Adam Montano
CS 415
PA2 - Mandelbrots

The three parts of this project concerned generating mandelbrot images. I implemented a sequential version using a single processor and two parallel versions. The parallel dynamic version creates a pool of work to complete that the master oversees and commands the slaves to perform. The parallel static version predetermines the work to do for the slaves and involves no load balancing.

Sequential Implementation

The sequential version generated a Mandelbrot image with a single processor. A nested for loop calculates each row and column by pixel with the cal_pixel function and saves it into a file. There is no MPI message passing and is very inefficient in comparison to the parallel versions.
Output from /bin/sequential.sh:

```
Image processing took:
357.971
```

Parallel Dynamic Task Assignment

The dynamic parallel version generated a Mandelbrot image with any number of processors. The master node oversees a work pool and assigns a different row to each slave processor. If a slave comes back with some results, the master node gives it a new row to compute as long as there are still rows that need work to be done. After each row is complete, the image is saved. There exists a threshold where the time will become greater even though the amount of processors has increased.
Output from /bin/dynamic.sh:

```
10.76
```

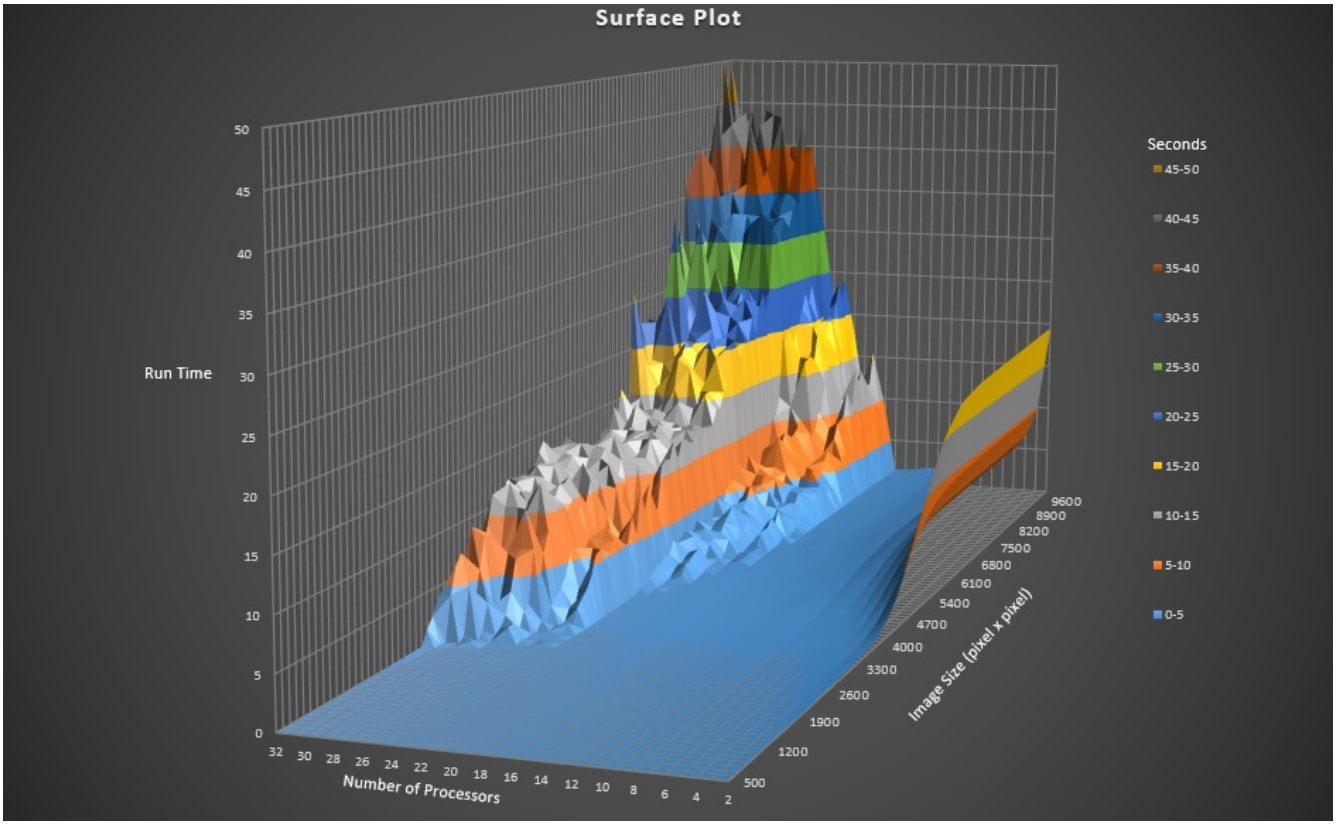
Parallel Static Task Assignment

The static parallel version generated a Mandelbrot image with any number of processors, similar to dynamic. The master node predetermines the rows for each slave process. Each slave generates their chunk of the image and sends it back to the master. The master saves it into an image.
Output from /bin/static.log

```
20.34
```

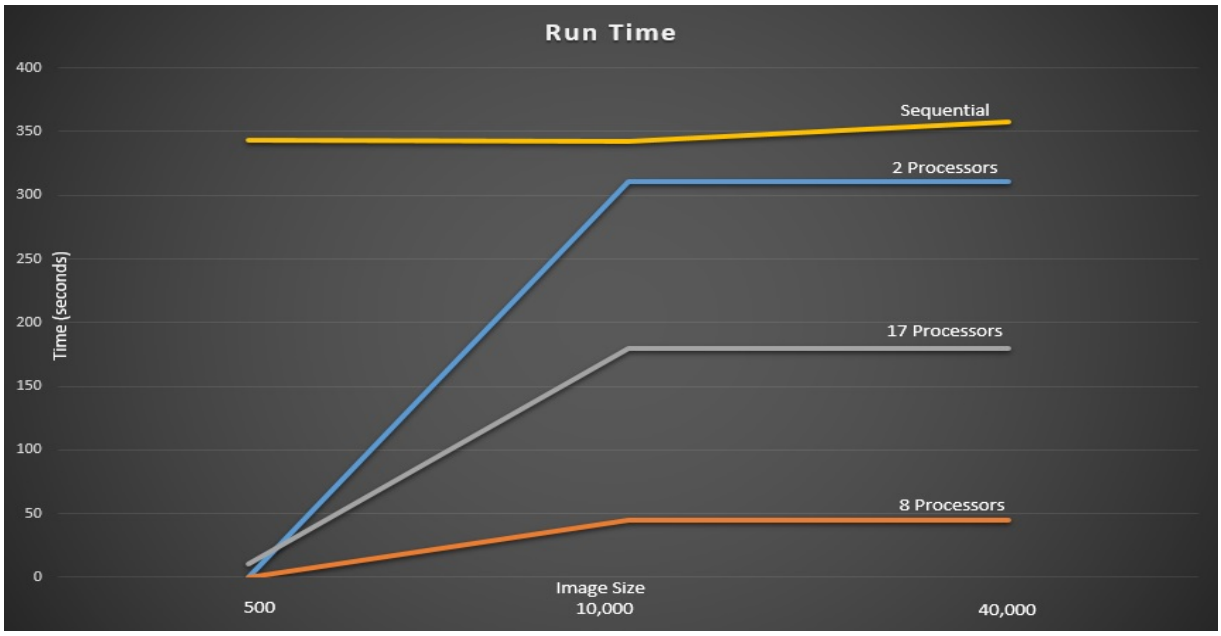
Graphs

This surface graph (Graph 1) plots out the run time of the processes, the image size and the number of processors. The range of image sizes is from 500 x 500 pixels to 10000 x 10000 pixels. The maximum amount of processors are 32. This surface plot was created with data recorded from the dynamic parallel implementation. There is a “goldilocks” spot that exists with the number of processors. At a certain threshold, the increasing amount of processors does not decrease the run time and too few processors creates a high run time. There is a lot of noise in the back corner most likely due to network congestion as the queue became chaotic.



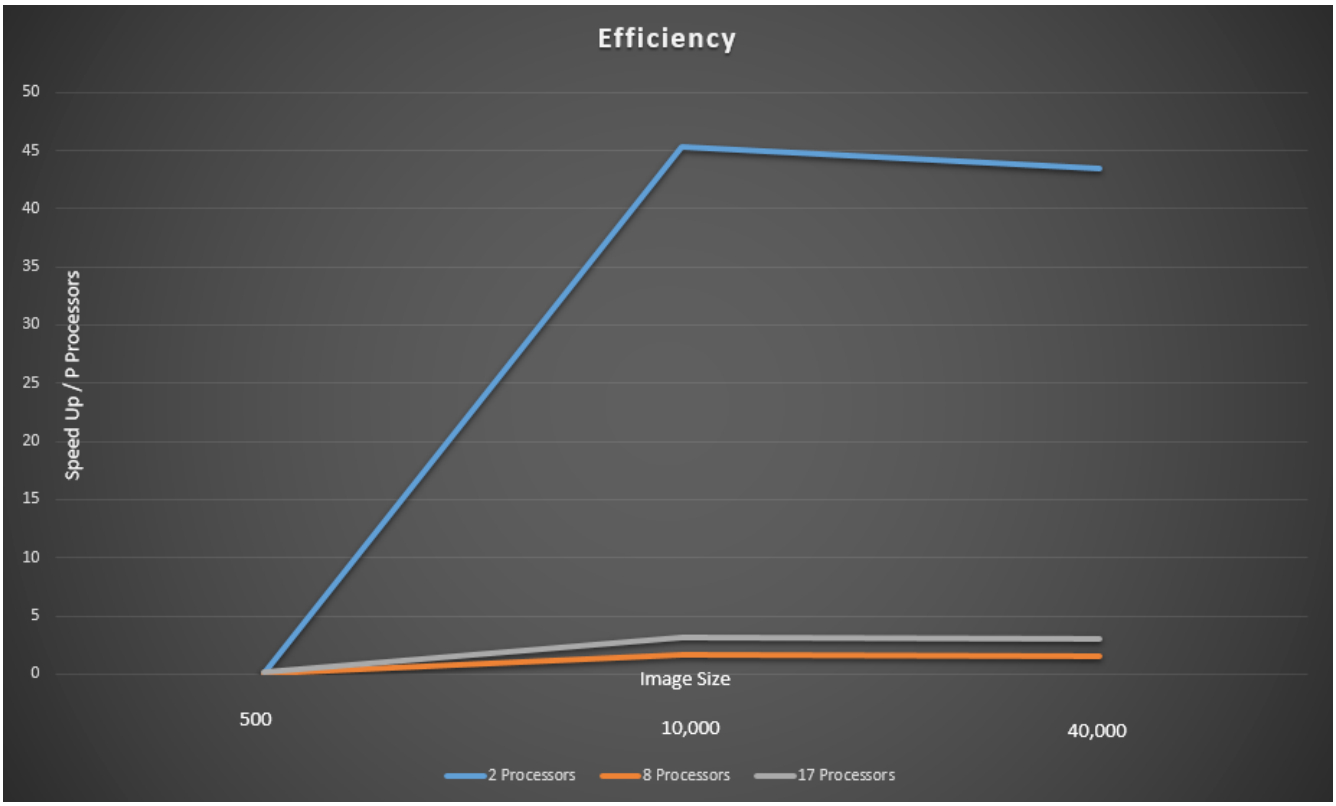
Graph 1.

A simple run time graph confirms the threshold theory. The run time graph (Graph 2) plots the run time with the dynamic parallel implementation with varying amount of cores and the sequential implementation. Although the sequential run time stays constant, the dynamic parallel implementation is much lower overall but a higher number of processors does not necessarily mean that the run time will be lower.



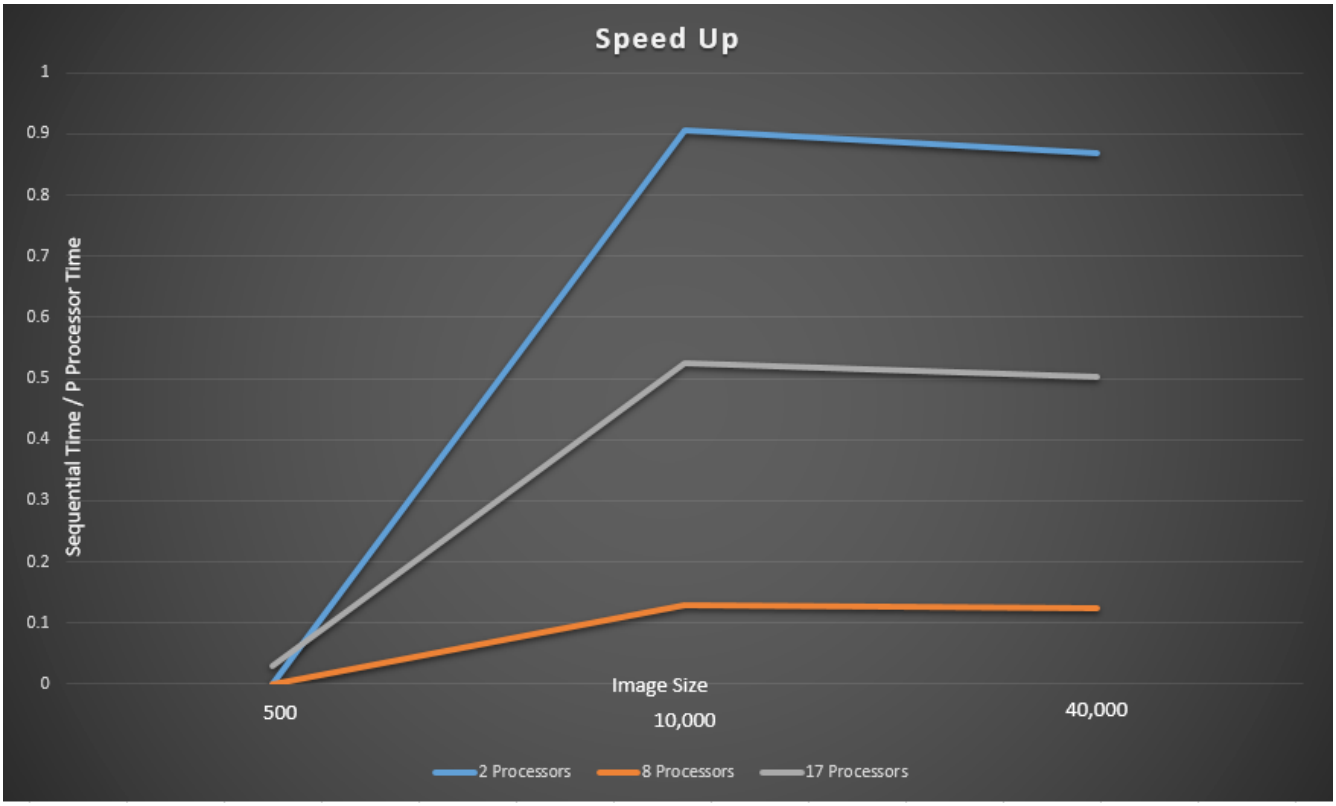
Graph 2

The efficiency graph (Graph 3) plots the speed up divided by the number of processors for each image size by the dynamic parallel implementation. This graph is somewhat harder to understand. It seems that the efficiency of the 2 processor version seems the least efficient, the 17 processor more efficient and the 8 processor is the most efficient. This seems to confirm the threshold theory earlier explained.



Graph 3

The speed up graph (Graph 4) plots the sequential time divided by the parallel time over the image size. The graph shows how the speed up for each implementation compares to the sequential. The graph exactly matches what one would expect for the speed up for each version. The 8 core version has the lower speed up, the 17 processor one has a worse speed up and the 2 core version has the worst speed up. This must mean that the speed up for each is much better than just doing sequential



Graph 4

Issues

The main issue with this project was determining how to use each result from the slave processors by the master. I could not understand how to but the row that was returned into the correct spot in the image. I ended up using an array that kept track of the rows sent to each slave process. The other large issue was dealing with the high amount of network congestion. This project takes a considerable amount of time to execute and everyone on the queue made this project very difficult, especially when debugging.