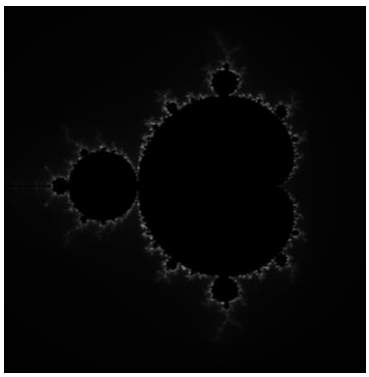
Adam Montano

Mandelbrot Report

CS 415



Introduction

The three parts of this project concerned generating Mandelbrot images. The implementations include both a sequential version of the software and an MPI based parallel version. There are two parallel versions, dynamic and static. The parallel dynamic version creates a pool of work to complete that the master process oversees and commands the slave processes to perform specific parts of the image. The parallel static version predetermines the work for each of the slaves and involves no load balancing. The master process only sends out the parts of the image and collects them after each slave is finished. Image on the cover is an example of the image being generated.

Sequential Implementation

The sequential version generated a Mandelbrot image with a single processor. A nested for loop calculates each row and column pixel 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 the parallel implementations.

Output from /bin/sequential.log

```
357.971
```

Data was collected by finding the run time from generating a Mandelbrot image of various sizes.

Image Size	**Time (seconds)**
Small (500x500)	0.057437
Medium (1000x1000)	20.1159
Large (40000x40000)	343.836

The largest imaged generated in under 5 minutes was about 40,000 pixels in width and height.

Below is a graph (Fig. 1) representing the above data set. Evidently, the larger the image correlates directly with the run time of the image generation. However, the run time becomes greater at a faster rate than the image size.

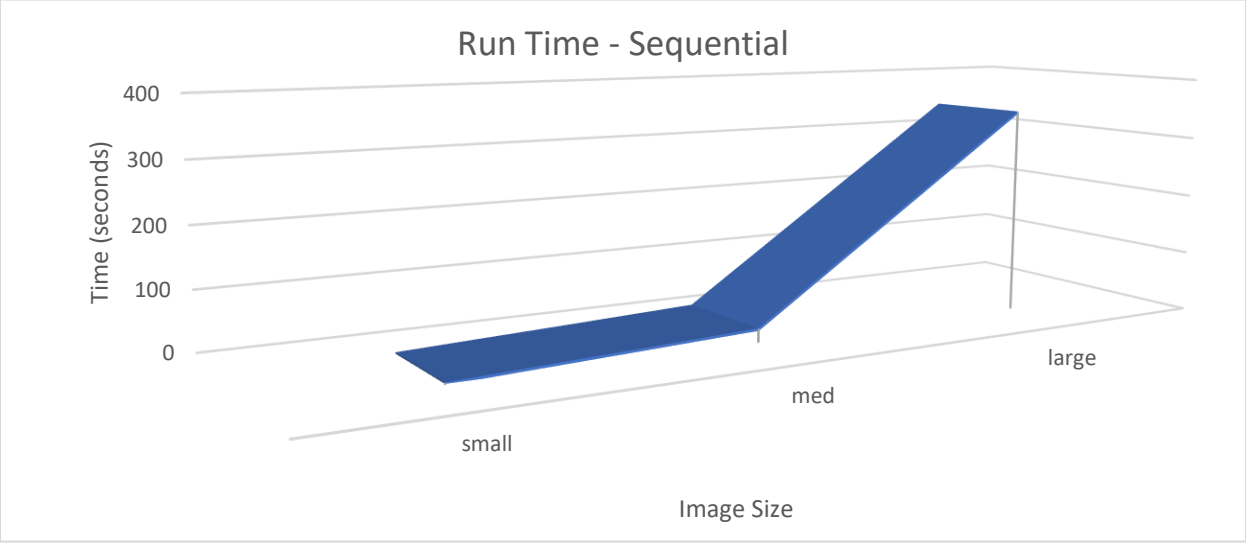


Figure 1 – Sequential Run Time Graph

Parallel Static Task Assignment

The static parallel version generated a Mandelbrot image with any number of processors, like dynamic. However, there is no work pool and 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
```

Data was collected by finding the run time from generating a Mandelbrot image of various sizes and various amount of processor cores.

Image Size	**Time**			
	2 Processors	*8 Processors*	*16 Processors*	*32 Processors*
Small (500x500)	0.105709	0.024143	0.014545	0.00059
Medium (1000x1000)	9.6682	9.64648	4.84401	2.19742
Large (40000x40000)	154.464	154.333	77.4208	38.5922

Below is a graph (Fig. 2) representing the above data set.

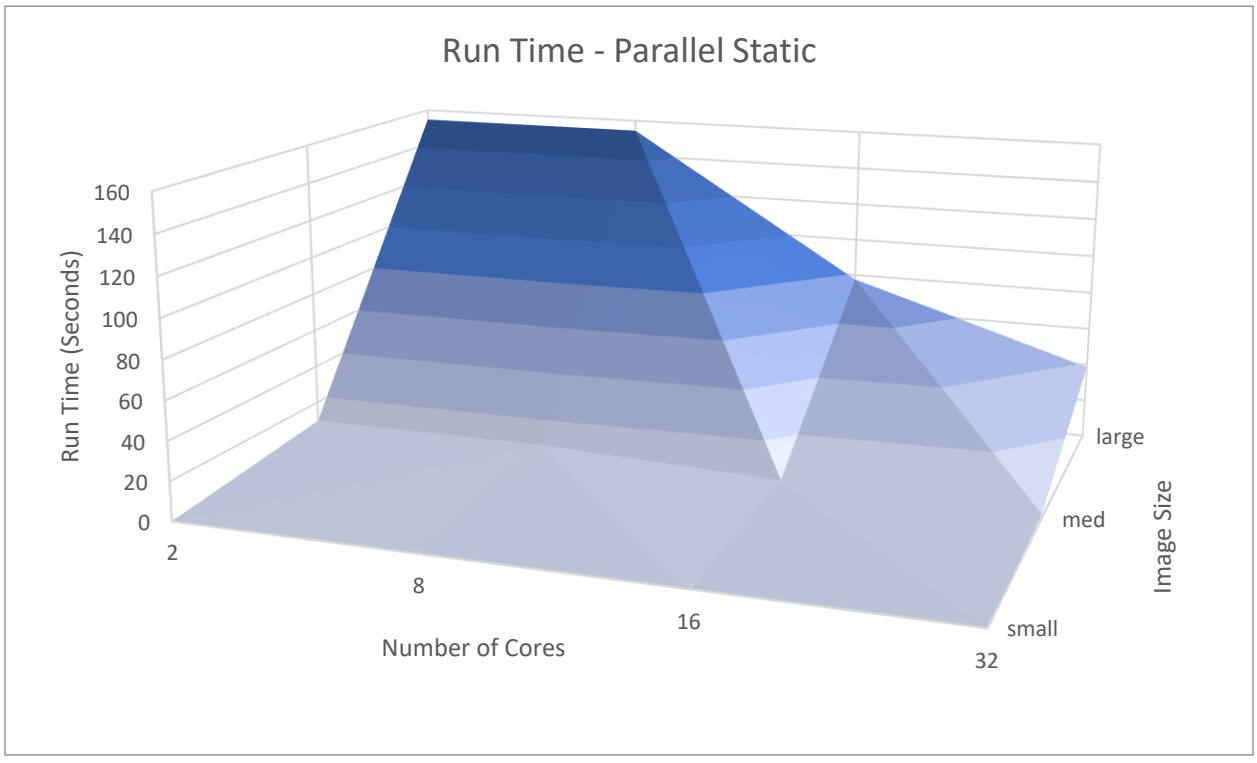


Figure 2 - Run Time Parallel Static

The above graph represents a surface plot of the data set with the number of cores as the long axis, the image size as the depth and run time as the vertical axis. It is evident that the greater the number of cores, the quicker the image will be processed. However, the run time is similar for both 2 and

8 cores processing a large image as depicted by the peak in the far-left corner of the surface plot. The number of cores is insignificant when generating a small image as many of them give a similar run time. A significant decrease in run time is only found when the number of cores is greater than 16 and the image is large. These observations must result from the nature of static task assignment, there exists some threshold where the image size does not matter, regardless of the number of cores, it will still be completed in a significant amount of time.

Below is a graph (Fig. 3) representing the data calculated for speed up.

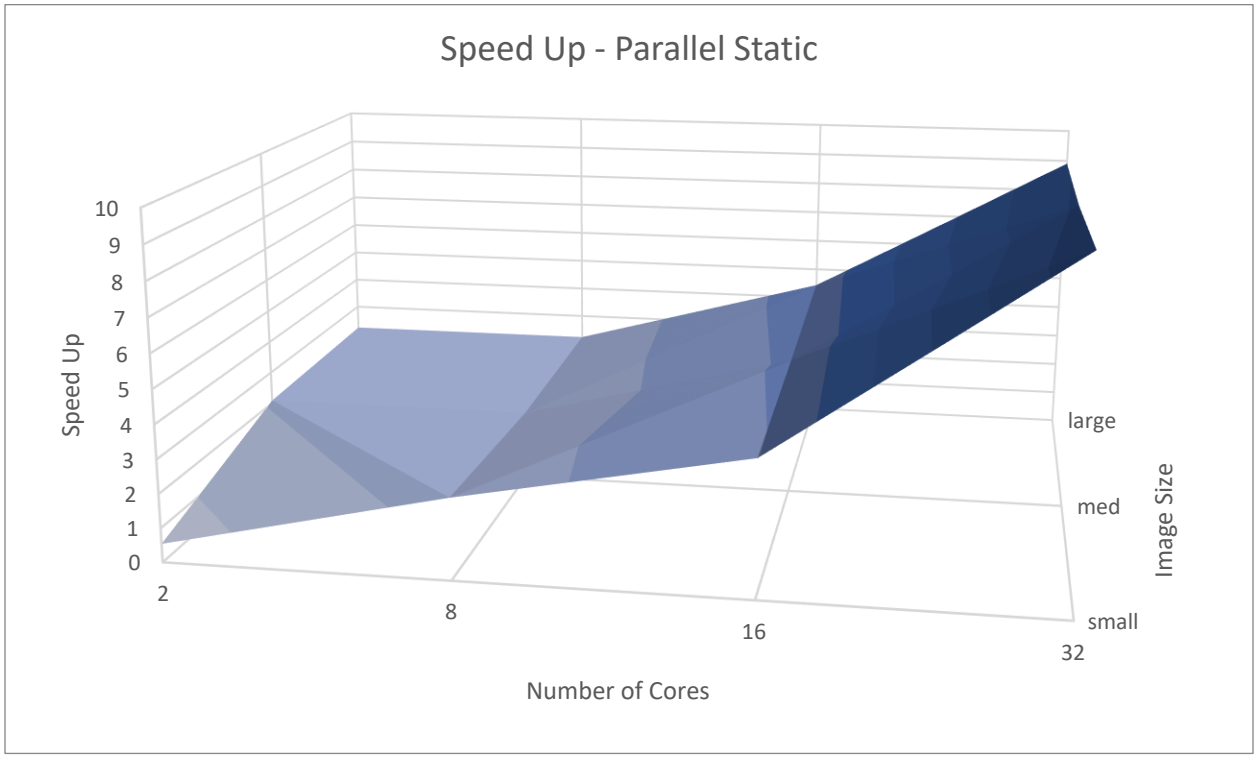


Figure 3 - Speed Up Parallel Static

The above graph represents the speed up of the parallel static implementation in contrast with the sequential implementation. Speed up is defined as the ratio of the run time sequentially divided by the run time in parallel. This graph makes it evident that regardless of the image size, there is a direct increase in speed up with an increase of cores. That is, the more cores there are, the faster the implementation. The speedup reaches slightly above 9 with 32 cores in this implementation.

The below graph (Fig. 4) is a surface plot representing the data calculated for efficiency. In parallel computing, efficiency is calculated as the ratio of speed up divided by the number of cores. This formula is meant to give a representation of the meaningful work each core will perform but in reality, the number of cores should give diminishing returns.

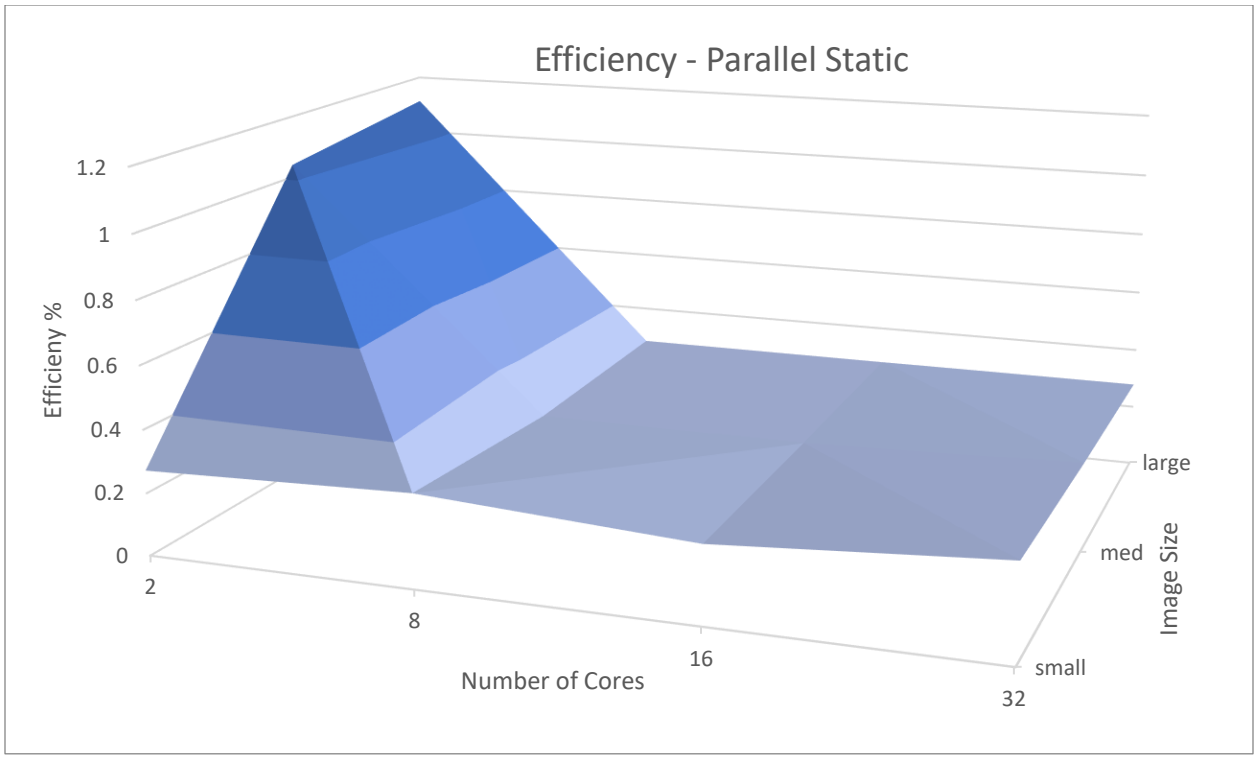


Figure 4 - Efficiency Parallel Static

The graph assures the prediction of diminishing returns. It is almost an antithesis of the speed up graph. The greater the number of cores does not guarantee a greater amount of efficiency but instead the opposite. This relates to Amdahl's law where a parallel process can only reach a certain threshold of optimization.

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 computer if 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 number of processors has increased.

Output from /bin/dynamic.log

```
10.76
```

Data was collected by finding the run time from generating a Mandelbrot image of various sizes and various amount of processor cores.

Image Size	**Time**			
	2 Processors	*8 Processors*	*16 Processors*	*32 Processors*
Small (500x500)	0.052019	0.008713	0.006618	0.005293
Medium (1000x1000)	19.4495	2.7862	1.44786	29.8874
Large (40000x40000)	310.642	44.5241	23.0609	243.764

Below is a graph (Fig. 5) representing the above data set.

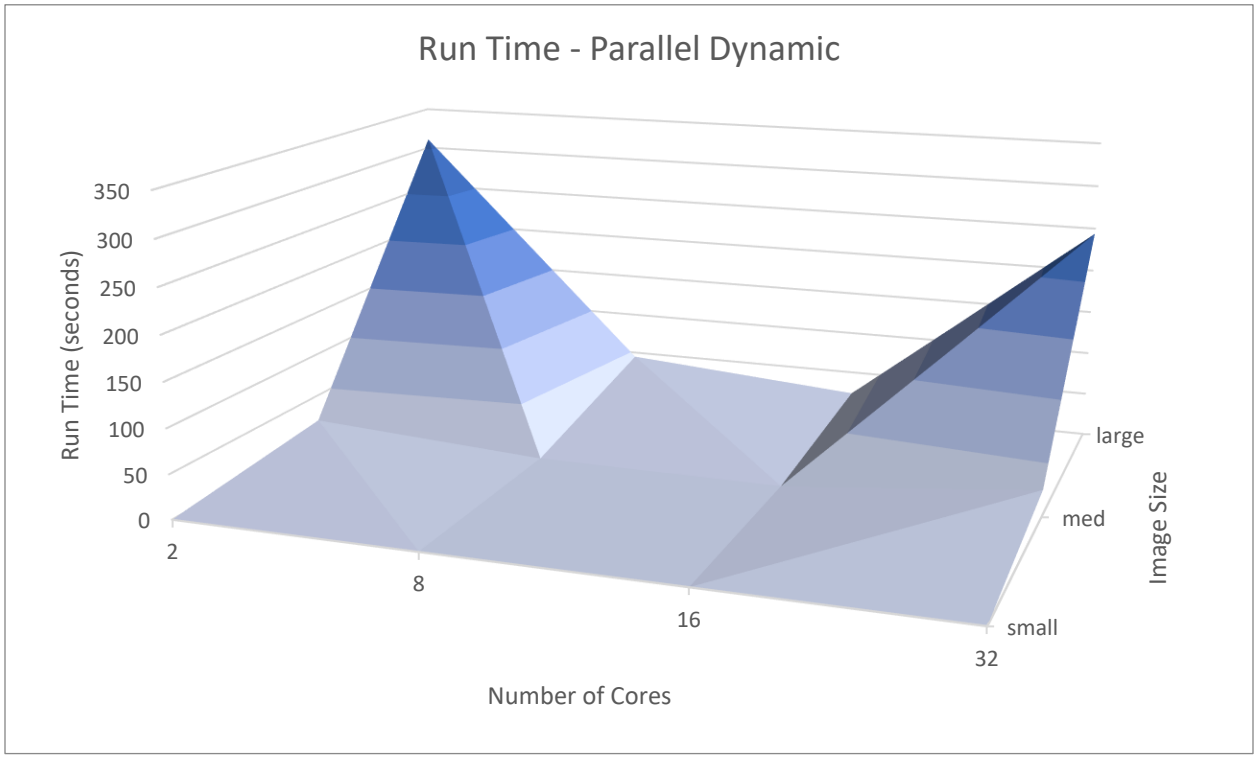


Figure 5 - Run Time Parallel Dynamic

The above graph represents a surface plot of the data set with the number of cores as the long axis, the image size as the depth and run time as the vertical axis. It is evident that a small image is

insignificant to process in terms of run time. However, there is an interesting relationship between the number of cores with large images and run time. There is a “goldilocks” spot for the number of cores and a large image. Too many cores will result in an overflow of slaves that increase the run time rather than optimizing the implementation. This increase in run time could also be due to the fact that the communication time will dominate with a large number of cores.

Below is a graph (Fig. 6) representing the data calculated for the speed up ratio based on the data set for sequential and parallel dynamic implementations. Speed up is defined as the run time sequentially divided by the run time in parallel.

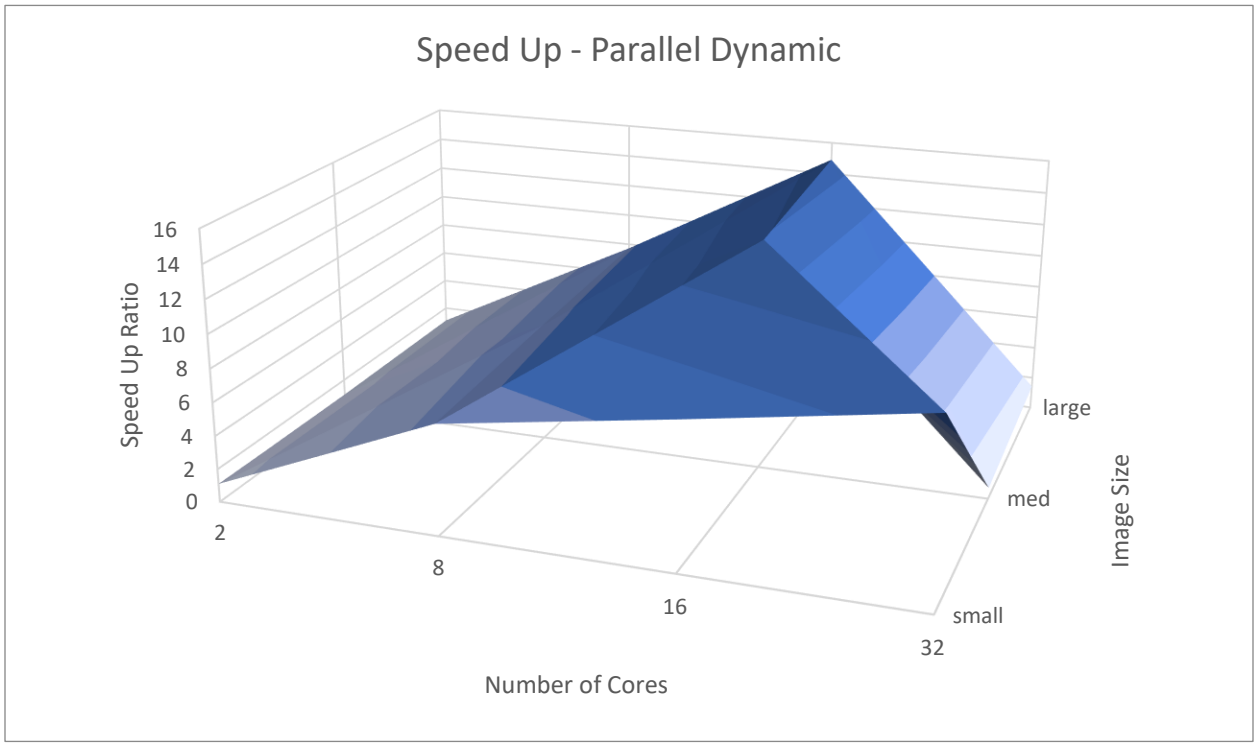
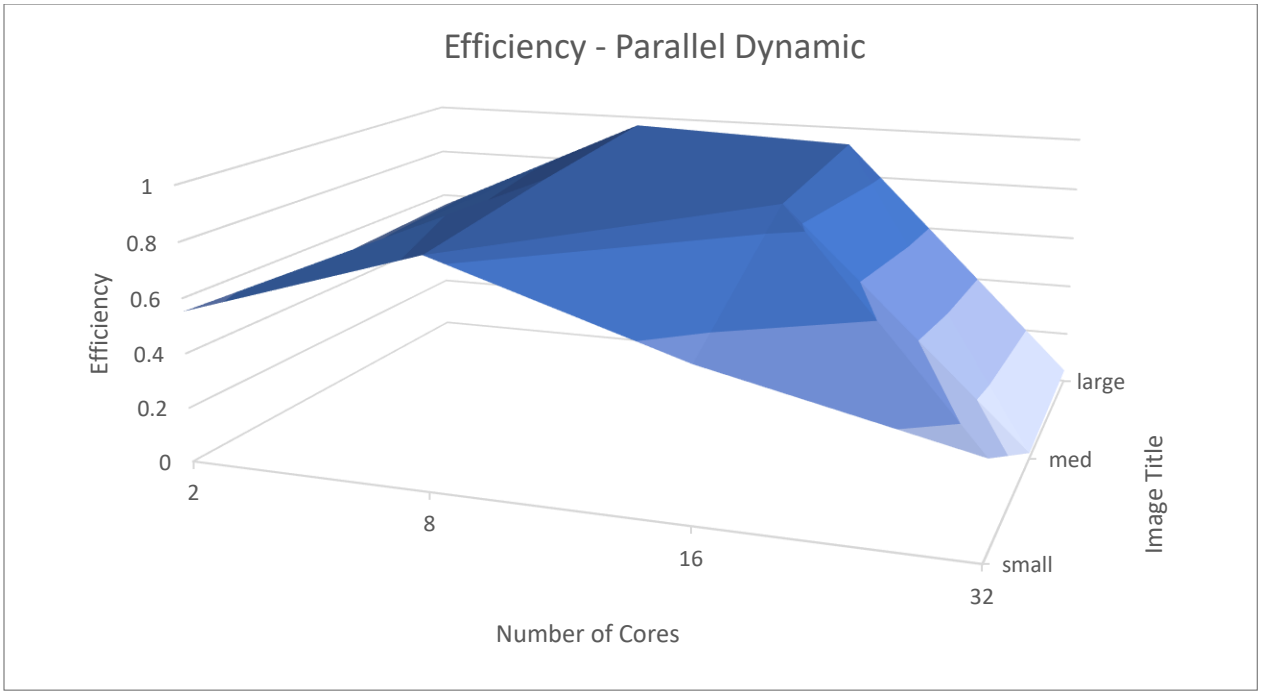


Figure 6 – Speed Up Parallel Dynamic

The above graph visually correlates with the previous graph. The speed up reaches a peak between the implementations with 32 and 8 cores. 16 cores seem to be the most optimal number of cores to implement the process with as it gives the greatest amount of speed up. The speed up drops off in both directions of increasing or decreasing numbers of cores, regardless of image size. This implies that the cost of communication and computation is most optimal at 16 cores.



The above graph (Fig. 7) is a surface plot representing the data calculated for efficiency. In parallel

Figure 7 - Efficiency Parallel Dynamic

computing, efficiency is calculated as the ratio of speed up divided by the number of cores. This formula is meant to give a representation of the meaningful work each core will perform. In this implementation, it is evident that the greater the number of cores does not produce a large amount of efficiency but neither does the opposite. The best efficiency correlates to the previous observations. The zone where the most efficiency happens is located at somewhere near the 8 and 16 core areas. This implies that the most optimal number of cores for most image sizes should be in this range.

Issues

The main issue confronted with this project was determining how to use each result from the slave processes by the master. It was difficult to understand how to put the row that was returned into the correct spot in the image. The best solution was using an array that kept track of the rows sent to each slave process. The other significant 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.

Conclusion

The dynamic and static task assignment implementations vary significantly in terms of speed up and efficiency. There are times when one outperformed the other and both have their advantages and disadvantages. Dynamic would be best for a homogenous cluster with not too many nodes.