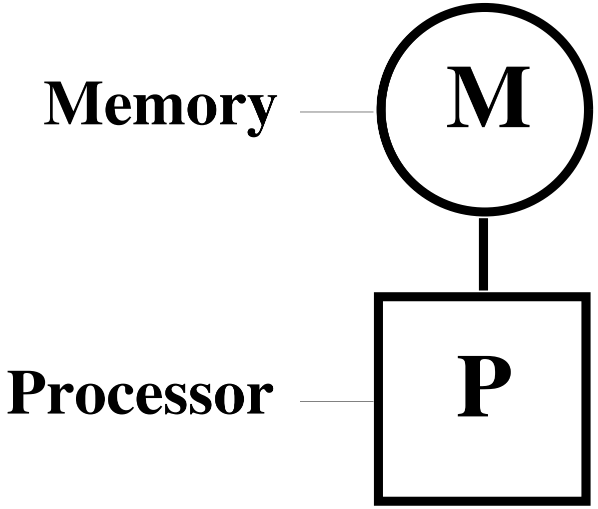
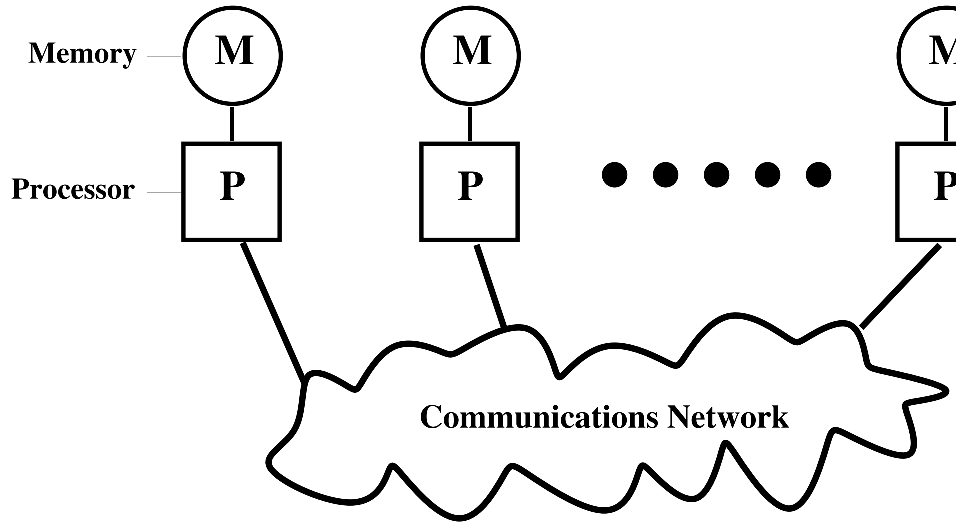
Writing Message-Passing Parallel Programs with MPI

Getting Started

Sequential Programming Paradigm



Message-Passing Programming Paradigm



Message-Passing Programming Paradigm (cont'd)

- Each processor in a message passing program runs a *sub-program*.
  - written in a conventional sequential language.
  - all variables are private.
  - communicate via special subroutine calls.

What is SPMD?

- *Single Program, Multiple Data*
- Same program runs everywhere.
- Restriction on the general message-passing model.
- Some vendors only support SPMD parallel programs.
- General message-passing model can be emulated.

Emulating General Message Passing with SPMD: C

```
main (int argc, char **argv)
    {
        if (process is to become a controller process)
        {
            Controller( /* Arguments /* );
        }
        else
        {
            Worker( /* Arguments /* );
        }
    }
```

Emulating General Message-Passing with SPMD: Fortran

```
PROGRAM
IF (process is to become a controller process) THEN
    CALL CONTROLLER ( /* Arguments /* )
ELSE
    CALL WORKER ( /* Arguments /* )
ENDIF
END
```

Messages

- Messages are packets of data moving between sub-programs.
- The message passing system has to be told the following information:
  - Sending processor
  - Source location
  - Data type
  - Data length
  - Receiving processor(s)
  - Destination location
  - Destination size

Access

- A sub-program needs to be connected to a message passing system.
- A message passing system is similar to:
  - Mail box
  - Phone line
  - fax machine
  - etc.

Addressing

- Messages need to have addresses to be sent to.
- Addresses are similar to:
  - Mail address
  - Phone number
  - fax number
  - etc.

Reception

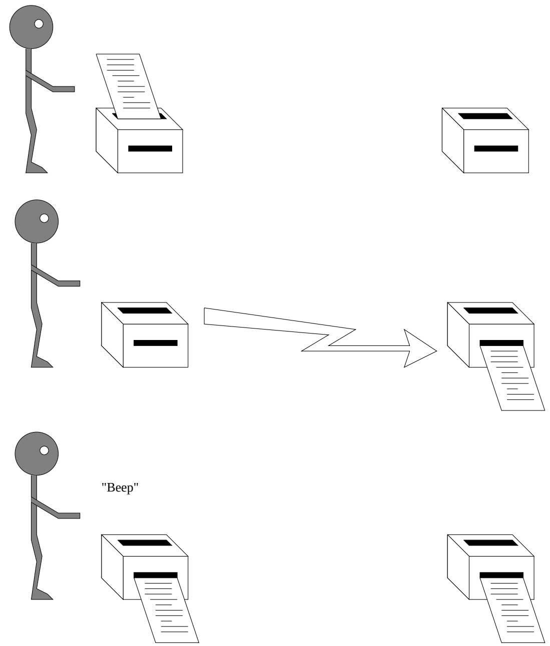
- It is important that the receiving process is capable of dealing with messages it is sent.

Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another
- Different types of point-to-point communication

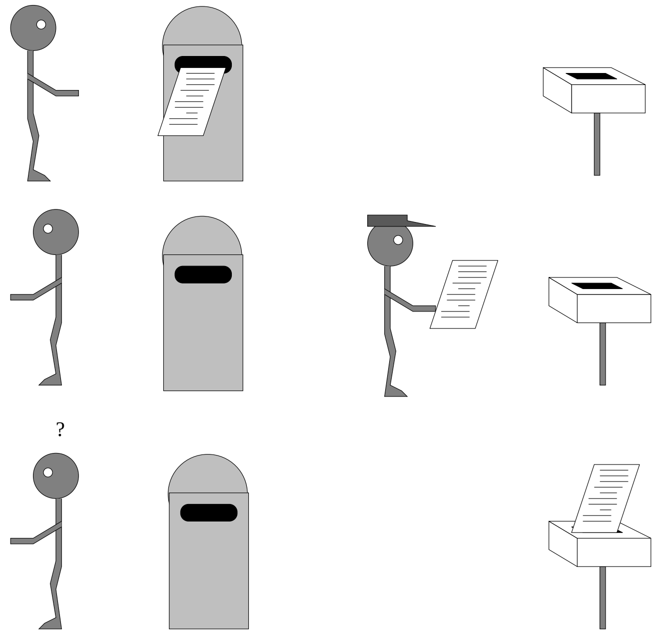
Synchronous Sends

- Provide information about the completion of the message.



Asynchronous Sends

- Only know when the message has left.

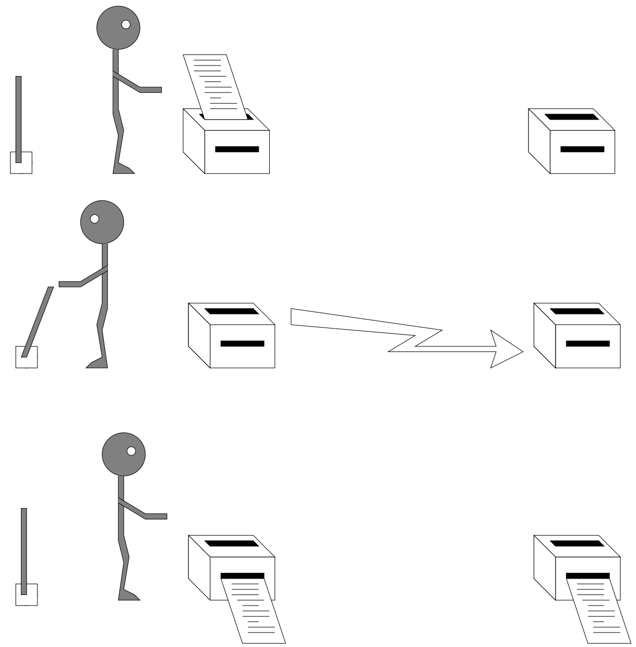


Blocking Operations

- Relate to when the operation has completed.
- Only return from the subroutine call when the operation has completed.

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 (cont'd)

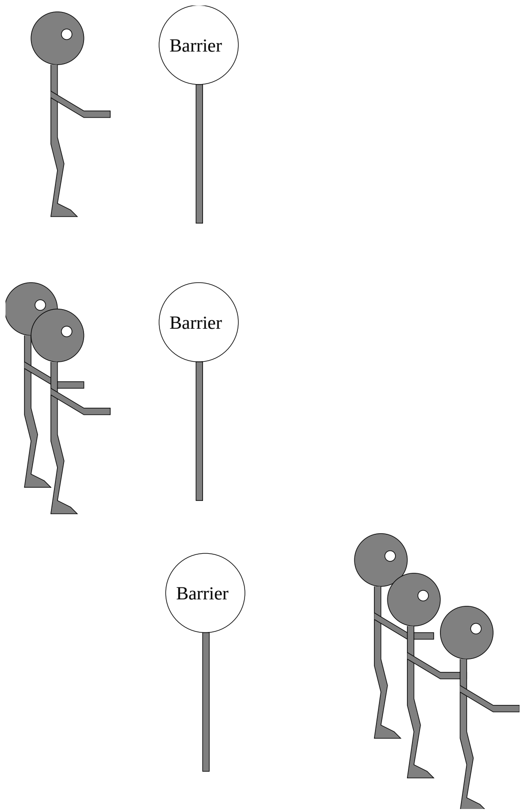
- All non-blocking operations should have matching wait operations. Some systems cannot free resources until wait has been called.
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation.
- Non-blocking operations are not the same as sequential subroutine calls as the operation continues after the call has returned.

Collective communications

- ❑ Collective communication routines are higher level routines involving several processes at a time.
- ❑ Can be built out of point-to-point communications.

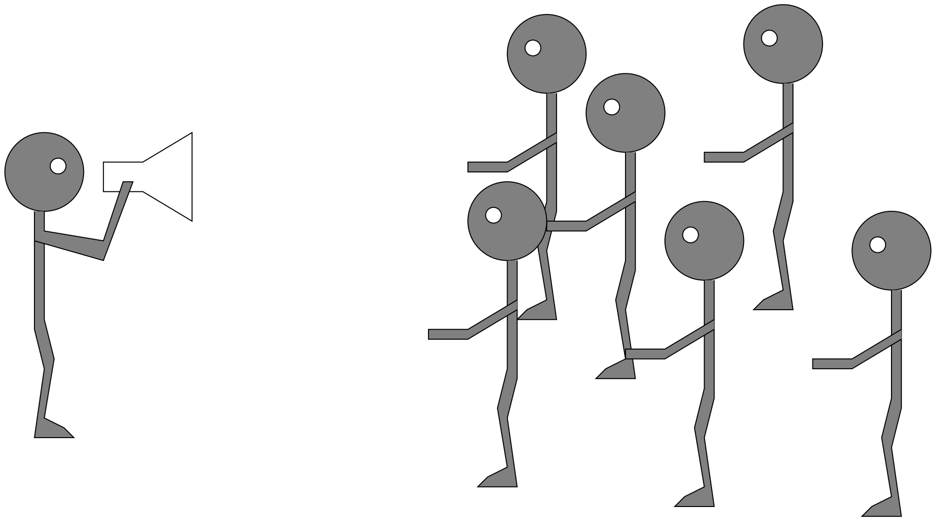
Barriers

- Synchronise processes.



Broadcast

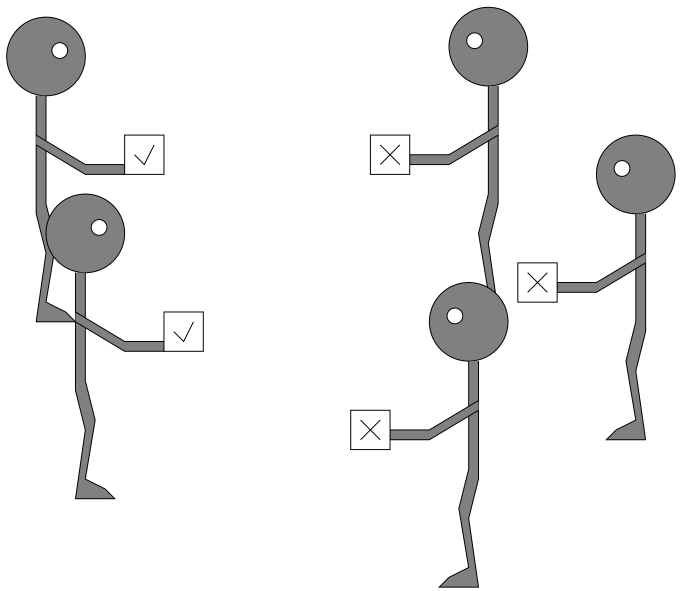
- A one-to-many communication.



Reduction Operations

- Combine data from several processes to produce a single result.

STRIKE



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.

MPI Programs

Header files

- C

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

- Fortran

```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

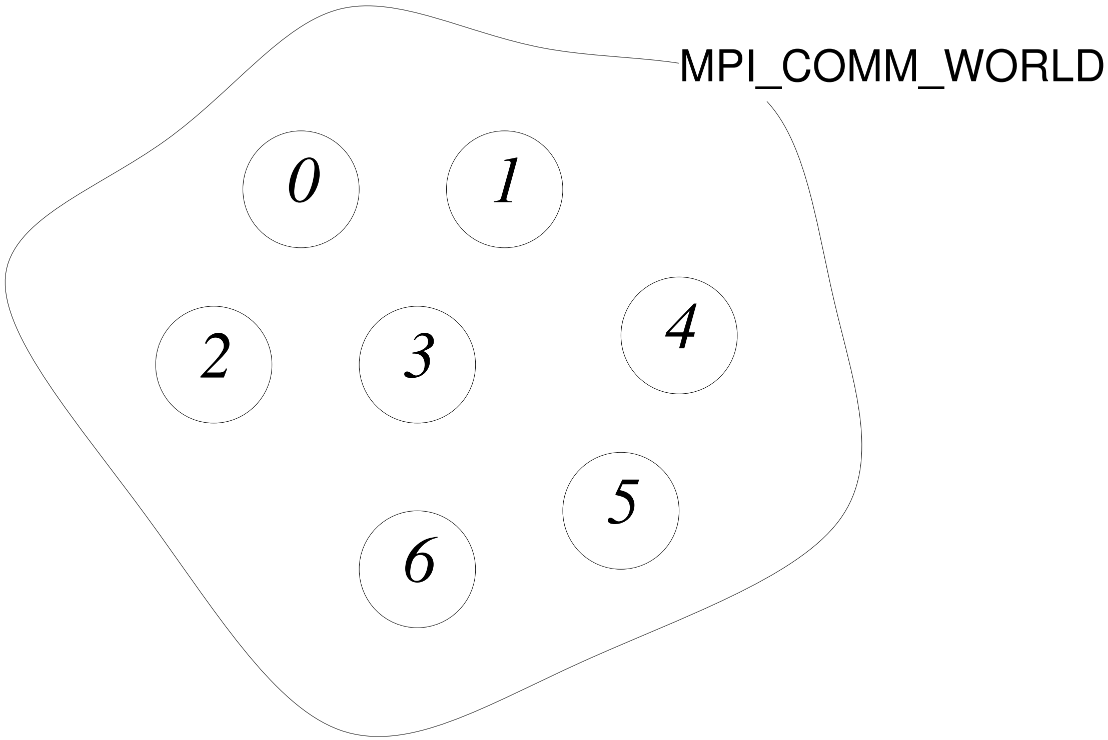
```
int MPI_Init(int *argc, char ***argv)
```

- Fortran

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

- Must be first routine called.

MPI_COMM_WORLD communicator



Rank

- How do you identify different processes?

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

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

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
```

Exiting MPI

- C

```
int MPI_Finalize()
```

- Fortran

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

- Must be called last by all processes.

Exercise: Hello World - the minimal MPI program

- Write a minimal MPI program which prints ``hello world''.
- Compile it.
- Run it on a single processor.
- Run it on several processors in parallel.
- Modify your program so that only the process ranked 0 in `MPI_COMM_WORLD` prints out.
- Modify your program so that the number of processes is printed out.

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

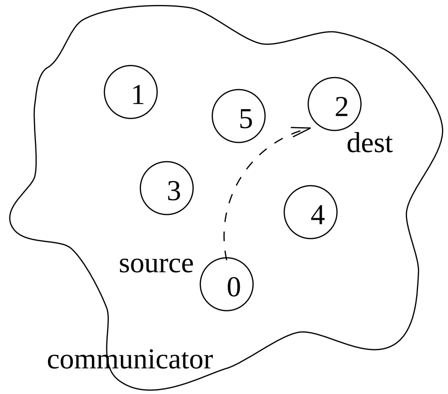
MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long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

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.
Ready send	Always completes (unless an error occurs), irrespective of whether the receive has completed.
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`
Ready send	`MPI_RSEND`
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
```

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.

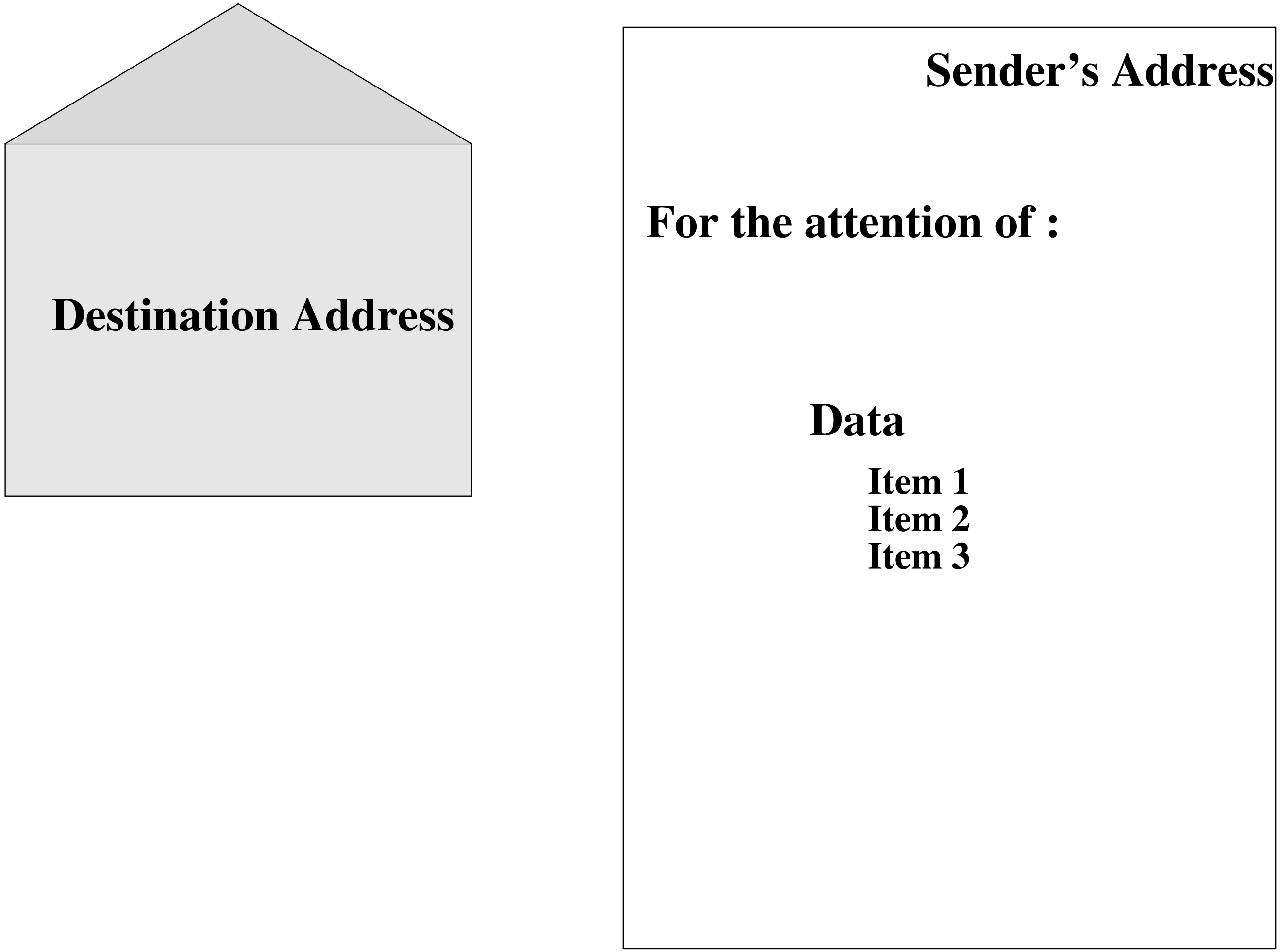
For a communication to succeed:

- 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.

Communication Envelope



Communication Envelope Information

- Envelope information is returned from `MPI_RECV` as `status`
- Information includes:
  - Source: `status.MPI_SOURCE` or `sta-tus(MPI_SOURCE)`
  - Tag: `status.MPI_TAG` or `status(MPI_TAG)`
  - Count: `MPI_Get_count` or `MPI_GET_COUNT`

Received Message Count

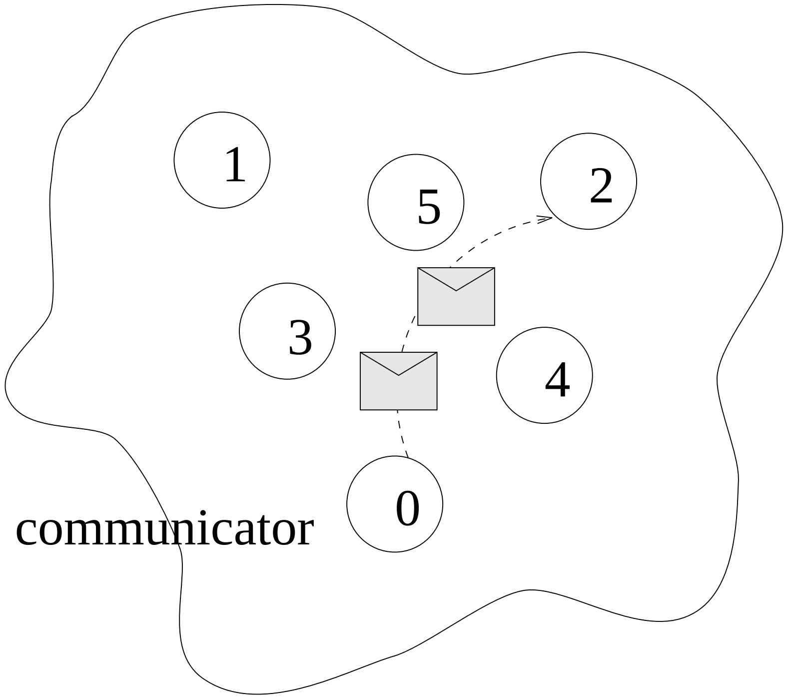
- C:

```
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.

Exercise - 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.

Timers

- C:

  double MPI_Wtime(void);

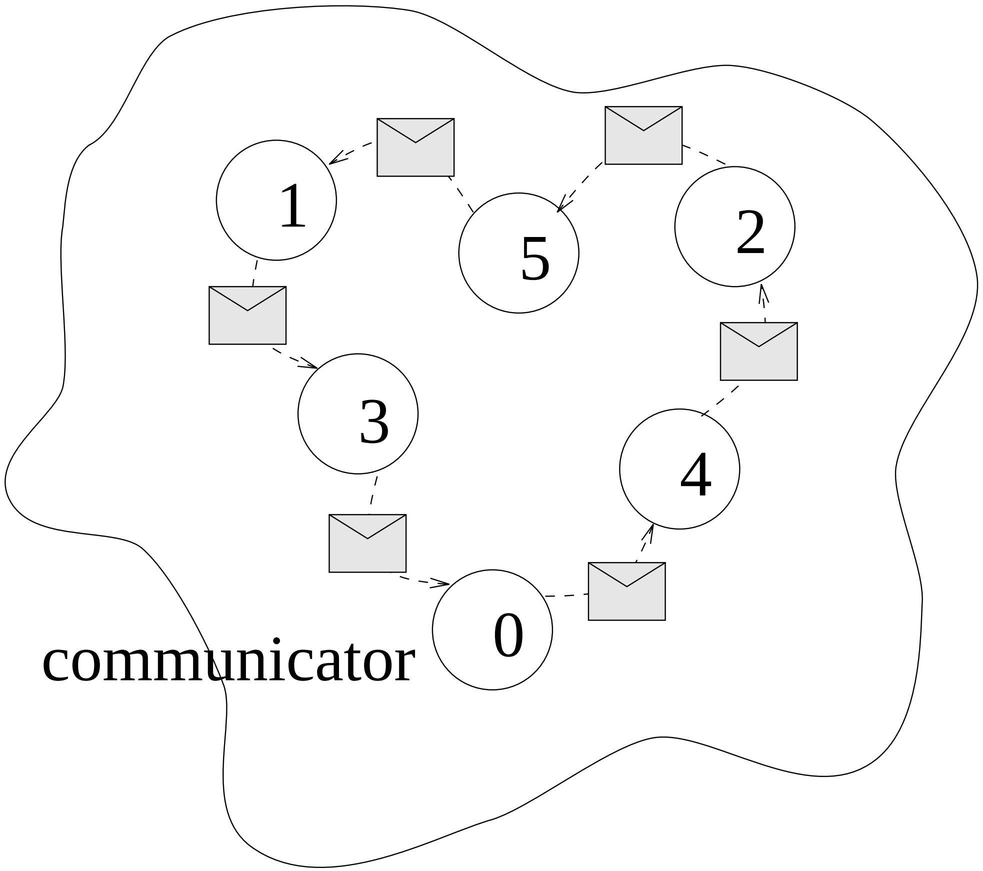
- Fortran:

  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

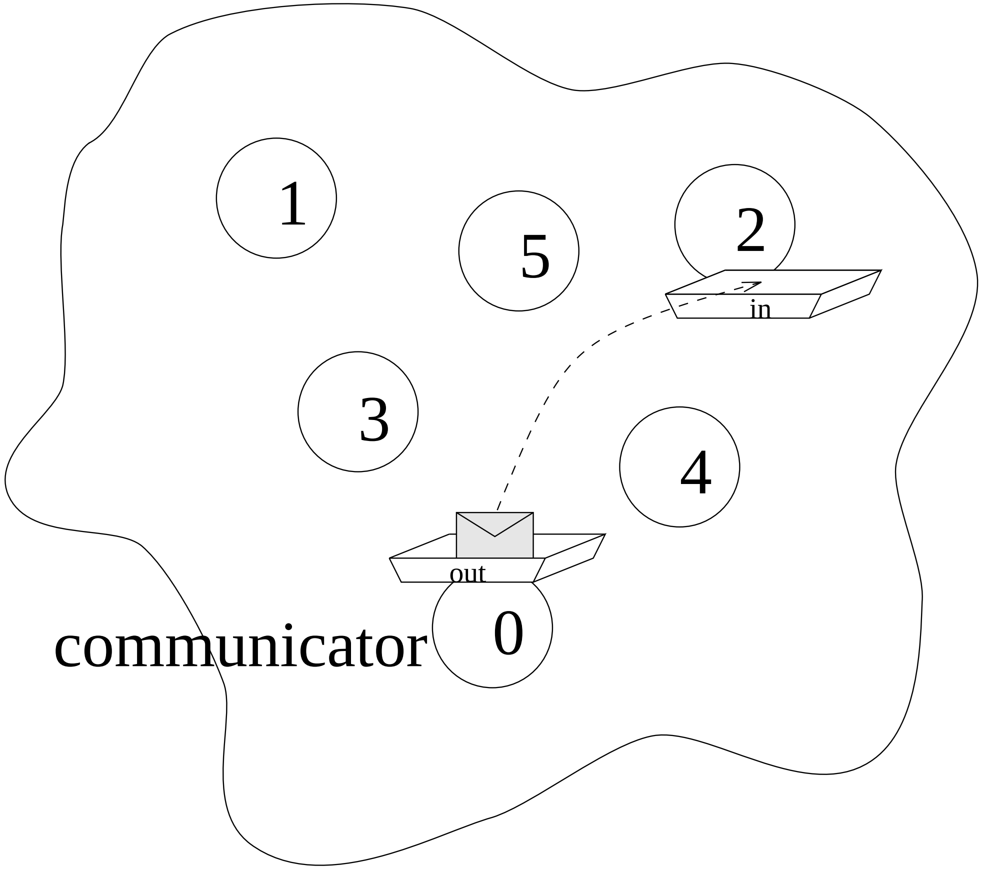
Deadlock



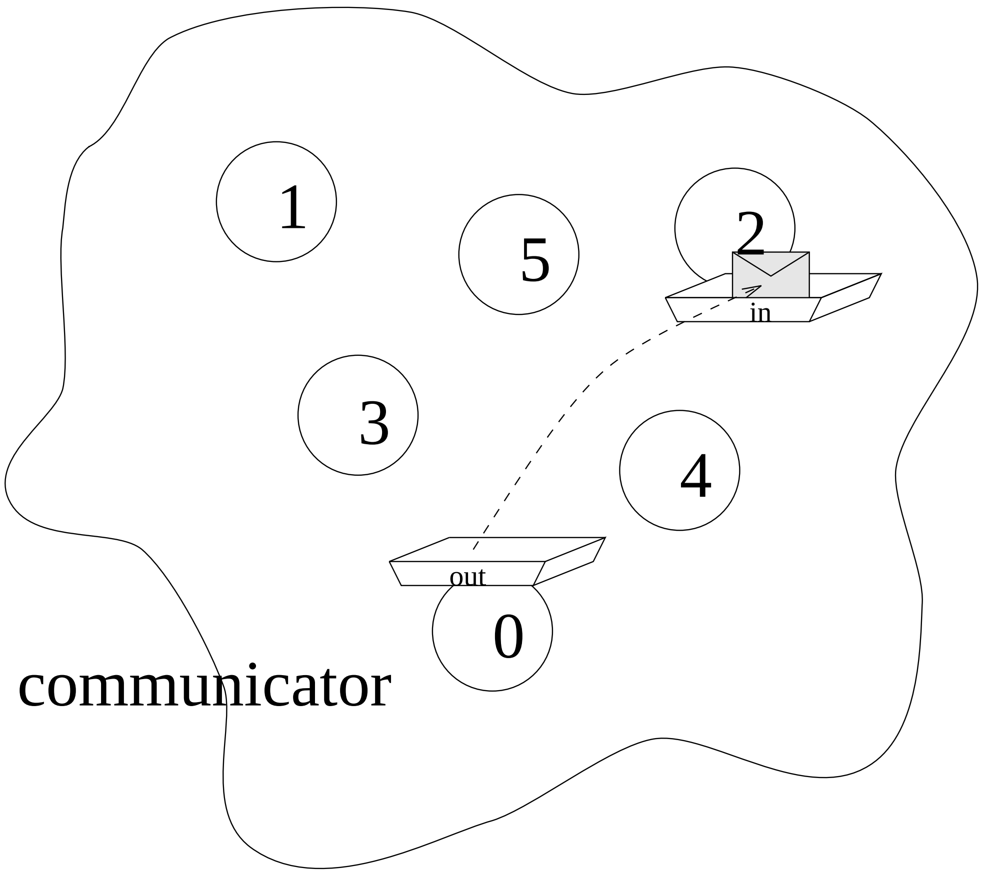
Non-Blocking Communications

- 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 Communication

- 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:

  MPI_Issend(buf, count, datatype, dest, tag, comm, handle)

  MPI_Wait(handle, status)

- Fortran:

  MPI_ISSEND(buf, count, datatype, dest, tag,comm, handle, ierror)

  MPI_WAIT(handle, status, ierror)

Non-blocking Receive

- C:

  MPI_Irecv(buf, count, datatype, src, tag,comm, handle)

  MPI_Wait(handle, status)

- Fortran:

  MPI_IRECV(buf, count, datatype, src, tag,comm, handle,
  ierror)

  MPI_WAIT(handle, 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:

  MPI_Wait(handle, status)

  MPI_Test(handle, flag, status)

- Fortran:

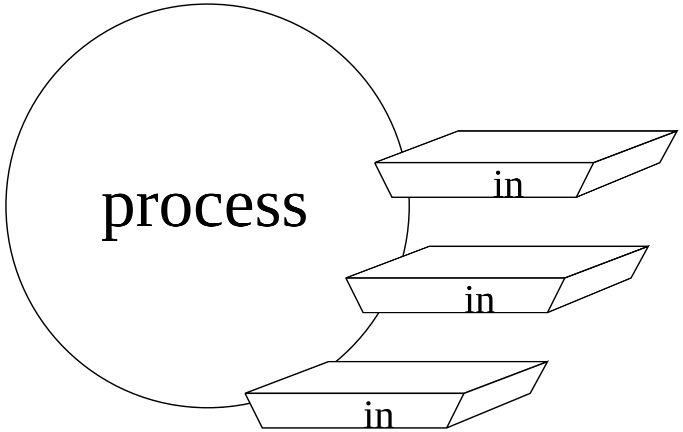
  MPI_WAIT(handle, status, ierror)

  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.

Testing Multiple Non-Blocking Communications



Exercise: Rotating information around a ring

- A set of processes are arranged in a ring.
- Each process stores its rank in `MPI_COMM_WORLD` in an integer.
- Each process passes this on to its neighbour on the right.
- Keep passing it until it’s back where it started.
- Each processor calculates the sum of the values.

Derived Datatypes

MPI Datatypes

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

Derived Datatypes - *Type Maps*

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:

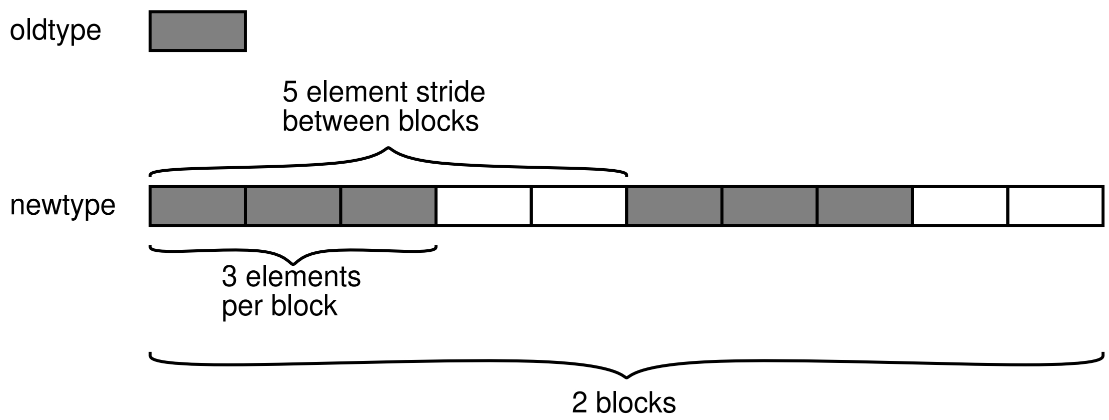
```
int MPI_Type_contiguous (int count, MPI_Datatype oldtype,
                         MPI_Datatype *newtype)
```

- Fortran:

```
MPI_TYPE_CONTIGUOUS (COUNT, OLDTYPE,
                     NEWTYPE)

INTEGER COUNT, OLDTYPE, NEWTYPE
```

Vector Datatype Example



- `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 Datatype

- C:

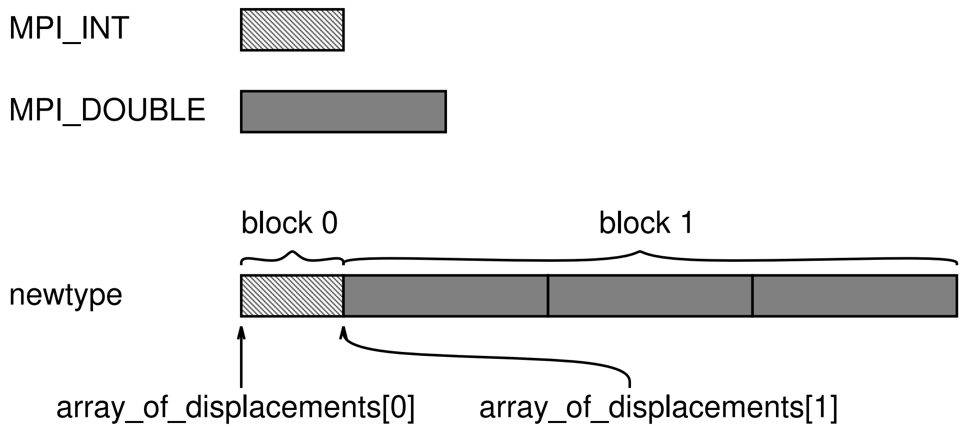
```
MPI_Type_extent (MPI_Datatype datatype,
                 int* extent)
```

- Fortran:

```
MPI_TYPE_EXTENT( DATATYPE, EXTENT, IERROR)

INTEGER DATATYPE, EXTENT, 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

Exercise: Derived Datatypes

- Modify the passing-around-a-ring exercise.
- Calculate two separate sums:
  - rank integer sum, as before
  - rank floating point sum
- Use a *struct* datatype for this.

Virtual Topologies

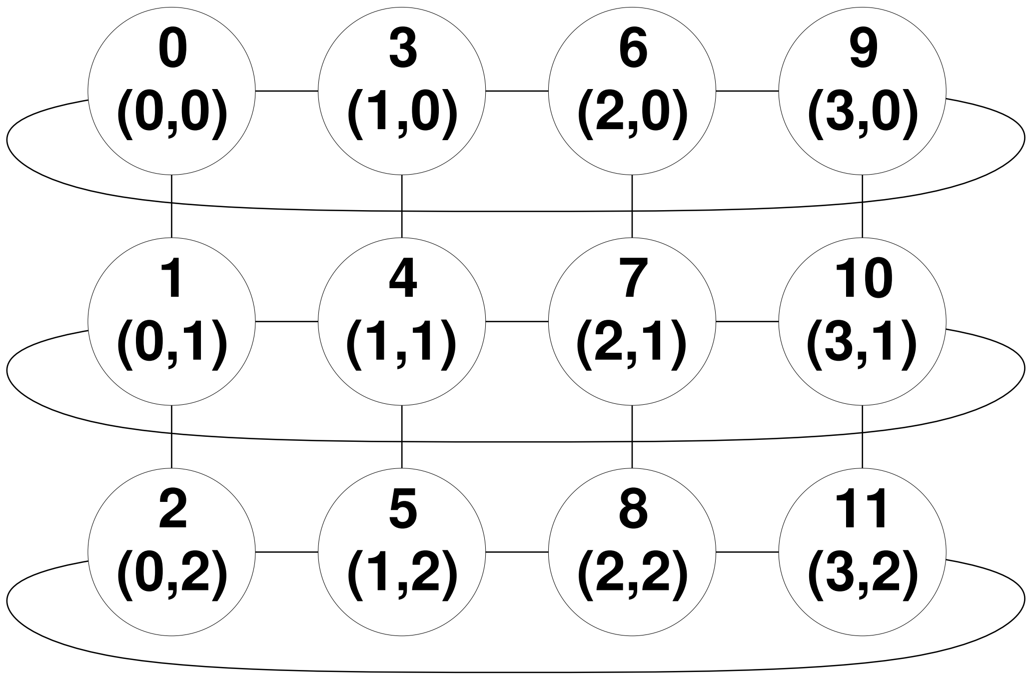
Virtual Topologies

- 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 Torus



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
```

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
```

Cartesian Partitioning

- Cut a grid up into `slices'.
- A new communicator is produced for each slice.
- Each slice can then perform its own collective communications.
- `MPI_Cart_sub` and `MPI_CART_SUB` generate new communicators for the slices.

Cartesian Partitioning with MPI_CART_SUB

- C:

```
int MPI_Cart_sub (MPI_Comm comm, int *remain_dims,
                  MPI_Comm *newcomm)
```

- Fortran:

```
MPI_CART_SUB (COMM, REMAIN_DIMS, NEWCOMM,
              IERROR)
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)
```

Exercise

- Rewrite the exercise passing numbers round the ring using a one-dimensional ring topology.
- Rewrite the exercise in two dimensions, as a torus. Each row of the torus should compute its own separate result.

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 Communication

- 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

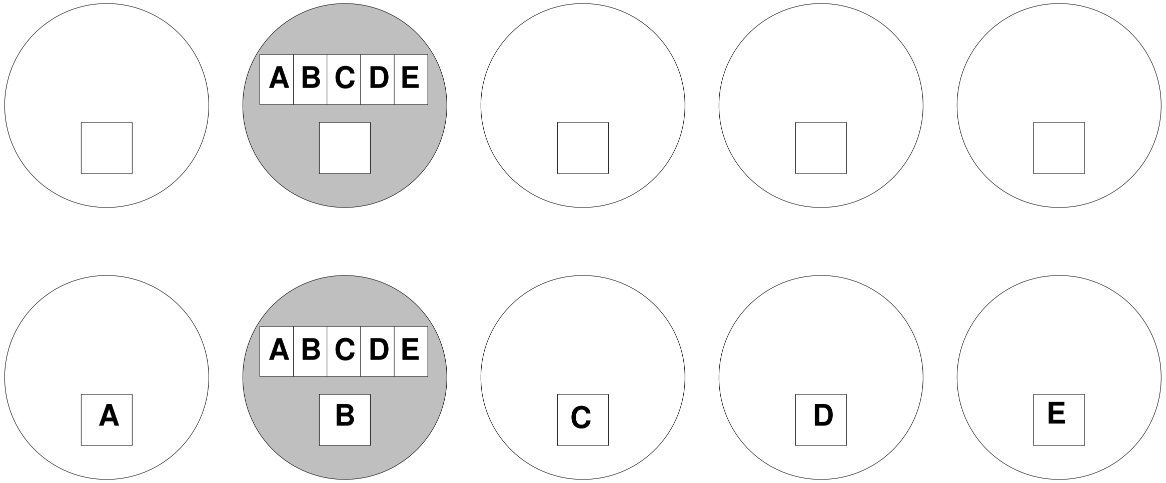
- C:

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

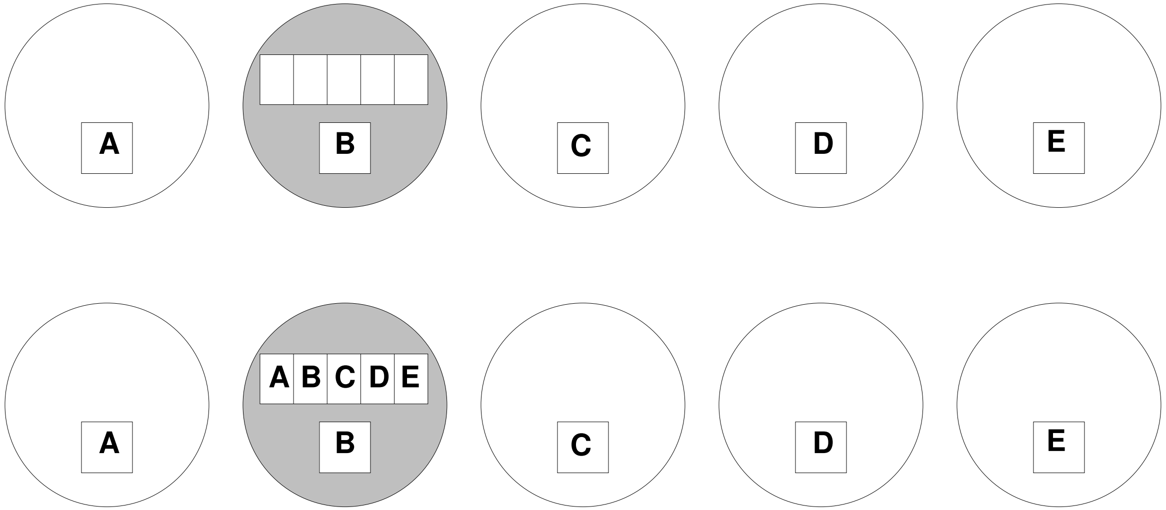
- Fortran:

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

Scatter



Gather



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

Example of Global Reduction

Integer global sum

- C:

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

- Fortran:

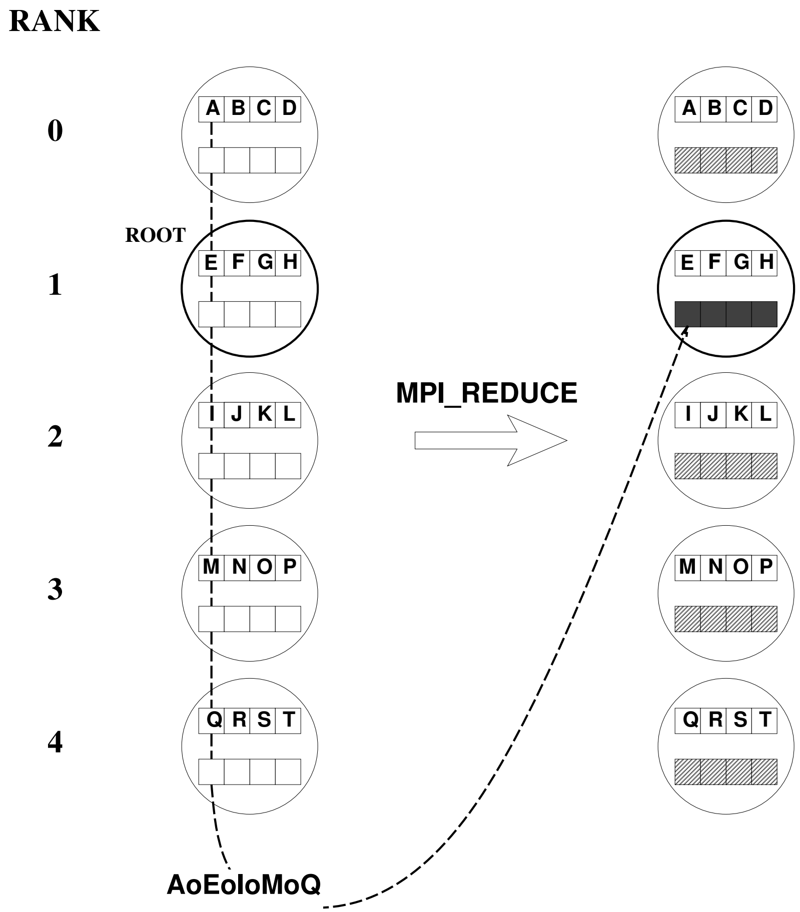
```
CALL MPI_REDUCE( x, result, 1, MPI_INTEGER,
           MPI_SUM, 0, MPI_COMM_WORLD, IERROR)
```

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

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 andlocation

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

User-Defined Reduction Operators

- Reducing using an arbitrary operator, ■
- C - function of type `MPI_User_function`:

```
void my_operator (      void *invec, void *inoutvec,int *len,
                        MPI_Datatype *datatype)
```

- Fortran - function of type

```
FUNCTION MY_OPERATOR (INVEC(*),INOUTVEC(*), LEN,
                                 DATATYPE)
    <type> INVEC(LEN), INOUTVEC(LEN)
    INTEGER LEN, DATATYPE
```

Reduction Operator Functions

- Operator function for ■ must act as:

  for (i = 1 to len)
      inoutvec(i) = inoutvec(i) ■ invec(i)

- Operator ■ need not commute

Registering a User-Defined Reduction Operator

- Operator handles have type `MPI_Op` or `INTEGER`
- C:

```
int MPI_Op_create (MPI_User_function *function,
                   int commute, MPI_Op *op)
```

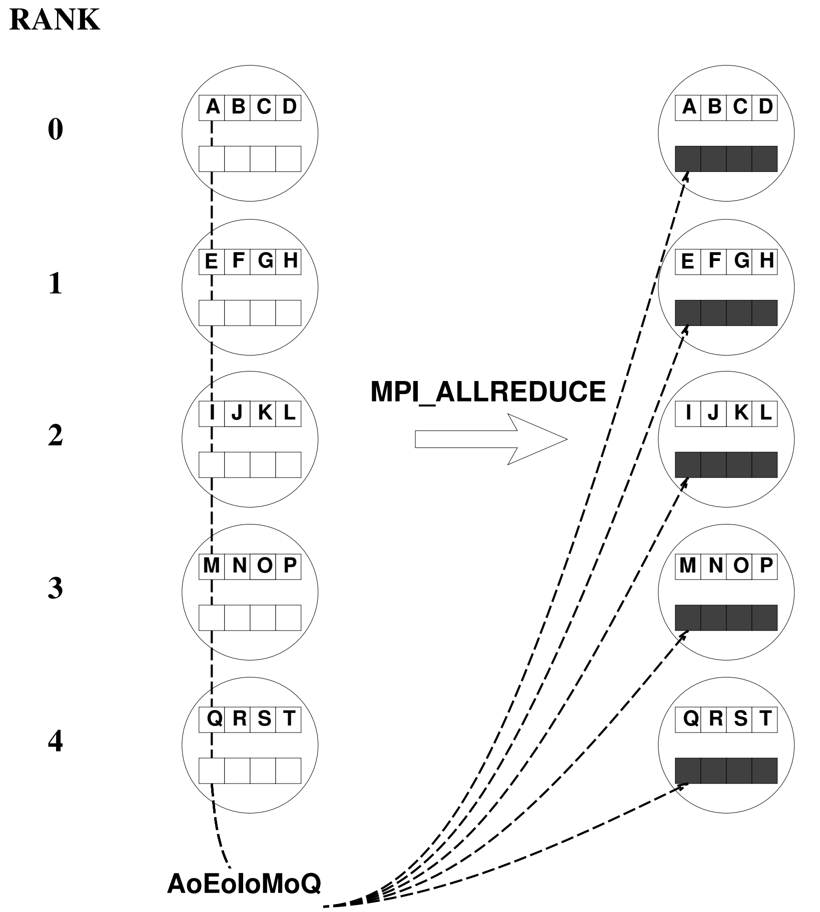
- Fortran:

```
MPI_OP_CREATE (FUNC, COMMUTE, OP, IERROR)
    EXTERNAL FUNC
    LOGICAL COMMUTE
    INTEGER OP, IERROR
```

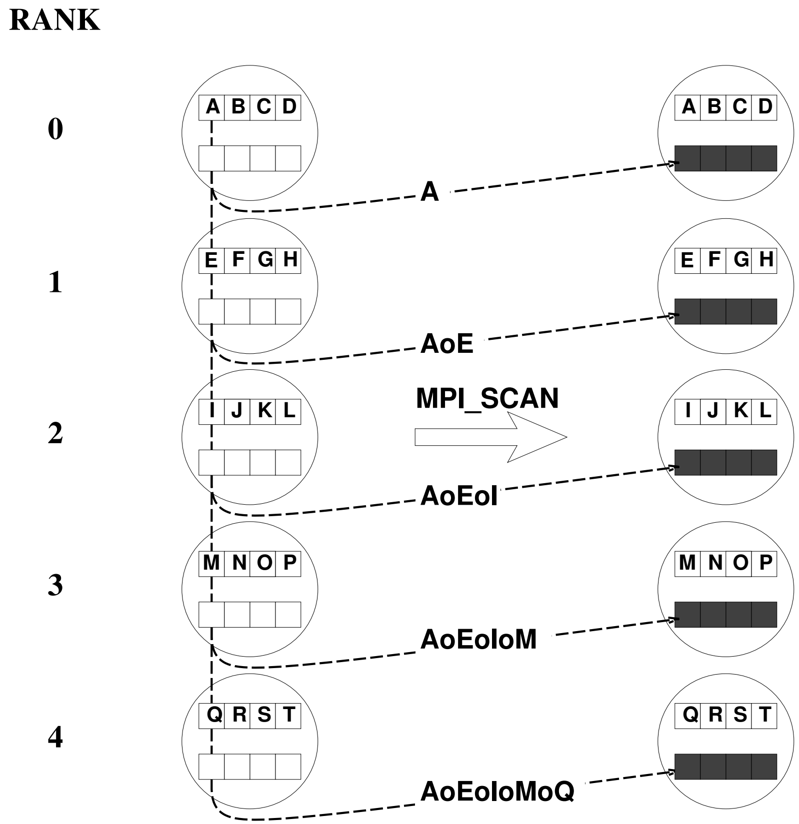
Variants of MPI_REDUCE

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

MPI_ALLREDUCE



MPI_SCAN



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

Exercise

- Rewrite the pass-around-the-ring program to use MPI global reduction to perform its global sums.
- Then rewrite it so that each process computes a partial sum.
- Then rewrite this so that each process prints out its partial result, in the correct order (process 0, then process 1, etc.).

Case Study: Foxes and Rabbits

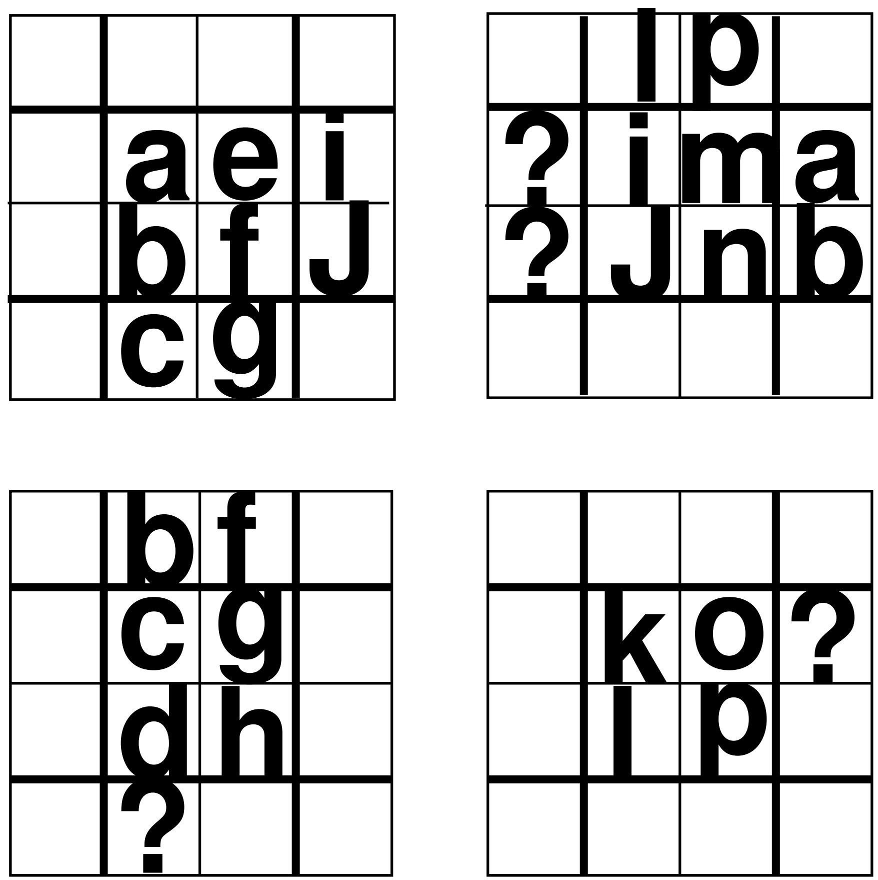
Foxes and rabbits

- Review some of the major MPI constructs.
- Look at some issues relevant for rewriting a sequential code in MPI.
- Gain confidence about writing realistic MPI programs.

Data Representation

- `Fox(i,j)` or `Fox[i][j]` is the number of foxes on the `i,j`-stretch of land.
- `Rabbit(i,j)` or `Rabbit[i][j]` is the number of rabbits on the `i,j`-stretch of land.
- Boundary conditions are periodic in the North-South direction with period `WE_Size` and periodic in the East-West direction with period `NS_Size`.

Halo Data



MPI Concepts Reviewed

- Cartesian Topologies (1-D and 2-D)
- Geometric Data Decomposition (1-D and 2-D)
- Point-to-Point Communications (Data Shifts)
- Collective Communications (Global Sums)

ECO Program

- SetMesh:
  - Virtual topology
- SetLand:
  - Set problem parameters
  - Set initial animal populations
  - Record the mapping between local and global indices for local data
- SetComm:
  - Define MPI data types to shift strided vectors across nearest neighbour processes
  - Precompute the ranks of nearest neighbour processes.

ECO Program (cont'd)

- Evolve:
  - Compute populations of foxes and rabbits from the populations of the previous year.
- FillBorder:
  - Shift halo data between nearest neighbour processes in all four cardinal directions.
- GetPopulation:
  - Sum the all the local population counts for a single specie.

EPCC's MPI implementation

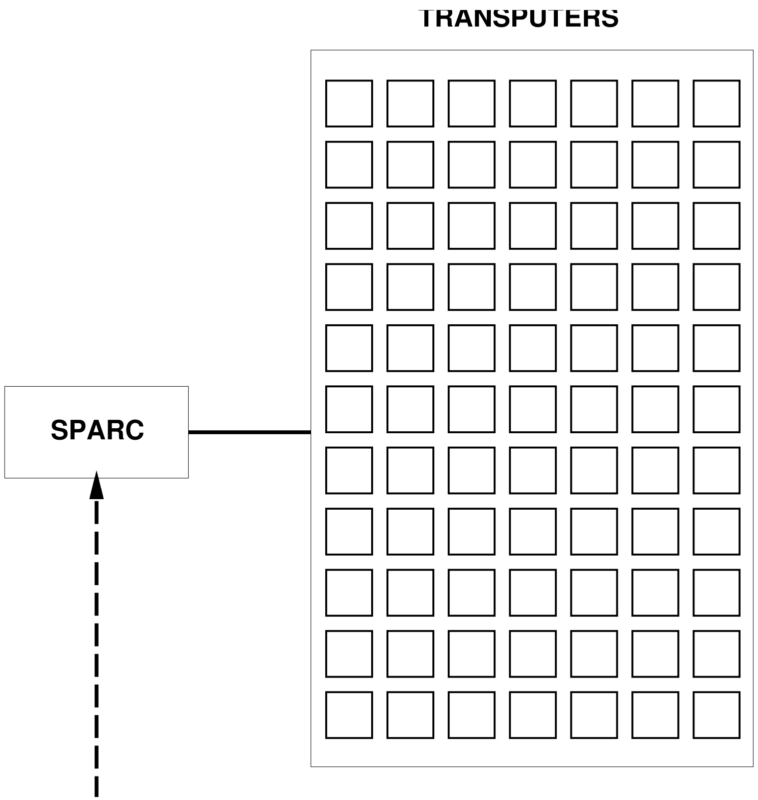
EPCC's MPI Implementation for CHIMP V2.1

- Can be used on all systems where CHIMP V2.1 runs:
  - Silicon Graphics running IRIX 4 or 5
  - Sun SPARC workstations running SunOS 4.1.x or Solaris 2.x
  - DEC Alpha running OSF/1
  - Meiko Computing Surface 1 - transputer, i860 and SPARC nodes
  - Meiko Computing Surface 2

How to obtain a copy of EPCC's MPI

- Available by anonymous ftp.
  - host: `ftp.epcc.ed.ac.uk`
  - directory: `pub/chimp/release`
  - file: `chimp.tar.Z`

The SSP Machine



- rlogin ssp

Finding Resources

csusers -a

```
user@ssp$ csusers -a
Resource     User        Attached
d2a          AVAILABLE
d2b          AVAILABLE
d2c          AVAILABLE
...

Class        Members
d68          d68a d68b
d51          d51a d51b d51c d51d d51e d51f
...
```

Requesting Resources

csattach

```
user@ssp$ csattach d17
Request for d17 granted.
d17a: attaching to 17 x T800
Total remaining allocation: 3294:12:21 processor hours
Timeout on this connection limited to: 193:46:36 hours
user@ssp$
```

Releasing Resources

csdetach

```
user@ssp$ csdetach
d17: detached
Connect time = 0:01:15; processor time = 0:21:15
Total remaining allocation: 3293:51:06 processor hours
user@ssp$
```

Initialising your environment

- `/home/chimp/chimpv2.1/bin/mpiinst`
- `logout`
- Login again.
- `echo $MPIHOME` - this should contain a valid pathname

Compiling MPI programs

- C

  ```
  mpicc -mpiarch t800 -o simple simple.c
  ```

- Fortran

  ```
  mpif77 -mpiarch t800 -o simple simple.F
  ```

Running MPI programs

- `mpirun <configuration file>`
- `-d` option for more information.
- Configuration file specifies which processes are to be run on which processors.

Configuration file 1

```
# Run one instance of 'simple' on a t800 processor
(simple): type=t800
```

Configuration file 2

```
# Four instances of 'simple' each on a t800 processor
4 (simple): type=t800
```

Configuration file 3

```
# N instances of 'simple' each on a t800 processor
$1 (simple): type=t800
```