School of Computing, University of Leeds, 2013–2014

Parallel Scientific Computing COMP3920

Coursework 1 – Computing the Mandelbrot Set

Deadline: 10:00am, Friday 1st November 2013

Introduction

This coursework is designed to give you practice implementing and assessing different partitioning strategies. The main program provided, `mandelbrot_cyclic.c` (along with the file `mandelbrot-gui.h`), implements a parallel solution for calculating and displaying the Mandelbrot set, in which the computation is distributed evenly between the worker processes in a cyclic manner and the master process displays the results. The program uses `Xlib` calls to create the graphics so to compile and run it on two processors you will need to type

```
    mpicc -Wall -o mandelbrot mandelbrot_cyclic.c -lX11
```

followed by

```
    mpiexec -n 2 ./mandelbrot
```

Compile and run the program to make sure that you know what the output from the code should look like. It should open a new window in which the Mandelbrot set will be drawn when you have pressed a key on the keyboard with the cursor inside the window. Pressing a key on the keyboard again will close the window and finish the program.

It is not crucial for you to fully understand the theory which underlies the algorithm used to create the Mandelbrot set (and you do not need to have studied complex numbers before) but the iteration is described here so that you can match it with the section of the code which executes it. It takes the form

$$z_{k+1} \;=\; {z_k}^2 + c$$

where $z = z_r + z_i\, i$ and $c = c_r + c_i\, i$ are both complex numbers. The properties of complex numbers mean that

$$z^2 + c \;=\; ({z_r}^2 - {z_i}^2 + c_r) + (2 z_r z_i + c_i)\, i\,,$$

which you should be able to match to the code inside the `while` loop contained in the function `worker_pgm`. The iteration is terminated either when $|z|^2 = {z_r}^2 + {z_i}^2$ becomes too large (the iteration is diverging) or a maximum number of iterations has been reached, whichever happens first. You are not expected to modify this iteration or the graphics (you should not need to modify `mandelbrot-gui.h`, or the functions `drawLine` and `main`, at all). However, you are asked to analyse the efficiency of different parallel versions of the code, for which it will be important to understand how the number of iterations (`count` in the supplied code) is used to generate the appearance of a given pixel.

As always, please ensure that you do not hinder anyone else's work when you use machines remotely and use `mpdallexit` when logging off.

Tasks

1. A serial version of the code, `mandelbrot_serial.c`, is also provided. Modify this code so that your program counts the total number of iterations taken to produce *each row of pixels* of the final image and writes this data to a file. This gives an approximation of the work required to produce each line of pixels on the screen. Construct a graph from this information which illustrates how the number of iterations varies with the row index for the region plotted. **[10 marks]**

2. Modify the parallel code `mandelbrot_cyclic.c` so that it uses the `MPI_Wtime` function to estimate the run-time of the code. Using this, construct a table which displays run-times for a range of values of `noprocs`. Make sure that you do not run more than one process per cpu core. **[5 marks]**

3. Modify your parallel code so that, instead of using cyclic decomposition, the computation is divided between the worker processes using strip/block decomposition. You should ensure that each process receives (as near as possible) the same number of rows of pixels to find colours for. Construct a second table, displaying run-times for a range of values of `noprocs` using this modified code. **[15 marks]**

4. Explain why cyclic decomposition is a significantly better approach to distributing the work than block decomposition in this situation. Hint: consider the graph you obtained for Task 1. **[5 marks]**

5. Modify your solution to Task 2 or Task 3 so that it uses work pool techniques to distribute the work efficiently between the processes. You may assume that each work package involves the computation of the colours for a single row. Construct a third table, displaying run-times for a range of values of `noprocs` using this new code. **[20 marks]**

6. Using the tables of results you have obtained from Tasks 2, 3 and 5, compare the relative speeds of each version and their scalability as `noprocs` is increased. Which is the best approach to distributing the workload in this situation? Explain why this is the case. **[10 marks]**

7. The parallel code `mandelbrot_zoom_cyclic.c` (with `mandelbrot_zoom-gui.h`), supplied with this coursework, also allows you to repeatedly zoom in on smaller regions of the Mandelbrot set. The instructions for its use are printed to the screen when you run the code. The code implements cyclic decomposition and you are *not* asked to modify it. Which of the three decomposition techniques considered above would you expect to be the best for this code? Explain your answer. **[10 marks]**

Total: 75 marks

Deliverables

Please submit your coursework in the form of a short report, via the postbox in the SSO. The report should include the following.

1. A graph illustrating workload distribution for Task 1, and tables of run-times for Tasks 2, 3 and 5.
2. A precise description of the modifications you have made to the original code to produce your solutions to Tasks 2, 3 and 5. It may be useful to include short sections of code to illustrate your modifications.
3. Your code for Task 5. You do not need to include `mandelbrot-gui.h`.
4. A discussion of the relative efficiency of the three decomposition techniques in response to Tasks 4, 6 and 7.