CSCI 347, Winter 2021, Project 2
Connor DeMarco and Myles Flack

3/5/21

1 Goals and Introduction

The overarching goal of this project is to create a system within C that can read a Portable PixMap file (PPM) and store the RGB values to then be later manipulated. In this particular case, we are applying this Laplacian filter to each pixel:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Using a filter that is 3x3 rather than simply changing the RGB values of each pixel requires extra code. Not only do we iterate over each pixel of the input image, but also traverse the surrounding area so that the filter can aggregate data for an output.

Concurrency is a widely utilized tool in the programming world so that software may run faster, or so we hope. Applying concurrency to this program made perfect sense due to the fact that applying a filter to each pixel of an image is a large repetitive process that is not dependent on sequential execution. A common mistake when working in computer vision is applying a filter chronologically, and "building" up the effect on itself. This produces a gradient in the image. However, storing the input and output image separately solves this issue. We applied concurrency using the pthread library's *pthread_create* and *pthread_join* functions. This allowed us to split the computational load into horizontal sections and be run at the same time.

2 Code

2.1 main()

The *main*() function, as always, was used as a starting point for the function and ended up being just a place to call our helper functions as well as print out a few simple results to terminal. Originally, we were also measuring time in main, simplifying the process by just recording the time it took to start and return the $apply\_filters()$ function. Eventually we decided to modify this for the current version to better comply with the hints given in the project page.

2.2 readImage()

readImage() Is quite simple, after opening the original file in binary mode, we are manually reading in the header of the PPM file, checking for the appropriate data such as the RGB cap and whether the file is in a P6 format. We then load the entirety of the pixel data into an array to be used later.

2.3 writeImage()

$writeImage()$ Is the shortest function in our code, in that it just dumps the entirety of our pixel data into the output file with a single $fwrite()$ call. We are also manually printing to the file for the important header information.

2.4 threadfn()

$threadfn()$ Is the computationally heavy portion of our code, and the reason that the asymptotic runtime is so poor. Here we are moving left to right and then down a row to iterate through each of the output pixels, as well as aggregating the surrounding information for the Laplacian filter. The size of the area we are iterating through is dependent on the number of threads the program is running on and therefore this function is not hard coded to simply move through (height * width) pixels.

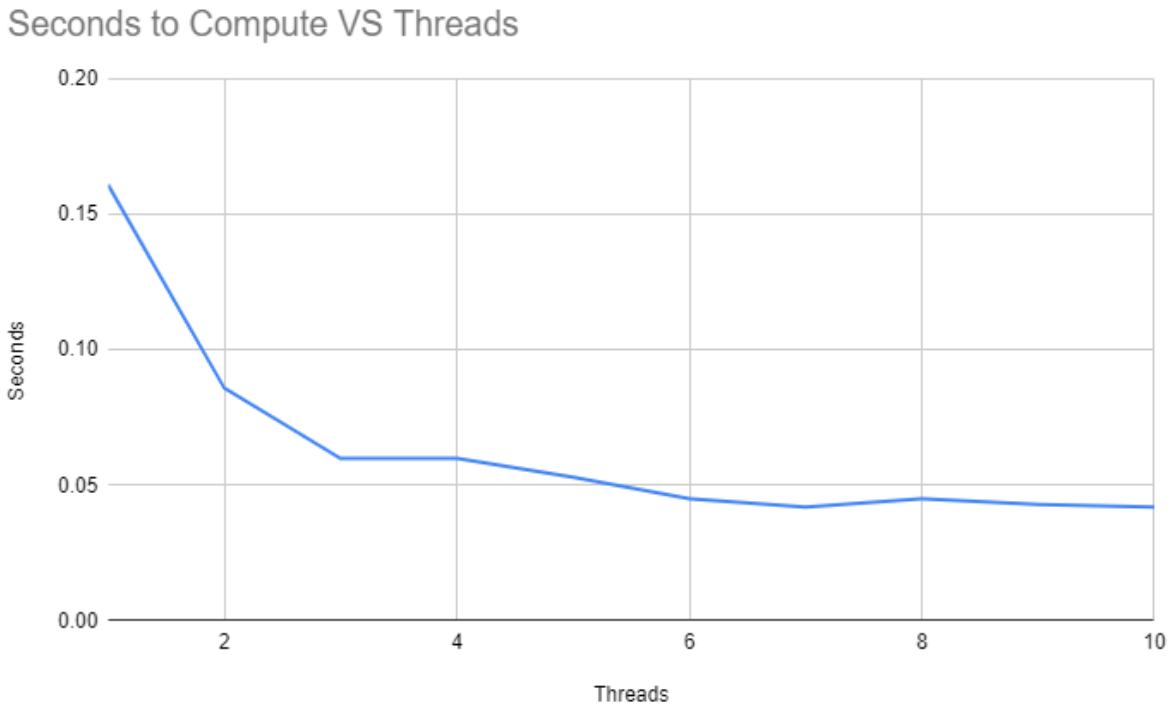
2.5 apply_filters()

$apply\_filters()$ Can be considered the brains of the operation, as well as the "dumbest/worst" part of the operation. Anyone that enjoys following coding standards would agree that we have committed a fairly egregious act in simply hard coding the number of desired threads, forcing the user to re-compile the code each time they want to test a different number of threads. Fortunately this is a small program and is not an issue, but in the future could be changed to accept user input in the terminal.

3 Testing

In order to test the efficiency of our code, we followed the hints provided in the assignment page and are tracking the time it takes for all of our threads to resolve inside $apply\_filters$. The code itself it tracking time down to the microsecond, however for readability we are only displaying the date down to the thousandth of a second. We tested the program's runtime thoroughly for 1-10 threads, as well as some outliers such as 1000 threads. Each thread was tested five times to produce an average, and then recorded in an excel document.

4 Results



Threads	1	2	3	4	5	6	7	8	9	10
Seconds	0.161	0.086	0.06	0.06	0.053	0.045	0.042	0.045	0.043	0.042

As you can see, the data creates a beautiful downward slope indicating that this piece of software does indeed benefit from being multi-threaded. When utilizing only a single thread, the worst runtime we recorded was 0.173 seconds, and an average time of 0.161 for the same thread count. This then quickly drops off to an average time of 0.086 seconds with two threads... almost perfectly half the average runtime. Three threads also saw a reduction in average runtime at 0.6 seconds, but not as significant.

As we continued to test, you can see there is an apparent bottleneck of some kind in our system. Whether this is hardware or software related is not perfectly clear, but intuition grants that this is related to the asymptotic complexity of the code, and the inefficiency of $four$ nested loops to iterate over the filter and images. It is important to note that this was performed on a single computer with an eight core processor, and that we had no worries about computer performance. If this had not been the case, and we were limited to a single core processor, the number of threads created in our code would not have mattered due to the fact that the CPU wouldn't have been able to handle more than one computation a time. While a single core processor is indeed capable of executing multiple tasks concurrently, this is really an illusion because it is switching between two tasks.

5 Conclusion

This project was great experience for our group in that it forced us to deepen our experience with the C language, and also get hand-on practice with threads and concurrency. The biggest hurtle for the two of us was still the syntax of handling all of the pointers and data within

the code and fleshing out the skeleton that we were handed. Had this assignment been in a language of our choice and no skeleton was provided, this code would have been a cake walk. Alas, this was not the point of the assignment and that is well understood.

In terms of what we learned about concurrency, this project was useful in solidifying some assumptions about software performance. Going into this class, one may assume that more cores engaged is always better, and that concurrency will do nothing but improve performance. In reality, silicone can only take us so far, and more threads is not always better. As seen in the data, we experienced a plateau in performance after five threads. There was some oscillation in the average runtime from here on out, but overall the performance stayed the same whether we wanted to use 10 or 1,000 threads. While we did not test this, I'm sure that the data would create an inverted bell curve if we continued to push into thousands of threads, where each thread is then being created and joined all for the purpose of a few pixel calculations.

Overall this project was a great learning experience and we are very pleased to have it producing the expected image results.