Exercise: Running a Simple Multicore Program

David Henty

1 Introduction

The basic aim of this exercise is to familiarise you with compiling and running parallel programs on HECToR. The program performs a simple form of image processing to try and sharpen up a fuzzy picture. You will be able to measure the time taken by the code on different numbers of cores to check that the execution time decreases with processor count as expected.

2 Compiling the code

On HECToR, you should do all the exercises in the `work` subdirectory of your home directory. All codes are supplied in a single compressed tar file called `XE6workshop.tar.gz`.

```
user@hector> cd work
user@hector> cp /home/d26/d26/guestadm/XE6workshop.tar.gz .
user@hector> tar xvf XE6workshop.tar.gz
```

In the `sharpen` directory, you will see six subdirectories containing versions of the code in C and Fortran, each parallelised with MPI, OpenMP and a hybrid of MPI and OpenMP. You can opt to work with whichever language you are most familiar with.

The following text examples assume you are using the OpenMP version of the code written in C. You will see similar (but not identical) output if you are using one of the other versions.

Now just change directory and use `make`:

```
user@hector> cd sharpen
user@hector> cd C-OMP
user@hector> make
```

which creates the executable program `sharpen`.

You can look at the input file using the `display` program, but you will have to update your `PATH` first:

```
user@hector> export PATH=$PATH:/work/d26/d26/guestadm/bin
user@hector> display fuzzy.pgm
```

and type `q` anywhere in the display window to quit `display`.

3 Running the Code

You cannot run jobs interactively on HECToR – you must submit jobs to the PBS batch system.

```
user@hector> qsub submit.sh
```

When the code has completed running it will produce an output image `sharp.pgm` and a log file `submit.sh.oXXXXX` which contains, among other things, information about the execution time (where XXXXX will be the job number assigned by PBS). Compare the sharpened image to the original fuzzy image: does it look any sharper?

You can control how many OpenMP threads (and MPI processes) are used by altering the following parameters in `submit.sh`:

OpenMP `mppdepth` controls the number of OpenMP threads.

MPI `mppwidth` controls the number of MPI processes; `mppnppn` controls the number of processes per node, which is normally set to 32 for pure MPI programs.

Hybrid Here you must set the three variables above as appropriate to select how many processes you want per node and how many threads per process. Normally, you would aim for 32 threads per node so every core is used.

4 Parallel Performance

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 do a number of runs and fill in Table 1: the IO time is the difference between the calculation time and the overall run time; the total CPU time is 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?

# Cores	Overall run time	Calculation time	IO time	Total CPU time
1				
2				
4				
7				
10				

Table 1: Time taken by parallel image processing code

The parallel speedup should closely follow Amdahl's law as the calculation splits cleanly into a purely serial phase (IO) and a perfectly parallel phase (calculation).