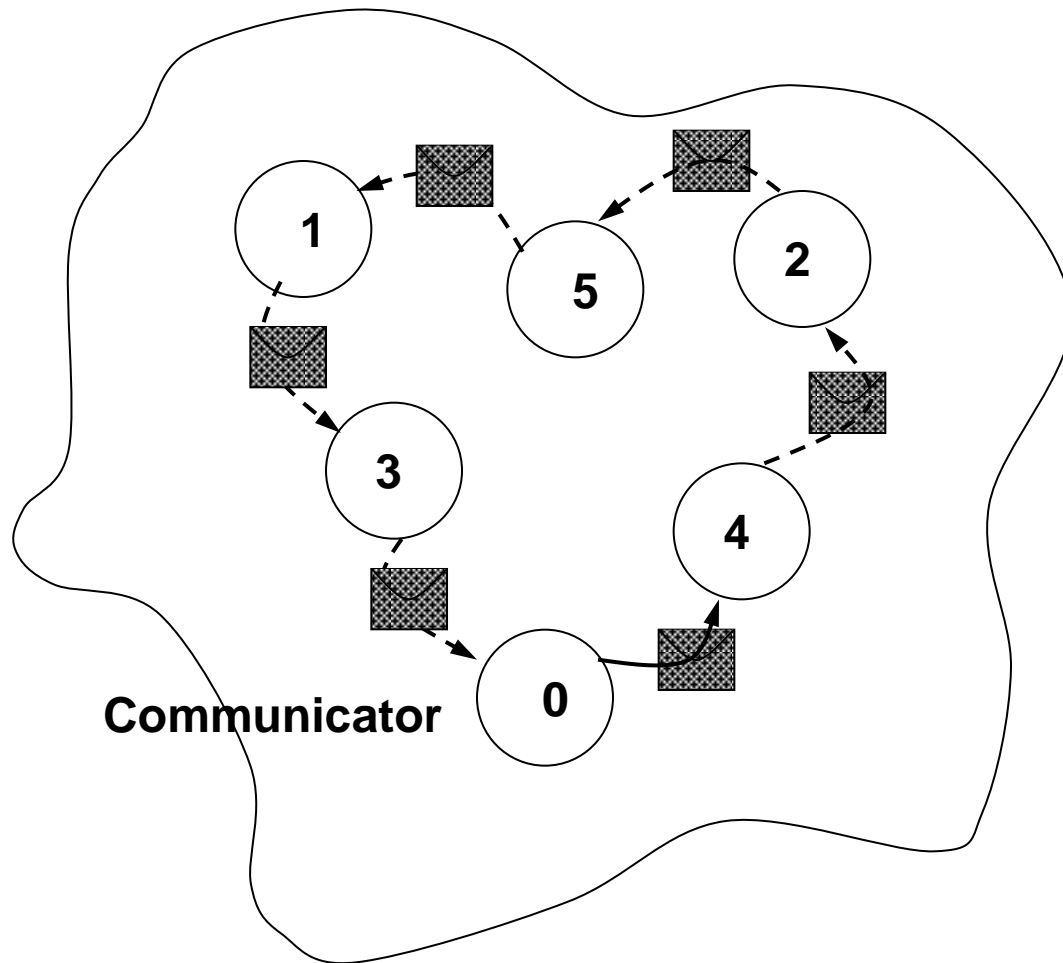


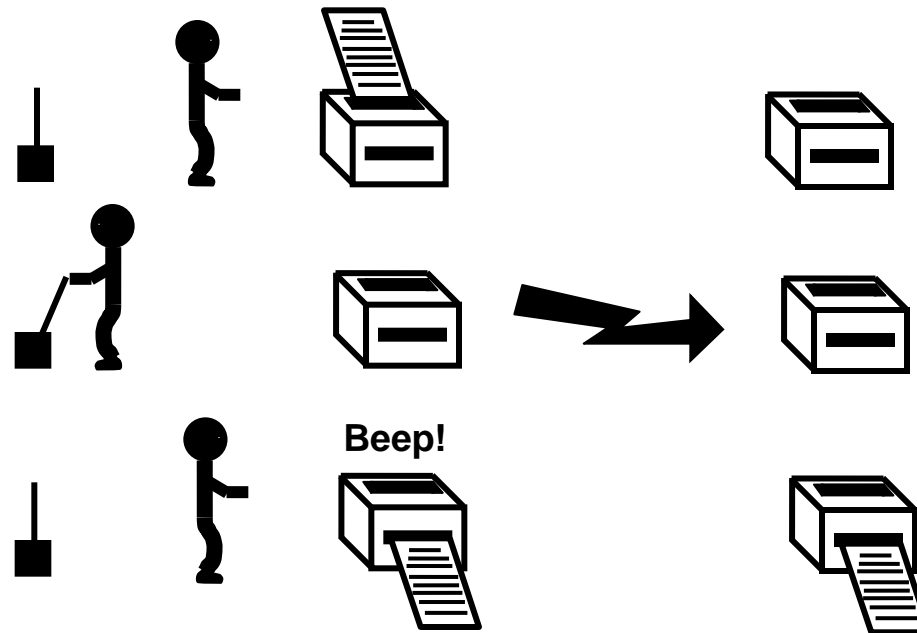
Non-Blocking Communications



- ▶ 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

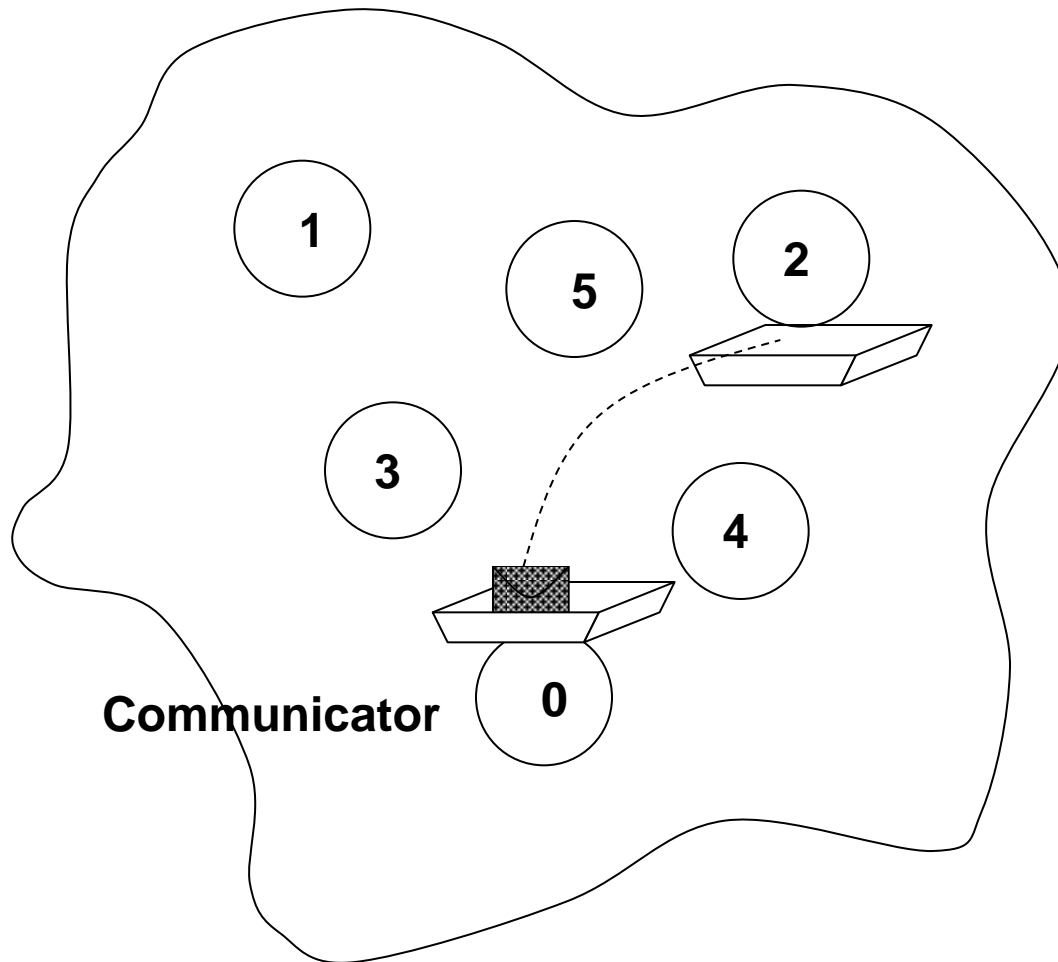
- ▶ Relate to when the operation has completed.
- ▶ Only return from the subroutine call when the operation has completed.
- ▶ These are the routines you used thus far
 - MPI_Ssend
 - MPI_Recv

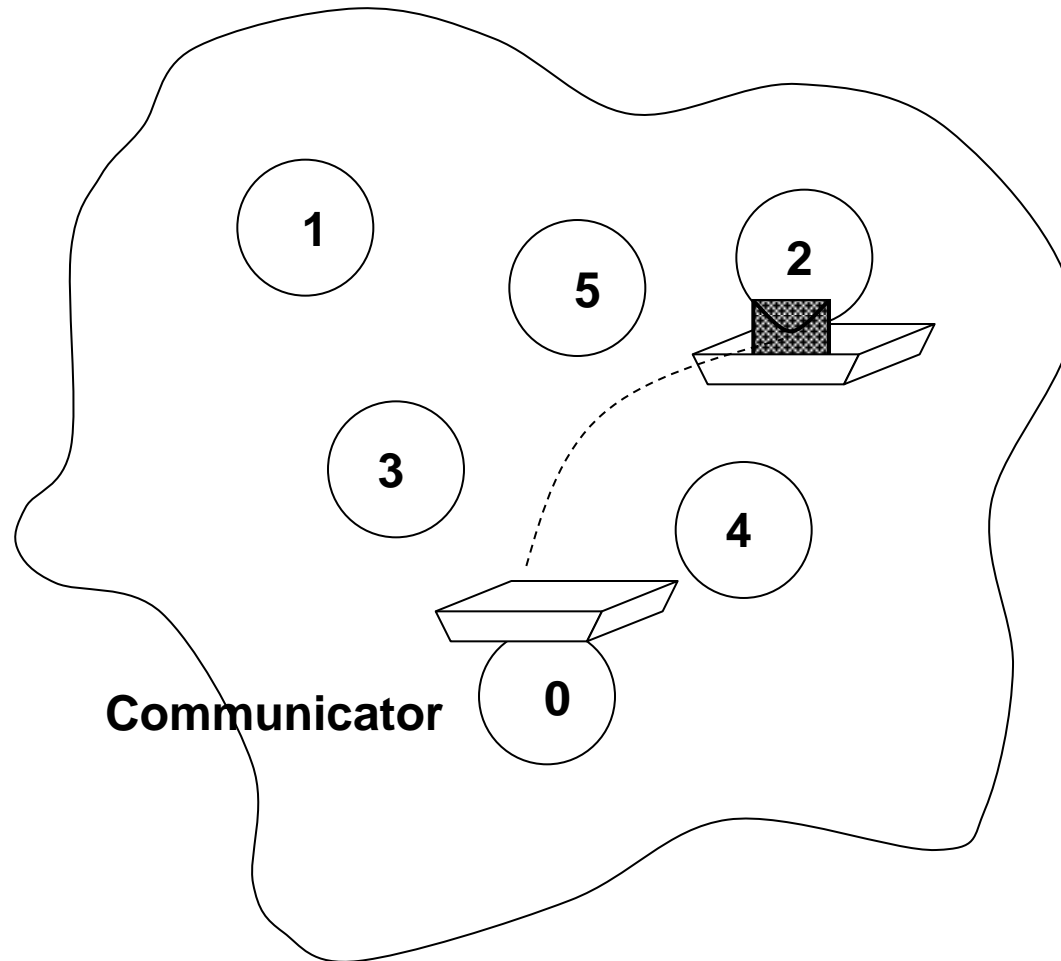
- Return straight away and allow the sub-program to continue to perform other work. At some later time the sub-program can *test* or *wait* for the completion of the non-blocking operation.



- ▶ 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.

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





- ▶ `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.

▶ 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)
```

▶ 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)
```

- ▶ 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.

NON-BLOCKING OPERATION	MPI CALL
Standard send	MPI_ISEND
Synchronous send	MPI_ISSEND
Buffered send	MPI_IBSEND
Ready send	MPI_IRSEND
Receive	MPI_IRECV

▶ Waiting versus Testing.

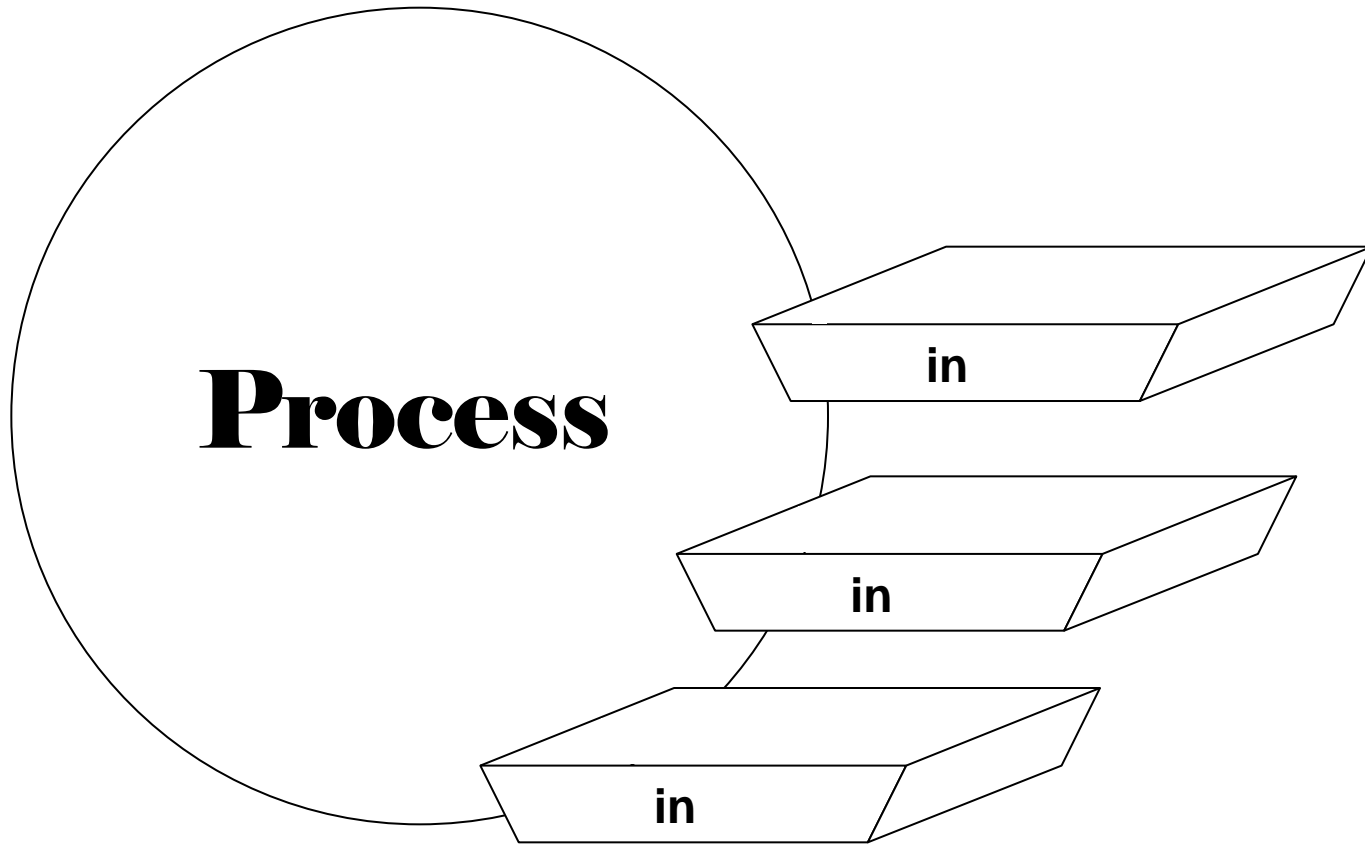
▶ C:

```
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)
```

- ▶ 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.



- ▶ 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)
```

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.

- ▶ 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

- ▶ 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 your 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 issend