Parallel Programming using the Message Passing Interface (MPI)

Prof Stephen Neethling

s.neethling@imperial.ac.uk

RSM 2.35

Shared and Distributed Memory Systems

- In this course you will be using both OpenMP (shared memory) and MPI (distributed memory) to parallelise code
- Shared Memory
  - All the cores have access to the same memory
  - Comparatively straightforward to turn serial code into parallel, but can be tricky to make efficient due to the need to minimise blocking when different cores want to access the same memory – Still need to seriously think about parallel aspects
  - Is restricted to the number of cores available on a single machine
- Distributed Memory
  - Can be run on clusters of separate machines
  - Need to swap information between the machines
  - Need to design the program from the outset to be parallel – Trying to directly convert a serial code is hard and often results in inefficient parallelisation
  - The Message Passing Interface (MPI) provides a library of functions and associated programs that allows for the exchange of information between processes in an efficient manner that removes the need to write low level networking code and provides a consistent framework even if the specific architecture changes
- Hybrid Systems
  - Most HPC systems consist of a large number of individual nodes each with multiple cores, with the nodes networked together
  - A hybrid approach in which OpenMP is used with shared memory on each of the nodes and with MPI used to communicate between the nodes can often be the most efficient approach for solving large problems
  - We will also look at remote memory access (RMA) and one-sided communications were some of the benefits of a hybrid system can be achieved within the MPI framework

Structure of the MPI section of this course

- First part of the course: Learn how to implement MPI based programs in C/C++
  - In this section we will concentrate on how to implement different aspects of MPI
  - The examples will seem a bit esoteric, but will be designed to illustrate individual aspects of MPI
- Second part of the course: Learn how to combine different aspects of MPI to solve different types of parallel problems
  - Develop longer programs that solve more realistic problems – Examples will, by necessity, still be somewhat simple
  - Different types of parallel architectures explored

Installing MPI library for Visual Studio

- These instructions are for Windows
  - James will be providing support in getting started for those of you using Macs
- You need to install Microsoft MPI
  - https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi
  - Click on “MS-MPI” under “Download” and download and install both “msmpisdk.msi” and “msmpisetup.exe”
- It will also be useful to add the location of “mpiexec.exe” to your system PATH
  - Open Window’s System Properties dialog box and click on Environment Variables
  - Click edit for the Path variable and add “C:\Program Files\Microsoft MPI\Bin\” (Assuming the default install location)

Including the Appropriate Libraries

- Create an “empty” project and add your source code to the project
- Once you have created the project you need to add the appropriate library
  - Right click on the project in the Solution Explorer and go to Properties
  - Click on VC++ Directories and (assuming the MPI SDK is installed in the default directory) add:
    - “C:\Program Files (x86)\Microsoft SDKs\Include” to the Include Directories
    - “C:\ Program Files (x86)\Microsoft SDKs \Lib\x64” or “C:\ Program Files (x86)\Microsoft SDKs \Lib\x86” to the Library Directories
      - Depends on whether you wish to compile your code as 32bit or 64bit
  - Click on Linker->Input
    - Under Additional Dependencies add “msmpi.lib”

A very basic MPI program

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

using namespace std;

int id, p;

int main(int argc, char *argv[])
{
	MPI_Init(&argc, &argv);

	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Comm_size(MPI_COMM_WORLD, &p);

	cout << "Processor " << id << " of " << p << endl;
	cout.flush();

	MPI_Finalize();
}
```

Header for all the MPI functions

main needs to take in arguments as they must be passed on to MPI_Init

Needs to be called to setup MPI communications

Reads the rank (number) of the current process

Reads the total number of processes that have been assigned

We will discuss later the idiosyncrasies of writing to stdout under MPI

Allows MPI to exit gracefully
Without this some communications on processes that have not yet finished running might not complete resulting in errors

An Important Note

- You need to think very differently about programs run with OpenMP compared to MPI
- In OpenMP you start a single copy of your program
  - Within that single program multiple threads will be spawned (and potentially go out of existence), but you have always got recourse to serial execution of certain sections of the code
- In MPI you start multiple copies of your program
  - The only way that one copy of your program (i.e. one process) knows what another copy is doing is through a communication.
  - There is no possibility of recourse to a serial section of code that knows everything
  - All aspects of the code are thus inherently occurring in parallel
    - You are running a (potentially large) number of completely separate but identical programs
    - The main way in which you can make the different copies of this program do different things is via the ids of the individual process (obtained using MPI_Comm_rank) and from subsequent communications

Running the Code under Windows

- Simply running this under Visual Studio will run it on a single core
  - Not very interesting!
- Build the code (“Build->Build Solution”), but don’t run it
- Open a Command Prompt and change directory to the location of the created executable
  - “*Project_Name*\Debug” if compiled as an x86 program under Debug mode
- Use “mpiexec” to run the code over multiple cores
  - mpiexec –n *#cores Project_Name*
- The *#cores* specified can be any number
  - Less than or equal to the number of cores available for efficient execution
  - Can specify more than the available cores to test functionality
  - “mpiexec” does recognise virtual cores (hyperthreading), but not as efficient as physical cores

Compiling and running under an HPC system

- James Percival will be coordinating and running an HPC based session this afternoon in which he will look at Makefiles etc.
  - In this course we are going to compiling and running examples on our local machines
  - …, but the real power of MPI is that it allows you to run code over 1000s of cores on hundreds of nodes if your problem demands it

- The main purpose of MPI is so that code can be run across multiple nodes such as on an HPC system
  - Get familiar with using the HPC system
  - Test the scaling of your code on more than the few cores available on your machines

Compiling under an HPC system

- Under Linux it is useful to use a Makefile to compile code
  - Allows you to easily include extra libraries and dependencies without having to type them in each time you compile something
- Create a file called “Makefile” in the directory where you have your *.cpp file and make it contain the following text:

```
MAKE   = make
TARGET = my_code
SOURCE = Example_1.cpp

default:
	mpicxx -o $(TARGET) $(SOURCE)
```

- You will need to load the following modules before you can use the Makefile:

```
module load intel-suite
module load mpi
```

  - You may find it useful to add it to your “.bash_profile” file
- To run the Makefile type “make”

Using a PBS script under HPC

- Note that this is about the simplest possible script
  - Later on you will need to add the ability to copy files between the temporary directories on the machines on which the processes will be running and your work directory
  - For the size of jobs (cores and time) that you can use see (https://www.imperial.ac.uk/admin-services/ict/self-service/research-support/rcs/computing/job-sizing-guidance/)
- Create a file "*my_script*.pbs" which contains the following text:

```
#PBS -N job_name
#PBS -l walltime=1:00:00
#PBS -l select=2:ncpus=32:mpiprocs=32:mem=1GB
#PBS -q queue_name

module load intel-suite
module load mpi

mpiexec ~/MPI_EXAMPLE/my_code
```

Name of the job (listed when using e.g. "qstat")

Maximum execution time (1 hour in this case)

select gives the number of machines requested and ncpus the number of cores per machine (the product of these will be the total number of processes used)

This allows the name of the queue to be specified. If this is not specified, the job will be assigned to an appropriate public queue

Modules should be loaded on the remote machines

As this is run on the remote machines you need to include the path for your code

Using a PBS script under HPC

- Once you have created the pbs script you should submit the job

  qsub *my_script*.pbs
- To check on the progress of your jobs use qstat
  - qstat or qstat –a gives all the information on your jobs (slightly different info)
  - qstat –Q gives information about the queues (useful to see how many other jobs are waiting to run)
- Your job will have an identifier associated with it of the form *number.pbs* (e.g. 2299941.pbs)
  - This can be used to kill a running or queueing job – qdel *number*.pbs
- After completing two files will be automatically generated
  - *job_name.onumber* – contains text sent to stdout
  - *job_name.enumber* – contains text sent to stderr

Writing to stdout

- Writing to stdout under MPI should be done with care
- There are a few things to note:
  - All output is sent back to the originating process for output
    - Can be slow as it is being sent over the network – unless debugging limit the amount that you write
  - It is often buffered
  - The order of output is correct for a single process, but not necessarily between processes
    - Beware when debugging, crashes might not occur where you think they have
    - Because it is often buffered and sent later, output from before a crash can go missing
      - Use cout.flush(); to try and minimise this issue
  - Unless for debugging purposes avoid sending output to stdout from processes other than zero
    - By sending to stdout from a single rank the output is in the correct order
    - In most MPI implementations rank zero is also where stdout is collated and so only writing from rank zero reduces network communications.

MPI_COMM_WORLD

- In MPI all communications require a communicator, which is essentially a list of processes involved in the communication
- It is possible to set your own up
  - Used if you have, for instance, different processes doing different tasks
- MPI_COMM_WORLD is a pre-defined communicator that includes all the processes
  - In this course we will only be having all the cores do the same thing (except for possibly the zero process) and therefore will only use MPI_COMM_WORLD

MPI_Barrier

- Because there is no guarantees about the order in which data sent to stdout by different processes is written out, it would be useful for debugging purposes that a message, for instance, written out when every process has passed a certain point
- MPI_Barrier allows you to do that as it requires every process to get to that point before it continues
  - It can thus be useful to call MPI_Barrier and then write an output on only a single process (usually zero) as you can be guaranteed that every process has got to that point
  - This is best done for debugging purposes – Having MPI_Barriers in your code can make it inefficient (processes unnecessarily waiting for other processes)
- Remember that MPI_Barrier must be in a portion of the code that every process gets to (and gets to in the same order/same number of times)
  - If not the code will hang as some processes will wait forever for the other processes to get to that point

An example with MPI_Barrier

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

using namespace std;

int id, p;

int main(int argc, char *argv[])
{
	MPI_Init(&argc, &argv);

	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Comm_size(MPI_COMM_WORLD, &p);

	cout << "Processor " << id << " of " << p << endl;
	cout.flush();

	MPI_Barrier(MPI_COMM_WORLD);
	if (id == 0) cout << "Every process has got to this point now!" << endl;

	MPI_Finalize();
}
```

In the next lecture we will actually send some information between the processes…