**TODO: avoca times**

**Introduction**

This report outlines a series of optimisations performed on the provided file mandelbrot.cpp, covering both optimisations to the logic of the program and parallelisation. Some individual optimisations are outlined below, followed by the combination of multiple optimisations to achieve the final product.

**-O2**

The easiest optimisation possible, the –O2 flag provided to the gcc compiler averaged a speed up of 2x.

**Bounds checking**

Analysis of the Mandelbrot set will show that a majority of points in the Mandelbrot set also fall within two shapes, the cardioid and the second bulb. These shapes can be seen in figure 1.

The equations provided in figures 2 and 3 can detect whether a point lies within these shapes in constant time, and so drastically reduce the average run time for the inset() function.

**Cycle checking**

Sometimes the operation fc(z) = z^2 + c will result in a cycle, a trivial example of such is c = 0 + 0i, where regardless of the number of iterations z will be equals to 0 + 0i.

If such a cycle exists then there is a guarantee that the point is in the Mandelbrot set. This can reduce the average time of the inset() function in cases where the point being checked is inside the Mandelbrot set.

While this optimisation does provide a speedup compared to the base file, any performance boosts are superseded by the Bounds checking optimisation provided above.

**Symmetry**

The Mandelbrot set is symmetric across the x-axis, and so optimisations can be made such that as little as half the number of points need be tested. For example the Mandelbrot set for the i range {-1, 1} is equivalent to two times the Mandelbrot set for the range {0, 1}.

Issue arise, however, from the lack of precision due to using doubles in the source program, and off-by-one errors occur in the symmetries. Because of this, the optimisation is not reliable enough be used.

**OpenMP**

Using a parallel for reduction in the code resulted in a ### speed up of the code when using four processors, and a ### speedup when using 16. An ideal PRAM model would expect values closer to 4x and 16x respectively, however actual implementation of the parallelisation has to account for overhead and asymmetries in the input data;

**Static vs Dynamic**

choosing between a static and dynamic schedule is a trade-off between the additional overhead of the dynamic thread allocation and the asymmetry of work allocated to each thread in the static schedule.

There is no easy way to accurately distribute work across each thread, and randomly assigning points to each thread in a static schedule results in a large enough overhead to make it not worth the effort.

Instead, trial and error was used, and a static schedule was found to have the best performance when the outer loop in mandelbrotSetcount() was parallelised with schedule(static,1).

**Inner vs Outer loop**

Choosing where to place the parallelisation in this code was another trade-off: place it too high (say at the for loop in main( )) and the risk of asymmetric work loads leading to idling threads is high, but place it too low (say the inner loop in mandelbrotSetCount( )) and each thread will be continually waiting to access a common resource, in essence sequentialising the program again.

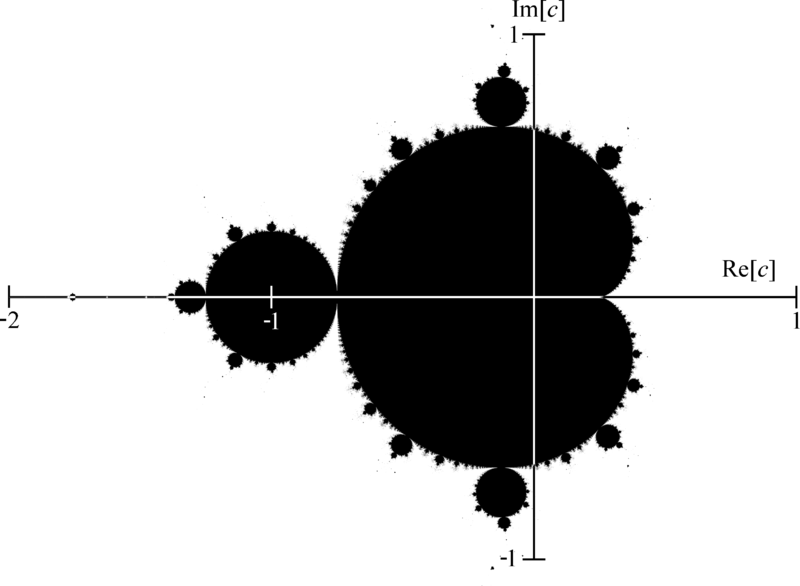
A happy medium between these two cases was parallelising the outer loop in mandelbrotSetCount( ).

**Final program**

The final program combined the

Cardioid

Second Bulb



**Figure 1: Cardioid and Second Bulb identification.**

**Figure 2: Equations for Cardioid test**

**Figure 3: Equation for second bulb test**

**Figure 4: Times and speedup test**