David Lee, Gavin Crane
Professor Elglaly
CSCI 347
March 7, 2022

PROJECT 2 REPORT

1&2.
Imath is a program that takes a ppm image file as an argument upon execution. The program reads the image, stores it, then applies a laplacian filter over the image data, and outputs the newly converted data into a new ppm file. We are given two structs, PPMPixel which contains the RGB values of a pixel, and parameter, which contains the image, resulting image, dimensions, and start and size for work. We implemented the following functions: main(), readImage(), apply_filters(), threadfn(), writeImage(), and a helper function: color_balancer().

In main, we do some basic error handling for checking valid input: i.e correct number of inputs. The first thing we do is to try to read the image file argument using the readImage() function. In readImage() we open the ppm file and we use fgets() to retrieve the majority of the header data. We specifically use scanf to retrieve the image dimensions. With this data, we make sure the image format and header data are correct and set the width and height variables to use later. We then allocate memory for PPMPixel buffer that we are going to read to. The buffer size is the width times the height of the image times three. We then read the open file into the buffer and return buffer that contains the read image data.

Back in main, we now start our timer to count execution time. We call apply_filters() with the read image buffer, dimensions, and timer. Now in apply_filters. At the top, we have the struct timeval to handle our timer, using the gettimeofday() function. We allocate memory for the resulting image and declare our number of threads which is determined by our global variable “THREADS”. We then have calculations to determine the work size of each thread (h/THREADS). We then allocate memory for a struct params array for the threads. Each thread needs its own data on where to start and where to end, which is what this struct contains. The struct also contains the read image, the resulting image to write to, and the dimensions.

Now we get to thread creation. We use two loops here. One for creating the thread that calls the threadfn() function and the other that joins the thread and ends its execution. In the loop, “i” identifies each thread. We take params[i] and fill it with the necessary data (where to start and where to end). All of this is calculated based on the iterator “i” and

work size. We also add the read image and result image, along with the dimensions to the params[i] struct. There is a check for if we are on the last thread, we must make sure it does the rest of the work in case the work does not divide evenly across all threads. Telling each thread where to start and its work size allows us to run them on the image all in parallel, theoretically improving performance.

We pass our params[i] to threadfn() in the pthread_create() function. In threadfn(), we iterate over a chunk of the image in two for loops (height and width). This “chunk” is determined by the params that was passed into it (start and size from thread creation). This allows us to get to each pixel in the image chunk. Within these two nested loops, we then iterate over the given laplacian filter, which is a 2d array of integers representing the filter. We use this filter to compute the resulting RGB values for each pixel, and account for when they go over or under the RGB range, and save the result to the image result.

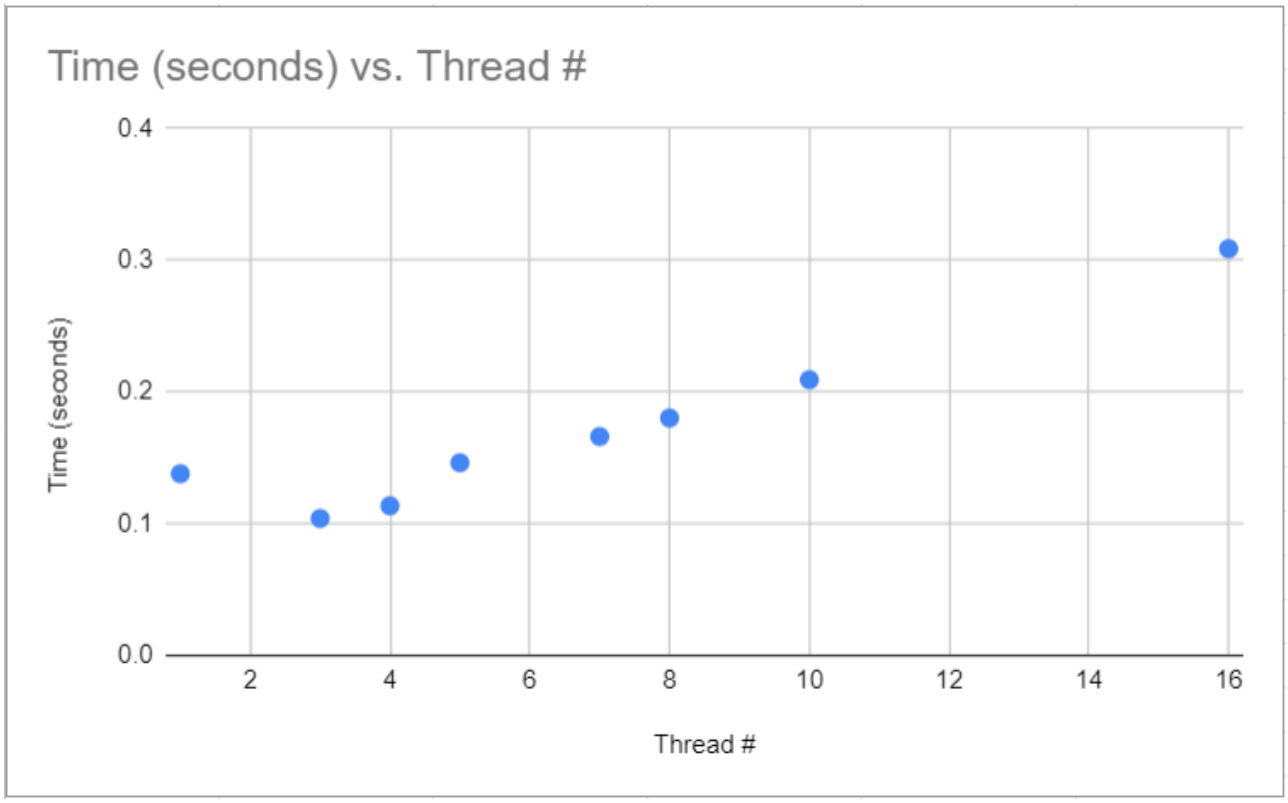
Once all the threads are joined. We stop the timer and the image result is complete and writeImage() is called with the resulting image, dimensions, and output file name. The output file name is just the original file name with “_laplacian.ppm” concatenated to it. In writeImage(). We create a new file and populate the header using fprint() to write the image format (P6), and dimensions (All the header data). We use fwrite() to write the image result to the newly created file.

3.
The experiment was conducted by running the program on sage_1.ppm with different thread counts. Data was collected by running n number of threads 10 times and taking the time (ms) to complete its process. We decided on these thread counts to test if there is any difference in running an odd amount of threads over even ones.

The program data was taken on lab machine cf-420-17, at around 5:00 pm on March 7th. The machine has a quad-core processor.

4.

Sage_1.ppm\threads	1	3	5	7	4	8	10	16	100
Avg(s)	0.1379	0.1039	0.1462	0.1661	0.1135	0.1802	0.2092	0.3087	1.7397
Avg(ms)	137879.7	103911.5	146231.3	166050.6	113460.6	180151.4	209180.8	308717.2	1739665
	139338	112245	132966	181780	108788	188791	206333	355480	1668343
	134186	108609	150646	168619	115619	174363	213135	301028	1671612
	134957	90347	144588	160083	108782	186466	238075	303996	1628409
	146467	92159	150017	162655	105669	174415	190436	301495	1604101
	142389	119230	155132	178677	123924	183262	228710	299343	1705218
	139547	115278	129580	160438	113322	163482	202303	290425	2285926
	128653	98384	147759	166386	104247	170402	217137	308026	1629392
	131024	90795	165156	171767	118572	183484	201220	317032	1831351
	148713	107096	135753	151044	130317	185050	200585	307664	1669014
	133523	104972	150716	159057	105366	191799	193874	302683	1703285



5.
From the data obtained. We observed that the image processes fastest between 2 and 4 threads. The assumption we have is that if the thread is lower than 3, the number of processes is too low to quickly process the whole image. Whereas if the number of threads is higher than 4, the number of processes is exhausting the limit of what the core processors can process eg. low free memory space, high disk uptime, etc. When we have more than 4 threads one of the CPU threads will have to swap between one of the program threads, slowing down execution time.

Some of the limitations of the experiments could include hardware reliability or the number of background processes that are running on the machine during data collection. Our data could change if we collected it during a different time of day, where the machine could be experiencing more or less load. There is also a possibility that our program is not optimized properly in delegating tasks for each thread.

6.
We learned that concurrency can help finish a task faster than just doing it in one go. There are also limitations with concurrency where the result can be the opposite of what we want. Overloading a system with too much concurrency, significantly slows down the system. The work required to create mass amounts of threads could be more work than the work that needs to be done. The CPU can have fewer threads than the software threads, slowing the performance.

Proper memory management is important when dealing with files containing a significant amount of data. Once we are done using a file reader or any data placeholder, we have to clear it. Obsolete data takes up memory space, so once we clear, we have more memory space for larger processes.

We also learn that we cannot manually text edit ppm files. We had an assumption that the header text (eg. P6, comments, dimensions, RGB value) are just regular text inside the ppm file. We tried to do a copy of the ppm image without its header and manually type it in after that, which resulted in the image corrupting. To avoid that from happening, in the program, we have to generate the header data first and follow up with the image pixel data using the FILE extension.

We learned that we did not need a mutex for our threads. Each threads' work was independent of other threads. They did share the read image and the resulting image to write to, however, the params struct for each thread told it its bounds and never went into another thread's bounds. Thus they never interfered with each other. We also

learned that you cant create and join threads in the same loop, because if you do, then they just execute in sequence, one at a time.

Sources:

- Great Learning Team, Introduction to Edge Detection | What is Edge Detection,Feb 16 2021
  https://www.mygreatlearning.com/blog/introduction-to-edge-detection/

- Netpbm, PPM- netpbm color image format
  Oct 09 2016
  http://netpbm.sourceforge.net/doc/ppm.html#index