Parallel Computing for Science & Engineering Spring 2013: MPI point-to-point 2

Instructors:

Victor Eijkhout, Research Scientist, TACC

Kent Milfeld, Research Associate, TACC

Wildcards

- Enables programmer to avoid having to specify a tag and/or source.
- Example:

```
MPI_Status status;
int        data[5];
int        ierr;
ierr = MPI_Recv(&data[0], 5, MPI_INT,
                MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD,&status);
```

- **MPI_ANY_SOURCE** and **MPI_ANY_TAG** are wild cards
- **status** structure is used to get wildcard values

Wildcards

- Enables programmer to avoid having to specify a tag and/or source.
- Example:

```
integer :: status;
integer :: data(5);
integer :: ierr;
call   MPI_Recv( data,    5, MPI_INTEGER,
                 MPI_ANY_SOURCE, MPI_ANY_TAG,
                 MPI_COMM_WORLD, status, ierr);
```

- **MPI_ANY_SOURCE** and **MPI_ANY_TAG** are wild cards
- **status** structure is used to get wildcard values

Wildcards

- **`MPI_PROC_NULL`**
  - can be used for destination or source in send or receive calls
  - operation completes immediately
  - no communications involved
- Great for handling edges of partitioned data
- Useful with **`MPI_Sendrecv`**

More on Status

- C
  - **status** (type `MPI_Status`) is a structure which contains three fields **`MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`**
  - `status.MPI_SOURCE`, `status.MPI_TAG`, and `status.MPI_ERROR` contain the source, tag, and error code respectively of the received message
- Fortran
  - **status** is an array of `INTEGER`s of length `MPI_STATUS_SIZE`, and the 3 constants **`MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`** are the indices of the entries that store the source, tag, & error
  - `status(MPI_SOURCE)`, `status(MPI_TAG)`, `status(MPI_ERROR)` contain respectively the source, the tag, and the error code of the received message.

Order Semantics

- Messages with the same tag are ordered, for the rest: make no assumptions on message ordering!
  - the first receive always matches the first send in the following

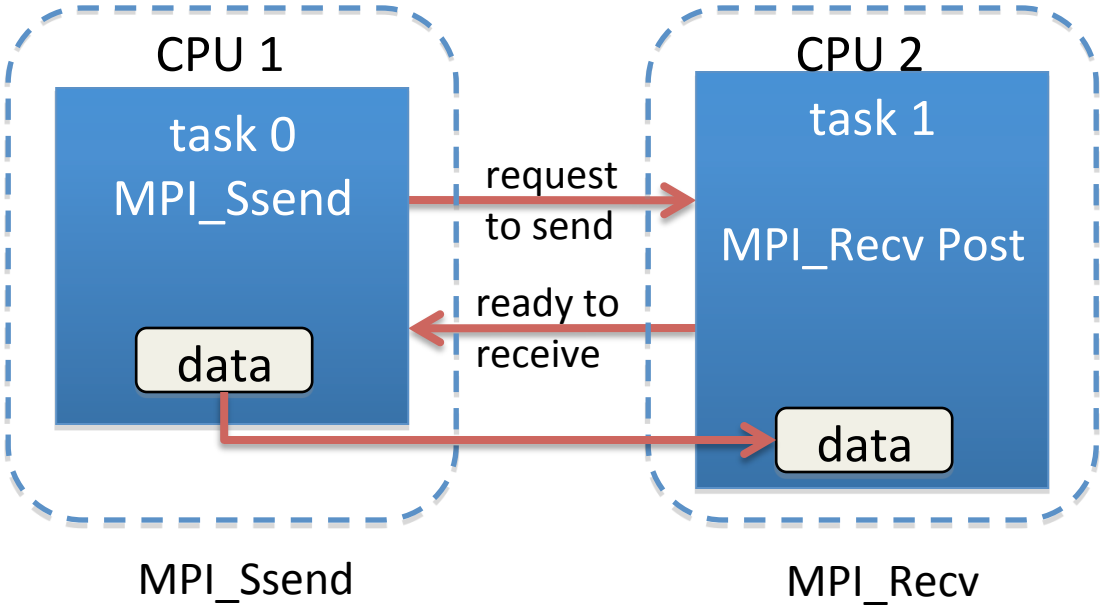
```
tag=123456
if (rank.EQ.0) then
  call MPI_BSend(b1,cnt,MPI_REAL,1,tag,comm,err)
  call MPI_BSend(b2,cnt,MPI_REAL,1,tag,comm,err)
ELSE ! rank.EQ.1
  call MPI_Recv(b1,cnt,MPI_REAL,0,tag,comm,
                status,ierr)
  call MPI_Recv(b2,cnt,MPI_REAL,0,tag,comm,
                status, ierr)
END if
```

MPI Receive Modes

IMPORTANT: From the MPI-2 Standard:

There is only one receive operation, but it matches any of the send modes. The receive operation described in the last section is blocking: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

Synchronous Communication



- Data isn't sent until Receive has been posted.
- Synchronous send returns when data area is safe for re-use.
- There is no MPI_Srecv

Synchronous Communication

- **Ssend C**

```
     ...
i=1;
if(irank == 0){
        MPI_Ssend(&i, 1, MPI_INT,     1, 9, MPI_COMM_WORLD);
}else {
        MPI_Recv( &j, 1, MPI_INT,     0, 9, MPI_COMM_WORLD, &status);
}
```

- **Ssend F90**

```
 ...
i=1
if(irank == 0) then
   call MPI_Ssend( i, 1, MPI_INTEGER, 1, 9, MPI_COMM_WORLD, ierr)
else
   call MPI_Recv(  j, 1, MPI_INTEGER, 0, 9, MPI_COMM_WORLD, status,ierr)
endif
```

MPI_Test

- Value of flags signifies whether a message has been delivered
- Similar to **`MPI_Wait`**, but does not block
- C

```
int flag;
ierr= MPI_Test(&request, &flag, &status);
```

- Fortran

```
logical flag
call MPI_Test(  request,  flag,  status, ierr)
```

MPI_Cancel

- Cancel a pending non-blocking send or receive
- C

```
MPI_Request request;
ierr= MPI_Cancel(&request);
```

- Fortran

```
integer request
call MPI_Cancel(  request, ierr)
```

MPI_Probe

- `MPI_Probe` allows incoming messages to be checked without actually receiving them
  - the user can then decide how to receive the data
  - Used when different actions need to be taken, depending on the "who, what, and how much" information of the message.

MPI_Probe

- C

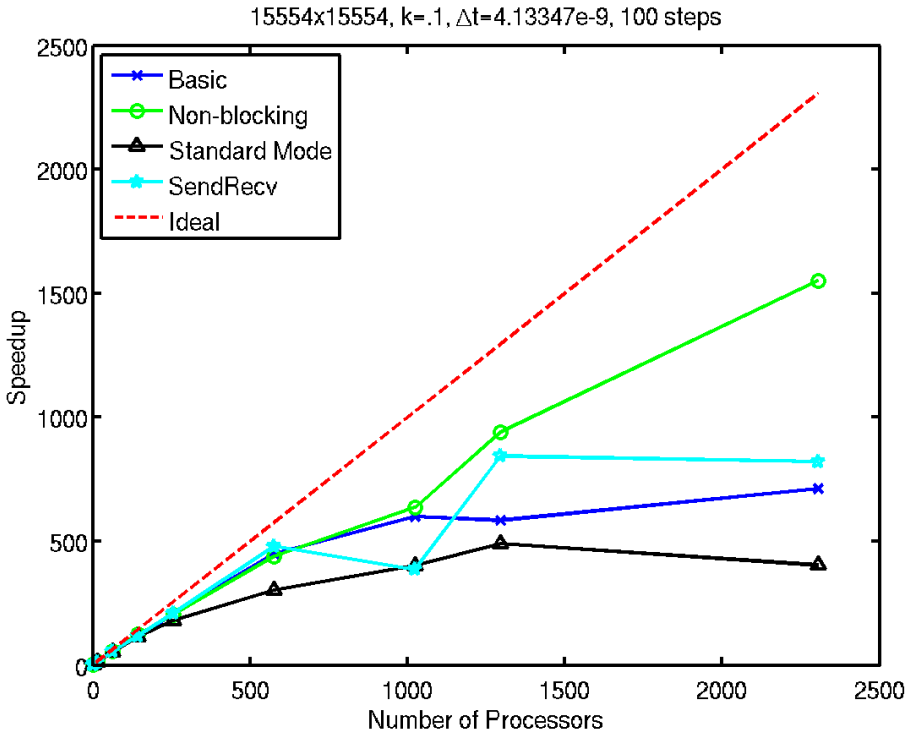
```
ierr=MPI_Probe(source, tag, comm, &status);
```

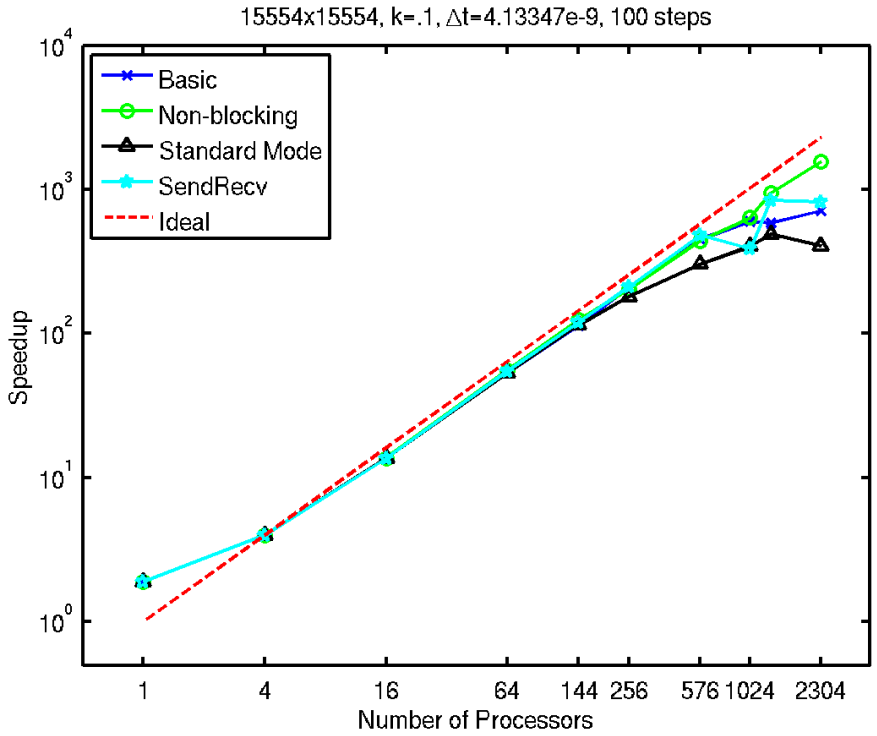
- Fortran

```
MPI_Probe(source, tag, comm,  status, ierr)
```

- Parameters
  - source: source rank or **MPI_ANY_SOURCE**
  - tag: tag value or **MPI_ANY_TAG**
  - comm: communicator
  - status: status object

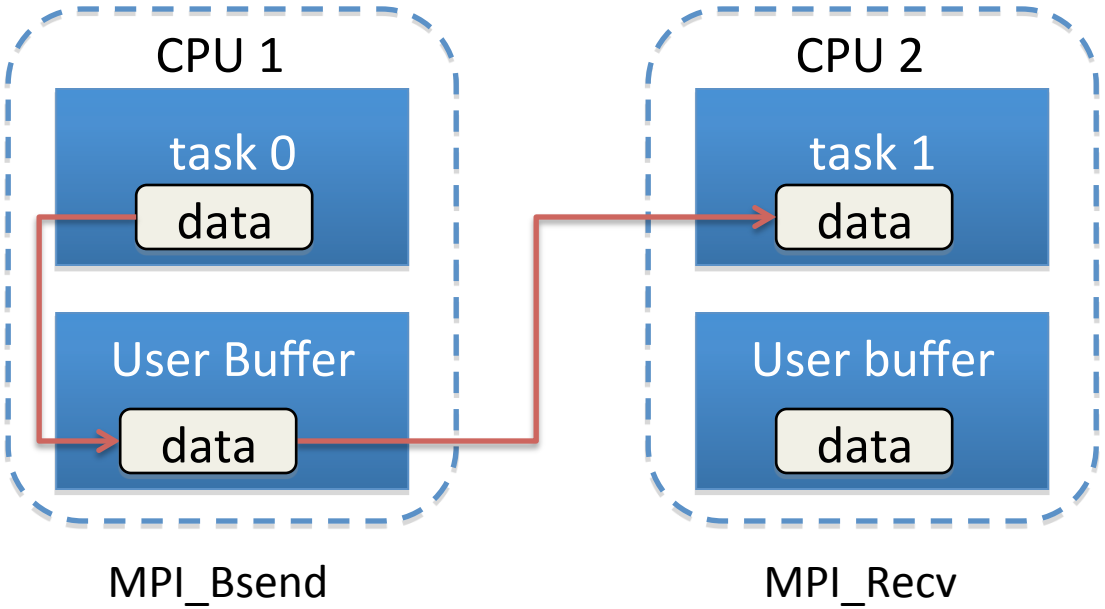
Scaling


15554x15554, k=.1, Δt=4.13347e-9, 100 steps
Basic
Non-blocking
Standard Mode
SendRecv
Ideal
Speedup
Number of Processors
0
500
1000
1500
2000
2500


15554x15554, k=.1, Δt=4.13347e-9, 100 steps
Basic
Non-blocking
Standard Mode
SendRecv
Ideal
Speedup
Number of Processors
1
4
16
64
144
256
576
1024
2304

Obscure stuff

Buffered Communication



- The contents of the message is copied into a system-controlled block of memory (User Buffer).
- MPI_Bsend returns when copy to User buffer is complete.
- There is no MPI_Brecv.
- Use MPI_BSend_OVERHEAD to provide room for message headers
- Fails if there isn't enough space for buffering
- Buffer area must contain (MPI_BSEND_OVERHEAD) room for each message.

User-Buffer Communication

- **BSend**

```
      ...
      character,allocatable,dimension(:) :: cbuffer

      call MPI_Init(ierr)
      call MPI_Comm_rank(MPI_COMM_WORLD, irank, ierr)

      i=1
      isize_bytes = sizeof(i) + MPI_BSend_OVERHEAD
      allocate( cbuffer(isize_bytes) )
      call MPI_Buffer_attach (cbuffer , isize_bytes , ierr )

      if(irank == 0) then
         call MPI_Bsend(i, 1, MPI_INTEGER, 1, 9, MPI_COMM_WORLD, ierr)
      else
         call MPI_Recv( j, 1, MPI_INTEGER, 0, 9, MPI_COMM_WORLD, status,
ierr)
      endif
```

User-Buffer Communication

- **BSend**

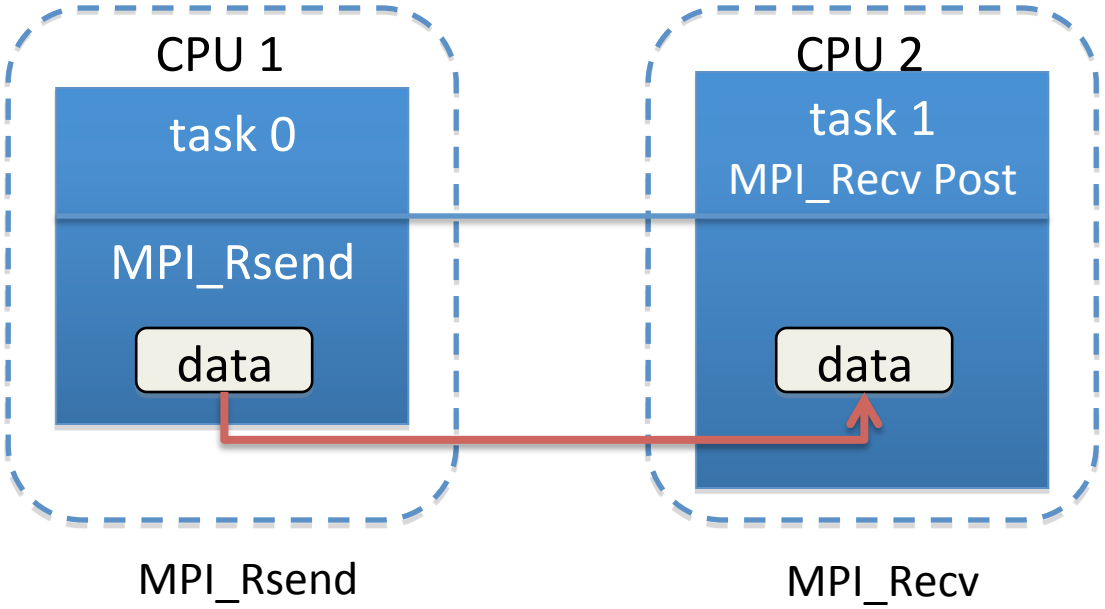
```
...
char* cbuffer;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &irank);

i = 1;
isize_bytes = sizeof(i) + MPI_BSend_OVERHEAD ;
cbuffer = malloc((size_t)isize_bytes );
MPI_Buffer_attach(cbuffer , isize_bytes );

if(irank == 0) {
   MPI_Bsend(&i, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
} else {
   MPI_Recv( &j, 1, MPI_INT, 0, 9, MPI_COMM_WORLD,
&status);
}
```

Ready Communication



- Receive is guaranteed to be posted.
- Ready returns when data area is safe for re-use.
- Not often used. Behavior is not defined if receive has not been posted first.
- There is no MPI_Rrecv. You might find it in some MPI implementations but it is NOT part of the MPI-2 standard

Ready Communication

- **Rsend F90**

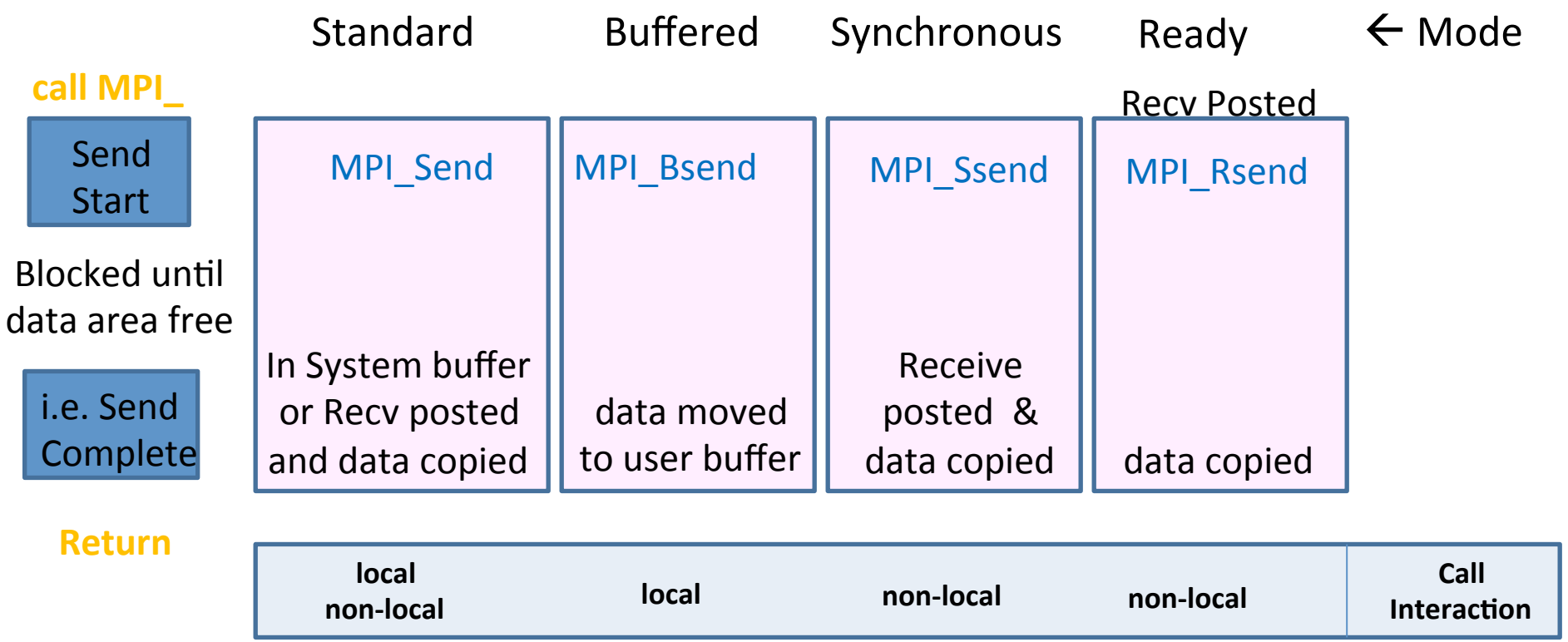
```
     ...
i=1;
if(irank == 0)then
   call MPI_Barrier(MPI_COMM_WORLD, ierr);
   call MPI_Rsend(i, 1, MPI_INTEGER, 1, 9, MPI_COMM_WORLD, ierr);
else
   call MPI_Irecv(…);
   call MPI_Barrier(MPI_COMM_WORLD, ierr);
     ...
endif
```

- **Rsend C**

```
 ...
i=1;
if(irank == 0){
   MPI_Barrier(MPI_COMM_WORLD);
   MPI_Rsend(&i, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
}else {
   MPI_Irecv(…);
   MPI_Barrier(MPI_COMM_WORLD);
...}
```

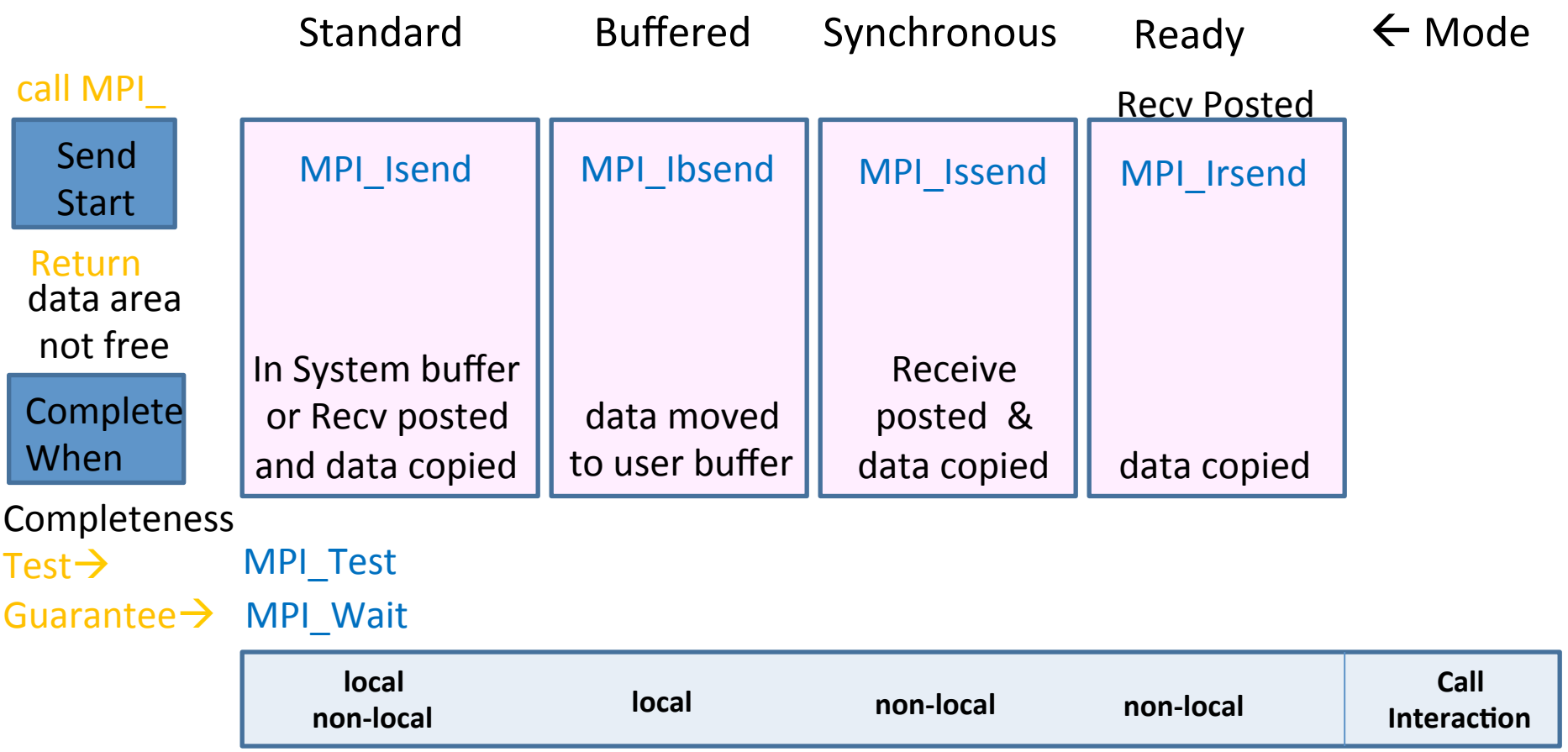
More about this later.
This allows Recv to be posted without blocking--
the call returns immediately.

Blocking Pt-2-Pt communications



MPI_Recv is used with MPI_Send, MPI_Bsend & MPI_Ssend and MPI_Rsend.

Non-Blocking Pt-2-Pt communications



MPI_Irecv or MPI_Recv is used with MPI_Isend, MPI_Ibsend, MPI_Issend, MPI_Irsend & blocking versions.