Image Sharpening 1

Practical Introduction to HPC

1 Aims

The aim of this exercise is to get you used to logging into an HPC resource, using the command line and an editor to manipulate files, and using the batch submission system. We will be using Cirrus for this exercise. Cirrus is one the UK's EPSRC-funded national Tier-2 HPC services, operated by EPCC at the University of Edinburgh, and is an SGI ICE XA system with a total of 10,080 cores (280 x 36-core nodes).

You can find more details on Cirrus and how to use it in the User Guide at: http://www.cirrus.ac.uk/docs/

2 Introduction

In this exercise you will run a simple program, in serial and parallel, to sharpen the provided image. Using your account, you will:

1. log onto the Cirrus frontend nodes;
2. copy the source code from Learn to your account using the ***scp*** command;
3. unpack the source code archive;
4. compile the source code to produce an executable file;
5. run a serial job on the login node;
6. submit a serial job to the compute nodes using the Slurm batch system;
7. submit a parallel job using the Slurm batch system;
8. run the parallel executable using an increasing number of cores and examine the performance improvement.

Please do ask questions in the tutorials or on the Teams channel if you do not understand anything in the instructions.

3 Instructions

3.1 Log into Cirrus frontend nodes and run commands

You should use your Cirrus username, your ssh key's passphrase and the MFA token (TOTP code) to log into Cirrus.

3.1.1 Procedure for Mac and Linux users

Open a command line *Terminal* and enter the following command:

```
ssh -X user@login.cirrus.ac.uk
```

You should be prompted first to enter your passphrase and then to enter your TOTP code. The –X flag allows remote programs to open up a new GUI window (e.g. when viewing output images).

3.1.2 Procedure for Windows users

Windows does not generally have SSH installed by default so some extra work is required. The following steps should help you with that (if you have MobaXterm installed skip to the first step):

1. Download MobaXterm
2. Launch it and start a “New Session”
3. Select “SSH” as a session type
4. Specify “login.cirrus.ac.uk” as the remote host and add your username
5. You connection will be saved on the left sidebar, so you don’t need to repeat the above steps. You can just start your session by clicking on the saved remote host.
6. When you start a new session, you will get a prompt to enter your passphrase and the TOTP code.
7. You are ready to use Cirrus.

The following demo may also be useful – https://mobaxterm.mobatek.net/demo.html.

3.2 Download and transfer the exercise file

Download the tar ball (*sharpen.tar.gz*) containing the source files from Learn and use the ***scp*** command (if you are using Linux or Mac) or drag-and-drop functionality of MobaXterm (if you are using Windows) to transfer them to Cirrus.

Note: the drop-and-drop in MobaXterm refers to moving files within the MobaXterm graphical interface itself, and not moving files between the MobaXterm interface and your desktop or another location on your machine.

To use the ***scp*** command, open your terminal and go to the directory containing the downloaded file, then type in on one line (using your username and correct project code):

```
scp sharpen.tar.gz
username@login.cirrus.ac.uk:/work/project_code/project_code/user
name/
```

Note: we are transferring the files directly to the /work directory, we could also transfer them to the /home directory.

The first argument specifies the name and location of the file being transferred from a system (can be both local or remote) and the second specifies the location the file is being transferred to (again, can be either local or remote). In this case, we are transferring the *sharpen.tar.gz* archive from the folder we are currently in (hence no path is specified just the file name) to our work directory on Cirrus.

It will prompt you for your passphrase and then transfer the file:

```
Enter passphrase for key 'your_location/.ssh/id_rsa':
username@login.cirrus.ac.uk's password:
sharpen.tar.gz                     100% 3119KB 779.1KB/s   00:04
```

Note: we could also transfer the file in the other direction e.g. from our home directory on Cirrus to the directory we are currently in on our local machine (the dot at the end indicates the current working directory):

```
scp
username@login.cirrus.ac.uk:/work/project_code/project_code/username/sharpen.tar
.gz .
```

3.3 Extract the exercise files

The file you downloaded is actually a tar archive (*tarball*), which really is a number of files collected together and compressed for the ease of distribution. To get access to those files you need to unpack the tarball using the following command (assuming you are in the same directory as the tarball):

```
tar -xzvf sharpen.tar.gz
```

The option specified by the letter 'v' means 'verbose' and so you will see all the files that are being unpacked:

```
sharpen/C-SER/
sharpen/C-SER/filter.c
...
sharpen/F-OMP/dosharpen.f90
sharpen/F-OMP/Makefile
sharpen/F-OMP/fuzzy.pgm
...
```

Note: if the archive was partially unpacked when you downloaded it (i.e. it became sharpen.tar) omit the letter z from the above command (i.e. *–xvf*).

If you are interested in the C examples move to the *C-SER* subdirectory; for Fortran, move to *F-SER*. For example:

```
cd sharpen/C-SER
```

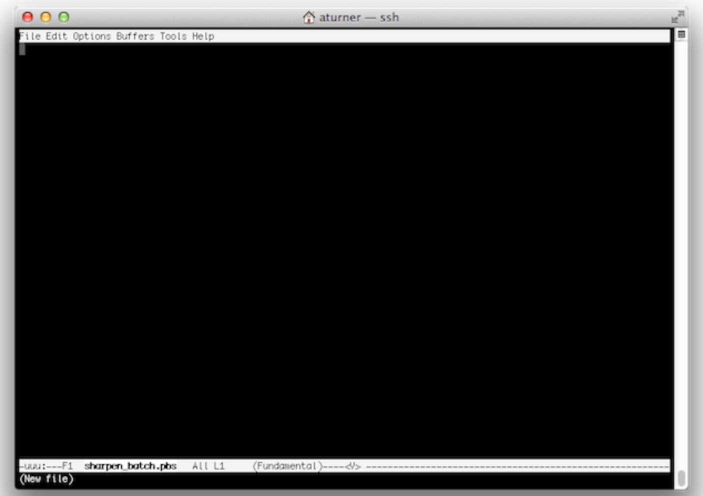
3.4 Using the Emacs text editor

Running interactive graphical applications on Cirrus can be very slow. It is therefore best to use Emacs in *in-terminal* mode. In this mode you can edit the file as usual but you must use keyboard shortcuts to run operations such as "save file" (remember, in this mode there are no menus that can be accessed using the mouse).

Start Emacs with the *emacs -nw* command and the name of the file you wish to edit or create. For example:

```
emacs -nw sharpen.slurm
```

The terminal will change to show that you are now inside the Emacs text editor:



```
-uuu:---F1  sharpen_batch.pbs   All L1     (Fundamental)----<V> ----------------------------------------------------------
(New file)
```

Typing will insert text as you would expect and backspace will delete text. You use special key sequences (involving the Ctrl and Alt buttons) to save files, exit Emacs and so on.

Files can be saved using the sequence "Ctrl-x Ctrl-s" (usually abbreviated in Emacs documentation to "C-x C-s"). You should see the following briefly appear in the line at the bottom of the window (the minibuffer in Emacs-speak):

```
Wrote ./sharpen.slurm
```

To exit Emacs and return to the command line use the sequence "C-x C-c". If you have changes in the file that have not yet been saved Emacs will prompt you (in the minibuffer) to ask if you want to save the changes or not.

Although you could edit files on your local machine using whichever windowed text editor you prefer it is useful to know enough to use an in-terminal editor as there will be times where you want to perform a quick edit that does not justify the hassle of editing and re-uploading.

You don't have to use Emacs, you use other text editors available on Cirrus e.g. vi, vim or nano.

3.5 Useful commands for examining files

There are a couple of commands that are useful for displaying the contents of plain text files on the command line that you can use to examine the contents of a file without having to open in in Emacs (if you want to edit a file then you will need to use Emacs). The commands are ***cat*** and ***less***. *cat* simply prints the contents of the file to the terminal window and returns to the command line. For example:

```
cat sharpen.slurm
...
```

This is fine for small files where the text fits in a single terminal window. For longer files you can use the ***less*** command: *less* gives you the ability to scroll up and down in the specified file. For example:

```
less Makefile
```

Once in *less* you can use the spacebar to scroll down and 'u' to scroll up. When you have finished examining the file you can use 'q' to exit *less* and return to the command line.

This program takes a fuzzy image and uses a simple algorithm to sharpen the image. A very basic parallel version of the algorithm has been implemented which we will use in this exercise. There are a number of versions of the sharpen program available:

- C-SER Serial C version
- F-SER Serial Fortran version
- C-MPI Parallel C version using MPI
- F-MPI Parallel Fortran version using MPI
- C-OMP Parallel C version using OpenMP
- F-OMP Parallel Fortran version using OpenMP

3.6 Compile the source code to produce an executable file

Before you can compile and run programs on Cirrus you must load appropriate modules. We are going to use the Intel compiler and the HPE Message Passing Toolkit (MPT), to load both of them type:

```
module load mpt
module load intel-20.4/compilers
```

To see what modules are available on the system you can use the ***module avail*** command, and to see what modules are currently loaded use the ***module list*** command.

We will first compile the serial C version (using the *make* command) of the code. Go inside the *sharpen/C-SER* subdirectory (using ***cd*** command), see what files are there (***ls*** command) and then compile the code using the ***make*** command.

Input:
```
ls
```

Output:
```
cio.c        filter.c   Makefile   sharpen.h    utilities.c
dosharpen.c  fuzzy.pgm  sharpen.c  sharpen.slurm  utilities.h
```

Input:
```
make
```

Output:
```
icc -O3 -DC_SERIAL_PRACTICAL -c sharpen.c
icc -O3 -DC_SERIAL_PRACTICAL -c dosharpen.c
icc -O3 -DC_SERIAL_PRACTICAL -c filter.c
icc -O3 -DC_SERIAL_PRACTICAL -c cio.c
icc -O3 -DC_SERIAL_PRACTICAL -c utilities.c
icc -O3 -DC_SERIAL_PRACTICAL -o sharpen sharpen.o dosharpen.o
filter.o cio.o utilities.o
```

This should produce an executable file called *sharpen*. For the Fortran version, the process is exactly the same as above, except you should move to the *F-SER* subdirectory and build the program there:

Input:
```
       cd sharpen/F-SER
       make
```

As before, this should produce a *sharpen* executable. Don't worry about the C file *utilities.c* – it is just providing an easy method for printing out various information about the program at run time, and it is most easily implemented in C rather than Fortran.

3.7 Running a serial job

You can run this serial program directly on the login nodes, e.g.:

Input:
```
       ./sharpen
```

Output:
```
       Image sharpening code running in serial
       Input file is: fuzzy.pgm
       Image size is 564 x 770
       Using a filter of size 17 x 17

       Reading image file: fuzzy.pgm
       ... done

       Starting calculation ...
       ... finished

       Writing output file: sharpened.pgm
       ... done

       Calculation time was 5.579000 seconds
       Overall run time was 5.671895 seconds
```

Note: Please remember that the login nodes are shared between all the users and so usually you are not supposed to run anything on the login nodes. In this case, the program only takes seconds to run so it doesn't matter too much. In the case of parallel code, it is impossible to run it on login nodes. The instructions on how to run jobs on the compute nodes are below.

3.8 Viewing the images

To see the effect of the sharpening algorithm, you can view the images using the display program from the ImageMagick suite. First you need to load the necessary module:

```
       module load ImageMagick
```

Input:
```
       display fuzzy.pgm &
       display sharpened.pgm &
```

It may take a while for the images to display so be patient. To close the program type "q" in the image window, click on the x in the left upper corner of the image or use the *Ctrl+C* command in the terminal window. The '&' symbol will make any command run in the background. In this case it will allow us to open multiple images and still have an active terminal. You can omit it and see what happens.

To view the image you will need an X window client installed. Linux or Mac systems will generally have such a program available, but Windows does not provide X windows functionality by default. MobaXterm allows viewing the image on Windows.

If you are using Mac system you may need to install XQuartz package to allow for SSH with X11 forwarding (i.e. opening additional windows for remote programs, for example to display output images).

For convenience, both images (before and after sharpening) are included below.

3.9 Running on the compute nodes

As with other HPC systems, use of the compute nodes on Cirrus is mediated by the job submission system. This is used to ensure that all users get access to their fair share of resources, to make sure that the machine is used as efficiently as possible, and to allow users to run jobs without having to be physically logged in.

Whilst it is possible to run interactive jobs (jobs where you log directly into the backend nodes on Cirrus and run your executable on the command line there) on Cirrus. While they are useful for debugging and development, they are not ideal for running long and/or large numbers of production jobs as you need to be physically interacting with the system to use them.

The solution to this, and the method that users generally use to run jobs on systems like Cirrus, is to run in a *batch* mode. In this case you put the commands you wish to run in a file (called a job script) and the system executes the commands in sequence for you with no need for you to be interacting. You will learn more about the batch schedulers later in the course. Cirrus uses the Slurm software to schedule jobs.

3.9.1 Using Slurm job scripts

We will first run the same serial program on the compute nodes. Look at the batch script *sharpen.slurm* inside the C-SER or F-SER directory:

Input:
```
    emacs -nw sharpen.slurm
```

The first line specifies which *shell* to use to interpret the commands we include in the script. Here we use the Bourne Again SHell (bash), which is the default on most modern systems.

The #SBATCH lines provide options to the job submission system where "-ntasks" specifies the number of processes. This is a serial job so ntasks is set to 1.

The "--time=00:01:00" sets the maximum job length to 1 minute and "--job-name=sharpen" sets the job name to "sharpen".

We also need to specify the budget to charge the job to the correct project – this is done using the "--account=" option. If you have your own personal budget, then you must set the account option to be your own personal budget within the right project, e.g., either "m22ol-your_username" or "m22ext-your_username". Otherwise, simply set the account option to be your project name, e.g., tc063.

Additionally, we are using the CPU nodes by requesting the standard partition, i.e., "--partition=standard" and the job is short so we can use the short queue by setting "--qos=short".

The remaining lines are the commands to be executed in the job. First the modules necessary to run the executable are loaded, then the *cd $SLURM_SUBMIT_DIR* command changes the directory to the submission directory (i.e. the directory the job was submitted from) and then the sharpen executable is run.

3.9.2 Submitting scripts to Slurm

Simply use the ***sbatch*** command:

Input:
```
    sbatch sharpen.slurm
```

Example output:

```
    Submitted batch job 56170
```

The number returned from the *sbatch* command (e.g. 561870 above) is the jobID, which is used as part of the name of the output file discussed below and is employed when you want to delete the job (for example, you have submitted the job by mistake).

3.9.3 Monitoring/deleting your batch job

The Slurm command *squeue* can be used to examine the batch queues and see if your job is queued, running or complete. *squeue* on its own will list all the jobs on Cirrus (usually hundreds) so you can use the "-u $USER" option to only show your jobs:

Input:
```
    squeue -u $USER
```

Output:

```
 JOBID PARTITION     NAME     USER  ST       TIME  NODES NODELIST(REASON)
   56170  standard  sharpen     user  R        0:06      1   r1i0n16
```

The letter under the ST column gives the status of the job. "PD" means your job is awaiting resource allocation (PENDING), "R" that it's running, "CG" means it's in the process of completion and "CD" means it's completed. There are also other job state codes and we will cover them later.

If you want to delete a job, you can use the *scancel* command with the jobID. For example:

Input:

```
scancel 56170
```

3.9.4 Finding the output

The job submission system places the standard output text and standard error text from your job into a single file: *slurm-<jobID>.out*. This file appears in the job's working directory once your jobs starts running.

To view the file's content type:

```
cat slurm-56170.out
```

Output:

```
Image sharpening code running in serial
Input file is: fuzzy.pgm
Image size is 564 x 770

Using a filter of size 17 x 17

Reading image file: fuzzy.pgm
... done

Starting calculation ...
... finished

Writing output file: sharpened.pgm
... done

Calculation time was 4.601818 seconds
Overall run time was 4.729456 seconds
```

4 Running in Parallel

To run a parallel job we need to use a job scheduler to submit it to the compute nodes. Depending on the version of the code (i.e. MPI or OpenMP) we need to use slightly different options in our slurm script.

4.1 Running an MPI parallel job on the compute nodes

Let's start with the MPI version – it uses the massage-passing interface (MPI) to parallelise the code. You will learn more about MPI and another application programming interface called OpenMP (see Section 4.3) later in the course. For now you just need to understand how to change the number of processes and hence the number of cores used to run our code.

When you open the sharpen.slurm batch script, in either the C-MPI or F-MPI directory, you will notice there is more SBATCH options than for the serial version. By default, the batch script is set up to run on 4 cores on a single node. This is specified by the following options:

- ***--nodes=1*** – sets the number of requested compute nodes to 1. Remember each CPU node on Cirrus has 36 cores.
- ***--ntasks=4*** – sets the total number of MPI processes to 4. This options accounts for MPI processes across all requested nodes.
- ***--tasks-per-node=4*** – sets the number of MPI processes per node. There are only 36 cores on each node so this number shouldn't be larger than 36.
- ***--cpus-per-tasks=1*** – sets the number of threads per MPI processes. For a pure MPI version this will be set to 1.

Feel free to submit a range of parallel jobs using the MPI version of the code in *C-MPI* or *F-MPI*. Remember that if you want to run on more than 36 cores then you will need to request more than one node with the "--nodes=" options. Also, make sure the values for ntasks, tasks-per-nodes and nodes are compatible e.g. do not set the number of tasks to greater than tasks-per-nodes*nodes.

4.2 Running an OpenMP parallel job on a compute node

Run the OpenMP code as well, which are found in either the C-OMP or F-OMP directories. OpenMP takes advantage of the fact that the memory within a single node is shared amongst all the cores. This means it can only run on 1 node at most. The changes to the Slurm script include:

- Setting ***nodes=1***
- Setting ***tasks-per-node=1*** – we are not using any MPI processes.
- Changing the number of ***cpus-per-tasks*** depending on how many OpenMP threads (hence cpu-cores) we want to use.
- Making sure the variable ***OMP_NUM_THREADS*** is equal to the number of ***cpus-per-tasks***.

Note that you can run using multiple threads on the login nodes but you must set *OMP_NUM_THREADS* before you run, e.g.:

```
export OMP_NUM_THREADS=4
./sharpen
```

The maximum number of cores you can use with OpenMP is 36.

4.3 Timings

If you examine the log file you will see that it contains two timings: the total time taken by the entire program (including IO) and the time taken solely by the calculation. The image input and output is not parallelised so this is a serial overhead, performed by a single processor. The calculation part is, in theory, perfectly parallel (each processor operates on different parts of the image) so this should get faster on more cores.

You should run a number of jobs on different number of cpu-cores (e.g. on 1, 4, 8 cores) to see how the results compare. Using the two timing results calculate:

- IO time – the difference between the calculation time and the overall run time; **Note:** this difference covers more than just the IO operations, but other operations shouldn't be significant contributors to the overall execution time.
- total CPU time - the calculation time multiplied by the number of cores.

Look at your results – do they make sense? Given the structure of the code, you would expect the IO time to be roughly constant, and the performance of the calculation to increase linearly with the number of cores: this would give a roughly constant figure for the total CPU time. Is this what you observe?