Project Assignment 2

Raymond M. Pistoresi

March 20, 2017

CS 415, Parallel Computing
Department of Computer Science and Engineering
University of Nevada, Reno
Spring 2017

Professor Dr. Frederick C. Harris, Jr.

Table of Contents

Introduction

This project assignment is the implementation of a program that generates the Mandelbrot set on the University of Nevada, Reno's cluster. The programs will output a file containing the Mandelbrot set as a pixel image and report the calculation times. There are essentially two program variations to this assignment: sequential and parallel execution. Within the parallel variation there are two options for implementation: static and dynamic load balancing. Runtime, efficiency, and speedup are analyzed and discussed with respect to their implementation and run environment during the experiment.

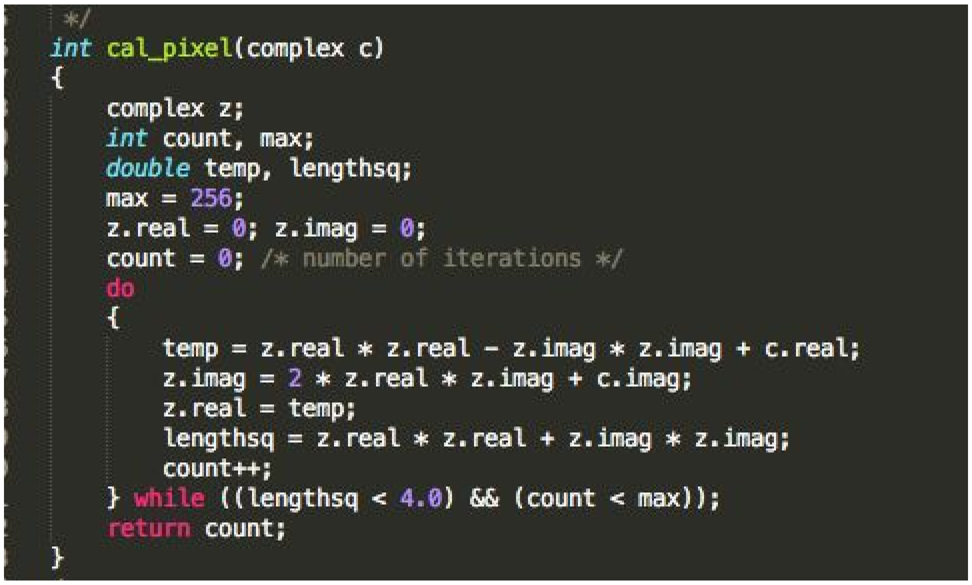
The results are outlined in the following format for the Data section of this report:

1. Sequential variation
2. Parallel variation
    a. Static load balancing
    b. Dynamic load balancing

Data

1. Sequential Variation

Although the code was written to run on the h1 cluster at the University using MPI, it can be easily modified, and was during development, on a single local machine since it only utilizes one processor in its execution. This variation simply iterates through all the pixels of a given image size and calculates whether or not that pixel exist in the Mandelbrot set using the algorithm pseudo code for cal_pixel() which was provided from the course's textbook, "Parallel Programming (2nd ed.) by Wilkinson and Allen" **(Appendix-1)**. In addition, Dr. Harris provided a header and implementation file for outputting pixel images to a filestream for use in this project as PIMFuncs.h and PIMFuncs.cpp **(Appendix-3)**. Both code resources with some modification were used to meet the requirements and testing of this experiment. Here, **(Fig.1)**, we see the algorithm from the aforementioned book implemented in code to calculate each pixel of the image and determine its inclusion in the set.



```
*/
int cal_pixel(complex c)
{
    complex z;
    int count, max;
    double temp, lengthsq;
    max = 256;
    z.real = 0; z.imag = 0;
    count = 0; /* number of iterations */
    do
    {
        temp = z.real * z.real - z.imag * z.imag + c.real;
        z.imag = 2 * z.real * z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real * z.real + z.imag * z.imag;
        count++;
    } while ((lengthsq < 4.0) && (count < max));
    return count;
}
```

Fig. 1: Function implementation of calculating one pixel and determine if it is in the Mandelbrot set.

A sequential approach uses two for loops to iterate through all rows and columns of the image and stores them respectively into a two dimensional array. Once all calculations are completed and values store, the image is outputted to a file "sequential.pgm" in black and white in the project's bin directory. Several tests were run with varying image sizes. The outputs for 1000x1000 pixels (**Fig. 2a & 2b)** and 32,000x32,000 **(Fig. 3a & 3b)** image sizes express the level of difference in detail that can be achieved.

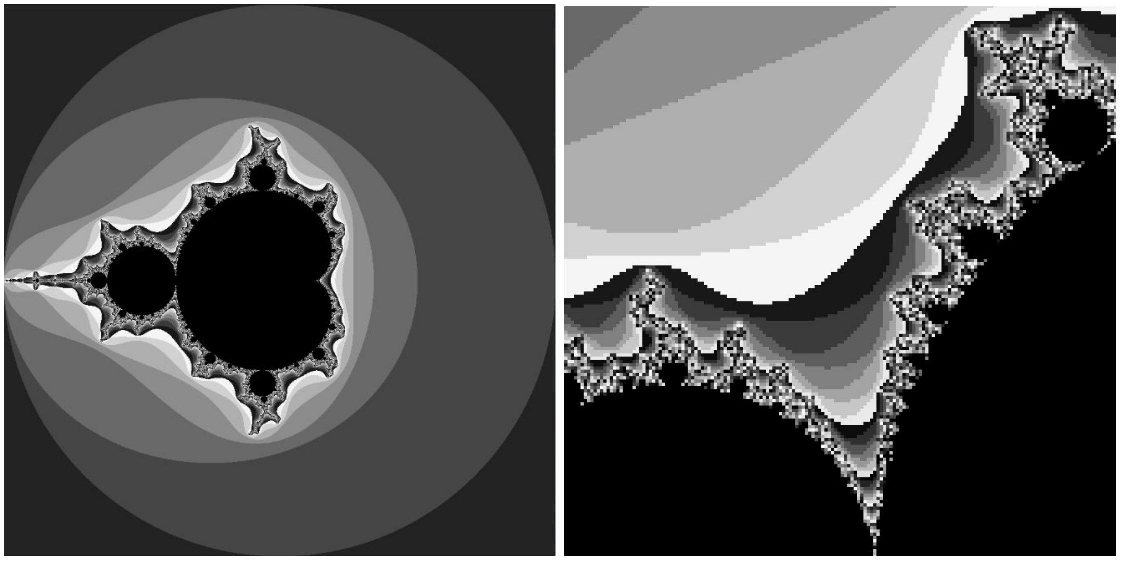


Fig. 2a & 2b: Output from running sequential Mandelbrot set program on an image size of 1000x1000 pixels with greyscale escape term 35.

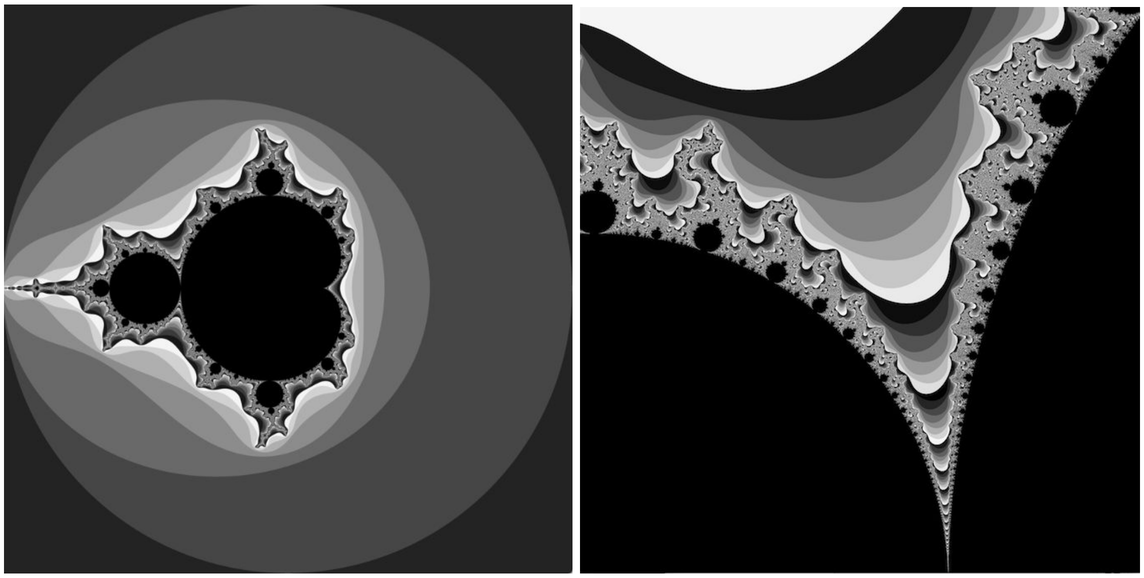
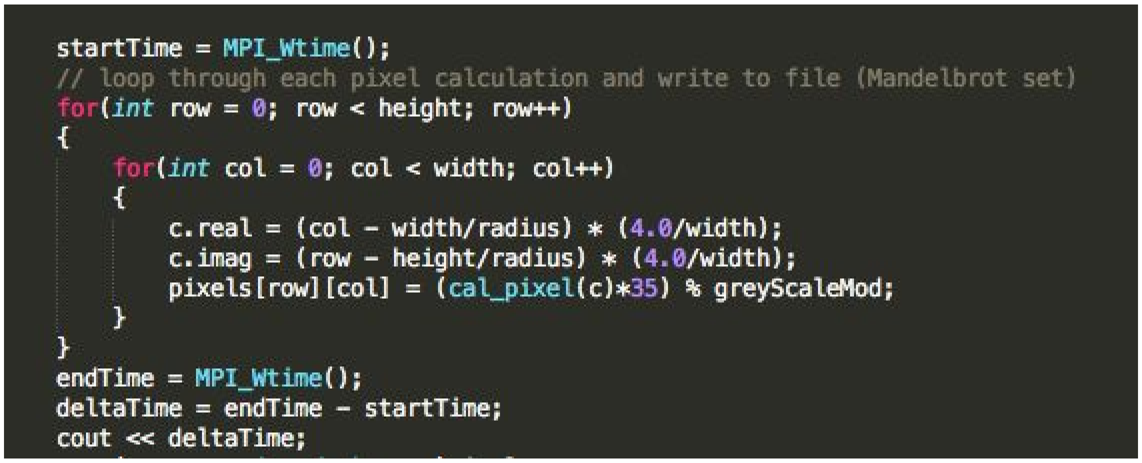


Fig. 3a & 3b: Output from running sequential Mandelbrot set program on an image size of 32,000x32,000px with greyscale escape term 35.

It clearly shows the image has the same structure regardless of the amount of detail displayed and this held true for all of the test runs at different images sizes.

Timing was calculated with respect to the program calculating the Mandelbrot set and is shown in implementation in **Fig. 4**. The time is returned to the user at the end of the program in milliseconds. Each image size was tested five times and plotted on the graph in **Fig. 5**. The image sizes 1x10^3 to 32x10^3 were used to get a good idea of how the sequential run times increase drastically given larger sizes.



```
startTime = MPI_Wtime();
// loop through each pixel calculation and write to file (Mandelbrot set)
for(int row = 0; row < height; row++)
{
    for(int col = 0; col < width; col++)
    {
        c.real = (col - width/radius) * (4.0/width);
        c.imag = (row - height/radius) * (4.0/width);
        pixels[row][col] = (cal_pixel(c)*35) % greyScaleMod;
    }
}
endTime = MPI_Wtime();
deltaTime = endTime - startTime;
cout << deltaTime;
```

Fig. 4: Timing routines used to calculate and return elapsed time.

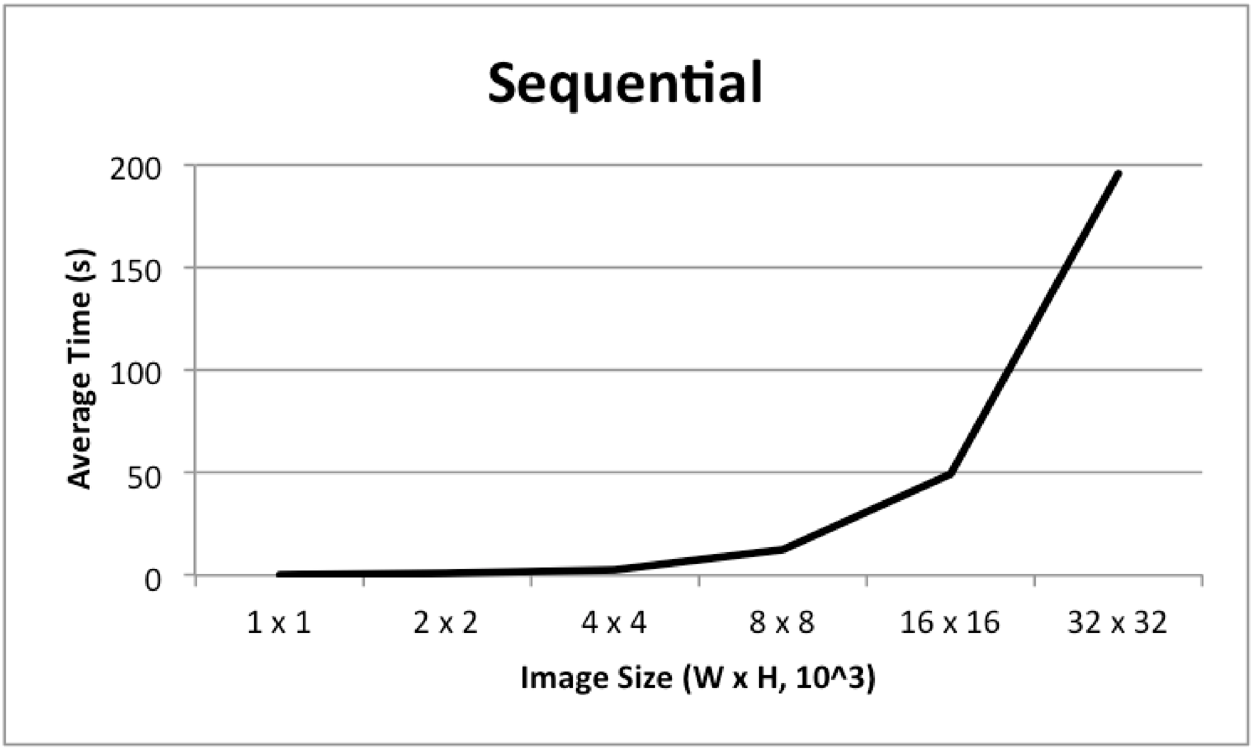


Fig. 5: Runtime for sequential calculations. Each image size runtime was averaged over five test runs.

The results for the sequential variation has been stored “Run_Results.xlsx” file in the bin directory.

In the lower resolution sizes, it was clear that a sequential program could do the calculations very quickly; however, as the image size increased, so did the calculation time. This problem lends itself to be nearly embarrassingly parallel due to the independent pixel calculations and the next section describes a faster and more efficient solution to calculate the Mandelbrot set.

2. Parallel Variation

During the sequential portion of this assignment, it was discovered that each pixel was calculated independent of every other pixel resulting in a nearly embarrassingly parallel algorithm. By using the MPI send and receive routines, it was simple to have a master process divide up the work to be done between slaves and receive the results once each slave completed their work. Since it is important to consider how a multiple processor program will distribute the work to achieve the best results, two load balancing techniques are explored: static and dynamic.

In both renditions of the parallel variation, many key aspects are the same such as dividing up work into rows of pixels by the master processor and the slaves do the calculations. The master then collects all the data and outputs the calculation timing and

Mandelbrot set .pgm image. The key differences are discussed in the sub headings below along with the data that resulted.

a. Static

For the static implementation, the number of processors is determined through the MPI communication world size and the image height is divided equally among the slave processors. Let it be noted here that this implementation does not have the master processor doing any of the pixel calculations; however, it is a possibility for improvement to the code that could later be implemented if the row lengths were much larger. If there is an odd number of rows to be divided among the processors, the remaining rows are sent in sequential order to the slaves who were first to receive any work. Again, here we could have sent chunks of pixels instead of rows as a different method for partitioning the work. The timing of our calculations starts when the master starts the row distribution process and ends once all the rows have been collected. The output text and files are not included in the calculation times as was the same in the sequential variation. After all the rows are distributed, a kill message is sent to all the slaves to let them know there is no more work to be done and the master then moves into a loop to wait and receive all the rows back from the slaves. The runtime results are shown in **Fig. 6** using this implementation. During the testing portion, image sizes remained the same as the sequential variation for comparison among varying amounts of processors, or cores.

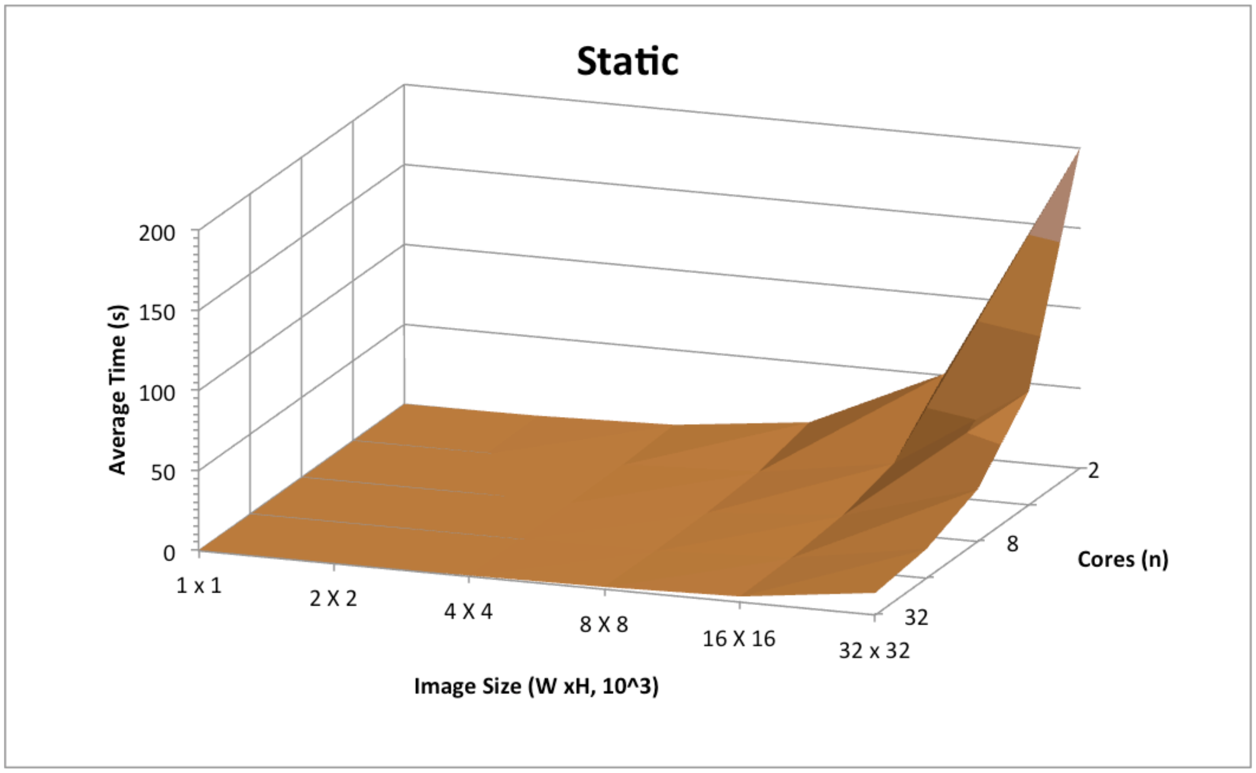


Fig. 6: 3D graph showing runtime result for static load balancing at various image sizes and number of cores.

It is clear that as the image size increases along with the number of cores, we get a great reduction in runtime. There is another drastic increase in time when the number of cores decreases. In **Table 1**, a better sense of what happen is given by comparison between the sequential and parallel variations.

		Static Speedup Factor				
		Sequential Time/Static Time				
	Cores (n)	2	4	8	16	32
Image Size (W x H, 10^3)	1 x 1	0.963	2.576	5.836	4.767	0.552
	2 X 2	0.954	2.706	5.968	7.215	1.894
	4 X 4	0.957	2.755	6.059	10.298	5.067
	8 X 8	0.958	2.767	6.049	10.727	8.751
	16 X 16	0.959	2.769	6.038	10.774	14.093
	32 x 32	0.958	2.766	6.028	10.732	14.290

Table 1: Static speedup(sequential time/static time) factor results.

By looking at the column where two cores are being used, the speedup factor is less than one and indicating a slowdown across all image sizes. This is due to the fact that only one slave is actually doing any calculations while the master just distributes and receives. In essence, this is doing the sequential variation but having to pass messages between the master and slave resulting in overhead and therefore slowdown.

Between four and thirty-two cores, as the cores and image size increase, so does the speedup. However, notice when thirty-two cores is reached for the first image size (1 x 10^3) x (1 x 10^3), a slowdown occurs again. This is due to the fact that there is not enough work for the slaves and they complete the jobs quicker than the master and distribute and collect the work back effectively.

Another interesting result of this comparison is the arcing trend that happens in all core and image combinations. It is known from Amdahl's Law that there is a maximum speedup achievable and our data is eluding to this fact. In the two and four core tests, not much speedup happens with the increase of image size but for the sixteen and thirty-two cores a 3- to 5- fold increase occurs. The sixteen core runs start leveling off at 10.7 seconds and drop off slightly. Likewise the thirty-two core runs start to level off around 14.0 seconds and it is safe to predict a similar trend in image sizes that go beyond the ones shown.

Efficiency results for this work distribution variation compared to the sequential variation can be seen in **Table 2** in the block on the right. Based on the data that has been shown thus far, it is easy to confirm the suspicion that using the largest core count on the smallest image would produce the least efficient execution. The 8 core run with an image size of (4 x 10^3) x (4 x 10^3) was the most efficient at 75.735%. Since the image size is evenly doubled each time, this might be hiding why this configuration was the most efficient even though it did not have the best speedup factor. With 32 cores being used, the row sizes, even at 32 X 10^3 were not big enough to create the least idle time and on all accounts were worse than the efficiency of using 2 cores.

		Static Speedup Factor					Static Efficiency				
		Sequential Time/Static Time					(Speedup Factor/n Cores)*100				
	Cores (n)	2	4	8	16	32	2	4	8	16	32
Image Size (W x H, 10^3)	1 x 1	0.963	2.576	5.836	4.767	0.552	48.169	64.410	72.953	29.791	1.725
	2 X 2	0.954	2.706	5.968	7.215	1.894	47.706	67.640	74.603	45.093	5.919
	4 X 4	0.957	2.755	6.059	10.298	5.067	47.847	68.884	75.735	64.360	15.836
	8 X 8	0.958	2.767	6.049	10.727	8.751	47.910	69.171	75.614	67.041	27.348
	16 X 16	0.959	2.769	6.038	10.774	14.093	47.929	69.234	75.474	67.339	44.039
	32 x 32	0.958	2.766	6.028	10.732	14.290	47.895	69.151	75.348	67.075	44.656

Table 2: Speedup Factor and Efficiency(%) results in parallel variation with static load balancing.

Achieving a speedup factor of 14.29 is pretty good, yet the efficiency is quite low. The next section will explore a dynamic load balancing variation that could possibly lead to higher speedup and or efficiency. Static versions are great when all the work is known beforehand and it can be evenly distributed but when the amount of work is unknown, a dynamic approach can be more practical.

b. Dynamic

During the static load balancing, all of the work was known beforehand and thus made it easy to determine how much work each processor should do. If the amount of work is not known ahead of time, the program needs a better way to manage how the work will distributed during the run. The scheduling of work should also consider the speed in which each slave does the work. On the university's cluster, the nodes are fairly even in term of speed and results were respectable for the static implementation. But, if there were issues with a slave or the configuration was poorly implemented, a slow processor or two could drastically reduce the expected improvements. In this version of dynamic load balancing, the master distributes one row of work to all the processors in the communication world and then moves into a receive and send loop. Giving one row to each processor gets the work started but only when a slave is done with that row will it receive another row to work from the master. With this change in distribution, the master does not have to wait for a slow slave before giving out more work and the work it does give out has the chance to be done quicker than that slower processor. The runtimes for this new variation are shown in **Fig. 7**.

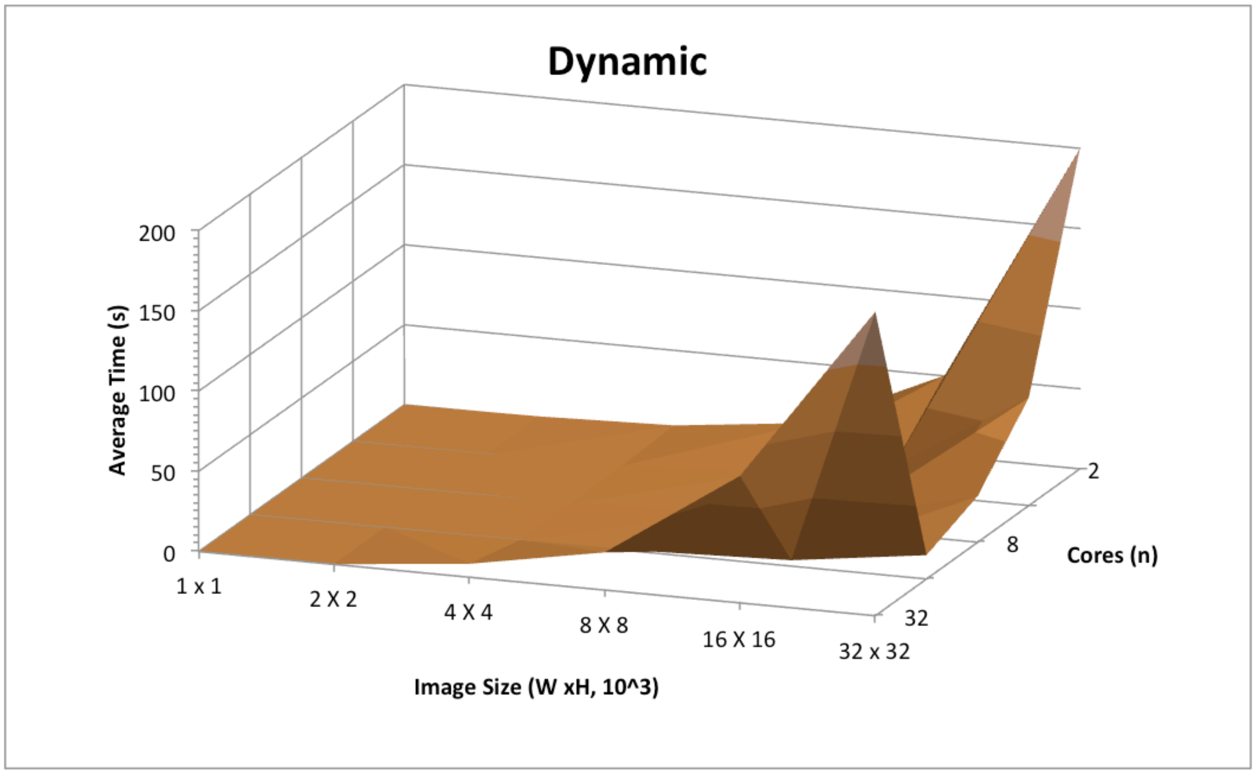


Fig. 7: 3D graph showing runtime result for dynamic load balancing at various image sizes and number of cores.

		Dynamic Speedup Factor				
		Sequential Time/Dynamic Time				
	Cores (n)	2	4	8	16	32
Image Size (W x H, 10^3)	1 x 1	0.963	2.722	6.543	4.992	2.569
	2 X 2	0.964	2.881	6.698	9.465	6.279
	4 X 4	0.971	2.899	6.789	11.559	0.368
	8 X 8	0.973	2.912	6.800	12.994	0.523
	16 X 16	0.974	2.917	6.813	13.455	0.620
	32 x 32	0.974	2.917	6.811	13.251	1.035

Table 3: Dynamic speedup(sequential time/static time) factor results.

Speedup between sequential and dynamic variations is shown in **Table 3**. Here the largest speedup factor is 13.46 in a 16 core run on an image size of (16 x 10^3) x (16 x 10^3). Again, there is a slowdown in the use of only 2 cores like in the static variation as expected. However, the interesting data here is on the 32 core runs. Very little speedup occurs unlike in the static variation. Since it was expected to have a greater speedup as the cores were increased, it was surprising to see this result. The natural conclusion is that because there are so many messages, or rows, being passed for the increasing image size, the cost of communication was much too great and completely bogged down the master process. After giving one row to each processor, the master was already so backed up with jobs to receive and then send back, it could never keep up with the slaves. In all versions up to 32 cores, there was a steady increase in speedup and then plateau just like in the static version as well.

To further investigate and gain some insight to whether the master was indeed the bottleneck in our runs, another test was performed using a hybrid approach to the load balancing. **Table 4** and **Fig. 8** depict the results from this experiment.

	32 Cores	Time (s)
Initial Rows Distributed (32x10^3)^2 image	5	50.7964
	10	30.7741
	50	13.7592
	100	12.2328
	150	14.9816
	500	66.9627

Table 4: Table showing number of rows distributed to all processors before dynamic send and receiving starts in the master node using 32 cores on image size of (32 x 10^3) x (32 x 10^3).

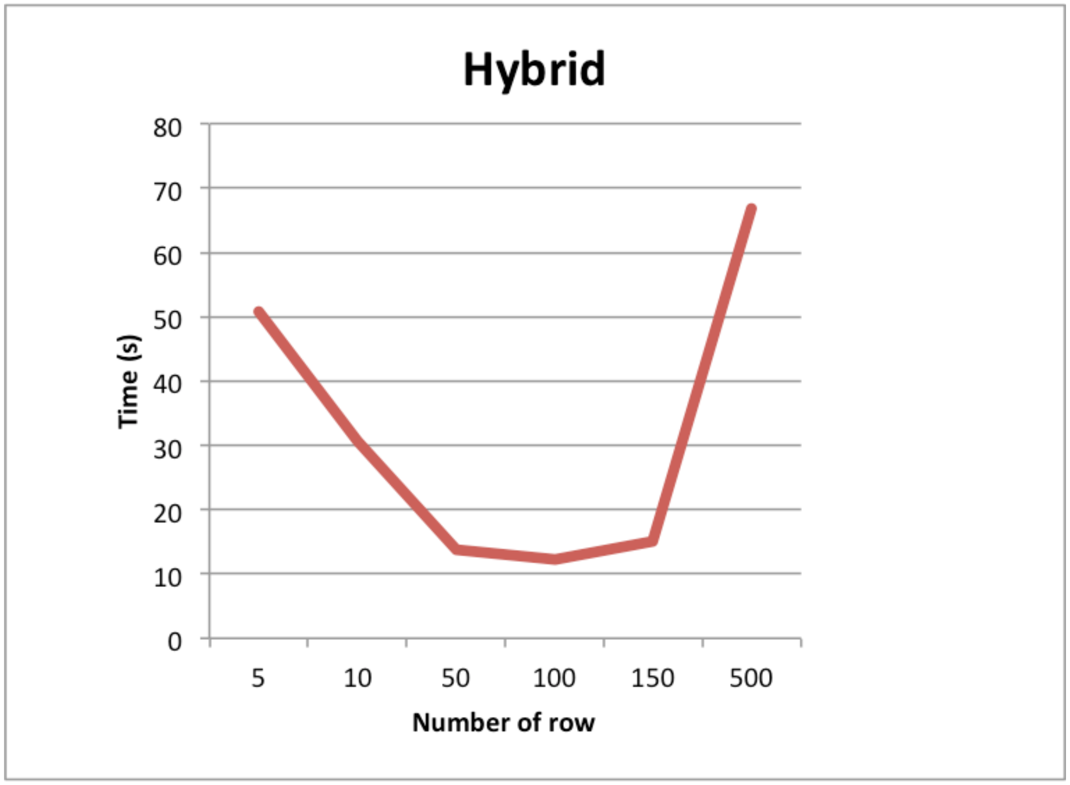


Fig. 8: Graph showing runtimes as a result of hybrid load balancing using 32 cores on image size of (32 x 10^3) x (32 x 10^3).

These results confirm the suspicion that there was indeed a bottleneck at the beginning of the dynamic distribution. As little as 5 rows were given to each processor at the beginning and the runtime dropped from 201.40 seconds to 66.96 seconds. This hybrid version also has a limit to how much it can improve the dynamic version run in this configuration and between 50 to 150 rows the runtime bottoms out. It is also important to note that these numbers of row given out and the results are not directly applicable to other core and image size parameters and further testing would need to be done to get a better idea of how the overall problem should be modified based on the goals desired.

The efficiency of the dynamic variations is shown in **Table 5**. Again, the most efficiency but not the most speedup was achieved by the 8 core configuration, but this time with the (16 x 10^3) x (16 x 10^3) image for 85.168%. And again, there is a bubble that each one achieves before starting to drop off. **Fig. 9** and **Fig. 10** might give a better idea of how the two different variations performed though.

		Dynamic Speedup Factor					Dynamic Efficiency				
		Sequential Time/Dynamic Time					(Speedup Factor/n Cores)*100				
	Cores (n)	2	4	8	16	32	2	4	8	16	32
Image Size (W x H, 10^3)	1 x 1	0.963	2.722	6.543	4.992	2.569	48.144	68.040	81.793	31.199	8.027
	2 X 2	0.964	2.881	6.698	9.465	6.279	48.223	72.034	83.722	59.157	19.623
	4 X 4	0.971	2.899	6.789	11.559	0.368	48.547	72.486	84.864	72.243	1.150
	8 X 8	0.973	2.912	6.800	12.994	0.523	48.644	72.810	85.006	81.213	1.634
	16 X 16	0.974	2.917	6.813	13.455	0.620	48.707	72.934	85.168	84.093	1.938
	32 x 32	0.974	2.917	6.811	13.251	1.035	48.688	72.926	85.135	82.817	3.234

Table 5: Speedup Factor and Efficiency(%) results in parallel variation with dynamic load balancing.

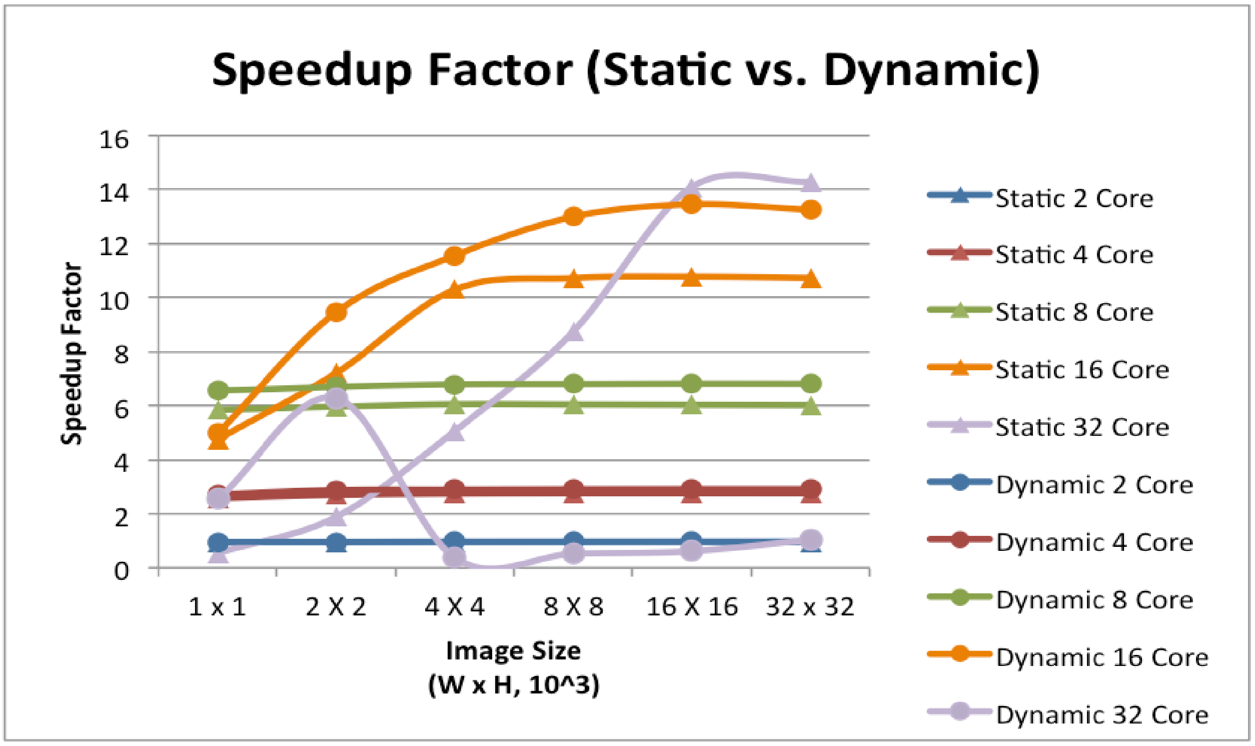


Fig. 9: Speedup factor results comparing the static and dynamic variations.

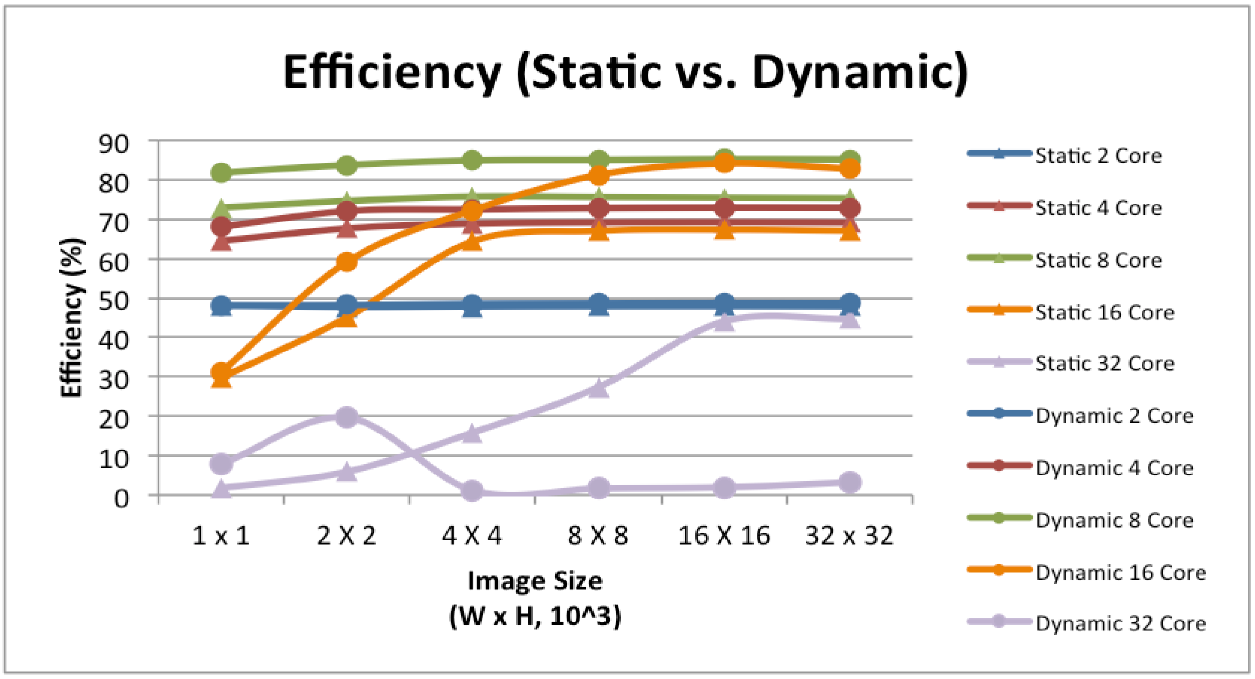


Fig. 10: Efficiency results comparing the static and dynamic variations.

Conclusion

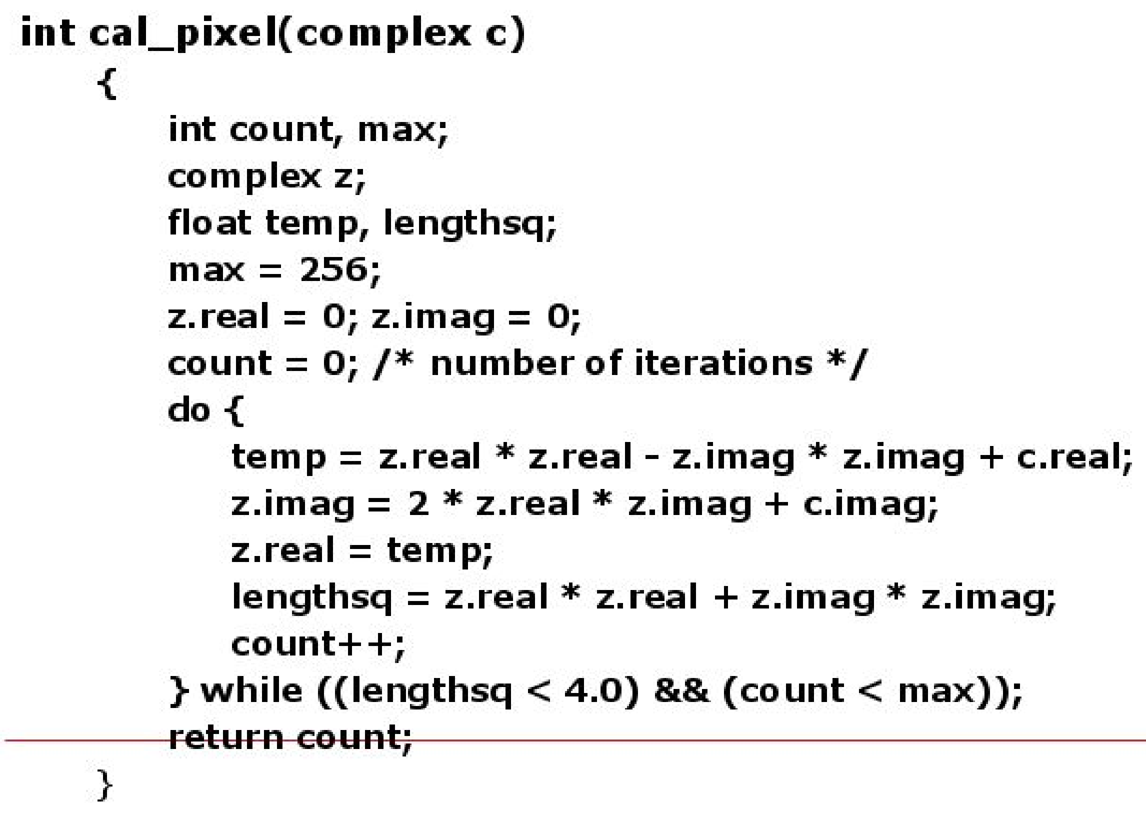
This project assignment was a huge lesson in how to program for a somewhat simple problem in using parallelization techniques. Thinking about how much data needs to be worked with, the configuration of the computing cluster, and implementation of different scheduling plays a massive role in how a problem can be solved. There were ways to achieve great speedup at the cost of efficiency and there were ways to completely waste most of the resources allocated. Every situation will be different but understanding the problem and what is desired can help aid in the direction that a parallel program can be written to achieve better performance. There seemed to be endless possibilities when testing the ways this project could have been done. The images could have been divided into blocks of pixels instead of simply sending rows the length of the image width for one example. Regardless of the variations that exists, much was discovered about parallelization, load balancing, and testing extreme values to gain some insight into their meaning.

In **Appendix-2** there contains the raw data for all the variations runtimes that were used in the respective graphs in this report for reference. The source code, graph data, scripts, makefile, executables, and readme files for this project all reside on the github repository htttps://github.com/rpistoresi/cs415_Pistoresi.git in the PA2 directory. Further instructions on how to compile and run the scripts for this experiment are located in

README.md. For each variation of the experiment, there are corresponding Excel sheets in the file Run_Results.xlsx located in the bin folder.

Appendix-1

“Parallel Programming (2nd ed.) by Wilkinson and Allen” algorithm to calculate one pixel to determine its set value:



```
int cal_pixel(complex c)
    {
        int count, max;
        complex z;
        float temp, lengthsq;
        max = 256;
        z.real = 0; z.imag = 0;
        count = 0; /* number of iterations */
        do {
            temp = z.real * z.real - z.imag * z.imag + c.real;
            z.imag = 2 * z.real * z.imag + c.imag;
            z.real = temp;
            lengthsq = z.real * z.real + z.imag * z.imag;
            count++;
        } while ((lengthsq < 4.0) && (count < max));
        return count;
    }
```

Appendix-2

Sequential runtimes:

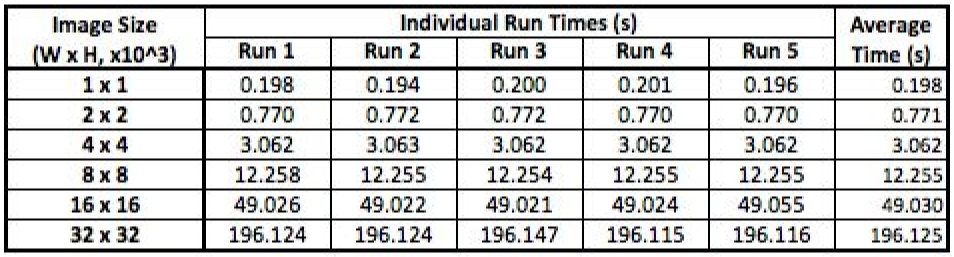


Image Size (W x H, x10^3)	Individual Run Times (s)					Average Time (s)
	Run 1	Run 2	Run 3	Run 4	Run 5	
1 x 1	0.198	0.194	0.200	0.201	0.196	0.198
2 x 2	0.770	0.772	0.772	0.770	0.770	0.771
4 x 4	3.062	3.063	3.062	3.062	3.062	3.062
8 x 8	12.258	12.255	12.254	12.255	12.255	12.255
16 x 16	49.026	49.022	49.021	49.024	49.055	49.030
32 x 32	196.124	196.124	196.147	196.115	196.116	196.125

Static runtimes:

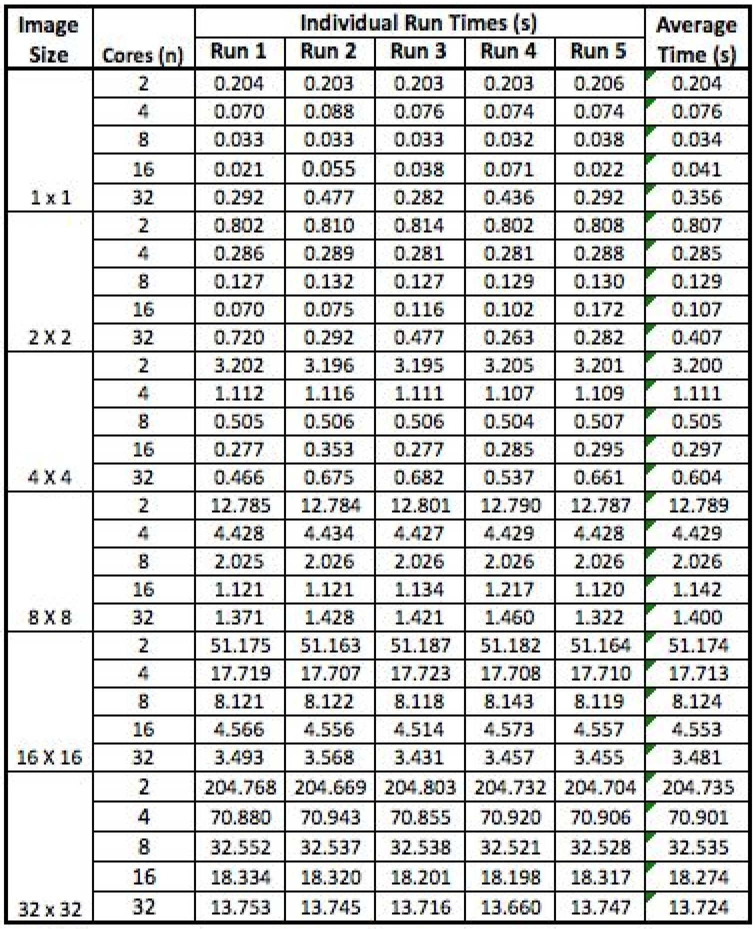


Image Size	Cores (n)	Individual Run Times (s)					Average Time (s)
		Run 1	Run 2	Run 3	Run 4	Run 5	
	2	0.204	0.203	0.203	0.203	0.206	0.204
	4	0.070	0.088	0.076	0.074	0.074	0.076
	8	0.033	0.033	0.033	0.032	0.038	0.034
	16	0.021	0.055	0.038	0.071	0.022	0.041
1 x 1	32	0.292	0.477	0.282	0.436	0.292	0.356
	2	0.802	0.810	0.814	0.802	0.808	0.807
	4	0.286	0.289	0.281	0.281	0.288	0.285
	8	0.127	0.132	0.127	0.129	0.130	0.129
	16	0.070	0.075	0.116	0.102	0.172	0.107
2 X 2	32	0.720	0.292	0.477	0.263	0.282	0.407
	2	3.202	3.196	3.195	3.205	3.201	3.200
	4	1.112	1.116	1.111	1.107	1.109	1.111
	8	0.505	0.506	0.506	0.504	0.507	0.505
	16	0.277	0.353	0.277	0.285	0.295	0.297
4 X 4	32	0.466	0.675	0.682	0.537	0.661	0.604
	2	12.785	12.784	12.801	12.790	12.787	12.789
	4	4.428	4.434	4.427	4.429	4.428	4.429
	8	2.025	2.026	2.026	2.026	2.026	2.026
	16	1.121	1.121	1.134	1.217	1.120	1.142
8 X 8	32	1.371	1.428	1.421	1.460	1.322	1.400
	2	51.175	51.163	51.187	51.182	51.164	51.174
	4	17.719	17.707	17.723	17.708	17.710	17.713
	8	8.121	8.122	8.118	8.143	8.119	8.124
	16	4.566	4.556	4.514	4.573	4.557	4.553
16 X 16	32	3.493	3.568	3.431	3.457	3.455	3.481
	2	204.768	204.669	204.803	204.732	204.704	204.735
	4	70.880	70.943	70.855	70.920	70.906	70.901
	8	32.552	32.537	32.538	32.521	32.528	32.535
	16	18.334	18.320	18.201	18.198	18.317	18.274
32 x 32	32	13.753	13.745	13.716	13.660	13.747	13.724

Dynamic runtimes:

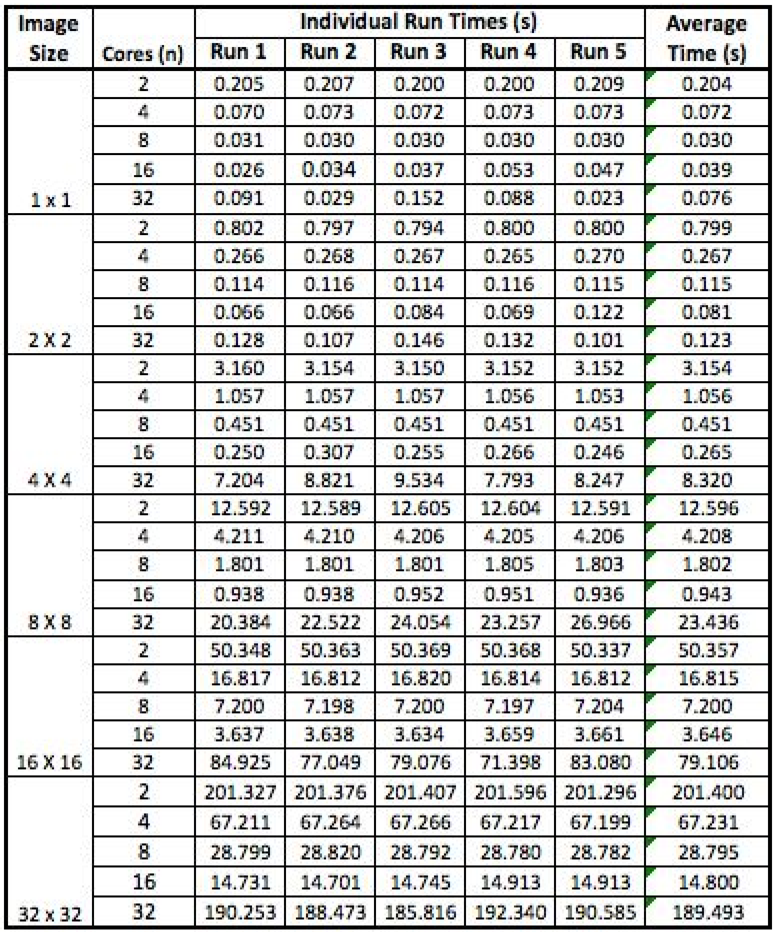


Image Size	Cores (n)	Individual Run Times (s)					Average Time (s)
		Run 1	Run 2	Run 3	Run 4	Run 5	
	2	0.205	0.207	0.200	0.200	0.209	0.204
	4	0.070	0.073	0.072	0.073	0.073	0.072
	8	0.031	0.030	0.030	0.030	0.030	0.030
	16	0.026	0.034	0.037	0.053	0.047	0.039
1 x 1	32	0.091	0.029	0.152	0.088	0.023	0.076
	2	0.802	0.797	0.794	0.800	0.800	0.799
	4	0.266	0.268	0.267	0.265	0.270	0.267
	8	0.114	0.116	0.114	0.116	0.115	0.115
	16	0.066	0.066	0.084	0.069	0.122	0.081
2 X 2	32	0.128	0.107	0.146	0.132	0.101	0.123
	2	3.160	3.154	3.150	3.152	3.152	3.154
	4	1.057	1.057	1.057	1.056	1.053	1.056
	8	0.451	0.451	0.451	0.451	0.451	0.451
	16	0.250	0.307	0.255	0.266	0.246	0.265
4 X 4	32	7.204	8.821	9.534	7.793	8.247	8.320
	2	12.592	12.589	12.605	12.604	12.591	12.596
	4	4.211	4.210	4.206	4.205	4.206	4.208
	8	1.801	1.801	1.801	1.805	1.803	1.802
	16	0.938	0.938	0.952	0.951	0.936	0.943
8 X 8	32	20.384	22.522	24.054	23.257	26.966	23.436
	2	50.348	50.363	50.369	50.368	50.337	50.357
	4	16.817	16.812	16.820	16.814	16.812	16.815
	8	7.200	7.198	7.200	7.197	7.204	7.200
	16	3.637	3.638	3.634	3.659	3.661	3.646
16 X 16	32	84.925	77.049	79.076	71.398	83.080	79.106
	2	201.327	201.376	201.407	201.596	201.296	201.400
	4	67.211	67.264	67.266	67.217	67.199	67.231
	8	28.799	28.820	28.792	28.780	28.782	28.795
	16	14.731	14.701	14.745	14.913	14.913	14.800
32 x 32	32	190.253	188.473	185.816	192.340	190.585	189.493

Appendix-3

PIMFuncs.h and PIMFunc.cpp source code provided by Dr. Harris:

```
#ifndef __PIMFUNCS_H__
#define __PIMFUNCS_H__

bool pim_write_black_and_white(const char * const fileName,
                               const int width,
                               const int height,
                               const unsigned char * pixels);
bool pim_write_black_and_white(const char * const fileName,
                               const int width,
                               const int height,
                               const unsigned char ** pixels);
bool pim_write_color(const char * const fileName,
                     const int width,
                     const int height,
                     const unsigned char * pixels);
bool pim_write_color(const char * const fileName,
                     const int width,
                     const int height,
                     const unsigned char ** pixels);
bool pim_write_color(const char * const fileName,
                     const int width,
                     const int height,
                     const unsigned char * red,
                     const unsigned char * green,
                     const unsigned char * blue);
bool pim_write_color(const char * const fileName,
                     const int width,
                     const int height,
                     const unsigned char ** red,
                     const unsigned char ** green,
                     const unsigned char ** blue);
```

```
#endif
#include <cstdio>
#include <cstring>

#ifdef _WIN32
  #define WRITE_FLAGS "wb"
#else
  #define WRITE_FLAGS "w"
#endif

/* Image file "Magic Numbers"
 * P1 = ascii, .pbm
 * P2 = ascii, .pgm
 * P3 = ascii, .ppm
 * P4 = black and white (0-1), .pbm
 * P5 = gray scale (0-255), .pgm
 * P6 = RGB (0-255), .ppm
 */
bool pim_write_black_and_white(const char * const fileName,
                               const int width,
                               const int height,
                               const unsigned char * pixels)
{
  FILE * fp = fopen(fileName, WRITE_FLAGS);

  if (!fp) return false;
  fprintf(fp, "P5\n%i %i 255\n", width, height);
  fwrite(pixels, width * height, 1, fp);
  fclose(fp);

  return true;
}
bool pim_write_black_and_white(const char * const fileName,
                               const int width,
                               const int height,
                               const unsigned char ** pixels)
{
  int i;
  bool ret;
  unsigned char * t = new unsigned char[width * height];

  for (i = 0; i < height; ++i) memcpy(t + width * i, pixels[i], width);
  ret = pim_write_black_and_white(fileName, width, height, t);
  delete [] t;
  return ret;
}
bool pim_write_color(const char * const fileName,
                     const int width,
                     const int height,
                     const unsigned char * pixels)
{
  FILE * fp = fopen(fileName, WRITE_FLAGS);

  if (!fp) return false;
  fprintf(fp, "P6\n%i %i 255\n", width, height);
  fwrite(pixels, width * height * 3, 1, fp);
  fclose(fp);

  return true;
}
bool pim_write_color(const char * const fileName,
                     const int width,
                     const int height,
                     const unsigned char * const * pixels)
{
  int i;
  bool ret;
  unsigned char * t = new unsigned char[width * height * 3];

  for (i = 0; i < height; ++i) memcpy(t + width * i * 3, pixels[i], width * 3);
  ret = pim_write_color(fileName, width, height, t);
  delete [] t;
  return ret;
}
bool pim_write_color(const char * const fileName,
                     const int width,
                     const int height,
                     const unsigned char * red,
                    const unsigned char * green,
                    const unsigned char * blue)
{
  int i, j;
  bool ret;
  unsigned char * p, * t = new unsigned char[width * height * 3];

  p = t;
  for (i = 0; i < height; ++i)
  {
    for (j = 0; j < width; ++j)
    {
      *(p++) = *(red++);
      *(p++) = *(green++);
      *(p++) = *(blue++);
    }
  }
  ret = pim_write_color(fileName, width, height, t);
  delete [] t;
  return ret;
}
bool pim_write_color(const char * const fileName,
                    const int width,
                    const int height,
                    const unsigned char ** red,
                    const unsigned char ** green,
                    const unsigned char ** blue)
{
  int i, j;
  bool ret;
  const unsigned char * r, * g, * b;
  unsigned char * p, * t = new unsigned char[width * height * 3];

  p = t;
  for (i = 0; i < height; ++i)
  {
    r = red[i];
    g = green[i];
    b = blue[i];
    for (j = 0; j < width; ++j)
    {
      *(p++) = *(r++);
      *(p++) = *(g++);
      *(p++) = *(b++);
    }
  }
  ret = pim_write_color(fileName, width, height, t);
  delete [] t;
  return ret;
}
```