Durham University

A brief introduction to parallel programming on a supercomputer

Instructors:
Dmitry Nikolaenko (RSEng)
Sean Baccas (RSEng)

Assistant: Zeynep Aki (RSEng)

Advanced Research Computing
Durham University

June 27, 2023

https://www.dur.ac.uk/arc/

Course Outline

Basics of parallel programming with OpenMP and MPI using Durham University's supercomputer, Hamilton.

Aims of the course:

- Introduction to parallel programming and to both shared- and distributed-memory model;
- Learn how to make a serial C code multi-threaded by adding `pragma` directives;
- Learn about synchronisation, critical region and atomic directive;
- Learn how to use MPI commands to pass messages;
- Learn about collective and combined parallel communications;
- Be familiarised with data handling and higher functions of MPI;
- Some examples of the best parallel use of a supercomputer.

Course Timing

- 10:00-10:10 – Get-to-know round
- 10:10-10:20 – Brief introduction to HPC and parallel programming models
- 10:10-10:40 – **1. Basics of OpenMP**
  - 10:40-11:00 – Practical session OpenMP
- 11:00-11:10 – “coffee break”

COFFEE BREAK

- 11:10-11:40 – **2. Basics of MPI: Point-to-point communications**
  - 11:40-12:00 – Practical session 2: “Ping pong!”
- 12:00-12:10 - “coffee break”

COFFEE BREAK

- 12:15-12:40 – **3. Basics of MPI: Collective communications**
  - 12:40-13:00 – Practical session 3: “Collective communication”
- Conclusion: advanced MPI

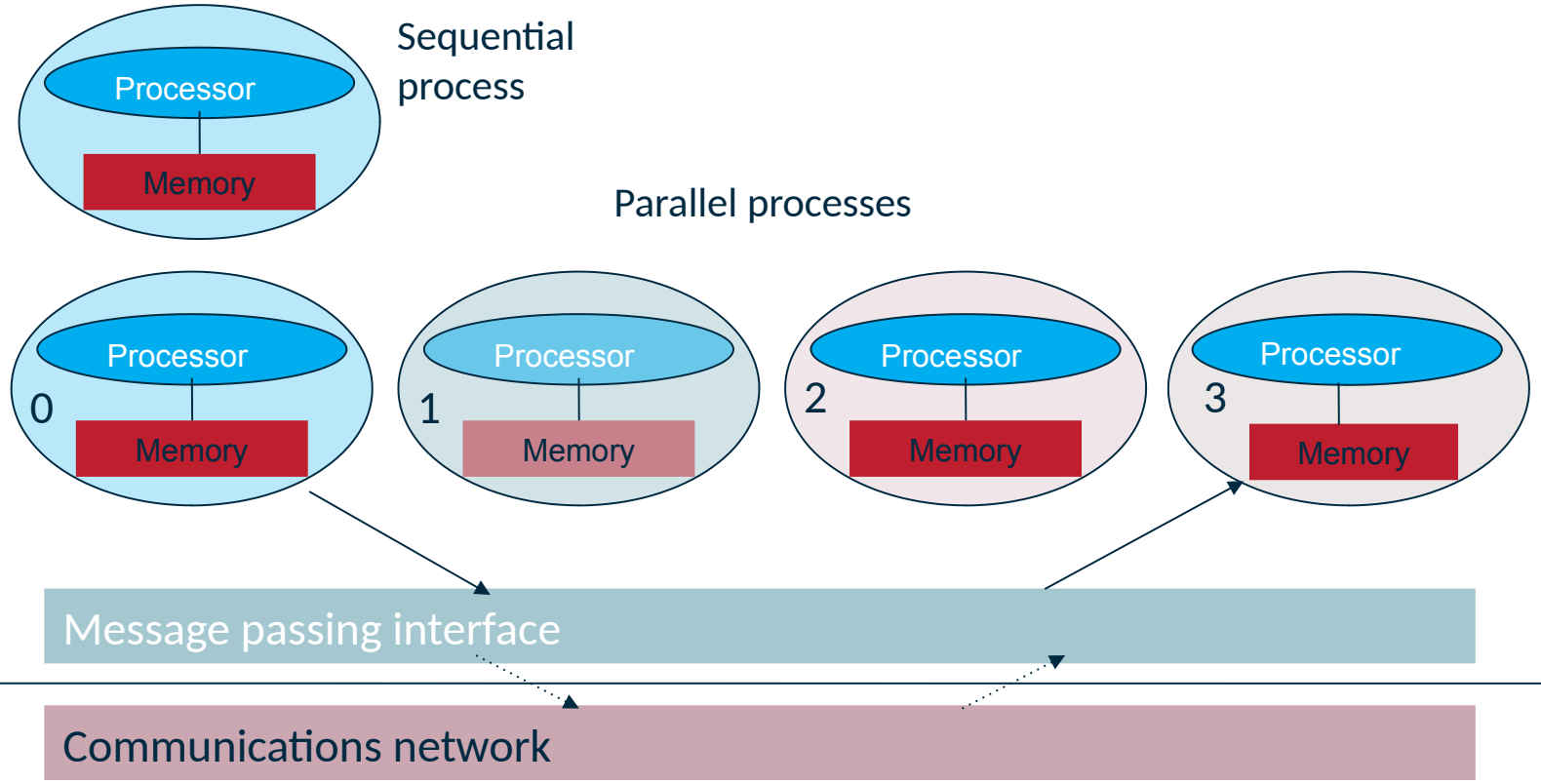
Durham University

Preliminaries

- Where are you from in the University and what is your research?
- What is your experience of parallel programming and high performance computing?
- How do you plan to use parallel programming in your research?
  - Are you currently trying to parallellise your code?
  - Are there any difficulties in this development?

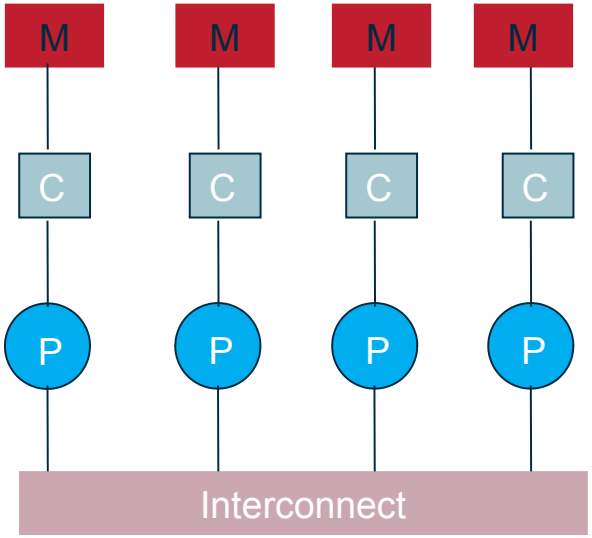
1. Brief intro to HPC and parallel programming models

1.1 Message-Passing Paradigm

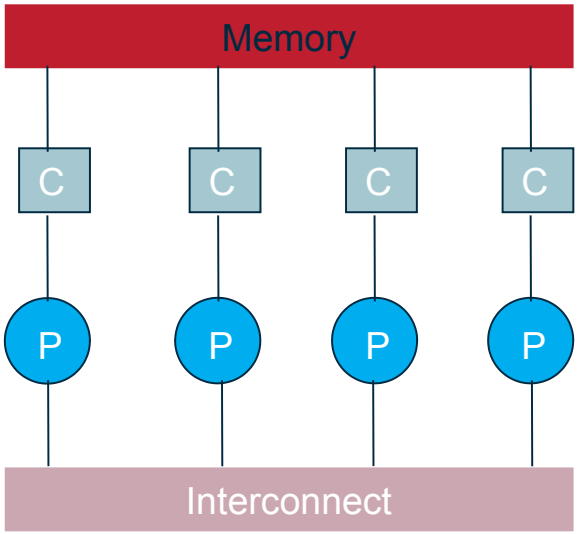


1. Brief intro to HPC and parallel programming models

1.2 Distributed memory and shared memory systems



Distributed memory system e.g. Beowulf cluster. Architecture matches message passing paradigm.



Shared-memory system.
e.g. multiprocessor desktop PCs.
Can use interconnect + memory as a communications network
(the basis of mixed-mode parallelism)

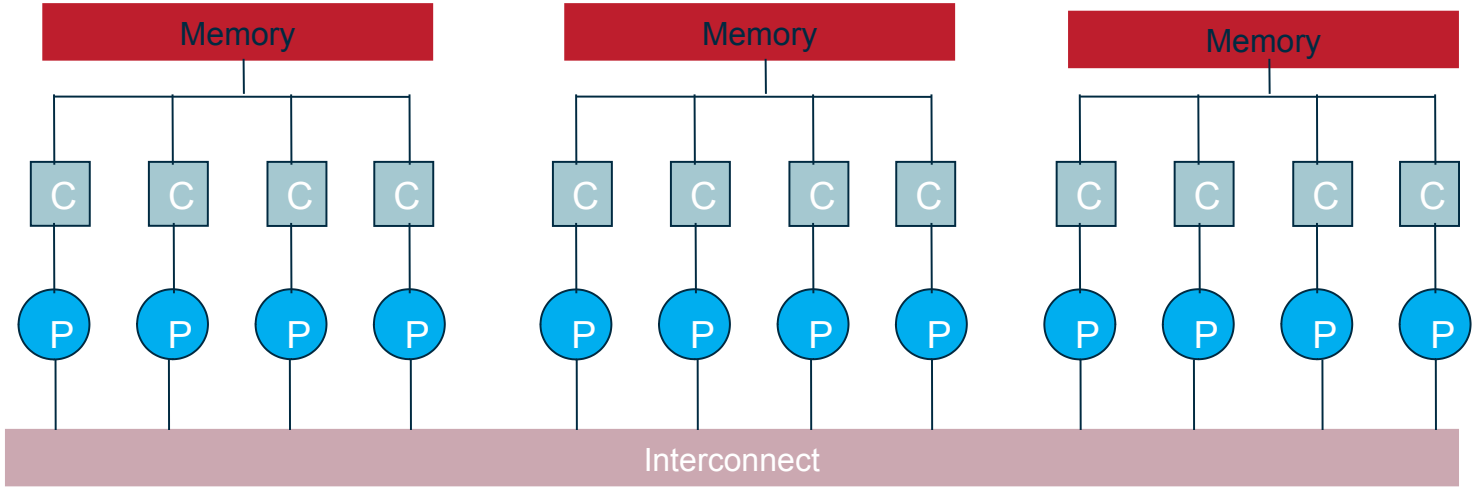
P Processor

C Cache

Durham University

1. Brief intro to HPC and parallel programming models

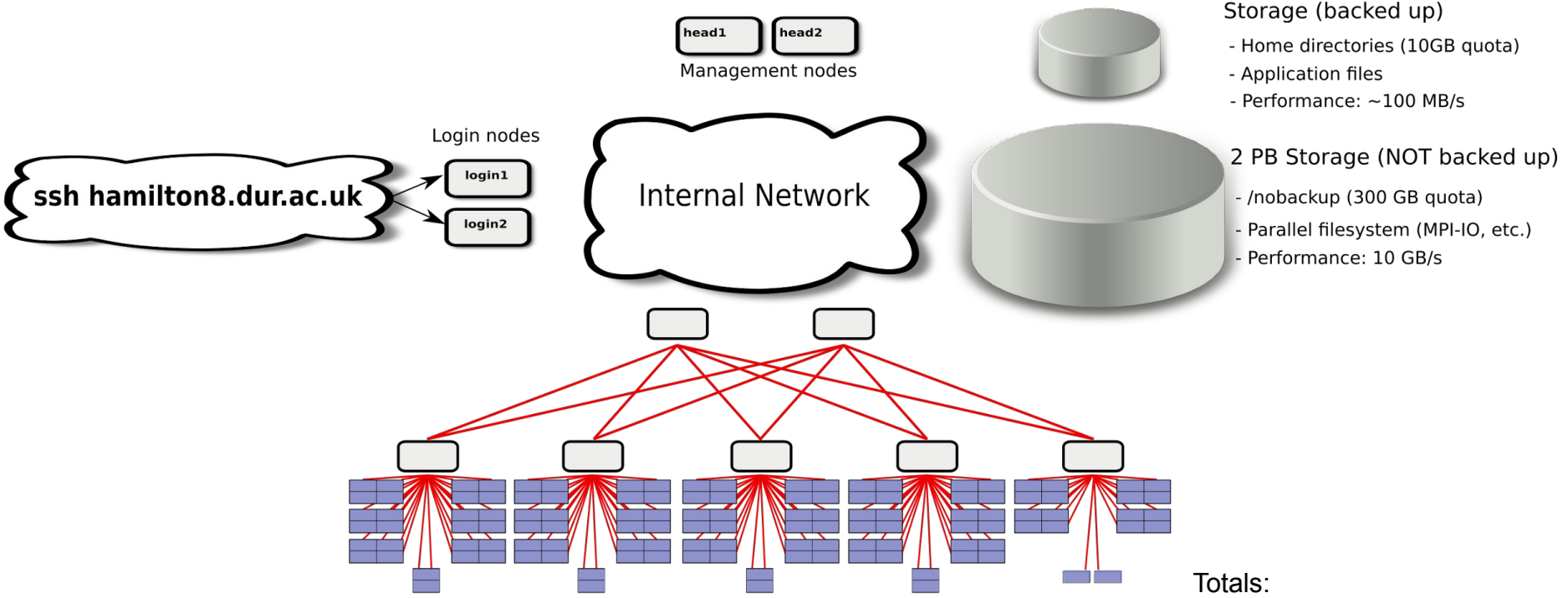
1.3 Shared memory clusters



Will use both memory/interconnect to communicate between processes.
Commonly now found shared memory clusters (e.g., Hamilton!)

1. Brief intro to HPC and parallel programming models

1.4 Machine architecture: Hamilton



Totals:
- 122 compute nodes
- 15,616 cores, 34T RAM
- 2 PB shared storage

1. Brief intro to HPC and parallel programming models

1.5 HPC terminology

- Nodes, sockets, cores, threads, processes per core
  - You can run multiple processes and threads per core
  - **MPI** (Message Passing Interface) and **OpenMP** (Open Multi-Processing) are two popular interfaces to describe parallelism. Such interfaces are commonly implemented in standard high-level language such as FORTRAN/C/C++
  - **OpenMP** provides shared-memory model and describes thread parallelism within a process (with common address space). It is realised using compiler directives to facilitate the parallelism
  - **MPI** provides distributed-memory model and describes parallelism between processes (with separate address spaces). It is implemented with calls to a parallel library

1. Brief intro to HPC and parallel programming models

1.5 HPC terminology (cont.)

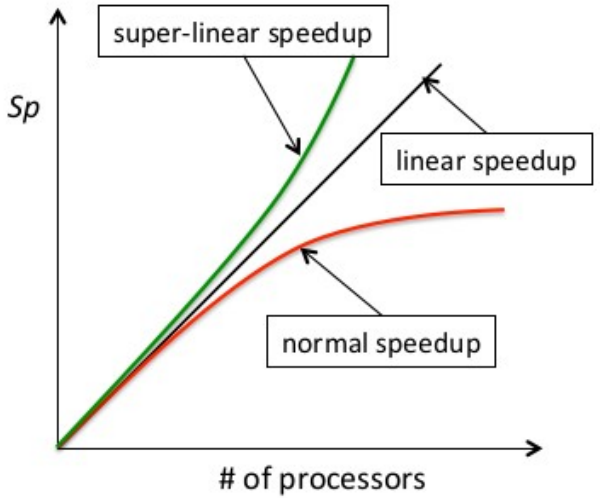
- To characterise performance of computing, processor speed is measured in *floating point operations per second* (*FLOPS*, *MFLOPS*, *GFLOPS*, etc.)
  - There are two speeds: '*peak*' – the best in theory; and '*sustained*' - on a benchmark or relevant user code. The latter can be anything between ~0% and ~80% of 'peak' speed
  - For example, a compute node on Hamilton8 has a theoretical *peak speed:* 4096 GFLOPS. HPL benchmark shows 77% efficiency
- To characterize performance of data transfer, intranode (between RAM and core) or internode (between nodes):
  - *Bandwidth* is the rate at which data can be transferred (*the higher the better)*, from *KB/s* to *GB/s*
  - *Latency* is the start-up time for data transfer (*the lower the better)*, from *ns* for L1 cache (a few clock cycles) to *ms* for Ethernet networks

1. Brief intro to HPC and parallel programming models

1.5 HPC terminology (cont.)

- Speedup:
  - *p* = # processes
  - *Ts* = execution time of the parallel algorithm on a single algorithm on a single process
  - *Tp* = execution time of the parallel algorithm on *p* processes
  - *Amdahl's law* expresses that the potential speedu is limited by the sequential part of the program

$$S_p = \frac{T_s}{T_p}$$



- Parallel efficiency:

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}$$

- Scalability:
  - Strong scaling (problem size is fixed), ideally time taken reduces in direct proportion to number of processes used
  - Weak scaling (problem size scales with # processes), ideally time taken is constant and problem scales directly with number of processes used
  - usually limited by communications, latency, idling / *load balancing* (static, dynamic)
  - Good load-balancing and efficient communication can clearly all be ruined by poor process placement!

2. Basics of OpenMP

- OpenMP allows independent units of work to be done by each of the processors on your system
- The aim is to write serial code, that can be easily parallelised with *pragma* directives
- We call this *shared memory parallelism* since all the processor threads on one system have access to the same memory block
- No need for communication like with MPI as we can read values from a common register
- We indicate to the compiler that we wish to parallelise certain sections by writing *pragma omp parallel.* Anything in the subsequent code block is run on all available threads.

```
10 #pragma omp parallel{
11
12 //code in here is run by all threads
13
14 }
```

2. Basics of OpenMP

- OpenMP allows independent units of work to be done by each of the processors on your system
- The aim is to write serial code, that can be easily parallelised with *pragma* directives
- We call this *shared memory parallelism* since all the processor threads on one system have access to the same memory block
- No need for communication like with MPI as we can read values from a common register
- In C, all we need is a single include:

```
#include <omp.h>
```

2. Basics of OpenMP

Multithreading isn't magic! We have numerous problems, such as:

- Expensive to spawn new threads
- Can be difficult to share work evenly across threads
- Run into problems like race conditions if not careful
- Much easier to run into segmentation faults, if we don't take care around critical tasks such as writing to files or particular memory addresses

2. Basics of OpenMP

2.1 Pragma directives

- We indicate to the compiler that we wish to parallelise certain sections by writing *pragma omp parallel.* Anything in the subsequent code block is run on all available threads.

```
7   int main(){
8     #pragma omp parallel num_threads(4)
9     {
10      //note that the code block begins on the next line
11      //code in here is run by all threads!
12    }
13  }
```

- Within a parallel section, we can also indicate that only one thread should perform some code, by stating *pragma omp single*

- Within a parallel section, we can also indicate that only the master thread should perform some code, by stating *pragma omp master*

- You can set the number of threads to be used by stating ***#pragma omp parallel num_threads(n)***, or by setting environment variable OMP_NUM_THREADS

```
~$
~$ export OMP_NUM_THREADS=8
```

2. Basics of OpenMP

2.2 Parallel regions

- We can use ***#pragma omp single*** to force only one thread to run a certain section.
- OpenMP doesn't give any guarantee as to which one this will be, but using ***#pragma omp master*** forces the block to use the master thread (id 0)

```
7   int main(){
8     #pragma omp parallel num_threads(4)
9     {
10      //note that the code block begins on the next line
11      //code in here is run by all threads!
12      foo();
13
14      #pragma omp single
15      bar();
16    }
17  }
18
19  void foo(){
20    printf("inside function foo, using thread number %d\n", omp_get_thread_num());
21  }
22
23  void bar(){
24    printf("inside function bar, using thread number %d\n", omp_get_thread_num());
25
26  }
```

```
inside function foo, using thread number 1
inside function bar, using thread number 1
inside function foo, using thread number 0
inside function foo, using thread number 3
inside function foo, using thread number 2
```

2. Basics of OpenMP

2.3 Parallel for

- Most parallelism will be achieved with ***parallel for***
- Only integers are allowed in the *for* condition; only one update is allowed at end
- Thread 0 will be assigned **i=0**, thread 1 will receive **i=1**, …, thread 0 will receive i=N,….
- Can also append this with **schedule(static, chunksize)** to give each thread **chunksize** parts to do before assigning work to the next thread
- Can use dynamic scheduling, but this hinders at runtime

```
3
4 #pragma omp parallel for
5 for (int i=0; i<100; i++)
6 {
7   //code goes here
8
9 }
10
```

Durham University

2. Basics of OpenMP

2.4 Simplest Parallelism

- The simplest form of parallelism is a series of operations that are independent of each other

- Consider two arrays *b* and *c* of the same length, and the following code

```
15
16 for (int i = 0; i < N; i++){
17   a[i] = b[i] + c[i]
18 }
```

- Each of these operations is independent of one another.
- No two memory addresses are read from or written to during this process
- This kind of problem is known as **embarrassingly parallel.**
- We can modify it as such:

```
16 #pragma omp parallel for
17 for (int i = 0; i < N; i++){
18   a[i] = b[i] + c[i];
19 }
```

2. Basics of OpenMP

2.5 Variable Sharing

- By default, all variables are shared between threads.
- In the example on the last slide, that means the array ***a*** is available to each thread (and modifiable!)
- In many circumstances, variables should be marked ***private***.
- Variables marked ***private*** will be newly constructed copies by each of the threads, and only modifiable by that same thread (herein lies some of the cost of spawning threads). Note that the default constructor will be used (ie a double marked ***private*** will be 0 inside a parallel region)
- Variables marked ***firstprivate*** will again by newly constructed, but will copy the original value
- The safest way to do this is to specify ***default(none)***

```
3 double       x = 9;
4 const double a = 10;
5 double       b = 11;
6 #pragma omp parallel private(x) shared(a) firstprivate(b) default(none)
7 {
8   //code goes here
9   printf("%f", x); //prints  0
10  printf("%f", a); //prints 10
11  printf("%f", b); //prints 11
12
13 }
```

Durham University

2. Basics of OpenMP

2.6 Race Conditions

- What do you expect to happen in the following code:

```
2
3 int sum = 0;
4
5 for (int i = 1; i <= 10; i++){
6   sum += i;
7 }
```

What about in this instead?

```
2
3 int sum = 0;
4
5 #pragma omp parallel for //tackle using all threads
6 for (int i = 1; i <= 10; i++){
7   sum += i;
8 }
```

2. Basics of OpenMP

2.6 Race Conditions (cont.)

We don’t get the desired result, since updating “sum” consists of three operations:

- Read sum
- Update sum (sum → sum+1)
- Store the new sum

If thread 0 reads “sum” to be 5, just before thread 1 stores it to be 7, we will get the wrong result!

2. Basics of OpenMP

2.6 Race Conditions (cont.)

The solution is using reduction.

- Each thread gets a copy of “val”, and combines at the end, using the operation we specify
- The syntax is *reduction*(operation : variable). Supports operations like “+”, “-”, “*”
- You can add multiple reduction statements

```
2
3 int sum = 0;
4
5 #pragma omp parallel for reduction(+ : sum)
6 for (int i = 1; i <= 10; i++){
7   sum += i;
8 }
```

2. Basics of OpenMP

2.6 Race Conditions (cont.)

We can alternatively use a directive such as “atomic” or “critical” to indicate that only one thread at a time can perform a certain step, but this hinders performance.

- Forcing each thread to wait for the others defeats the point of using multiple threads

```
1 int sum = 0;
2
3 #pragma omp parallel for
4 for (int i = 1; i <= 10; i++){
5
6   #pragma omp critical
7   sum += i;
8 }
```

2. Basics of OpenMP

2.6 Tasks

Tasks are a new form of parallelism that allow for execution of arbitrary code blocks

- We specify a task with ***#pragma omp task***
- Each task is placed on a pool and picked up by a thread when there is one available. This can be good for reducing CPU idle time
- Since every thread runs each piece of code within a parallel region, we can ensure our tasks are only spawned once by using the ***master*** directive
- We can force execution to wait until the tasks are finished with a simple barrier

```
3
4  #pragma omp parallel
5
6  {
7    //code goes here
8    #pragma omp master
9    {
10
11     #pragma omp task
12     foo();
13
14     #pragma omp task
15     bar();
16
17     //wait for all tasks and sub-tasks to finish
18     #pragma omp taskwait
19
20   }
21
22 }
23
24
```

Durham University

3 Basics of MPI

The General Message Passing Paradigm

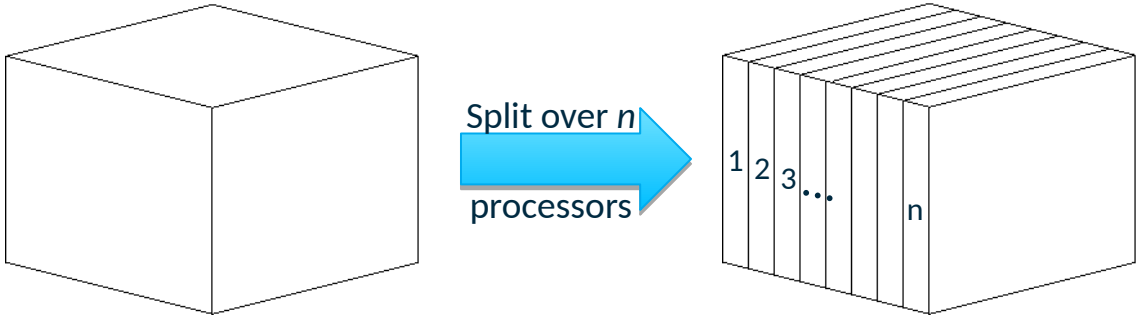
- All variables are private to each process. Values of variables are held in local memory – a distributed memory parallel computer
- Processes communicate via special subroutine calls to an external library
- Typically:
  - Communications are written in a conventional sequential language
  - A single program is compiled and executed across each processor
  - There is a generic interface i.e. the method/route of communication is hidden.

3 Basics of MPI

- “The goal of the Message Passing Interface, simply stated, is to develop a widely used standard for writing message-passing programs. As such, the interface should establish a practical, portable, efficient, and flexible standard for message passing.”
  - MPI-1, MPI-2, MPI-3, MPI-4 (1139pp, approved by the MPI Forum June 2021)
- There are multiple implementations (“flavours”) of this standard specification
  - MPICH
  - Open MPI (not the same as OpenMP!)
  - MVAPICH
  - Vendor-specific implementations – Intel® MPI, Cray MPI…

3 Basics of MPI

- Aim: to reduce time taken to achieve strong scaling
- Objective: decompose a task into smaller tasks which can be performed simultaneously i.e. in parallel
- Approach 1: domain or data decomposition:



- Approach 2: functional decomposition:
  - e.g. integration: splitting the interval $\int_a^b f(x)$ over *n* processors, e.g. *(b-a)/n*
  - e.g. Fourier transforms (1D to 3D), passing out pages from a book to each proces, sections of a database

3 Basics of MPI

3.1 Writing your first MPI program

The Standard…	C	FORTRAN
Essential header files	`#include <mpi.h>`	`include 'mpif.h'`

Initialisation (**always** the first MPI procedure called. Never called more than once)	`int main (int argc, char *argv[]){`
	
	`MPI_Init(&argc, &argv);`
		`INTEGER IERR`
		`CALL MPI_INIT(IERR)`

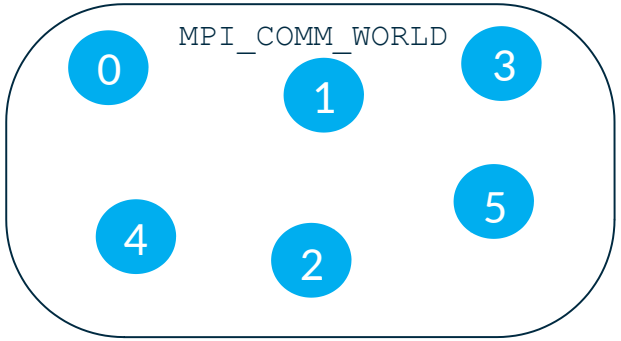
Finalisation (**Essential**. Must be the last MPI procedure called)	`MPI_Finalize();`	`CALL MPI_FINALIZE(IERR)`

Function syntax…	Case sensitive…
	`Error = MPI_Xxxx(parameter, …);`
	`MPI_Xxxx(parameter)`
		Case insensitive…
		`CALL MPI_XXXX(parameter, …, IERR)`
		
		`IERR` returns 0 (success) or 1 (fail), same as `return()` in C.

3 Basics of MPI

3.1 Writing your first MPI program

- Communicators define a group of processes between which message passing can occur.
- By default, the communicator `MPI_COMM_WORLD` is automatically generated at initialization, does not need to be declared and contains all processes when execution begins:

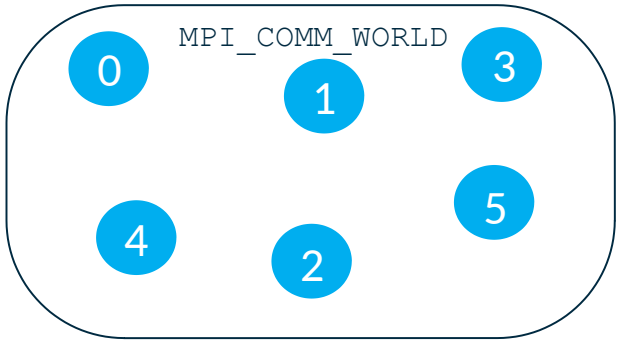


3 Basics of MPI

3.1 Writing your first MPI program

The MPI *rank* returns an integer number for each ‘process’ in a ‘communicator’ group, numbered from 0 in both C and FORTRAN. The rank is only defined by MPI and is not linked to any other identified e.g. CPU #, core #, node #

C	FORTRAN
`int rank;` `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`	`INTEGER RANK, IERR` `CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)`



The communicator
`MPI_COMM_WORLD`
(queried by the commands)
contains ranks 0 to 5

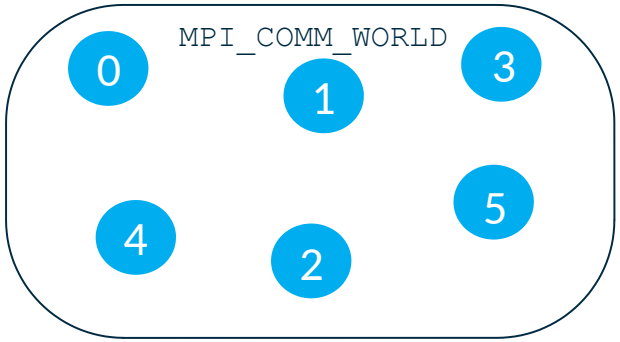
Durham University

3 Basics of MPI

3.1 Writing your first MPI program

The MPI *size* returns the total number of ranks in a communicator group, again an integer.

C	FORTRAN
`int size;` `MPI_Comm_size(MPI_COMM_WORLD, &size);`	`INTEGER SIZE, IERR` `CALL MPI_COMM_SIZE(MPI_COMM_WORLD, SIZE, IERR)`



The communicator
`MPI_COMM_WORLD`
(queried by the commands)
contains 6 ranks in total

3 Basics of MPI

3.1 Writing your first MPI program

A code checklist…

- Headers: `mpi.h` / `mpif.h`
- Initialisation before anything else MPI: `MPI_Init`
- Rank, size commands: `MPI_Comm_rank` / `MPI_Comm_size`
  - Insert some code employing MPI functionality here!
- Finalisation should be the last MPI procedure: `MPI_Finalize`

With only the header, initialization and finalization, any MPI code will compile and run equivalently to a serial code.

3 Basics of MPI

3.1 Writing your first MPI program

Compilation on Hamilton, after you have logged into your account…

- Hamilton has a module system and by default, no modules are available.
- To see what is available: `module avail`
- To load compilers and MPI:

```
module load intel/2021.4
module load intelmpi/2021.6
```

- To compile:

C	FORTRAN
`mpicc my_prog.c -o myprogram`	`mpif90 my_prog.f -o myprogram`

- To very briefly test on the login node using 4 processes:

```
mpirun -np 4 ./program
```

But do not make a habit of doing this! Use the queues…

3 Basics of MPI

3.1 Writing your first MPI program

To fairly share the available resources, Hamilton has a queueing system.

- To access this, you must write and submit a job script:
- Submit: `sbatch job.sh`
- Status: `squeue –u user`
- Estimated start time: `squeue –start –u user`
- Cancel: `scancel jobID`
- Cancel all your jobs: `scancel –u user`
- Get account info: `sacct –u user`
- Get job info (e.g. total memory used etc.): `sacct –j jobID`

Job.sh file contents:

```
#!/bin/bash
#SBATCH --job-name="my-first-script"
#SBATCH -o myscript.%A.out
#SBATCH -e myscript.%A.err
#SBATCH -p test.q
#SBATCH -t 00:05:00
#SBATCH -N 1 # number of nodes
#SBATCH -n 4 # number of tasks (MPI ranks)
#SBATCH -c 4 # number of cores per task

module purge
module load intel/2021.4
module load intelmpi/2021.6
mpirun ./myprogram
```

Practical 1: Hello World!

- Write a minimal MPI program that prints "`Hello World!`"
  - Serial template code is available for C and FORTRAN on Hamilton here:-
    `/home/lcgk69/Courses/BasicProgrammingMPI/practical1/helloworld.c`
    `/home/lcgk69/Courses/BasicProgrammingMPI/practical1/helloworld.f90`
- Compile your code.
- Run it on a single processor on the login node.
- Run it on a single processor *via the batch queue*. Job script:
  `/home/lcgk69/Courses/BasicProgrammingMPI/practical1/job.sh`
- Run it on several processors in parallel *via the batch queue*.
- Modify the code (with an if statement) such that only rank 0 prints "`Hello World!`"
- Modify the code such that the ranks print:-

"`Hello World! I am rank # of size #.`"

Practical 1 Review

C

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

int main (int argc, char *argv[]) {
   int rank, size;

   MPI_Init(&argc, &argv); /*
Initialise MPI */
   MPI_Comm_rank(MPI_COMM_WORLD,
&rank); /* Get rank */
   MPI_Comm_size(MPI_COMM_WORLD,
&size); /* Get size */

   printf("Hello from rank %d of size
%d.\n", rank, size);

   MPI_Finalize();
}
```

FORTRAN

```
PROGRAM helloworld
IMPLICIT none
include 'mpif.h'
INTEGER rank, size, ierr
! Initialise MPI
CALL MPI_Init(ierr)
! get processor rank
CALL MPI_Comm_rank(MPI_COMM_WORLD,
rank,ierr)
! Get total number of processors
CALL MPI_Comm_size(MPI_COMM_WORLD,
size,ierr)

write (*,*) 'Hello from rank ',rank,'
of size ',size

call MPI_FINALIZE(ierr)
end program helloworld
```

Durham University

See also: `/home/lcgk69/Courses/BasicProgrammingMPI/solutions1/`

3 Basics of MPI

3.2 Point-to-point communications

Messages

- Data types

Communication modes and completion

- Sends: synchronous / buffered / ready / standard
- Receive
- Success criteria
- Wildcarding

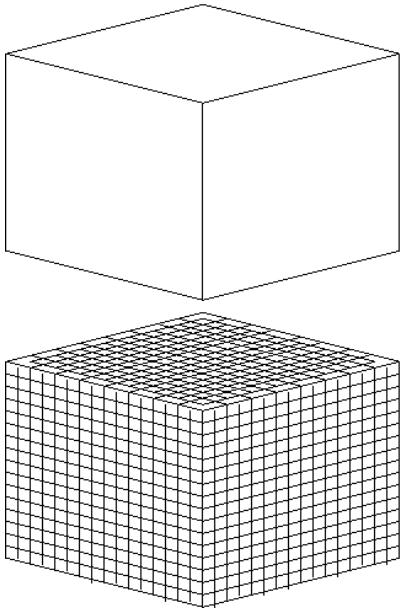
Communication envelope

Message order preservation

Combined send and receive

3 Basics of MPI

3.2 Point-to-point communications

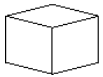


- An example: this is a representation of a domain for a piece of CFD software that solves the equations of fluid dynamics to evolve a fluid with time:-
- The domain is broken down into a number of cells:- (e.g. 20 x 20 x 20: 8000 cells)
- If solving Euler's equation takes 1 second to evolve the fluid in a cell by one second of simulation time, a single processor would take 8000s to update this whole grid by 1s of simulation time. Evolving the grid by days would correspondingly take 8000x longer – *decades!*
- In some cases, each task is self-contained – cells need only know about their own conditions to calculate their update – and the simulation becomes "embarrassingly parallel".

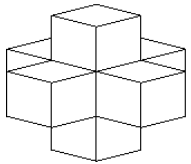
3 Basics of MPI

3.2 Point-to-point communications

In reality, and certainly in this CFD example, this is not the case.

In our code, each cell: 

In order to calculate the flow between cells and update its own fluid conditions, the code needs to know about the conditions in its neighbours in every direction:



What happens if a neighbouring cell is held in different memory on another process?

- Communication must occur between processes
- "Message passing" is the context in which this takes place, using a message passing interface, or **MPI** library
- The message passing system needs to be aware of the following information:
  1) The 'rank' of the message source
  2) Source buffer: variable / array location
  3) MPI data type
  4) The 'rank' of message destination
  5) Destination buffer
  6) Size of sending and receiving buffer(s)
- Messages contain a number of elements of a particular data type.

3 Basics of MPI

3.2 Point-to-point communications

C: MPI Data types	FORTRAN: MPI Data types
MPI_CHAR	MPI_CHARACTER
MPI_SHORT	
MPI_INT	MPI_INTEGER
MPI_LONG	
MPI_UNSIGNED_CHAR	MPI_LOGICAL
MPI_UNSIGNED_SHORT	MPI_COMPLEX
MPI_UNSIGNED	
MPI_UNSIGNED_LONG	
MPI_FLOAT	MPI_REAL
MPI_DOUBLE	MPI_DOUBLE_PRECISION
MPI_LONG_DOUBLE	MPI_REAL8
MPI_BYTE	MPI_BYTE
MPI_PACKED	MPI_PACKED

3 Basics of MPI

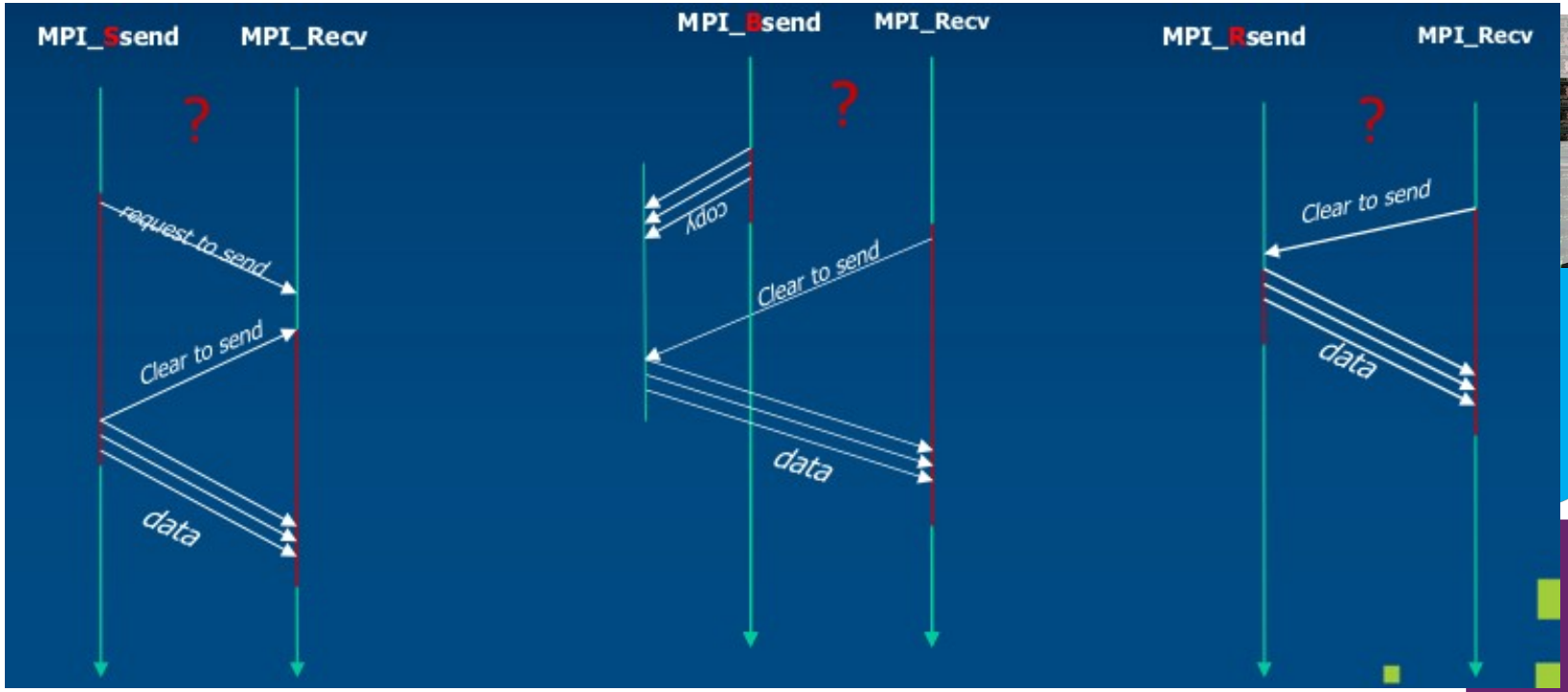
3.2 Point-to-point communications

Sender mode	MPI Call	Completion status
Synchronous send	`MPI_Ssend`	Only completes when the receive has completed.
Buffered send	`MPI_Bsend`	Always completes (unless an error occurs), irrespective of receiver.
Standard send	**`MPI_Send`**	**Can be synchronous or buffered (often implementation dependent).**
Ready send	`MPI_Rsend`	Always completes (unless an error occurs), irrespective of whether the receive has completed.
Receive	**`MPI_Recv`**	**Completes when a message arrives.**

3 Basics of MPI

3.2 Point-to-point communications

Communication modes - explained



3 Basics of MPI

3.2 Point-to-point communications

FORTRAN sending syntax:

```
CALL MPI_SSEND(buf, count, datatype, dest, tag, comm,
ierr)
```

- `buf`: start of data to be sent.
- `count`: number of elements to send (integer).
- `datatype`: type of data.
- `dest`: destination process (integer).
- `tag`: label to identify this instance (integer).
- `comm`: communicator group.
- `ierr`: integer error code

e.g. sending `1` integer in `data` to `rank=2` (`tag=100`)

```
CALL MPI_SSEND(data, 1, MPI_INTEGER,
     2, 100, MPI_COMM_WORLD, ierr)
```

3 Basics of MPI

3.2 Point-to-point communications

C sending syntax:

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

- `*buf`: pointer to start of data.
- `count`: number of elements to send.
- `datatype`: type of data.
- `dest`: destination process.
- `tag`: label to identify this instance of communication.
- `comm`: communicator group.

e.g. sending `1` integer `data` to `rank=2` (`tag =100`)

```
MPI_Ssend(&data, 1, MPI_INT,
   2, 100, MPI_COMM_WORLD);
```

3 Basics of MPI

3.2 Point-to-point communications

FORTRAN receiving syntax:

```
CALL MPI_RECV(buf, count, datatype, source, tag, comm,
status, error)
```

- `buf`: starting location where data should be put
- `count`: number of elements to receive (integer)
- `datatype`: type of data
- `source`: sending process rank (integer)
- `tag`: message identifier (integer)
- `comm`: communicator
- `status`: integer array of size MPI_STATUS_SIZE
- `error`: integer error code

e.g. receiving `1` integers into `data2` from `rank=1` (`tag=100`)

```
CALL MPI_RECV(data2, 1, MPI_INT, 1, 100,
     MPI_COMM_WORLD, status, error)
```

3 Basics of MPI

3.2 Point-to-point communications

C receiving syntax:

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

- `*buf`: pointer to start of receiving buffer
- `count`: number of elements to receive
- `datatype`: type of data
- `source`: sending process rank
- `tag`: message identifier
- `comm`: communicator
- `*status`: pointer to message envelope

e.g. receiving `1` integers into `data2` from `rank=1` (`tag=100`)

```
MPI_RECV(&data2, 1, MPI_INT, 1, 100,
        MPI_COMM_WORLD, &status);
```

3 Basics of MPI

3.2 Point-to-point communications

C example:

```
#include <mpi.h>
int main (int argc, char *argv[]){
  int rank, size, n=5;
  int sbuf[n], rbuf[n];
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    MPI_Ssend(&sbuf[0],n,MPI_INT,1,99,MPI_COMM_WORLD);
  }
  if (rank == 1) {
    MPI_Recv(&rbuf[0],n,MPI_INT,0,99,MPI_COMM_WORLD,
             &status);
  }
  MPI_Finalize();
}
```

3 Basics of MPI

3.2 Point-to-point communications

FORTRAN example:

```
PROGRAM mpi
IMPLICIT NONE
INCLUDE 'mpif.h'
INTEGER :: rank, size, status(MPI_STATUS_SIZE), ierr
INTEGER, PARAMETER :: n=5
INTEGER :: sbuf(n), rbuf(n)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr);
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr);

IF (rank .EQ. 0) THEN
   CALL MPI_SSEND(sbuf(1),n,MPI_INTEGER,1,99,MPI_COMM_WORLD,ierr)
ENDIF
IF (rank .EQ. 1) THEN
   CALL MPI_RECV(rbuf(1),n,MPI_INTEGER,0,99, &
             MPI_COMM_WORLD,status,ierr)
ENDIF

CALL MPI_FINALIZE(ierr);
END PROGRAM mpi
```

3 Basics of MPI

3.2 Point-to-point communications

Wildcarding:

The receiving process can wildcard

To receive from any source:

- Set source to `MPI_ANY_SOURCE`

To receive with any tag:

- Set tag to `MPI_ANY_TAG`

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

3 Basics of MPI

3.2 Point-to-point communications

The `status` communication envelope:

Like a letter there is much more information in a message than just the body text:

- Sender's address
- Reference number
- How many pages

Returned in the `status` parameter are:

- Source
- Tag
- Error code

It is also possible to query the received count

3 Basics of MPI

3.2 Point-to-point communications

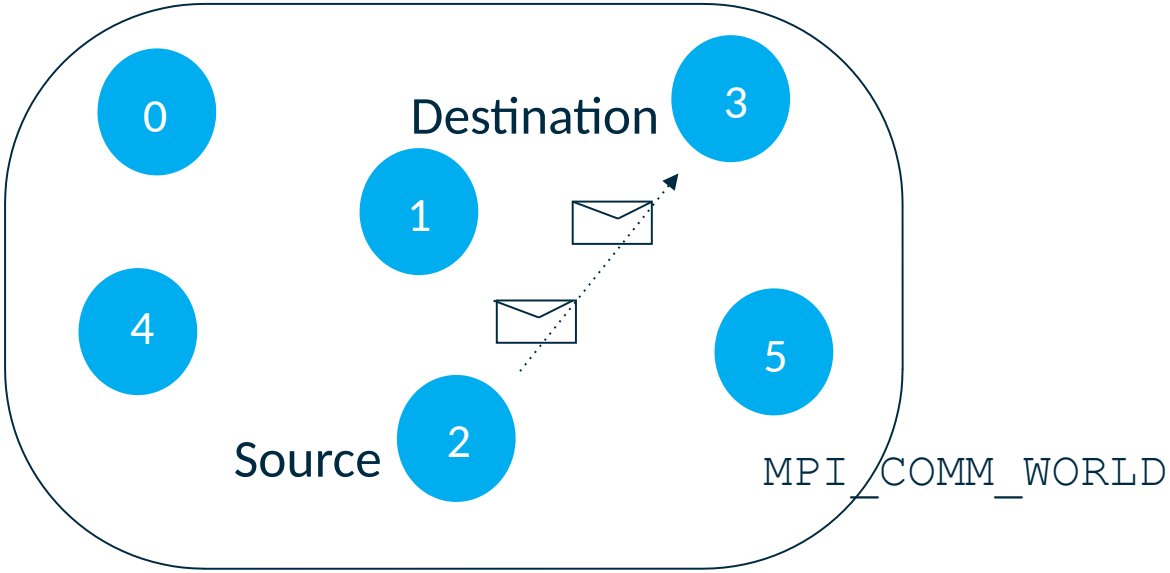
- In C, `status` is a structure containing three fields
- In FORTRAN, `status` is an array of `INT`s of size `MPI_STATUS_SIZE`

Querying the status parameter	C	FORTRAN
Source process	`source=status.MPI_SOURCE;`	`source=status(MPI_SOURCE)`
Tag	`tag=status.MPI_TAG;`	`tag=status(MPI_TAG)`
Error code	`error=status.MPI_ERROR;`	`error=status(MPI_ERROR)`
Count	`MPI_Get_count(&status, MPI_datatype, &count);`	`MPI_GET_COUNT(status, MPI_datatype, count, ierr)`

3 Basics of MPI

3.2 Point-to-point communications

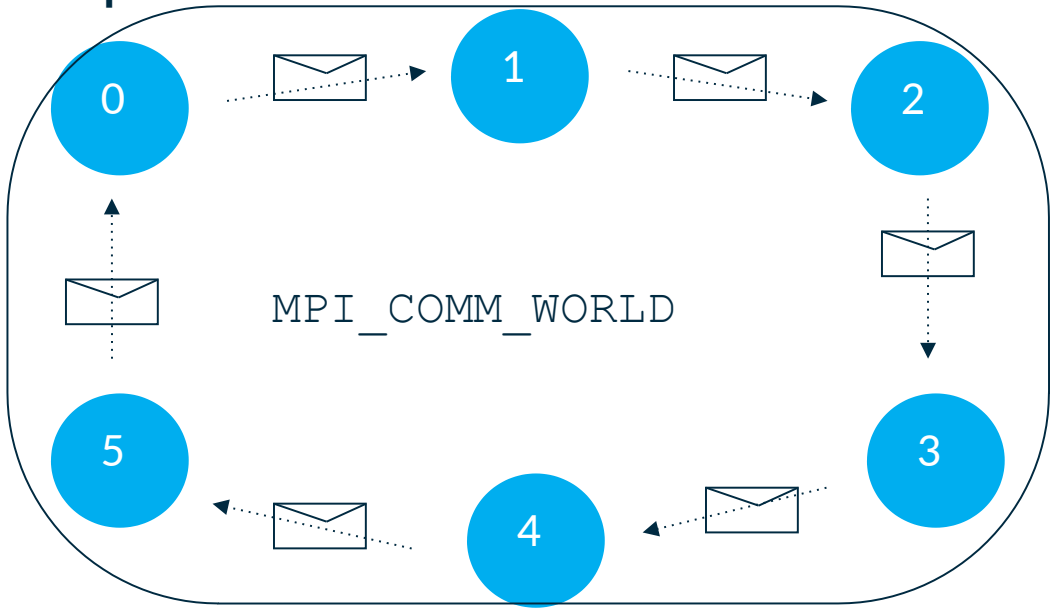
The order of messages is preserved:



- Messages do not overtake each other.
- This is also true for non-synchronous (buffered) sends.

3 Basics of MPI

3.2 Point-to-point communications



Deadlock

- Deadlock occurs if all processes post a synchronous send before a receive operation.
- All processes will hang or ‘deadlock’, waiting for a receive that has never been posted.

A new type of bug for programmers to be aware of.

3 Basics of MPI

3.2 Point-to-point communications

Deadlock avoidance: carry out non-blocking communication

Sending process	Receiving process
Initiate send, non-blocking (`MPI_Issend`)	Initiate receive, non-blocking (`MPI_Irecv`)
Perform other tasks	Perform other tasks
Wait for completion (`MPI_Wait`)	Wait or test for completion (`MPI_Test`)

Relies upon a ‘request’ handle

- Allocated when a communication is initiated.
- Can be queried to test whether non-blocking operation has been completed.
- A non-blocking call followed by an explicit wait, is identical to the blocking communication.

3 Basics of MPI

3.2 Point-to-point communications

Deadlock avoidance 2:

`MPI_Send` and `MPI_Recv` can be carefully ordered to avoid deadlocks. This can be difficult and time consuming.

MPI also provides a very useful *combined* send and receive function, `MPI_Sendrecv`, which is guaranteed not to deadlock.

- This routine sends a message and posts a receive, then blocks until the send data buffer is free and the receive data buffer has received its data.

3 Basics of MPI

3.2 Point-to-point communications

MPI_Sendrecv

C:

```
int MPI_Sendrecv( void *sendbuf, int sendcount,  MPI_Datatype
sendtype, int destination, int sendtag, void *recvbuf, int
recvcount, MPI_Datatype recvtype, int source, int recvtag,
MPI_Comm comm, MPI_Status &status);
```

FORTRAN:

```
REAL sendbuf(*)
REAL recvbuf(*)
INTEGER sendcount, dest, sendtag
INTEGER recvcount, source, recvtag
INTEGER comm, status(MPI_STATUS_SIZE), ierr

CALL MPI_SENDRECV(sendbuf[1], sendcount, MPI_REAL, dest,
sendtag, recvbuf[1], recvcount, MPI_REAL, source, recvtag,
comm, status, ierr)
```

3 Basics of MPI

3.2 Point-to-point communications

MPI_Sendrecv

- `MPI_PROC_NULL` can be specified instead of the rank of the source or the destination
    - Useful for doing non-circular shifts with `MPI_Sendrecv`
- A message sent by `MPI_Sendrecv` can be received by a regular receive operation
- A message sent by a regular send can be received by `MPI_Sendrecv`
- The send and receive buffers must not overlap
    - If you want to use the same buffer for both the send and receive, use `MPI_Sendrecv_replace`

Practical 2: point-to-point communications

1. Node pair communication
    - Write a program in which two processes repeatedly pass a message (e.g. a random integer) back and forth, altering the message along the way.
    - Template:
      `/home/lcgk69/Courses/BasicProgrammingMPI/practical2/PingPong.c`
      `/home/lcgk69/Courses/BasicProgrammingMPI/practical2/PingPong.f90`
2. Bonus exercise: cycling communication
    - Modify the node pair communication program so that several processes pass a message around the group, printing at each stage.
    - Perform a simple mathematical alteration of the message on each process and populate an array across the nodes with the data.

Practical 2 Review

The principles of node pair communication

```
   send = 8  /* Initialise send buffer */
     Loop 100 times  /* repeat for 100 iterations */

         On Processor 1 {
/* blocking send on first processor to second */
             MPI_Ssend(send,1,MPI_INT, 1, 1, MPI_COMM_WORLD);
/* blocking receive on first processor from second */
             MPI_Recv(recv,1,MPI_INT, 1, 2, MPI_COMM_WORLD, &status);
             send = recv + 1;
         } whilst on Processor 2 {
/* blocking receive on second processor from first */
             MPI_Recv(recv,1,MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
             send = recv + 1;
/* blocking send on first processor to second */
             MPI_Ssend(send,1,MPI_INT, 0,2,MPI_COMM_WORLD);
         }
     }
```

For the complete answers, please consult the solutions:

```
/home/lcgk69/Courses/BasicProgrammingMPI/solutions2/
```

3 Basics of MPI

3.3 Collective communications

- Introduction & characteristics
- Barrier Synchronisation
- Broadcast
- Scatter
- Gather
- Global reduction operations
  - Predefined operations
  - User-defined operations
- Partial sums

3 Basics of MPI

3.3 Collective communications

Collective communication involves a group of processes.

Called by *all* processes in a communicator.

Examples:

- Broadcast, scatter, gather (Data Distribution)
- Global sum, global maximum, etc. (Reduction Operations)
- Barrier synchronisation

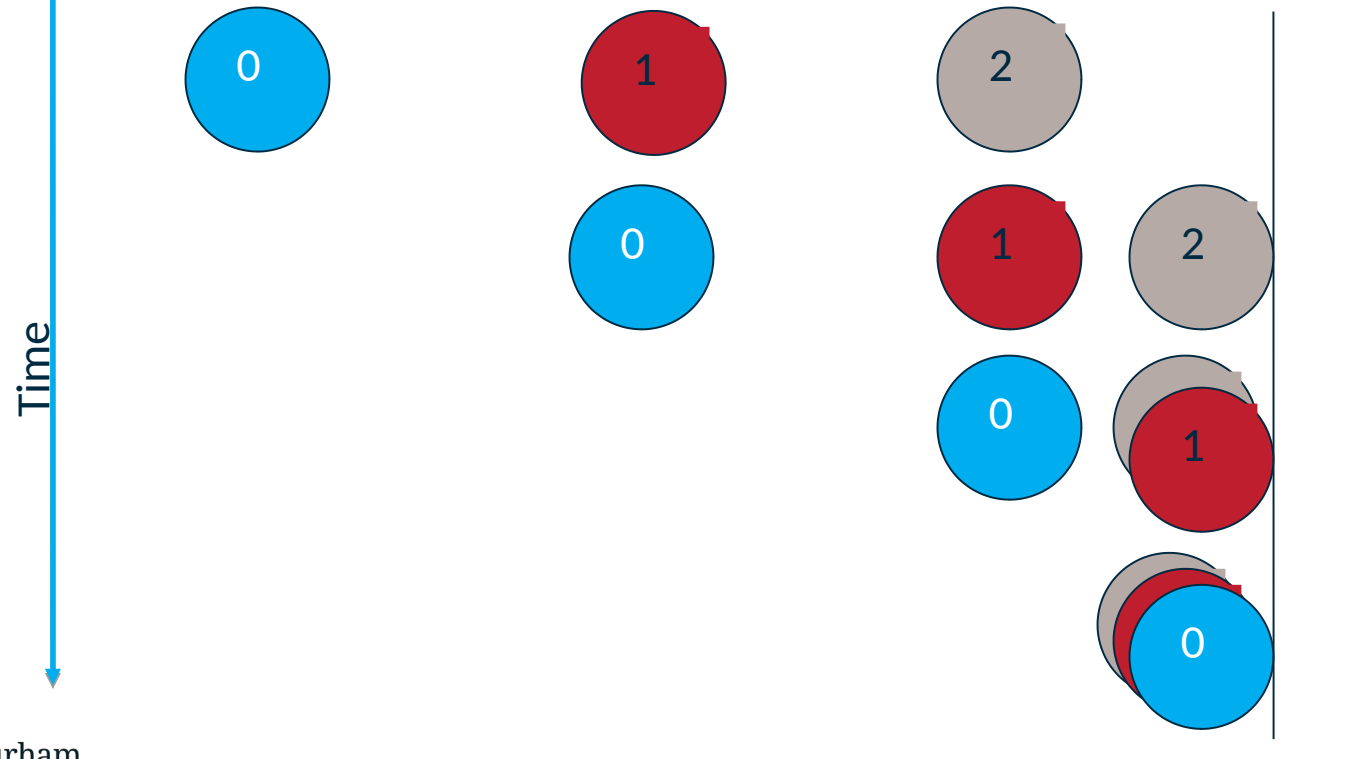
Characteristics

- Collective communication will not interfere with point-to-point communication and vice-versa.
- All processes must call the collective routine.
- Synchronization not guaranteed (except for barrier)
- No non-blocking collective communication
- No tags
- Receive buffers must be exactly the right size

3 Basics of MPI

3.3 Collective communications

Barrier Synchronisations: MPI_Barrier()



Durham University

3 Basics of MPI

3.3 Collective communications

Barrier Synchronisation

Each processes in communicator waits at barrier until all processes encounter the barrier.

Fortran:

```
INTEGER comm, error
CALL MPI_BARRIER(comm, error)
```

C:

```
MPI_Barrier(MPI_Comm comm);
```

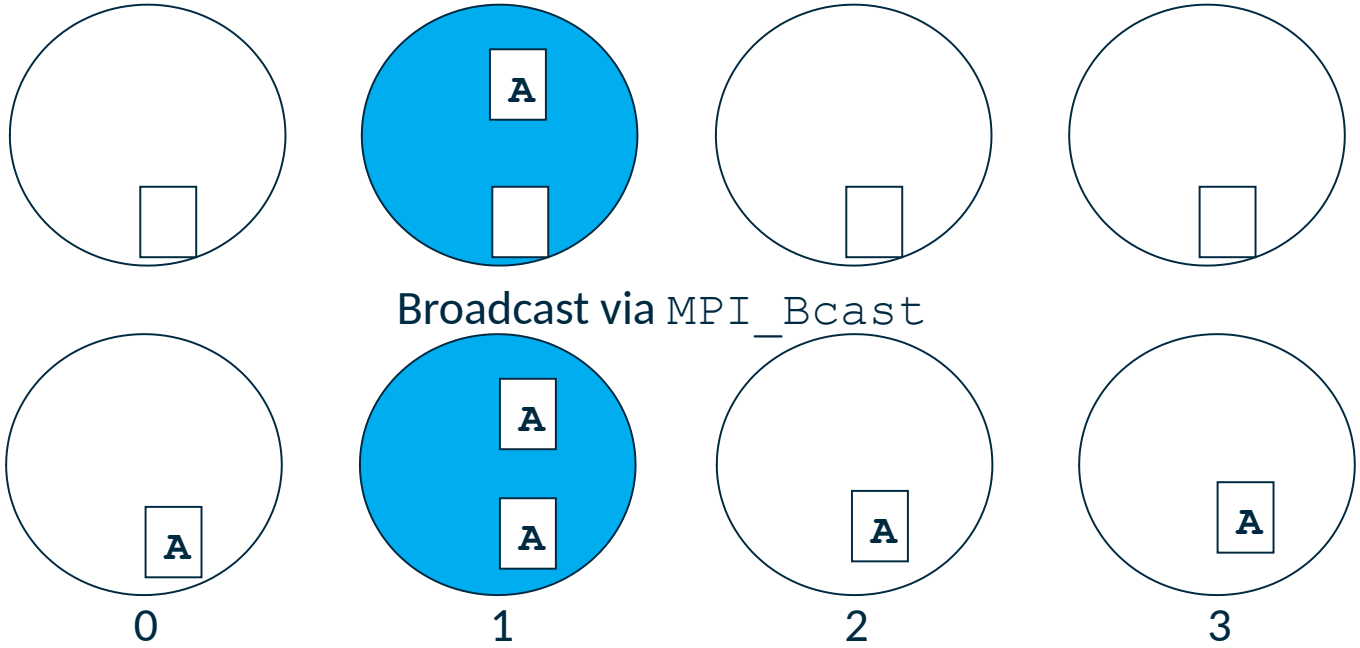
Note:

Barrier calls are exceptionally useful for avoiding ‘racing’ issues, where one processor can race ahead of the others and set up deadlock.

3 Basics of MPI

3.3 Collective communications

Broadcasting: duplicates data from one process to all other processes in communicator group



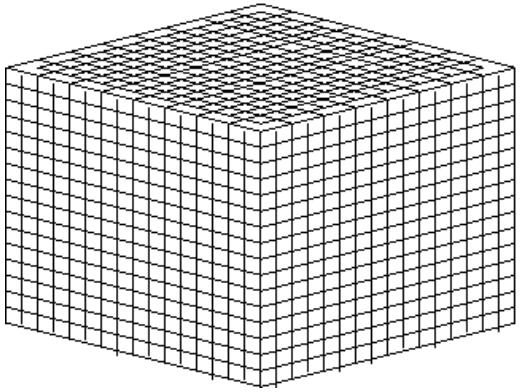
As with all collective processes, must be called simultaneously by every process

3 Basics of MPI

3.3 Collective communications

Recall our fluid dynamics example...

- Each cell in the domain



has to be advanced in time by the same amount – the *timestep.*

This timestep could be ‘broadcast’ by a master processor.

3 Basics of MPI

3.3 Collective communications

Broadcast syntax:

Fortran:

```
INTEGER count, datatype, root, comm, ierr
CALL MPI_BCAST(buffer[1],count,datatype,root,comm,ierr)
```

C:

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

- e.g broadcasting `deltat` from rank `0` to the entire group:

```
double deltat;
MPI_Bcast(deltat, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

3 Basics of MPI

3.3 Collective communications

Multiple data gathering and scattering routines exist

- `MPI_Scatter` – scatters data from a single process to all processes
- `MPI_Gather` – gathers data from all processes to a single process
- `MPI_Allgather` – each process receives a copy of the gathered data.
- `MPI_Alltoall` – gathers data and scatters (possibly different) data from all to all processes - very much the basis of parallelized Fourier transforms

  - Note: this command can be very taxing for the interconnection, sending multiple small messages between all processes. It seems particularly demanding on the newest variety of architecture with ~128 cores in a dual-CPU node.

- Gather/scatter with varying amount of data on each process
  - `MPI_GATHERV, MPI_SCATTERV, MPI_ALLGATHERV, MPI_ALLTOALLV`

3 Basics of MPI

3.3 Collective communications

Global Reduction operations

- Compute a result involving data distributed over a group of processes.
- Suppose that each process *i* has computed a number X_i and that the result needed *X* is the sum of these. This global sum is an example of a *reduction operation*.
- In MPI, a set of binary reduction operations are defined for predefined MPI data types.
  - All binary operations are assumed to be associative: (x*y)*z = x*(y*z)
  - All the predefined binary operations are also commutative: x*y = y*x
    - It is possible to define non-commutative binary operations.
- The order in which the reduction is done is unspecified. MPI guarantees the result will only be the same to within round-off errors.

3 Basics of MPI

3.3 Collective communications

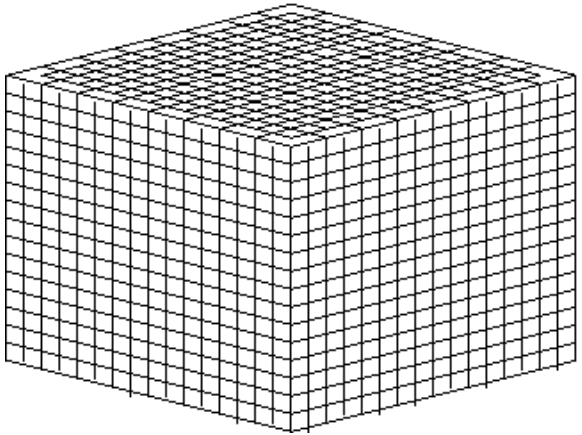
MPI name	Function	C	FORTRAN
MPI_MAX	Maximum		MAX(a_1… …a_n)
MPI_MIN	Minimum		MIN(a_1… …a_n)
MPI_SUM	Sum	+	+
MPI_PROD	Product	*	*
MPI_LAND	Logical AND	&&	.AND.
MPI_BAND	Bitwise AND	&	
MPI_LOR	Logical OR	||	.OR.
MPI_BOR	Bitwise OR	|	
MPI_LXOR	Logical exclusive OR	!=	.NEQV.
MPI_BXOR	Bitwise exclusive OR	^	
MPI_MAXLOC	Maximum and location		
MPI_MINLOC	Minimum and location		

3 Basics of MPI

3.3 Collective communications

Recalling our fluid dynamics example...

- Each cell in the domain



has to be advanced in time by the *timestep.*

Each processor can calculate its own timestep based on it's section and then the minimum of all these values is used as the global timestep.

3 Basics of MPI

3.3 Collective communications

`MPI_Allreduce`: Combines values from all processes and distributes the result back to all processes. Function syntax:

Fortran

```
INTEGER count, type, count, rtype, comm, error
CALL MPI_ALLREDUCE(sbuf[1], rbuf[1], count, rtype, op, comm, error)
```

C:

```
MPI_Allreduce(void *sbuf, void *rbuf, int count, MPI_Datatype
   datatype, MPI_Op op, MPI_Comm comm);
```

For example, in our CFD case:

```
MPI_Allreduce(deltat, deltat_global_min, 1,
   mpi_real, MPI_MIN, MPI_COMM_WORLD, ierr)
```

3 Basics of MPI

3.3 Collective communications

`MPI Scan`: Computes the scan (partial reductions) of data on a collection of processes.
Function syntax:

Fortran:

```
REAL sendbuf(*), recvbuf(*)
INTEGER count, type, count, rtype, comm, error
CALL MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm, ierr)
```

C:

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

Practical 3: collective communications

Intro to MPI

1. Collective communication with `MPI_Allreduce`
   - Compute the global sum of all ranks of the processes using MPI global reduction
   - Template:
     `/home/lcgk69/Courses/BasicProgrammingMPI/practical3/collective.c`
     `/home/lcgk69/Courses/BasicProgrammingMPI/practical3/collective.f90`
2. Collective communication with `MPI_Scan`
   - Rewrite the previous program so that each process computes a partial rank sum
   - Additional task: make sure that the output is in natural order

Extra

A. Advanced topics (part I)

- I/O using MPI-IO
  - The best idea is just to use libraries built using MPI-IO: Parallel HDF5 (parallel IO in the HDF format), NetCDF (network Common Data Form)
- Cartesian Topologies
  - Create with `MPI_Cart_create`; translate rank into coordinates with `MPI_Cart_coords`; locate neighbours in every direction with `MPI_Cart_shift`
- Derived data types
  - Construct data types with `MPI_Type_contiguous`, `MPI_Type_vector`, etc.; commit with `MPI_Type_commit`; free with `MPI_Type_free`
- User-defined operations
  - Bind a user-define operation `MPI_Op_create`; free after use with `MPI_Op_free`

A. Advanced topics (part II)

- Creating new communicators
  - Split an existing communicator into multiple non-overlapping communicators `MPI_Comm_split`; create a duplicate of a communicator `MPI_Comm_dup`
  - Subdivide a communicator using process groups – extract the process group associated with the input communicator `MPI_Comm_group`; make a new group from selected members or by manipulating groups (see below); form a communicator based on the input group `MPI_Comm_create` (or a newer, more efficient, `MPI_Comm_create_group`)
  - Create new groups – `MPI_Group_union`, `MPI_Group_intersection`, `MPI_Group_difference`, `MPI_Group_incl`, `MPI_Group_excl`, ...
  - Free groups and communicators `MPI_Group_free` and `MPI_Comm_free`

B. Some libraries using MPI

- Numerical libraries
  - BLACS – Basic Linear Algebra Communication Subprograms (http://www.netlib.org/blacs/)
  - PBLAS – Parallel Basic Linear Algebra Subprograms (http://www.netlib.org/scalapack/pblas_qref.html)
  - ScaLAPACK – Scalable Linear Algebra PACKage (http://www.netlib.org/scalapack/)
  - FFTW – “Fastest Fourier Transform in the West” (http://www.fftw.org/)
  - NAG Parallel Library (https://www.nag.com/content/nag-mpi-parallel-library)
  - PETSC – Portable, Extensible Toolkit for Scientific Computation (https://petsc.org)
  - deal.II – Differential Equations Analysis Library (https://www.dealii.org/)

B. Some useful advice for programming on MPI

- Adding MPI can destroy a code
  - Always maintain a serial version so its possible to compile and run serial and parallel versions and compare output
- To ease clarity, separate out communication routines
  - Separate file
  - Dummy library for serial code
  - Avoids explicit MPI references in main code
- It's possible to do most things with only `MPI_Send` and `MPI_Recv` if portability is a great concern
  - Collective routines (`MPI_Gather`, `MPI_Bcast`, `MPI_Scatter`) are often better optimised than writing your own versions
- Parallel debugging can be hard. With **gdb**, the following opens <*NP*> xterminals, in each of them, you'll need to type `run` to begin executing:
  - `mpirun -np <NP> xterm -e gdb ./program`

Durham University

That's it! Good luck writing your own parallel code!

Thank you!

Feedback
https://bit.ly/arc_trainingfeedback

Email: arc@durham.ac.uk
RSE team: arc-rse@durham.ac.uk
Web: https://www.dur.ac.uk/arc/