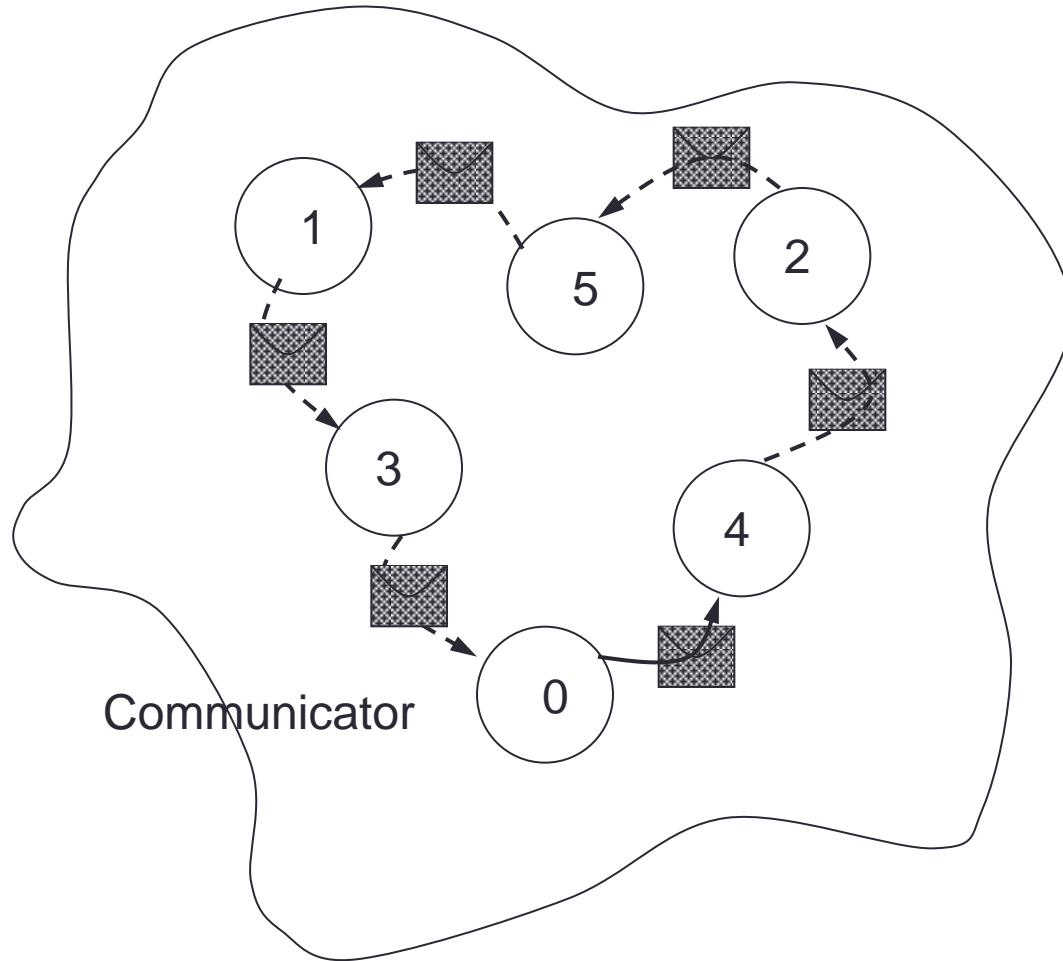


Non-Blocking Communications

Deadlock



Completion

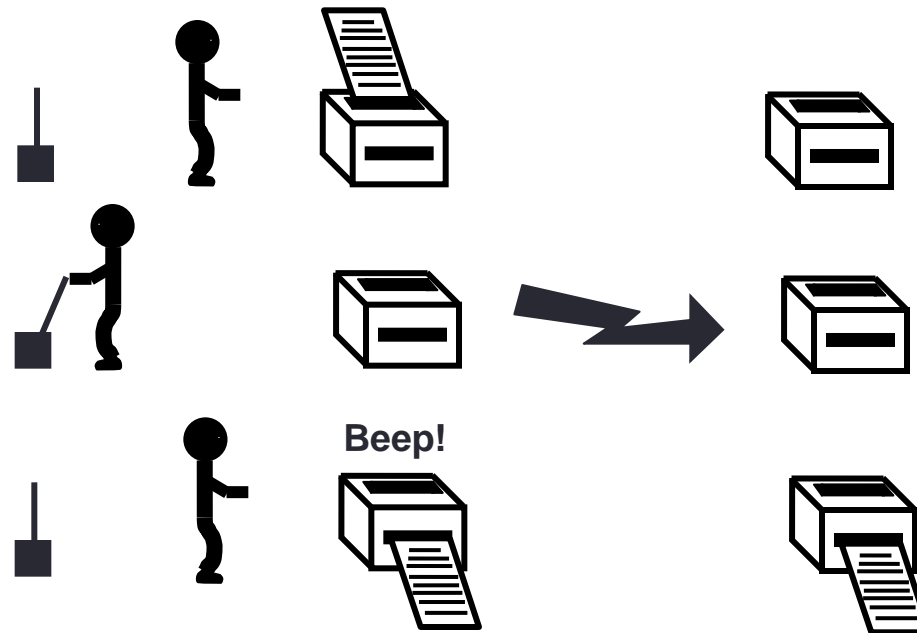
- The *mode* of a communication determines when its constituent operations complete.
 - i.e. synchronous / asynchronous
- The *form* of an operation determines when the procedure implementing that operation will return
 - i.e. when control is returned to the user program

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_Recv`

Non-Blocking Operations

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



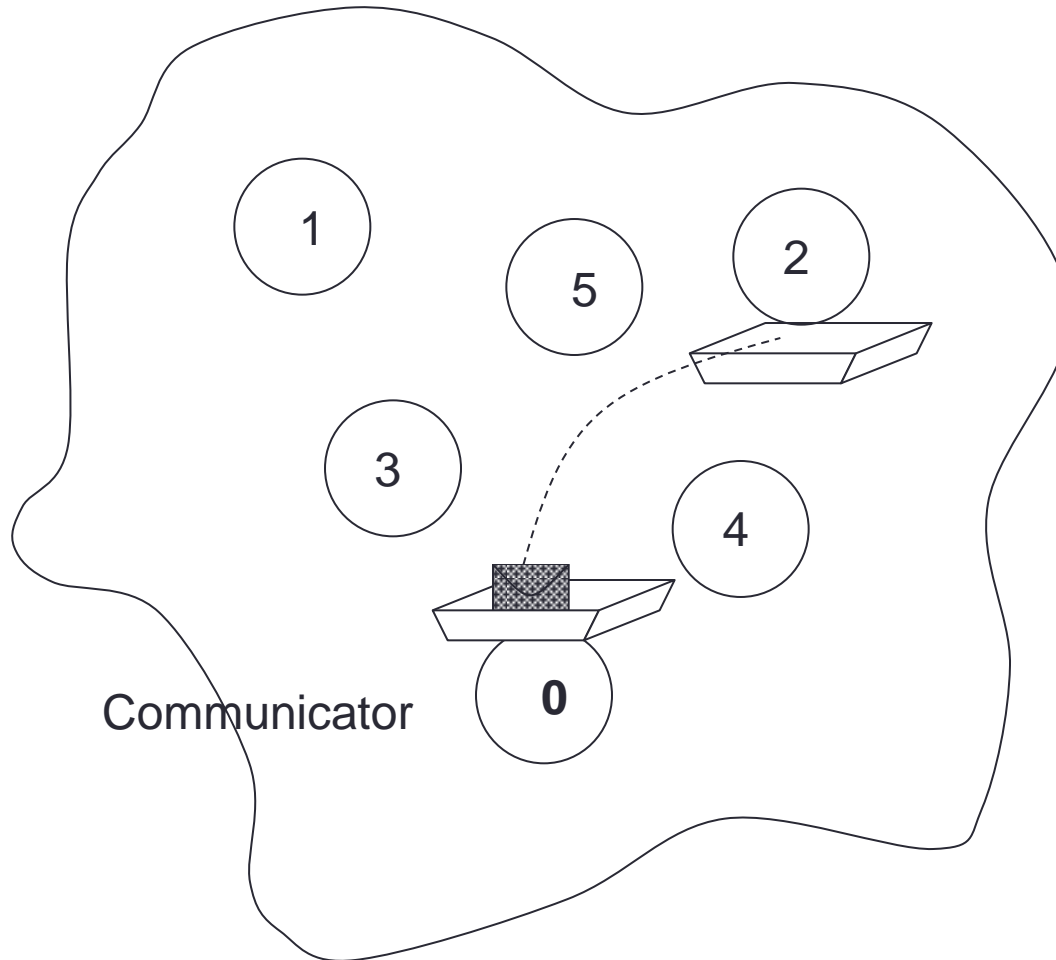
Non-Blocking Operations

- 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.

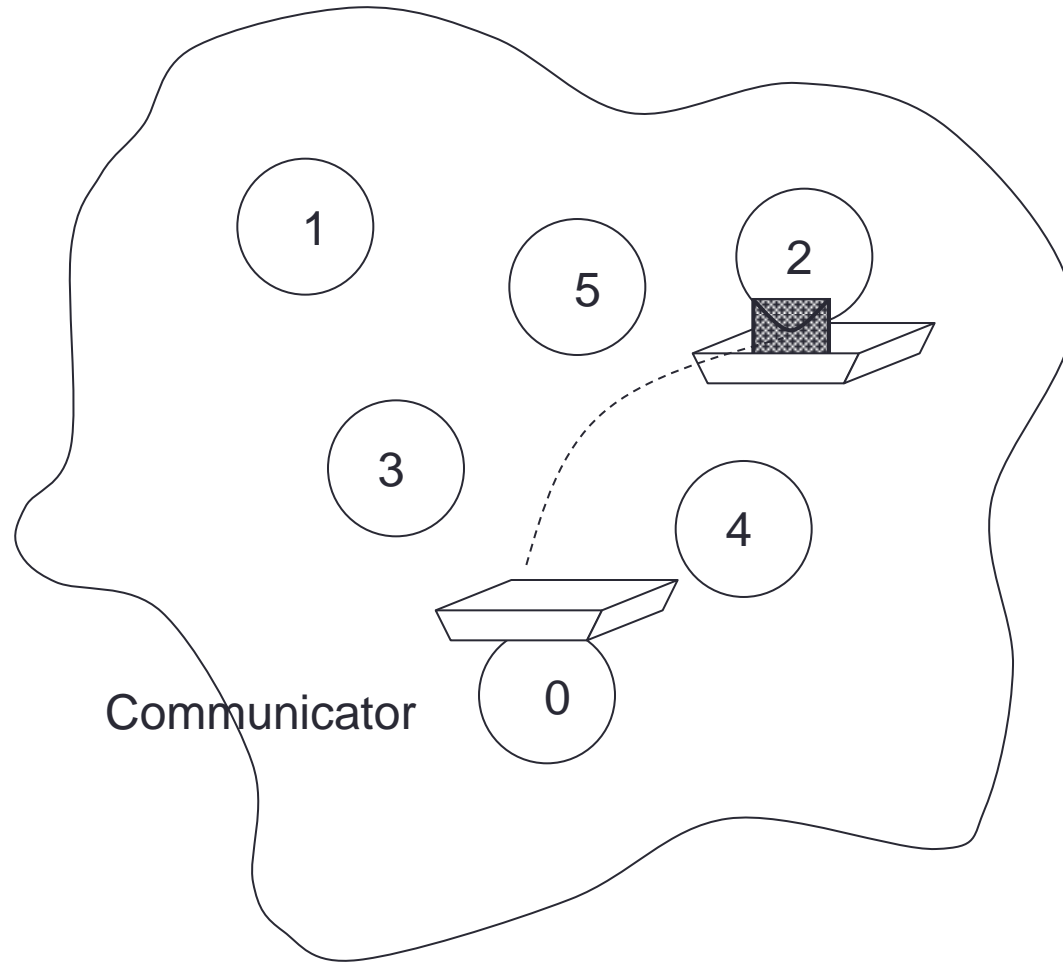
Non-Blocking Communications

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

Non-Blocking Send



Non-Blocking Receive



Handles used for Non-blocking Comms

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

Non-blocking Synchronous Send

- C:

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

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

- Fortran:

```
MPI_ISSEND(buf, count, datatype, dest,  
          tag, comm, request, ierror)
```

```
MPI_WAIT(request, status, ierror)
```

Non-blocking Receive

- C:

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

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

- Fortran:

```
MPI_IRecv(buf, count, datatype, src,  
         tag, comm, request, ierror)
```

```
MPI_WAIT(request, status, ierror)
```

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.
- 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
Ready send	MPI_IRSEND
Receive	MPI_Irecv

Completion

- Waiting versus Testing.
- C:

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

- Fortran:

```
MPI_WAIT(handle, status, ierror)  
  
MPI_TEST(handle, flag, status, ierror)
```

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_while_Irecv_happens();
    // now wait for receive to complete;
    MPI_Wait(&request, &status);
}
```


Example (Fortran)

```
integer request
integer, dimension(MPI_STATUS_SIZE) :: status

if (rank == 0) then

    CALL MPI_ISSEND(sendarray, 10, MPI_INTEGER, 1, tag,
                   MPI_COMM_WORLD, request, ierror)
    CALL Do_something_else_while Issend_happens()
    ! now wait for send to complete
    CALL MPI_Wait(request, status, ierror)

else if (rank == 1) then

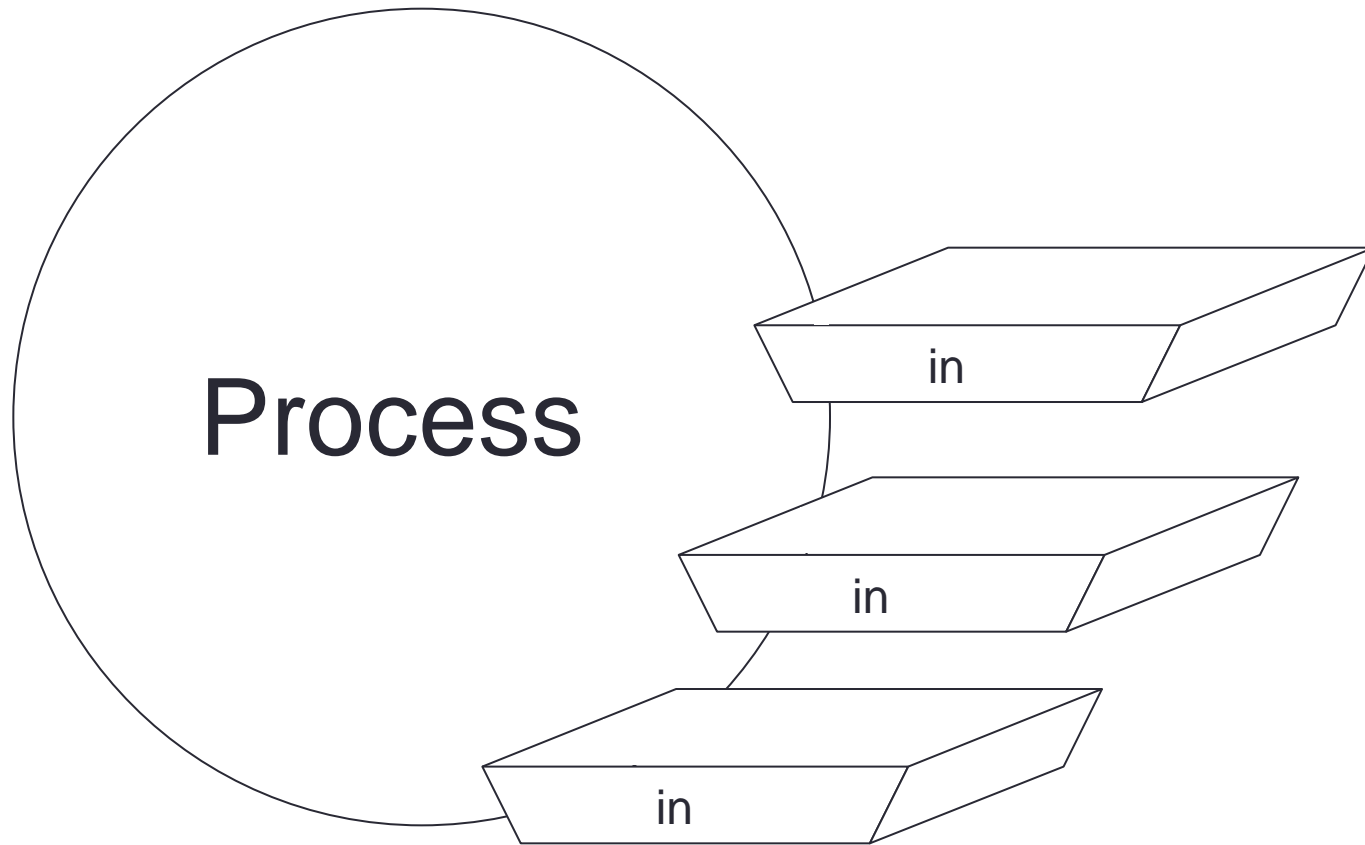
    CALL MPI_IRecv(recvarray, 10, MPI_INTEGER, 0, tag,
                  MPI_COMM_WORLD, request, ierror)
    CALL Do_something_else_while Irecv_happens()
    ! now wait for receive to complete
    CALL MPI_Wait(request, status, ierror)

endif
```

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.

Testing Multiple Non-Blocking Comms



Combined Send and Receive

- Specify all send / receive arguments in one call
 - MPI implementation avoids deadlock
 - useful in simple pairwise communications patterns, but not as generally applicable as non-blocking

```
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);
```

```
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag,  
             recvbuf, recvcount, recvtype, source, recvtag,  
             comm, status, ierror)
```

Exercise

Rotating information around a ring

- See Exercise 4 on the sheet
- Arrange processes to communicate round a ring.
- Each process stores a copy of its rank in an integer variable.
- Each process communicates this value to its right neighbour, and receives a value from its left neighbour.
- Each process computes the sum of all the values received.
- Repeat for the number of processes involved and print out the sum stored at each process.

Possible solutions

- Non-blocking send to forward neighbour
 - blocking receive from backward neighbour
 - wait for forward send to complete
- Non-blocking receive from backward neighbour
 - blocking send to forward neighbour
 - wait for backward receive to complete
- Non-blocking send to forward neighbour
- Non-blocking receive from backward neighbour
 - wait for forward send to complete
 - wait for backward receive to complete

Notes

- Your neighbours *do not change*
 - send to left, receive from right, send to left, receive from right, ...
- You *do not alter* the data you receive
 - receive it
 - add it to you running total
 - pass the data *unchanged* along the ring
- You *must not access* send or receive buffers until communications are complete
 - cannot read from a receive buffer until after a wait on **irecv**
 - cannot overwrite a send buffer until after a wait on **isend**