



Durham  
University

# A very brief introduction to parallel programming with a supercomputer

Dmitry Nikolaenko  
Research Software Engineer  
Advanced Research Computing

October 20, 2022

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

# Course Outline

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

Aims of the course:

- Get introduced to parallel programming;
- 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

- 10:00-10:15 - Introducing myself/yourself
- 10:15-10:45 – An introduction to parallel computing
  - 10:45-11:00 – Practical session 1: “Hello World!”
- 11:00-11:15 – Break time from screen aka “coffee break”
- 11:15-11:45 – Point-to-point communications
  - 11:45-12:00 – Practical session 2: “Ping pong!”
- 12:00-12:15 - Break time from screen aka “coffee break”
- 12:15-12:45 – Non communications, I/O & general advice
  - 12:45-13:00 – Practical session 3: “Collective comm”



COFFEE BREAK



COFFEE BREAK



# Preliminaries

What is your background?

- Who are you? Where are you from in the University?
- What is your experience of high performance computing?
- How do you plan to use parallel programming in your research?
  - Are you currently trying to develop parallelised software?
  - Are you having any problems in this development?



# 1. Introduction and MPI basics

## Intro to MPI

### Parallel computing terminology

- Nodes, sockets, cores and processors
  - A physical device within a network of other tools that's able to send, receive, or forward information (a “node”)
  - A hardware package (a “socket”) within a node
  - An independent execution unit (a “core”) within a socket
  - A largely dependent unit (a “thread”) within that core that is more logical abstraction than anything else.
  - A desktop machine used to contain a single Central Processing Unit (CPU).
  - Nowadays, we are used to multi-CPU desktop machines, with ‘multi-core’ CPUs, often with 32 or more processing ‘cores’.
  - In rackable (as opposed to desktop) format, these are multi-CPU, multi-core ‘nodes’.



# 1. Introduction and MPI basics

## Intro to MPI

### Parallel computing terminology

- Processes per core
  - You can run multiple processes per core.
  - Classical MPI parallelisation is a single MPI process per core.
  - Today, we will only talk about a classical MPI parallelization scenario and I will (probably) use the words 'core' and 'process' interchangeably.
  - Hybrid MPI/openMP typically balances an MPI process per shared memory area (e.g. multi-CPU node or single CPU) with openMP threads across all the cores with access to that shared memory; *the balance should be tuned for best performance!*



# 1. Introduction and MPI basics

## Intro to MPI

### Parallel computing terminology

- Parallel computing
  - As opposed to serial computing on a single core/processor.
  - Simultaneous use of more than one core/processor to solve a given problem.
- Massively parallel computing (MPP) at exascale
  - Simultaneous use of  $10^5$  or more cores/processors
- Distributed computing
  - use of a network of cores/processors to solve a given problem.
- Parallelisation
  - Turning a serial computation into a parallel one across multiple cores/CPU's and distributed memory as opposed to threading on a shared-memory multi-core CPU.



# 1. Introduction and MPI basics

## Intro to MPI

### Parallel programming

- Most parallel programs are in a standard high-level language, with calls to a parallel library and/or using compiler directives to facilitate the parallelism.
- Most commonly still in C/C++ and FORTRAN.
- Compiled using wrappers to the standard compilers, ideally pre-optimized for the specific architecture.
- If compiler directives are used, appropriate flags will also be needed.
- The wrappers automatically link in the parallel libraries so you don't have to do manually.
- Bindings now exist that extend MPI support to other languages by wrapping an existing MPI implementation, such as MPICH or OpenMPI:
  - e.g. for Java, Python (import mpi4py) and R.





# 1. Introduction and MPI basics

## Intro to MPI

### Processor speeds

- For scientific use, speed is measured in floating point operations per second (FLOPS).
- This is the theoretical number of adds/subtracts/multiplies per second.
- MegaFLOPS, GigaFLOPS, TeraFLOPS, PetaFLOPS, ExaFLOPS, ...

There are two speeds:

- The 'peak' is the best the core/CPU/node can do in theory
  - A romantic speed – rarely achieved in practice!
- The 'sustained' speed on a benchmark or relevant user code.
  - This is a far more useful measure for us!
  - It can vary considerably, dependent on not only software, but also importantly the hardware and interconnections between hardware.
  - It can be anything between ~0% and ~80% of 'peak' speed,



# 1. Introduction and MPI basics

## Intro to MPI

### Theoretical peak speed

- A compute node on **Hamilton8** has a theoretical peak speed in FLOPS:
  - $2.0 \text{ GHz} * 2 \text{ CPUs} * 64 \text{ cores} * 16 \text{ FLOPS/cycle} = \underline{4096 \text{ GFLOPS}}$
  - HPL benchmark on a Hamilton8 node (showing 77% efficiency):  
WC01C2C2 155904 232 4 8 799.41 3.1602e+03
- A compute node on **Hamilton7** has a theoretical peak speed in FLOPS:
  - $2.2 \text{ GHz} * 2 \text{ CPUs} * 12 \text{ cores} * 16 \text{ FLOPS/cycle} = \underline{844.8 \text{ GFLOPS}}$
  - This takes a single precision (8-bit) calculation and assumes complete AVX2 vectorization (256-bit; 8x16), fused-multiply-add (FMA) and the cores at 2.2GHz simultaneously (typically only a single core per CPU can 'boost' to 2.9GHz).
  - An HPL benchmark result from a Hamilton7 node (showing 93% efficiency):  
cn7018.hpc.dur.ac.uk.out.0:WR11C2R4 72192 192 4 6 316.81 7.9175e+02



# 1. Introduction and MPI basics

## Intro to MPI

### Interconnection: bandwidth and latency

- Used to characterize the performance of data transfer e.g. transferring data from RAM to core at one extreme to transferring from node to node at the other.
- Both need to be well optimized for good performance across a range of applications

Bandwidth is the rate at which data can be transferred (*the higher the better*)

- Wide range – from Kbits/s to fast RAM and cache memory (Gbits/s)

Latency is the start-up time for data transfer (*the lower the better*)

- Again, a wide range, from nanoseconds for L1 cache (a few clock cycles) to several microseconds for RAM and milliseconds or longer for ethernet networks.



# 1. Introduction and MPI basics

## Intro to MPI

### Why does the interconnection between processors matter?

- For example, consider a supercomputer with many fast CPUs, but slow interconnect
  - It has a high peak performance (peak per CPU x number of CPUs)
  - But a low sustained performance for typical users using the interconnect
- Now consider a smaller system, where the design principle was a high performance (i.e. large bandwidth, low latency) interconnect, but with mid-market CPUs
  - This system has a lower peak performance (it won't crown the TOP500!)
  - But crucially it has a far higher sustained performance for a range of typical users
  - It's possibly faster for a given user problem than a much bigger system with poor interconnect
  - These smaller systems are also less expensive – so tend to be favoured for Tier 3 (single institution) and Tier 2 (regional consortium) HPC systems.



# 1. Introduction and MPI basics

## Intro to MPI

### A comparison of interconnections...

- Typical gigabit ethernet (University network)
  - up to 125Mbit/s bandwidth, 100 microsecond latency
- Infiniband in ~2014
  - Up to 40Gbit/s bandwidth, typically a few microseconds latency
- Truescale QDR Infiniband (Hamilton6) – 56Gbit/s bandwidth
- Intel Omni Path (Hamilton7) – 100Gbit/s bandwidth
- Hamilton 8 – 200Gbit/s Infiniband interconnect
  - Latency hasn't dropped any more and remains around a few microseconds

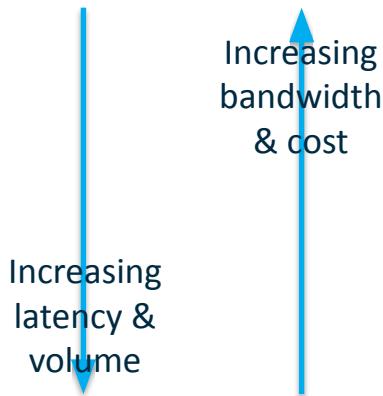


# 1. Introduction and MPI basics

## Intro to MPI

### Memory and storage

- Measured in kbyte (KB), megabytes (MB), gigabytes (GB), terabytes (TB)
- Speed of memory is finally comparable processor speeds
  - e.g. DDR4 now approaches 4400MHz – faster than the ‘cores’
  - “Next gen” AMD EPYC “Genoa” goes to DDR5 – **5200 MHz!**
- Hierarchy – fast/expensive/small to slow/cheap/huge
  - Registers (~64-bit, 8 byte) – 2 clock cycles to fetch & return
  - Cache (L1-L4; KB-MB) – 4/8/16/32 clock cycles
  - RAM (GB-TB) – 64/128/256 clock cycles <<< 1 microsecond
  - SSD (GB-TB) – 35 to 100 microseconds to access
  - HDD (TB-?) – milliseconds to access
  - Tape (TB-PB-EB!) – days to access!



# 1. Introduction and MPI basics

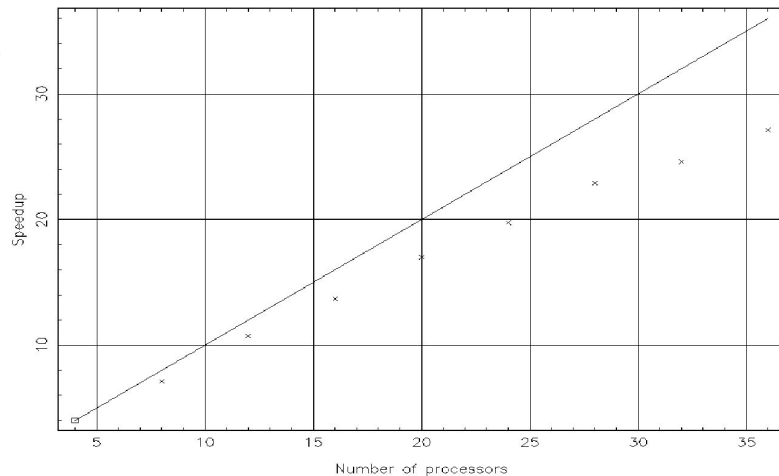
## Intro to MPI

### Algorithmic speedup

- The numerator becomes the time for the parallel algorithm on a single process:

$$S_p = \frac{T_{\text{parallelsingle}}}{T_{\text{parallelmultiple}}}$$

- Be aware of which definition is being used – the numerator is almost always greater with this latter definition, since parallel algorithms generally contain extra operations to accommodate communication and synchronisation.
- The ideal is to produce a linear speedup



# 1. Introduction and MPI basics

## Intro to MPI

### Scalability

- The ability of a program to exhibit good speedup on a large number of cores.
- Good scalability requires
  - Minimising communication
  - Effective distribution of tasks or data, resulting in good “*load balancing*”.
- Strong scaling
  - User wishes to solve *a given problem in a shorter time*.
  - Problem size is *fixed*, time taken to solution reduces with additional processors.
  - Ideally time taken **reduces in direct proportion to number of processes used**.
- Weak scaling
  - User wishes to solve *a bigger problem in the same time*.
  - Problem size *scales* with number of processes
  - Ideally time taken is **constant** and problem **scales directly with number of processes used**.





# 1. Introduction and MPI basics

## Intro to MPI

### Amdahl's Law

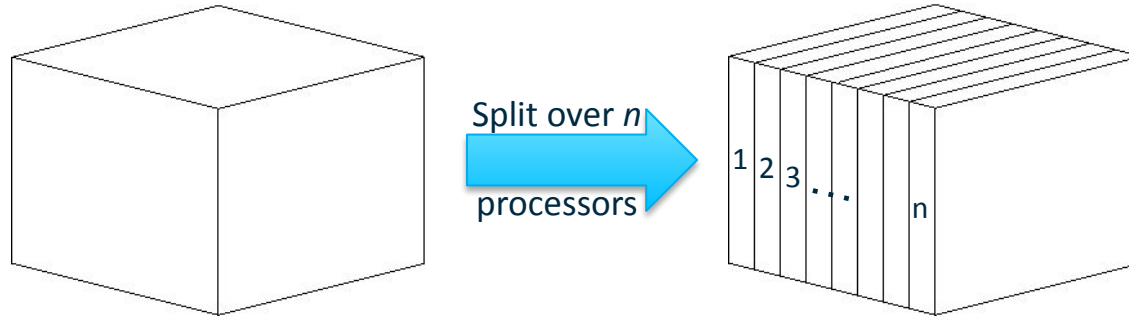
- No parallel programming course is complete without Amdahl's Law!
- Used to estimate potential speedup.
- Expresses that the potential speedup is limited by the sequential part of the program.
- For example, if 5% of the algorithm is serial, then the maximum possible speedup is  $1/0.05 = 20$ . Any parallel program will be at best 20x faster than the serial version.
- **BUT** this omits possibility of new parallel algorithms with smaller  $f$ , use of higher speed and memory (e.g. cache only) and that the time spent in serial execution is an ever decreasing percentage of the total time as the problem size is increased
- In practice, Amdahl's Law has little relevance
  - Parallel solutions exist for almost all computational problems
  - Scalability is usually limited by communications, latency, idling / load balancing



## 2. Parallel programming to achieve strong scaling

### Intro to 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:



# 2. Parallel programming to achieve strong scaling

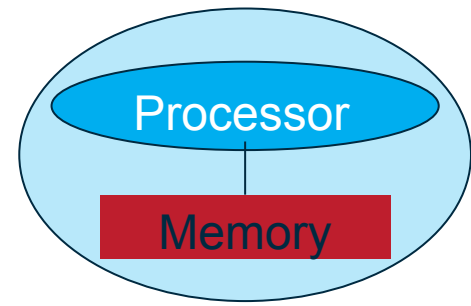
## Intro to MPI

### Load balancing

- Whichever approach of decomposition is adopted, to achieve strong scaling, roughly similar amounts of work should be performed by each process
- *Static* load balancing
  - Typically there are the same number of tasks as processes
  - Balanced before execution, either before compilation or at start time
- *Dynamic* load balancing
  - Typically there are a much larger number of tasks than processes
  - Tasks sit in a pool during execution waiting to be picked up by the next available process
- In recent times, dynamic balancing has now begun to replace classical examples of static load balancing (e.g. splitting a grid) as it can often scale better to larger numbers of processors (to exascale).
- Good load-balancing and efficient communication can clearly all be ruined by:

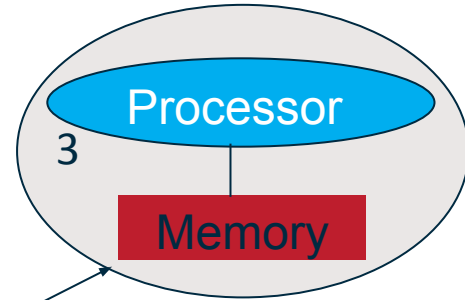
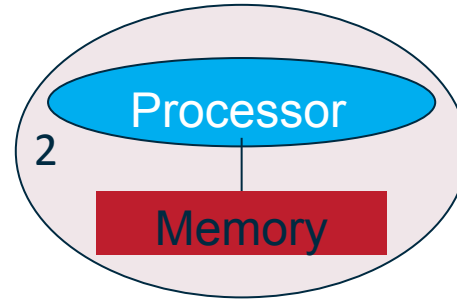
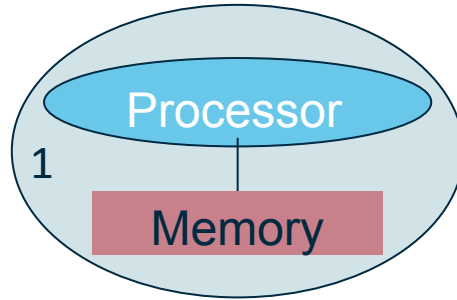
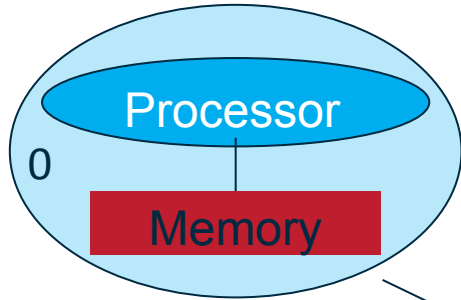
Poor process placement!





Sequential  
process

Parallel Process



# The Message-Passing Paradigm



# 2. Parallel programming to achieve strong scaling

## Intro to MPI

### The General Message Passing Paradigm

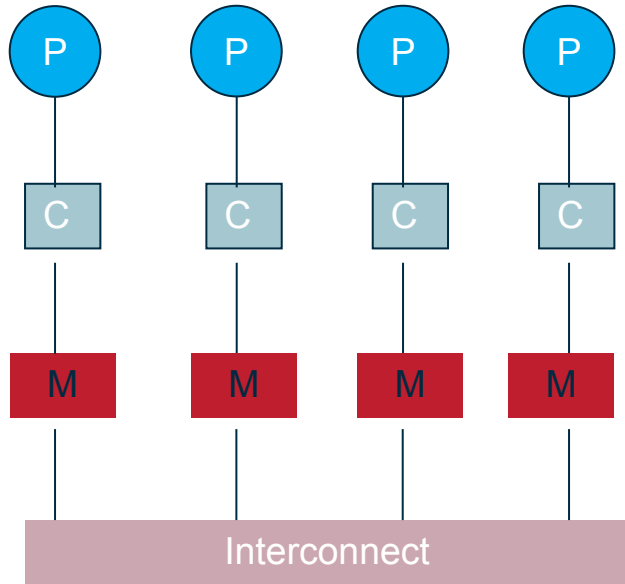
- All variables are private to each process. Values of variables are held in local memory
  - a distributed memory parallel computer
- Processes communicate via special subroutine calls to an external library
- Typically:
  - Communications are written in a conventional sequential language
  - A single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



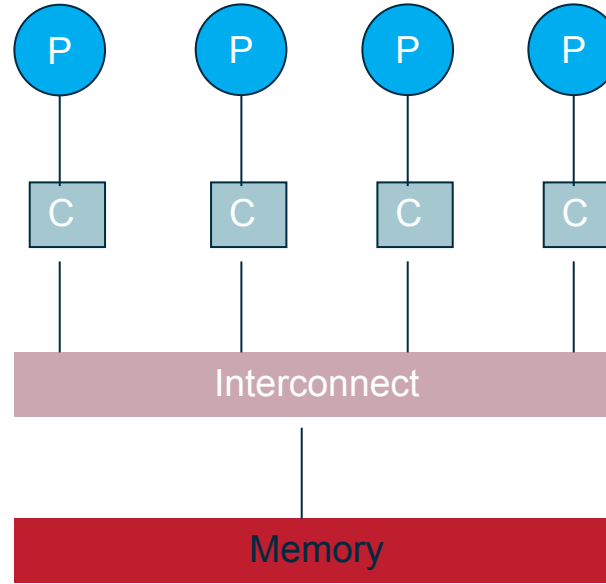
# 2. Parallel programming to achieve strong scaling

## Intro to MPI

### Machine architecture and the General Message Passing Paradigm



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)

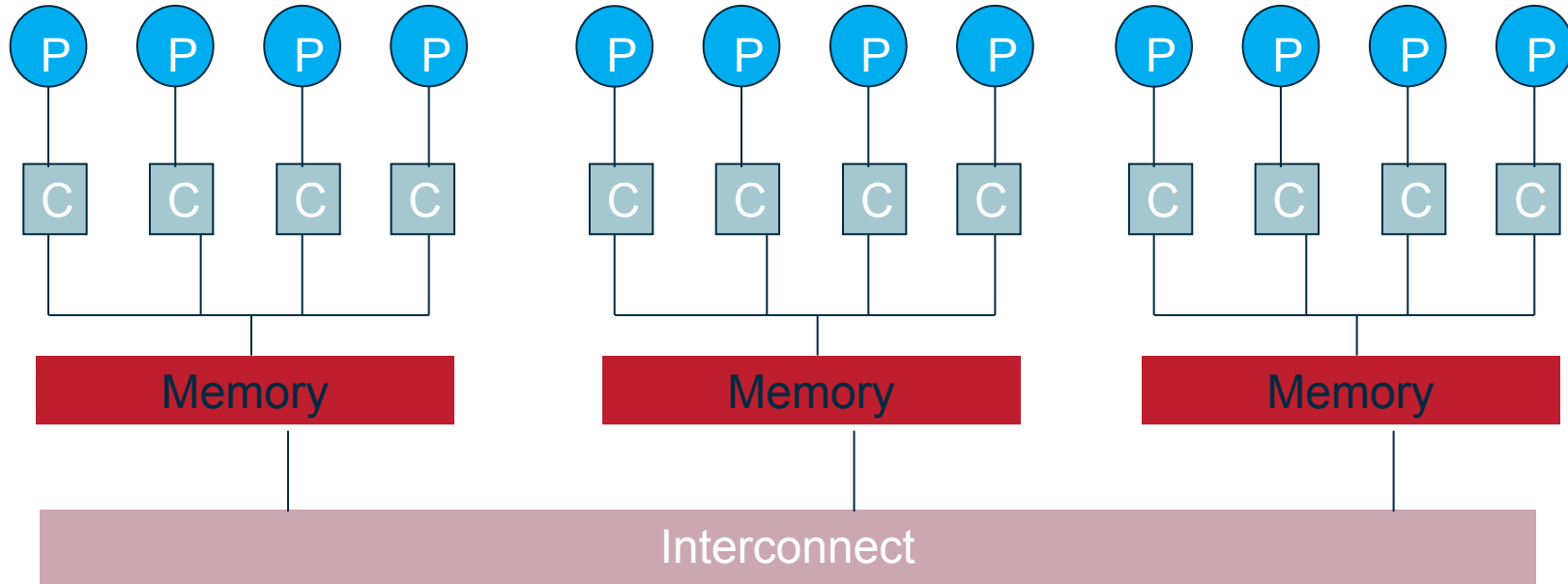


## 2. Parallel programming to achieve strong scaling

### Intro to MPI

#### Machine architecture and the General Message Passing Paradigm

- Commonly now find shared memory clusters (e.g. Hamilton etc.)

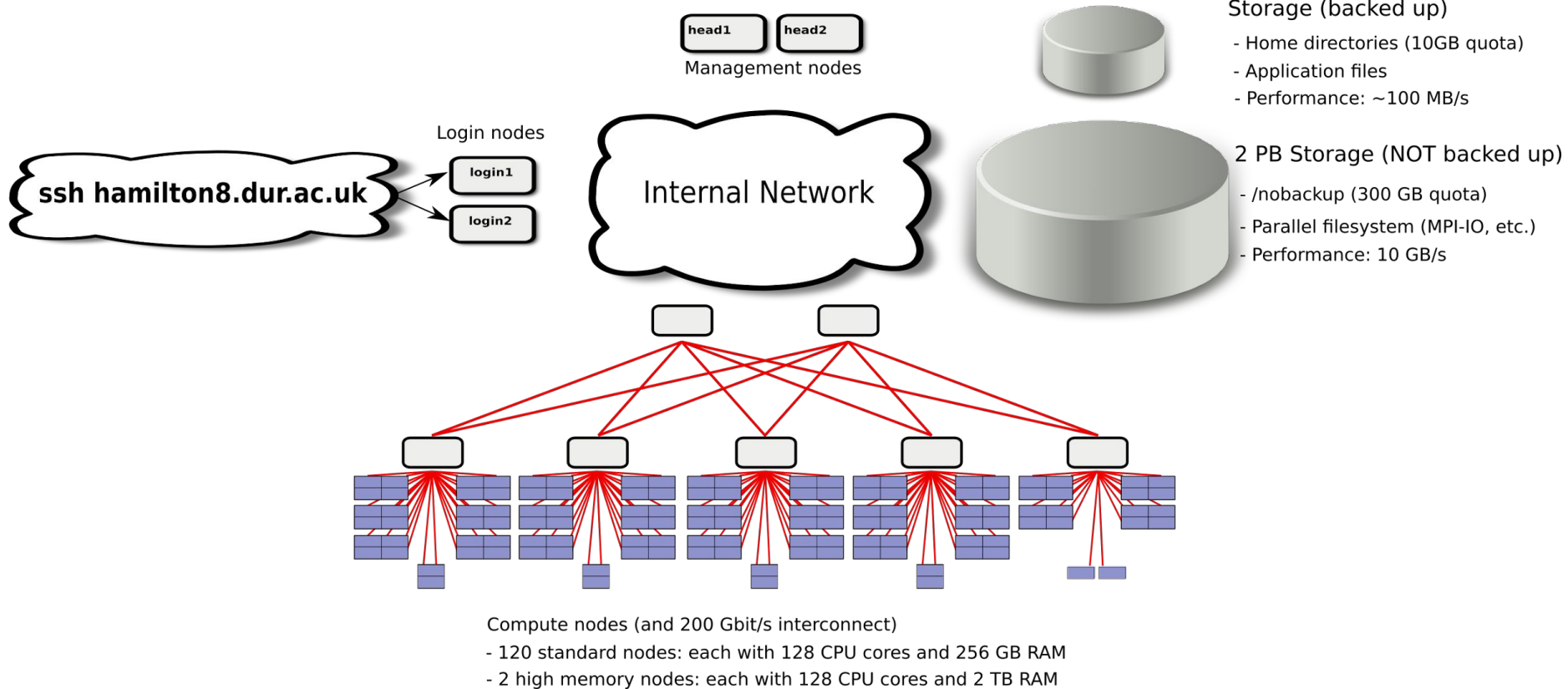


e.g. SGI origin, HPC-x architecture, Hamilton!

Will use both memory/interconnect to communicate between processes.

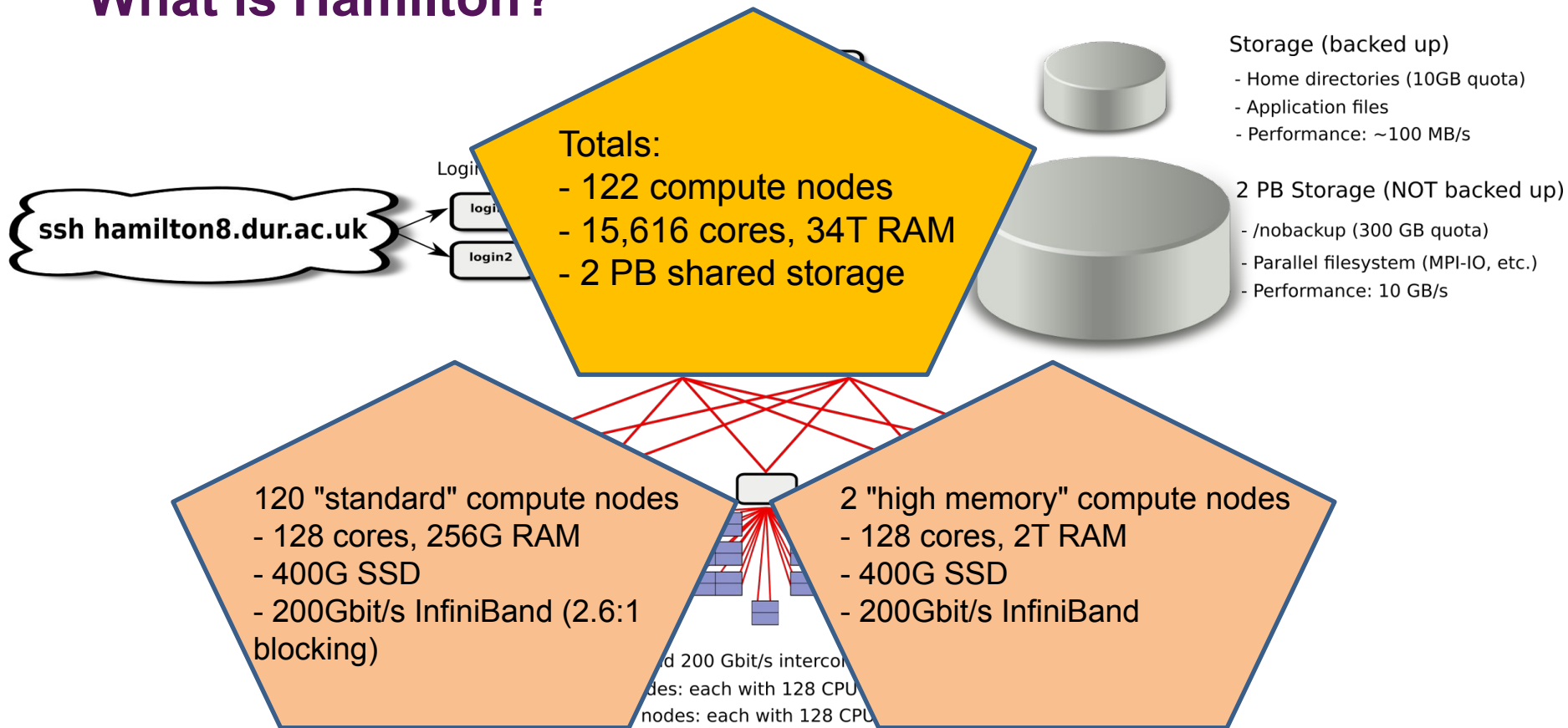


# What is Hamilton?





# What is Hamilton?



# 2. Parallel programming to achieve strong scaling

## Intro to 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. Writing your first MPI program

## Intro to MPI

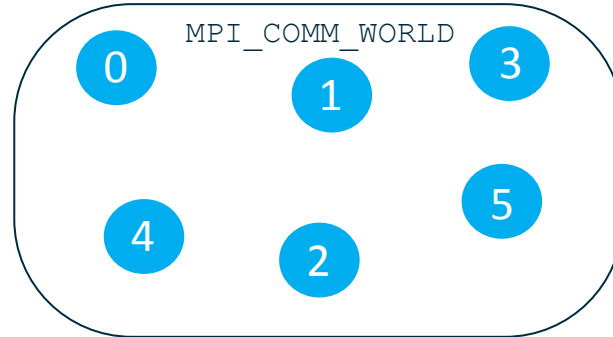
<i>The Standard...</i>	C	FORTRAN
Essential header files	<pre>#include &lt;mpi.h&gt;</pre>	<pre>include 'mpif.h'</pre>
Initialisation ( <u>always</u> the first MPI procedure called. Never called more than once)	<pre>int main (int argc, char *argv[]){  MPI_Init(&amp;argc, &amp;argv);</pre>	<pre>INTEGER IERR CALL MPI_INIT(IERR)</pre>
Finalisation ( <u>Essential</u> . Must be the last MPI procedure called)	<pre>MPI_Finalize();</pre>	<pre>CALL MPI_FINALIZE(IERR)</pre>
Function syntax...	<p>Case sensitive...</p> <pre>Error = MPI_Xxxx(parameter, ...); MPI_Xxxx(parameter)</pre>	<p>Case insensitive...</p> <pre>CALL MPI_XXXX(parameter, ..., IERR)</pre> <p>IERR returns 0 (success) or 1 (fail), same as return () in C.</p>



# 3. Writing your first MPI program

## Intro to MPI

- 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:

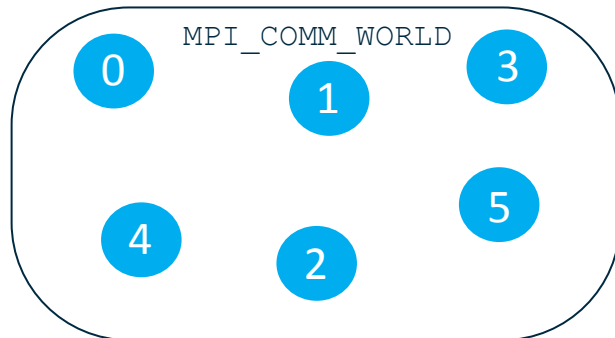


# 3. Writing your first MPI program

## Intro to MPI

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
<pre>int rank; MPI_Comm_rank(MPI_COMM_WORLD , &amp;rank);</pre>	<pre>INTEGER RANK, IERR CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)</pre>



The communicator  
MPI\_COMM\_WORLD  
(queried by the commands)  
contains ranks 0 to 5

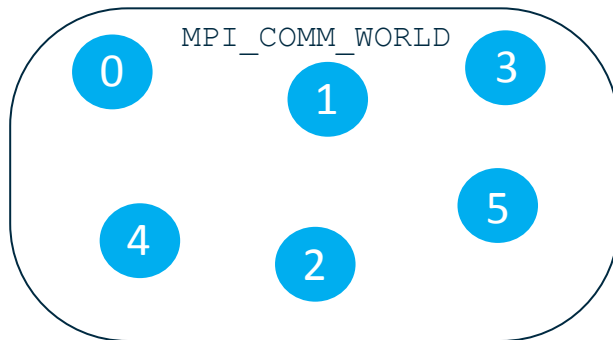


# 3. Writing your first MPI program

## Intro to MPI

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

C	FORTRAN
<pre>int size; MPI_Comm_size(MPI_COMM_WORLD , &amp;size);</pre>	<pre>INTEGER SIZE, IERR CALL MPI_COMM_SIZE(MPI_COMM_WORLD, SIZE, IERR)</pre>



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



# 3. Writing your first MPI program

## Intro to MPI

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. Writing your first MPI program

## Intro to MPI

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
<code>mpicc my_prog.c -o myprogram</code>	<code>mpif90 my_prog.f -o myprogram</code>

- 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. Writing your first MPI program

## Intro to MPI

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

Job.sh file contents:

- 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`

```
#!/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!

## Intro to MPI

- 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

## Intro to MPI

C	FORTRAN
<pre>#include &lt;stdio.h&gt; #include &lt;mpi.h&gt;  int main (int argc, char *argv[]) {     int rank, size;      MPI_Init(&amp;argc, &amp;argv); /* Initialise MPI */     MPI_Comm_rank(MPI_COMM_WORLD, &amp;rank); /* Get rank */     MPI_Comm_size(MPI_COMM_WORLD, &amp;size); /* Get size */      printf("Hello from rank %d of size %d.\n", rank, size);      MPI_Finalize(); }</pre>	<pre>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</pre>



# 4. Point to point communications

## Intro to MPI

### Messages

- Data types

### Communication modes and completion

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

### Communication envelope

### Message order preservation

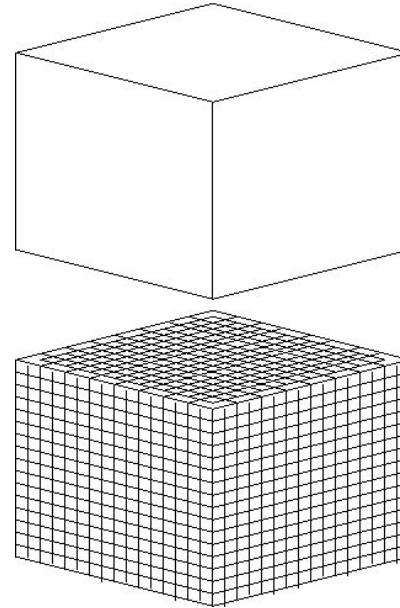
### Combined send and receive



# 4. Point to point communications

## Intro to MPI

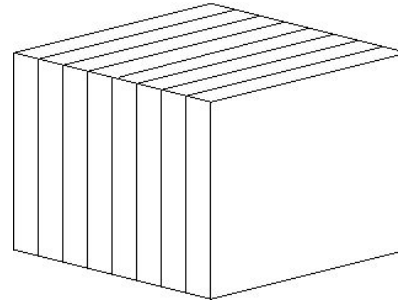
- An example: this is a representation of a domain for a piece of CFD software that solves the equations of fluid dynamics to evolve a fluid with time:-
- The domain is broken down into a number of cells:- (e.g. 20 x 20 x 20: 8000 cells)
- If solving Euler's equation takes 1 second to evolve the fluid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



# 4. Point to point communications

## Intro to MPI


- If we slice the domain and distribute the slices between processors in a parallel computer, we can in theory accelerate the process
- With only 8 processors, each one updates 1000 cells and nominally the program takes only 1000s per update.
- If we ran with 1000 processors (very feasible nowadays!), we would be in real time. (weather forecasting uses enough processors to simulate faster than real time!)
- There are some nuances to consider though...
  - 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*”. This, along with phrases like “*task farming*” cover cases where each process can compute independently. Strong scaling can be easily achieved.



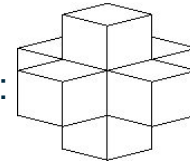
# 4. Point to point communications

## Intro to MPI

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

In our code, each cell: 

Needs to know about the conditions in its neighbours in every direction:



In order to calculate the flow between cells and update its own fluid conditions.

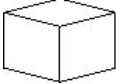


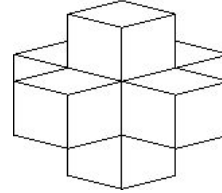
What happens if a neighbouring cell is held in different memory on another process?

- Communication must occur between processors
- “Message passing” is the context in which this takes place, using a message passing interface, or **MPI** library

# 4. Point to point communications

## Intro to MPI

Each cell  needs to know about its neighbours



in order to update.

Cell conditions can be passed across boundaries between processors in packages which may be individual variables, or an array of variables,

Messages are packets of data moving between processes.

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. These are basic or derived datatypes, built from basic types.





# 4. Point to point communications

## Intro to MPI

C: MPI Data types	FORTTRAN: 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

## Intro to MPI

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



# 4. Point to point communications

## Intro to MPI

`MPI_Ssend`: the safest form of communication

Processes synchronise

Sender process specifies synchronous mode

Blocking: both processes wait until the transaction has completed

For a communication to succeed:

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Message types must match
- Receiver's buffer must be large enough



# 4. Point to point communications

## Intro to MPI

### FORTTRAN 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)
```



# 4. Point to point communications

## Intro to MPI

### 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);
```



# 4. Point to point communications

## Intro to MPI

### FORTTRAN 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)
```



# 4. Point to point communications

## Intro to MPI

### 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);
```



# 4. Point to point communications

## Intro to MPI

### 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();
}
```





# 4. Point to point communications

## Intro to MPI

FORTTRAN 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
```



# 4. Point to point communications

## Intro to MPI

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.



# 4. Point to point communications

## Intro to MPI

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



# 4. Point to point communications

## Intro to MPI

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

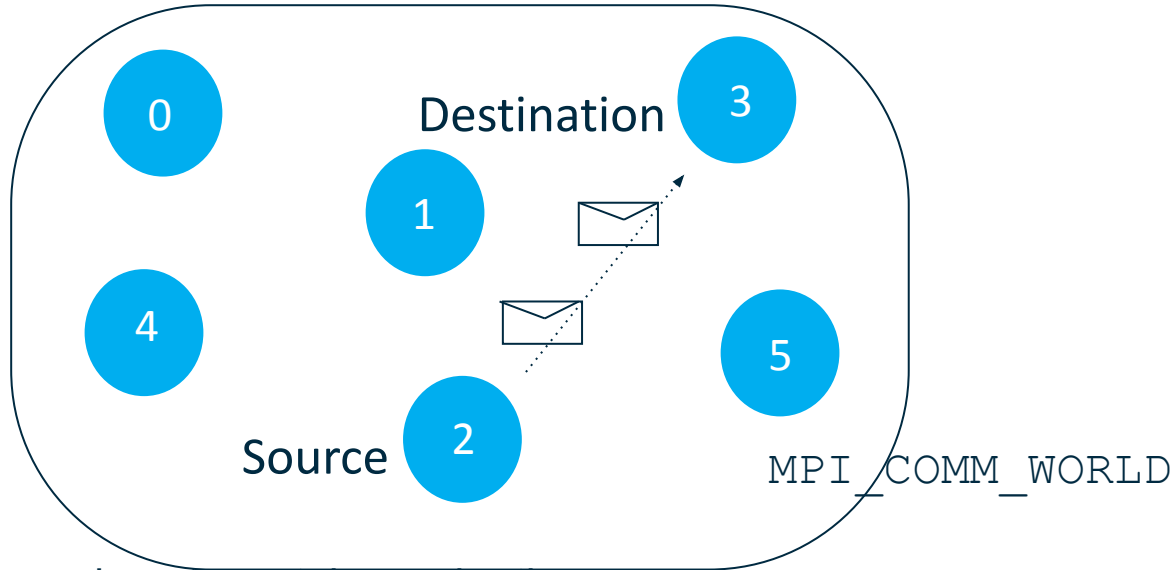
Querying the status parameter	C	FORTRAN
Source process	<code>source=status.MPI_SOURCE;</code>	<code>source=status(MPI_SOURCE)</code>
Tag	<code>tag=status.MPI_TAG;</code>	<code>tag=status(MPI_TAG)</code>
Error code	<code>error=status.MPI_ERROR;</code>	<code>error=status(MPI_ERROR)</code>
Count	<code>MPI_Get_count(&amp;status,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



# 4. Point to point communications

## Intro to MPI

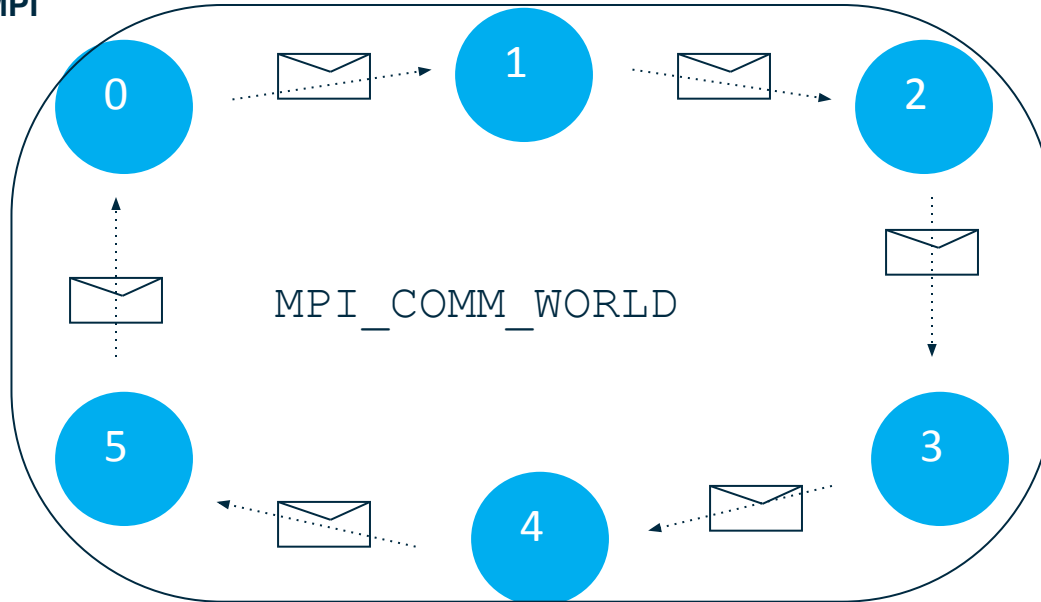
The order of messages is preserved:



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

# 4. Point to point communications

## Intro to MPI



**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.



# 4. Point to point communications

## Intro to MPI

Deadlock avoidance: carry out non-blocking communication

Sending process	Receiving process
Initiate send, non-blocking ( <code>MPI_Isend</code> )	Initiate receive, non-blocking ( <code>MPI_Irecv</code> )
Perform other tasks	Perform other tasks
Wait for completion ( <code>MPI_Wait</code> )	Wait or test for completion ( <code>MPI_Test</code> )

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

## Intro to MPI

### 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.



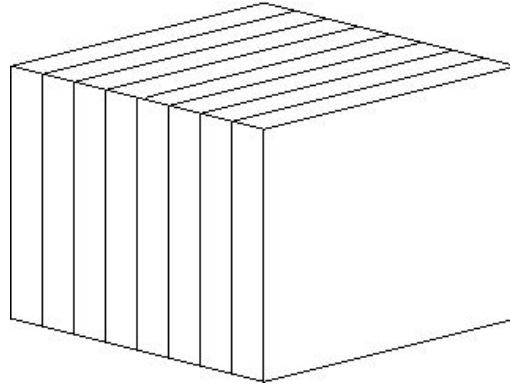


# 4. Point to point communications

## Intro to MPI

### *Recalling our fluid dynamics example*

- When each processor has 1000 cells of our 8000-cell domain



- `MPI_Sendrecv` is a useful way of combining the transfer of all the border cells in one go i.e.

Border slice =  $20 \times 20 = 400$  cells

5 pieces of information per cell:

Sending and receiving an array of 2000 variables simultaneously.



# 4. Point to point communications

## Intro to MPI

### MPI\_Sendrecv

C:

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

FORTTRAN:

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

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



# 4. Point to point communications

## Intro to MPI

### 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

## Intro to MPI

### 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. 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

## Intro to MPI

### 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/>



# 5. Collective communications

## Intro to MPI

Introduction & characteristics

Barrier Synchronisation

Broadcast

Scatter

Gather

Global reduction operations

- Predefined operations
- User-defined operations

Partial sums



# 5. Collective communications

## Intro to MPI

Collective communication involves a group of processes.

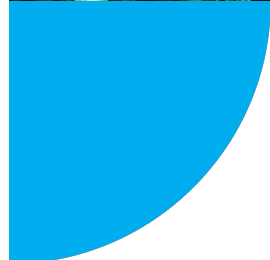
Called by *all* processes in a communicator.

Examples:

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

Characteristics

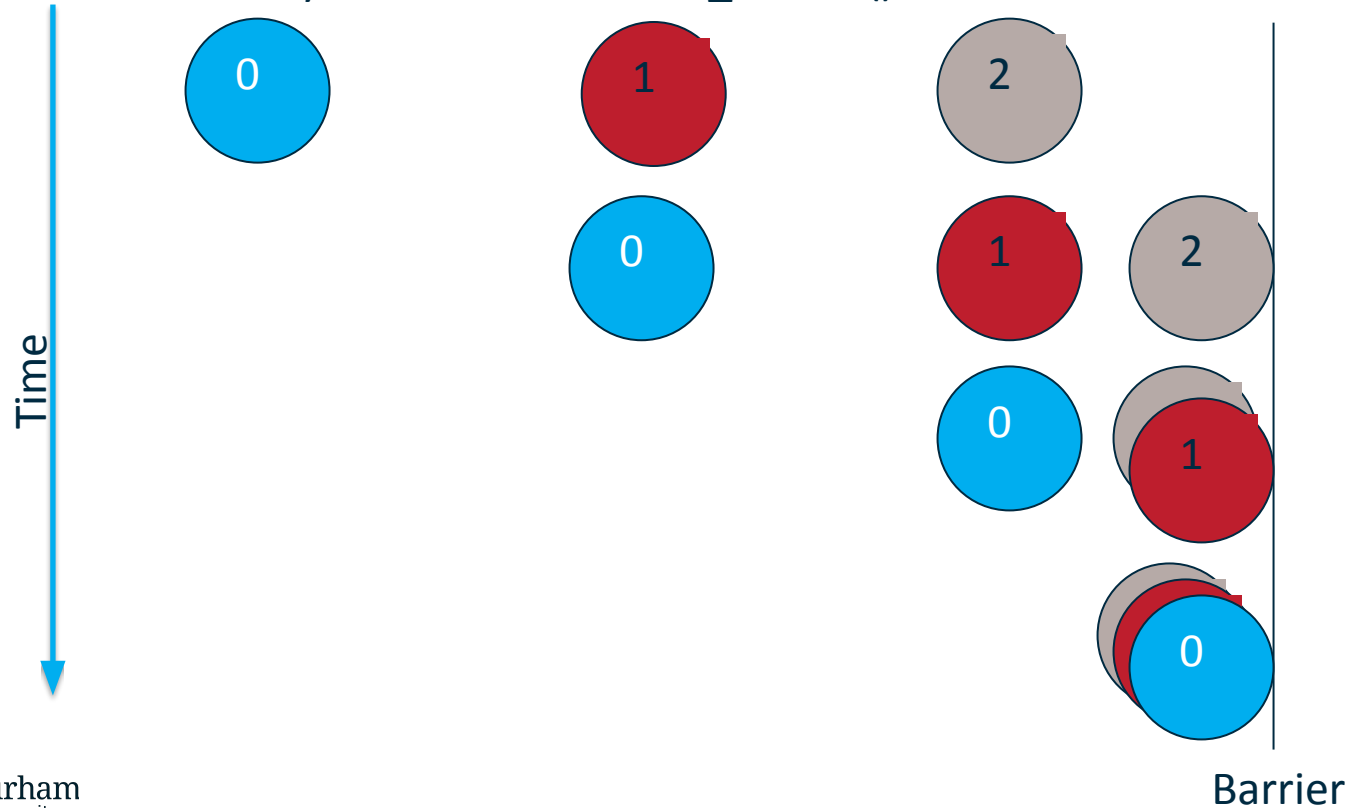
- Collective communication will not interfere with point-to-point communication and vice-versa.
- All processes must call the collective routine.
- Synchronization not guaranteed (except for barrier)
- No non-blocking collective communication
- No tags
- Receive buffers must be exactly the right size



# 5. Collective communications

Intro to MPI

Barrier Synchronisations: MPI\_Barrier()





# 5. Collective communications

## Intro to MPI

### 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:

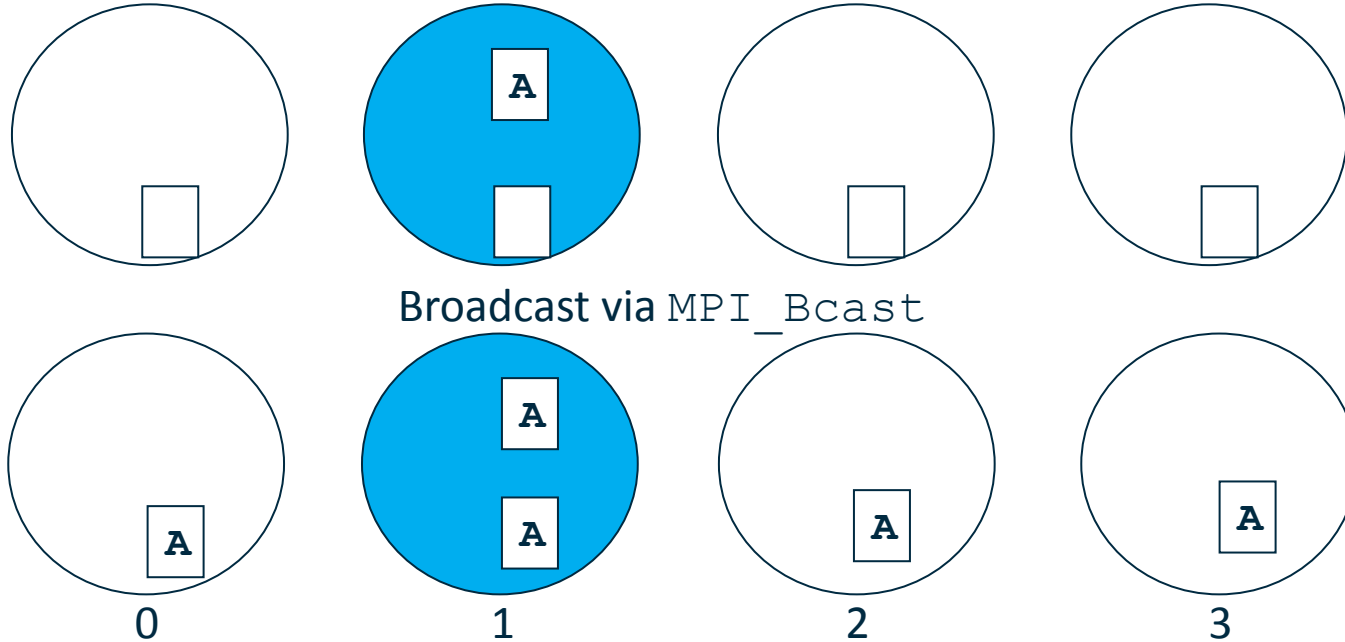
Barrier calls are exceptionally useful for avoiding 'racing' issues, where one processor can race ahead of the others and set up deadlock.



# 5. Collective communications

## Intro to MPI

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

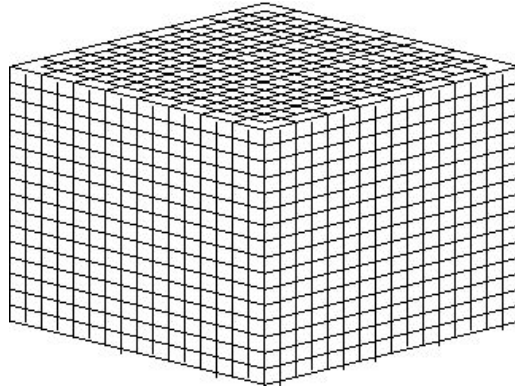


# 5. Collective communications

## Intro to MPI

*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

## Intro to MPI

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);
```



# 5. Collective communications

## Intro to MPI

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`



# 5. Collective communications

## Intro to MPI

### 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

## Intro to MPI

MPI name	Function	C	FORTRAN
MPI_MAX	Maximum		MAX(a <sub>1</sub> ... ..a <sub>n</sub> )
MPI_MIN	Minimum		MIN(a <sub>1</sub> ... ..a <sub>n</sub> )
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		

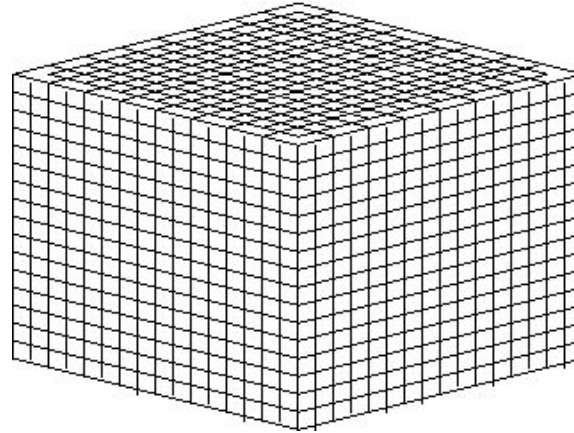


# 5. Collective communications

## Intro to MPI

*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

## Intro to MPI

`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)
```



# 5. Collective communications

## Intro to MPI

`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



# 6. I/O using MPI-IO

## Intro to MPI

Of course it's possible to just do standard serial I/O:

- C: `fopen`, `fprintf`, `fclose`
- FORTRAN: `open`, `write`, `close`
- Extreme care needs to be taken over the naming and location of files.
- Options:
  1. Each process could write its data portion to separate data files
    - fast, but difficult to analyse
  2. Each process could write sequentially to one file
    - slow!
  3. One particular process collects data and then writes out that datafile
    - slow, memory demanding!
  4. **We have MPI-IO! Parallel data file handling!**



# 6. I/O using MPI-IO

## Intro to MPI

### Overview

- Initialise and open a file on all processes.
- Each process computes its 'offset' from the start of the file.
- Each process uses this offset to set a file 'view'.
- A single command synchronously called from all processes in the group reads data from or writes data to a single file.
- Finalise and close the file on all processes.

Can be tuned for particular hardware e.g. RAID arrangements.



# 6. I/O using MPI-IO

## Intro to MPI

In principle:

- Requires initialisation of a file across whole communicator group  
`MPI_FILE_OPEN`
- Requires each process in group to set particular 'view' for writing  
`MPI_FILE_SET_VIEW`  
can use an offset e.g. `offset = rank*count`
- Or seek a particular position to read from  
`MPI_FILE_SEEK`
- Uses MPI commands similar to Send and Recv to write/read to a file  
`MPI_FILE_WRITE_ALL` / `MPI_FILE_READ_ALL`
- Requires finalisation of the file across whole communicator group  
`MPI_FILE_CLOSE`
  - Can be complex (e.g. in handling n-dimensional arrays)



# 7. Higher functions

## Intro to MPI

But it's probably best to take a fifth option and use libraries built using MPI-IO!

- Parallel HDF5 (PHDF5) is library for parallel IO in the HDF format.
  - C and FORTRAN interfaces.
  - Files are cross-compatible with serial HDF5 and sharable between platforms.
  - Designed to have a single file image for all parallel processes.
  - Supports MPI programming, but not (as far as I'm aware) shared memory programming – it is built on MPI-IO.
- NetCDF
  - NetCDF (network Common Data Form) is a set of interfaces for array-oriented data access and a freely-distributed collection of data access libraries for C, FORTRAN, C++, Java and other languages.
  - The NetCDF libraries support a machine-independent format for representing scientific data.
  - Together, the interfaces, libraries and format support the creation, access and sharing of scientific data.
  - Sacrifices some performance (as against HDF5 & MPI-IO directly) for a somewhat simpler API.



# 7. Higher functions

## Intro to MPI

### Cartesian Topologies

- As each MPI process has a rank, there is always an order defined within a group: 1D from 0 to size-1
- In MPI, it is possible to more elegantly define which processes should exchange information by defining a set of connections between processes, known as a topology
- Cartesian topologies are pre-defined in MPI. They associate each process with a co-ordinate in a Cartesian system, allowing mapping of processes onto a user's 2D/3D simulation space.
  - Create with `MPI_Cart_create`
  - Translate rank into coordinates with `MPI_Cart_coords`
  - Locate neighbours in every direction with `MPI_Cart_shift`

BUT, be very careful with boundaries and communications at vertices etc.





# 7. Higher functions

## Intro to MPI

### Derived data types

- Today, we have only covered the basic data types in communication
- MPI data types for non-basic types, e.g. Contiguous or strided arrays in C, or FORTRAN90 types, can, however, be constructed.
- As long as you are sure of the size and representation of your data, you can transmit raw data using the data type `MPI_BYTE`.
  - Construct – various MPI commands  
`MPI_Type_contiguous`, `MPI_Type_vector`, etc.
  - Commit with `MPI_Type_commit`
  - Use, as you would any other MPI data type
  - After use, free with `MPI_Type_free`



# 7. Higher functions

## Intro to MPI

### Defining binary operators

- In the event that the MPI predefined binary operators are not sufficient, MPI provides a mechanism for users to define their own.

- Binding a function as an operator: `MPI_Op_create`
- Free after use with: `MPI_Op_free`
- In C, the prototype for an `MPI_User_Function` is

```
typedef void MPI_User_function(void *invec, void *inoutvec,  
int *len, MPI_Datatype *datatype);
```

- In FORTRAN, a user defined operation is a subroutine declared as `EXTERNAL`

```
SUBROUTINE USER_FUNCTION( INVEC, INOUTVEC, LEN, DATATYPE)  
INTEGER LEN, DATATYPE  
<DATATYPE> INVEC(LEN), INOUTVEC(LEN)
```



# 7. Higher functions

## Intro to MPI

- In MPI, a collection of processing elements constitute a group. The automatic MPI group is `MPI_COMM_WORLD`, which contains all the processes at execution.
- It is possible to define your own groups
  - Bind with `MPI_Comm_group`
  - Find out group size with `MPI_Group_size`
  - Find out rank in group with `MPI_Group_rank`
  - Create subsets etc with  
`MPI_Group_incl`, `MPI_group_excl`  
`MPI_Group_union`  
`MPI_Group_intersection`  
`MPI_Group_difference`
  - Copying with `MPI_Comm_dup`
  - Group destructor is `MPI_Group_free`



# 7. Higher functions

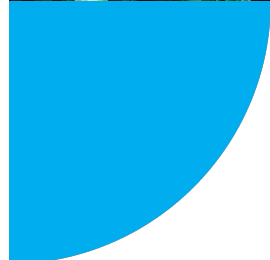
## Intro to MPI

### Numerical libraries – BLACS

- Basic Linear Algebra Communication Subprograms
- Frequently occurring operations in linear algebra
- Portable, machine specific versions built on MPI
- See <http://www.netlib.org/blacs/>

### Numerical libraries – PBLAS

- Parallel Basic Linear Algebra Subprograms
- Similar to BLAS in functionality
  - Same three levels
    - Level 1 – Vector Operations
    - Level 2 – Vector-Matrix Operations
    - Level 3 – Matrix-Matrix Operations
- Act on globally distributed arrays
- Built on top of the BLAS and BLACS libraries
- See [http://www.netlib.org/scalapack/pblas\\_gref.html](http://www.netlib.org/scalapack/pblas_gref.html)



# 7. Higher functions

## Intro to MPI

### Numerical libraries – ScaLAPACK

- Scalable Linear Algebra PACKage
- Efficient
  - Uses optimised computation and communication engines and LAPACK/BLAS for good serial performance on individual processes
- Scalable
- Portable
  - Machine dependencies contained within BLAS and BLACS
- Easy to use – calling interface similar to LAPACK
- Only a subset of LAPACK routines are currently provided
- See <http://www.netlib.org/scalapack/>



# 7. Higher functions

## Intro to MPI

### Numerical libraries – FFTW

- “Fastest Fourier Transform in the West”
- FFTW is a library for computing the discrete Fourier Transform in one or more dimensions of both real and complex data
- FFTW has become the FFT library of choice for most applications
- Serial and parallel versions of this library exist
- FFTW/2.1.5 has recently been superseded as the parallel FFT library by FFTW/3 – 3.3.9 is currently the latest official release.
- See <http://www.fftw.org/>



# 7. Higher functions

## Intro to MPI

### Numerical libraries – NAG Parallel Library

- Aimed at typical applications required in industrial, commercial and research environments
- Excellent performance and scalability across a wide range of systems, including shared memory platforms
- Routines can be easily called from other languages
- See <http://www.nag.com/numeric/fd/FDdescription.asp>
- Personal/Group license required

Also, PETSC (open source),

& DEAL.II (Differential Equations Analysis Library, also open source)



# 7. Higher functions

## Intro to MPI

Several profiling tools are available

These can enable rapid identification of bottlenecks, as well as allow for optimisation.

- Compiler based reporting
  - Intel: opt-report and vec-report
- Third party software
  - Tau, papi, scalasca, Intel cluster suite





# 7. Higher functions

## Intro to 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 (gather, broadcast, scatter) are often better optimised than writing your own versions



# 7. Higher functions

## Intro to MPI

- Debugging
  - Parallel debugging can be hard  
e.g. 6 months to write the code and produce output, 2 years to debug and produce 'identical' output to serial run
  - Truly identical results are very difficult to produce on runs involving different numbers of processors
    - You may need to write software to compare outputs and set a tolerance (single precision tolerance:  $10^{-6}$ , double precision tolerance:  $10^{-12}$ ) to look for any difference over and above the rounding error
  - Often writing outside arrays is a common error
    - compiling with `-g` and running within `gdb` can easily locate the problem
  - Poor scaling?  
Vary input data and number of processes to characterise the problem  
Insert timers, locate bottlenecks, experiment with different MPI routines  
Try profiling: MPE logging and upshot to view logs



# 7. Higher functions

## Intro to MPI

- Debugging
  - Often writing outside arrays is a common error
    - compiling the serial version with `-g` and running within the gdb debugger can easily locate the problem, rather than using `printf` statements.
  - It is possible to parallel debug with gdb:-

```
mpirun -np <NP> xterm -e gdb ./program
```
  - This gives you `<NP>` xterminals. You will need to type `run` in each one to begin executing your code.





Durham  
University

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

**Thank you!**

**Feedback**

**[https://bit.ly/arc\\_trainingfeedback](https://bit.ly/arc_trainingfeedback)**

**Email: [arc@durham.ac.uk](mailto:arc@durham.ac.uk)**

**RSE team: [arc-rse@durham.ac.uk](mailto:arc-rse@durham.ac.uk)**

**Web: <https://www.dur.ac.uk/arc/>**