CS 415 Parallel Computing

Mandelbrot Sets

March 2, 2012

Erin Keith

Introduction:

This project focused on computing Mandelbrot sets and outputting an image which illustrates values contained in that set. A Mandelbrot set is a set of complex numbers, such that

$$M = \left\{ c \in \mathbb{C} \mid \lim_{n \to \infty} Z_n \neq \infty \right\}$$

where:

$$Z_0 = c$$
$$Z_{n+1} = Z_n^2 + c$$

Since the complex set can be represented in a two dimensional Cartesian coordinate system, and we can limit the number of iterations to 256, the form in Figure 1 will be displayed.

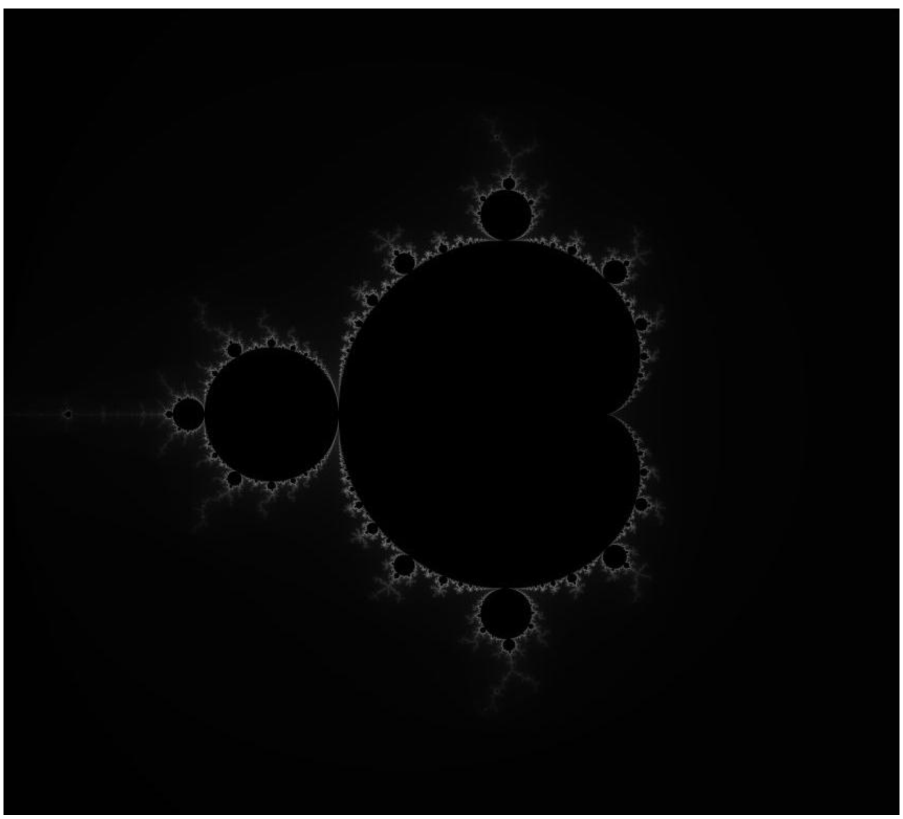

Figure 1: Mandelbrot Fractal

Procedure:

Sequential Program

To be able to determine speedup time, it was imperative to first create the algorithm, as provided in the book, sequentially. A struct, containing real and imaginary variables, was helpful for handling complex numbers. One complex struct handled the Cartesian coordinates as a representation of the set of complex numbers between -2-2i, and 2+2i, scaled by an image factor. Another complex struct started at zero and throughout the iterations the following formulas were computed:

$$z_{real} = z_{real}^{2} - z_{imag}^{2} + c_{real}$$

$$z_{imag} = 2z_{real}z_{imag} + c_{imag}$$

as long as the magnitude of z was less than 2 and the iterations were less than 256. The number of iterations was then stored as the pixel value for the image at that coordinate.

To determine the amount of time it took to calculate images of different sizes, each image size runtime was averaged over 10 executions on the grid, and the runtime values were gathered in Table 1

Table 1: Sequential Time over Pixels

Pixels Dimensions	Time (sec)
500	0.126
1000	0.253
1500	0.485
2000	0.798
2500	1.226
3000	1.793
3500	2.394
4000	3.052
4500	3.882
5000	4.817
5500	5.801
6000	6.828
6500	8.127
7000	10.15
7500	12.126
8000	16.479
8500	18.824
9000	22.705
9500	26.058
10000	35.256

The data gathered in Table 1 is represented by Figure 2, which clearly shows a curve resembling an exponential growth trend.

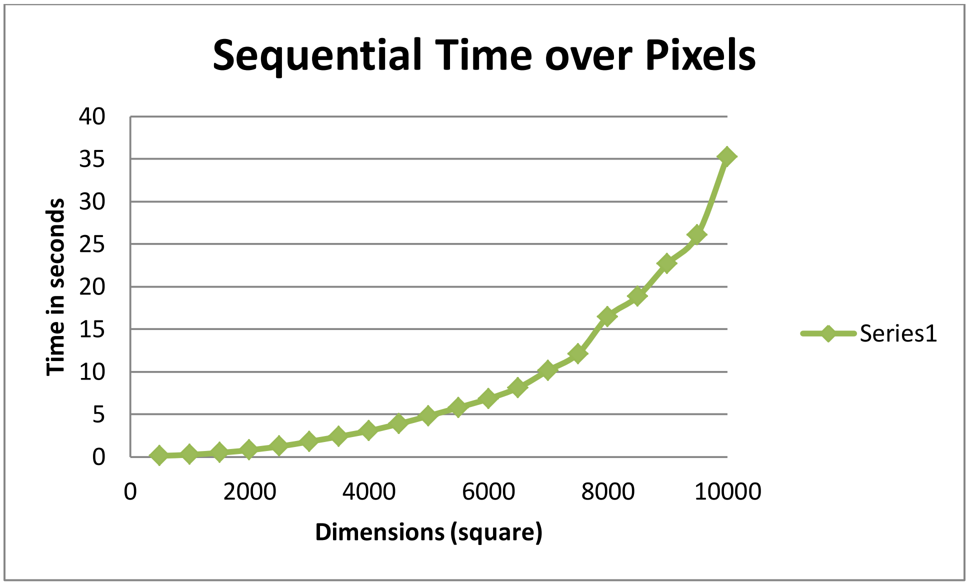

Figure 2: Sequential Runtime for Different Image Sizes

Part II: Static Parallel

The next part of the assignment was to design a parallel algorithm, assigning the slave processes a fixed range of rows for which to compute the values. Since the dimensions of my image were declared as global constants, I determined that it would be redundant to send the slaves their ranges. I made the slaves "intelligent" and had them compute their range to save communication time. The master then determined the appropriate range based on the ID of the process whose message it received. Runtimes for each image dimension was run on a range of processors, from 2 to 20 in two processor increments; the data is displayed in Table 2.

Table 2: Static Parallel Runtimes over Pixels and Processors

Pixels	2	4	6	8	10	12	14	16	18	20
500	0.073	0.064	0.049	0.041	0.038	0.036	0.036	0.036	0.033	0.029
1000	0.202	0.167	0.121	0.125	0.106	0.085	0.074	0.067	0.100	0.191
1500	0.577	0.501	0.367	0.280	0.230	0.189	0.256	0.304	0.271	0.241
2000	1.027	0.890	0.652	0.500	0.408	0.337	0.459	0.538	0.523	0.408
2500	1.610	1.399	1.015	0.779	0.634	0.525	0.455	0.329	0.281	0.284
3000	2.323	2.005	1.458	1.119	0.913	0.751	0.648	0.470	0.616	0.826
3500	3.159	2.5	1.985	1.518	1.241	1.025	1.296	1.265	1.294	1.221
4000	4.110	3.0	2.5	1.987	1.629	1.335	1.530	1.772	1.746	1.442
4500	5.205	4.0	3.0	2.512	2.040	1.692	1.564	1.615	1.515	1.294
5000	6.975	5.342	3.800	3.0	2.886	2.284	1.545	1.282	1.172	1.434
5500	6.091	5.041	3.366	3.0	2.5	2.051	2.121	1.5	1.4	2.0

6000	7.289	6.000	4.346	3.365	2.725	2.5	1.936	1.761	1.878	2.228
6500	8.539	7.043	5.097	3.952	3.202	3.0	2.5	2.110	2.198	2.458
7000	9.905	8.296	5.904	4.574	3.706	3.5	3.0	2.5	2.439	2.649
7500	12.098	10.617	7.707	7.0	6.0	5.0	4.0	3.0	3.0	3.0

Since this was the second time running the data, with a much larger data set, I found an error in my program, which creates a segmentation fault for predictable conditions. Unfortunately, I did not have time to address this problem, so only a range of 500 to 7500 square pixel images was used. The missing data was filled in with round numbers (indicated by two decimal places) to fill in gaps in the graph. Figure 3 displays the graphical representation of the runtime data from Table 2.

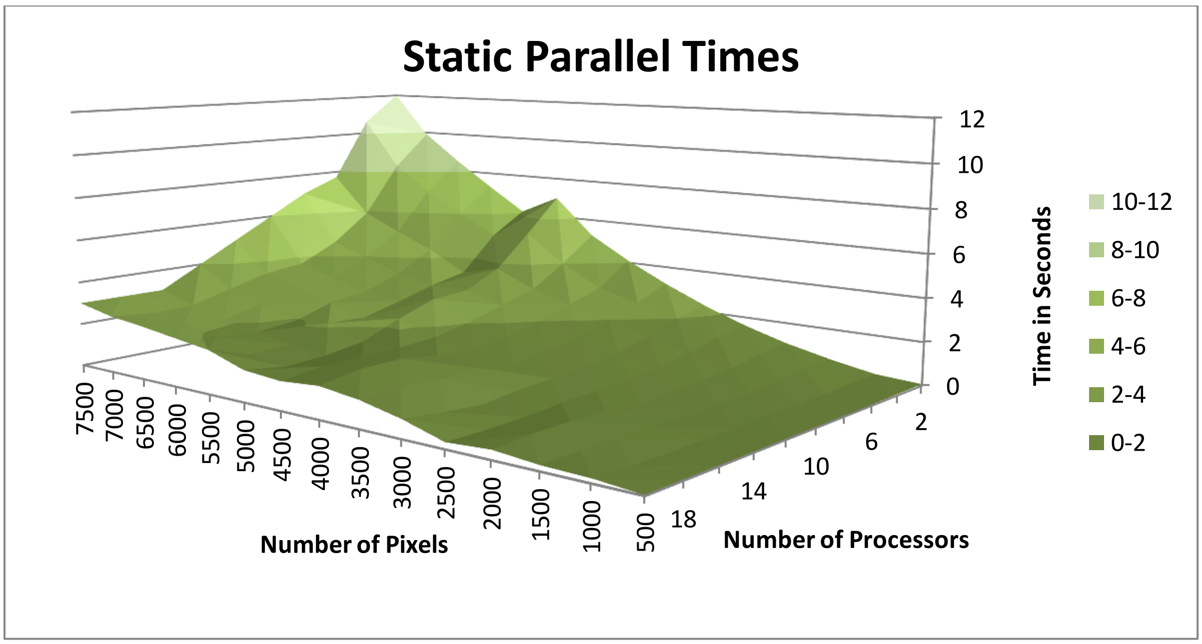

Figure 3: Static Parallel Runtimes

Part III: Dynamic Parallel

Finally, I attacked the problem of dynamically assigning rows from a workpool to processors as they became available. Instead of using the books suggestion to send the array of pixel values and the row number, I used a lookup table to keep track of the rows being processed based on processor ID, which helped cut down on communication time.

The runtimes created by the dynamic parallel program are contained in Table 3.

Table 3: Dynamic Parallel Runtimes over Pixels and Processors

Pixels	2	4	6	8	10	12	14	16	18	20
500	0.067	0.046	0.038	0.040	0.031	0.028	0.028	0.027	0.104	0.103
1000	0.253	0.123	0.099	0.084	0.074	0.069	0.064	0.062	0.139	0.137

1500	0.570	0.259	0.178	0.151	0.147	0.138	0.125	0.129	0.176	0.194
2000	1.015	0.453	0.359	0.238	0.207	0.182	0.166	0.188	0.264	0.241
2500	2.372	1.289	0.839	0.567	0.415	0.339	0.345	0.324	0.380	0.406
3000	3.171	1.280	0.683	0.544	0.471	0.478	0.438	0.434	0.500	0.487
3500	3.103	1.263	0.835	0.406	0.369	0.562	0.544	0.526	0.540	0.580
4000	4.091	2.029	1.498	1.226	1.056	0.958	0.666	0.651	0.715	0.697
4500	5.121	2.059	1.362	1.155	0.971	0.941	0.888	0.839	0.834	0.797
5000	4.940	2.735	1.929	1.391	1.221	1.099	0.987	0.979	1.000	0.959
5500	5.894	2.721	1.848	2.282	1.704	1.891	1.521	1.433	1.518	1.625
6000	8.111	3.041	2.318	1.834	1.691	1.585	1.419	1.235	1.208	1.128
6500	10.622	4.199	2.955	2.259	1.874	1.617	1.497	1.391	1.845	1.892
7000	12.398	4.746	3.290	2.268	2.498	2.242	2.064	2.007	1.520	1.295
7500	14.245	5.620	3.714	3.170	2.675	2.064	2.815	2.565	2.563	2.628
8000	16.210	6.336	4.180	3.951	3.669	3.280	3.004	3.049	3.022	2.979
8500	16.248	6.931	4.783	4.018	3.681	3.138	3.048	2.826	2.841	2.928
9000	24.080	8.723	5.989	5.160	4.492	3.949	3.820	3.594	3.412	3.574
9500	22.891	8.911	5.910	5.266	4.669	4.702	4.373	4.442	4.479	4.362
10000	25.159	8.910	5.888	4.989	4.449	4.112	3.998	3.806	3.928	3.888

Figure 4 illustrates the runtime of the dynamic parallel program over the same ranges as the static parallel program.

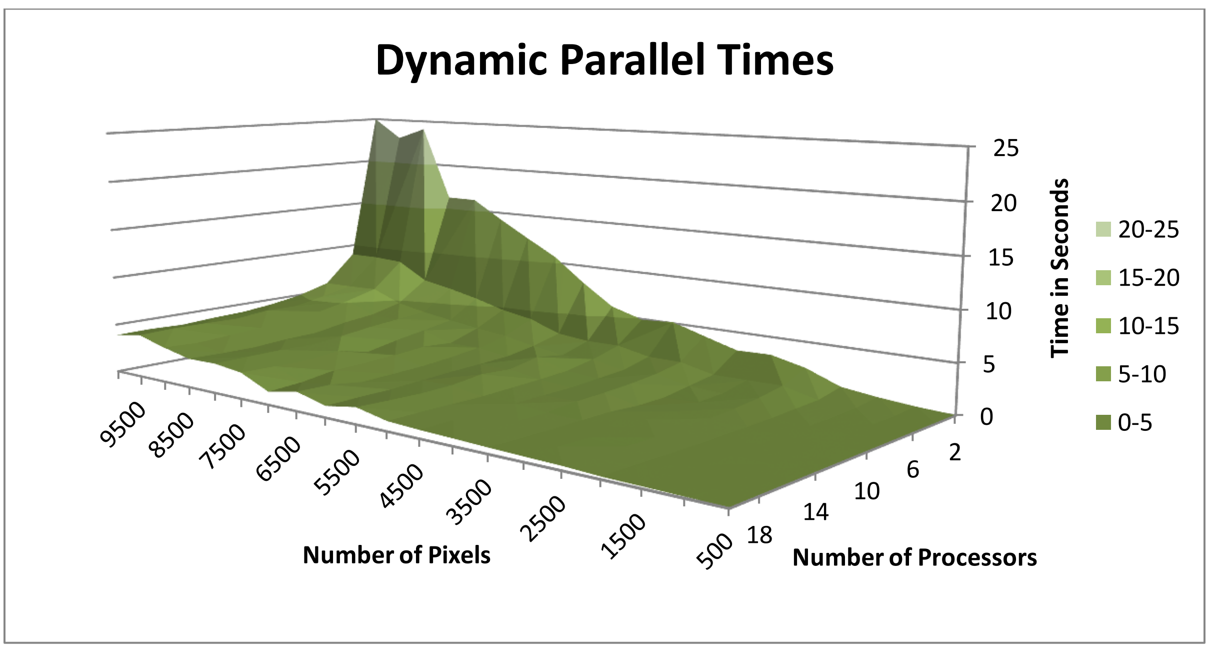

Figure 4: Dynamic Parallel Runtimes

Analysis:

The curve generated by the sequential program clearly demonstrates some sort of threshold around the 8000 x 8000 pixel images. Otherwise the curve shows the expected linear increase.

The statically parallel implementation of the program shows a shallower, more linear curve than the dynamic, as I'm dividing the work by processor, directly. It is important to note that the dynamic program's overall runtimes were less than the static program, except for the runtimes for two processors, which were actually faster in the static program, since there was significantly less communication.

The overall runtimes of the dynamically implemented parallel program show a more sharply decreasing trend for increasing processors than the static program. Using two processors for the parallel programs simulates the sequential environment. There is a steep drop from two to four processors, as this is actually the first instance of multiple processors being used. If you remove the two processor data, a trough is evident, on the graph. If I had had enough time, I would have analyzed these minimums to try to determine some sort of relationship, which would lead to an optimal percentage of work to dole out to each processor.

There is also significant speedup of parallel over sequential algorithms, as given by the following equation:

$$\text{Speedup} = \frac{t_{\text{sequential}}}{t_{\text{dynamic parallel}}}$$

The predictably increasing trend, with a maximum speedup of roughly 9, is contained in Table 4.

Table 4: Speedup of Dynamic Parallel Runtimes over Sequential

Pixels	2	4	6	8	10	12	14	16	18	20
500	1.867	2.739	3.290	3.179	4.090	4.436	4.572	4.716	1.209	1.223
1000	0.998	2.065	2.546	3.028	3.403	3.641	3.934	4.100	1.818	1.848
1500	0.851	1.876	2.723	3.215	3.298	3.511	3.888	3.753	2.755	2.496
2000	0.786	1.763	2.226	3.351	3.855	4.391	4.811	4.254	3.023	3.306
2500	0.517	0.951	1.462	2.164	2.958	3.619	3.553	3.778	3.230	3.016
3000	0.566	1.401	2.626	3.295	3.805	3.755	4.092	4.135	3.585	3.679
3500	0.772	1.895	2.868	5.890	6.492	4.259	4.398	4.548	4.437	4.129
4000	0.746	1.504	2.037	2.489	2.891	3.187	4.582	4.692	4.269	4.377
4500	0.758	1.885	2.851	3.361	4.000	4.124	4.371	4.629	4.656	4.870
5000	0.975	1.761	2.497	3.464	3.945	4.384	4.883	4.918	4.816	5.025
5500	0.984	2.132	3.138	2.542	3.404	3.068	3.815	4.048	3.822	3.570
6000	0.842	2.245	2.945	3.723	4.038	4.309	4.812	5.527	5.651	6.055
6500	0.765	1.935	2.750	3.597	4.338	5.026	5.430	5.843	4.404	4.295
7000	0.819	2.139	3.085	4.475	4.064	4.526	4.918	5.058	6.677	7.839
7500	0.851	2.158	3.265	3.825	4.534	5.875	4.308	4.727	4.730	4.614
8000	1.017	2.601	3.943	4.171	4.492	5.024	5.486	5.405	5.452	5.532
8500	1.159	2.716	3.935	4.685	5.114	5.998	6.177	6.662	6.627	6.430

9000	0.943	2.603	3.791	4.400	5.055	5.750	5.943	6.317	6.654	6.354
9500	1.138	2.924	4.409	4.949	5.581	5.542	5.959	5.867	5.818	5.974
10000	1.401	3.957	5.988	7.067	7.924	8.574	8.819	9.262	8.976	9.068

Speedup for each image size and processor is illustrated by Figure 5. Then general trend is increased speedup as image size and the number of processors increase, which is to be expected. The surprise is that the processor number with greatest speedup is 16, for which I cannot think of an explanation (except that maybe the grid was running extra fast).

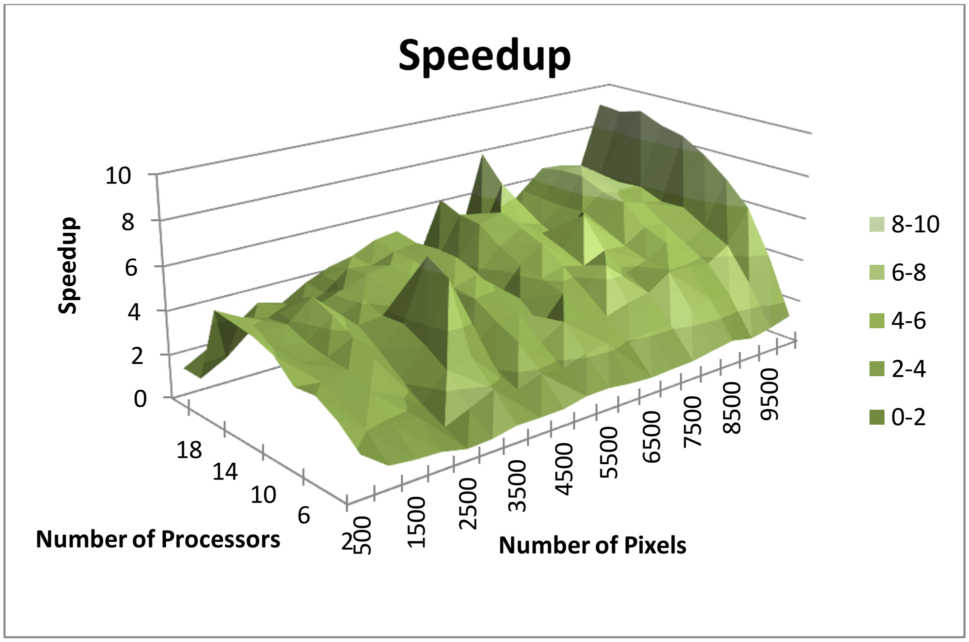

Figure 5: Speedup of Dynamic Parallel over Sequential

Finally, and most interestingly, is the efficiency of each processor in speeding up processing of different image sizes, shown in Table 5. Here we have:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Processors}}$$

Table 5: Efficiency of Dynamic Parallel Runtimes

Pixels	2	4	6	8	10	12	14	16	18	20
500	0.934	0.685	0.548	0.397	0.409	0.370	0.327	0.295	0.067	0.061
1000	0.499	0.516	0.424	0.379	0.340	0.303	0.281	0.256	0.101	0.092
1500	0.426	0.469	0.454	0.402	0.330	0.293	0.278	0.235	0.153	0.125
2000	0.393	0.441	0.371	0.419	0.386	0.366	0.344	0.266	0.168	0.165
2500	0.258	0.238	0.244	0.270	0.296	0.302	0.254	0.236	0.179	0.151
3000	0.283	0.350	0.438	0.412	0.380	0.313	0.292	0.258	0.199	0.184
3500	0.386	0.474	0.478	0.736	0.649	0.355	0.314	0.284	0.247	0.206
4000	0.373	0.376	0.340	0.311	0.289	0.266	0.327	0.293	0.237	0.219

4500	0.379	0.471	0.475	0.420	0.400	0.344	0.312	0.289	0.259	0.243
5000	0.488	0.440	0.416	0.433	0.395	0.365	0.349	0.307	0.268	0.251
5500	0.492	0.533	0.523	0.318	0.340	0.256	0.272	0.253	0.212	0.179
6000	0.421	0.561	0.491	0.465	0.404	0.359	0.344	0.345	0.314	0.303
6500	0.383	0.484	0.458	0.450	0.434	0.419	0.388	0.365	0.245	0.215
7000	0.409	0.535	0.514	0.559	0.406	0.377	0.351	0.316	0.371	0.392
7500	0.426	0.539	0.544	0.478	0.453	0.490	0.308	0.295	0.263	0.231
8000	0.508	0.650	0.657	0.521	0.449	0.419	0.392	0.338	0.303	0.277
8500	0.579	0.679	0.656	0.586	0.511	0.500	0.441	0.416	0.368	0.322
9000	0.471	0.651	0.632	0.550	0.506	0.479	0.424	0.395	0.370	0.318
9500	0.569	0.731	0.735	0.619	0.558	0.462	0.426	0.367	0.323	0.299
10000	0.701	0.989	0.998	0.883	0.792	0.714	0.630	0.579	0.499	0.453

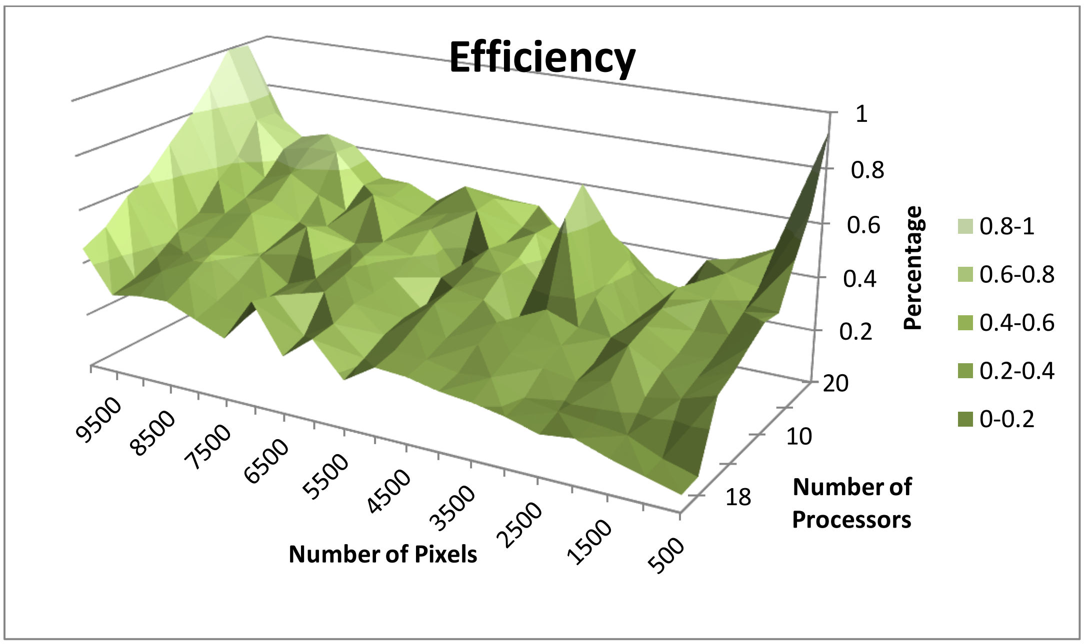

Figure 6: Efficiency of Dynamic Parallel Process

As the analysis of dynamic runtimes led to insight of how to better assign workload to processors, I also think that these efficiency percentages hold the key. I would have liked to spend time finding the conditions for best and worst efficiencies, and to work out a relationship between them.

Conclusion:

My Mandelbrot programs were successful in demonstrating the speedup of a parallel implementation over a sequential one. My implementations also provide a good example of increased efficiency in dynamically parallel programs over statically parallel programs. Although I am especially pleased that my designs reduced communication time significantly, there are still many aspects of this program to analyze to produce better efficiencies and runtimes.