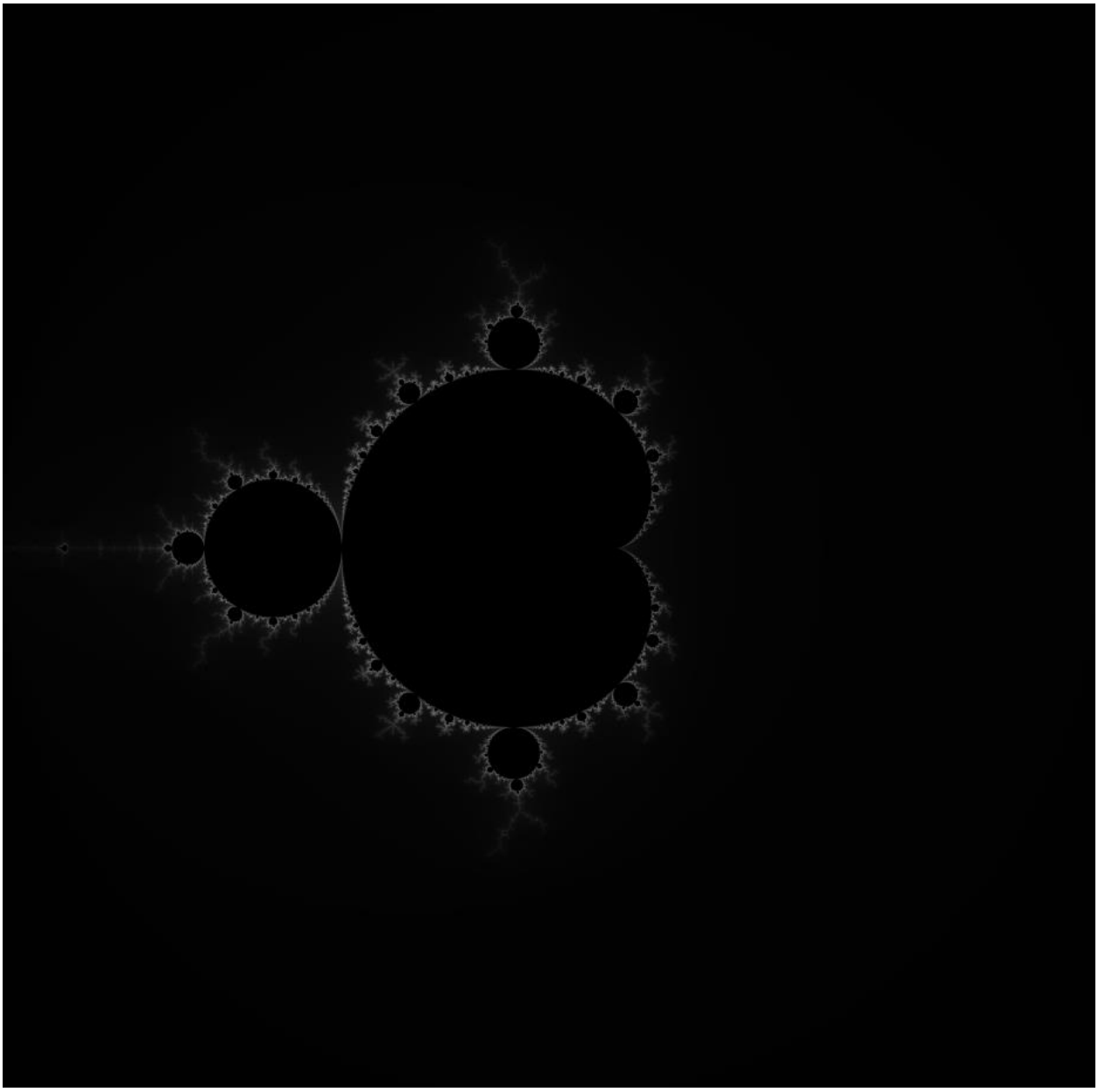
Tyler DeFoor

PA2 – Mandelbrot

3/16/2017

Dr. Harris

Dynamically generated 15,000 x 15,000 Mandelbrot



Sequential

Sequential Mandelbrot is a program that generates an image the represents the Mandelbrot set. It generates images of configurable resolutions. It was accomplished in a completely sequential manner, without any parallelization.

The main function generates a 2d array of unsigned chars filled with values ranging from 0-255, calculated by the function “calculate.” The calculate function accepts a custom struct named Complex, which is comprised of two floats representing the coefficients for the real and complex portions of the complex number. From this struct calculate determines whether or not a function is in the Mandelbrot set by iterating up to 256 times, checking to see if the value derived from the function is greater than four. If it is, the calculation stops and the count is returned. If it reaches 256 calculations, it returns the count and finishes.

The main function has two for loops, one that loops from 0 to the height – 1 and another that loops from 0 to the width – 1. After the loops are completed, the pixmap is completed and sent to a predefined function that outputs a PIM file to view.

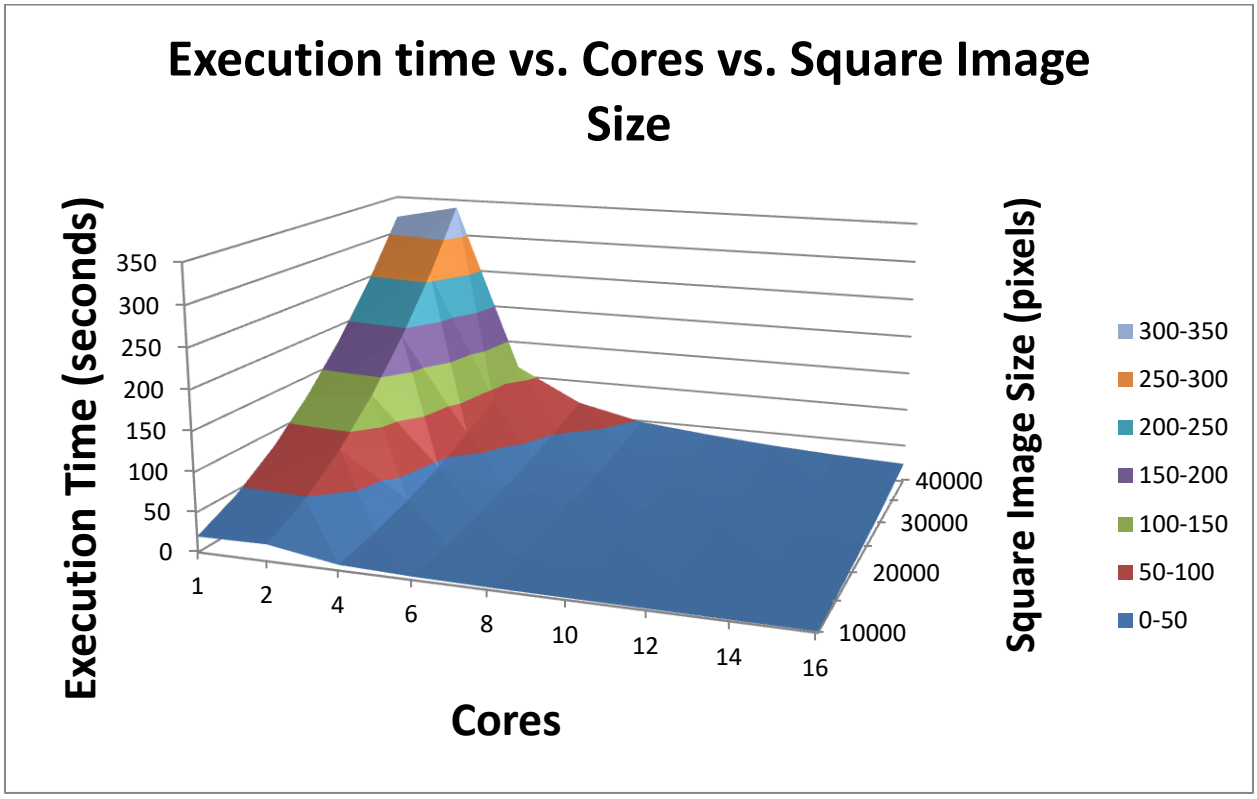
Dynamic

Dynamic Mandelbrot is a dynamically allocated parallel solution to generating an image that represents the Mandelbrot set. It generates images of configurable resolutions. The main function is executed in both the master and the slaves.

The master generates a 2d array of unsigned chars filled with values ranging from 0-255, calculated by the slaves. After initializing an array of unsigned chars for a specific row it sends one row number to each slave and waits for them to respond. It then waits to receive a row from each slave, listening to receive from any source with any tag. When it receives a row, it checks the status variable of the receive to determine the sender and the tag. The tag the slave sends is the row number it calculated. The master then loops from 0 to width – 1, copying that row to the pixmap at the row the slave tagged the send with. When the row has been put in the pixmap, the master sends that slave another row number to calculate and continues to receive. After all rows have been received, the pixmap is completed. It then sends -1 to all the slaves, signaling that they should stop running. Finally, the pixmap is sent to a predefined function that outputs a PIM file to view.

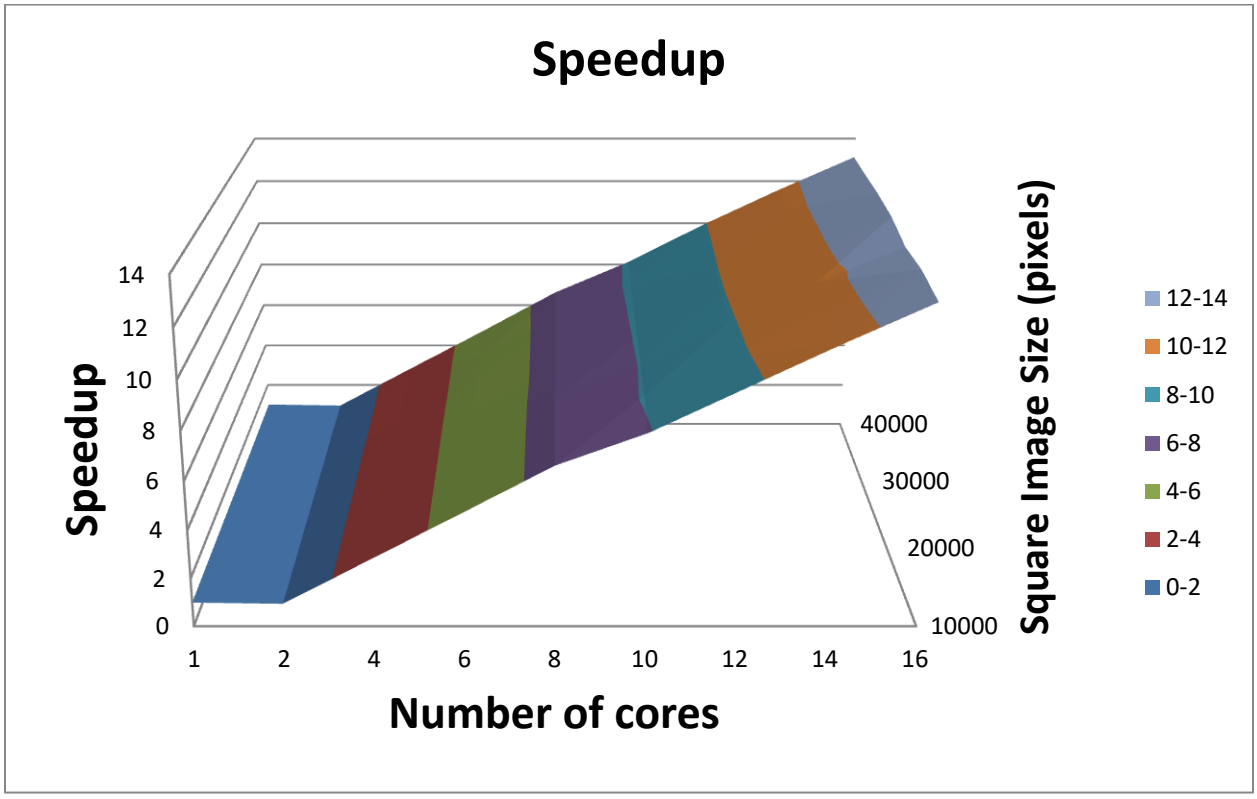
The slaves wait to receive a row number, looping until they receive a terminator of value -1. Upon receiving a row number, it calculates all the pixels that are in that row and places them in an unsigned char array. It then sends the array to the master process with the tag set as the row number it calculates. This loops until completion.

Dynamic vs. Sequential Analysis: Time



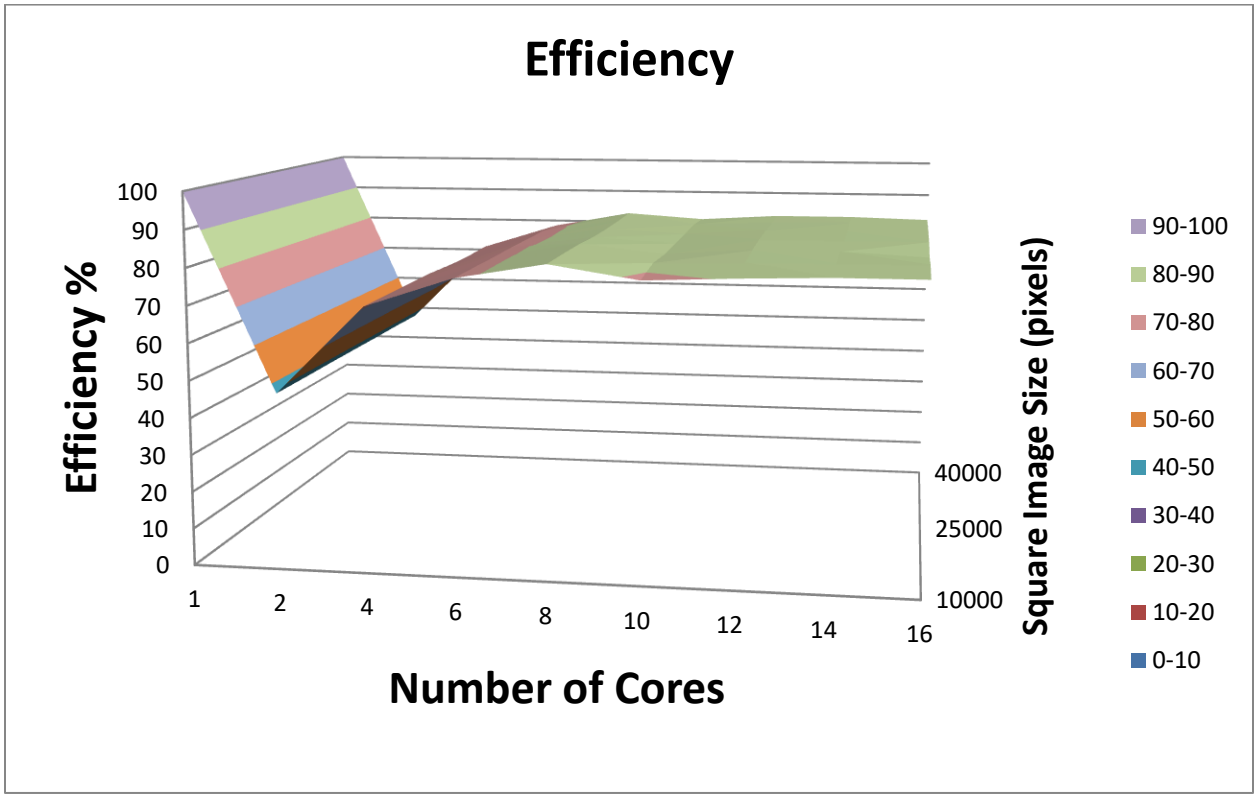
The graph above plots execution time against image size and number of cores used. Overall, the Dynamic was significantly faster. At 2 cores, the Dynamic actually slowed down compared to the Sequential version. This is due to the fact that with two cores, it is only a master and a slave. This translates into essentially doing Sequential with the added time of message passing. After that initial decrease in speed, the Dynamic blows Sequential out of the water. Sequential takes 321.347 seconds to calculate a 40,000 by 40,000 image while Dynamic with 16 cores takes 24.4683 seconds. Dynamic is faster in every case other than when it is executed on only two processors.

Dynamic vs. Sequential Analysis: Speedup



The graph above plots speedup against image size and number of cores used. The maximum speedup achieved by Dynamic in comparison to Sequential is 13.22165238 at 25,000 by 25,000 at 16 cores. The speedup was consistent past 2 cores and overall sublinear. This is due to inefficient message passing and network latency. The rows are passed back and received one at a time, which could easily increase and give better speedup at higher cores.

Dynamic vs. Sequential Analysis: Efficiency



The graph above plots efficiency against image size and number of cores used. Efficiency takes a deep downward spike at two cores and then climbs up fairly steadily from there. It decreases slightly at 10 cores, staying relatively stable at between 80% and 83% from then on. Overall the efficiency is relatively good discounting the inefficiency of two cores. It is highest at 82.93114543% on a 15,000 by 15,000 image and 8 cores being used.

Future Work and Improvements

The Dynamic implementation of Mandelbrot can easily be improved. Some testing would have to be done to see exactly what would be optimal for the H1 cluster in terms of slave workload vs. message passing. This would be done by increasing the amount of work per slave while still increasing efficiency. Likewise, a Static implementation of Mandelbrot can be created with a basis in Dynamic.