Edge Detector Concurrency Report

Isaac Boaz

March 18, 2024

Abstract

The goal of the edge detector project was to implement a concurrent edge detector program utilizing a laplacian filter.

1 Structure

The structure of the program follows basic C programming requirements. Each function was written to complete its assigned task with no side effects. The basic workflow of the program is as follows:

1. Call the program with the PPM file paths to be processed.
2. Create a `file_name_args` struct array for each file argument.
3. Populate the array with appropriate file information.
4. Create a thread for each file argument.
    4.1. Each thread reads in the image.
    4.2. Creates `LAPLACIAN_THREADS` threads to process the image.
    4.3. Feeds the image to the laplacian filter threads with the appropriate row range.
5. Waits for all threads to finish.

1.1 Global Variables

The one global variable used in this program is `total_elapsed_time`, which is responsible for tracking the total time taken to process all images. All other references to globals are constant defines.

1.2 Functions

1.2.1 compute_laplacian_threadfn

This function is responsible for doing the actual laplacian filter computation. Given a `parameter struct` pointer, it iterates from the given `start` through to the `end` and applies the laplacian filter to each pixel.

As the image is passed by reference, the function does not return anything.

1.2.2 apply_filters

This function is responsible for creating the threads that will apply the laplacian filter. It takes in pointers to an image, its dimensions, and a pointer to the `elapsedTime` double.

The function creates `LAPLACIAN_THREADS` threads and assigns each thread a range of rows to process. The function then waits for all threads to finish. Lastly, the function calculates the total time taken to process the image and updates the `elapsedTime` pointer respectively.

1.2.3 write_image

This function is responsible for writing a given `PPMPixel struct` given a filename, widht, and height. A hardcoded `"P6"` is written to the file, as well as the provided width, height, and maximum color value.

1.2.4 read_image

Much like `write_image`, this function is responsible for reading in a `PPMPixel struct` from a given filename. The function reads in the file header and then reads in the pixel data.

If the file does not contain the valid `"P6"` header, the function exits early.

1.2.5 manage_image_file

This function is a basic wrapper around `read_image`, `apply_filters`, and `write_image`. It takes in a `file_name_args struct` pointer and calls the aforementioned functions in order.

1.2.6 main

This function is responsible for parsing the command line arguments and creating the necessary threads to process each image.

2 Concurrency

Concurrency is an important aspect of this program and how it achieves efficency. Multhreading is heavily used to allow for processing multiple images at multiple times.

2.1 Threads

As described in the above functions and structure, one thread is used for each image file provided, with that thread creating subsequent threads to handle each image.

As a result, the number of threads created can be calculated as follows:

$$\text{num\_threads} = \text{num\_files} + \text{num\_files} \times \text{LAPLACIAN\_THREADS}$$

An analysis of how the value of `LAPLACIAN_THREADS` affects the program is provded in section 4.

2.2 Race Conditions

Race conditions were avoided by ensuring that each thread was given a unique range of rows to process. This was done by calculating the number of rows to process and then dividing that number by the number of threads to create per image.

Since there are no global variables used for storing arrays or data, (all image is passed by reference), there is no possibility of a race condition occurring.

3 Experiments

The experiment run was executed on Western Washington University's Lab Machines. Specifically, the `cf167-23` was used for this experiment. All tests used the same base image (cayuga_1.ppm) which was 1440060 bytes in size.

3.1 Conditions

The primary question to be answered by the experiment was how the number of threads used to process the laplacian filter affected the total time taken to process the image.

The number of threads used to process the laplacian filter was varied from 1 up to 1028 threads. With each variance, the edge detector was ran 100 times and the average time taken to process the image was recorded.

4 Results

The results of the experiment are graphed in Figure 1.

5 Results Analysis

Based on the results, it is clear that the number of threads used to process the laplacian filter has a significant impact on the total time taken to process the image. As the number of threads increases from 1 to 64, the total time taken

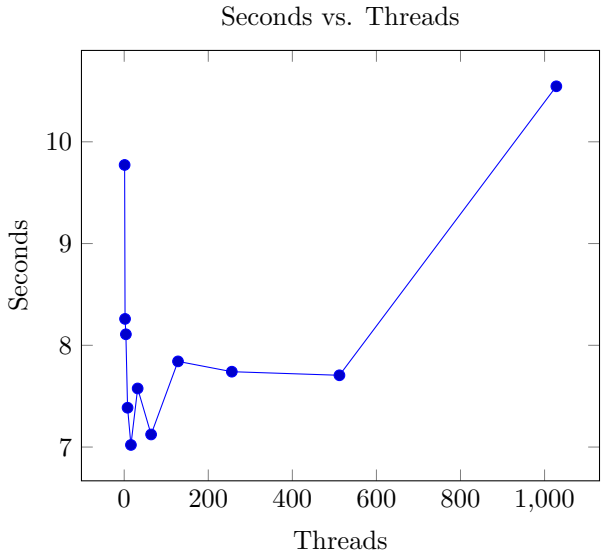

Figure 1: Seconds vs. Threads

to process the image decreases. However, after 64 threads, the total time taken to process the image increases as the number of threads increases.

It should be noted that the results of this experiment may vary depending on the image used. Outside of software differences, the hardware used to run the experiment may also affect the results.

6 Conclusion

This program was successfully able to utilize an arbitrary amount of threads (greater than zero) to appropriately process an image using a laplacian filter. The results show that the performance of program is signficatnly altered off of the number of threads used.

It is likely that the number of threads used could be optimized based off the image size. Further experiments could be run to determine the optimal number of threads to use for a given image size.

Additionally, experiments for how the program handles multiple images at once (instead of one at a time many times) could be run to determine how the program scales with multiple images.