Image Sharpening 2

Practical Introduction to HPC

epcc

1 Aims

The aim of this exercise is to quantify the parallel performance of the image sharpening code from the last week. You already have the source code and the necessary Slurm scripts.

In this exercises you should:

- Run both MPI and OpenMP versions of the code (for either C or Fortran versions) on a range of cores;
- Collect the timing data to fill Tables 1 and 2;
- Calculate and plot the ideal, overall and calculation speedup, and parallel efficiency (Tables 3 and 4);
- Analyse the performance of the code and compare it with your expectations.

2 Instructions

Remember that to run the exercise on different number of MPI processes or OpenMP threads you need to modify their respective Slurm scripts and submit your jobs (using the ***sbatch*** command) for execution on compute nodes.

You can confirm if your job run on a desired number of cores by checking the first line of the job output (i.e. the ***slurm-<JobID>.out*** file). For example running on 36 cores using MPI (processes) and OpenMP (threads) will give, respectively:

```
Image sharpening code running on 36 processor(s)

Image sharpening code running on 36 thread(s)
```

Note: remember you need to use your own budget code (e.g., **project_code** or ***project_code-your_username***) to be able to submit a job. You also need to be in your **/work** directory on Cirrus!

2.1 Changing the Slurm script

You should remember from last week how to control the number of MPI processes and OpenMP threads in a parallel job. You may need to modify the following options to the #SBATCH directive:

- ***--nodes*** – sets the number of nodes to be used for your job.
- ***--ntasks*** – sets the total number of parallel processes (across all nodes).
- ***--tasks-per-node*** – sets the number of parallel processes per node.
- ***--cpus-per-tasks*** – sets the number of threads per parallel processes.

Note: you must also set the OMP_NUM_THREADS environment variable when using OpenMP. The provided script includes a line which sets OMP_NUM_THREADS to the value of the *cpus-per-tasks* variable.

The ***--exclusive*** option ensures that you have exclusive access to a compute node. You can find more details about running jobs and different Slurm options in the Cirrus online documentation.

2.2 Timings

As you probably recall the log file 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 several jobs on a range of cores, determine the minimum time, and fill in Table 1 (for MPI) and Table 2 (for OpenMP). Remember that:

- 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.

# Cores	Overall run time	Calculation time	IO time	Total CPU time
1				
2				
4				
7				
10				
36				
72				
108				
144				

Table 1: Time taken by parallel image processing code – MPI version.

# Cores	Overall run time	Calculation time	IO time	Total CPU time
1				
2				
4				
7				
10				
12				
18				
24				
36				

Table 2: Time taken by parallel image processing code – OpenMP version.

Note: As explained previously, the IO time should be roughly constant regardless of the number of cores used. If you are observing significant differences in the IO time, then it's worth re-running the job. The additional time may be an artifact of the other processes running on the system. In performance analysis it is common to re-run the jobs several times to ensure the results are consistent and repeatable.

2.3 Speedup and Efficiency

Once you have completed the above tables, you should have the information required to compute the speedup and parallel efficiency. Remembering that the speedup is the ratio of runtime at 1 core compared to the runtime at *P* cores, and the efficiency is the ratio of

the speedup observed on P cores compared to the number of P cores, you should fill Tables 3 and 4.

# Cores	Ideal Speedup	Overall speedup	Calculation Speedup	Parallel Efficiency
1				
2				
4				
7				
10				
36				
72				
108				
144				

Table 3: Speedup for parallel image processing code – MPI version.

# Cores	Ideal Speedup	Overall speedup	Calculation Speedup	Parallel Efficiency
1				
2				
4				
7				
10				
12				
18				
24				
36				

Table 4: Speedup for parallel image processing code – OpenMP version.

2.4 Performance analysis

Once you have collected all the data, use a tool of your choosing to plot the results with respect to the number of cores used i.e. plot the number of cores on the x-axis and the speedup or efficiency on the y-axis.

Look at your results – do they make sense? What can you say about the performance of both codes? Look at your plots and answer the below questions:

- How do the overall and calculation speedup results compare to the ideal speedup?
- How do you explain the difference between the overall and calculation speedup?
- How does speedup change with the increasing number of cores? Can you explain the shape of the curve?
- What can you say about the parallel efficiency of both codes?
- What does parallel efficiency tell you about using a larger number of cores?
- Compare the single node performance of the MPI and OpenMP codes – what do you see? Can you explain the difference in their behavior?