Aaron St.John
CS 473
Final Project

Final Project Write-up

The Mandelbrot set is a mathematically defined set of points on the complex number plane. The set defines a fractal, and image that is infinitely complex and self-referential. To determine whether a point lies within the Mandelbrot set, several iterations of the mathematical formula $z_{n+1} = z_n^2 + c$ are calculated, where c is the point being considered and $z_0 = c$. If the formula diverges, it is considered outside of the set. However, if the formula converges to zero, the point is considered to be inside of the set.

Approximating the Mandelbrot set on a computer is a fairly easy task. It is also considered an embarrassingly parallel problem because the problem can divided among separate processes in such a way that the processes do not need to communicate with each other.

For my final project, I designed a program that generated the Mandelbrot set. This program uses Windows multi-threading to do calculations in parallel. My program also utilized an advanced workpool strategy to divide the work among threads.

Windows multithreading is an ideal environment for this type of parallel processing. Windows schedules processor time on a thread-by-thread basis. That means that if a single process creates four different threads on a computer with four processors, each thread will be run on it's own processor. All threads owned by the same processor also have access to the same memory, which allows for very quick communication. In addition to using shared memory for communication, Windows provides a message-passing interface. Programs can also use Windows native GUI to provide helpful visualization to the user.

As mentioned before, my program used a workpool strategy, which is ideal on this type of coarse-grained system. However, the strategy used is more advanced than is usually encountered. The program divides a section of the complex plane into a number of squares, and a pre-defined number of threads are created. Each thread is then designed a square to process. After a pre-defined number of iterations are calculated, the square is returned to the master thread.

When a square is returned to the master thread, it placed in into one of two queues to await further processing. If the square is wholly inside the Mandelbrot set, it is placed in one queue. If the square is partly inside the set and partly outside the set, it is placed in a different queue. If a square is wholly outside the Mandelbrot set, it does not need to be added to a queue, because no further processing is needed.

After all of the squares have been processed once, squares are then re-assigned based on probability. The master thread will choose a queue using pre-defined odds and assign the first item into that queue to any free threads. This process continues until the user interrupts the program.

By using this system, my program is able to generate a more accurate approximation of the Mandelbrot set quicker than more traditional methods. After a rough approximation is produced, my program will concentrate of the border areas more heavily, allowing it to quickly fill in details.

After implementing my program, I ran several tests to determine optimal values for the size of the work chunks, number of iterations, and border preference. To conduct these tests, I generate approximations for three different views of the Mandelbrot set. I

ran my program until a certain percentage of points had diverged, and timed how long it took to reach this point.

In every case tested, the calculation ran faster if a border square was assigned 75% of the time versus 25% of the time, which suggests that a strong border preference is optimal, which I expected. Most of the points that will be eliminated will come off of the borders, rather than the interior of the Mandelbrot set.

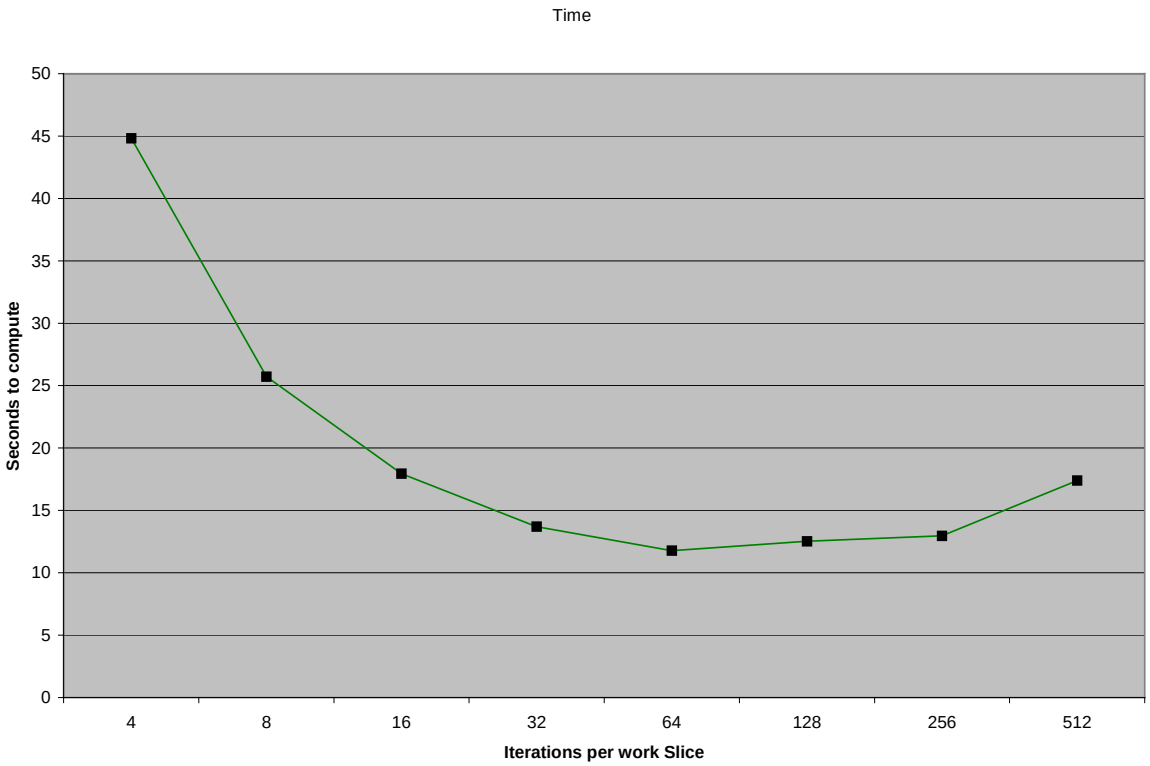
In half of the cases, the calculation ran faster if smaller squares were used, and in the other half, larger squares led. This suggests that the optimal square size is dependent on several factors, rather than being constant. It is my hypothesis that the shape of the border determines what square size is optimal, but more research would be needed to determine this.

When testing for an optimal number of iterations to run per work assignment, I found that the calculations were completed quickest if 64 iterations were done per work assignment. If too few iterations are done, calculation is slowed down due message passing overhead rises. However, if too many iterations are done, a single thread is 'tied up' for too long and the advantage of the advanced workpool strategy is defeated.

Several things could be done to improve my program. It would be helpful to find a formula that determined optimal values for square size for a given area of the Mandelbrot set, rather than having to guess which values will work well by hand. The program could also use a different workpool assignment strategy. Perhaps squares could be prioritized based on their proximity to a border, rather than simply being divided into border and non-border categories. The workpool assignment system could also be made to adapt to changing conditions while the program is being run.

Data

View	Border Odds	Size	Time (seconds)
1	25	Large	82.178
1	25	Small	40.898
1	75	Large	21.261
1	75	Small	13.67
2	25	Large	46.607
2	25	Small	98.351
2	75	Large	40.068
2	75	Small	36.653
3	25	Large	395.439
3	25	Small	249.209
3	75	Large	100.574
3	75	Small	76.49



Program Output Examples