Project Assignment 02 Report

CS415

3-13-2017

By: Jervyn Suguitan

Description of Assignment

Project 02's main purpose was to teach students about the mandelbrot set and how to parallelize the code that creates the mandelbrot set. After parallelizing the code, students also learn about the speedup that occurs from using a parallelized program compared to a sequential program. Next students will also compare the times from different types of images and number of processors used. The project was split into 3 parts a) calculate Mandelbrot using a sequential algorithm , b) calculate Mandelbrot using parallel algorithm statically and c) calculate Mandelbrot using parallel algorithm dynamically. The timing function that was used to time calculation for this project was MPI_Wtime. Also for the project, I only did parts A and C as I did not need to do static parallel code. So to reiterate I only did sequential code and dynamic code and will only report my findings on these codes.

Part 1: Sequential

The first part of the project tasks students in figuring out how to code Mandelbrot sequentially. Using code that was given to us plus code from the textbook this task was relatively easy. In its most basic form, the code has 2 hard coded values width and height which detail how big the mandelbrot set picture will be. Once this is done a double for loop is used that iterates through each pixel of every row of the pictures. There is code that calculates the color of each pixel that was given in the book. After calculating all the pixel colors, input the 2D array of pixels into a function that makes it into a picture. An example of the Mandelbrot set is in Figure 1 below. One of the problems with the pictures below is that the shades of gray are not clear so it appears that the Mandelbrot set is not correct.

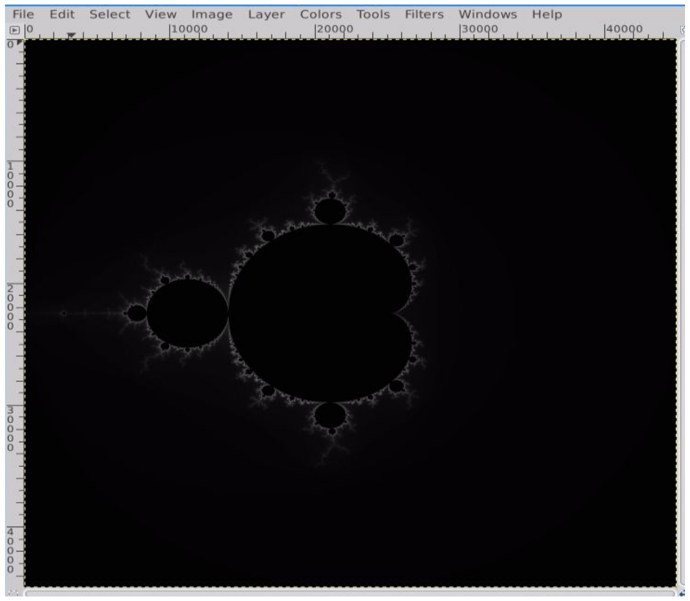


Fig. 1: Picture of the Mandelbrot set. The resolution of this picture is 45000x45000. The shades of gray are not seen well due to poor quality of the program that renders the picture.

Figure 2 showcases the times for 3 different resolutions of the Mandelbrot Set. These resolutions (640x640, 16000x16000, and 45000x45000) were chosen to showcase the different speed up from sequential to parallel which will be shown later in the report. Fig 2. has three different time scatterplot graphs. The times for each of the resolutions do not vary large amounts, one reason this occurs is that there is no message passing between master and slave which takes up time. There is no message passing since it is sequential there is no need to use multiple processors thus the program uses a “ -n1 “ to show that we will only use one core. Also since everything is calculated a pixel at a time the times will not vary too much since there can not be any speed up and the only variation is the speed of the processor that works on this data.

Problems that I have encountered during this part of the project is that I didn’t know that I still had to use MPI_Init even though there was no sending to slave functions at all. I needed it for the MPI_Wtime functions.

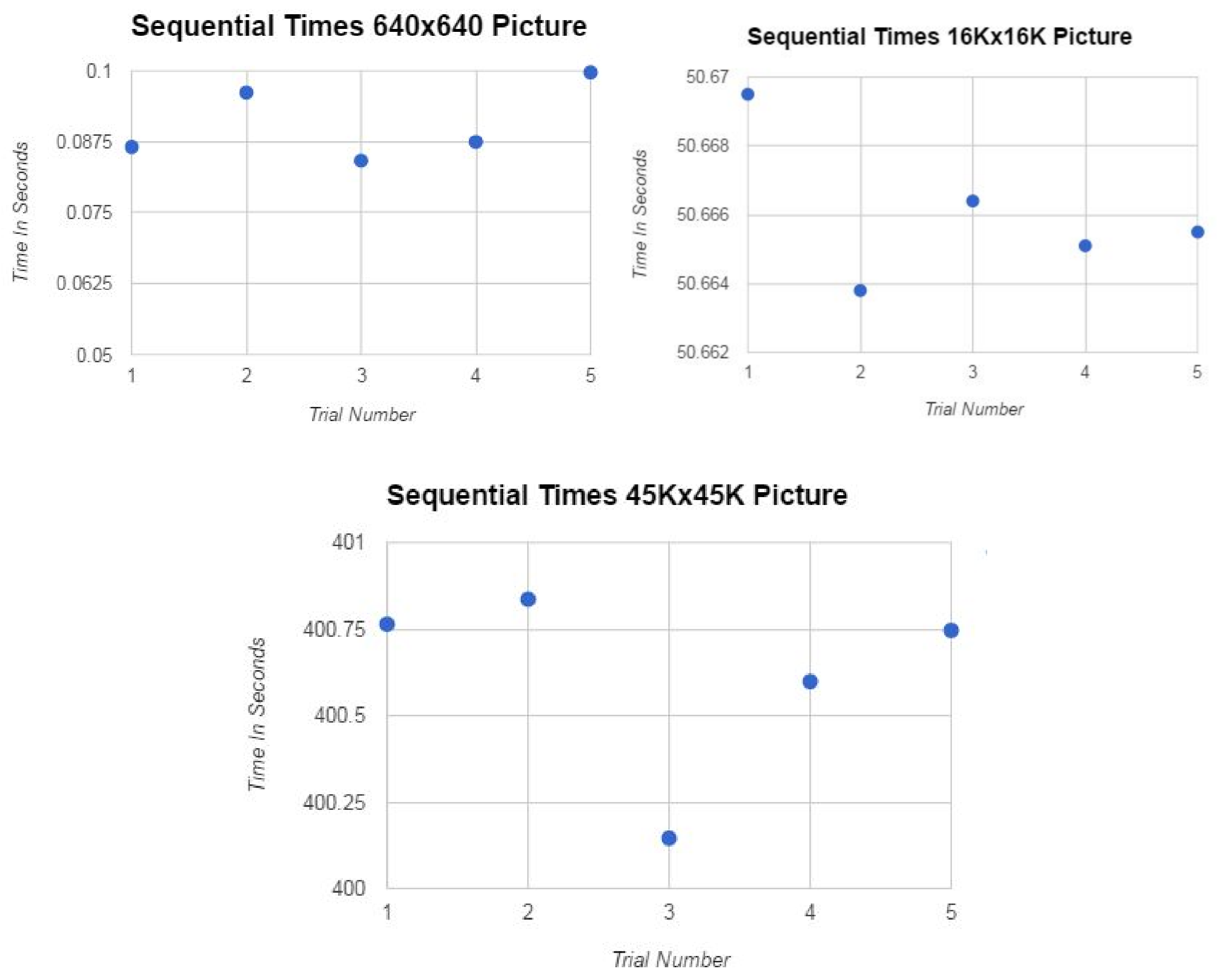


Fig. 2: Graphs that showcase the sequential times for different resolution of the Mandelbrot Set. The Y axis shows the time in seconds and the x-axis shows the trial number for that specific resolution. The times do not vary large amounts since everything is sequential thus there can not be a large speed up. Speed up or slow down may occur due to the processor speed which could be different for every trial.

Part 2: Dynamic

The next part of the projects tasks students to parallelize their sequential code. There is two types of ways to parallelize the code, statically and dynamically. In this portion of the report I will only talk about the dynamic code. The difference between static parallel code and dynamic parallel code for the mandelbrot set is that in static parallel each slave is given a set number of rows and once they are done calculating each row they stay idle. In dynamic once a slave is done with all of its rows, it will check if other slaves currently have extra rows then they will work on them. This creates load balancing between all processors. The goal of load balancing is to make all the slaves constantly working thus speeding creating a speed up.

The idea for the dynamic code is somewhat similar to the sequential code. Essentially the dynamic code breaks up the rows into a specified amount and sends part of the Mandelbrot Set to a slave. Each slave then works on part of the set then returns the rows that were worked on to the master. Once it is done with its current rows, it will keep checking to work on more rows until it receives a Final tag which kills all slave operation.

Figure 3 below shows the times of dynamic and sequential code running 45000x45000 resolution. The bar graph below clearly states the dynamic is much quicker than sequential. This is due to the amount of processors splitting up the work, rather than a single processor work a pixel at a time. Fig. 3 also shows that the times for dynamic 48 core is slower than dynamic 24 core. Although one may think that the 48 core would speed up the process even further since there's more cores, this is false due to many reasons. One reason slow down occur is that there is more message passing between master and slave which can cause slowdown. Another reason is that during testing of the dynamic 48 core, there was many people other people testing thus network was very congested which could slow down calculations. There is diminishing returns as more processors are used for the Mandelbrot set.

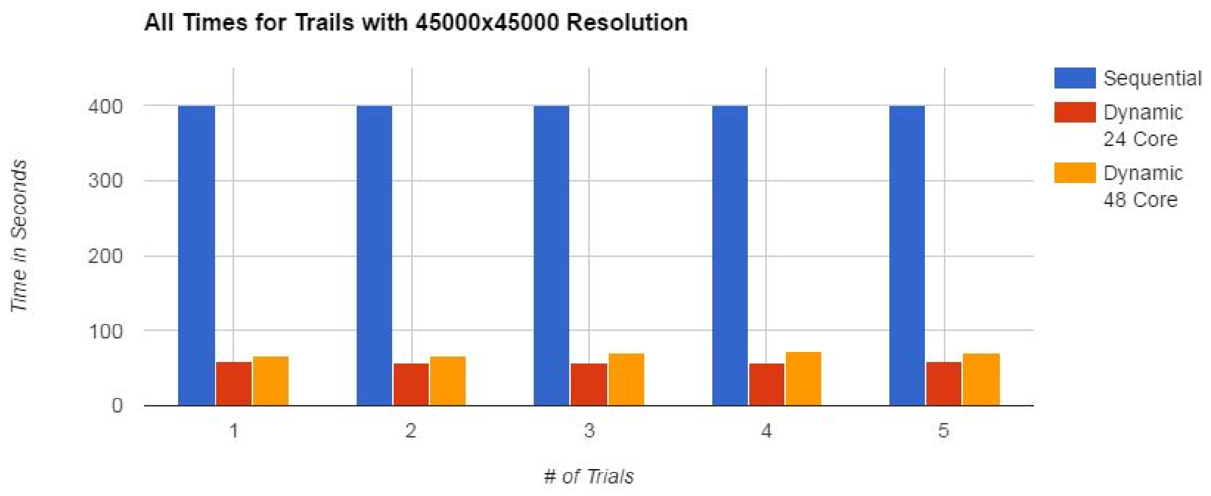
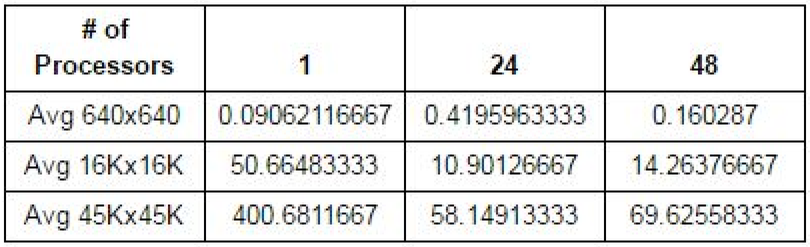


Fig. 3: Bar graph that shows the different times between sequential, dynamic 24 core, and dynamic 48 core. From this graph we see that the dynamic is much faster than the sequential run time. However somewhat surprisingly, the dynamic 48 core is slower than the dynamic 24 core. This may be due to the diminishing returns for using more cores. Slow down could occur due to more message passing being done.

Table 1: Table that shows the average run times of resolutions and the number of processors on that resolution in **Seconds**

Average Runtime Based on Number of Processors and Resolution



# of Processors	1	24	48
Avg 640x640	0.09062116667	0.4195963333	0.160287
Avg 16Kx16K	50.66483333	10.90126667	14.26376667
Avg 45Kx45K	400.6811667	58.14913333	69.62558333

Figure 4 is a compilation of all the run times with x-axis showing the Number of Cores used, y-axis is the resolutions and the z-axis the times. From this we can again clearly see that using 1 core (sequential) is the slowest. With 24 core dynamic being around 1.25 times faster than the 48 core dynamic. To reiterate again this could be due to the 48 core needing to use message passing much more than the 24 core at high resolutions. The speed up can be calculated using times seen in Table 1 which showcase the average times of processors on different resolutions. The speed up which I used S(p) = Time of Sequential / Time of Parallel can be seen on Table 2. The speed up is less for 48 cores compared to 24 cores due to the overhead that can slow down 48 cores such as more message passing

Based on the average times given in Table 1 we can say that the parallel for 24 cores and 48 cores is not cost optimal for these chosen resolutions since the parallel time multiplied with the number of processors is always greater than the sequential time. To be cost optimal parallel time complexity multiplied number of processors should be equal to the sequential time complexity.

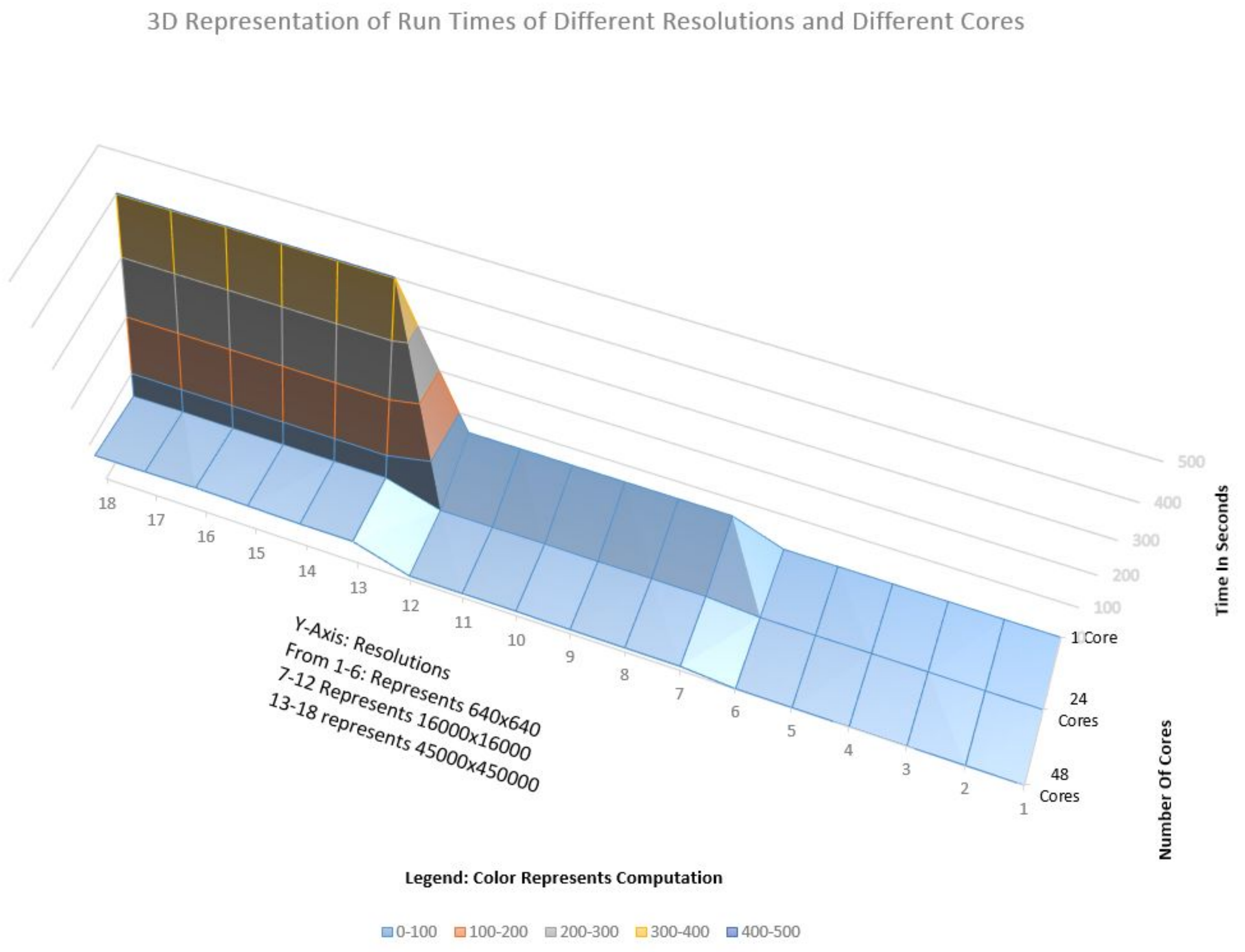


Figure 4. 3D graph that represent the Run Times based on Resolution and number of Cores used. As more Cores are used there is more overhead which explains the slow down for 48 cores. 1 Core however is slower in all resolutions being around 4 times slower which is seen in Table 2.

Table 2: Table that shows the speed up of 1 core versus 48 cores and 1 core versus 24 cores

Speed Up depending on Number of Cores and Resolution

	1 core Time / 24 Core Time	1 core Time/ 48 Core Time
16K x 16 K	4.647609758	3.551995382
45K x 45K	6.890578478	5.754797985

Conclusion

This project teaches students about the mandelbrot set and how to parallelize the code that creates the mandelbrot set. The project, once completed, shows how the amount of data worked on and the amount of processors used influence the computation time. To be more precise for large amounts of data using dynamic code, more cores may receive the result of diminishing returns due to the extra message passing. This project also teaches why certain timings could occur which could be from network congestion and many students using the h1 node at the same time, etc.