CS361 Assignment 1

Dennis George djg365

April 12 2021

1 Sequential Implementation

The sequential implementation for this assignment was the most straightforward, and required the least debugging for all of the sections. Below is a table outlining my runtimes for all of the benchmarks for part 1 (all times are in seconds).

ShipSquare9000-100	ShipMasts10000-400	Dave5000-1000	Benchmark
4.85	5.9	19.673	77.903

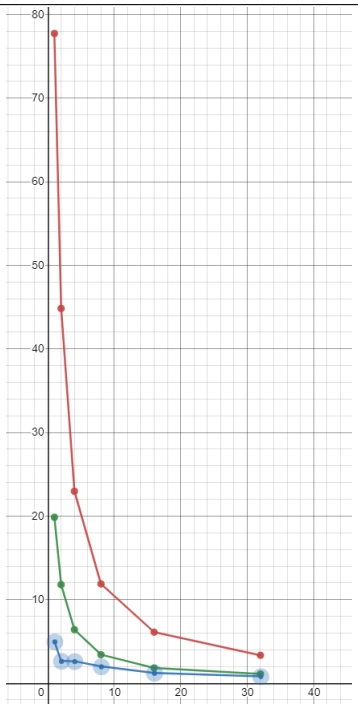
One thing that is interesting to note, is that while I was running the program on TUX, I had to increase the Java max heap size. There were a few executions of my program that were not running to completion without this flag. I did not need to reference any student's code on the wiki. My first implementation was including the amount of time it took to build the .png, and was not competetive with any of the times that were on the wiki. Reading thru Slack, and some of the wiki posts showed me that I should not be timing that portion of the program. We only cared about how long it would take to compute the grid. The point of this assignment was not to improve how fast the image drawing library could run. The benchmark that I created took quite some time to find. I kept looking online to find some good regions, but I wasn't able to find too many useful resources. The benchmark parameters that I posted on the wiki ended up being the benchmark that everybody is using for the second part of this assignment.

2 Parallel Implementation

2.1 Static Strategy

My plan for the static strategy was to build a thread object that was responsible for computing a certain section of the overall grid. In order to keep the division of labor simple, I would divide the grid into $numThreads$ row-chunks. Each row-chunk would be from xlo to xhi and ylo and yhi would be scaled by $size/numThreads$. I found that I was able to test $threadNum$'s of 16 and 32 in addition to the requested counts. Below is a table outlining me runtimes for all of the benchmarks for part 1a (all times are in seconds).

Threads	ShipSquare9000-100	Dave5000-1000	Benchmark
1	4.99	19.885	77.732
2	2.66	11.789	44.832
4	2.63	6.414	22.974
8	2.01	3.419	11.886
16	1.23	1.824	6.11
32	0.82	1.1	3.345



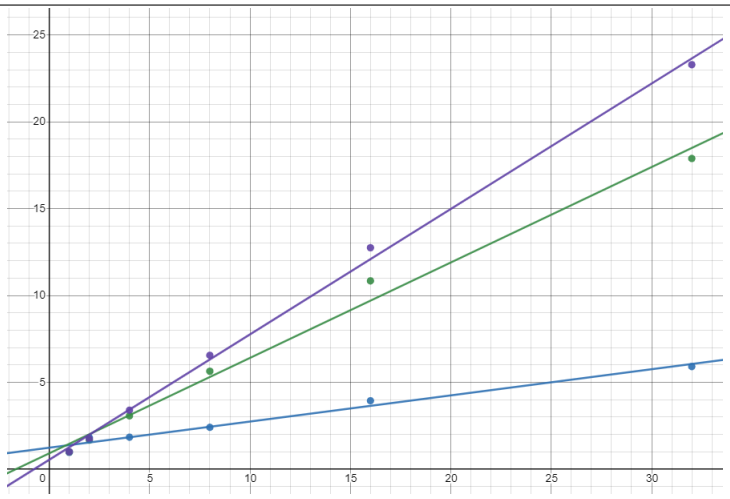
This graph plots the above table, and shows the decrease in runtime as the amount of threads increase. The red line plots the Benchmark, the green line ploys the Dave5000-1000 image, and the blue line plots ShipSquare9000-100. The results are expected: as the number of threads increase, the amount of time the algorithm takes also decreases.

2.1.1 Speedup and Efficiency

Below is a table that shows results for speedup and efficiency for all of the requested benchmarks.

Speedup

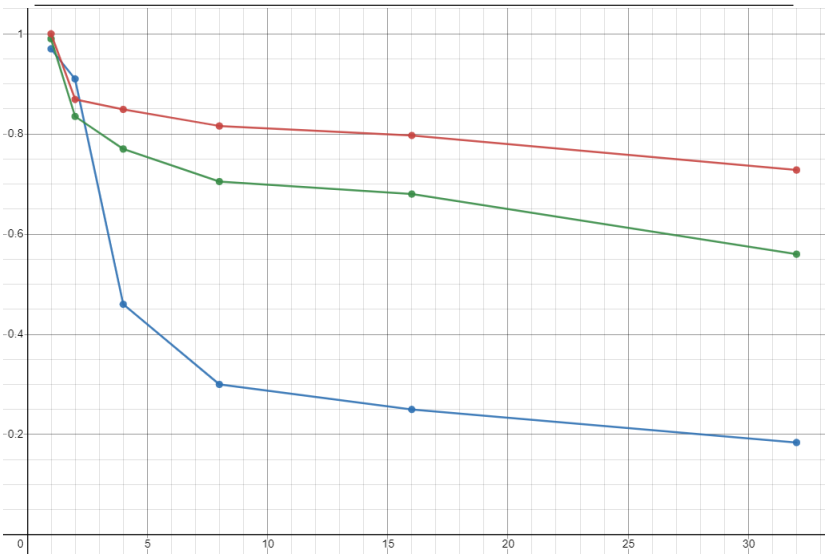
Threads	ShipSquare9000-100	Dave5000-1000	Benchmark
1	0.97	0.995	1.0
2	1.82	1.67	1.738
4	1.84	3.07	3.395
8	2.41	5.64	6.554
16	3.94	10.84	12.75
32	5.91	17.89	23.289



The above graph shows the speedup of the static implementation vs. the amount of threads. It is clear that as the number of threads increased, we would expect the program to speed up.

Efficiency

Threads	ShipSquare9000-100	Dave5000-1000	Benchmark
1	0.97	0.99	1.0
2	0.91	0.835	0.869
4	0.46	0.77	0.849
8	0.3	0.705	0.816
16	0.25	0.68	0.797
32	0.184	0.56	0.728

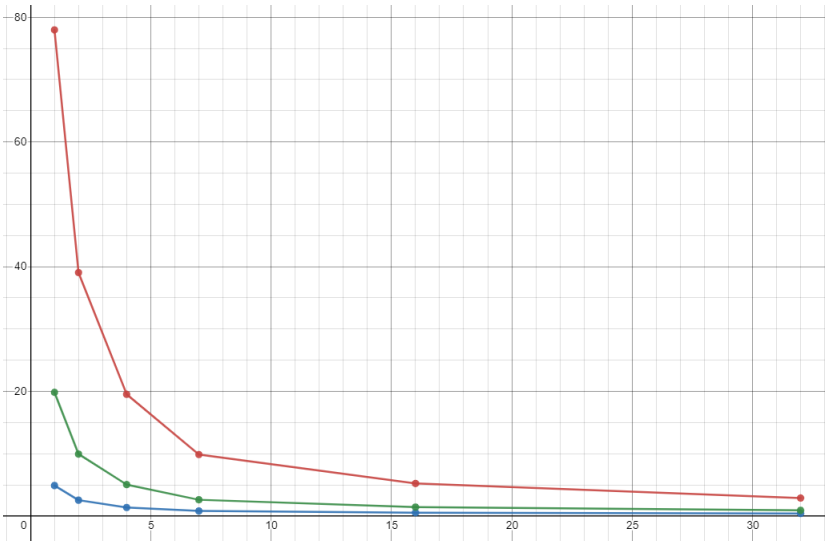


The above graph shows the efficiency of the static implementation vs. the amount of threads. It is clear that as the number of threads increased, the efficiency also decreased.

2.2 Dynamic Strategy

My plan for the dynamic strategy was to implement my own custom thread pool, and allocate work. Each thread would be responsible from pulling the latest piece to work on from the queue, computing that section, and updating the value in the appropriate slot in the grid. If the work was divided up correctly, each thread should be working on a separate chunk of work, and when finished, will look for more work. I chose $k = 100$ for all of the times in this analysis. I found that increasing and reducing k did not have much of an effect on the runtimes of my program on TUX. I found a boilerplate for a custom thread pool online (https://www.javacodemonk.com/implement-custom-thread-pool-in-java-without-executor-framework-ca10e61d) and I credit the design of my ThreadPoolExecutor class to this resource. I also had trouble dividing up the code into k x k regions, so I went to the wiki to see how other students were doing it. I credit my logic for splitting the regions of work to pdd35. I also ran in to a race condition while working on my dynamic implementation. On my computer, while running the code in IntelliJ, everything was working as expected. When I copied the code over to TUX, I was getting blank output images, and the program was terminating very quickly. Upon further inspection, I discovered that the code that starting all the threads could theoretically start before the code that would add chunks of work to the work queue. When the threads start, they look to see if the queue is empty, and if it is not, it begins to compute. In the case that the queue is empty, the thread is finished. Since it is possible that the queue is empty when the thread is started, this now becomes a critical section. In order to fix this, I added a flag that would also be checked, that would notify when the main code is finished adding items to the queue. This stopped the threads from terminating early, and was only noticed because of the issues I was seeing on TUX.

Threads	ShipSquare9000-100	Dave5000-1000	Benchmark
1	4.905	19.851	77.97
2	2.556	9.954	39.061
4	1.383	5.058	19.552
8	0.82	2.624	9.874
16	0.556	1.449	5.231
32	0.432	0.92	2.905

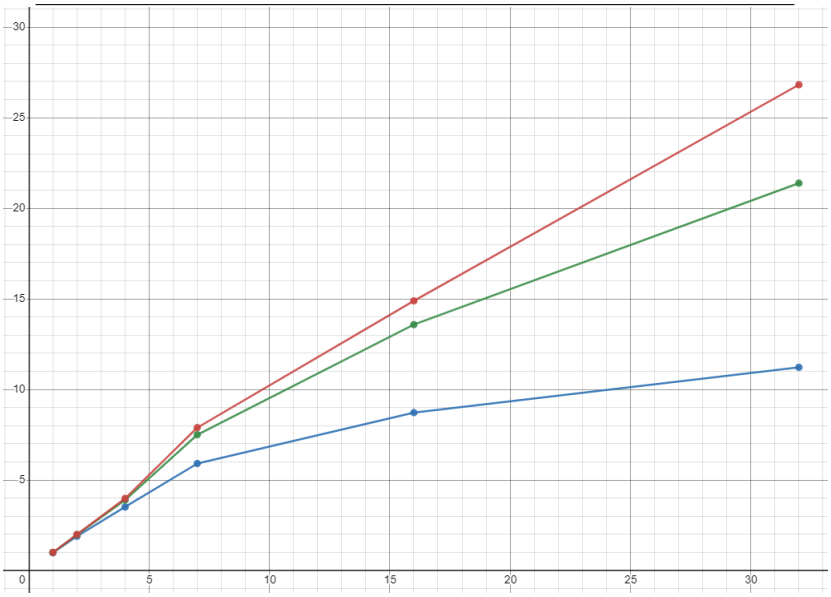


This graph plots the above table, and shows the decrease in runtime as the amount of threads increase. The red line plots the Benchmark, the green line ploys the Dave5000-1000 image, and the blue line plots ShipSquare9000-100. The results are expected: as the number of threads increase, the amount of time the algorithm takes also decreases.

2.2.1 Speedup and Efficiency

Below is a table that shows results for speedup and efficiency for all of the requested benchmarks.
Speedup

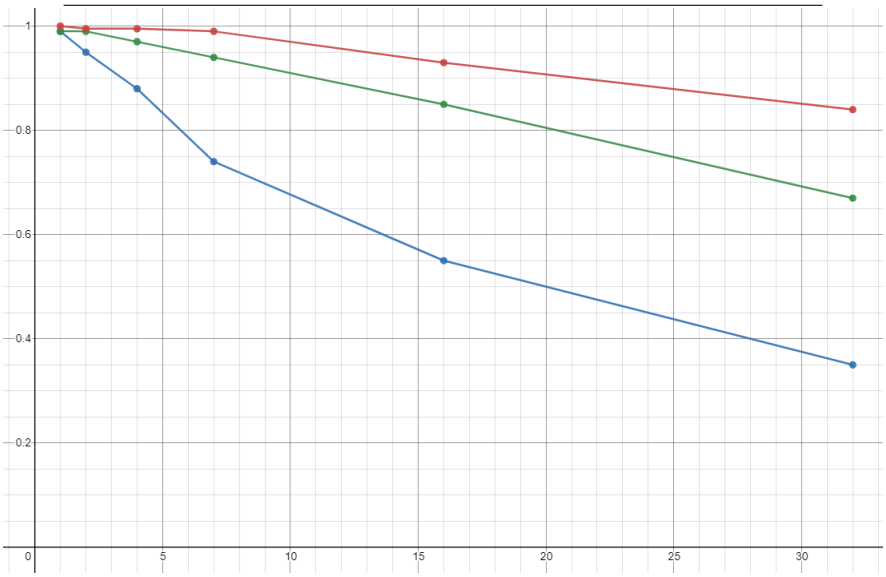
Threads	ShipSquare9000-100	Dave5000-1000	Benchmark
1	0.99	0.99	1.0
2	1.897	1.98	1.99
4	3.51	3.89	3.98
8	5.91	7.5	7.89
16	8.72	13.58	14.89
32	11.22	21.39	26.82



The above graph shows the speedup of the static implementation vs. the amount of threads. It is clear that as the number of threads increased, we would expect the program to speed up.

Efficiency

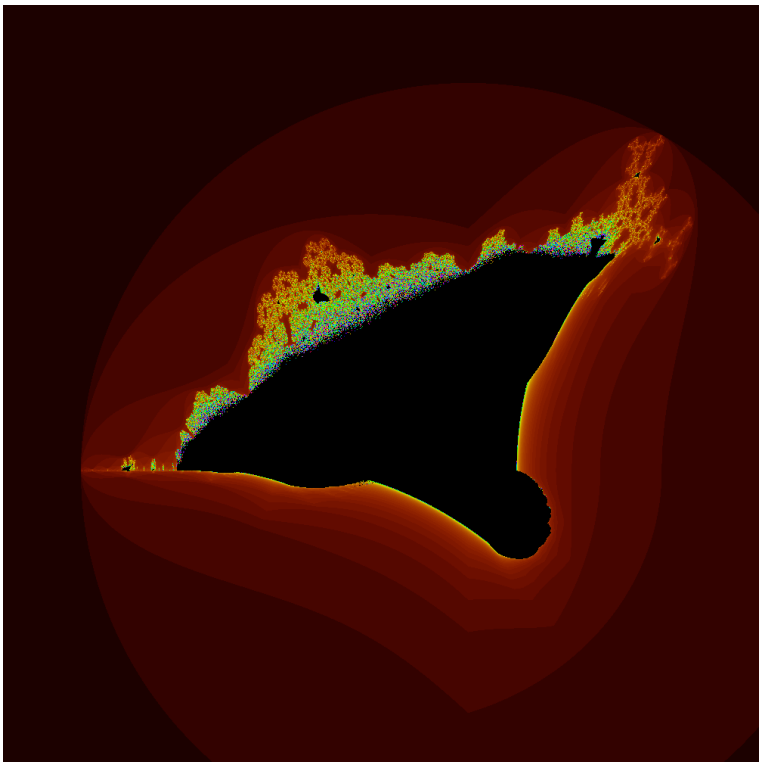
Threads	ShipSquare9000-100	Dave5000-1000	Benchmark
1	0.99	0.99	1.0
2	0.95	0.99	0.995
4	0.88	0.97	0.995
8	0.74	0.94	0.99
16	0.55	0.85	0.93
32	0.35	0.67	0.84



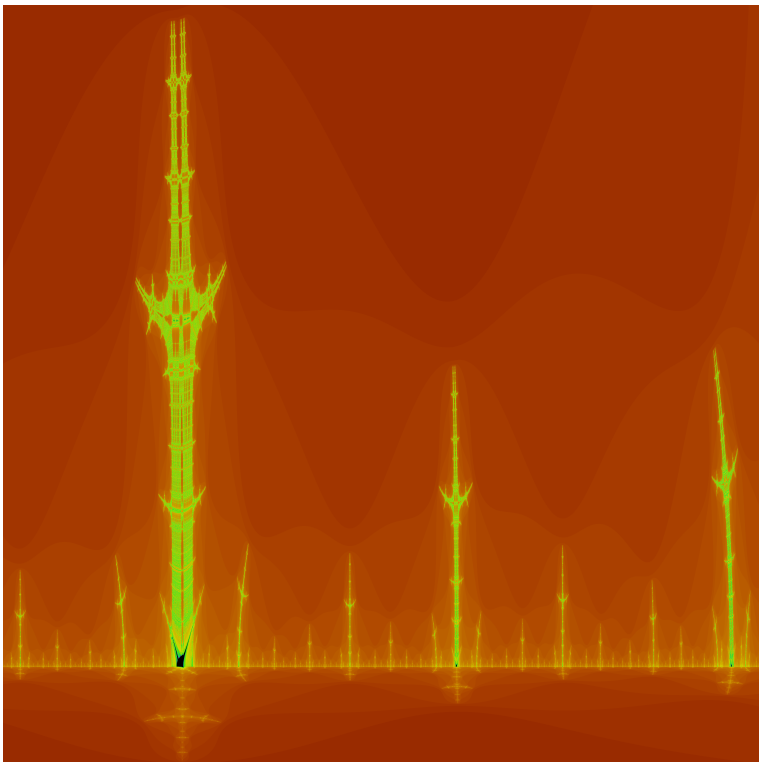
The above graph shows the efficiency of the static implementation vs. the amount of threads. It is clear that as the number of threads increased, the efficiency also decreased.

3 Generated Images

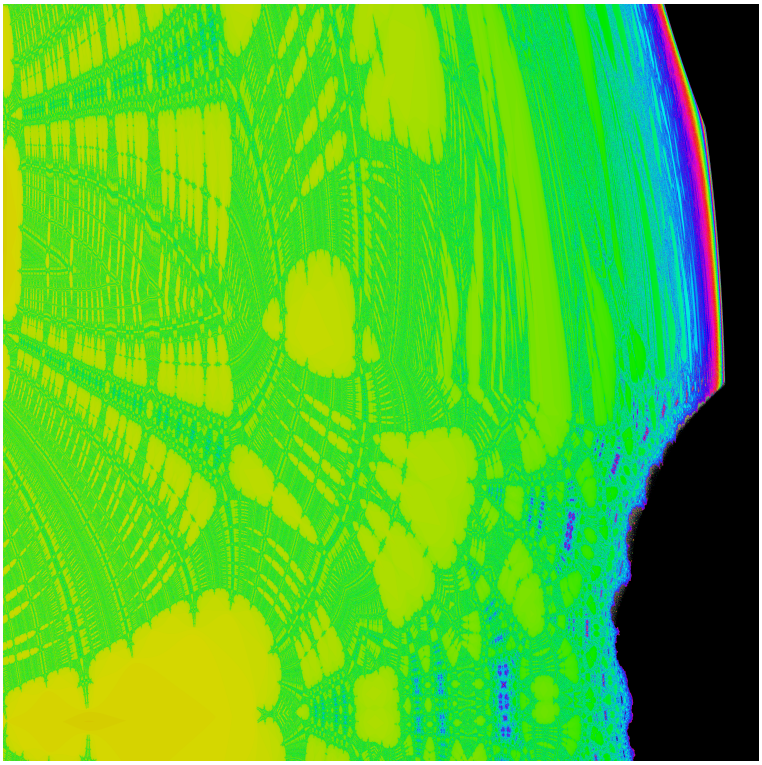
3.1 ShipSquare9000-100



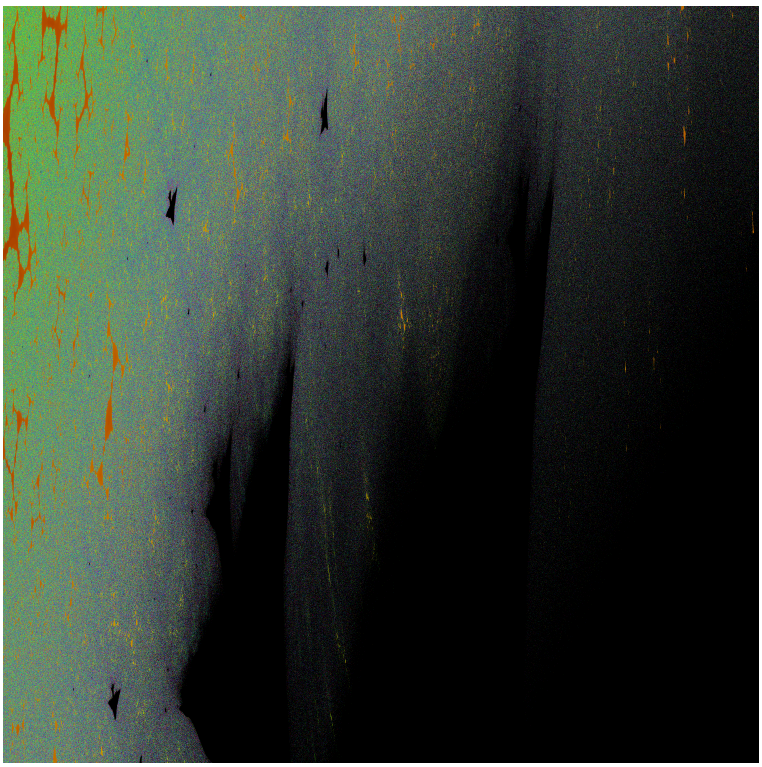
3.2 ShipMasts-10000-400



3.3 Dave-5000-1000



3.4 Benchmark



4 Conclusions

My parallel implementation was definitely much faster than the sequential implementation. This can be proven by looking at the above tables, and confirming that as we added more processors working on the problem, the time it took to compute decreased. When it comes to comparing static vs. dynamic division of labor, I have a few conclusions. The static method seems to definitely work, however when more threads are added, the efficiency starts to take a hit. Looking at the graph for efficiency vs. threads, the right half of the graph starts to flatline, particularly after 8 threads. While the static method may be easier to program, you might not get the longterm benefits of adding concurrency your code. The dynamic method proved to have the shorter runtimes of the two parallelized implementations. The dynamic method also proved to have a tighter spread on the efficacy and speedup as the number of threads increased. Of course, the dynamic method is not perfect, as we also see some convergence around the 8-16 thread mark. The dynamic method for the benchmark seemed to be the most impressive, as it was able to maintain 0.84 efficiency all the way up to 0.84 seconds.