HPC Workshops

Exercise 3

2019

Christian Arnold

MPI and queue systems (SLURM)

1) Calculating π in parallel

The directory `calculate_pi_serial` contains a code which allows you to calculate the numerical value of the constant *pi* using the relation between the area of a circle and a square:

$$A_{\text{circle}} = \pi * r^2 \qquad\qquad A_{\text{square}} = 4r^2, \tag{1}$$

if the circle fits exactly into the square. Considering a set of evenly distributed 2D random points (x, y) $(0 <= x, y < r)$ in the upper right corner of the square, the probability to find a point within the circle is

$$p_{\text{in}} = \frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{\pi}{4}. \tag{2}$$

Drawing a large set N_{total} of random points we can thus calculate the value of π by counting the number of points N_{circle} which are in the circle:

$$\pi_{\text{numerical}} = 4\frac{N_{\text{circle}}}{N_{\text{total}}}. \tag{3}$$

You can compile the code using the Makefile as before. It can be run calling
`./mycode N`
where N is the total number of random points (note that N is a 32-bit signed integer and must thus not exceed 2^{31}).

1. Compile the code with the default settings and run it for different choices of N to determine how the accuracy of the result depends on N.

2. Load the appropriate MPI-module on the cluster (`intelmpi/intel/2018.2` on Hamilton; `intel_mpi/2018` with `intel_comp/2018` on cosma), change the compiler in the Makefile to mpicc and include the mpi headers (`#include <mpi.h>`) in the `.c` files.

3. Include statements in the main routine of the code which initialise and finalise MPI, find the total number of tasks and the task ID for each task and output these in a print statement.

4. Now run the code on 2 cores using the `mpirun` command and check if the previous implementations work.

5. So far, the calculations are only carried out parallel, but each core does the same work (which is not very useful). Change the random number seed in `calculate_pi.c` such that the code does actually use different random numbers on each task and confirm this by (a few) printing individual numbers on each task.

6. Also, make sure that the total number of random points stays constant irrespective of the number of parallel tasks used.

7. To finalise your work on the source code, sum the number of points in the circle and the total number of points among the different tasks and use these numbers to calculate π. The `MPI_Allreduce` command might be useful.

8. Now test your code by running it on the login node on 2, 4, 8 and 16 cores. Plot the runtime against the number of cores.

2) Using SLURM

On big computing clusters (like Hamilton or Cosma) there are so called queueing systems in place to make sure that everyone gets a fair share of the total computing time available. So far, you have only been running programs interactively on the login-nodes. These are not meant to be used for expensive calculations, but merely to submit jobs (i.e. certain computing tasks) to the actual compute nodes. The queueing system is organised in different queues, which serve different purposes (we are going to use the test.q on hamilton and the cosma5 queue on cosma, depending which cluster you're logged into).

1. Use the `sinfo` command to show the different queues and their status.

2. Now have a look at the `slurm_script_hamilton.sh` job-script, which you will use to submit a job to the compute nodes. Instructions for the queueing system start with `#SBATCH` in the SLURM language.

3. Add the modules you compiled the code with to the script.

4. Add the correct command to run the program.

5. The script is now complete, submit it using `sbatch slurm_script.sh`

6. Check the status of all jobs using the `squeue` command; you can filter for your own jobs with `squeue | grep USERNAME` .

7. Run the job for different numbers of cores and have a look at the output files.