Distributed Memory

We now turn to distributed memory programming
Distributed Memory

We now turn to distributed memory programming

We could use interfaces like threads or OpenMP and have an underlying or virtualising infrastructure that converts them to message passing between processors over a network.
Distributed Memory

We now turn to distributed memory programming

We could use interfaces like threads or OpenMP and have an underlying or virtualising infrastructure that converts them to message passing between processors over a network

Good programmers don’t like that as it hides the source of the cost of distributed parallelism from the programmer, making it harder to design and write efficient programs
Distributed Memory

We now turn to distributed memory programming.

We could use interfaces like threads or OpenMP and have an underlying or virtualising infrastructure that converts them to message passing between processors over a network.

Good programmers don’t like that as it hides the source of the cost of distributed parallelism from the programmer, making it harder to design and write efficient programs.

So most distributed programs are explicitly message passing, or have some other way of making the cost of an operation more clear.
Distributed Memory

The big player in this field is *Message Passing Interface* (MPI)
Distributed Memory

The big player in this field is *Message Passing Interface* (MPI)

You may hear about

- PVM: Parallel Virtual Machine, a predecessor to MPI
- SHMEM: SHared MEMory, only on Cray (SGI) machines
- UPC: Unified Parallel C, a supposed successor to MPI
MPI

MPI is what Big Science uses, when terabytes of data crunching is needed
MPI is what Big Science uses, when terabytes of data crunching is needed

And remember distributed systems are not good for small programs due to the overhead of the messaging outweighing the parallelism gained
MPI

MPI is what Big Science uses, when terabytes of data crunching is needed.

And remember distributed systems are not good for small programs due to the overhead of the messaging outweighing the parallelism gained.

MPI runs the same program on multiple processors (SPMD), but definitely not in lockstep.
MPI is what Big Science uses, when terabytes of data crunching is needed.

And remember distributed systems are not good for small programs due to the overhead of the messaging outweighing the parallelism gained.

MPI runs the same program on multiple processors (SPMD), but definitely not in lockstep.

The processes communicate via messages.
MPI

MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.
MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems
MPI

MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems

MPI is actually a standard with several competing implementations
MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems

MPI is actually a standard with several competing implementations

Code written to the standard should run on any implementation
MPI

MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems

MPI is actually a standard with several competing implementations

Code written to the standard should run on any implementation

But frequently doesn’t
MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems

MPI is actually a standard with several competing implementations

Code written to the standard should run on any implementation

But frequently doesn’t

The MPI standard specifies a huge number of functions, covering a wide range of different types of messaging
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int rc, myrank, nproc, namelen;
    char name[MPI_MAX_PROCESSOR_NAME];

    rc = MPI_Init(&argc, &argv);
    if (rc != MPI_SUCCESS) {
        printf("Error starting MPI program\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    continued
if (myrank == 0) {
    printf("main reports %d procs\n", nproc);
}

namelen = MPI_MAX_PROCESSOR_NAME;
MPI_Get_processor_name(name, &namelen);
printf("hello world %d from '%s'
", myrank, name);

/* implicit barrier in Finalize */
/*MPI_Barrier(MPI_COMM_WORLD);*/

MPI_Finalize();
return 0;
Notes:

• MPI_Init(&argc, &argv);
  Set up the system: you must always do this. A batch processing system (e.g., SLURM) starts the processes on all the processors, while MPI_Init sets up the connections between them.

• Later versions of MPI allow MPI_Init(NULL, NULL) but the above is preferable as it provides more information to the MPI system.

• rc
  Always check to make sure it worked.

• MPI_Comm_world
  The system can be sub-divided into subsets of processors called communicators. The WORLD communicator is all processors; MPI_Comm_self refers to just the calling processor.
Notes:

- **MPI_Init(&argc, &argv);** Set up the system: you must always do this. A batch processing system (e.g., SLURM) starts the processes on all the processors, while **MPI_Init** sets up the connections between them.
Notes:

- `MPI_Init(&argc, &argv);` Set up the system: you must always do this. A batch processing system (e.g., SLURM) starts the processes on all the processors, while `MPI_Init` sets up the connections between them.

- Later versions of MPI allow `MPI_Init(NULL, NULL)` but the above is preferable as it provides more information to the MPI system.
Notes:

- `MPI_Init(&argc, &argv);` Set up the system: you must always do this. A batch processing system (e.g., SLURM) starts the processes on all the processors, while `MPI_Init` sets up the connections between them.
- Later versions of MPI allow `MPI_Init(NULL, NULL)` but the above is preferable as it provides more information to the MPI system.
- `rc` Always check to make sure it worked.
Notes:

- `MPI_Init(&argc, &argv);` Set up the system: you must always do this. A batch processing system (e.g., SLURM) starts the processes on all the processors, while `MPI_Init` sets up the connections between them.
- Later versions of MPI allow `MPI_Init(NULL, NULL)` but the above is preferable as it provides more information to the MPI system.
- `rc` Always check to make sure it worked.
- `MPI_COMM_WORLD` The system can be sub-divided into subsets of processors called *communicators*. The `WORLD` communicator is all processors; `MPI_COMM_SELF` refers to just the calling processor.
• MPI_Comm_rank Each process in a communicator has a unique rank within that communicator: this is just an integer from 0 to *size of the communicator* − 1. So, for *WORLD* the rank ranges from 0 to *total number of processors* − 1
MPI

- **MPI_Comm_rank** Each process in a communicator has a unique rank within that communicator: this is just an integer from 0 to \textit{size of the communicator} – 1. So, for \texttt{WORLD} the rank ranges from 0 to \textit{total number of processors} – 1

- **MPI_Comm_size** Get the size of the communicator
• MPI_Comm_rank Each process in a communicator has a unique rank within that communicator: this is just an integer from 0 to *size of the communicator* – 1. So, for *WORLD* the rank ranges from 0 to *total number of processors* – 1

• MPI_Comm_size Get the size of the communicator

• if (myrank == 0) All processors run the same code (SPMD). This is how we get different things happening on different processors
MPI

- **MPI_Comm_rank** Each process in a communicator has a unique rank within that communicator: this is just an integer from 0 to *size of the communicator* – 1. So, for `WORLD` the rank ranges from 0 to *total number of processors* – 1
- **MPI_Comm_size** Get the size of the communicator
- **if (myrank == 0)** All processors run the same code (SPMD). This is how we get different things happening on different processors
- **MPI_Finalize** All procs must always call this to tidy up their MPI state
MPI

Compile using mpicc:

mpicc -Wall -o hellompi hellompi.c
Batch file runnit.slm:

#!/bin/sh
#SBATCH --account=cm30225
#SBATCH --partition=teaching
#SBATCH --job-name=HelloMPI
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=16
#SBATCH --time=00:01:00

mpirun -np $SLURM_NTASKS ./hellompi
You submit the job to SLURM using

sbatch runit.slm
Output:

```
hello world 3 from 'node-sw-154'
hello world 4 from 'node-sw-154'
hello world 5 from 'node-sw-154'
hello world 6 from 'node-sw-154'
hello world 7 from 'node-sw-154'
hello world 8 from 'node-sw-154'
hello world 9 from 'node-sw-154'
hello world 10 from 'node-sw-154'
hello world 11 from 'node-sw-154'
hello world 12 from 'node-sw-154'
hello world 13 from 'node-sw-154'
hello world 14 from 'node-sw-154'
hello world 15 from 'node-sw-154'
main reports 32 procs
hello world 0 from 'node-sw-154'
hello world 1 from 'node-sw-154'
hello world 2 from 'node-sw-154'
```
MPI

hello world 20 from 'node-sw-160'
hello world 22 from 'node-sw-160'
hello world 29 from 'node-sw-160'
hello world 18 from 'node-sw-160'
hello world 31 from 'node-sw-160'
hello world 23 from 'node-sw-160'
hello world 25 from 'node-sw-160'
hello world 28 from 'node-sw-160'
hello world 30 from 'node-sw-160'
hello world 16 from 'node-sw-160'
hello world 17 from 'node-sw-160'
hello world 19 from 'node-sw-160'
hello world 21 from 'node-sw-160'
hello world 24 from 'node-sw-160'
hello world 26 from 'node-sw-160'
hello world 27 from 'node-sw-160'
Notes:

- We used two nodes, using 16 cores on each
Notes:

- We used two nodes, using 16 cores on each.
- `ntasks-per-node` is important here as sometimes you want fewer MPI tasks on a node than there are cores on that node: an MPI task can itself be multithreaded (not your coursework!)
Notes:

- We used two nodes, using 16 cores on each.
- `ntasks-per-node` is important here as sometimes you want fewer MPI tasks on a node than there are cores on that node: an MPI task can itself be multithreaded (not your coursework!)
- `node-sw-154` and `node-sw-160` are the names of the two nodes that happened to be allocated; the next run may well get different nodes.
Notes:

- We used two nodes, using 16 cores on each
- `ntasks-per-node` is important here as sometimes you want fewer MPI tasks on a node than there are cores on that node: an MPI task can itself be multithreaded (not your coursework!)
- `node-sw-154` and `node-sw-160` are the names of the two nodes that happened to be allocated; the next run may well get different nodes
- Processes 0–15 are on `node-sw-154` while processes 16-31 are on `node-sw-160`, but it might happen the other way around
• Output in a random order, even for the “main reports 32 procs” which we might think happens first!
• Output in a random order, even for the “main reports 32 procs” which we might think happens first!
• We do see “main reports” before “hello world 0”, though!
• Output in a random order, even for the “main reports 32 procs” which we might think happens first!
• We *do* see “main reports” before “hello world 0”, though!
• MPI has a mechanism for routing prints on any node back via the network to a single point: this results in all kinds of timing variations in output
SLURM also gives us a few details on the job:

##############################################################
----------------------- Balena Job Details -------------------
JobID : 54160
No. of nodes : 2
No. of CPUs : 32
Time limit : 00:01:00
Elapsed time : 00:00:04
Nodelist : node-sw-[154,160]

##############################################################
• MPI is SPMD, so this code is *not* synchronised across processors
MPI

- MPI is SPMD, so this code is *not* synchronised across processors
- For example, when proc 0 is doing its `printf` the other processors may well already be doing `MPI_Get_processor_name`
MPI

- MPI is SPMD, so this code is *not* synchronised across processors
- For example, when proc 0 is doing its `printf` the other processors may well already be doing `MPI_Get_processor_name`
- Or perhaps still `MPI_Comm_size`
• MPI is SPMD, so this code is *not* synchronised across processors
• For example, when proc 0 is doing its `printf` the other processors may well already be doing `MPI_Get_processor_name`
• Or perhaps still `MPI_Comm_size`
• But many MPI calls do have a synchronisation and block the calling processor until all processors involved in that call are done
Exercise. Does adding a MPI_Barrier after the “main reports” conditional ensure the message comes out first?
In the batch file, `mpirun` sets up the processors and processes involved.
In the batch file, `mpirun` sets up the processors and processes involved.

Depending on the MPI implementation, this might be clever and sort out the best transport between them, e.g., in memory for processors on the same node and on the network for processors on different nodes.
In the batch file, `mpirun` sets up the processors and processes involved.

Depending on the MPI implementation, this might be clever and sort out the best transport between them, e.g., in memory for processors on the same node and on the network for processors on different nodes.

Or it might simply use network connections, regardless.
In the batch file, `mpirun` sets up the processors and processes involved.

Depending on the MPI implementation, this might be clever and sort out the best transport between them, e.g., in memory for processors on the same node and on the network for processors on different nodes.

Or it might simply use network connections, regardless.

The programmer uses the same MPI functions to send messages whatever the underlying mechanism.
You need to tell `mpirun` the number of processors so `MPI_Init` can set things up.
You need to tell `mpirun` the number of processors so `MPI_Init` can set things up.

In particular, `-np` needs to be the same as the number of cores requested by the batch file.
You need to tell `mpirun` the number of processors so `MPI_Init` can set things up

In particular, `-np` needs to be the same as the number of cores requested by the batch file

Fortunately, an environment variable `SLURM_NTASKS` gets calculated for us