



Durham
University

A brief introduction to parallel programming on a supercomputer

Instructors:

Dmitry Nikolaenko (RSEng)
Thomas Flynn (RSEng)

June 5, 2025

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

Course Outline

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

Aims of the course:

- Introduction to **parallel programming** 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 **collective** and combined parallel communications

Course Schedule



COFFEE BREAK

09:00-09:15 - Brief introduction to HPC and parallel programming models

09:15-10:00 - **Basics of OpenMP**

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**

10:15-11:30 - **Basics of MPI: Point-to-point communications**

11:15-11:30 – Practical 4: “Ping pong!”

11:30-12:30 – **Basics of MPI: Collective communications**

12:00-12:15 – Practical 5: “Collective communication”

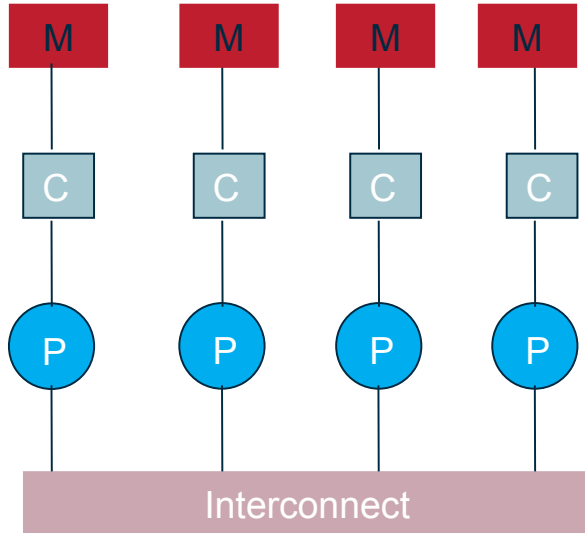


Introduction

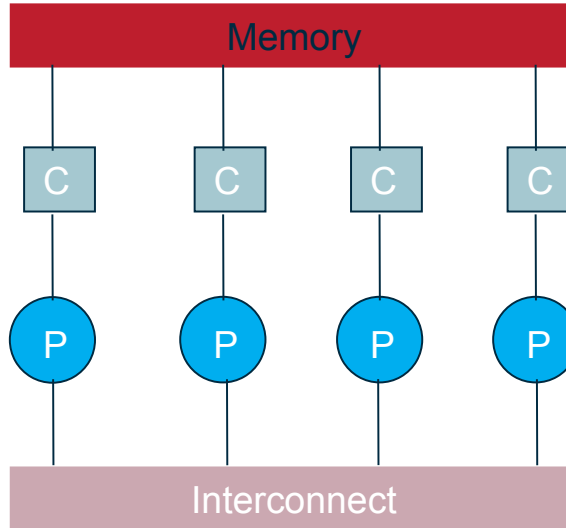


1. Brief intro to HPC and parallel programming models

1.1 Distributed memory and shared memory systems



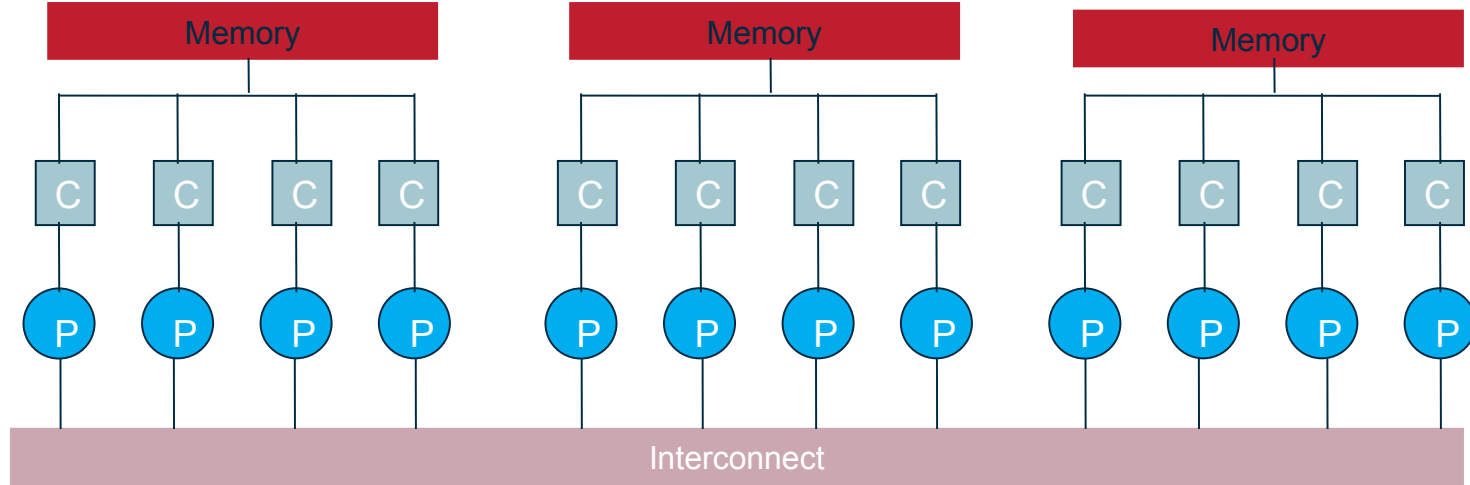
Distributed memory system. This parallelism relies on message passing



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

1. Brief intro to HPC and parallel programming models

1.2 Basic HPC cluster structure

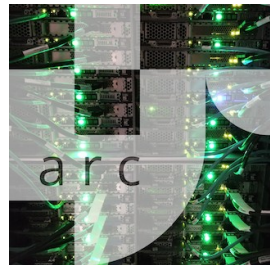


Modern clusters look more like this: **shared** memory on individual **nodes** with **distributed** memory between **nodes**

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



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 mpi`
- 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`

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`



OpenMP



3 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
```



3 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



3 Basics of OpenMP

3.1 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
```



3 Basics of OpenMP

3.2 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



3 Basics of OpenMP

3.3 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



3 Basics of OpenMP

Example: scheduling

- We will now work through the

[BasicParallelProgramming/omp/examples/scheduling](#)

completed example

- 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



3 Basics of OpenMP

3.4 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!



3 Basics of OpenMP

3.5 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 be 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
}
```



3 Basics of OpenMP

3.6 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;
}
```



3 Basics of OpenMP

3.6 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 ($\text{sum} \rightarrow \text{sum}+1$)
- 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!



3 Basics of OpenMP

3.6 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;
}
```



3 Basics of OpenMP

3.6 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;
}
```



3 Basics of OpenMP

Example: race conditions

- We will now work through the

[BasicParallelProgramming/omp/examples/raceConditions](#)

completed example

- 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!



3 Basics of OpenMP

Practical: race conditions

- Have a go at the

[BasicParallelProgramming/omp/practicals/factorial](#)

practical

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



3 Basics of OpenMP

3.7 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  
  
    }  
  
}
```



3 Basics of OpenMP

Example: tasks

- We will now work through the

[BasicParallelProgramming/omp/examples/tasks](#)

completed example

- 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



3 Basics of OpenMP

Practical: calculation pi

- Have a go at the

[BasicParallelProgramming/omp/practicals/piCalc](#)

practical

- 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:

[BasicParallelProgramming/omp/practicals/mandelBrot](#)



MPI



2 Basics of MPI

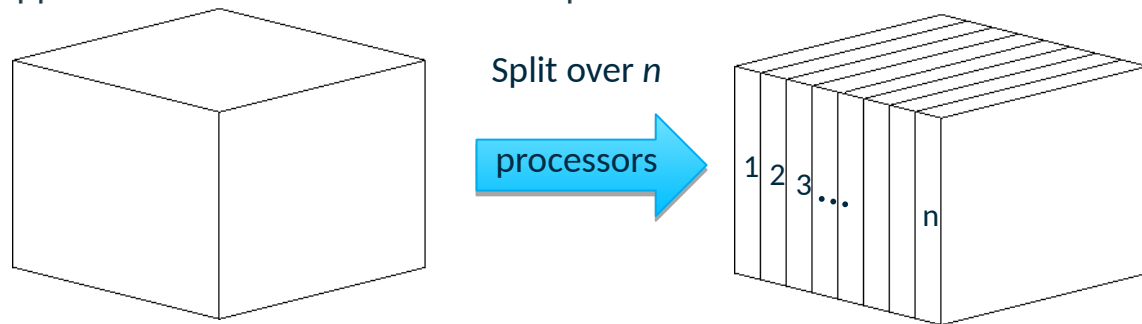
The General Message Passing Paradigm

- All variables are private to each process. Values of variables are held in local memory
- 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.
- “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.”
- There are multiple flavours: OpenMPI, MPICH, MVAPICH, etc. including vendor specific such as Intel, Cray, etc.



2 Basics of MPI

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



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



2 Basics of MPI

2.1 Writing your first MPI program: introduction

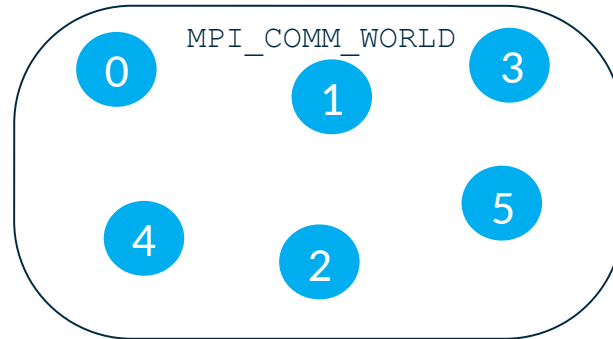
<i>The Standard...</i>	C	FORTRAN
Essential header files	<code>#include <mpi.h></code>	<code>include 'mpif.h'</code>
Initialisation (always the first MPI procedure called. Never called more than once)	<pre>int main (int argc, char *argv[]) { MPI_Init(&argc, &argv);</pre>	<pre>INTEGER IERR CALL MPI_INIT(IERR)</pre>
Finalisation (Essential . 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>



2 Basics of MPI

2.1 Writing your first MPI program: communicators

- 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:

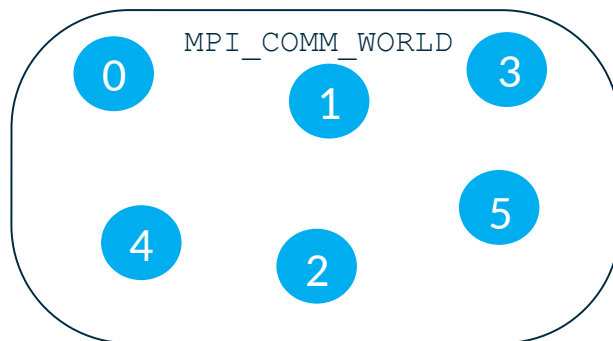


2 Basics of MPI

2.1 Writing your first MPI program: MPI rank

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, &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

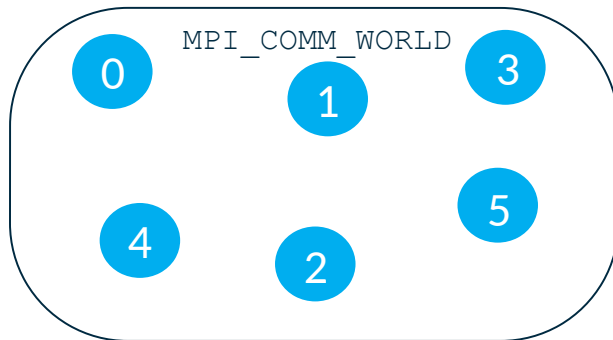


2 Basics of MPI

2.1 Writing your first MPI program: MPI size

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, &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



2 Basics of MPI

2.1 Writing your first MPI program: 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.



2 Basics of MPI

2.1 Writing your first MPI program: compilation

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 gcc  
module load openmpi
```

- To compile:

C	FORTRAN
<pre>mpicc my_prog.c -o myprogram</pre>	<pre>mpif90 my_prog.f -o myprogram</pre>

- 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...



2 Basics of MPI

2.1 Writing your first MPI program: submitting jobs on Hamilton

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 gcc
module load openmpi
mpirun ./myprogram
```



Practical 0: Hello World!

- Write a minimal MPI program that prints “Hello World!”
 - Serial template code is available for C and FORTRAN on Hamilton here:-
`exercises/practical0/helloworld.c`
`exercises/practical0/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:
`exercises/practical0/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 0: Review

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



2 Basics of MPI

2.2 Point-to-point communications: contents

Messages

- Data types

Communication modes and completion

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

Communication envelope

Message order preservation

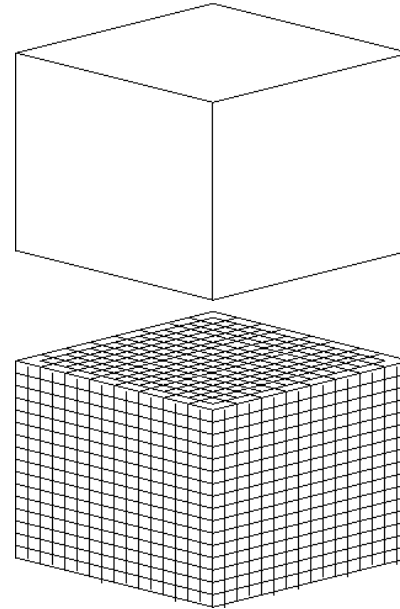
Combined send and receive



2 Basics of MPI

2.2 Point-to-point communications: 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. $20 \times 20 \times 20$: 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*”.



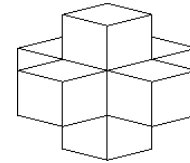
2 Basics of MPI

2.2 Point-to-point communications: 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?

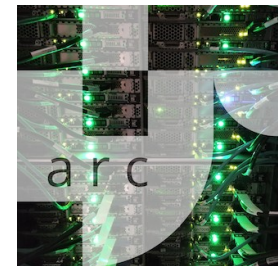
- Communication must occur between processes
- “Message passing” is the context in which this takes place, using a message passing interface, or **MPI** library
- The message passing system needs to be aware of the following information:
 - 1) The ‘rank’ of the message source
 - 2) Source buffer: variable / 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.



2 Basics of MPI

2.2 Point-to-point communications: data types

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



2 Basics of MPI

2.2 Point-to-point communications: sender mode

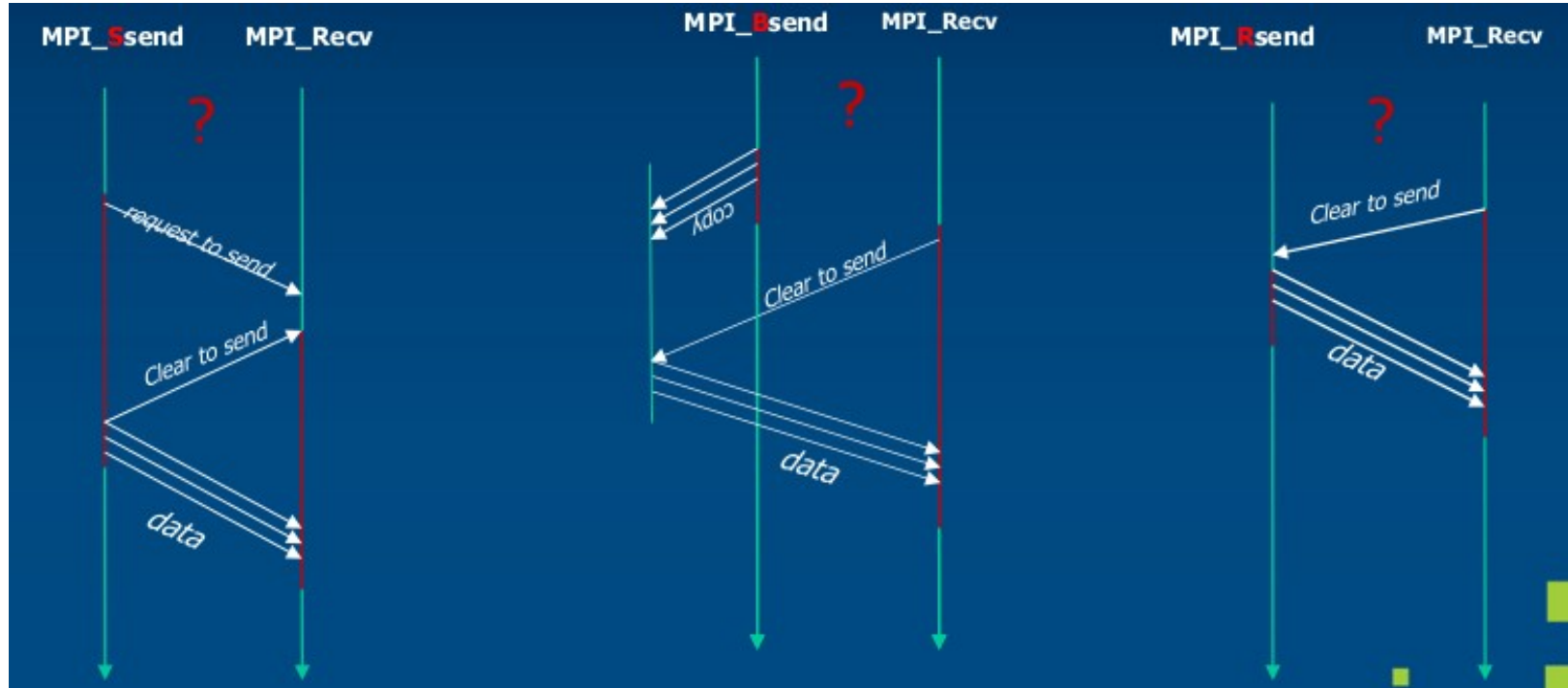
Sender mode	MPI Call	Completion status
Synchronous send	<code>MPI_Ssend</code>	Only completes when the receive has completed.
Buffered send	<code>MPI_Bsend</code>	Always completes (unless an error occurs), irrespective of receiver.
Standard send	<code>MPI_Send</code>	Can be synchronous or buffered (often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



2 Basics of MPI

2.2 Point-to-point communications: sequence diagram for sender modes

Sender modes - explained



2 Basics of MPI

2.2 Point-to-point communications: Synchr. Send command (C 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);
```



2 Basics of MPI

2.2 Point-to-point communications: Synchr. Send command (FORTRAN 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)
```



2 Basics of MPI

2.2 Point-to-point communications: Receive command (C 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);
```



2 Basics of MPI

2.2 Point-to-point communications: Receive command (FORTRAN 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)
```



2 Basics of MPI

2.2 Point-to-point communications: C example for Synchronous Send

```
#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();
}
```



2 Basics of MPI

2.2 Point-to-point communications: FORTRAN example for Synchronous Send

```
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
```



2 Basics of MPI

2.2 Point-to-point communications: Wildcarding

The receiving process can wildcard

To receive from any source:

- Set source to `MPI_ANY_SOURCE`

To receive with any tag:

- Set tag to `MPI_ANY_TAG`

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



2 Basics of MPI

2.2 Point-to-point communications: `status` communication envelope

Like in 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

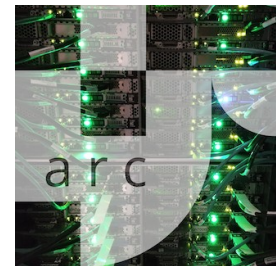


2 Basics of MPI

2.2 Point-to-point communications: `status` in C and FORTRAN

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

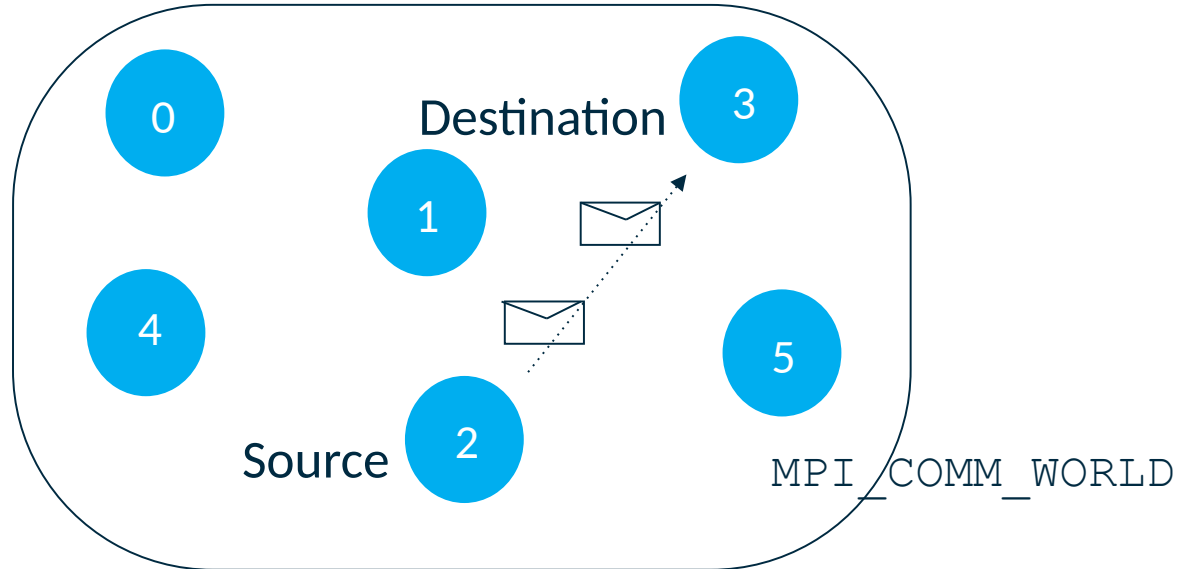
Querying the status parameter	C	FORTRAN
Source process	<code>source=status.MPI_SOURCE;</code>	<code>source=status (MPI_SOURCE)</code>
Tag	<code>tag=status.MPI_TAG;</code>	<code>tag=status (MPI_TAG)</code>
Error code	<code>error=status.MPI_ERROR;</code>	<code>error=status (MPI_ERROR)</code>
Count	<code>MPI_Get_count(&status,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



2 Basics of MPI

2.2 Point-to-point communications: order of messages

The order of messages is preserved:

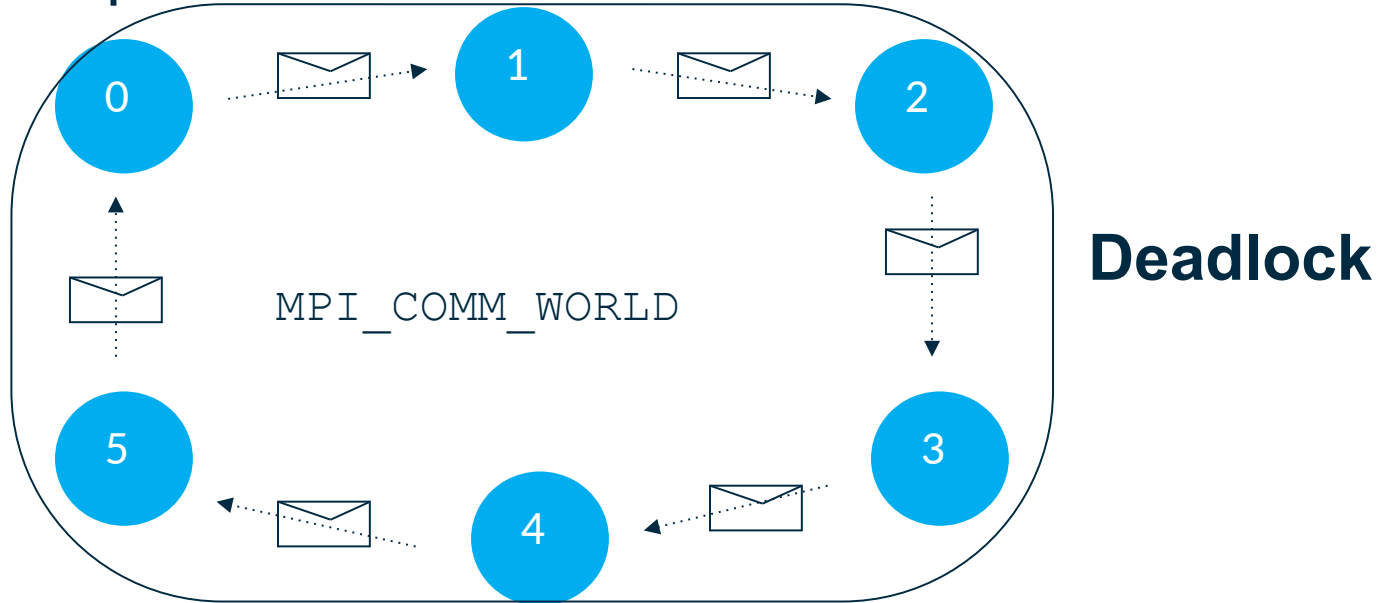


- Messages do not overtake each other.
- This is also true for non-synchronous (buffered) sends.



2 Basics of MPI

2.2 Point-to-point communications: deadlock



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



2 Basics of MPI

2.2 Point-to-point communications: non-blocking communications

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.



2 Basics of MPI

2.2 Point-to-point communications: 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.



2 Basics of MPI

2.2 Point-to-point communications: MPI_Sendrecv syntax (C and FORTRAN)

MPI_Sendrecv

C:

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

FORTRAN:

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

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



2 Basics of MPI

2.2 Point-to-point communications: `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 1: 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:
`exercises/practical1/PingPong.c`
`exercises/practical1/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 1: Review

The principles of node pair communication

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

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

For the complete answers, please consult the solutions:

[mpi/solutions1/](#)



2 Basics of MPI

2.3 Collective communications: contents

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



2 Basics of MPI

2.3 Collective communications: 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

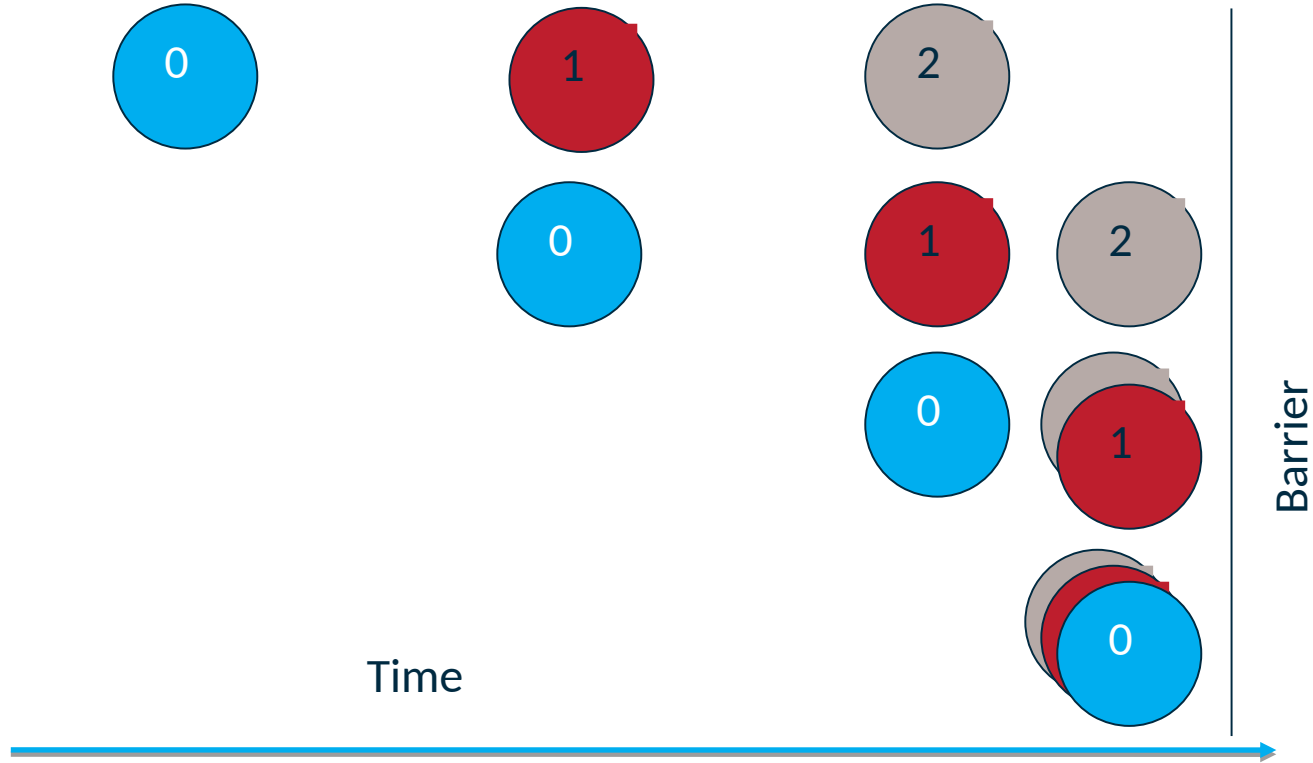
Characteristics

- 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



2 Basics of MPI

2.3 Collective communications: Barrier Synchronisation



2 Basics of MPI

2.3 Collective communications: Barrier Synchronisation syntax (C and FORTRAN)

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:

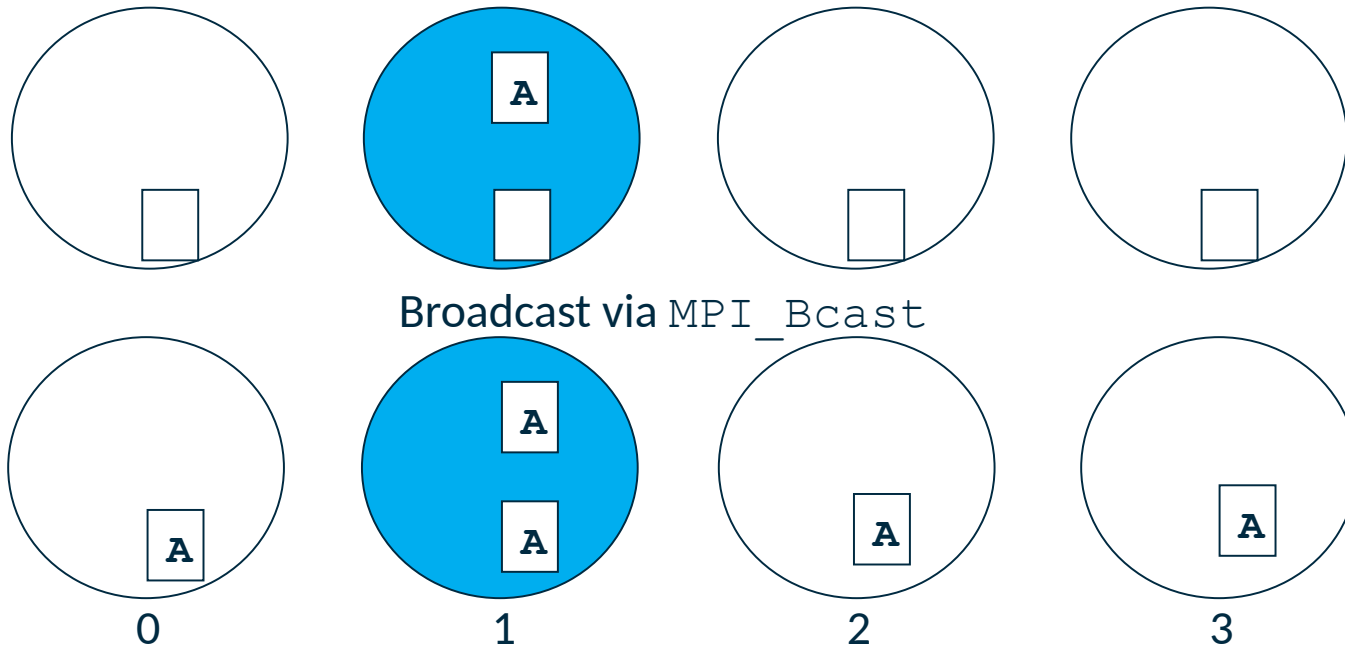
Barrier calls are exceptionally useful for avoiding 'racing' issues, where one processor can race ahead of the others and set up deadlock.



2 Basics of MPI

2.3 Collective communications: Broadcasting

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



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

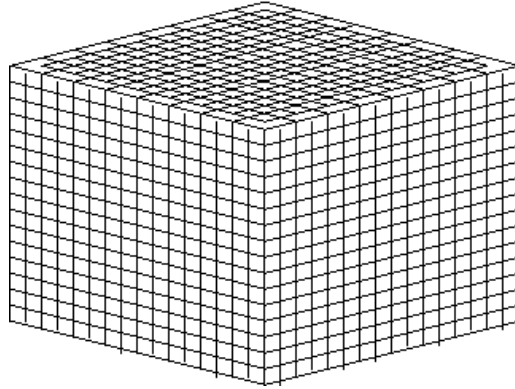


2 Basics of MPI

2.3 Collective communications: 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.



2 Basics of MPI

2.3 Collective communications: Broadcast syntax (C and FORTRAN)

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);
```



2 Basics of MPI

2.3 Collective communications: gathering and scattering routines

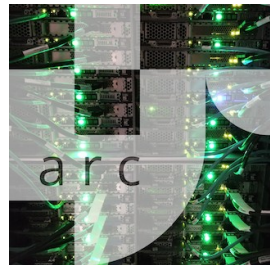
- `MPI_Scatter` – scatters data from a single process to all processes
- `MPI_Gather` – gathers data from all processes to a single process
- `MPI_Allgather` – each process receives a copy of the gathered data
- `MPI_Alltoall` – gathers data and scatters (possibly different) data from all to all processes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



2 Basics of MPI

2.3 Collective communications: global reduction operations

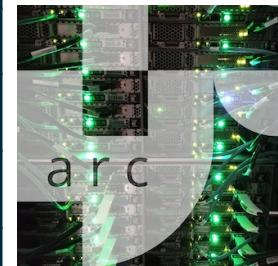
- Compute a result involving data distributed over a group of processes
- Suppose that each process i has computed a number X_i and that the result needed X is the sum of these. This global sum is an example of a *reduction operation*.
- In MPI, a set of binary reduction operations are defined for predefined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



2 Basics of MPI

2.3 Collective communications: examples of global reduction operations

MPI name	Function	C	FORTTRAN
MPI_MAX	Maximum		MAX(a ₁a _n)
MPI_MIN	Minimum		MIN(a ₁a _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		

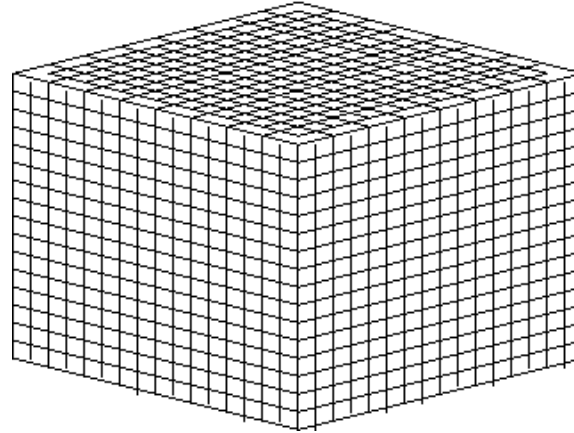


2 Basics of MPI

2.3 Collective communications: CFD example 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 its section and then the minimum of all these values is used as the global timestep.



2 Basics of MPI

2.3 Collective communications: `MPI_Allreduce` syntax (C and FORTRAN)

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)
```



2 Basics of MPI

2.3 Collective communications: MPI_Scan syntax (C and FORTRAN)

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 2: collective communications

1. Collective communication with `MPI_Allreduce`

- Compute the global sum of all ranks of the processes using MPI global reduction
- Template:
`exercises/practical2/collective.c`
`exercises/practical2/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 2: Review

For the complete answers, please consult the solutions:
`mpi/solutions2/`





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

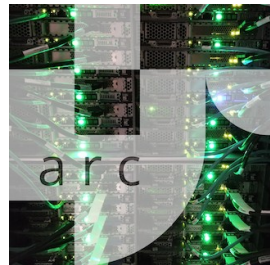


Old slides from here

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



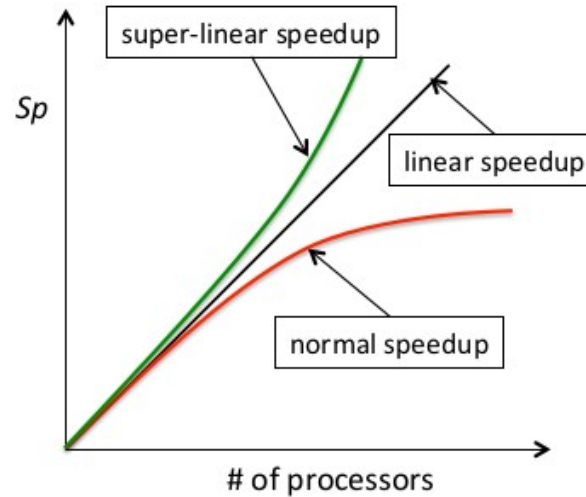
1. Brief intro to HPC and parallel programming models

1.5 HPC terminology (cont.)

- Speedup:
$$S_p = \frac{T_s}{T_p}$$
 - p = # processes
 - T_s = execution time of the parallel algorithm on a single algorithm on a single process
 - T_p = 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-defined 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/>)



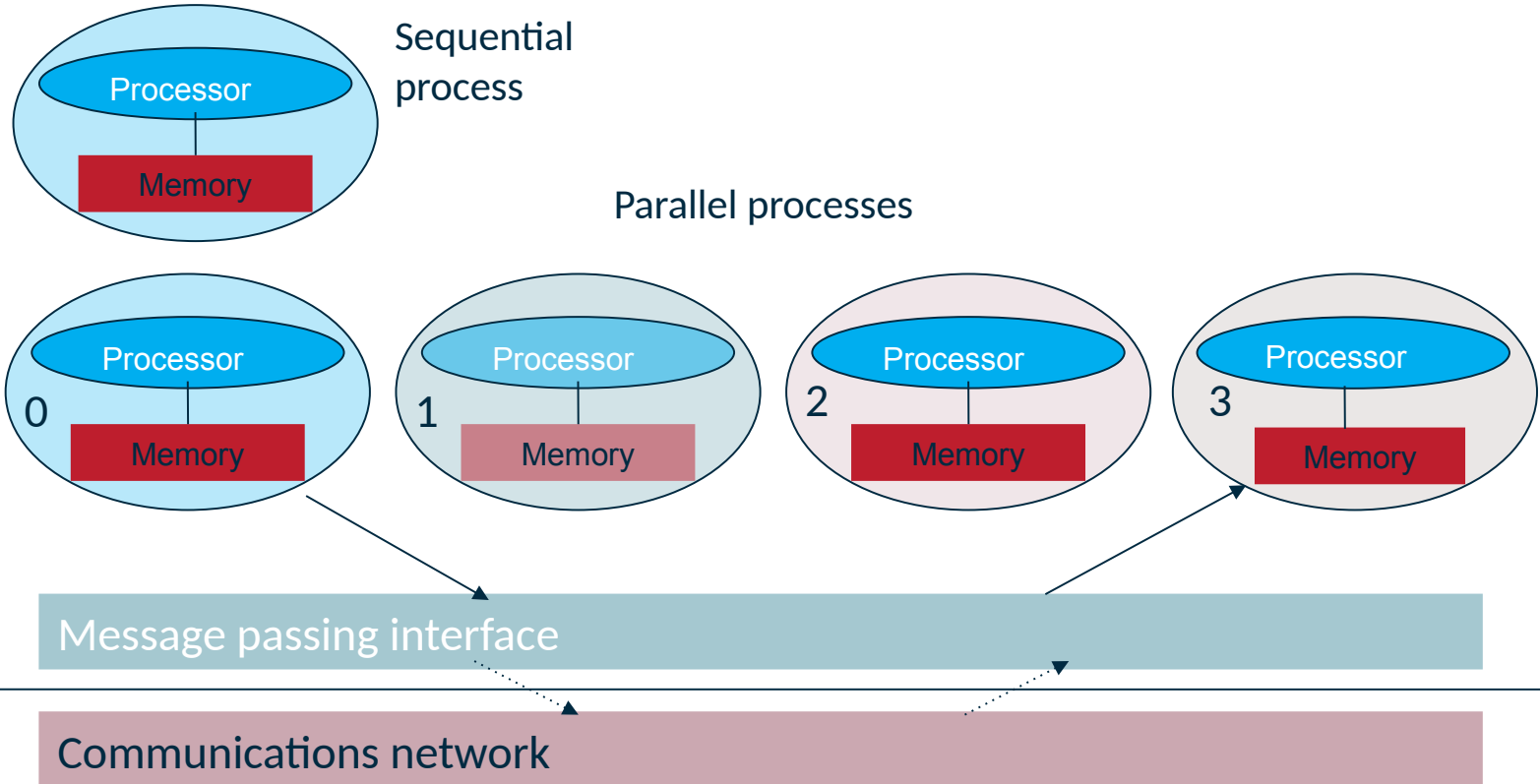
B. Some useful advice for programming on MPI

- 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

