received – the actual data received might be smaller!

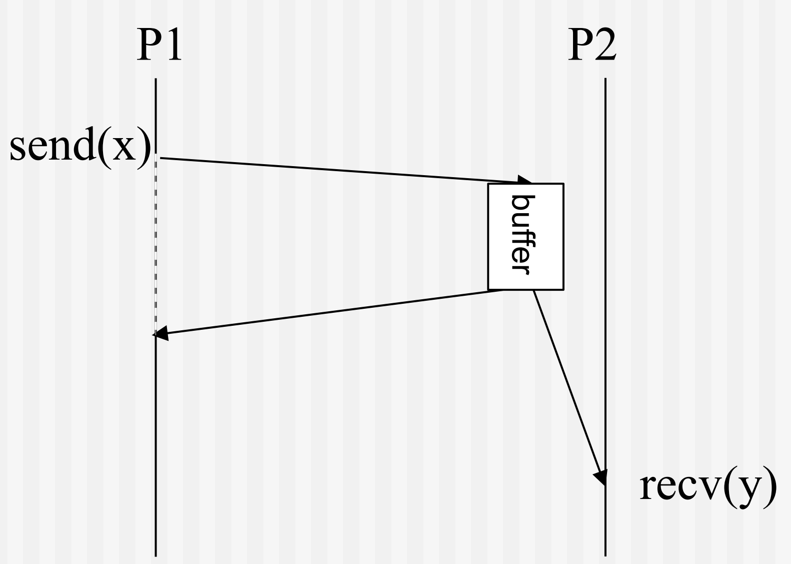
- Again, actual information is stored in `STATUS` variable:

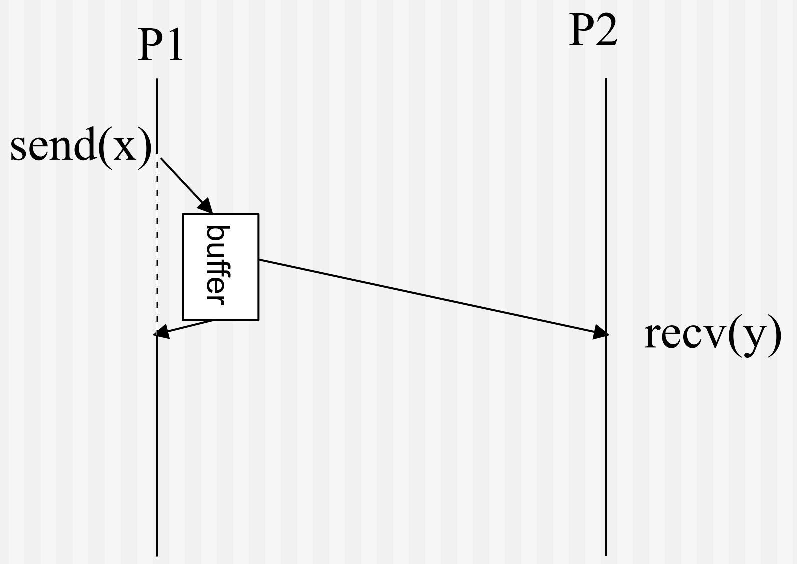
```
int MPI_Get_count(MPI_Status *status,
    MPI_Datatype dtype, int *count);
```

## End of Today’s Lecture

# A Word on Buffering

- MPI implementations typically use (**internal**) message buffers
  - Sending process can safely modify the sent data once it is copied into the buffer, irrespectively of status of receiving process
  - Receiving process can buffer incoming messages even if no (user space) receiving buffer is provided, yet
  - Buffers can be on both sides





## Note

This system buffer is **DIFFERENT** to the message buffer you specify in the `MPI_Send` or `MPI_Recv` calls!

## A Word on Buffering Cont’d

- The efficiency of MPI implementations critically depends on how buffers are being handled
  - A great source for optimization
  - Out of scope for this lecture
- Different handling of buffers can show different effects – hard to debug!
  - E.g. while in general no handshake between sending and receiving process is needed (i.e sending process may continue after data is copied into buffer even if no matching receive has been posted, yet) large messages or lack of buffering space may require synchronization with receiving process
  - Sometimes explicit buffers are required (see later) and lack of sufficient buffer space will cause the communication to fail.

## Blocking and Completion

- Both `MPI_Send` and `MPI_Recv` are blocking
  - They program only continues after they are completed
- The command is completed once it is safe to (re)use the data
  - `MPI_Recv`: data has been fully received
  - `MPI_Send`: can be completed even if no non-local action has been taking place. WHY?
  - Once data is copied into a send buffer `MPI_Send` can complete

## Message Order

- MPI messages are non-overtaking
  - If the sender sends two messages to the same destination they have to be received in the same order

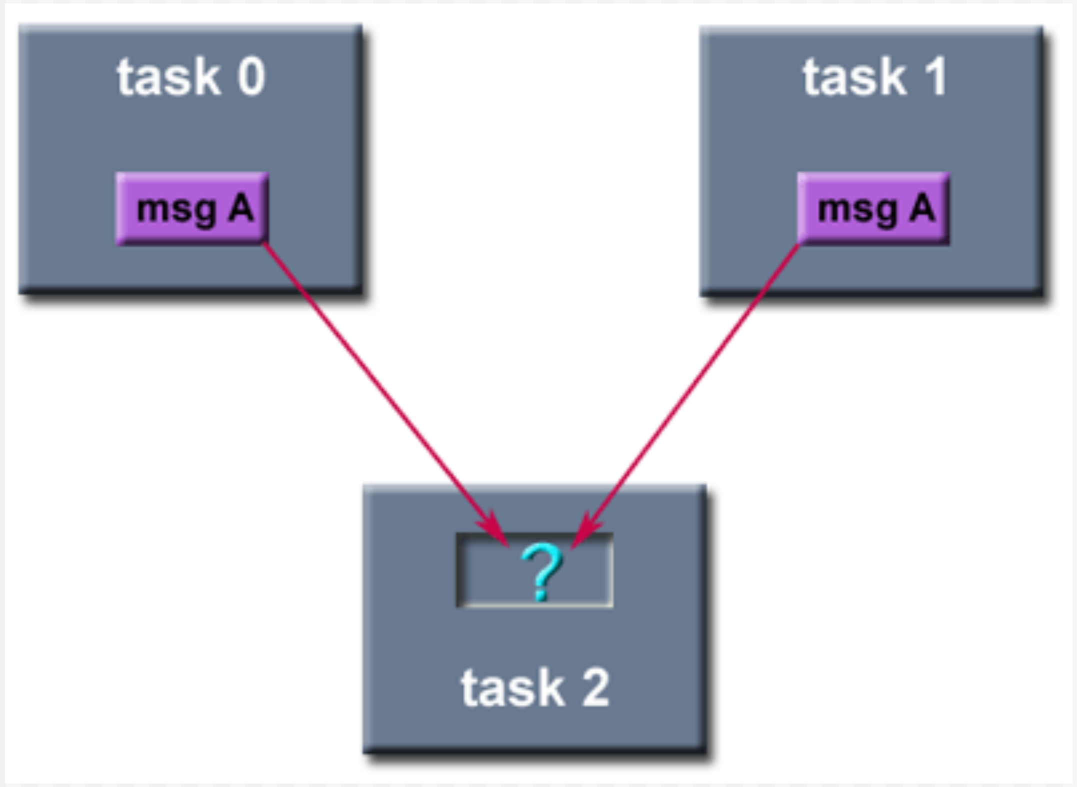
```
IF (rank.EQ.0) THEN
  CALL MPI_SEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)

ELSE    ! Rank.EQ.1

  CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG,
                comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag2, comm,
                status, ierr)
END IF
```

## Deadlock or not?

```
IF (rank.EQ.0) THEN
  CALL MPI_SEND(buf1, count, MPI_REAL, 1, tag1, comm,
                ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm,
                ierr)

ELSE    ! rank.EQ.1

  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm,
                status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm,
                status, ierr)

END IF
```

## Fairness

- MPI makes no guarantees about fairness
  - If there are two matching sends (from different sources) for a receive any of these can be successful
  - MPI does not prevent operation starvation (e.g. sends that will never be picked up)



## What have we learned?

- The semantics of `MPI_Send/MPI_Recv` are quite implementation dependent

- How can we get more control on what is actually happening?
  - MPI provides different communication modes with different semantics

## MPI Communication Modes

- Synchronous mode
  - Syntax: `MPI_Ssend(…)`
  - Semantics: handshake required, send will block until matching receive has been posted and receiving has started
- Ready mode
  - Syntax: `MPI_Rsend(…)`
  - Semantics: user guarantees that matching receive has already been posted; similar to synchronous but no need for handshake
- Buffered mode
  - Syntax: `MPI_Bsend(…)`
  - Semantics: send buffer will be used and command returns once data is locally copied; send buffer needs to be provided by user

## Discussion

- Standard `MPI_Send(…)` behaves like `MPI_Bsend` or `MPI_Ssend` depending on message size and internal buffer space
- For portability and safety reasons you should always assume `MPI_Ssend` semantics
  - Don’t assume `MPI_Send(…)` will return irrespectively of matching receive status

## Discussion Cont'd

- `MPI_Bsend` will fail if not enough buffer space is available
  - You must provide sufficient buffer space in user space to an MPI process:

```
int MPI_Buffer_attach( void* buffer, int size)
MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR)

int MPI_Buffer_detach( void* buffer_addr, int* size)
MPI_BUFFER_DETACH( BUFFER_ADDR, SIZE, IERROR)
```

- This buffer is only used for buffered send and detach will block until all data is actually sent.

## Pros and Cons of different modes

| Advantages | Disadvantages |
|---|---|
| **Synchronous Mode** | |
| Safest, most portable | Can occur substantial synchronization overhead |
| **Ready Mode** | |
| Lowest total overhead | Difficult to guarantee that receive precedes send |
| **Buffered Mode** | |
| Decouples send from receive | Potentially substantial overhead through buffering |
| **Standard Mode** | |
| Most flexible, general purpose | Implementation dependent |

## Deadlocks

- Deadlocks are common (and hard to debug) errors in message passing programs
- A deadlock occurs when two (or more) processes wait on the progress of each other:

```
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 );
```

# How to avoid Deadlocks?

- Careful organize the communication in your program
  - Make sure sends are always paired with receives in the correct order
  - A difficult task in large programs!

- Don’t depend on how specific implementations handle their internal buffers
  - A program may work well with certain problem sizes but deadlock once you increase the problem size or move to a different architecture or MPI implementation because of internal buffer limitations

## Communication modes revisited

DEAD LOCK

```
IF (rank.EQ.0) THEN
  CALL MPI_SSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! rank.EQ.1
  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

SYS DEP.

```
IF (rank.EQ.0) THEN
  CALL MPI_SEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! rank.EQ.1
  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

OK (If ...)

```
IF (rank.EQ.0) THEN
  CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! rank.EQ.1
  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

## Help to avoid Deadlock

- Careful ordering of send/receives is facilitated by a combined send/receive command:

```
int MPI_Sendrecv( void *sendbuf, int sendcount,
                  MPI_Datatype sendtype,
                  int dest, int sendtag,
                  void *recvbuf, int recvcount,
                  MPI_Datatype recvtype,
                  int source, int recvtag, MPI_Comm
                  comm, MPI_Status *status )
```

- Advantage: order of send/receive irrelevant; receive will not be blocked by potentially blocking send
- Very useful for shift operations

## Sendrcv Example

```
if (myid == 0) then
   call mpi_send(a,1,mpi_real,1,tag,MPI_COMM_WORLD,ierr)
   call mpi_recv(b,1,mpi_real,1,tag,MPI_COMM_WORLD,
                 status,ierr)
elseif (myid == 1) then
   call mpi_send(b,1,mpi_real,0,tag,MPI_COMM_WORLD,ierr)
   call mpi_recv(a,1,mpi_real,0,tag,MPI_COMM_WORLD,
                 status,ierr)
end if

if (myid == 0) then
   call mpi_sendrecv(a,1,mpi_real,1,tag1,
                     b,1,mpi_real,1,tag2,
                     MPI_COMM_WORLD, status,ierr)
elseif (myid == 1) then
   call mpi_sendrecv(b,1,mpi_real,0,tag2,
                     a,1,mpi_real,0,tag1,
                     MPI_COMM_WORLD, status,ierr)
end if
```

## Help to avoid Deadlocks Cont'd

- Careful message ordering
  - Always a good idea!
- Buffered communication
  - But comes with (quite substantial) overhead
- Non-blocking calls

## Non-blocking Communication

- For all send/receive calls there is a non-blocking equivalent named `I(x)send/Irecv`

- Non-blocking calls will return immediately irrespectively of the send/receive status
  - They actually only **initiate** the action
  - Actual sending/receiving of messages will be handled internally in the MPI implementation
  - Calls return a handle that allows to check the progress of sending/receiving

- Blocking and non-blocking calls can be intermixed
  - A blocking receive can match a non-blocking send and vice-versa.

# Non-blocking Syntax

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

```
MPI_ISEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, REQ, IERR)
```

- Request is the handle to the request

- **Important**: None of the arguments passed to `MPI_ISEND` should be read or written until the send operation is completed.

## Completion of non-blocking send/receives

```
int MPI_Wait( MPI_Request *request, MPI_Status
*status );
MPI_WAIT(REQUEST, STATUS, IERR )
```

- `MPI_Wait` is blocking and will only return when the message has been sent/received
  - After `MPI_Wait` returns it is safe to access the data again

```
int MPI_Test( MPI_Request *request, int *flag,
              MPI_Status *status );
MPI_TEST(REQUEST, FLAG, STATUS, IERR)
```

- MPI_Test returns immediately
  - Status of request is returned in flag (true for done, false when still ongoing)

## Example

```
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 );
}
```

- No deadlock because non-blocking receive is posted before send

## Discussion

- Non-blocking communication has two main benefits:
  - Helps avoid deadlocks
  - Allows to overlap communication with computation (latency hiding)
    - More about that later on
- Disadvantage:
  - Makes code more complex to read/understand and thus debug and maintain.

## Summary

- MPI provides blocking and non-blocking communication
- 4 communication modes

- You should now be able to program message passing applications
- Everything you want to do can be done with the (6) basic commands you know now.
  - But many tasks would be awkward and inefficient – hence the lecture continues

- Beware deadlocks!