Parallel Programming using the Message Passing Interface (MPI)

Prof Stephen Neethling

s.neethling@imperial.ac.uk

RSM 2.35

Shared and Distributed Memory Systems

- Thus far you have implemented parallel programs using OpenMP
  - 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
- We want to run problems on clusters of machines that don’t share memory
  - 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

Structure of the MPI section of this course

- First half of course: Learn how to implement MPI based programs in C/C++ under both Windows and Linux
    - 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 half of 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

- You need to install Microsoft MPI
  - https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi
  - Click on Downloads 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:\MS_mpi\Bin\” (Again assuming the default install location)

Including the Appropriate Libraries

- The MPI code should be compiled as a Console Application
- 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:\MS_mpi\Include” to the Include Directories
    - “C:\MS_mpi\Lib\x64” or “C:\MS_mpi\Lib\x86” to the Library Directories
      - Depends on whether you wish to compile it as 32 or 64bit (32 bit is default under Visual Studio)
  - 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

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 under cx1

- 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 to run your code under cx1

- 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
- Create a file "*my_script*.pbs" in the same directory as your program which contains the following text:

```
#PBS -N job_name
#PBS -l walltime=1:00:00
#PBS -l select=3:ncpus=12:mem=1GB
#PBS -q queue_name

module load intel-suite
module load mpi

mpiexec ~/my_path/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 to run your code under cx1

- 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 –a gives all the information on your jobs
  - 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*.cx1 (e.g. 2299941.cx1)
  - This can be used to kill a running or queueing job – qdel *number*.cx1
- 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 bank to the originating process for output
  - 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
    - Rank zero should be on the machine originating process and thus does not involve network communication

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
- 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 section we will actually send some information between the processes…