Lab Report: Project Three

Multi-processing and multi-threading

Author: Kat Herring

1 Purpose

This report details the effects of multi-processing and multi-threading on execution time.

2 Environment

Specifications of machine used for testing:

- Windows 10 Home v. 1607
- Intel Core i5- 2450M CPU @ 2.5 GHz
- 8 GB RAM
- 64-bit OS

Mandelmovie runs with the underlying call to mandel using the following arguments:

```
./mandel -x -.09999 -y -.835599 -s <zoom> -m 4000 -W 875 -H 875 -o <filename>
```

Where <zoom> varies between 2 and .000001, and <filename> is mandel<n>.bmp (n defined between 0-50 by mandelmovie). This call takes between 5-15 seconds to complete, with time increasing at larger scales.

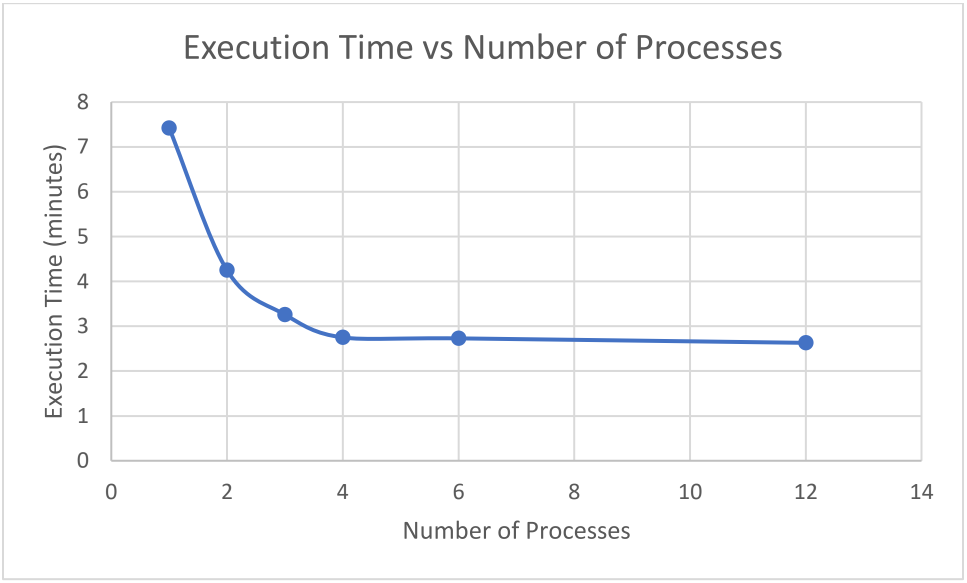
3 Mandelmovie

Mandelmovie generates 50 images scaled from 2-.000001. These images can then be compiled into a movie used ffmpeg.

3.1 Data

Number of Processes	Execution Time	Execution Time (Dec)
1	7m25s	7.42
2	4m15s	4.25
3	3m16s	3.26
4	2m45s	2.75
6	2m44s	2.73
12	2m38s	2.63

3.2 Graph



3.3 Analysis

Increasing the number of running processes dramatically speeds up execution time to a certain point (around four processes), after which the OS is likely unable to run any more processes simultaneously due to system constraints.

3.3.1 Optimal Number

Four appears to be the optimal number of processes. Beyond that point, speed gains are minimal, with each additional process only eliminating about a second of execution time.

3.3.2 "Too many" Issues

I did not run into any issues with there being "too many" processes running at once. Within my program, I limited the maximum number of processes to be created at once to fifty, since each process generates a single image and there are fifty images to be generated. If images to generate is changed, the process maximum is modified accordingly. It is possible that extremely large values of n would cause system errors, but that is beyond the scope of this experiment.

4 Mandel Multi-threading

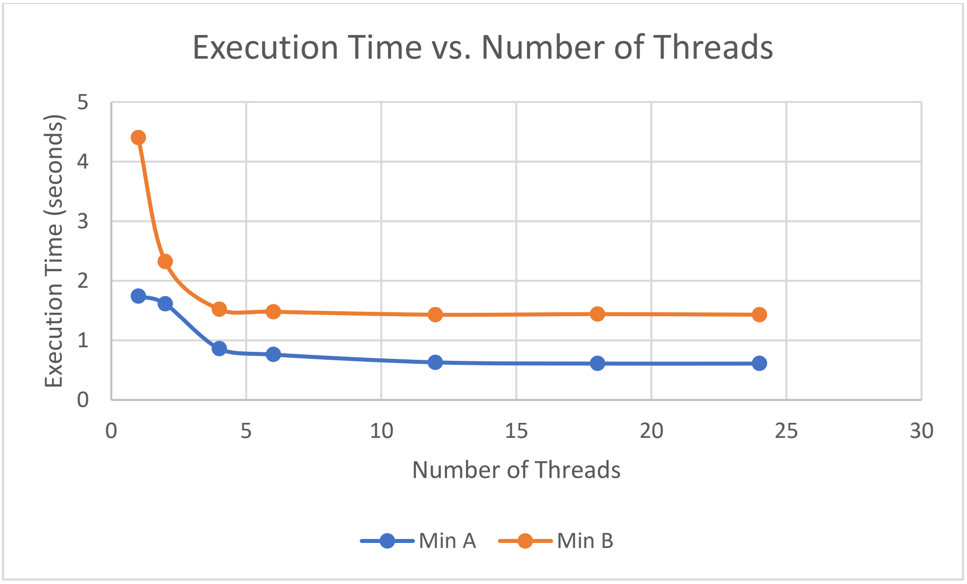
mandel generates images in the Mandelbrot set.

4.1 Data

- **A:** `./mandel -x -.5 -y .5 -s 1 -m 2000`
- **B:** `./mandel -x 0.2869325 -y 0.0142905 -s .000001 -W 1024 -H 1024 -m 1000`

Threads	A0	A1	A2	A3	A4	B0	B1	B2	B3	B4	Min A	Min B
1	1.76	1.82	2.04	1.74	1.78	4.74	4.4	4.67	4.9	4.51	1.74	4.4
2	2.19	1.82	1.61	1.75	1.77	2.32	2.48	2.5	2.34	2.34	1.61	2.32
4	0.87	0.86	0.87	0.84	0.86	1.56	1.55	1.52	1.52	1.54	0.86	1.52
6	0.79	0.79	0.78	0.76	0.76	1.49	1.48	1.5	1.5	1.48	0.76	1.48
12	0.63	0.63	0.64	0.69	0.63	1.44	1.44	1.46	1.43	1.47	0.63	1.43
18	0.61	0.62	0.62	0.61	0.62	1.47	1.44	1.44	1.45	1.44	0.61	1.44
24	0.61	0.61	0.62	0.61	0.61	1.43	1.46	1.45	1.49	1.43	0.61	1.43

4.2 Graph



4.3 Analysis

Increasing the number of threads appears to improve execution time to a certain point, beyond which the execution time remains approximately constant.

4.3.1 Optimal Number

In both cases, the optimal number of threads appears to be four. After that point, the speed increases are negligible, with improvements measured in the hundredths of seconds if a difference is detectable at all.

4.3.2 Shape Differences

Configuration B has a much more dramatic speed increases at the beginning due to the larger image size and the way that the program is threaded. The image is divided into subsets of a size Width x (Height/<numThreads>), with slight variation to account for rounding. For a larger image, the effect of threading is more noticeable as adding a single additional thread reduces each thread's workload by greater number of lines than Configuration B even requires (512 vs 500).

This effect levels out with a higher number of threads as the decrease in lines between each additional thread becomes much smaller. Eventually this reaches a point that the overhead of threading is roughly equal to the time saved by adding an additional thread. The remaining time can be accounted for as the overhead for the computation of each line of the image.