Durham University

Parallel programming on a supercomputer

Instructors:
Thomas Flynn (RSEng)
Dmitry Nikolaenko (RSEng)
Martha Correa-Delval (RSEng)

October 30, 2025

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

Course Outline

Parallel programming with **OpenMP** and **MPI** using Durham University's supercomputer, Hamilton.

Aims of the course:

- Introduction to **parallel programming** basics
- See how **OpenMP** is used for **shared**-memory parallelism
- Learn how adding '*pragmas*' to an existing serial code can allow for multi-threading
- See how **MPI** is used for **distributed**-memory parallelism
- Learn how to use MPI commands to **pass messages**
- Learn about **point-to-point** and **collective** parallel communications

Course Schedule

09:00-09:15 – Setup; brief introduction to HPC and parallel programming models

09:15-10:00 – **OpenMP threads and tasks**

09:30-09:45 – Practical 1 and 2: “Scheduling” and “Race Conditions”

09:50-10:00 – Practical 3: “Task Parallelism”

10:00-10:15 – **Break**

10:15-11:15 – **Basics of MPI and point-to-point communications**

11:00-11:15 – Practical 4: “Ping pong!”

11:15-11:25 – **Break**

11:25-12:25 – **Collective communications**

12:10-12:25 – Practical 5: “Collective communication”

12:25-12:30 – Conclusion

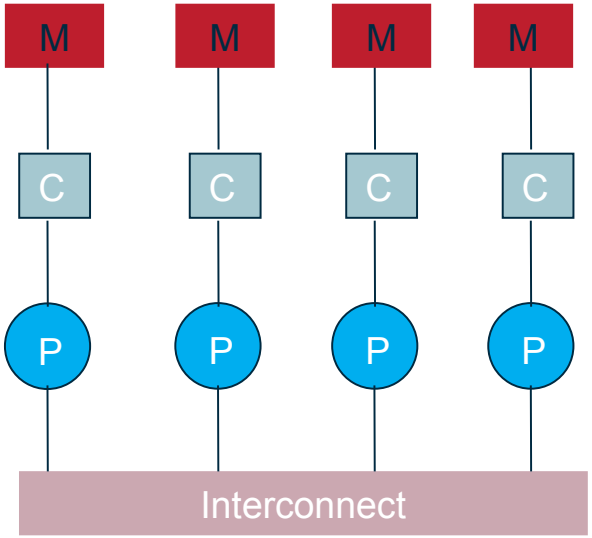
Durham University

Introduction

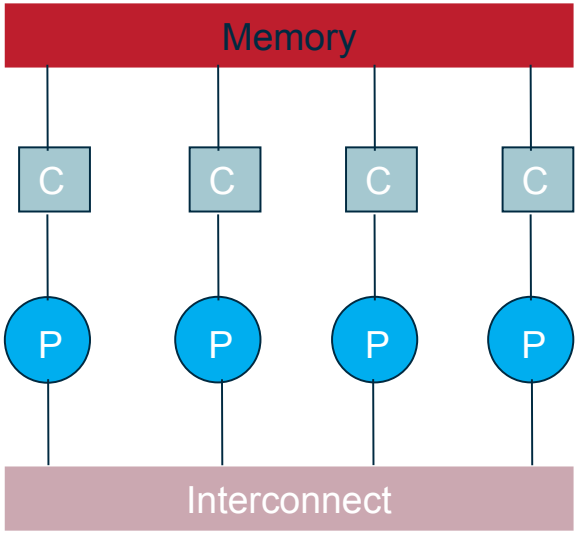
Durham University

1 Brief intro to HPC and parallel programming models

1.1 Distributed memory and shared memory systems



Distributed memory system. This parallelism relies on message passing



Shared memory system, e.g. multiprocessor desktop PCs. This parallelism is sometimes known as **multithreading**.

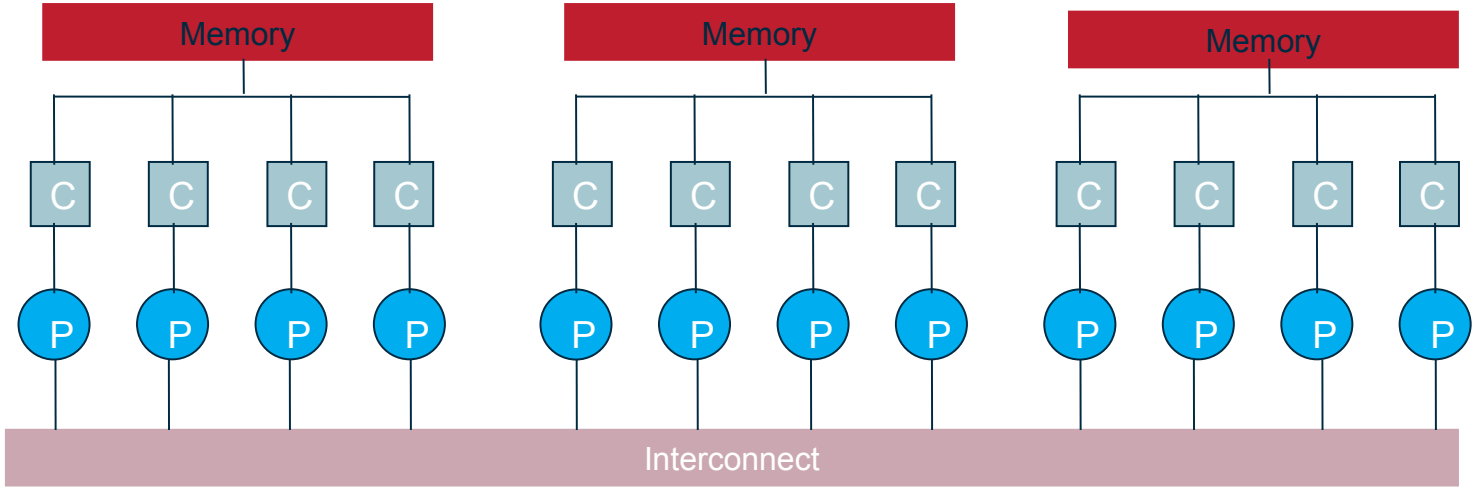
Durham University

P Processor

C Cache

1 Brief intro to HPC and parallel programming models

1.2 Basic HPC cluster structure



Modern clusters look more like this: **shared** memory on individual **nodes** with **distributed** memory between **nodes**

1 Brief intro to HPC and parallel programming models

1.3 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

Setup

Durham University

Setup

Hamilton (Durham's HPC Cluster)

- Accessing Hamilton
  - Ensure you have an active account on Hamilton (request access if needed).
  - Use SSH to connect:

```
ssh <username>@hamilton8.dur.ac.uk
```

- Environment setup
  - Load necessary modules (compilers for C and Fortran, MPI library):

```
module load gcc openmpi
```

- Compile & run
  - Confirm everything is functioning by compiling a small MPI/OpenMP "hello world" program

```
mpicc helloworld.c -o helloworld
```

  - Submit a job to run on Hamilton
  - `sbatch job.sh`

OpenMP

Durham University

2 Basics of OpenMP

- OpenMP allows independent units of work to be done by each of the processors on your system
- OpenMP uses a so-called ***'fork and join'*** model, i.e., the code executes ***serially*** until it hits a ***parallel region*** where the code is parallelised across threads
- 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 values can be read from a common register
- OpenMP is included as a library, though it must be flagged at compile time

C

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

Fortran

```
use omp_lib
```

2 Basics of OpenMP

The fundamental building block of OpenMP is the parallel region:

C

```
#pragma omp parallel{
        // code goes here
}
```

Fortran

```
!$omp parallel
        ! code goes here
!$omp end parallel
```

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

- **Expensive** to spawn new threads
- Can be difficult to share work evenly across threads – ***load imbalance***
- 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

- The **pragmas** and **directives** that we use to define parallel regions are typically written as `#pragma omp parallel [clause]` or `!$omp parallel [clause]` in which *[clause]* defines certain qualities of the parallel regions, e.g.,

```
#pragma omp parallel num_threads(4) {
        // code here is ran in parallel by 4 threads
}
```

- 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`

- The number of threads can also be set by the environment variable:

```
export OMP_NUM_THREADS=4
```


2 Basics of OpenMP

2.2 Parallel regions

```
int main(){
        #pragma omp parallel num_threads(4){
                foo();
        #pragma omp single
                bar();
        }
}
void foo(){
        printf("function foo, using thread number%d\n", omp_get_thread_num());
}
void bar(){
        printf("function bar, using thread number%d\n", omp_get_thread_num());
}
```

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


2 Basics of OpenMP
2.3 Parallel for

- Most parallelism will be achieved with **parallel for** (C) or **parallel do** (Fortran)

C

```
#pragma omp parallel for
for (int i=0; i<100; i++){
        // code goes here
}
```

Fortran

```
!$omp parallel do
do i = 1, 100
        ! code goes here
end do
!$omp end parallel do
```

- With no clauses, the loop iterations are split up as equally as possible between the threads
- 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

2 Basics of OpenMP
Example: scheduling

- We will now work through the completed example:

  omp/examples/scheduling

- This example shows the basics of parallel for loops along with scheduling of the loop
- The first loop shows a standard OpenMP parallelised loop
- The second loop shows a scheduled parallel loop, with static scheduling and a chunksize of 2

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

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

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

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

  - Though typical real world examples are more complex!

2 Basics of OpenMP

2.5 Variable Sharing

- By default, all variables are shared between threads (and can be modified!), so in many circumstances, variables should be marked **private**
- Variables marked **private** will be newly copied to each thread, and only modifiable by that thread (increasing computational cost!). Note that the default constructor will be used (i.e. 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)**

```
double x = 9;
const double a = 10;
double b = 11;
#pragma omp parallel private(x) shared(a) firstprivate(b) default(none)
{
        printf("%f", x); // prints 0
        printf("%f", a); // prints 10
        printf("%f", b); // prints 11
}
```

Durham University

2 Basics of OpenMP
2.6 Race Conditions

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

```
int sum = 0;
for (int i = 1; i <= 10; i++){
        sum += i;
}
```

- What about in this instead?

```
int sum = 0;
#pragma omp parallel for
for (int i = 1; i <= 10; i++){
        sum += i;
}
```


2 Basics of OpenMP

2.6 Race Conditions (cont.)

We don't get the desired result! This is because updating **sum** consists of three operations:

1) Read sum
2) Update sum (sum → sum+1)
3) 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 subset of the data to compute, 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

```
int sum = 0;
#pragma omp parallel for reduction(+ : sum)
for (int i = 1; i <= 10; i++){
        sum += i;
}
```


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

This forces each thread to wait for the others defeats the point of using multiple threads

```
int sum = 0;
#pragma omp parallel for
for (int i = 1; i <= 10; i++){
        #pragma omp critical
        sum += i;
}
```

2 Basics of OpenMP
Example: race conditions

- We will now work through the completed example

omp/examples/raceConditions

- This example is summing the first 10 integers with various parallelisms
- First, no parallelism. Correct!
- Second, naively parallelise the loop. Incorrect!
- Third, use a critical to avoid the race condition. Correct but slow!
- Finally, use a reduction operation. Correct and performant!

2 Basics of OpenMP

Practical 1: race conditions

- Have a go at the practical

omp/practicals/factorial

- The practical includes a serial version of calculation a factorial: n! = n x (n-1) x (n-2) x … x 1, e.g., 4! = 4 x 3 x 2 x 1
- The aim is to write a parallel version of this function but be careful about race conditions!

2 Basics of OpenMP

2.7 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**

```
#pragma omp parallel {

        // code goes here
        #pragma omp master {

                #pragma omp task
                foo();

                #pragma omp task
                bar();
        #pragma omp taskwait

        }

}
```

2 Basics of OpenMP

Example: tasks

- We will now work through the completed example

omp/examples/tasks

- This example shows some different ways that task parallelism can be used to build quite complex parallel workflows
- The print statements indicate which threads are running which code block

2 Basics of OpenMP

Practical 2: calculation pi

- Have a go at the practical:

  omp/practicals/piCalc

- This example uses a method for computing the mathematical constant π =3.1415926535...

- The practical includes a serial version, which you can parallelise. Why not try different parallelism as used in the Mandelbrot set example:

  omp/mandelBrot

COFFEE BREAK

Durham University

MPI

Durham University

3 Basics of MPI

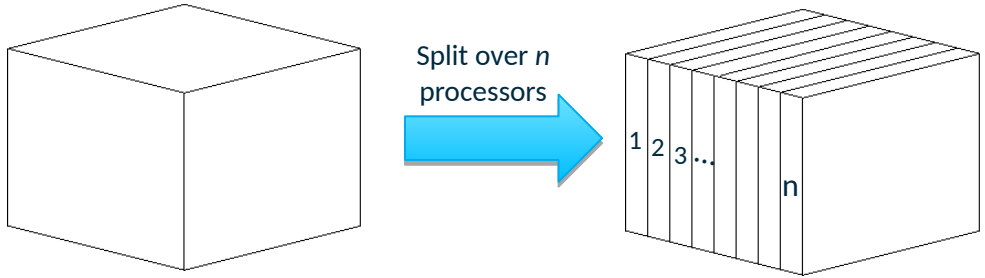
The General Message Passing Paradigm

- Processes have ***separate memory***; variables are private to each process.
- Processes exchange data via ***MPI library calls*** (send/receive and collectives).
- Single Program, Multiple Data (SPMD): *one program runs across all processes*; each process uses its rank to decide what to do.
- MPI provides a *portable, efficient, widely adopted* standard for message passing.
- Common implementations: Open MPI, MPICH, MVAPICH; also vendor-tuned builds.

3 Basics of MPI:

3.1 Goals and parallelisation approaches

- Goal: *Reduce time* to solution via strong scaling.
- Strategy: *Decompose the problem* so work runs concurrently across ranks.



- Approach 1 — ***Data/Domain Decomposition***:
  - Split data into subdomains; each rank owns a piece and exchanges halos with neighbours
  - Examples: grids/meshes, matrices, blocks of arrays
- Approach 2 — ***Functional Decomposition***:
  - Split pipeline/stages or independent tasks across ranks
  - Examples: integrals split by intervals, FFT stages, per-file processing, service roles.

3 Basics of MPI

3.2 What our first MPI program will do (C / Fortran)

- Include *headers*: **mpi.h / mpif.h**
- *Initialise* MPI: **MPI_Init()**
- Get your *rank* and *size*: **MPI_Comm_rank, MPI_Comm_size**
- Do all rank-based work (SPMD) here and communicate data between processes
  - *Print something per-rank so you see parallel execution*
- *Finalise* MPI: **MPI_Finalize**
- Pitfall: Using MPI calls before MPI_Init or after MPI_Finalize

C	FORTRAN
`#include <mpi.h>`	`include 'mpif.h'`
`MPI_Init(&argc, &argv);`	`INTEGER IERR` `CALL MPI_INIT(IERR)`
`MPI_Comm_rank(MPI_COMM_WORLD, &rank);` `MPI_Comm_size(MPI_COMM_WORLD, &size);`	`CALL MPI_Comm_rank(MPI_COMM_WORLD, rank)` `CALL MPI_Comm_size(MPI_COMM_WORLD, size)`
`MPI_Finalize();`	`CALL MPI_FINALIZE(IERR)`

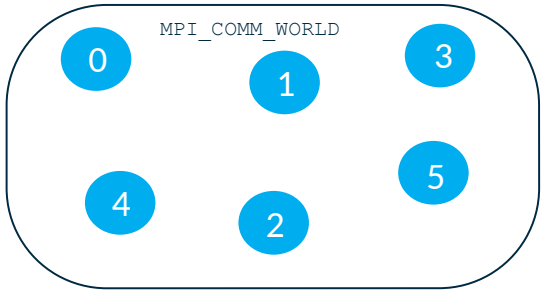
// Case sensitive…
`Error = MPI_Xxxx(parameter, …);`
`MPI_Xxxx(parameter)`
// Result is returned from the function
	! Case insensitive…
	`CALL MPI_XXXX(parameter, …, IERR)`
	
	! `IERR` returns 0 (success) or 1 (fail)

3 Basics of MPI

3.3 Communicator, rank, size

- MPI ***communicator*** defines a group of processes between which message passing can occur. By default, the communicator **MPI_COMM_WORLD** is automatically generated at initialization, and it contains *all processes* as execution begins.
  - We'll use only MPI_COMM_WORLD in all our examples today. However, new communicators can be created to group processes
- MPI ***rank*** is the unique process ID in a communicator, (**0..size-1**), which is not tied to a CPU, a core or a node.
  - Use case: "Branches logic per-rank (e.g., rank 0 reads input, others compute)."
- MPI ***size*** is the number of ranks in a communicator, **size**.

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



Durham University

3 Basics of MPI

3.4 Let's compile and run our first MPI program

After you have logged into your account on Hamilton:

- List available modules: **module avail**
- Load compilers and MPI: **module load gcc openmpi**
- To compile:

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

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

- Write and submit a job script (see on the right)
- 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 (total memory used etc.): `sacct -j jobID`

```
#!/bin/bash
#SBATCH --job-name="my-program"
#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)
             #(--ntasks can be used too)
#SBATCH -c 1 # number of cores per task
             #if you're not using threads
             #(--cpus-per-task can be used too)
module purge
module load gcc openmpi
mpirun ./myprogram
```

Practical 3: Hello World!

- Write a minimal MPI program that prints "`Hello World!`"
  - Serial template code is available for C and FORTRAN on Hamilton here:
    `mpi/practicals/helloworld/helloworld.c`
    `mpi/practicals/helloworld/helloworld.f90`
- Compile your code.
- Run it on a single processor on the login node.
- Run it on several processors in parallel *via the batch queue* using a job script
  `mpi/practicals/helloworld/job.sh`
- Modify the code (with an *if*-statement) so that only MPI rank 0 prints: "`Hello World!`"
- Adjust the code so that each MPI rank prints information about itself, for example:
  "`Hello World! I am rank # of size #.`"

Practical 3: Review

C

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

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

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &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
```

For solutions, see also: `mpi/solutions/helloworld`

Durham University

4 Point-to-point communications in MPI

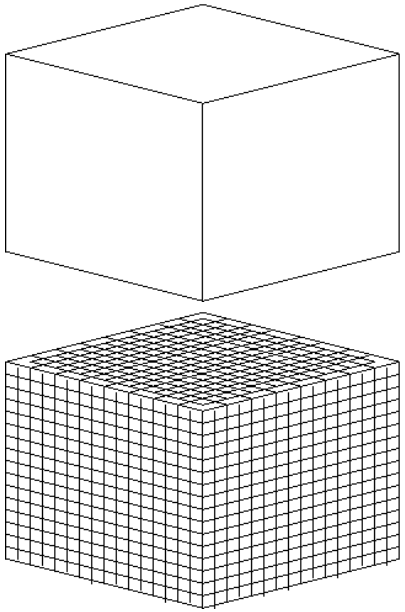
4.1 Contents

- Messages
  - Data types
- Communication modes and completion
  - Sends: synchronous / buffered / ready / standard
  - Receive
  - Success criteria
  - Wildcarding and communication envelope
- Message order preservation
- Combined send and receive

4 Point-to-point communications in MPI

4.2 CFD example

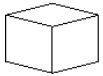
- This is a representation of a domain for a piece of CFD software that solves the equations of fluid dynamics to evolve a fluid with time
- The domain is broken down into a number of cells (e.g. 20 x 20 x 20: 8000 cells)
- If solving Euler's equation takes 1 second to evolve the fluid in a cell by one 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*".



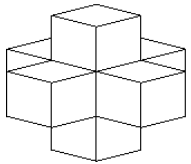
4 Point-to-point communications in MPI

4.3 Data for communication

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.

4 Point-to-point communications in MPI

4.4 Data types

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

4 Point-to-point communications in MPI

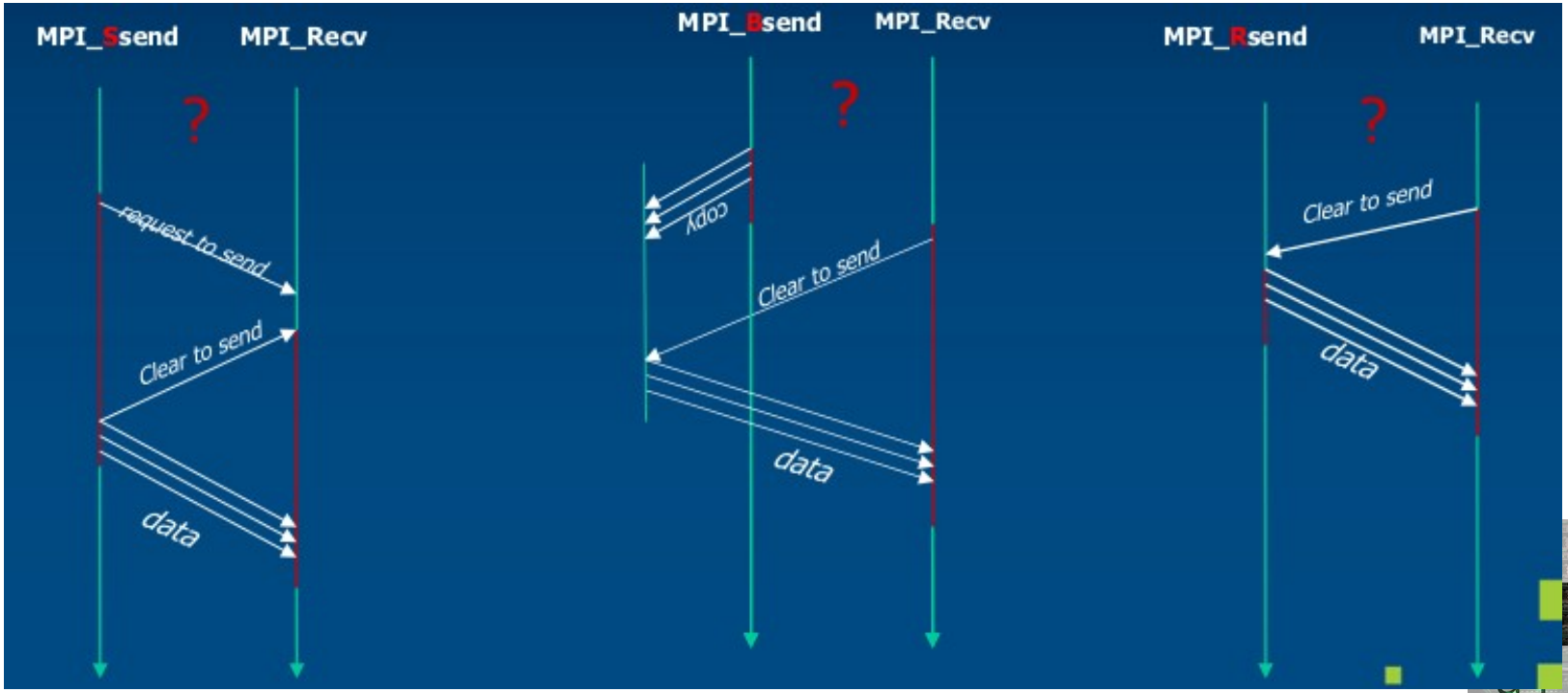
4.5 Sender mode

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.**

4 Point-to-point communications in MPI

4.6 Sequence diagram for sender modes

Sender modes - explained



4 Point-to-point communications in MPI

4.7 Synchronous Send command

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

Arguments:

- `buf`: start of data to be sent
- `count`: number of elements to send
- `datatype`: type of data
- `dest`: MPI rank of the receiving process
- `tag`: message identifier
- `comm`: communicator
- `ierr`: integer error code (Fortran only)

e.g. sending `1` integer `data1` to `rank=2` (`tag=100`):

`MPI_Ssend(&data1, 1, MPI_INT, 2, 100, MPI_COMM_WORLD);`	`CALL MPI_SSEND(data1, 1, MPI_INTEGER, 2, 100, MPI_COMM_WORLD, ierr)`

4 Point-to-point communications in MPI

4.8 Receive command

C	FORTRAN
`MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`	`CALL MPI_RECV(buf, count, datatype, source, tag, comm, status, error)`

Arguments:

- `buf`: starting location of receiving buffer
- `count`: number of elements to receive
- `datatype`: type of data
- `source`: MPI rank of the sending process
- `tag`: message identifier
- `comm`: communicator
- `status`: message envelope (of size `MPI_STATUS_SIZE`)
- `ierr`: integer error code (Fortran only)

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

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

4 Point-to-point communications in MPI

4.9 An example for Synchronous Send

C

```
#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();
}
```

FORTRAN

```
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
```

4 Point-to-point communications in MPI

4.10 `status` communication envelope

Like a letter, a message has more than just the text: sender address (*source*), reference number (*tag*), and number of pages (*element count*). MPI returns this envelope information in `status` on the receiver.

In C, `status` is a `struct` with named fields; in Fortran, it's an `INTEGER` array 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)`

4 Point-to-point communications in MPI

4.11 Wildcarding

What wildcarding is (receive-side only):

- You can accept a message without specifying the exact sender and/or tag
- Source wildcard: set source = `MPI_ANY_SOURCE`
- Tag wildcard: set tag = `MPI_ANY_TAG`

What you learn from status:

- The actual sender and tag are returned in status on completion of `MPI_Recv` (or `MPI_Probe`, *see the next slide*)
- Use `MPI_Get_count(status, datatype, &count)` to get how many elements arrived

Typical usage patterns:

- Many-to-one intake: collect results from whichever worker finishes first
- Event loop: `MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, ...)` then dispatch on `status.MPI_TAG/status.MPI_SOURCE`

C

```
MPI_Status st;
int n;

MPI_Recv(buf, maxn, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &st);

MPI_Get_count(&st, MPI_INT, &n);

printf("Got %d ints from rank %d (tag %d)\n", n,
st.MPI_SOURCE, st.MPI_TAG);
```

FORTRAN

```
integer :: status(MPI_STATUS_SIZE)
integer :: n

call MPI_Recv(buf, maxn, MPI_INTEGER, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)

call MPI_Get_count(status, MPI_INTEGER, n, ierr)

write (*, '(A,I0,A,I0,A,I0,A)') 'Got ', n, ' ints from rank ',
status(MPI_SOURCE), ' (tag ', status(MPI_TAG), ')'
```

4 Point-to-point communications in MPI

4.12 Probe and Get_count commands

What: Non-consuming inspection of the next matching message; returns status with `MPI_SOURCE` and `MPI_TAG`
Why: Determine message size via `MPI_Get_count` and allocate appropriately; drive event loops

C	FORTRAN

`MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
`MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`
	`MPI_Probe(source, tag, comm, status, ierr)`
	
	`MPI_Get_count(status, datatype, count, ierr)`

Arguments:

- `source`: rank of expected sender, or `MPI_ANY_SOURCE`
- `tag`: expected tag, or `MPI_ANY_TAG`
- `comm`: communicator (e.g., `MPI_COMM_WORLD`)
- `status`: filled with metadata about the matched message (*source*, *tag*, *error*)
- `datatype`: the datatype you intend to interpret the message as (e.g., `MPI_INT`, `MPI_DOUBLE`); used to compute how many elements arrived
- `count`: on return, set to the number of received elements of the given datatype (or `MPI_UNDEFINED` if the message size is not an integral multiple of the datatype)
- `ierr`: MPI error code (e.g., `MPI_SUCCESS`) which is simply returned from the function in C

Note:

- *Non-consuming*: the message remains in the queue; use `MPI_Get_count(&status, datatype, &count)` to get element count, then `MPI_Recv(...)` to actually receive it

Example:

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

MPI_Get_count(&status, MPI_INT, &n);
```

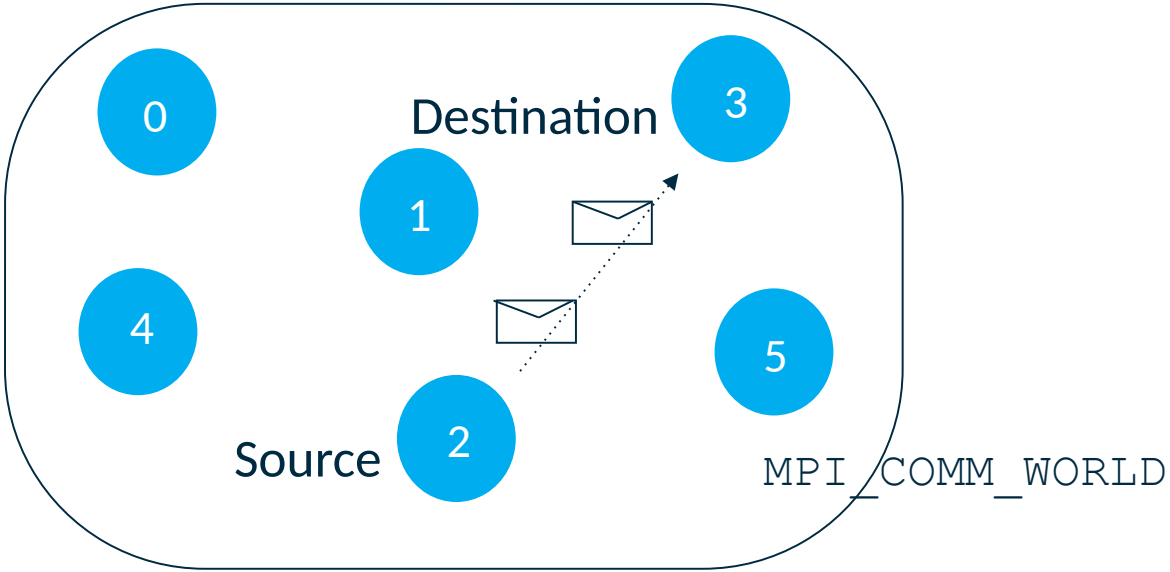
```
call MPI_Probe(data2, 1, MPI_INT, 1, 100, MPI_COMM_WORLD,
status, error)

call MPI_Get_count(status, MPI_INTEGER, n, ierr)
```

4 Point-to-point communications in MPI

4.13 Order of messages

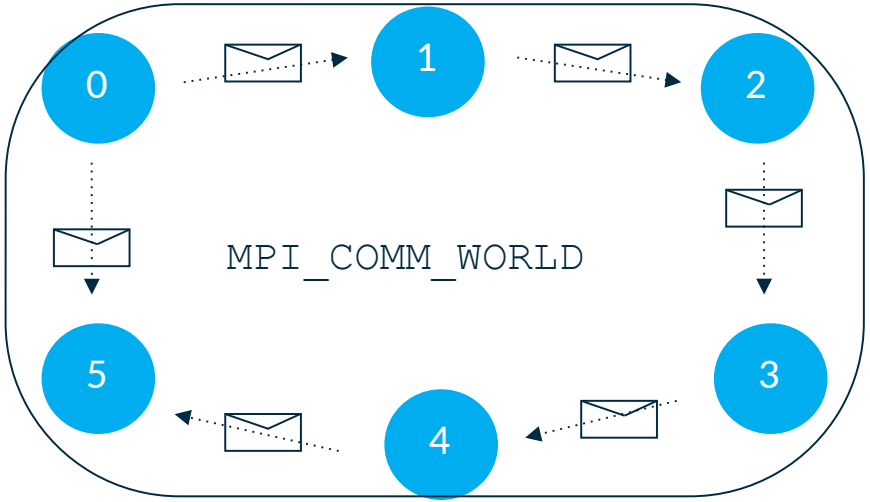
The order of messages is preserved:



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

4 Point-to-point communications in MPI

4.14 Deadlock



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.

4 Point-to-point communications in MPI

4.15 Non-blocking 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.

4 Point-to-point communications in MPI

4.16 Combined command `MPI_Sendrecv`

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.

4 Point-to-point communications in MPI

4.17 MPI_Sendrecv syntax

C	FORTRAN
`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);`	`MPI_Sendrecv(sendbuf, sendcount, sendtype, destination, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierr)`

Arguments:

- `sendbuf` and `recvbuf`: start of data to be sent and received, respectively
- `sendcount` and `recvcount`: number of elements to send and receive, respectively
- `sendtype` and `recvtype`: type of data sent and received, respectively
- `destination` and `source`: MPI rank of the receiving and sending process, respectively
- `sendtag` and `recvtype`: identifier/label of the sent and received message, respectively
- `comm`: communicator
- `status`: message envelope (of size `MPI_STATUS_SIZE`)
- `ierr`: integer error code (Fortran only)

Example in Fortran (showing also data types of variables):

```
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)
```


4 Point-to-point communications in MPI

4.18 `MPI_PROC_NULL`, `MPI_Sendrecv_replace`

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 4: 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:
      `mpi/practicals/pingpong/pingpong.c`
      `mpi/practicals/pingpong/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 4: Review

The principles of node pair communication (in C)

```
   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:

```
mpi/solutions/pingpong
```

Durham University

COFFEE BREAK

Durham University

5 Collective communications in MPI

5.1 Contents

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

5 Collective communications in MPI

5.2 Introduction

Collective communication involves a group of processes.

Called by *all* processes in a communicator.

Examples:

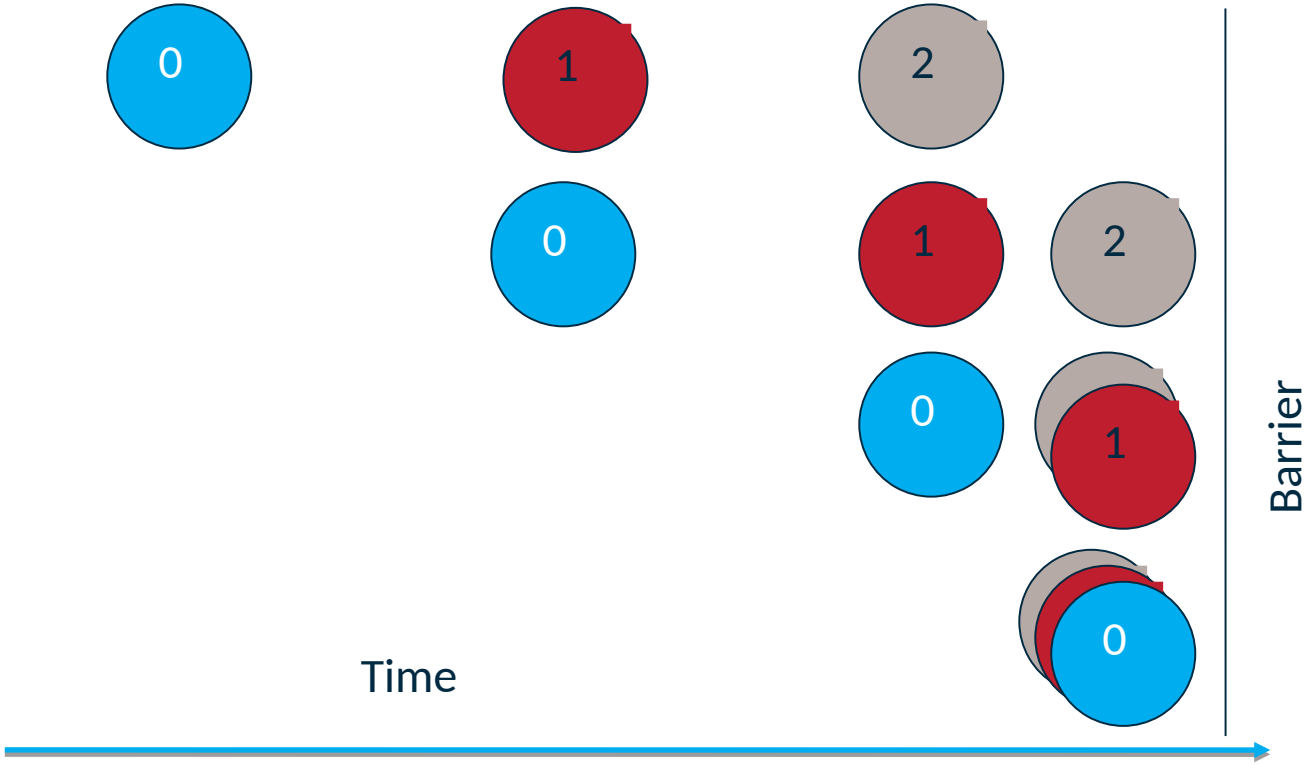
- Broadcast, scatter, gather (Data Distribution)
- Global sum, global maximum, etc. (Reduction Operations)
- Barrier synchronisation

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

5 Collective communications in MPI

5.3 Barrier Synchronisation



5 Collective communications in MPI

5.4 Barrier Synchronisation syntax

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

C:

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

Fortran:

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

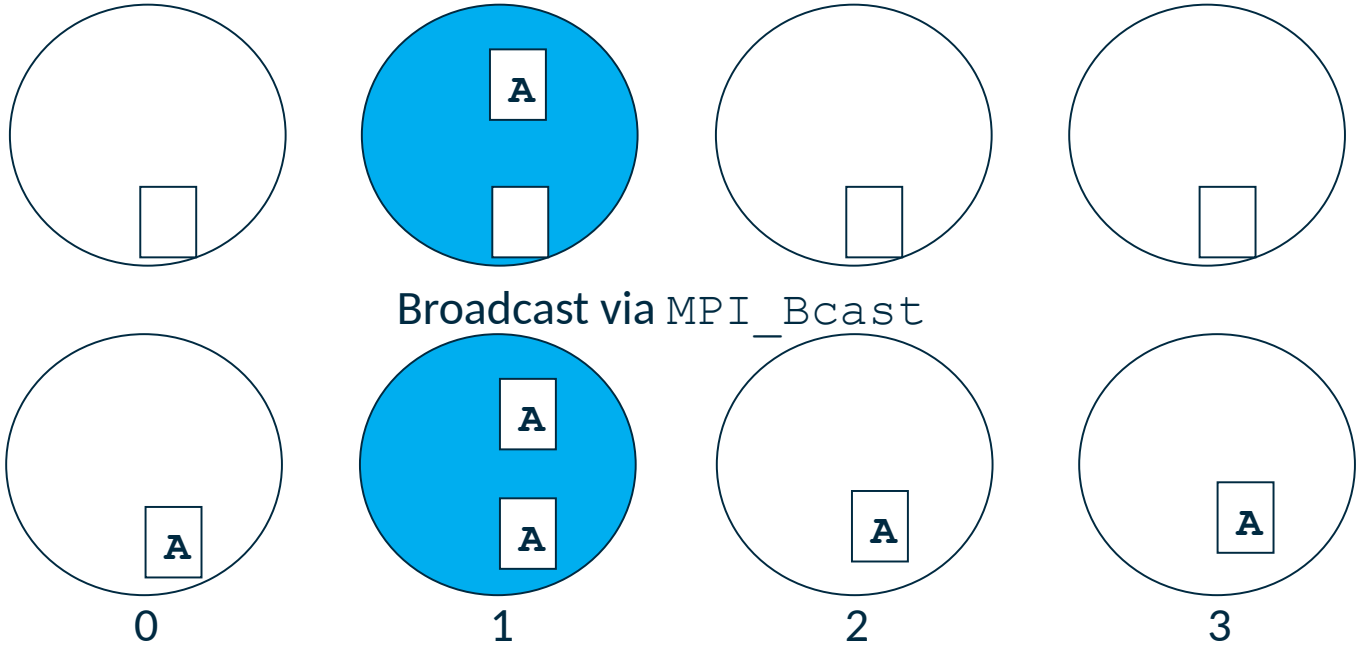
Note:

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

5 Collective communications in MPI

5.5 Broadcasting

Duplicates data from one process to all other processes in communicator group



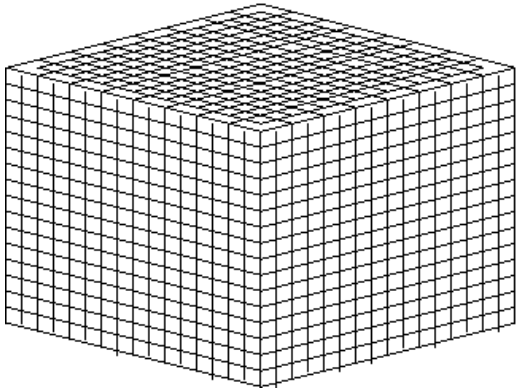
As with all collective processes, must be called simultaneously by every process

5 Collective communications in MPI

5.6 CFD example again

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.

5 Collective communications in MPI

5.7 Broadcast syntax

C:

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

Fortran:

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

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

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

5 Collective communications in MPI

5.8 Gathering and scattering routines

- `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.

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

5 Collective communications in MPI

5.9 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.

5 Collective communications in MPI

5.10 Global reduction operations

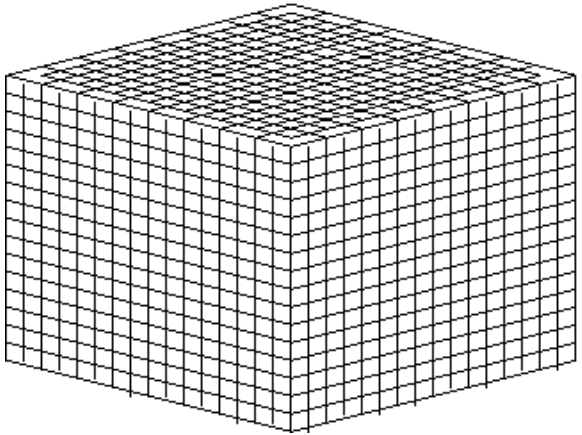
MPI name	Function	C	FORTRAN
MPI_MAX	Maximum		MAX(a_1… …a_n)
MPI_MIN	Minimum		MIN(a_1… …a_n)
MPI_SUM	Sum	+	+
MPI_PROD	Product	*	*
MPI_LAND	Logical AND	&&	.AND.
MPI_BAND	Bitwise AND	&	
MPI_LOR	Logical OR	||	.OR.
MPI_BOR	Bitwise OR	|	
MPI_LXOR	Logical exclusive OR	!=	.NEQV.
MPI_BXOR	Bitwise exclusive OR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

5 Collective communications in MPI

5.11 CFD example once again

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.

5 Collective communications in MPI

5.12 `MPI_Allreduce` syntax

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

C:

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

Fortran

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

For example, in our CFD case:

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

5 Collective communications in MPI

5.13 `MPI_Scan` syntax

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

C:

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

Fortran:

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

Practical 5: collective communications

1. Collective communication with `MPI_Allreduce`
   - Compute the global sum of all ranks of the processes using MPI global reduction
   - Template:
     `mpi/practicals/collective/collective.c`
     `mpi/practicals/collective/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

Practical 5: Review

For the complete answers, please consult the solutions:

```
mpi/solutions/collective
```

Durham University

Good luck writing your own parallel code!

Thank you for attention!

Feedback
https://forms.office.com/e/hQ0Ni5brPU?origin=lprLink

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

Scan the QR or use link to join

Durham University

Old slides from here

1. Brief intro to HPC and parallel programming models

1.5 HPC terminology (cont.)

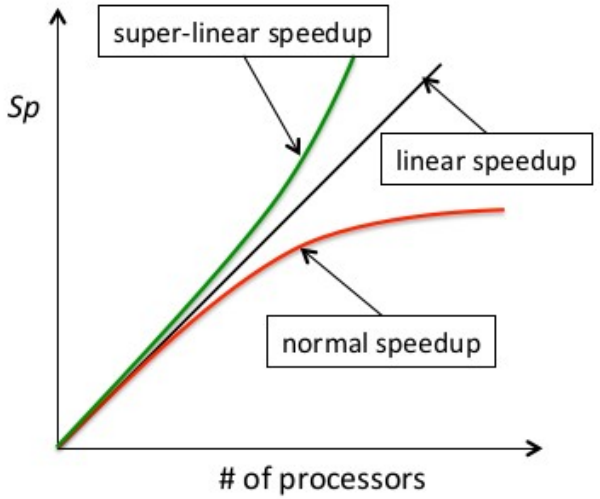
- To characterise performance of computing, processor speed is measured in *floating point operations per second* (*FLOPS*, *MFLOPS*, *GFLOPS*, etc.)
  - There are two speeds: '*peak*' – the best in theory; and '*sustained*' - on a benchmark or relevant user code. The latter can be anything between ~0% 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: $S_p = \frac{T_s}{T_p}$
  - *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 speed is limited by the sequential part of the program



- 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!

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

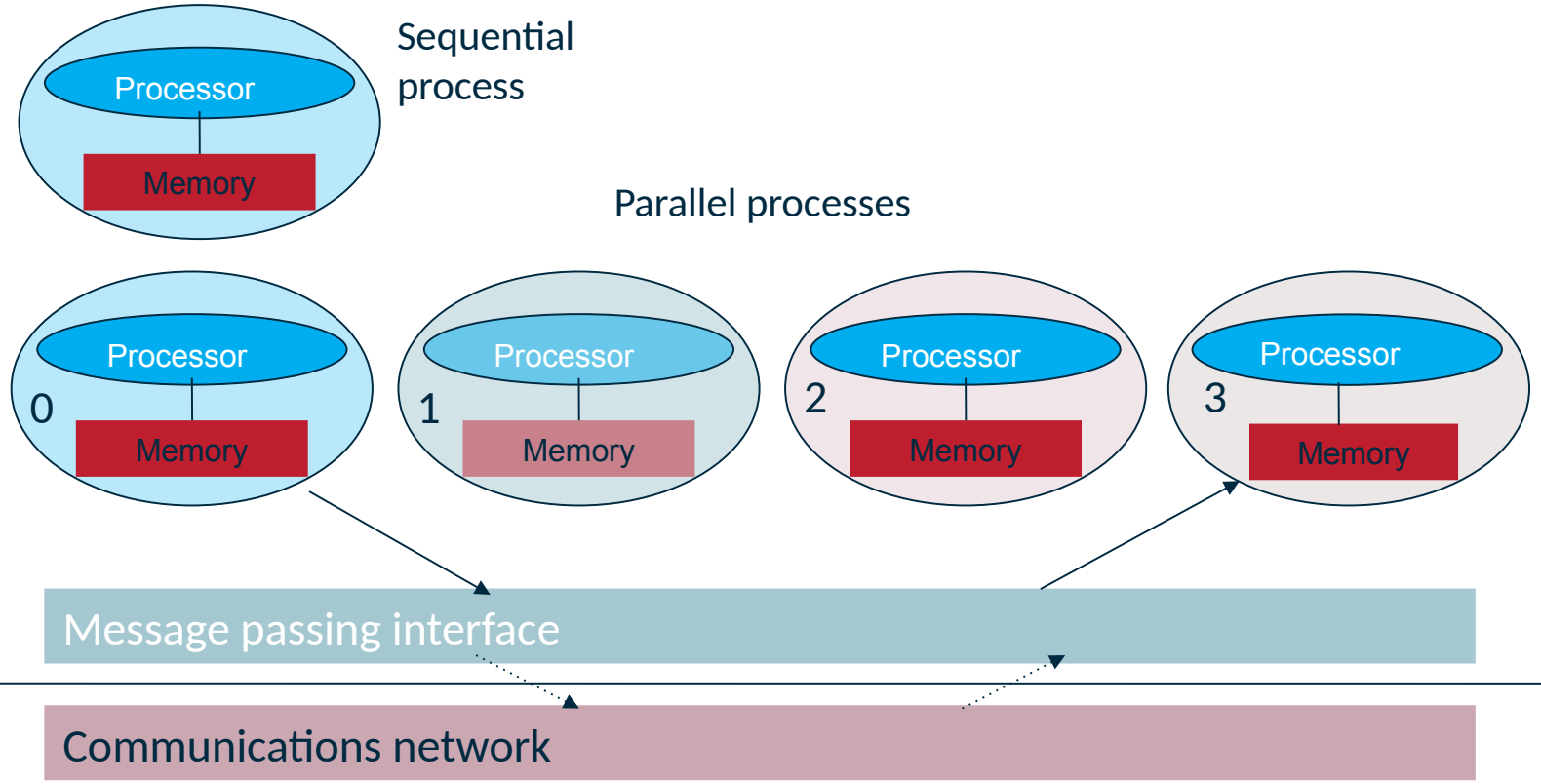
- To ease clarity, separate out communication routines
  - Separate file
  - Dummy library for serial code
  - Avoids explicit MPI references in main code

- 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`

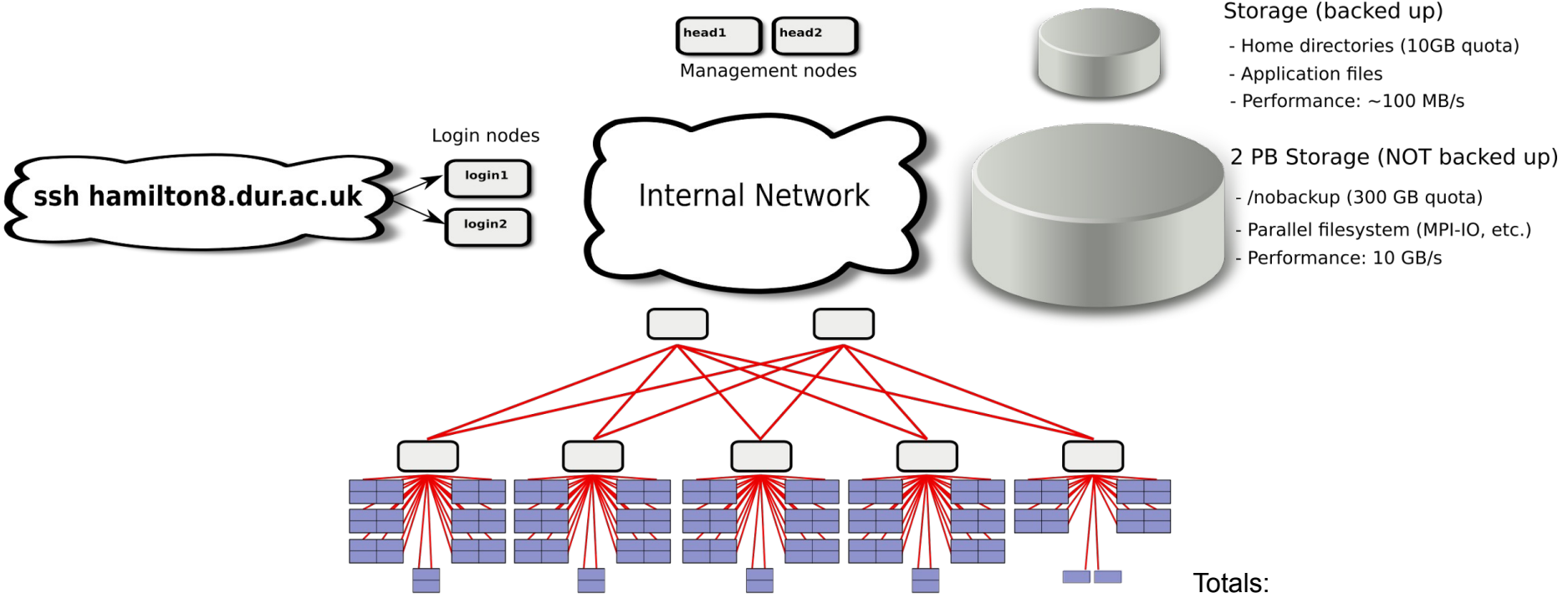
1. Brief intro to HPC and parallel programming models

1.1 Message-Passing Paradigm



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

Setup

Google Colab

- Access
  - Sign in with your Google account at Google Colab.

- Create a Non-Root User
  - By default, Google Colab runs as root. This adds a user without a password and no additional user info (non-root usage):

```
!adduser --disabled-password --gecos "" colabuser
```

- Environment setup
  - Move to the /content directory

```
%cd /content
```

  - Clone the repository containing the example code:

```
!git clone https://github.com/DurhamARC/BasicParallelProgramming.git
```

  - Navigate to the example MPI directory:

```
%cd BasicParallelProgramming/mpi/solutions0
```

- Compile & run
  - Compile a small MPI/OpenMP program

```
!mpicc helloworld.c -o helloworld
```

  - Run the program as a Non-Root User (`--host localhost:2` ensures it runs on the current machine with 2 processes):

```
!sudo -u colabuser mpirun -np 2 --host localhost:2 ./helloworld
```