Parallel Mandelbrot Analysis

CSE 30341: Project 3 Parallel Programming

Dylan Greene

24 February 2017

1 Introduction

The Mandelbrot set is a set of complex numbers which result in very interesting fractal shapes when plotted. The purpose of this project was to use parallel programming methods to speed up the generation of these interesting Mandelbrot images. To speed up the generation of individual images, pthreads in C were used to implement thread based parallelism. Additionally, a program to generate sequences of Mandelbrot images slowly zooming in to the specified region. This program was written to use multiple processes to implement parallelism.

The benchmarking which follows was done on a Dell XPS 15 laptop with an Intel Core i7 4710HQ Quad Core Processor with hyperthreading (8 threads total). The operating system used was Ubuntu 16.04 LTS. The configuration for the chosen custom image region can be obtained with the command:

```
./mandel -x 0.2929859127507 -y 0.6117848324958 -m 10000
         -s 0.00000000005 -H 1024 -W 1024
```

The image sequence is generated form the mandelbrotmovie executable. The parameters for mandelbrotmovie are defined at the top of the mandelbrotmovie.c file and match the parameters above. It can be run with the command:

```
./mandelmovie [NUMBER OF PROCESSES]
```

The number of processes is simply passed as a command line argument and is optional. That will generate a sequence of 50 images named mandelN.bmp, where N is the frame number.

Additionally, the benchmarking was done using a simple shell script and the time command. For Configuration A and Configuration B, the benchmarking was performed as follows, where the 'N' following '-n' at the end was the number of threads. Configuration A was run as:

```
time ./mandel -x -.5 -y .5 -s 1 -m 2000 -n N
```

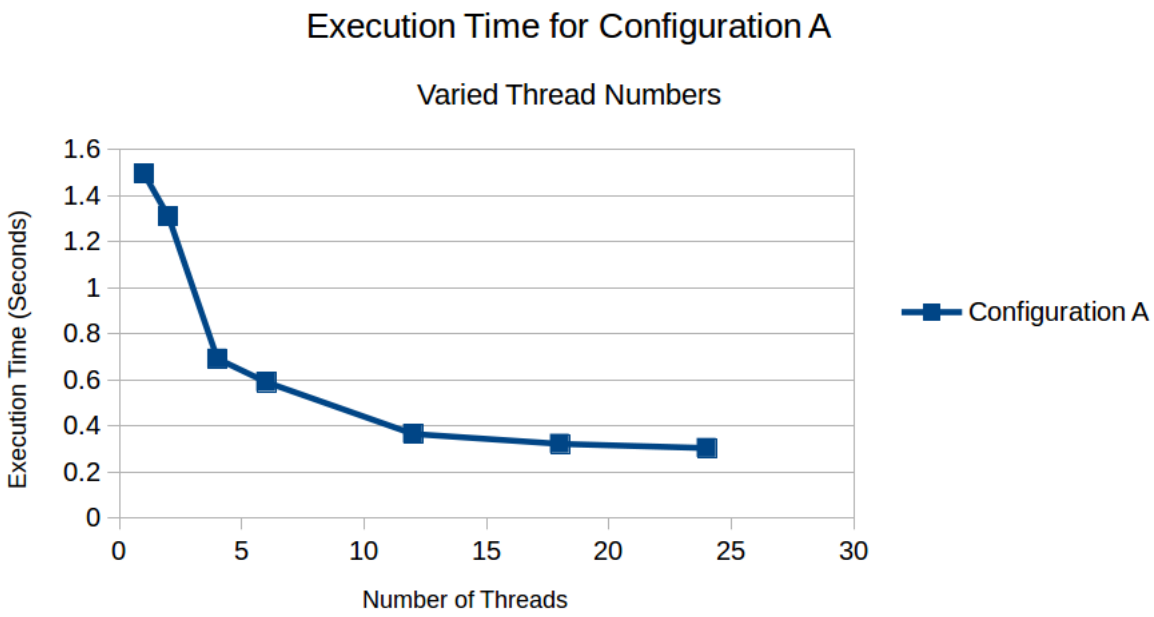
Configuration B was run as:

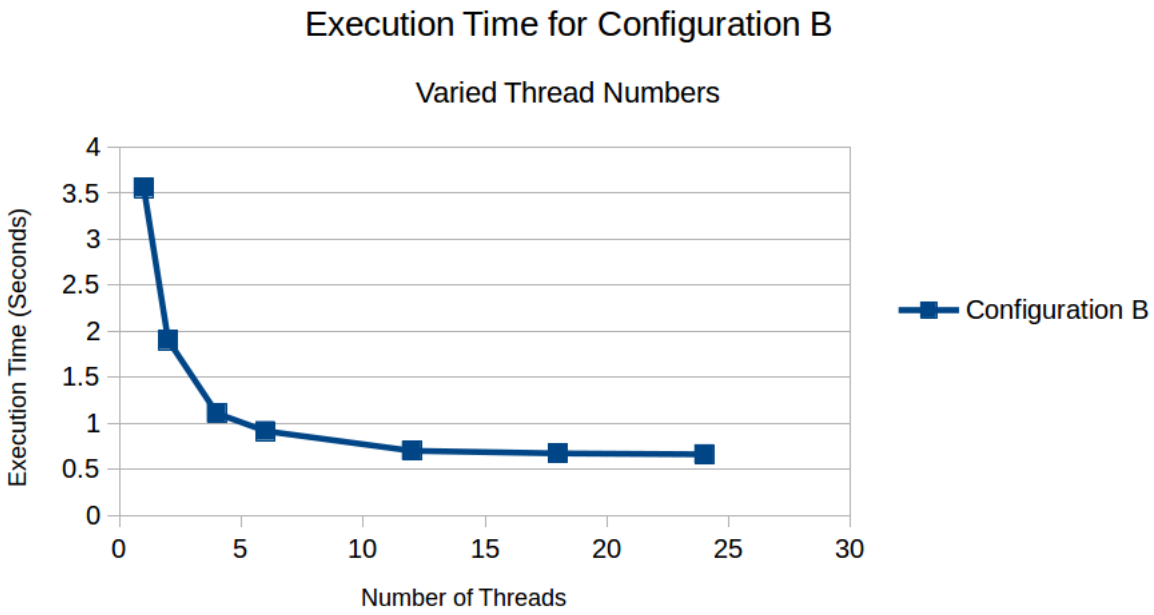
```
time ./mandel -x 0.2869325 -y 0.0142905 -s .000001 -W 1024
              -H 1024 -m 1000 -n N
```

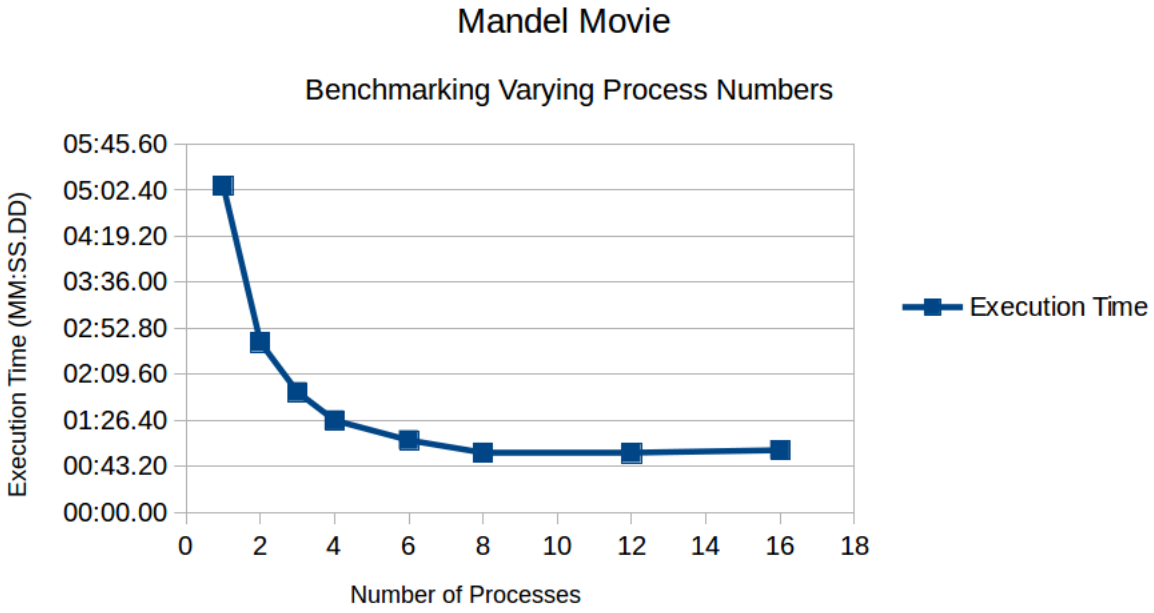
The real time was then recorded from the output. Trials were run 5 times for each value of N, and the smallest time was used in the results section below. Additionally, these same benchmarking procedures were followed for the custom image region described above. The graph from this data is also included.

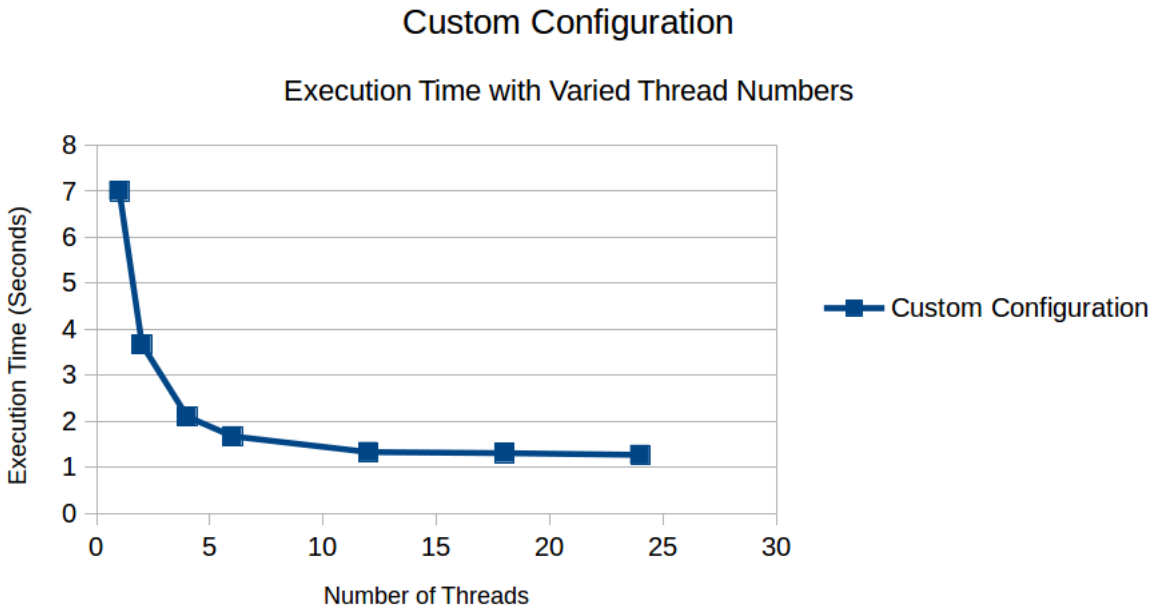
In benchmarking the mandelmovie, the methodology is slightly different. For these benchmarks, the test was only run once each since each takes considerably longer. The final image in the sequence is the custom image described above. As stated, the benchmarking of that one image being generated is also included.

2 Results




Execution Time for Configuration B
Varied Thread Numbers
Execution Time (Seconds)
4
3.5
3
2.5
2
1.5
1
0.5
0
0
5
10
15
20
25
30
Number of Threads
Configuration B


Mandel Movie
Benchmarking Varying Process Numbers
Execution Time (MM:SS.DD)
05:45.60
05:02.40
04:19.20
03:36.00
02:52.80
02:09.60
01:26.40
00:43.20
00:00.00
0
2
4
6
8
10
12
14
16
18
Number of Processes
Execution Time



3 Discussion

The results from these benchmarks were very interesting. The overall trend of each graph is very easy to distinguish. It is clear that both threading and using multiple processes suffer from diminishing returns. Still, there is clearly a large performance increase across the board when using parallelism.

To first analyze the mandel program with threading itself, we can see that the two curves A and B have different shapes. Curve A does not improve as much as B does from the threading as B is improving at a much faster rate with each thread. This is very noticeable in the decrease in time from 1 thread to 2 threads for each curve. Curve B has a drastic improvement, while A has a rather small one. This can perhaps be explained by the region and parameters being used.

Specifically, it is of note that B is a much more computationally expensive region to generate than A. Note that it is far more zoomed in. Thus, the portion of the code spent computing B rather than just setting up or finishing is much higher for B. Further, that is the portion of the code that is affected by threading, so it makes sense that B would be more affected than A by adding a thread. This aside, the performance improvements from threading are very significant in each. By looking at the graph, it can be seen that diminishing returns hit hard after 12

processes are used. Therefore, It would be best to use 12 processes since creating many more processes is just unnecessary as the performance improvements are minuscule and at that point more and more system resources are being used to create threads which do not really help.

Regarding the mandelmovie execution time with varying number of processes, it can be seen how the execution time decreases up until the number of cores is reached. In the case of the test machine, there were 4 cores, but hyperthreading means that there are actually 8 cores seen by the operating system. Therefore, the execution time continues to decrease until 8 processes are created. After 8, the execution time actually increases slightly because at that point, no more processes are actually being worked on by the CPU concurrently. Further, the increase can be explained by the fact that actually creating and forking new processes uses substantial system resources and takes some time.

4 Appendix

Below is the raw data used in the graphs:

Configuration A with varying number of threads.

Number of Threads	Configuration A Execution Time (s)
1	1.495
2	1.309
4	0.691
6	0.589
12	0.364
18	0.321
24	0.303

Configuration B with varying number of threads.

Number of Threads	Configuration B Execution Time (s)
1	3.558
2	1.903
4	1.109
6	0.917
12	0.703
18	0.675
24	0.665

Mandel Movie with varying number of processes.

Number of Processes	Mandel Movie Execution Time (MM:SS.DD)
1	05:06.65
2	02:39.60
3	01:52.97
4	01:26.46
6	01:07.82
8	00:56.14
12	00:55.64
16	00:58.51

Custom image region and configuration with varying number of threads.

Number of Threads	Custom Configuration Execution Time (s)
1	7.001
2	3.671
4	2.104
6	1.678
12	1.335
18	1.3127
24	1.275