Message Passing Programming with MPI

What is MPI?

MPI Forum

- First message-passing interface standard.
- Sixty people from forty different organisations.
- Users and vendors represented, from the US and Europe.
- Two-year process of proposals, meetings and review.
- *Message Passing Interface* document produced.

Goals and Scope of MPI

- MPI's prime goals are:
  - To provide source-code portability.
  - To allow efficient implementation.
- It also offers:
  - A great deal of functionality.
  - Support for heterogeneous parallel architectures.

Header files

- C:

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

- Fortran:

```
include ‘mpif.h’
```

MPI Function Format

- C:

```
error = MPI_Xxxxx(parameter, ...);

MPI_Xxxxx(parameter, ...);
```

- Fortran:

```
CALL MPI_XXXXX(parameter, ..., IERROR)
```

Handles

- MPI controls its own internal data structures.
- MPI releases `handles' to allow programmers to refer to these.
- C handles are of defined **`typedefs`.**
- Fortran handles are **`INTEGER`**s.

Initialising MPI

- C:

```
int MPI_Init(int *argc, char ***argv)
```

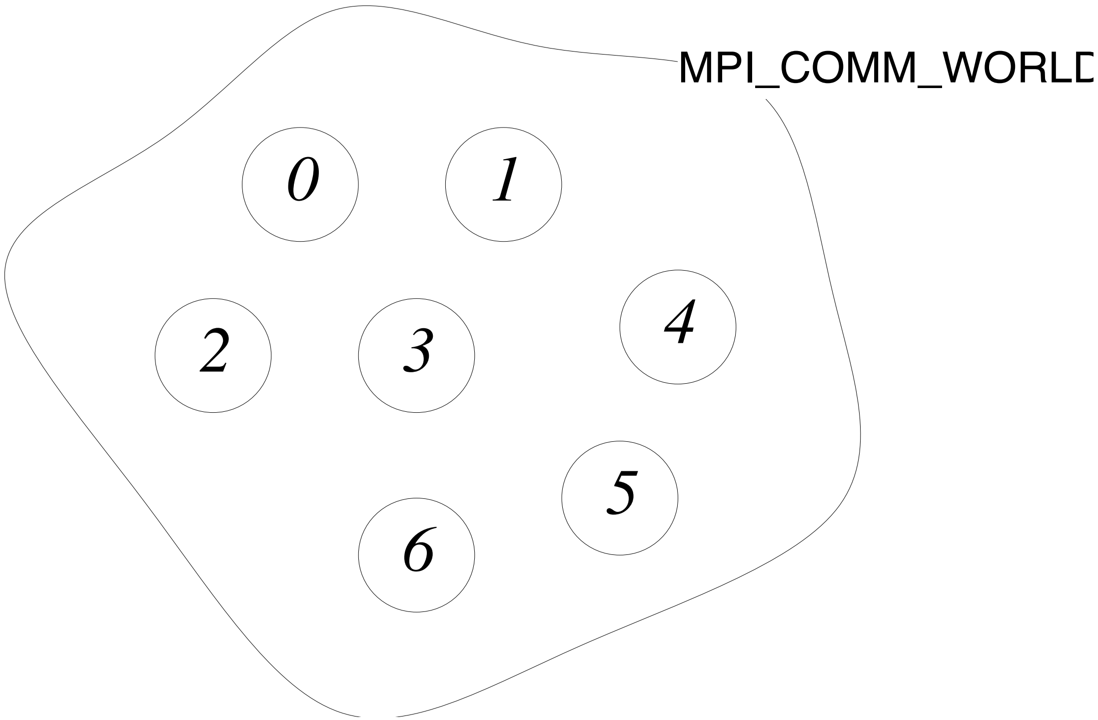
- Fortran:

```
MPI_INIT(IERROR)
   INTEGER IERROR
```

- Must be the first MPI procedure called.

MPI_COMM_WORLD

Communicators



Rank

- How do you identify different processes in a communicator?

```
MPI_Comm_rank(MPI_Comm comm, int *rank)

MPI_COMM_RANK(COMM, RANK, IERROR)
   INTEGER COMM, RANK, IERROR
```

- The rank is not the PE number.

Size

- How many processes are contained within a communicator?

```
MPI_Comm_size(MPI_Comm comm, int *size)

MPI_COMM_SIZE(COMM, SIZE, IERROR)
   INTEGER COMM, SIZE, IERROR
```

epcc

Exiting MPI

- C:

```
int MPI_Finalize()
```

- Fortran:

```
MPI_FINALIZE(IERROR)
   INTEGER IERROR
```

- Must be the last MPI procedure called.

Messages

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

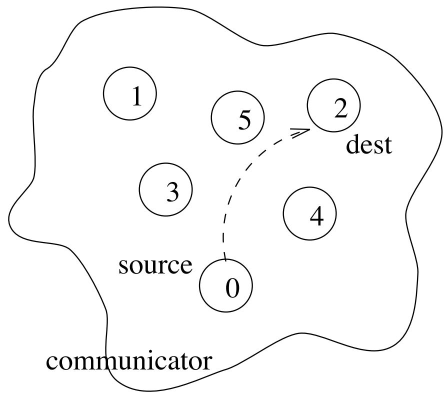
MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long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

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	
MPI_PACKED	

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.

Sending a message

- C:

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

- Fortran:

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

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
```

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.

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

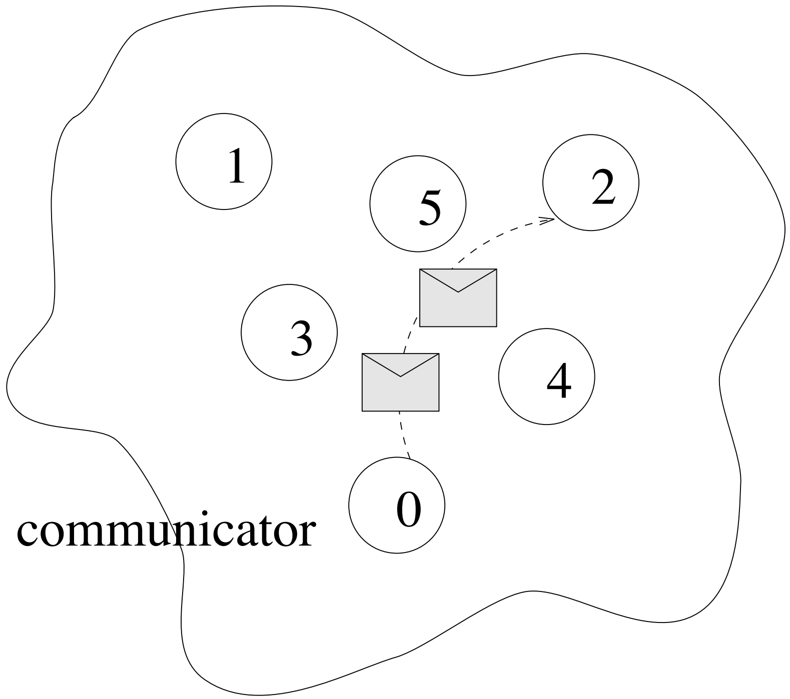
- C:

```
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.

Timers

- C:

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

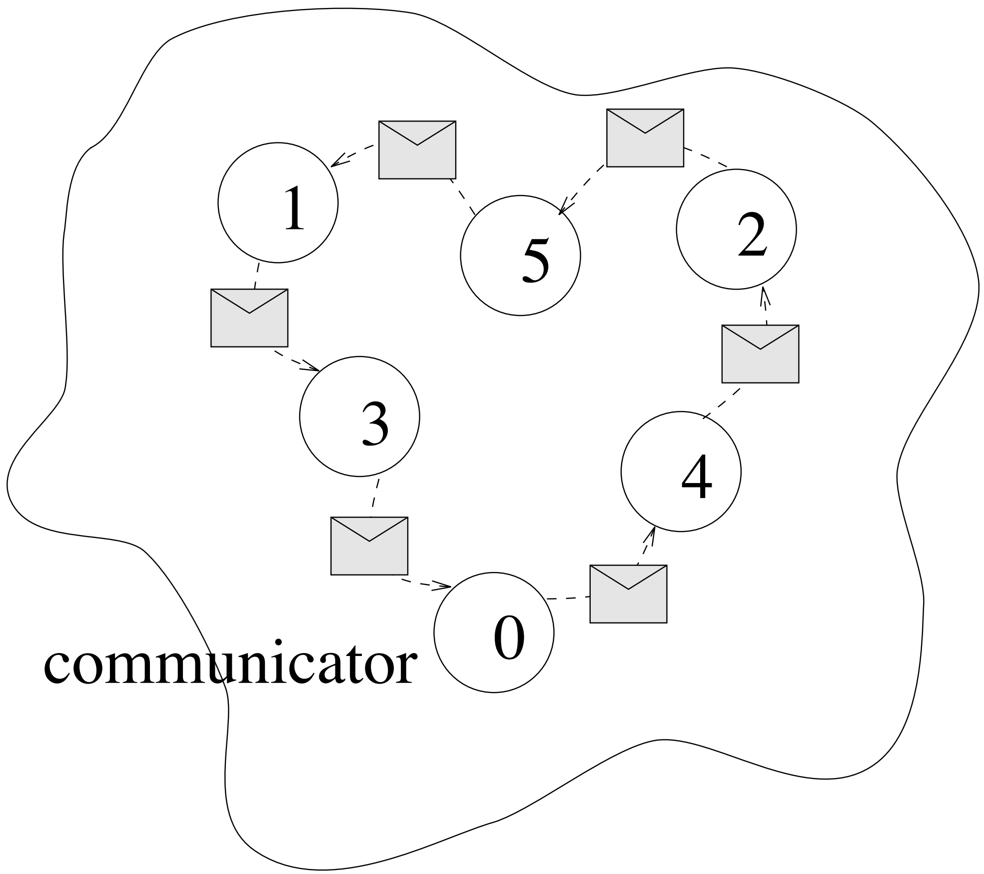
- Fortran:

```
DOUBLE PRECISION MPI_WTIME()
```

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

Non-Blocking Communications

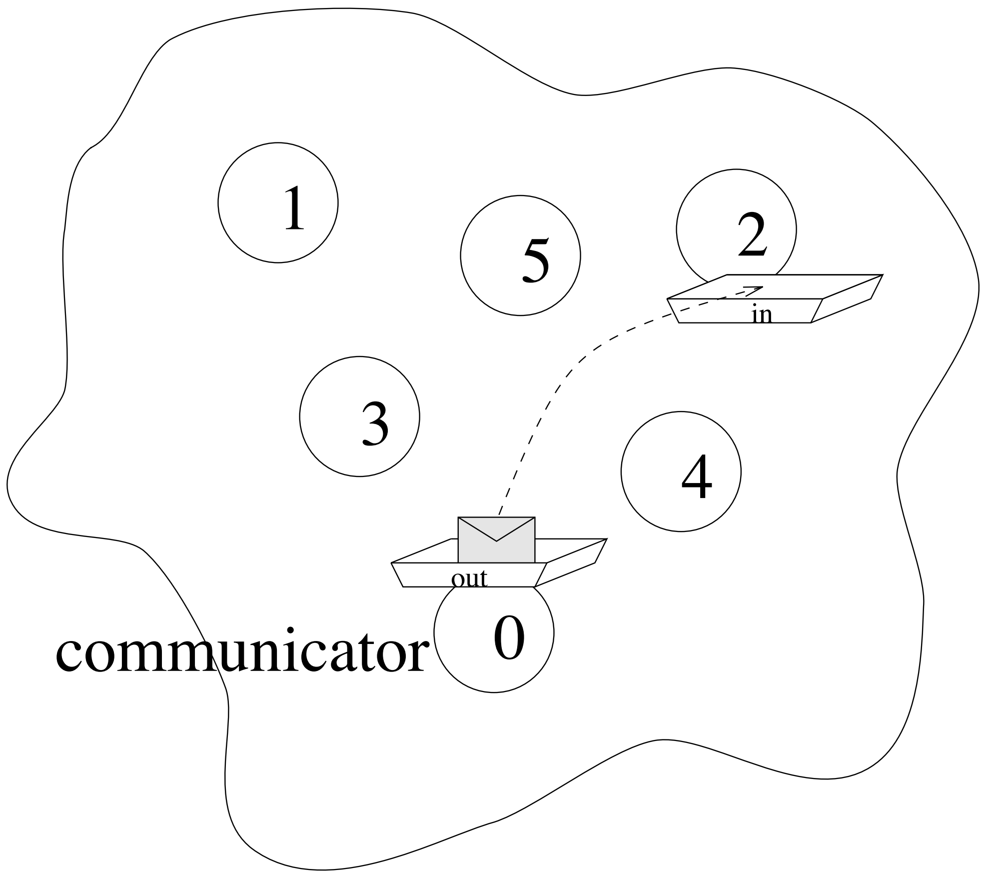
Deadlock



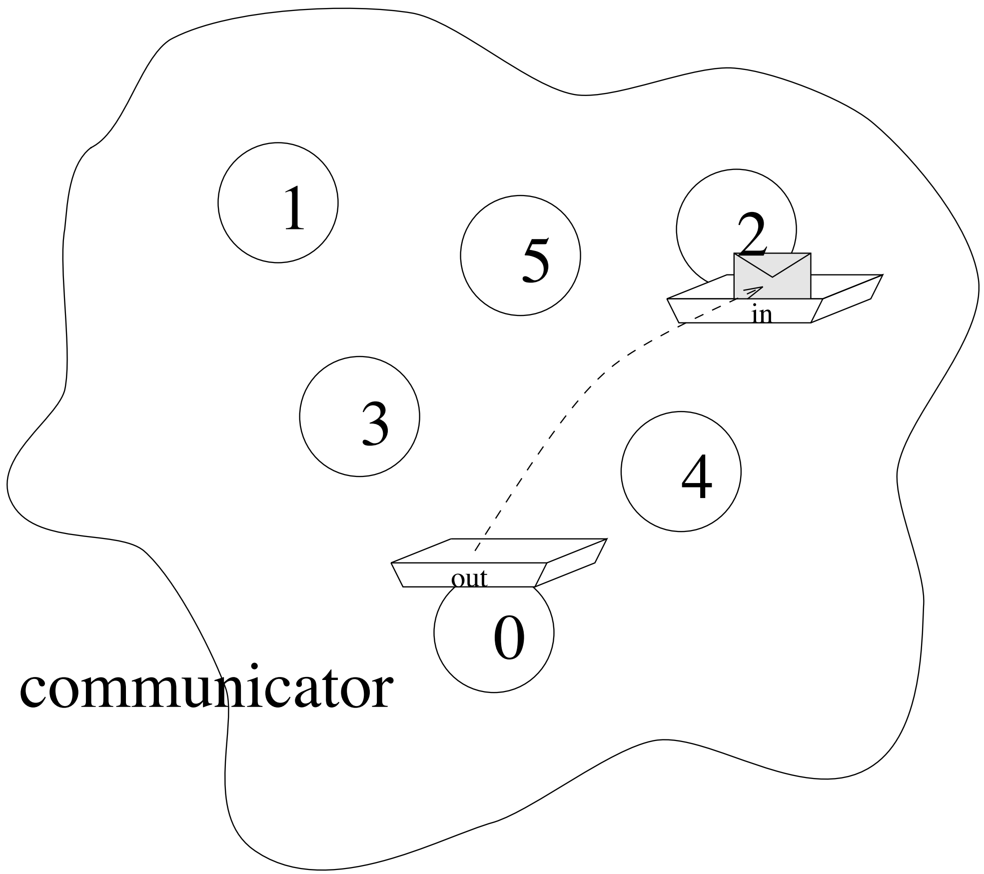
Non-Blocking Communications

- 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_Isend(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_ISEND(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.

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)
```

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.

MPI Datatypes

- Basic types
- Derived types
  - vectors
  - structs
  - others

Derived Datatypes - *Type*

basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1

Contiguous Data

- The simplest derived datatype consists of a number of contiguous items of the same datatype.
- C:

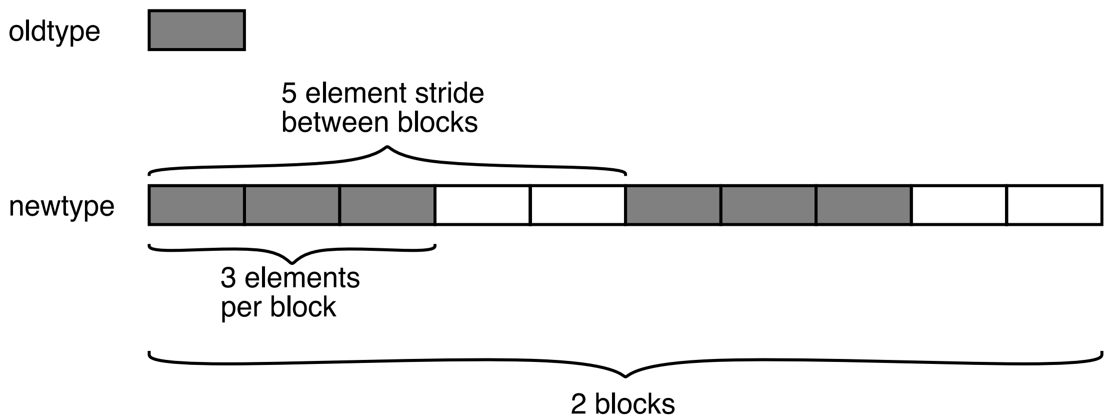
```
int MPI_Type_contiguous(int count,
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Fortran:

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE,
                    IERROR)
   INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

Vector Datatype Example

A 3X2 block of a 5X5 Fortran array



- count = 2
- stride = 5
- blocklength = 3

Constructing a Vector Datatype

- C:

```
int MPI_Type_vector (int count,
      int blocklength, int stride,
      MPI_Datatype oldtype,
      MPI_Datatype *newtype)
```

- Fortran:

```
MPI_TYPE_VECTOR (COUNT, BLOCKLENGTH,
   STRIDE, OLDTYPE, NEWTYPE, IERROR)
```

Extent of a Datatatype

- C:

```
int MPI_Type_extent (MPI_Datatype datatype,
                     MPI_Aint *extent)
```

- Fortran:

```
MPI_TYPE_EXTENT( DATATYPE, EXTENT,
                 IERROR)
INTEGER DATATYPE, EXTENT, IERROR
```

Address of a Variable

- C:

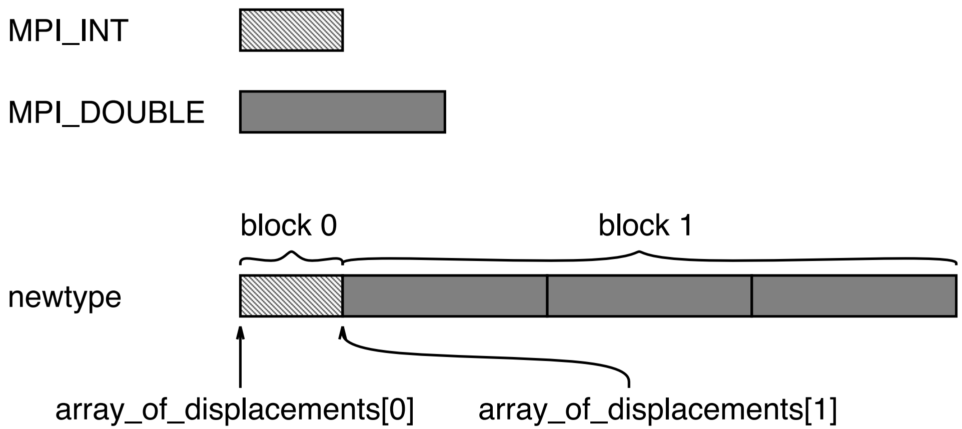
```
int MPI_Address (void *location, MPI_Aint
                 *address)
```

- Fortran:

```
MPI_ADDRESS( LOCATION, ADDRESS, IERROR)

<type> LOCATION (*)
INTEGER ADDRESS, IERROR
```

Struct Datatype Example



- `count = 2`
- `array_of_blocklengths[0] = 1`
- `array_of_types[0] = MPI_INT`
- `array_of_blocklengths[1] = 3`
- `array_of_types[1] = MPI_DOUBLE`

Constructing a Struct Datatype

- C:

```
int MPI_Type_struct (int count,
   int *array_of_blocklengths,
   MPI_Aint *array_of_displacements,
   MPI_Datatype *array_of_types,
   MPI_Datatype *newtype)
```

- Fortran:

```
MPI_TYPE_STRUCT (COUNT,
   ARRAY_OF_BLOCKLENGTHS,
   ARRAY_OF_DISPLACEMENTS,
   ARRAY_OF_TYPES, NEWTYPE, IERROR)
```

Committing a datatype

- Once a datatype has been constructed, it needs to be committed before it is used.
- This is done using **MPI_TYPE_COMMIT**
- C:

```
int MPI_Type_commit (MPI_Datatype *datatype)
```

- Fortran:

```
MPI_TYPE_COMMIT (DATATYPE, IERROR)
   INTEGER DATATYPE, IERROR
```

Virtual Topologies

Virtual Topologies

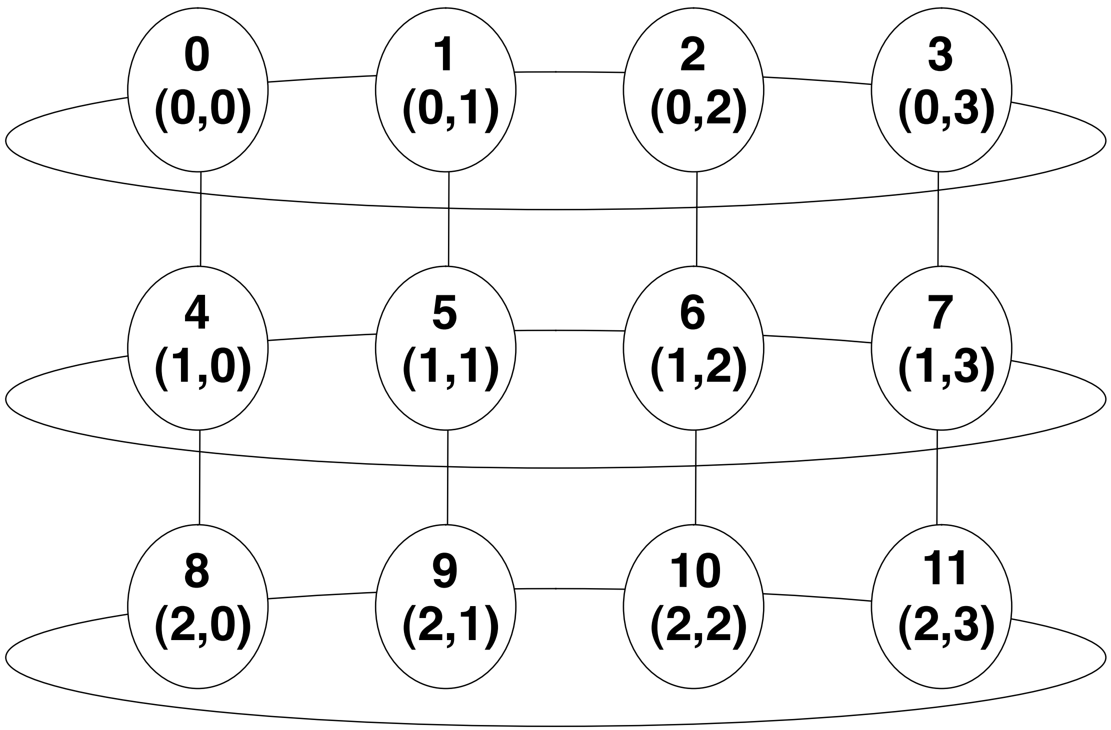
- Convenient process naming.
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimise communications.

How to use a Virtual Topology

- Creating a topology produces a new communicator.
- MPI provides ``mapping functions''.
- Mapping functions compute processor ranks, based on the topology naming scheme.

Example

A 2-dimensional Cylinder



Topology types

- Cartesian topologies
  - each process is “connected” to its neighbours in a virtual grid.
  - boundaries can be cyclic, or not.
  - processes are identified by cartesian coordinates.
- Graph topologies
  - general graphs
  - not covered here

Creating a Cartesian Virtual Topology

- C:

```
int MPI_Cart_create(MPI_Comm comm_old,
      int ndims, int *dims, int *periods,
      int reorder, MPI_Comm *comm_cart)
```

- Fortran:

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS,
      PERIODS, REORDER, COMM_CART, IERROR)

INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART,
        IERROR
LOGICAL PERIODS(*), REORDER
```

Balanced Processor Distribution

- C:

```
int MPI_Dims_create(int nnodes, int ndims,
                    int *dims)
```

- Fortran:

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)

INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

Example

- Call tries to set dimensions as close to each other as possible.

dims before the call	function call	dims on return
(0, 0)	MPI_DIMS_CREATE(6, 2, dims)	(3, 2)
(0, 0)	MPI_DIMS_CREATE(7, 2, dims)	(7, 1)
(0, 3, 0)	MPI_DIMS_CREATE(6, 3, dims)	(2, 3, 1)
(0, 3, 0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

- Non zero values in dims sets the number of processors required in that direction.

WARNING:- make sure dims is set to 0 before the call!

Cartesian Mapping Functions

Mapping process grid coordinates to ranks

- C:

```
int MPI_Cart_rank(MPI_Comm comm,
                  int *coords, int *rank)
```

- Fortran:

```
MPI_CART_RANK (COMM, COORDS, RANK, IERROR)

INTEGER COMM, COORDS(*), RANK, IERROR
```

Cartesian Mapping Functions

Mapping ranks to process grid coordinates

- C:

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
                    int maxdims, int *coords)
```

- Fortran:

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS,
                      COORDS, IERROR)

INTEGER COMM, RANK, MAXDIMS, COORDS(*),
        IERROR
```

Cartesian Mapping Functions

Computing ranks of neighbouring processes

- C:

```
int MPI_Cart_shift(MPI_Comm comm,
        int direction, int disp,
        int *rank_source, int *rank_dest)
```

- Fortran:

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP,
        RANK_SOURCE, RANK_DEST, IERROR)

INTEGER  COMM, DIRECTION, DISP,RANK_SOURCE,
        RANK_DEST, IERROR
```

Collective Communications

Collective Communication

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Barrier synchronisation.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.

Characteristics of Collective Comms

- Collective action over a communicator.
- All processes must communicate.
- Synchronisation may or may not occur.
- All collective operations are blocking.
- No tags.
- Receive buffers must be exactly the right size.

Barrier Synchronisation

- C:

```
int MPI_Barrier (MPI_Comm comm)
```

- Fortran:

```
MPI_BARRIER (COMM, IERROR)
INTEGER COMM, IERROR
```

Broadcast

- C:

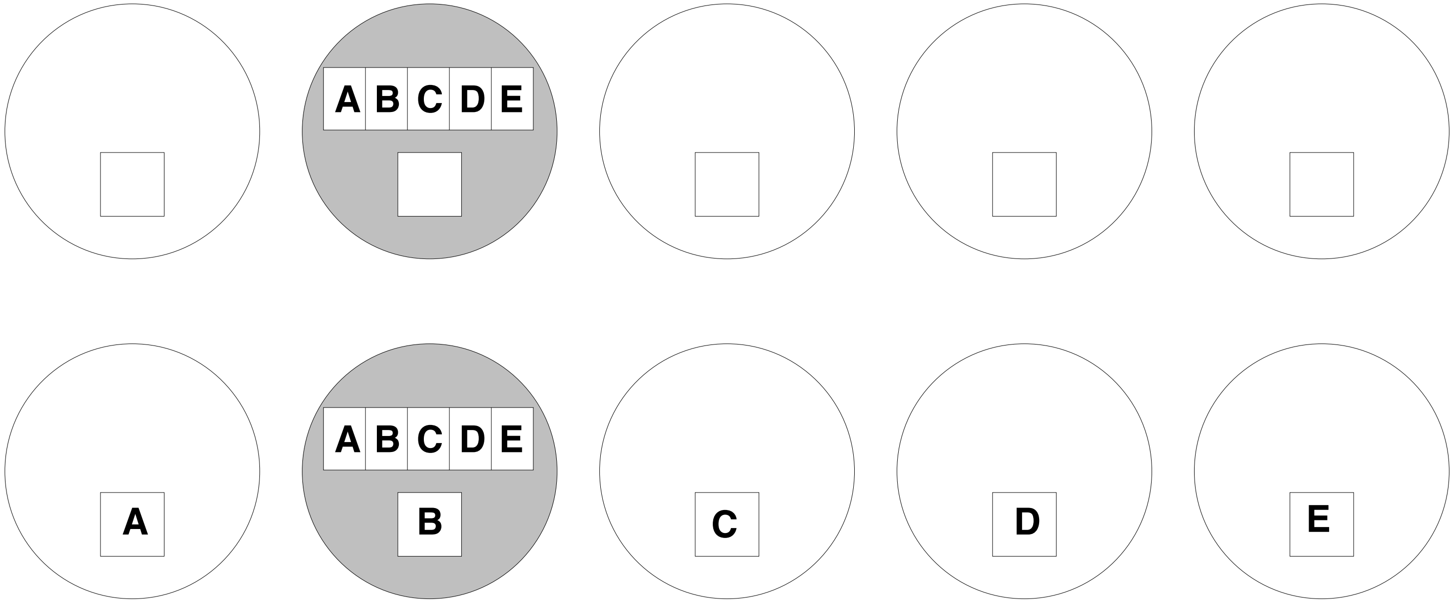
```
int MPI_Bcast (void *buffer, int count,
               MPI_Datatype datatype, int root,
               MPI_Comm comm)
```

- Fortran:

```
MPI_BCAST (BUFFER, COUNT, DATATYPE, ROOT,
           COMM, IERROR)

<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

Scatter



Scatter

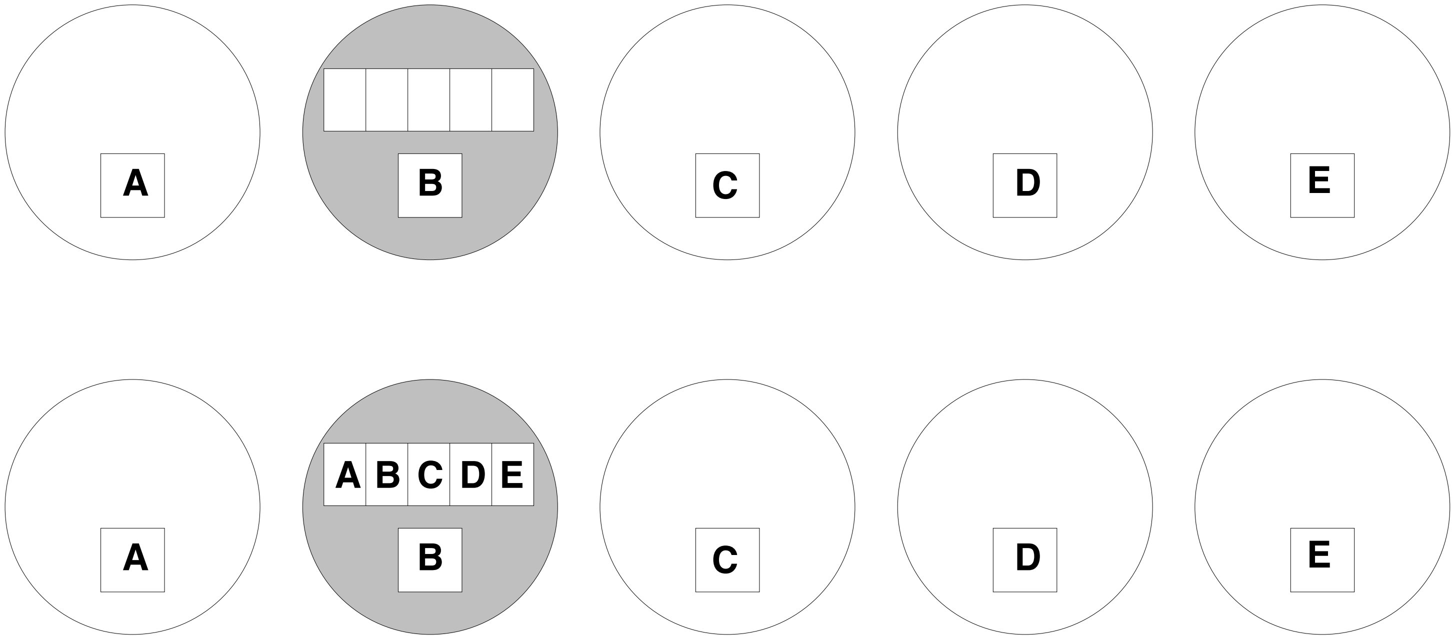
- C:

```
int MPI_Scatter(void *sendbuf,
    int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int root,
    MPI_Comm comm)
```

- Fortran:

```
    MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE,
                RECVBUF, RECVCOUNT, RECVTYPE,
               ROOT, COMM, IERROR)
    <type> SENDBUF, RECVBUF
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
    INTEGER RECVTYPE, ROOT, COMM, IERROR
```

Gather



Gather

- C:

```
int MPI_Gather(void *sendbuf, int sendcount,
      MPI_Datatype sendtype, void *recvbuf,
      int recvcount, MPI_Datatype recvtype,
      int root, MPI_Comm comm)
```

- Fortran:

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE,
      RECVBUF, RECVCOUNT, RECVTYPE,
      ROOT, COMM, IERROR)
<type> SENDBUF, RECVBUF
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
INTEGER RECVTYPE, ROOT, COMM, IERROR
```

Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes.
- Examples:

  global sum or product

  global maximum or minimum

  global user-defined operation

Predefined Reduction Operations

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

MPI_REDUCE

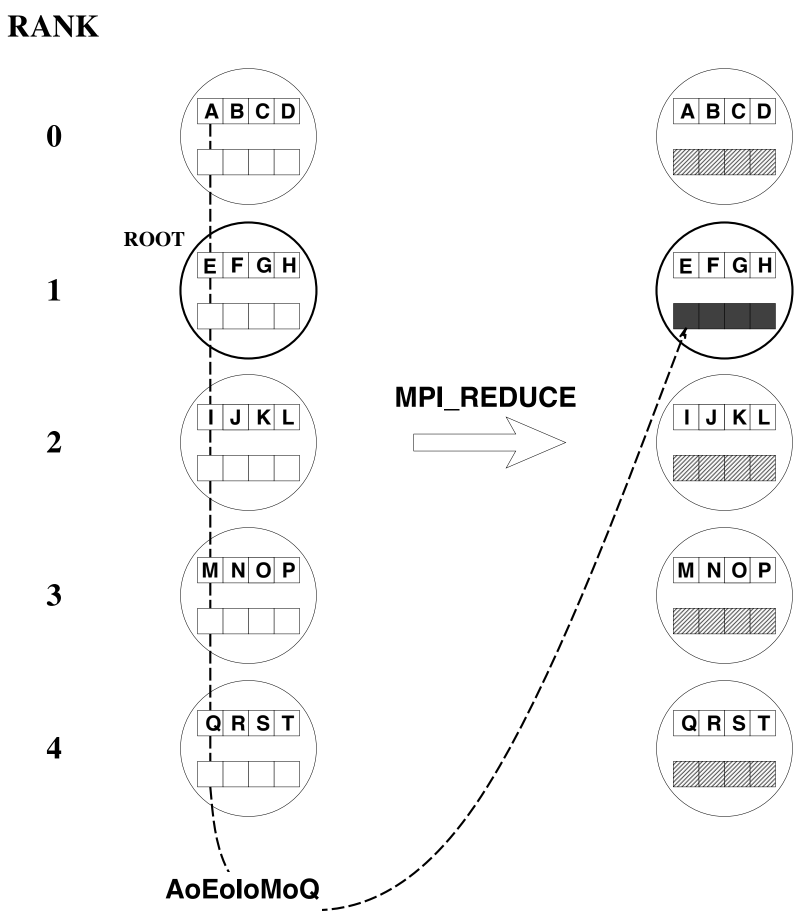
- C:

```
int MPI_Reduce(void *sendbuf,
    void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op,
    int root, MPI_Comm comm)
```

- Fortran:

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT,
   DATATYPE, OP, ROOT, COMM, IERROR)
<type> SENDBUF, RECVBUF
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
INTEGER RECVTYPE, ROOT, COMM, IERROR
```

MPI_REDUCE


RANK
0
1
2
3
4
ROOT
A B C D
E F G H
I J K L
M N O P
Q R S T
MPI_REDUCE
AoEoIoMoQ

Example of Global Reduction

Integer global sum

- C:

```
MPI_Reduce(&x, &result, 1, MPI_INT, MPI_SUM,
           0, MPI_COMM_WORLD)
```

- Fortran:

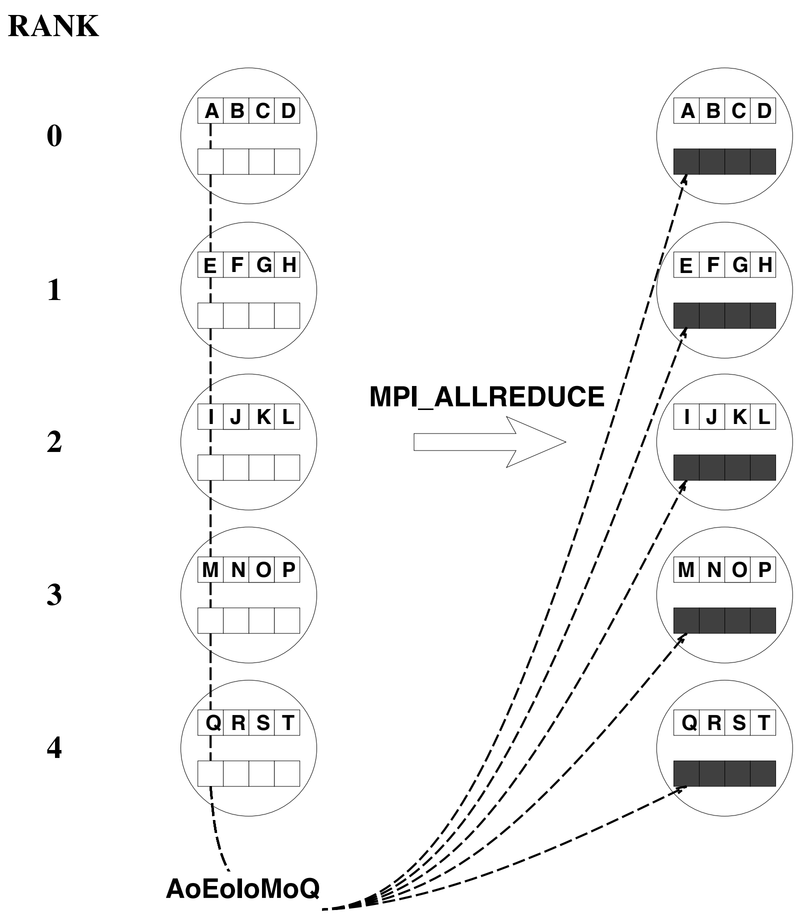
```
CALL MPI_REDUCE(x, result, 1, MPI_INTEGER,
        MPI_SUM, 0, MPI_COMM_WORLD, IERROR)
```

- Sum of all the **x** values is placed in **result.**
- The result is only placed there on processor 0.

Variants of MPI_REDUCE

- **MPI_ALLREDUCE** – no root process
- **MPI_REDUCE_SCATTER** – result is scattered
- **MPI_SCAN** – “parallel prefix”

MPI_ALLREDUCE



MPI_ALLREDUCE

Integer global sum

- C:

```
int MPI_Allreduce(void* sendbuf,
         void* recvbuf, int count,
         MPI_Datatype datatype,
         MPI_Op op, MPI_Comm comm)
```

- Fortran:

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT,
           DATATYPE, OP, COMM, IERROR)
```