The Mandelbrot Set

COMP509 Project – Due 29th May 2015.

The Mandelbrot Set

The *Mandelbrot Set,* originally published by Benoit Mandelbrot in 1977 in his book *The Fractal Geometry of Nature,* is the most famous fractal object every discovered. The Mandelbrot set consists of a set of points on a plain defined by:

Given a point (x,y), compute a sequence of other points (a_i,b_i) for I -0 ,1,2 .. according to the formulas:

$$a_0 = 0$$
$$b_0 = 0$$

$$a_{i+1} = (a_i)^2 - (b_i)^2 + x$$
$$b_{i+1} = 2a_ib_i + y$$

If each point in the sequence (a_ib_i) stays finite, then (x,y) is a member of the set. If the sequence of points (a_ib_i) shoots off into infinity, then (x,y) is not a member of the set. Figure 1.0 shows part of the Mandelbrot set represented using a 1920x1080 resolution image generated from the supplied sample implementation.

For more information on the Mandelbrot set please review:

http://en.wikipedia.org/wiki/Mandelbrot_set

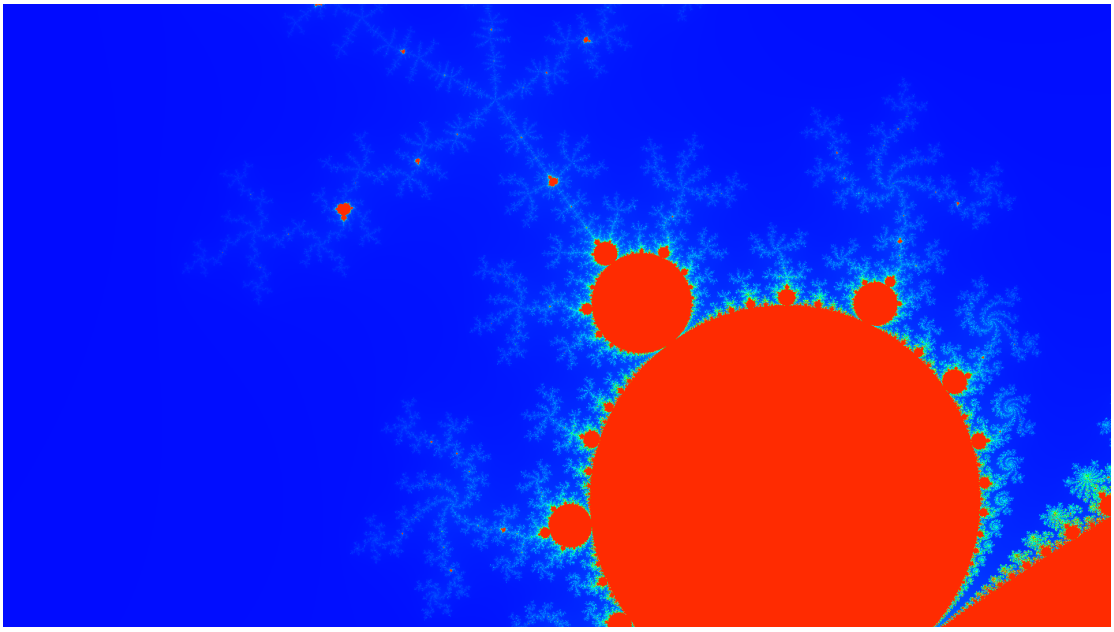

Figure 1.0 Part of the Mandelbrot set represented as an image.

Computing the Mandelbrot set

As it is not possible to compute an infinite sequence of points and still get an answer in a finite time, an alternate criterion is needed to determine membership in the Mandelbrot set. The Mandelbrot set is a compact set, contained in the closed disk radius of 2 around the origin. In fact (x, y) only belongs to the Mandelbrot set if and only if

$$\sqrt{(a_i^2 + b_i^2)} \leq 2$$

for all *i*.

That is, if :

$$\sqrt{(a_i^2 + b_i^2)} > 2$$

for some *i,* then the sequence will inevitably shoot off into infinity. If we select an arbitrary, but large enough limit on *i*, (say 1000), then we can determine if (x,y) is a member of the set: If *i* reaches 1000 and $\sqrt{(a_i^2 + b_i^2)} \leq 2$, then (x, y) is a member of the set. If $\sqrt{(a_i^2 + b_i^2)} > 2$ before *i* reaches 1000, then (x, y) is not a member of the set.

This approach is used in the following algorithm:

```
while(iter < MAX_ITER && zmagsqr <= 4.0){
      ++iter;
      a = (aold * aold) - (bold * bold) + x;
      b = 2.0 * aold*bold + y;

      zmagsqr = a*a + b*b;

      aold = a;
      bold = b;

 }
```

In this approach, the number of completed iterations (`iter`) before $\sqrt{(a_i^2 + b_i^2)} > 2$ is used to generate the colour for a specific pixel (where the pixel coordinates are used for the x, y values)

My Single Threaded Implementation

The algorithm in the previous section has been applied in my sample application (mandelbrot.c):

http://turing.une.edu.au/~comp309/markdown_lectures/assignments/project_01/

This program uses libbmp (i.e. bmpfile.h, bmpfile.c) to produce bitmap formatted images from the Mandelbrot set. The program contains a number of constants defined:

- RESOLUTION – The resolution of the fractal, effectively to the zoom level
- XCENTER – Position of the fractal in the image
- YCENTER - Position of the fractal in the image
- MAX_ITER – Maximum *i* value from the algorithm in the previous section
- WIDTH - Image size (i.e. image resolution)
- HEIGHT - Image size (i.e. image resolution)

Analyze the example, make changes to the values and review your results to understand how these are used in the algorithm. Compile and run my example using the supplied makefile

Your Task

To start with your task is to construct two programs:

- The first program will be either a Multi-process or Multithreaded program that produces a fractal image. You are free to use any approach (e.g. ring of processes, pthreads, shared memory etc.) to achieve this – the only constraint is that the implementation should run on a single machine.

- The second program will be a distributed implementation using either the PVM or MPI platforms. You are free to use either approach – the only constraint is that that the implementation should be distributed application.

Both implementations should use the libbmp to produce a bitmap image of your fractal. You are free to pull apart my example application and use any parts and make any improvements you require.

Your final task is to analyze and compare the performance of your two approaches. To do this you will need to add additional code to your implementation to record the runtime and should construct some simple tests, such as varying the image and fractal resolutions (or any other factors you believe will affect the runtime) and tabulate this data. These results should be included (and discussed) in a **report** of approximately 1000 words that outlines how your implementations work and the advantages/disadvantages of both. This document will be submitted as a PDF document.

Marking Guide

This project is very open-ended, as such there won't be a prescriptive marking scheme. However marks will be distributed as follows:

- Single Machine implementation – **40%**
- Distributed implementation – **40%**
- Analysis Report - **20%**

Things to consider:

- Consistent use of style and comments
- Include a makefile to build your project
- Compilation Warnings (I will be looking for –Wall)
- Error checking throughout
- Make sure there aren't any deadlocks, race conditions, segmentation faults, buffer overrun etc.
- Usage of hard-wired constants
- You should use consistent algorithm configurations between your implementations to make your comparison valid.
- You should make use of graphs and tables to present your comparisons.

Hints and Tips

- Remember that the cluster you will be working is a virtualized system sitting on the turing.une.edu.au hardware, so it is not a truly distributed system. This will impact your performance comparison.
- Review the HSV colour model to see how the colour gradient function works: http://en.wikipedia.org/wiki/HSL_and_HSV