# Messages

# Reusing this material

# Messages

- A message contains a number of elements of some particular datatype.
- MPI datatypes:
  - Basic types.
  - Derived types.
- Derived types can be built up from basic types.
- C types are different from Fortran types.

# MPI Basic Datatypes - C

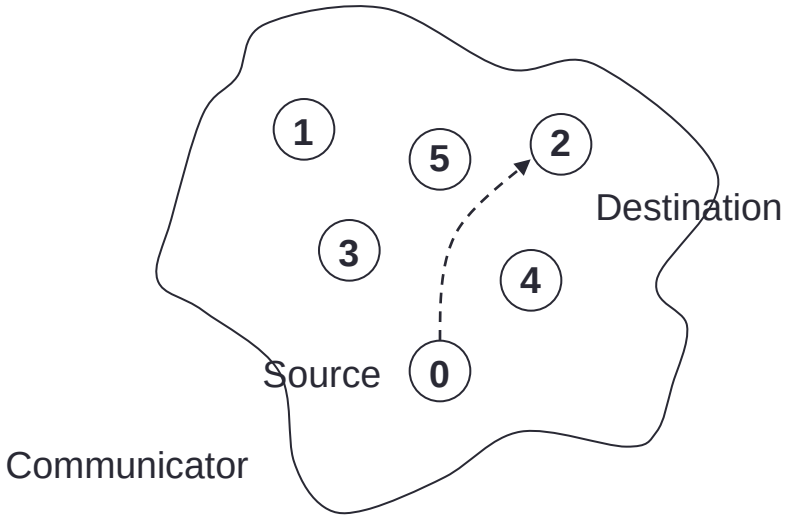
| MPI Datatype | C datatype |
|---|---|
| **MPI_CHAR** | signed char |
| **MPI_SHORT** | signed short int |
| **MPI_INT** | signed int |
| **MPI_LONG** | signed long int |
| **MPI_UNSIGNED_CHAR** | unsigned char |
| **MPI_UNSIGNED_SHORT** | unsigned short int |
| **MPI_UNSIGNED** | unsigned int |
| **MPI_UNSIGNED_LONG** | unsigned long int |
| **MPI_FLOAT** | float |
| **MPI_DOUBLE** | double |
| **MPI_LONG_DOUBLE** | long double |
| **MPI_BYTE** | |

# MPI Basic Datatypes - Fortran

| MPI Datatype | Fortran Datatype |
|---|---|
| **MPI_INTEGER** | INTEGER |
| **MPI_REAL** | REAL |
| **MPI_DOUBLE_PRECISION** | DOUBLE PRECISION |
| **MPI_COMPLEX** | COMPLEX |
| **MPI_LOGICAL** | LOGICAL |
| **MPI_CHARACTER** | CHARACTER(1) |
| **MPI_BYTE** | |

# Point-to-Point Communication

# Point-to-Point Communication




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

# Point-to-point messaging in MPI

- Sender calls a SEND routine
  - specifying the data that is to be sent
  - this is called the *send buffer*
- Receiver calls a RECEIVE routine
  - specifying where the incoming data should be stored
  - this is called the *receive buffer*
- Data goes into the receive buffer
- Metadata describing message also transferred
  - this is received into separate storage
  - this is called the *status*

# Communication modes

| Sender mode | Notes |
|---|---|
| Synchronous send | Only completes when the receive has completed. |
| Buffered send | Always completes (unless an error occurs), irrespective of receiver. |
| Standard send | Either synchronous or buffered. |
| Receive | Completes when a message has arrived. |

# MPI Sender Modes

| OPERATION | MPI CALL |
|---|---|
| Standard send | **`MPI_Send`** |
| Synchronous send | **`MPI_Ssend`** |
| Buffered send | **`MPI_Bsend`** |
| Receive | **`MPI_Recv`** |

# Sending a message

- C:

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

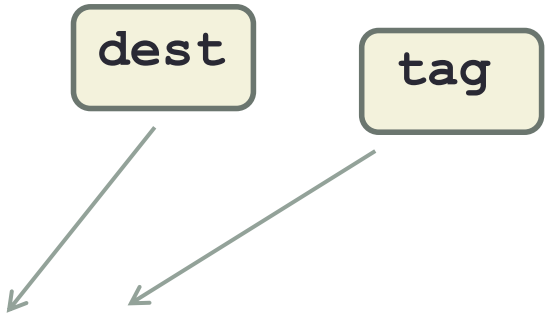
- Fortran:

```
MPI_SSEND(BUF, COUNT, DATATYPE, DEST,
          TAG, COMM, IERROR)

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

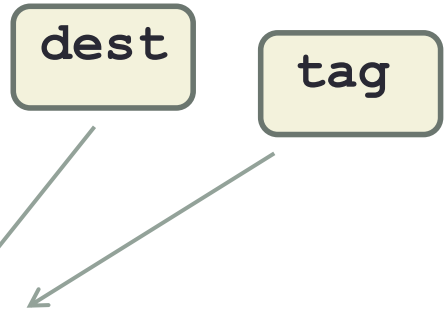
# Send data from rank 1 to rank 3

```
// Array of ten integers
int x[10];
...
if (rank == 1)
MPI_Ssend(x, 10, MPI_INT, 3, 0, MPI_COMM_WORLD);
```




```
// Integer scalar
int x;
...
if (rank == 1)
MPI_Ssend(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);
```

# Send data from rank 1 to rank 3




```
! Array of ten integers
integer, dimension(10) :: x
...
if (rank .eq. 1)
CALL MPI_SSEND(x, 10, MPI_INTEGER, 3, 0,
               MPI_COMM_WORLD, ierr)
```

```
! Integer scalar
integer :: x
...
if (rank .eq. 1)
CALL MPI_SSEND(x, 1, MPI_INTEGER, 3, 0,
               MPI_COMM_WORLD, ierr)
```

# Receiving a message

- C:

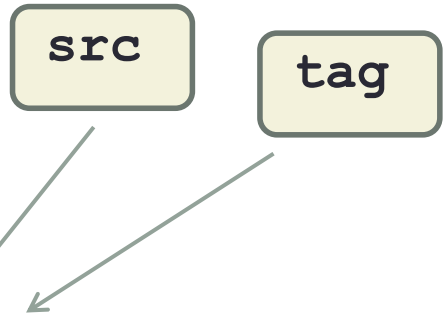
```
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm, MPI_Status *status)
```

- Fortran:

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

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

# Receive data from rank 1 on rank 3




```
int y[10];
MPI_Status status;
...
if (rank == 3)
MPI_Recv(y, 10, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
```

```
int y;
...
if (rank == 3)
MPI_Recv(&y, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
```

# Receive data from rank 1 on rank 3

```
integer, dimension(10) :: y
integer, dimension(MPI_STATUS_SIZE) :: status
...
if (rank .eq. 3)
CALL MPI_RECV(y, 10, MPI_INTEGER, 1, 0,
              MPI_COMM_WORLD, status, ierr)

integer :: y
...
if (rank .eq. 3)
CALL MPI_RECV(y, 1, MPI_INTEGER, 1, 0,
              MPI_COMM_WORLD, status, ierr)
```

# Synchronous Blocking Message-Passing

- Processes synchronise.
- Sender process specifies the synchronous mode.
- Blocking: both processes wait until the transaction has completed.

# 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.
- Message types 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


**Sender's Address**

**For the attention of:**

**Data**

**Item 1**

**Item 2**

**Item 3**

# Commmunication 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

- C:

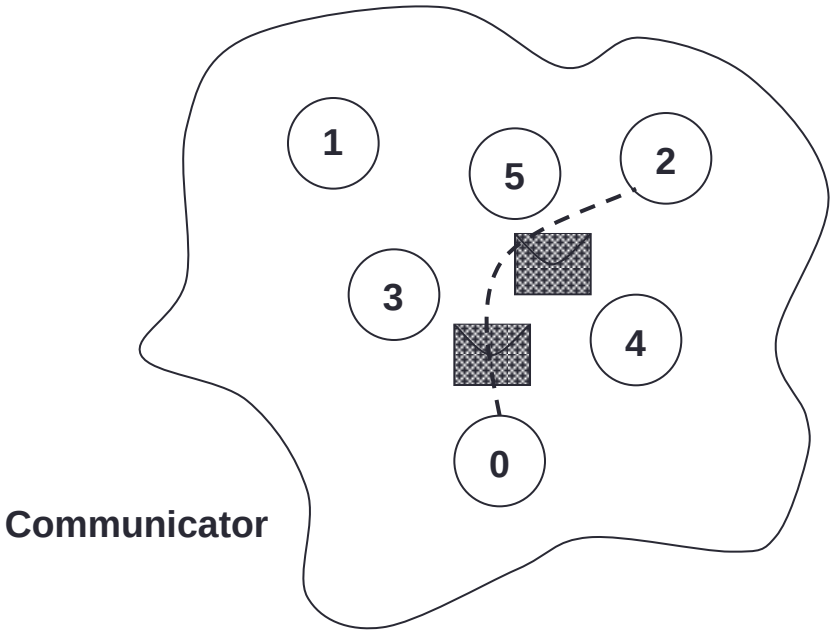
```
int MPI_Get_count( MPI_Status *status,
                   MPI_Datatype datatype,
                   int *count)
```

- Fortran:

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

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

# Message Order Preservation




- Messages do not overtake each other.
- This is true even for non-synchronous sends.

# Message Matching (i)

Rank 0:

```
Ssend(msg1, dest=1, tag=1)
Ssend(msg2, dest=1, tag=2)
```

Rank 1:

```
Recv(buf1, src=0, tag=1)
Recv(buf2, src=0, tag=2)
```

- buf1 = msg1; buf2 = msg2
- Sends and receives correctly matched

# Message Matching (ii)

Rank 0:

```
Ssend(msg1, dest=1, tag=1)
Ssend(msg2, dest=1, tag=2)
```

Rank 1:

```
Recv(buf2, src=0, tag=2)
Recv(buf1, src=0, tag=1)
```

- Deadlock (due to synchronous send)
- Sends and receives incorrectly matched

# Message Matching (iii)

Rank 0:

```
Bsend(msg1, dest=1, tag=1)
Bsend(msg2, dest=1, tag=1)
```

Rank 1:

```
Recv(buf1, src=0, tag=1)
Recv(buf2, src=0, tag=1)
```

- buf1 = msg1; buf2 = msg2
- Messages have same tags but matched in order

# Message Matching (iv)

Rank 0:

```
Bsend(msg1, dest=1, tag=1)
Bsend(msg2, dest=1, tag=2)
```

Rank 1:

```
Recv(buf2, src=0, tag=2)
Recv(buf1, src=0, tag=1)
```

- buf1 = msg1; buf2 = msg2
- Do not *have* to receive messages in order!

# Message Matching (v)

Rank 0:

```
Bsend(msg1, dest=1, tag=1)
Bsend(msg2, dest=1, tag=2)
```

Rank 1:

```
Recv(buf1, src=0, tag=MPI_ANY_TAG)
Recv(buf2, src=0, tag=MPI_ANY_TAG)
```

- buf1 = msg1; buf2 = msg2
- Messages *guaranteed* to match in send order
  - examine status to find out the actual tag values

# Message Order Preservation

- If a receive matches multiple messages in the “inbox”
  - then the messages will be received in the order they were sent
- Only relevant for multiple messages from the same source

# Exercise – Calculation of Pi

- See Exercise 2 on the exercise sheet
- Illustrates how to divide work based on rank
  - and how to send point-to-point messages in an SPMD code
- Notes:
  - the value of *N* in the expansion of pi is not the same as the number of processors
  - you should expect to write a program such as *N*=100 running on 4 processors
  - your code should be able to run on any number of processors
  - do not hard code the number of processors in your program!
- If you finish the pi example you may want to try Exercise 3 (ping-pong) but it is not essential

# Timers

- C:

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

- Fortran:

```
DOUBLE PRECISION MPI_WTIME()
```

- Time is measured in seconds.
- Time to perform a task is measured by consulting the timer before and after
  - subtract values to get elapsed time
- Modify your program to measure its execution time and print it out.