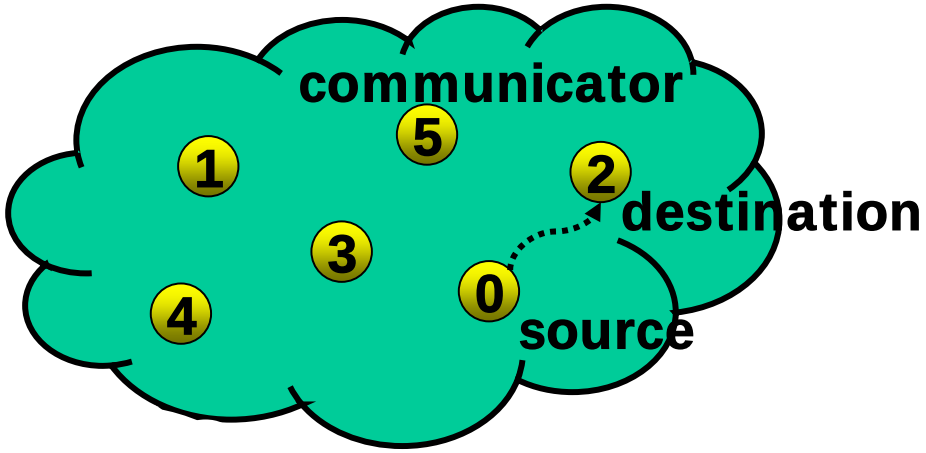
Point-to-Point Communications

- Definitions
- Communication modes
- Routine names (blocking)
- Sending a message
- Memory mapping
- Synchronous send
- Buffered send
- Standard send
- Ready send
- Receiving a message
- Wildcarding
- Communication envelope
- Received message count
- Message order preservation
- Sample program
- Timers
- Exercise: Processor Ring
- Extra Exercise 1: Ping Pong
- Extra Exercise 2: Broadcast

Ohio Supercomputer Center

Point-to-Point Communication



- Communication between two processes
- Source process *sends* message to destination process
- Destination process *receives* the message
- Communication takes place within a communicator
- Destination process is identified by its rank in the communicator

Definitions

- "Completion" of the communication means that memory locations used in the message transfer can be safely accessed
  - Send: variable sent can be reused after completion
  - Receive: variable received can now be used

- MPI communication modes differ in what conditions are needed for completion

- Communication modes can be **blocking** or **non-blocking**
  - Blocking: return from routine implies completion
  - Non-blocking: routine returns immediately, user must test for completion

Communication modes

Mode	Completion Condition
Synchronous send	Only completes when the receive has initiated
Buffered send	Always completes (unless and error occurs), irrespective of receiver
Standard send	Message sent (receive state unknown)
Ready send	Always completes (unless and error occurs), irrespective of whether the receive has completed
Receive	Completes when a message has arrived

Routine Names (blocking)

MODE	MPI CALL
Standard send	MPI_SEND
Synchronous send	MPI_SSEND
Buffered send	MPI_BSEND
Ready send	MPI_RSEND
Receive	MPI_RECV

Sending a message

C:

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

Fortran:

```
CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

<type>   BUF(*)
INTEGER  COUNT, DATATYPE, DEST, TAG
INTEGER  COMM, IERROR
```

Arguments

`buf`	starting address of the data to be sent
`count`	number of elements to be sent
`datatype`	MPI datatype of each element
`dest`	rank of destination process
`tag`	user flag to classify messages
`comm`	MPI communicator of processors involved

```
MPI_SEND(data,500,MPI_REAL,6,33,MPI_COMM_WORLD,IERROR
                                )
```

Memory mapping

The 2-D Fortran array

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

Is stored in memory as:

("column-major")

1,1
2,1
3,1
1,2
2,2
3,2
1,3
2,3
3,3

Synchronous send (`MPI_Ssend`)

- Completion criteria: receiving process sends an acknowledgement ("handshake"), which must be received by sender before the send is considered complete
- Use if need to know that message has been received
- Sending and receiving processes synchronize
  - Regardless of who is faster
  - Processor idle time is probable
- Safest communication method

Ohio Supercomputer Center

Buffered send (`MPI_Bsend`)

- Completion criteria: Completes when message copied to buffer
- Advantage: Guaranteed to complete immediately (predictability)
- Disadvantage: User cannot assume there is a pre-allocated buffer and must explicitly attach it
- Control your own buffer space using MPI routines

```
MPI_Buffer_attach
MPI_Buffer_detach
```

Standard send (`MPI_Send`)

- Completion criteria: Unknown!
- Simply completes when the message has been sent
- May or may not imply that message has arrived at destination
- Don’t make any assumptions (implementation dependent)

Ready send (`MPI_Rsend`)

- Completion criteria: Completes immediately, but successful only if matching receive already posted
- Advantage: Completes immediately
- Disadvantage: User must synchronize processors so that receiver is ready
- Potential for good performance

Receiving a message

C:

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

Fortran:

```
CALL MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
     STATUS, IERROR)

<type>   BUF(*)
INTEGER  COUNT, DATATYPE, DEST, TAG
INTEGER  COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

For a communication to succeed…

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Receiver's buffer must be large enough

Wildcarding

- Receiver can wildcard
- To receive from any source

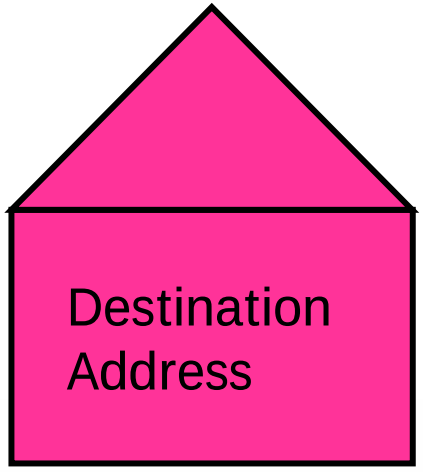
  `MPI_ANY_SOURCE`

  To receive with any tag

  `MPI_ANY_TAG`

- Actual source and tag are returned in the receiver's `status` parameter

Communication envelope



envelope ***routes*** the data

Sender's Address

For the attention of:

Data
Item 1
Item 2
Item 3

Communication envelope information

- Envelope information is returned from `MPI_RECV` as `status`

- Information includes:
  - Source: `status.MPI_SOURCE` or `status(MPI_SOURCE)`
  - Tag:`status.MPI_TAG` or `status(MPI_TAG)`
  - Count: `MPI_Get_count` or `MPI_GET_COUNT`

Received message count

- Message received may not fill receive buffer
- `count` is number of elements actually received

C:

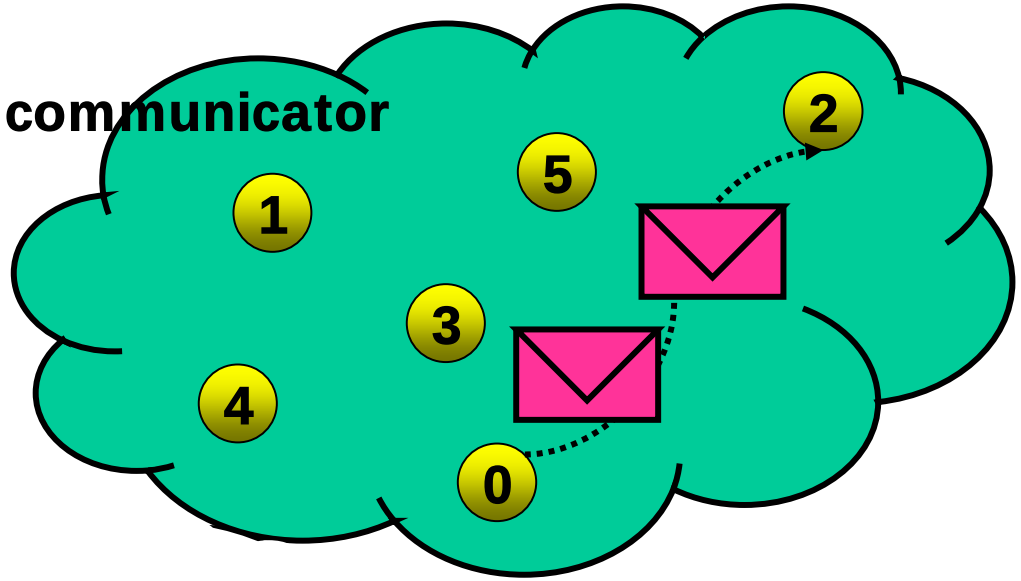
```
int MPI_Get_count (MPI_Status *status,
    MPI_Datatype datatype, int *count)
```

Fortran:

```
CALL
MPI_GET_COUNT(STATUS,DATATYPE,COUNT,IERROR)

INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE
INTEGER COUNT,IERROR
```

Message order preservation



- Messages do no overtake each other
- Example: Process 0 sends two messages
  Process 2 posts two receives that match either message: Order preserved

Sample Program #1 - C

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

/* Run with two processes */

 int main(int argc, char *argv[]) {
    int rank, i, count;
    float data[100],value[200];
    MPI_Status status;

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

    if(rank==1) {
       for(i=0;i<100;++i) data[i]=i;
       MPI_Send(data,100,MPI_FLOAT,0,55,MPI_COMM_WORLD);
    } else {

   MPI_Recv(value,200,MPI_FLOAT,MPI_ANY_SOURCE,55,MPI_COMM_WORLD,&status);
       printf("P:%d Got data from processor %d \n",rank, status.MPI_SOURCE);
       MPI_Get_count(&status,MPI_FLOAT,&count);
       printf("P:%d Got %d elements \n",rank,count);
       printf("P:%d value[5]=%f \n",rank,value[5]);
    }
    MPI_Finalize();
 }
```

```
          Program Output
P: 0 Got data from processor 1
P: 0 Got 100 elements
P: 0 value[5]=5.000000
```

Sample Program #1 - Fortran

```
      PROGRAM p2p
C Run with two processes
      INCLUDE 'mpif.h'
      INTEGER err, rank, size
      real data(100)
      real value(200)
      integer status(MPI_STATUS_SIZE)
      integer count
      CALL MPI_INIT(err)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
      if (rank.eq.1) then
         data=3.0
         call MPI_SEND(data,100,MPI_REAL,0,55,MPI_COMM_WORLD,err)
       else
         call MPI_RECV(value,200,MPI_REAL,MPI_ANY_SOURCE,55,
     &                 MPI_COMM_WORLD,status,err)
         print *, "P:",rank," got data from processor ",
     &                 status(MPI_SOURCE)
         call MPI_GET_COUNT(status,MPI_REAL,count,err)
         print *, "P:",rank," got ",count," elements"
         print *, "P:",rank," value(5)=",value(5)
      end if
      CALL MPI_FINALIZE(err)
      END
```

```
Program Output
P: 0 Got data from processor 1
P: 0 Got 100 elements
P: 0 value[5]=3.
```

Timers

- Time is measured in seconds
- Time to perform a task is measured by consulting the timer before and after

C:

```
double MPI_Wtime(void);
```

Fortran:

```
DOUBLE PRECISION MPI_WTIME()
```

Class Exercise: Processor Ring

- A set of processes is arranged in a ring
- Each process stores its rank in `MPI_COMM_WORLD` in an integer
- Each process passes this on to its neighbor on the right
- Each processor keeps passing until it receives its rank back

Extra Exercise 1: Ping Pong

- Write a program in which two processes repeatedly pass a message back and forth
- Insert timing calls to measure the time taken for one message
- Investigate how the time taken varies with the size of the message

Extra Exercise 2: Broadcast

- Have processor 1 send the same message to all the other processors and then receive messages of the same length from all the other processors
- How does the time taken vary with the size of the messages and the number of processors?