Parallel Algorithms on a cluster of PCs

Ian Bush
Computational Science & Engineering Department
Daresbury Laboratory
I.J.Bush@dl.ac.uk
(With thanks to Lorna Smith and Mark Bull at EPCC)

Overview

- This lecture will cover
  - **General Message passing concepts**
    - *Message passing model*
    - *SPMD*
    - *communication modes*
    - *collective communications*

- An extremely brief look at what MPI actually looks like

Message Passing Model

- The message passing model is based on the notion of processes
  - **can think of a process as an instance of a running program, together with the program's data**
- In the message passing model, parallelism is achieved by having many processes co-operate on the same task
- Each process has access only to its own data
- Processes communicate with each other by sending and receiving messages

Process Communication

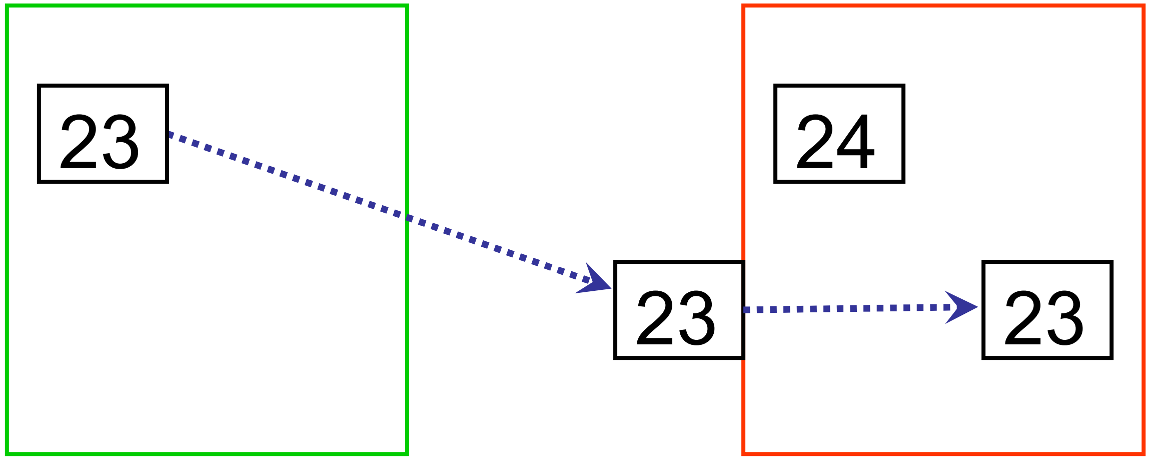
Process 1 | Process 2

Program

```
a=23
Send(2,a)
```

```
Recv(1,b)
a=b+1
```

Data



SPMD

- Most message passing programs use the Single-Program-Multiple-Data (SPMD) model
- All processes run the same program
- Each process has a separate copy of the data
- To make this useful, each process has a unique identifier
- Processes can follow different control paths through the program, depending on their process ID
- Usually run one process per processor

Messages

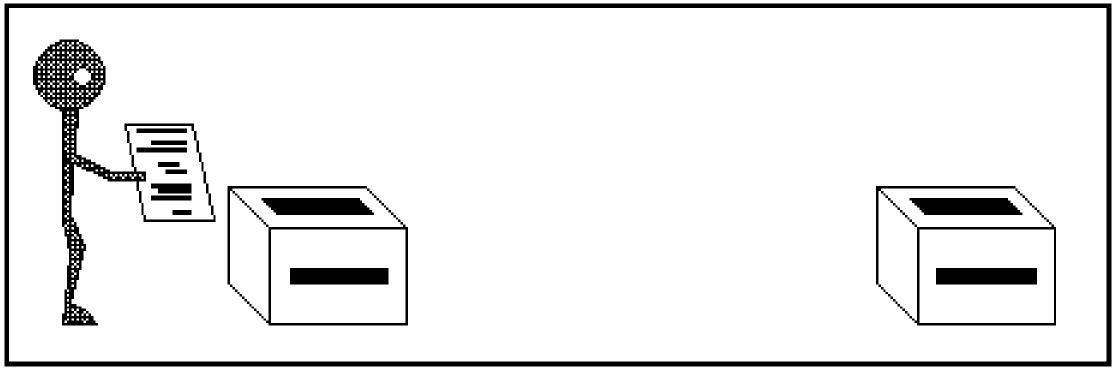
- A message transfers a number of data items of a certain type from the memory of one process to the memory of another process
- A message typically contains
  - **the ID of the sending processor**
  - **the ID of the receiving processor**
  - **the type of the data items**
  - **the number of data items**
  - **the data itself**
  - **a message type identifier**

Communication modes

- Sending a message can either be synchronous or asynchronous
- A synchronous send is not completed until the message has started to be received
- An asynchronous send completes as soon as the message has gone
- Receives are usually synchronous - the receiving process must wait until the message arrives

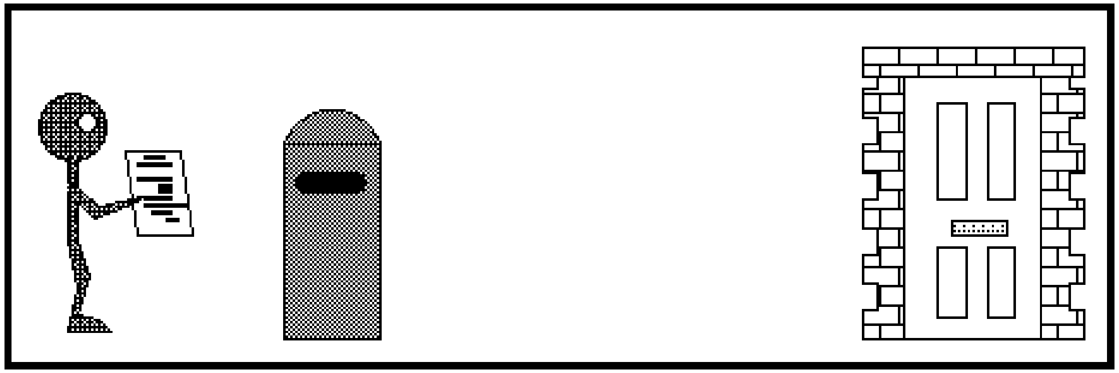
Synchronous or Blocking send

- Analogy with faxing a letter.
- Know when letter has started to be received.



Asynchronous or Non-Blocking send

- Analogy with posting a letter.
- Only know when letter has been posted, not when it has been received.



Point-to-Point Communications

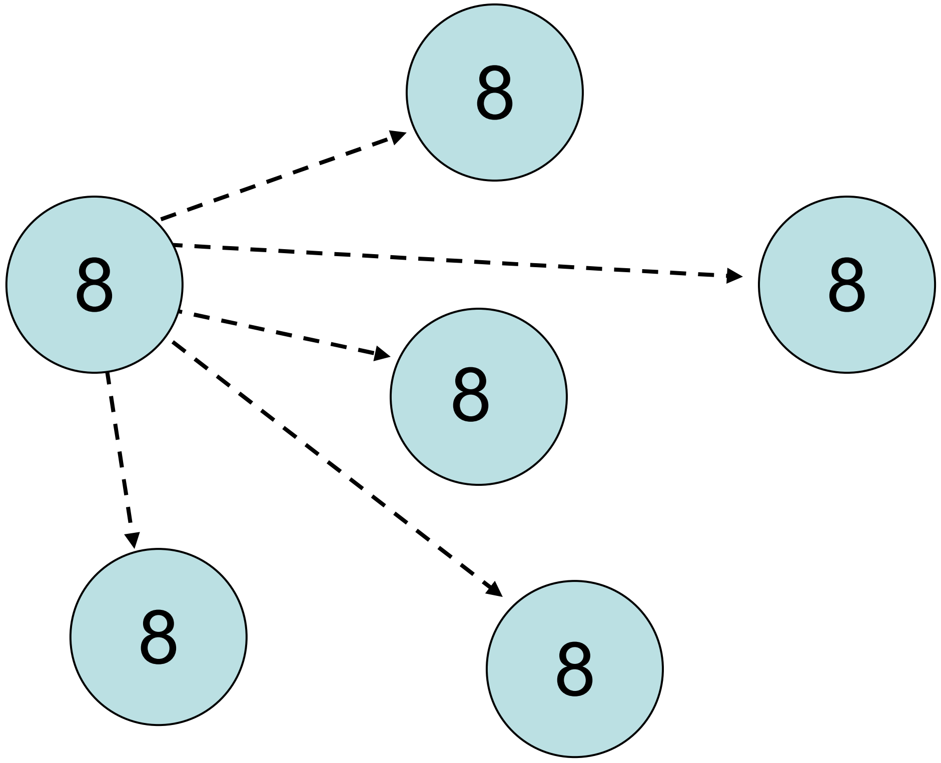
- We have considered two processes
  - **one sender**
  - **one receiver**
- This is called point-to-point communication
  - **simplest form of message passing**
  - **relies on matching send and receive**
- Close analogy to sending personal emails

Collective Communications

- A simple message communicates between two processes
- There are many instances where communication between groups of processes is required
- Can be built from simple messages, but often implemented separately, for efficiency

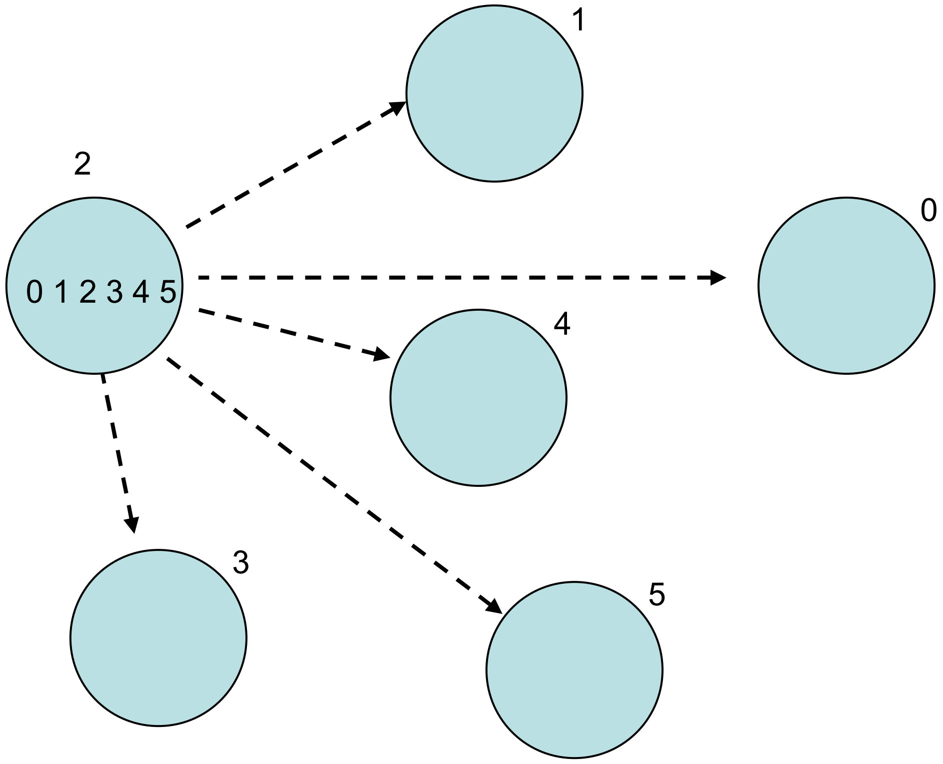
Broadcast

- From one process to all others



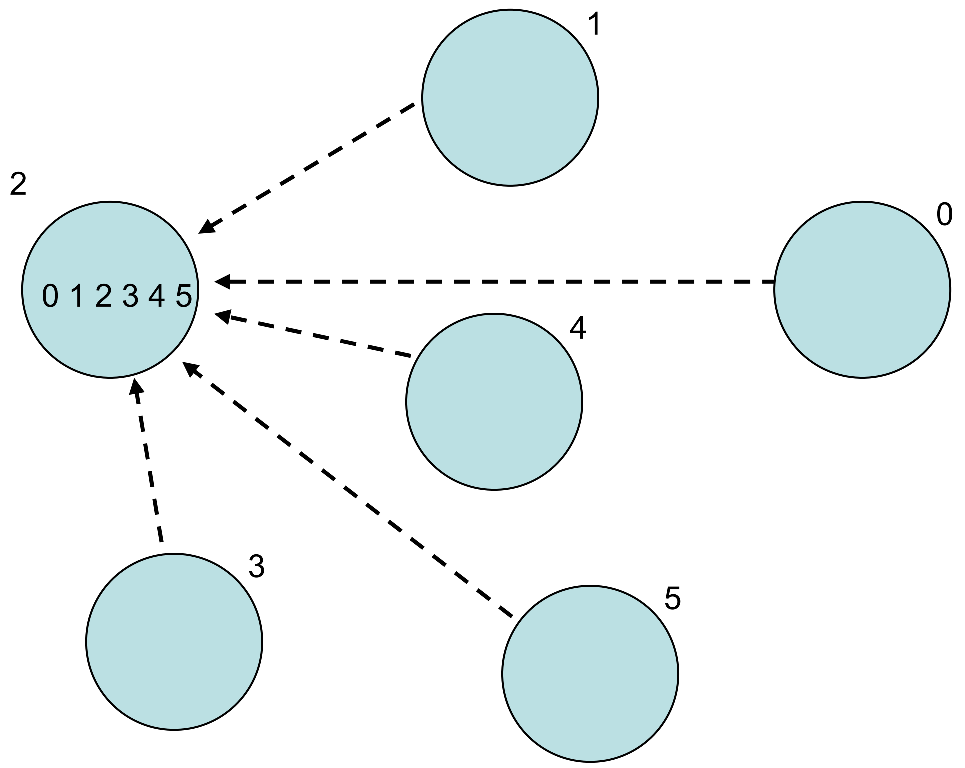
Scatter

- Information scattered to many processes



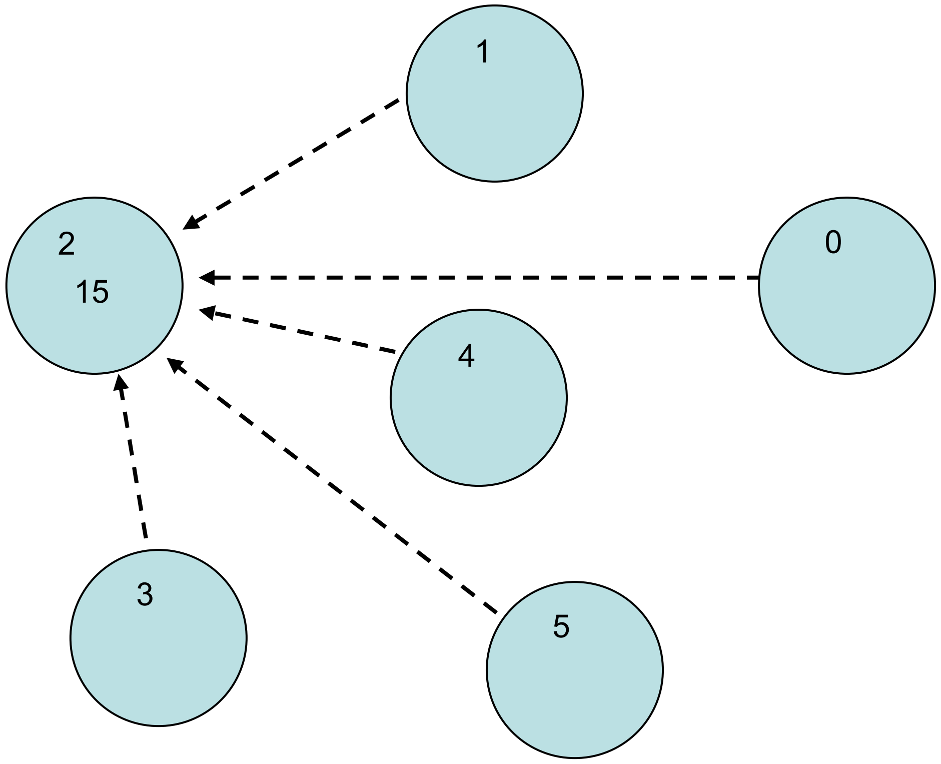
Gather

- Information gathered onto one process



Reduction

- Form a global sum, product, max,min etc.



Issues

- Sends and receives must match
  - danger of deadlock
- Possible to write very complicated programs
  - most scientific codes have a simple structure
  - often results in simple communications patterns
- Use collective communications where possible
  - may be implemented in efficient ways

Summary of Message Passing Concepts

- Messages are the *only* form of communication
  - **all communication is therefore explicit**
- Most systems use the SPMD model
  - **all processes run exactly the same code**
  - **each has a unique ID**
  - **processes can take different branches in the same codes**
- Basic form is point-to-point
  - **collective communications implement more complicated patterns that often occur in many codes**

Pros And Cons Of Message Passing

- Pros
  - **It's very general**
    - *Can express any parallel algorithm in this form, irrespective of archtiecture*
  - **As a consequence it is very portable**
- Cons
  - **It's very general**
    - *Can result in quite complex code*
      - Best to `hide' message passing as far as possible in its own layer
  - **Difficult to run resulting code on a serial processor**
    - *But not impossible, just need an MPI library*
      - But does that impair the serial performance
  - **It's two sided**
    - *What if one CPU wants data from another CPU, but that CPU has no idea when it will want it. The receiving process may have to wait a LONG time*
    - *One sided protocols exist in MPI 2, but not universally available and quite inefficient on some machines*

The Message Passing Interface (MPI)

- MPI is a *de facto* standard for message passing
  - **All vendors supply it**
  - **A number of free implementations available (e.g. MPICH, LAM) which run on Linux, FreeBSD, Windows …**
- MPI Forum founded in 1992
  - **Comprised of all major vendors and many major academic parallel computing centres around the world.**
- MPI 1.0 introduced in 1994, first implementation the next day !
- MPI 1.1 and 1.2 introduced corrections and clarifications. MPI 1.2 is the standard I would recommend
- MPI 2.0 introduced in 1997, but implementations are less common

What does MPI Include ?

- Point to Point
  - **Blocking and non-blocking**
- Collectives
- Communicators
- Process topologies
- User defined data types
- Intracommunicators
- Profiling interface
- Fortran, C, C++ interfaces
- The kitchen sink !

MPI has around 150 routines in the library.

You need to know six to get started.

Hello World in MPI

```
Program Hello
      Implicit None
      Include ´mpif.h’
      Integer :: me, numprocs, error
      Call mpi_init( error )
      Call mpi_comm_rank( mpi_comm_world, me, error )
      Call mpi_comm_size( mpi_comm_world, numprocs, error )
      Write( *, * ) ´Hello from ‘, me, ´ of ’, numprocs
      Call mpi_finalize( error )
End Program Hello
```

Compile (typically) by

```
mpif90 –o progname hello.f90
```

Run by

```
mpirun –np 4 ./progname
```

The Output

```
Hello from 1 of 4
Hello from 3 of 4
Hello from 2 of 4
Hello from 0 of 4
```

NB: You can make no assumption about at what time what part of the process will run relative to any other. They are all independent tasks. They do not need to follow process 0. They don't need to follow any process at all !

(Unless you program them to)

Hence the ordering above. Run it again and you might get a totall different ordering.

MPI – the big six (1)

- Call mpi_init(error)
  - Initialize MPI
- Call mpi_finalize(error)
  - **Close down MPI**
- Call mpi_comm_rank(mpi_comm_world, me, error)
  - Obtain my processor ID. $0 \le me < numprocs$ run on
- Call mpi_comm_size(mpi_comm_world, numprocs, error)
  - **How many processors am I running on**

MPI – The Big Six (2)

- Call mpi_send(buffer, count, type, destination, tag, &
  mpi_comm_world, error)
  - Send *count* data items of type *type* to processor *destination* using the message tag *tag*
  - Will cover *type* and *tag* in the next slides
  - BLOCKING !!!
- Call mpi_recv(buffer, count, type, source, tag, & mpi_comm_world, status, error)
  - Receive *count* data items of type *type* from processor *source* using the message tag *tag*
  - BLOCKING !!!

MPI – The Big Six (3) – The *type* Argument

Type specifies what sort of thing is being sent or received. It is simply an integer that is defined in mpif.h. You SHOULD use the symbolic forms as each MPI implementation WILL use different values. Some symbolic values are:

- MPI_CHARACTER
- MPI_INTEGER
- MPI_REAL
- MPI_DOUBLE_PRECISION
- MPI_COMPLEX
- MPI_BYTE

It's fairly obvious what these mean !

MPI – The Big Six (4) – The *tag* argument

- Say you want to send a number of different messages from one processor to another of different types, lengths, …… How does the receiving end know which message to correspond with which receive
  - **Use the *tag* argument to differentiate – simply an integer variable so for each of the sends and each of the recvs use a different number**
  - **In practice rarely useful, at least in my experience, and I just tend to set it to 10**
- It is also possible to set *tag* to mpi_any_tag at the recv end (also source to mpi_any_source) to receive the first message with any tag (and maybe any source)
  - **DO NOT DO THIS IF AT ALL AVOIDABLE !**
  - **In my experience this**
    - ***Just asks for deadlock and other, stranger, bugs***
    - ***Shows you don't totally understand what is happening***
    - ***Shows you haven't thought it through properly***

A Simple MPI Program

```
Program Simple
      Implicit None
      Include 'mpif.h'
      Integer :: numprocs, me, procs, error
      Character( Len = 8 ) :: msg = 'Wotcha!!'
      Call mpi_init( error )
      Call mpi_comm_rank( mpi_comm_world, me, error )
      Call mpi_comm_size( mpi_comm_world, numprocs, error )
      If( me == 0 ) Then
            Do procs = 0, numprocs – 1
                  Call mpi_send( msg, 8, MPI_CHARACTER, procs, 10, &
                        mpi_comm_world, error )
            End Do
      Else
            Call mpi_recv( msg, 8, MPI_CHARACTER, 0, 10, &
                  mpi_comm_world, status, error )
      End If
      Write( *, * ) msg, ' from proc ', me
      Call mpi_finalize( error )
End Program Simple
```

So Now You Can Do it All !

- Everything can be performed by these routines, all else is for convenience
- However MPI has a lot more, so I'll quickly cover a couple of useful other things
  - **Collectives**
  - **Communicators**
- But first I'll cover THE CLASSIC MPI BUG

THE CLASSIC MPI BUG

Two processes running though the same bit of code:

```
If( me == 0 ) Then
    them = 1
Else
    them = 0
End If
Call mpi_send( buffer, count, MPI_REAL, them, 10, mpi_comm_world, error )
Call mpi_recv( buffer, count, MPI_REAL, them, 10, mpi_comm_world, &
    status, error )
```

Proc 0 and proc 1 swap buffers, right ?

NO !!!!!!

THE CLASSIC MPI BUG

```
If( me == 0 ) Then
      them = 1
Else
      them = 0
End If
Call mpi_send( buffer, count, MPI_REAL, them, 10, mpi_comm_world, error )
Call mpi_recv( buffer, count, MPI_REAL, them, 10, mpi_comm_world, &
      status, error )
```

MPI_SEND IS BLOCKING and so this will DEADLOCK !

Unfortunately the MPI standard says that the programmer must assume mpi_send is blocking, but implementations can do whatever is best. So this nasty bit of code may, and will work on some machines, but not on others, or for one message size, but not others.

BEWARE !

THE CLASSIC MPI BUG

- Using the simple call we have used so far it is

```
If( me == 0 ) Then
      them = 1
      Call mpi_send( buffer, count, MPI_REAL, them, 10, mpi_comm_world, &       error )
      Call mpi_recv( buffer, count, MPI_REAL, them, 10, mpi_comm_world, &
                status, error )
Else
      them = 0
      Call mpi_recv( buffer, count, MPI_REAL, them, 10, mpi_comm_world, &
                status, error )
      Call mpi_send( buffer, count, MPI_REAL, them, 10, mpi_comm_world, &       error )
End If
```

- Alternatives are mpi_sendrecv, or non-blocking communications with mpi_isend and mpi_irecv, the latter being the most general solution. However not really enough time to go into unless someone is interested.

MPI Collectives

- As said in the general concepts part of the talk collectives are also very useful
- One simple one is the broadcast mpi_bcast
  - Call mpi_bcast(buffer, count, type, root, mpi_comm_world, error)
  - This broadcasts the data in buffer held on processor root to all the other processors
  - We can use this to simplify our simple MPI program

A Simple MPI Program Using MPI_BCAST

```
Program Simple
    Implicit None
    Include 'mpif.h'
    Integer :: numprocs, me, procs, error
    Character( Len = 8 ) :: msg
    Call mpi_init( error )
    Call mpi_comm_rank( mpi_comm_world, me, error )
    Call mpi_comm_size( mpi_comm_world, numprocs, error )
    If( me == 0 ) Then
          Read( *, * ) msg
    End If
    Call mpi_bcast( msg, 8, MPI_CHARACTER, 0, mpi_comm_world, &  error )
    Write( *, * ) msg, ' from proc ', me
    Call mpi_finalize( error )
End Program Simple
```

Other Useful Collectives

- The most commonly used is MPI_ALLREDUCE
  - **Do a reduction across all processors**
  - **Operation can be from a large selection e.g.**
    - *Max*
    - *Min*
    - *Sum*
    - *Product*
    - *Logical and*
    - *Logical Or*
    - *........*
- Also MPI_GATHER, MPI_SCATTER, MPI_ALLTOALL …….

Communicators (or what is this mpi_comm_world thing anyway ?)

- In MPI ALL communications occur within what is known as a communicator
- A communicator is essentially a set of processors, not necessarily all the processors in the job
- MPI gives you for free a number of communicators, of which by far the most important is mpi_comm_world
- However you are free to define your own using a number of routines. MPI_COMM_SPLIT, which allows you to split an exisiting communicator into a number of subsets, is both the most powerful and most useful
- This is a very useful concept

Using Communicators

- So to use comunicators
  - **Create one of your own**
  - **Wherever we have used mpi_comm_world in the past now use your own, call it my_comm**

- So we can say
  - **Call mpi_comm_rank(mpi_comm_world, me, error)**
    - *Give me my rank in the communicator mpi_comm_world, i.e. my global rank*
  - **Call mpi_comm_rank(my_comm, me, error)**
    - *Give me my rank in the communicator I have just created*
  - **Call mpi_comm_size(my_comm, numprocs, error)**
    - *How many procs are in the communicator I just created*
  - **Call mpi_bcast(stuff, count, MPI_REAL, 0, my_comm, error)**
    - *Broadcast stuff across all the communicators*

Works for ALL message passing, can even split your own communicators !

Why is this useful ?

- Say we are running on 8 processors and have 4 tasks that can be performed independently
  - **Natural to give each of the tasks to two of the processors**
  - **May be necessary when, e.g., each of the tasks is too big for one processor**
- Could do this all in mpi_comm_world but would have all sort of if conditions to decide which task this CPU belongs to, and code would become very messy
- Much neater to split into 4 communicators
  - **Typically once the new communicators are created all processors execute the same code, the communicators taking care of the communications**
- Solid State people – Say we have 4 k points and are running on 8 processors … think about it.

More About MPI

- MPI is ubiquitous in parallel computing
- There is an awful lot of information out there, including both MPI tutorials and applications of MPI to specific problems
- I would suggest that if you want to learn more simply google for MPI. The earliest hits include both the standard document and a number of good tutorials (I like the stuff at ORNL myself, but choose one that suits you)
- However it really is a practical art, and the only true way is to get going is to practice it yourself.