Workshop 7: The Mandelbrot set

The Mandelbrot set is the set of complex numbers which contains the points for which the iteration

$$z_{i+1} = z_i^2 + C \tag{1}$$

remains finite, where C is a complex number and the initial value $z_0 = C$. This apparently simple relation leads a fractal structure (see figure 1 which shows the Mandlebrot set plotted on the complex plane).

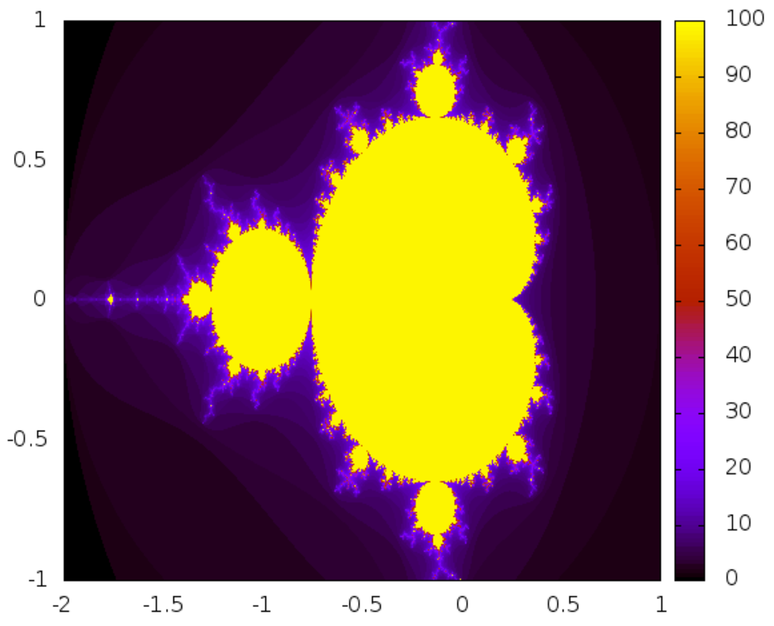

Figure 1: The Mandelbrot set

In this workshop you will measure the parallel speed-up of a C program which calculates the Mandelbrot set, and we will see how to use loop scheduling to improve the parallel performance. This workshop uses Isca (the university's HPC cluster).

1 Getting set up

Isca is physically located in the university data centre so it is accessed remotely by connecting to a log in node using SSH. On the module ELE page there are instructions describing how to log in to a remote Linux computer under the "Workshop 2" section.

- Start by logging in to Isca as described in the Workshop 6 instructions.
- The example programs for this workshop are provided as a tar file. There are two versions of the tar file (one for ECM3446 and one for ECMM461) Take a copy of the tar file with the correct module code by running either

```
cp /lustre/projects/Research_Project-IscaTraining/HPC/workshop7_ecm3446.tar .
```

or

```
cp /lustre/projects/Research_Project-IscaTraining/HPC/workshop7_ecmm461.tar .
```

where the dot at the end of the command causes the file to be copied into the current working directory. When you first log in to Isca the current working directory is your home directory, which is mounted on the log in nodes and all the compute nodes. There should be sufficient space in your home directory for all the workshops on this module.

- Unpack the tar file by running

  ```
  tar xvf workshop7_ecm3446.tar
  ```

  or

  ```
  tar xvf workshop7_ecmm461.tar
  ```

  This will create a directory called `workshop7` which contains the example program for this workshop.

2 Running the serial program

- Start by loading the Intel Cluster Toolkit module

  ```
  module load intel/2017b
  ```

- Then compile a serial version of the program

  ```
  cd workshop7
  gcc -o mandelbrot mandelbrot.c -lm
  ```

  where the `-lm` flag is required to link with the maths library and the `-fopenmp` flag is not included.

- When you have built the executable run it by submitting a job to the queue using the command

  ```
  sbatch run_mandelbrot.sh
  ```

  A file called `mandlebrot.dat` is generated when the program runs. This file is an ASCII file with three columns (Re, Im, Number of iterations).

- If you would like to view the output then you can plot the data in the `mandlebrot.dat` file using the gnuplot script `plot_mandelbrot`

  ```
  gnuplot plot_mandelbrot
  ```

  This generates a PNG file which can be copied to your local computer for viewing (to do this run `scp` or `pscp` on your local computer).

- The job script uses the `time` command to measure how long the program takes to run. The output from the time command can be found in the SLURM output and the total elapsed time is the “real” time. The program should take about 9 seconds to run.

3 Parallel performance

In this section we will measure the parallel performance of the Mandelbrot program. For this test we will use a larger problem size to increase the run time, and switch off the I/O so that we can measure the speed-up of the parallelised loop.

- Open the `mandelbrot.c` file in a text editor and increase the size of the calculation by changing `nRe` to 12000 and `nIm` to 8000.
- Next change the `doIO` parameter to `false` to switch off the IO.
- The first step of the scaling test is to measure the serial run time. Re-compile the program **without** OpenMP and submit a job to the queue to run the serial program again

  ```
  gcc -o mandelbrot mandelbrot.c -lm
  sbatch run_mandelbrot.sh
  ```

  The time for this version of the program to run is the T_0 value which will be used in calculating the parallel speed-up.

  Question 1: What is the value of T_0 for this version of the program?
- The next step is to compile an OpenMP executable. The program already contains the required OpenMP directives so you just to rebuild the executable and add the `-fopenmp` flag:

  ```
  gcc -fopenmp -o mandelbrot mandelbrot.c -lm
  ```
- The job script `run_mandelbrot_omp.sh` runs the program with 2, 4, 8, 12, and 16 OpenMP threads. Submit this job script to run the OpenMP executable for the scaling test:

  ```
  sbatch run_mandelbrot_omp.sh
  ```
- Look in the job output to see how long the program took to run with different numbers of threads. Use this information to calculate the parallel speed-up

  $$S_N = \frac{T_N}{T_0} \tag{2}$$

  for 2, 4, 8, 12, and 16 and threads.
- Plot the parallel speed-up (vertical axis) against the number of threads (horizontal axis) and compare this with the ideal speed-up.

  Question 2: Is the program making efficient use of the available resources?

4 Loop scheduling

In this final section we will investigate the effect of loop scheduling on parallel performance.

- Add a clause to the `#omp parallel for` which specifies a loop schedule and re-run the scaling test using static, dynamic and guided schedules. Remember to re-compile the executable after changing the schedule.

  Question 3: Which loop schedule performs best and why?

  Question 4: What is the default loop schedule?