High Performance Computing

R. Guivarc'h

Ronan.Guivarch@toulouse-inp.fr

2025 - ModIA

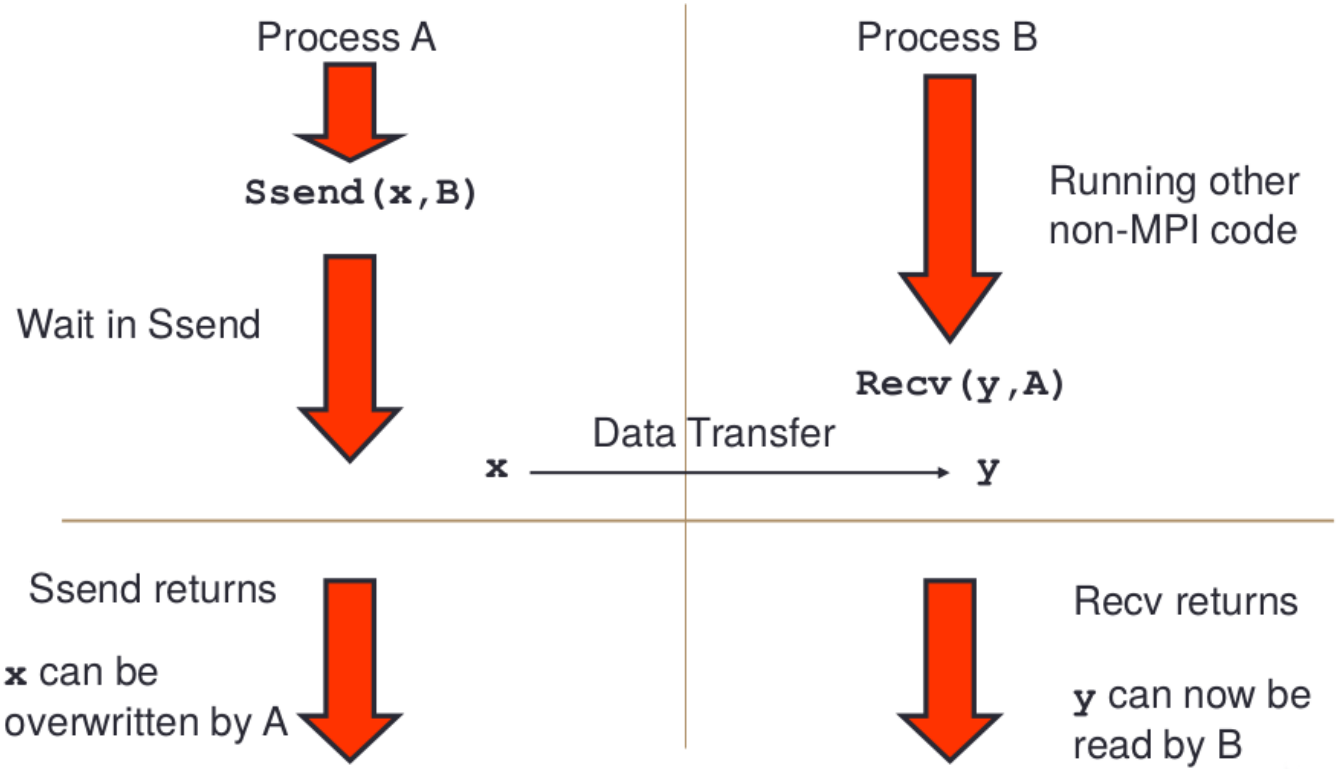
6. Synchronous and Asynchronous Modes

Synchronous and Asynchronous Modes

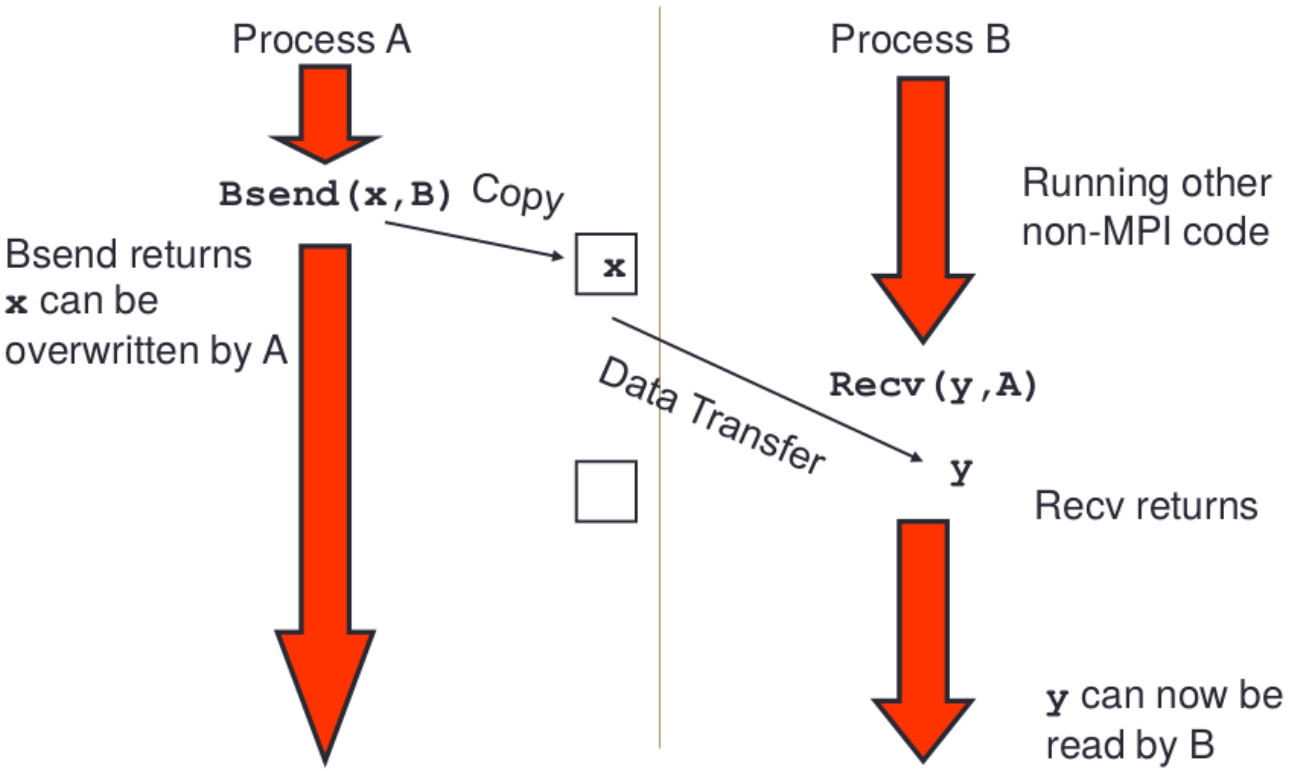
MPI Modes

- MPI_Ssend (Synchronous Send)
  - guaranteed to be synchronous
  - routine will not return until message has been delivered
- MPI_Bsend (Buffered Send)
  - guaranteed to be asynchronous
  - routine returns before the message is delivered
  - system copies data into a buffer and sends it later on
- MPI_Send (standard Send)
  - may be implemented as synchronous or asynchronous send
  - this causes a lot of confusion (see later)

MPI_Ssend


Process A
Ssend(x,B)
Wait in Ssend
Process B
Running other non-MPI code
Recv(y,A)
Data Transfer
x
y
Ssend returns
x can be overwritten by A
Recv returns
y can now be read by B

MPI_Bsend


Process A
Process B
Bsend(x,B)
Copy
Bsend returns
x can be
overwritten by A
x
Running other
non-MPI code
Recv(y,A)
Data Transfer
y
Recv returns
y can now be
read by B

Bsend: notes

- `Recv` is always synchronous
  - if process B issued `Recv` before the `Bsend` from process A, then B would wait in the `Recv` until `Bsend` was issued
- Where does the buffer space come from?
  - for `Bsend`, the user provides a single large block of memory
  - make this available to MPI using `MPI_Buffer_attach`
- If A issues another `Bsend` before the `Recv`
  - system tries to store message in free space in the buffer
  - if there is not enough space then BSEND will FAIL!

MPI_Send

- Problems
  - `Ssend` runs the risk of deadlock
  - `Bsend` less likely to deadlock, and your code may run faster, but
    - the user must supply the buffer space
    - the routine will FAIL if this buffering is exhausted
- `MPI_Send` tries to solve these problems
  - buffer space is provided by the system
  - `Send` will normally be asynchronous (like `Bsend`)
  - if buffer is full, `Send` becomes synchronous (like `Ssend`)
- `MPI_Send` routine is unlikely to fail
  - but could cause your program to deadlock if buffering runs out

MPI_Send

- This code is **NOT** guaranteed to work
  - will deadlock if `Send` is synchronous
  - is guaranteed to deadlock if you use `Ssend`!

Examples with `06_Send_Recv`

Solutions

- To avoid deadlock
  - either match sends and receives explicitly
  - e.g. for ping-pong
    - process A sends then receives
    - process B receives then sends
- For a more general solution use non-blocking communications (see later)

7. Non Blocking Communications

Non Blocking Communications

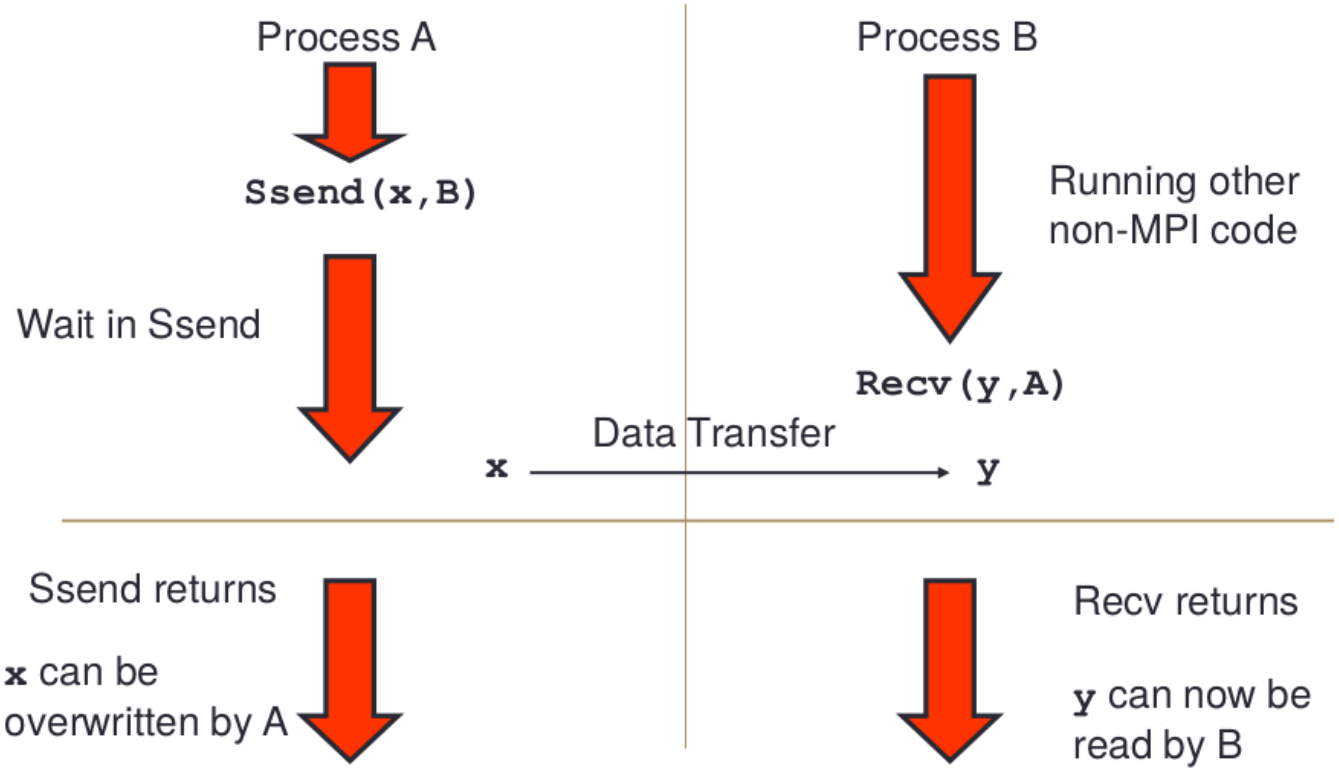
Completion

- The **mode** of a communication determines when its constituent operations complete
  - i.e. synchronous / asynchronous
- The **form** of an operation determines when the procedure implementing that operation will return
  - i.e. when control is returned to the user program

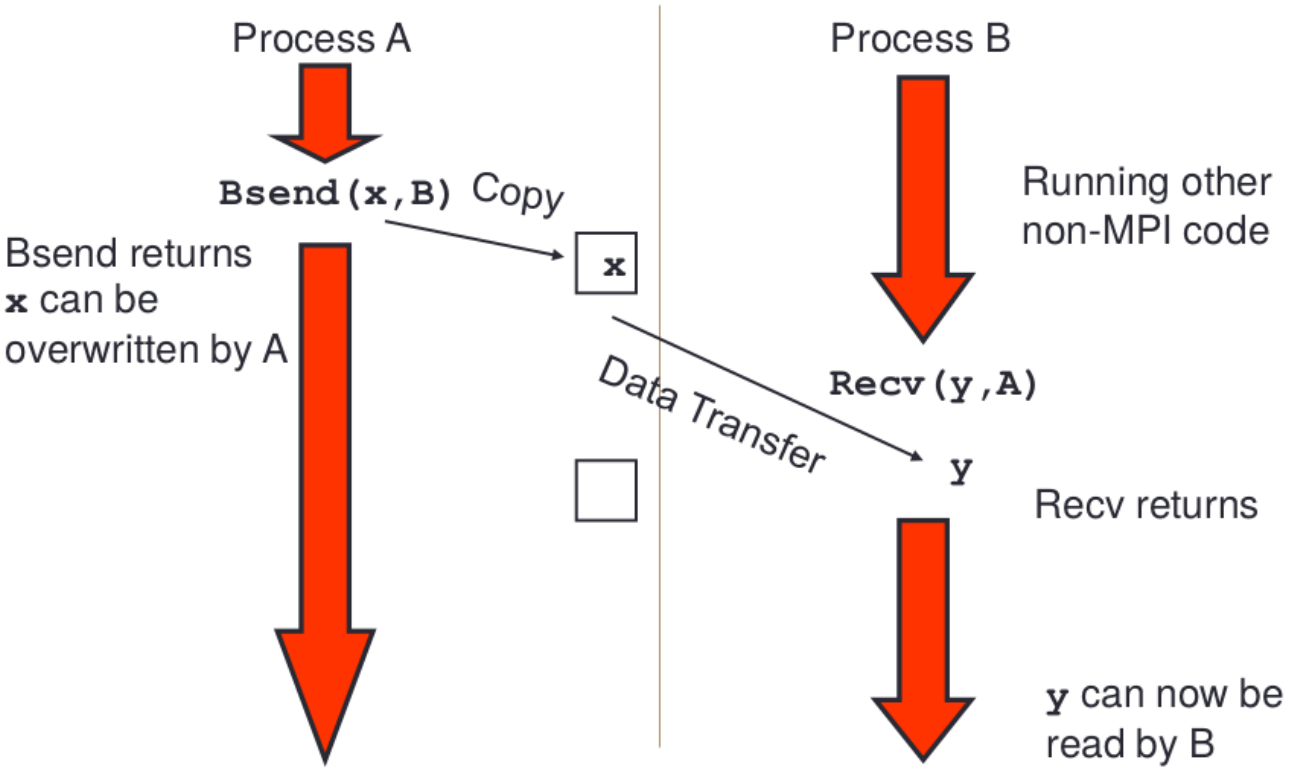
Blocking Operations

- Relate to when the operation has completed
- Only return from the subroutine call when the operation has completed
- These are the routines you used thus far
  - `MPI_Ssend`
  - `MPI_Bsend`
  - `MPI_Recv`

MPI_Ssend

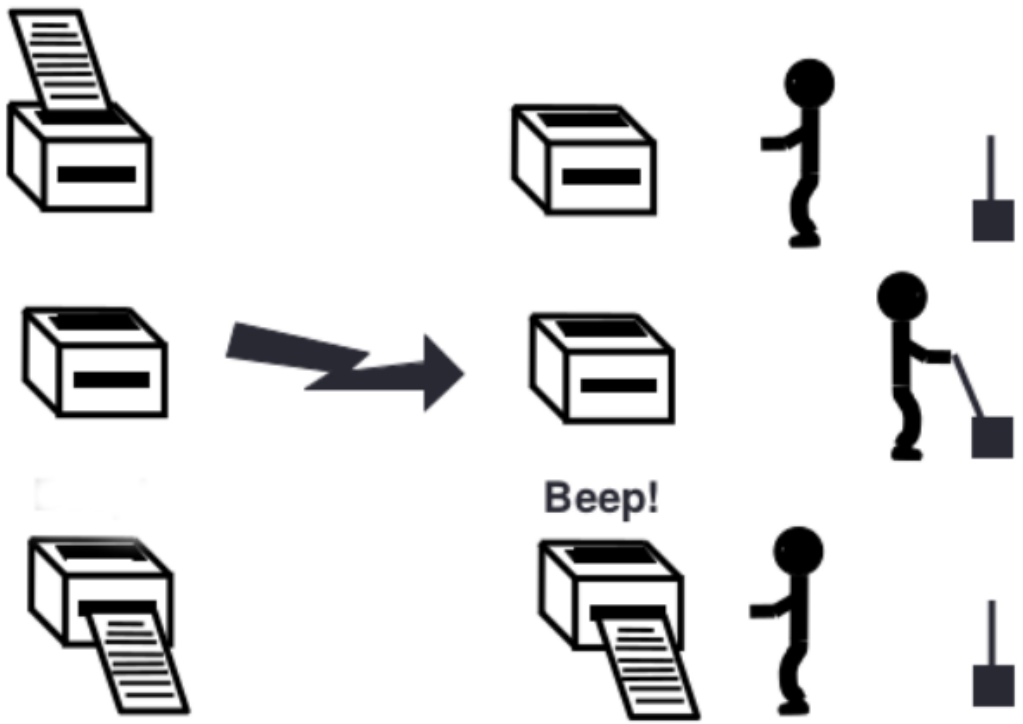

Process A
Ssend(x,B)
Wait in Ssend
Process B
Running other non-MPI code
Recv(y,A)
Data Transfer
x
y
Ssend returns
x can be overwritten by A
Recv returns
y can now be read by B

MPI_Bsend


Process A
Process B
Bsend(x,B)
Copy
Bsend returns
x can be
overwritten by A
x
Running other
non-MPI code
Recv(y,A)
Data Transfer
y
Recv returns
y can now be
read by B

Non-Blocking Operations

- Return straight away and allow the user program to continue to perform other work
- At some later time the user program can **test** or **wait** for the completion of the non-blocking operation



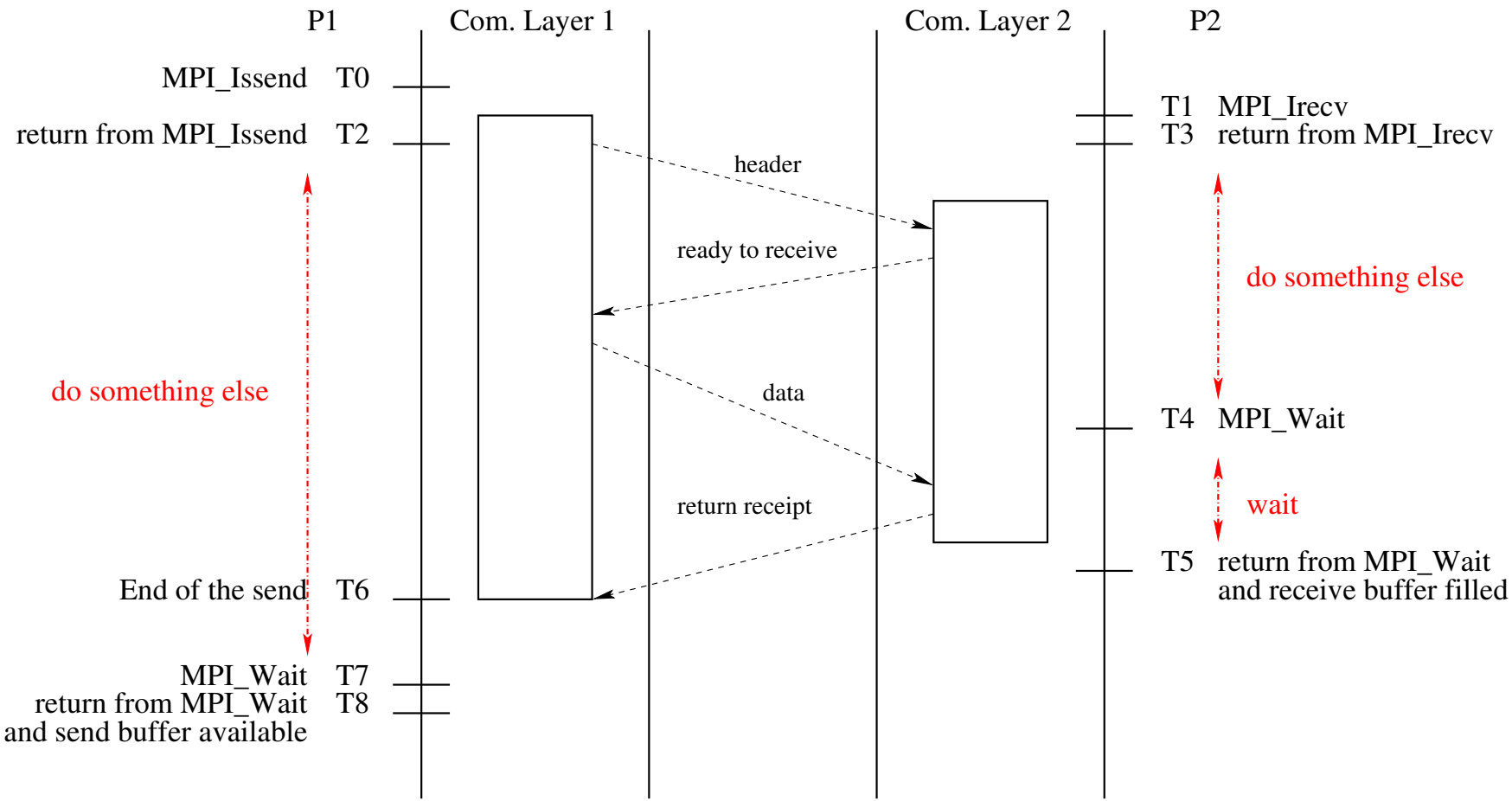
Non-Blocking Operations

- All non-blocking operations should have matching wait operations. Some systems cannot free resources until wait has been called
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation
- Non-blocking operations are not the same as sequential subroutine calls as the operation continues after the call has returned

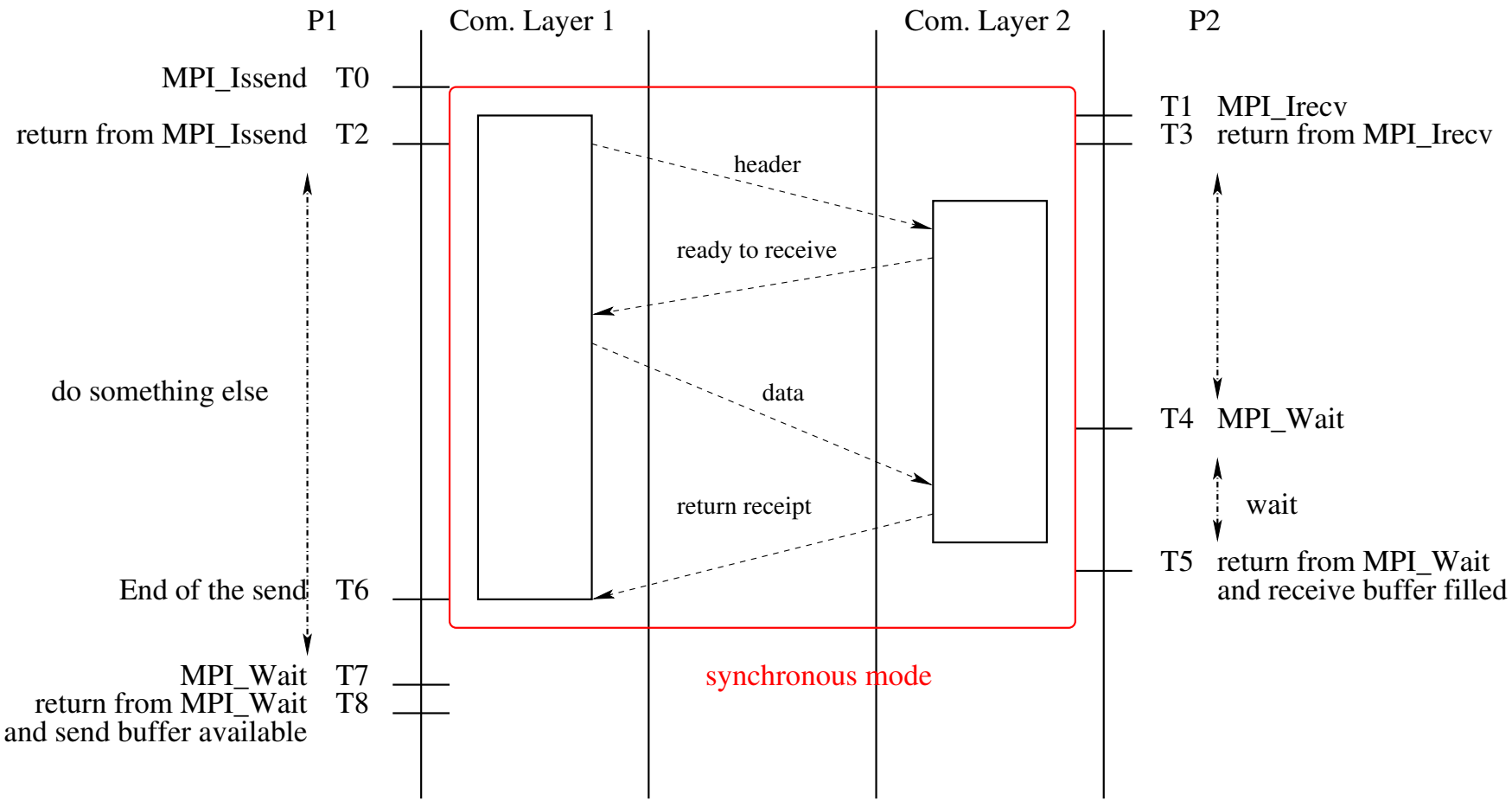
Non-Blocking Communications

- Separate communication into three phases:
  1. Initiate non-blocking communication
  2. Do some work (perhaps involving other communications?)
  3. Wait for non-blocking communication to complete

Non-Blocking Communication (send and receive)



Mode vs. Form


P1
Com. Layer 1
Com. Layer 2
P2
MPI_Issend T0
return from MPI_Issend T2
T1 MPI_Irecv
T3 return from MPI_Irecv
header
ready to receive
data
return receipt
do something else
T4 MPI_Wait
wait
T5 return from MPI_Wait
and receive buffer filled
End of the send T6
MPI_Wait T7
return from MPI_Wait T8
and send buffer available
synchronous mode

Handles used for Non-blocking Comms

- datatype same as for blocking (`MPI_Datatype`)
- communicator same as for blocking (`MPI_Comm`)
- A request handle is allocated when a communication is initiated (`MPI_Request`)

Non-blocking Synchronous Send

"I" stands for *Immediate*

- C:

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

Non-blocking Receive

- C:

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype
              int src, int tag, MPI_Comm comm,
              MPI_Request *request);
```

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 or standard
- Synchronous mode affects completion, not initiation

Communication Modes

Non-Blocking Operation	MPI Call
Standard send	`MPI_Isend`
Synchronous send	`MPI_Issend`
Buffered send	`MPI_Ibsend`
Receive	`MPI_Irecv`

Completion

- C:

```
int MPI_Wait(MPI_Request *request,
             MPI_Status *status);

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

Example (C)

```
MPI_Request request;
MPI_Status status;

if (rank == 0)  {
  MPI_Issend(sendarray, 10, MPI_INT, 1, tag,
             MPI_COMM_WORLD, &request);
  Do_something_else_while_Issend_happens();
  // now wait for send to complete
  MPI_Wait(&request, &status);
} else if (rank == 1) {
  MPI_Irecv(recvarray, 10, MPI_INT, 0, tag,
            MPI_COMM_WORLD, &request);
  Do_something_else_without_data_from_recvarray
    _while_Irecv_happens();
  // now wait for receive to complete;
  MPI_Wait(&request, &status);
  Do_something_with_the_data_in_recvarray();
}
```

Multiple Communications

- Test or wait for completion of one message
- Test or wait for completion of all messages
- Test or wait for completion of as many messages as possible