Concurrency Experiment

Matthew Latanafrancia   Jared Dettwiller

The goal of the imath program is to perform edge detection on a picture. Edge detection is the process of checking each pixel to see if there's a large change in the RGB values of said pixel in comparison to its immediate neighbors. Pixels that are considered edges keep their original RGB values, while non-edge pixels change to pure black. The result is an image that is very close to greyscale, with colored lines where the edges are.

The program works by first taking in an image in PPM (Portable Pixmap) format. It checks to make sure that the image header contains a format of P6, and if satisfied, will move on to the rest of the header to collect info such as the image width, height and maximum RGB value. If the RGB value is not at 255, we will reject the image. Once it has finished reading the header, the program will proceed to read through all the image's pixels. At each pixel, it will store the pixel's RGB values into a specialized array, which will be used for edge detection later.

Once the image reading has finished, the program will send the data array to the apply filter function, which will create an amount of threads designated by the user in order to split up the work. Each thread will then go to the threadfn function, where they will apply the Laplacian filter (used for edge detection) to each pixel in their work area. Using the formulas for each RGB value, it will properly apply the filter to its section, then wait for all the other threads to finish before being joined together.

Finally, once the filter has been applied, the program will write the data to a new file called "laplacian.ppm". Once finished, the result is a filtered image that contains all the edges from the original image.

Concurrency is when we have the same task being worked on simultaneously by different entities. We want to use concurrency in our image processor because this would allow us to work on multiple pixels at once. To do this, we can make a multiple number of threads to work on different sections of an image. This would allow the program to be working on multiple pixels at once. Doing this should greatly improve the runtime of our program.

However, using threads will introduce racing conditions in our program. A race condition is where multiple threads are trying to do something to the same value at an address at once. One thread can be faster than another and you must take care of that in the code. The main racing condition in our program was when finishing our image processing where the threads would end. We need our threads to end at the same time or else we might be returning an incomplete result image out of the apply_filters function.

To solve this, each thread is going to have to wait for each thread to get to a specific point after creating threads. We can do this using pthread_join on each thread which will make the main program wait for each thread to finish before continuing to writing the image into a ppm file.

With our experiments, our goal is to test out how changing different variables will change the runtime of our program. What we are focusing on are the number of threads used to process an image, and the resolution of the image that we are processing. In order to make accurate conclusions to our experiment, we're going to conclude our experiments in two different experiments. Our first experiment will change the number of threads in each test case. Our second experiment will change the resolution of the image used in each test case.

With our first experiment, we used the same ppm image for each test case to keep the experiment consistent. To further give us consistent times for each test case, our program will run through the filtering process for the image 50 times. The time for each of these runs will be saved so they can be used to then average the time per run through the program. To see the difference when using multiple threads, our tests will consist of using 1, 2, 4, 8, and 16 threads in each of our test cases. We can then compare the averages we have for each of the tests.

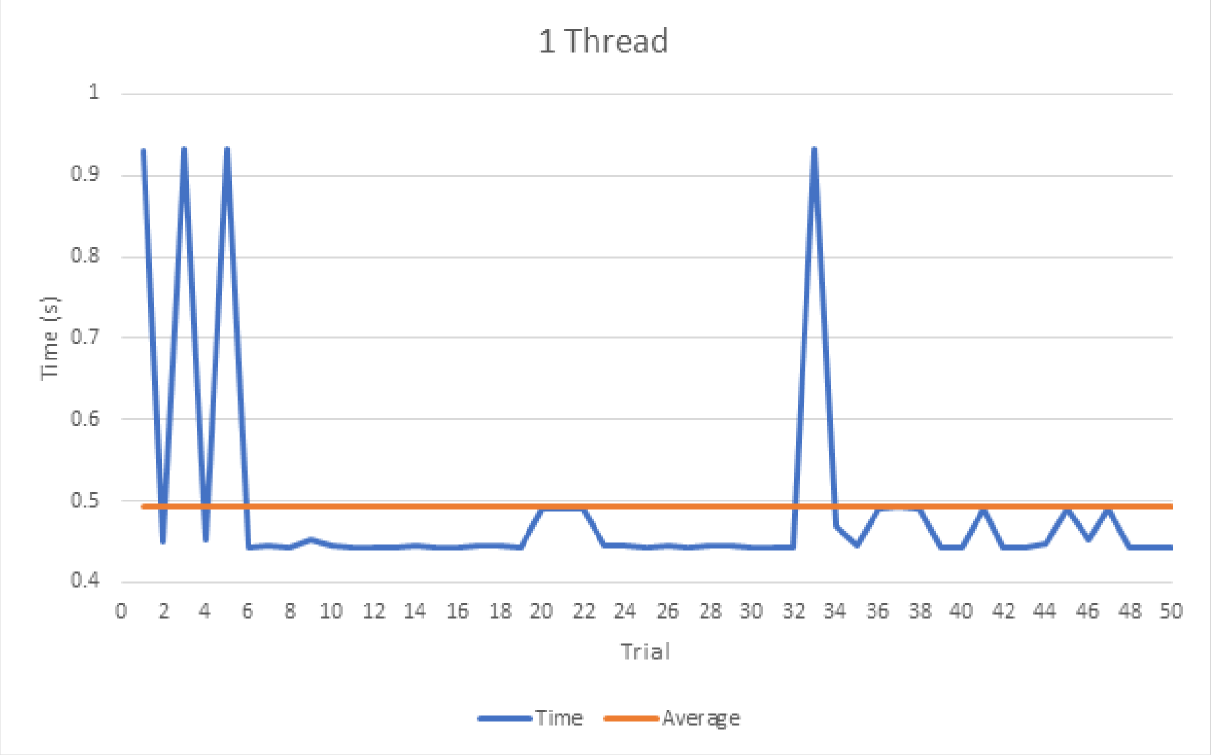
Our hypothesis for this experiment is that when we use a greater number of threads, the average time per run through the program will be shorter. This is because when we have a greater number of threads, we will have more threads working on processing the image at once. In other words, when we have a single thread, only one pixel is being worked on at a time whereas when we have multiple threads, multiple pixels theoretically will be worked on at the same time. Therefore, this should make the runtime faster with multiple threads.

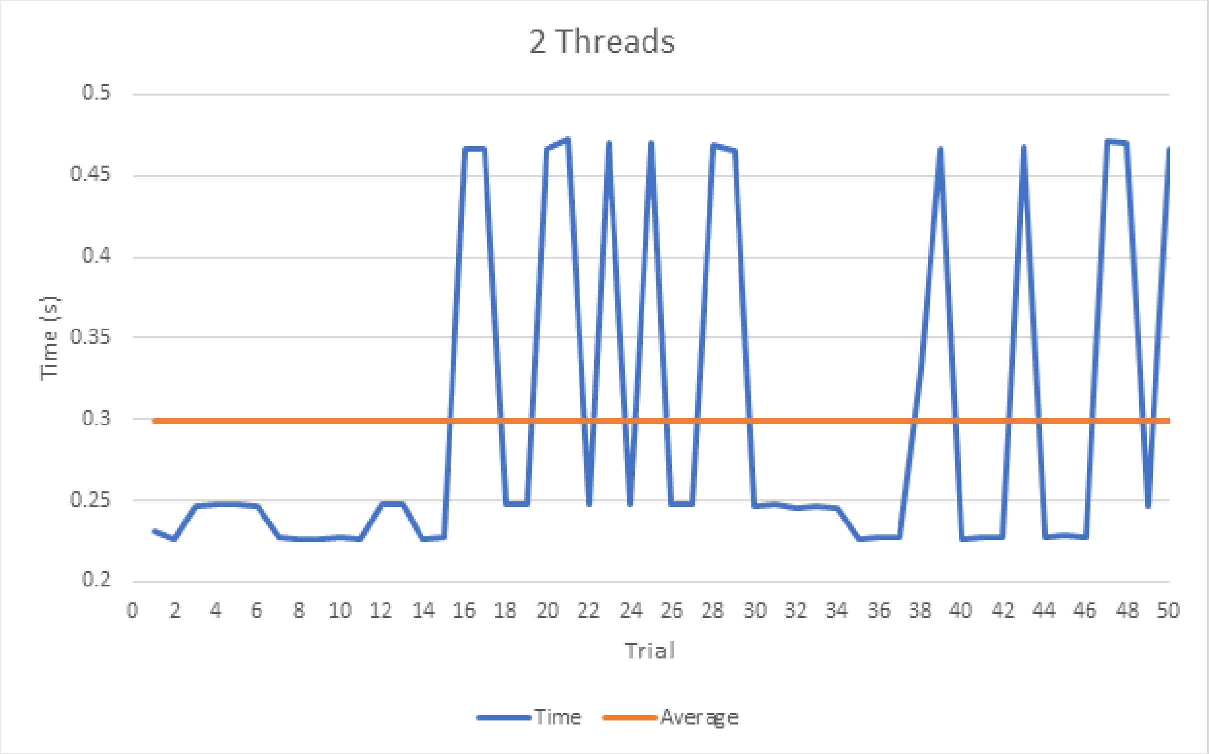
With our second experiment, we will use the same image with different resolutions to test the differences in runtime when we change the resolution of the project. Will the program behave differently, and how much time will it take? We used the same process as our first experiment, however, now we can use a constant number of threads. For our test cases, we will be using four threads for each different resolution. We want to keep our number of threads constant so we can only test resolution. The datapoints that we will gather from these tests will be like the last experiment. We are going to take fifty consecutive runs and average those runtimes for each different resolution.

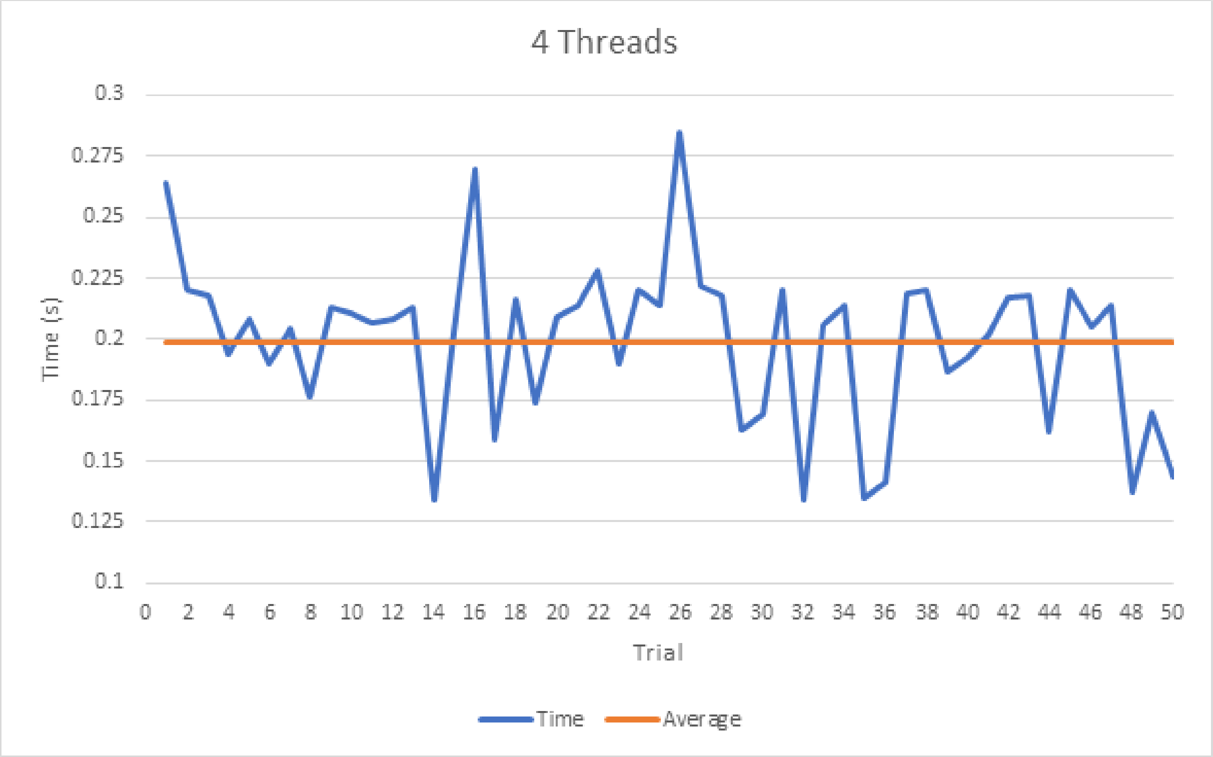
Our hypothesis for this experiment is that when we use a greater resolution when processing the image, the runtime will be greater. This is because we have a greater number of pixels to run through per thread when the resolution is of a greater size. The program would need to do more work to process the whole image, which consequently would mean that the program would need more time to do the extra work.

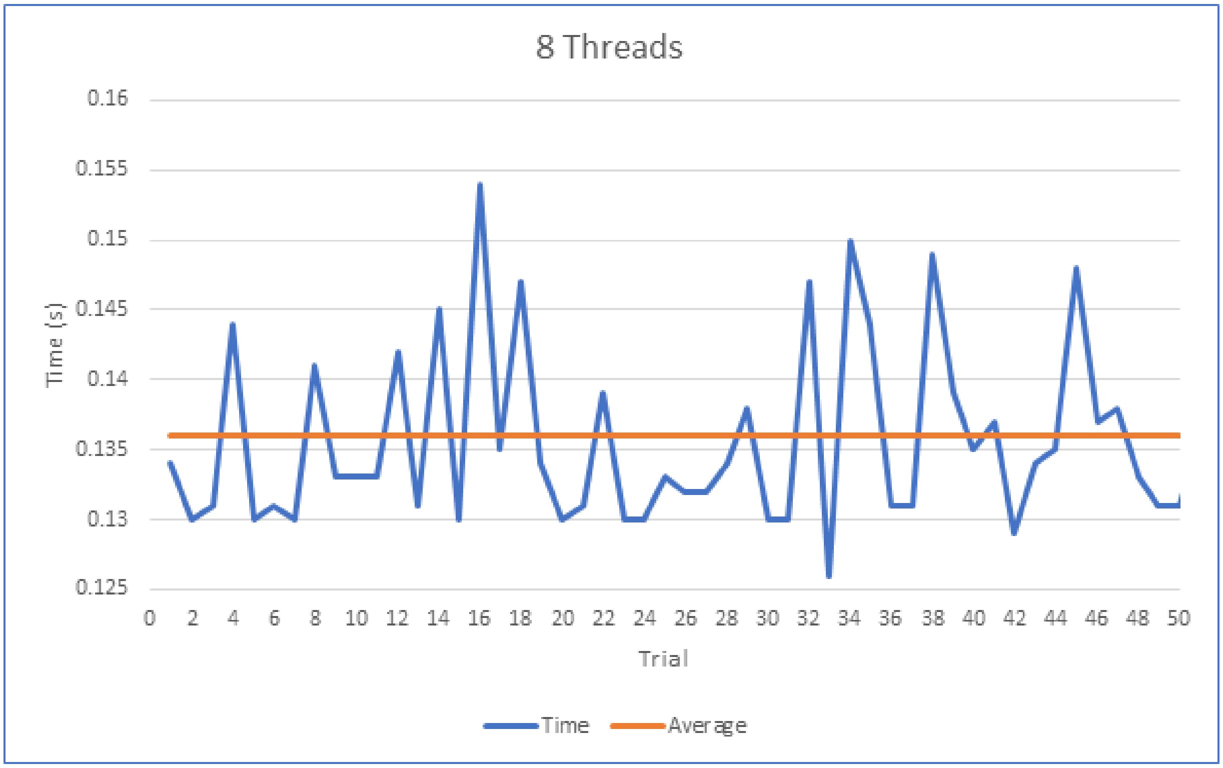
Our images that we used are from https://filesamples.com/formats/ppm. The file "sample_1920x1280.ppm" is our PPM file that we will use for our thread experiment. All four photos at this link will be used for our resolution experiment.

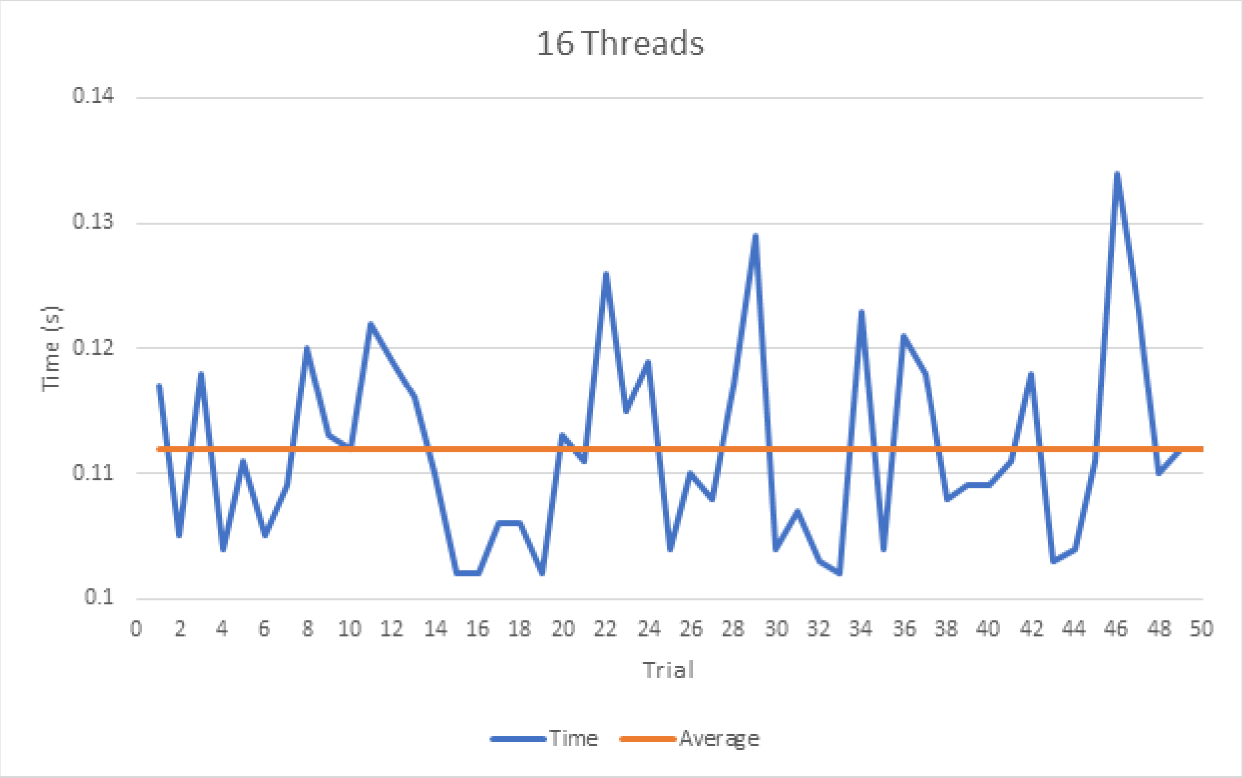
Below are pictures of the graph of our experiment first experiment.


1 Thread
1
0.9
0.8
0.7
0.6
0.5
0.4
Time (s)
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
50
Trial
Time
Average


2 Threads
0.5
0.45
0.4
0.35
0.3
0.25
0.2
Time (s)
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
50
Trial
Time
Average


4 Threads
0.3
0.275
0.25
0.225
0.2
0.175
0.15
0.125
0.1
Time (s)
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
50
Trial
Time
Average


8 Threads
0.16
0.155
0.15
0.145
0.14
0.135
0.13
0.125
Time (s)
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
50
Trial
Time
Average



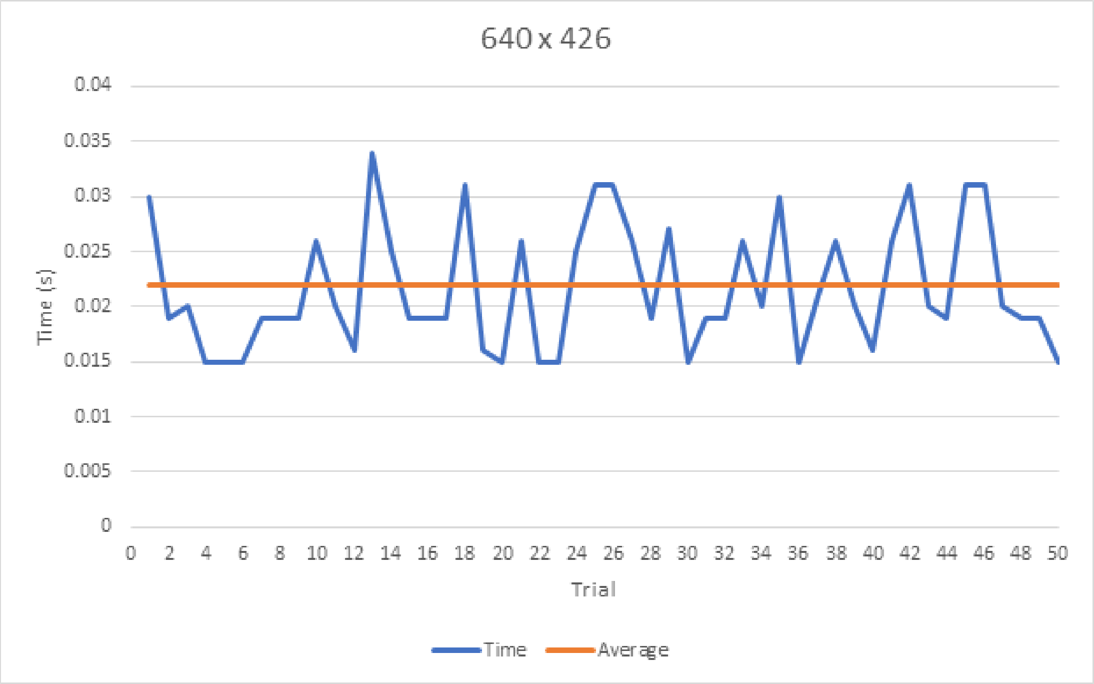
With our threads experiment, we saw that our hypothesis was correct. We can see that the average time when using only one thread for image processing is 0.492 seconds. Comparing this to the average time of 0.112 when using 16 threads, the time when using 1 thread is longer by 0.38 seconds. As we continue using more threads, the average time for our image run will continue getting shorter as shown by the graphs.

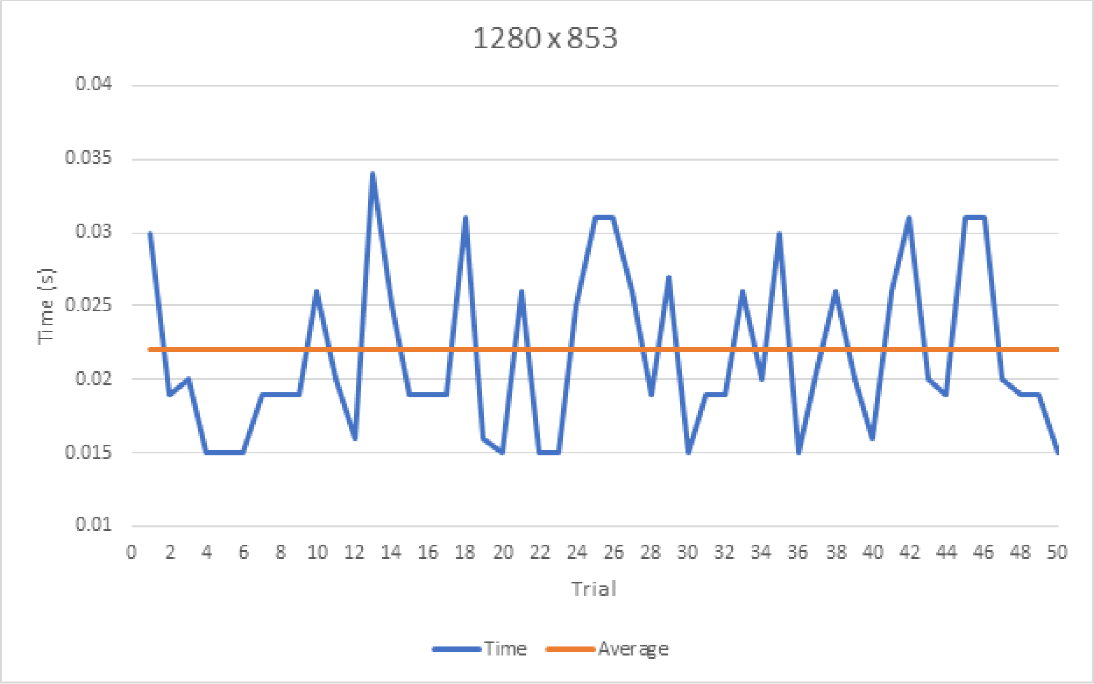
Regarding runtime of our image processing, what we noticed is that from using 1 thread to 2 threads, the difference in time is clear. However, the difference in average time the more threads we use gets smaller and smaller. This means that when we start using more and more threads, in our test case, the differences between the average times get smaller the larger the average times. Using this data, you can possibly see how many threads we should use for the most optimal thread to time ratio. In other words, we can find how many threads are most efficient in our algorithm.

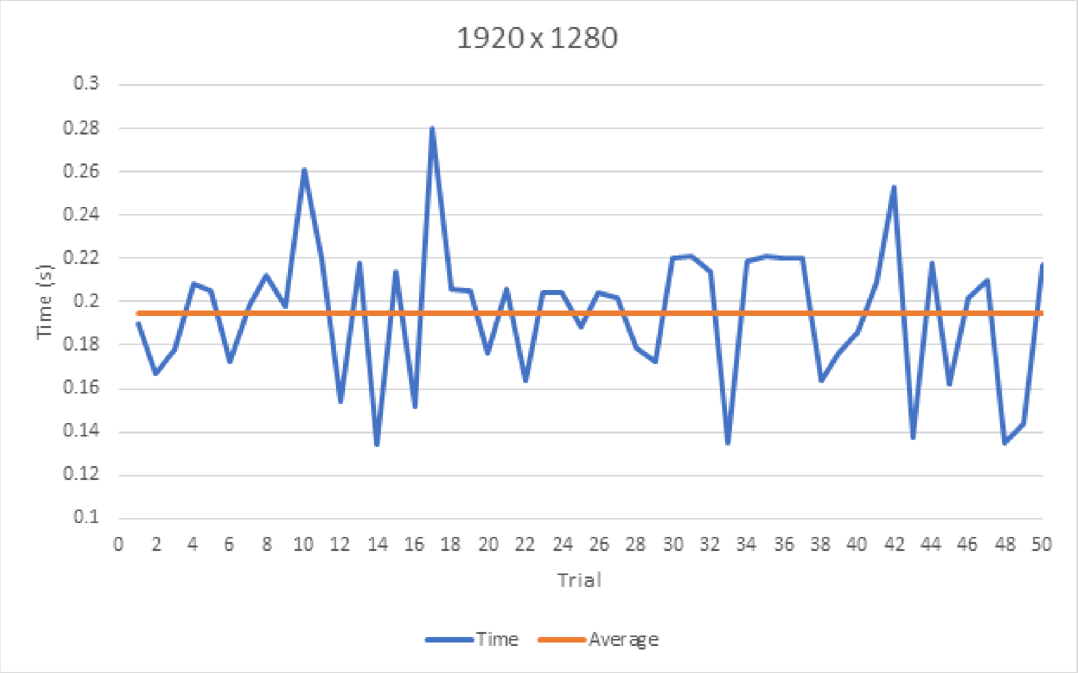
A contributor to that difference could be connected to how when we use more and more threads in our program, the more stable our program seems to be. The graphs corresponding to a larger number of threads used can be seen having more spikes in runtime than the graphs corresponding to a lower number of threads used. Most of the time, our test cases when we use 1 thread are level while when we use more than 1 thread, the data tends to spike more.

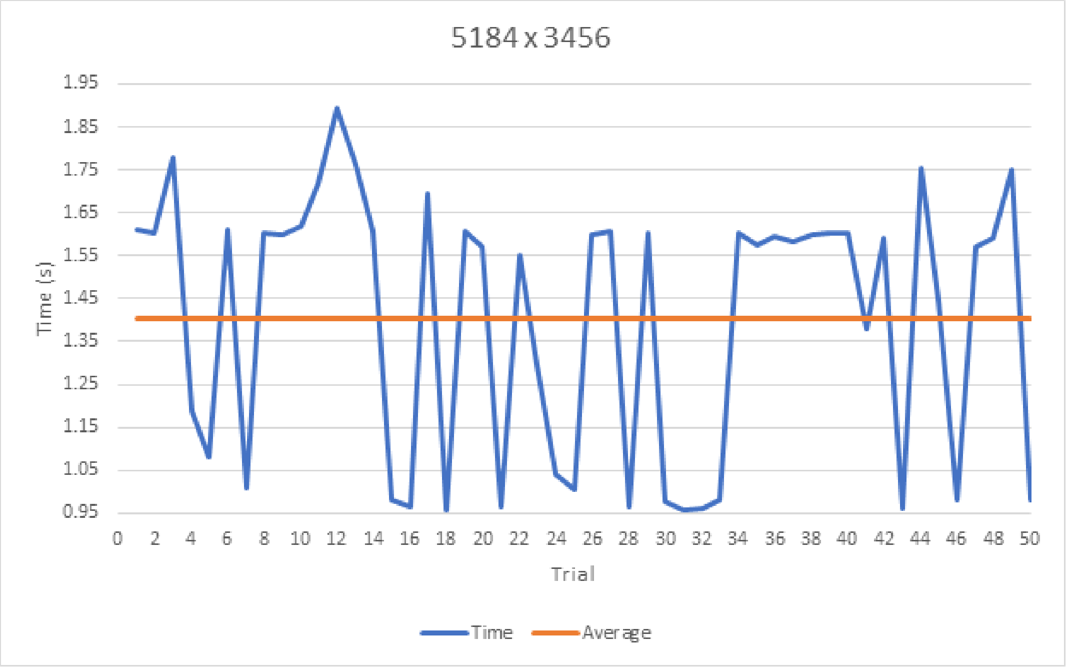
Our assumption for this behavior is that the introduction to multiple thread processes affects the runtime by adding more variables into our runtime. Since these processes can be faster and/or slower than each other, when we add more threads, we should see more fluctuations in our graph because the threads won't be running exactly at the same time. Therefore, the fluctuation in our graph will consequently change.

With experiment 2, we can see some of the same characteristics as in experiment 1.









Firstly, we can see that our hypothesis about the length of runtime was correct. Comparing the average runtime for our smallest image to the average runtime for our largest image, we can clearly see that the smaller image takes a shorter time to process. This can be attributed to our program needing to do less work on the smaller image than the larger image. Since the larger image has more pixels that need to be processed, our program will need to calculate the values of more pixels. Due to the increasing load of work, our runtime is going to get longer as we have a larger resolution.

For example, our average runtime for our smallest resolution is 0.022 seconds and the average runtime for our largest resolution is 1.403 seconds. From those 2 values, we can see that the average runtime is greater for a larger image. For every one of the other test cases in experiment 2, it suggests the same idea.

One of the more interesting attributes that we can see from both experiments that we conducted is how the average runtime corresponds to the range of our data. Since we saw that the runtime data wasn't as constant as we hypothesized it would be, we calculated the range of our experiment test cases.

Experiment 1:

# of threads	Minimum runtime (s)	Maximum runtime (s)	Range size (max-min)
1	0.442	0.933	0.491
2	0.226	0.473	0.247
4	0.134	0.285	0.151
8	0.126	0.154	0.028
16	0.102	0.134	0.032

Experiment 2:

Resolution	Minimum runtime (s)	Maximum runtime (s)	Range size (max-min)
640x426	0.015	0.034	0.019
1280x853	0.06	0.126	0.066
1920x1280	0.134	0.280	0.146
5184x3456	0.958	1.895	0.937

When comparing the range sizes of our test cases (either number of threads or the resolution we used), we can see that when the average time of that case is shorter, than the range size of that test case will be larger than another case where the average time is longer. This means that we can expect the time fluctuation in terms of range to be smaller with a shorter average time than a longer average time. This was the trend in both experiment 1 and experiment 2.

For example, our lowest average time was when we were processing an image with a resolution of 640 x 426 pixel and our largest average time is when we were processing an image with a resolution of 5184 x 3456 pixels. These cases contributed to the smallest and largest range size of 0.019 seconds and 0.937 seconds respectively.

Spending less time on an image means that there is less time for the systems threads to fluctuate. In other words, when we are spending less time on an image, the threads are not running for as long so if a thread is fluctuating, the thread will fluctuate for a shorter amount of time than when you're spending more time on image processing. Spending more time on an image will also mean that a thread will have more time to have fluctuations during its process.

In conclusion, we have learnt that when a thread does more work, the runtime of that thread will be shorter. To mitigate the runtime, we can use more threads in our program. This is because when we add more threads, the work that needs to be done is split amongst each of the threads. This gives each thread less work to do, which will end up decreasing our runtime as we have seen in experiment 1. We also see the same trend in experiment 2 since a larger resolution means more work for the program, effectively meaning more work on each of the threads being used as well.

We've also learned that when we use a different number of threads, the fluctuation of runtime will be different. Running one thread was consistent because it was only running on one thread and didn't need to factor in other fluctuating or slower threads. When a thread finishes before another thread, then that thread will have to wait for the other threads to finish before moving on to writing the image. This is where the fluctuation occurs and why there is an average.

The last thing that we learned from our experiments is that the range sizes of each test case of our experiments is based on how long we spend in the processing part of the program. When we are spending more time processing an image, the range is larger since we are spending more time on the image.

Using this information, we can see that depending on the resolution, there is an optimal workload you should keep for each thread. When we use too many threads, the time difference can be shorter or close to non-existent. When we tested 8 threads and 16 threads, it seemed to be that the difference was not as great, however when we tested 1 and 2 threads the difference was more obvious and noticeable.

This can be connected to what we learned about concurrency; that we can use concurrency in order to optimize our program in parts. We are implementing concurrency in our project using threads. When we use more threads, more work is being done at the same time which consequently affects our runtime. As we've seen from our experiment, when we have more work being done at the same time, our runtime will be shorter.

Concurrency is an important thing to implement in use cases like image processing when we have a large task to complete and can be split into different parts. In our case, we were able to split our image processing into different parts where each thread can go through. We could use what we learned about concurrency and apply it to other use cases when necessary. Even though you don't have to use them in only large tasks, you can see more of a difference when using multiple threads for a large task.

Overall, we have a better understanding about how threading and concurrency works after working on this project. We have seen and demonstrated the importance of concurrency in some applications and how it can optimize certain tasks.