This is 5611

We are now moving to 5611 where we will talk about parallel programming for a bit.

What is MPI?

- A message passing library specification
- MPI-3.1 document released in June 2015
- Not a new language — built on C or FORTRAN
- Many implementations: mpich, mvapich, openmpi, commercial options
- Used to program parallel computers, clusters etc.
- Designed to provide access to advanced parallel hardware in a portable format
- MPI code, written properly, should be able to compile on any machine and run successfully

Message Passing Model

- A process on a system has its own address space
- No other process (usually!) has access to it
- Within a process you can have multiple threads each sharing the address space
- MPI is used for communicating between processes, each with a separate address space
- This communication consists of
  - Copying of data from one process's address space to another process's address space
  - Synchronization

Getting started

- Your computer will need a version of MPI installed
- Add `#include <mpi.h>` to the top of your code
- Add `MPI_Init(&argc, &argv);` at the start of main()
- Add `MPI_Finalize();` to the end of main()
- Compile your code using the mpicc compiler
  - This is not actually a new compiler
  - It is a shell script that uses the standard compilers (gcc, pathcc, pgcc) but adds appropriate flags to pick up the MPI headers and library
- Submit your code to the queuing system and run it

Getting resources on Lonsdale

- We use the SLURM resource manager on Lonsdale to control access to compute nodes
- Paddy will go into a lot more detail on using SLURM
- For now we will just get some interactive nodes from the debug partition

```
lonsdale% module load default-gcc-openmpi-4.9.3-1.8.6
lonsdale% salloc -p debug -t 1:00:00 -n 16 bash
salloc: Job is in held state, pending scheduler release
salloc: Pending job allocation 38186
salloc: job 38186 queued and waiting for resources
salloc: job 38186 has been allocated resources
salloc: Granted job allocation 38186
dfrost@lonsdale01:~$
```

Hello MPI

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

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    printf("Hello world\n");
    MPI_Finalize();
}

lonsdale% mpicc simple.c
lonsdale% mpiexec -n 4 a.out
Hello world
Hello world
Hello world
Hello world
```

Who am I?

- Usually you will want to know how many processes are taking part in the computation and which one of them is which
- MPI provides two functions that address these questions
- `MPI_Comm_size( MPI_COMM_WORLD, &nprocs);`
- `MPI_Comm_rank( MPI_COMM_WORLD, &myid);`
- `MPI_COMM_WORLD` is a *communicator* which includes all the processes participating in the calculation. It is created by the system on your behalf.
- Like all good C code the ranks are counted from zero
- So in a 16 proc calculation they are numbered 0 to 15

Hello MPI version 2

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

int main(int argc, char *argv[]) {
    int nprocs, myid;
    char hostname[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_size( MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank( MPI_COMM_WORLD, &myid);
    gethostname(hostname, 100);
    printf("Hello world. I am proc %d of %d on %s\n",
        myid, nprocs, hostname);
    MPI_Finalize();
}
```

Synchronization

- By default, there is no synchronization in MPI programs
- Each process goes off and executes at its own pace
- The simplest way to force synchronization is using a barrier
- `MPI_Barrier(MPI_COMM_WORLD);`
- The faster executing processes wait at the barrier until all other processes have reached it
- These barriers slow down your code — only use them when you have to!

Data in MPI

- MPI datatypes are a wrapper around basic data types in C which allow portability between machines with different architectures
C Datatype	MPI Datatype
char	MPI_CHAR
int	MPI_INT
long int	MPI_LONG
float	MPI_FLOAT
double	MPI_DOUBLE

Exchanging Data

- Basic communication in MPI is carried out by co-operative point-to-point methods
- The data is explicitly sent by one process and explicitly received by another
- This means that processes can't mess with another's address space
- These communications are synchronized
- Use the following functions
  - `MPI_Send(...);`
  - `MPI_Recv(...);`
- Need to specifiy how much data, of what type and who you wish to send it to
- Messages also have a tag to help identify them. These are not so useful in synchronised communications.
- Usually set to 0 or else use `MPI_ANY_TAG` on the receiving end

MPI_Send

- `MPI_Send(start, count, datatype, destination, tag, communicator);`
  - Start is an address — usually a pointer to an array
  - Count is the number of items you wish to send
  - Datatype is as described previously
  - Destination is the rank of the process you wish to receive the data
  - Tag is as described previously
  - A communicator is a group of processes. We will just use the default `MPI_COMM_WORLD`, which contains all processes in the calculation, for the moment

MPI_Recv

- `MPI_Recv(start, count, datatype, source, tag, communicator, status);`
  - Start is an address — usually a pointer to an array
  - Count is the number of items you wish to receive
  - Datatype is as described previously
  - Source is the rank of the sending process
  - Tag is as described previously
  - A communicator is a group of processes. We will just use the default `MPI_COMM_WORLD`, which contains all processes in the calculation, for the moment
  - Status is a pointer to an `MPI_Status` data structure which contains further information
- Receiving fewer items than count is acceptable but receiving more is an error

Send and Receive example

```
int A[10];

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(rank == 0) {
    for(i=0;i<10;i++)
        A[i] = 100+i;
    MPI_Send(A, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if(rank == 1) {
    MPI_Recv(A, 10, MPI_INT, 0, MPI_ANY_TAG,
        MPI_COMM_WORLD, &status);

    // Should print 102
    printf("A[2] = %d\n", A[2]);
}
```

Send and Receive example

P0	5	14	−231	177	573	−53	92	43	−61	3
P1	0	0	0	0	0	0	0	0	0	0

P0	5	14	−231	177	573	−53	92	43	−61	3
P1	5	14	−231	177	573	−53	92	43	−61	3

Matching Sends and Receives

- Sends and Receives are matched on a triplet of information
  - Sender's rank matches Source in *MPI_RECV*
  - Receiver's rank matches Destination in *MPI_SEND*
  - Tags match
- MPI doesn't care if your datatypes don't match
- Need to be careful if there are several communications between sets of processes
- Will become more of a problem later

Send and Receive example 2

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(rank == 0) {
    sprintf(buffer, "Hello world!");
    len = strlen(buffer);
    MPI_Send(buffer, len, MPI_CHAR, 1, 0,
        MPI_COMM_WORLD);
} else if(rank == 1) {
    MPI_Recv(buffer, 200, MPI_CHAR, 0, 0,
        MPI_COMM_WORLD, &stat);
    printf("%d got message =>%s<=\n", rank, buffer);
}
```

Write any program now

- With the 7 basic MPI functions we have examined we can now write any parallel program we like
  - `MPI_Init`
  - `MPI_Finalize`
  - `MPI_Comm_size`
  - `MPI_Comm_rank`
  - `MPI_Barrier`
  - `MPI_Send`
  - `MPI_Recv`
- We will look at lots of other functions but each of them are just cleverer ways of carrying out operations that can be made up of these functions