Homework 5

By Alexi Chryssanthou

1 Introduction

Our goal was to perform the parallelization of a 2D convolution calculation, and then compare runtime and speedup to a sequential version.

To accomplish that task, I wrote the “task” package, and implemented the “editor” to use said package. While positive speedup was achieved (>1) over the sequential version, the thread to thread performance over 2 threads was not as beneficial as expected.

2 Decomposition

The task was to spawn a finite number of threads that would wait to complete work. Though no necessarily required, I managed functional and data decomposition at the same time using channels.

I accomplished this by reading in each csv line and creating a task pipeline based on effects and color layer for each image. The pipeline was sequentially queued with these tasks (e.g. img1, blur, red; img2, sharpen, blue, etc.) which were then consumed by the “process” routines.

Each task contained a struct for the data associated with each color layer and a struct of mutexes. In order to avoid image corruption while calculating the convolutions, each color layer in the image struct had a mutex associated with it. Whenever a color was being updated, we locked that color’s mutex, updated the color layer, and then unlocked the color’s mutex. In this way, we could have simultaneous operations being performed on the same image without issue.

This arrangement allowed for effects and color layers to be processed concurrently and independently, achieving both functional and data decomposition.

3 Specifications

To test the (potential) speed up from sequential to parallel, a series of tests were run. All tests were run back to back from terminal, on a cold-booted 2017 13” MacBook Pro running macOS Mojave (Version 10.14.1), with an Intel Core i5 3.1 GHz processor (2 physical cores, 4 logical cores) and 8 GB of 2133 DDR3 RAM.

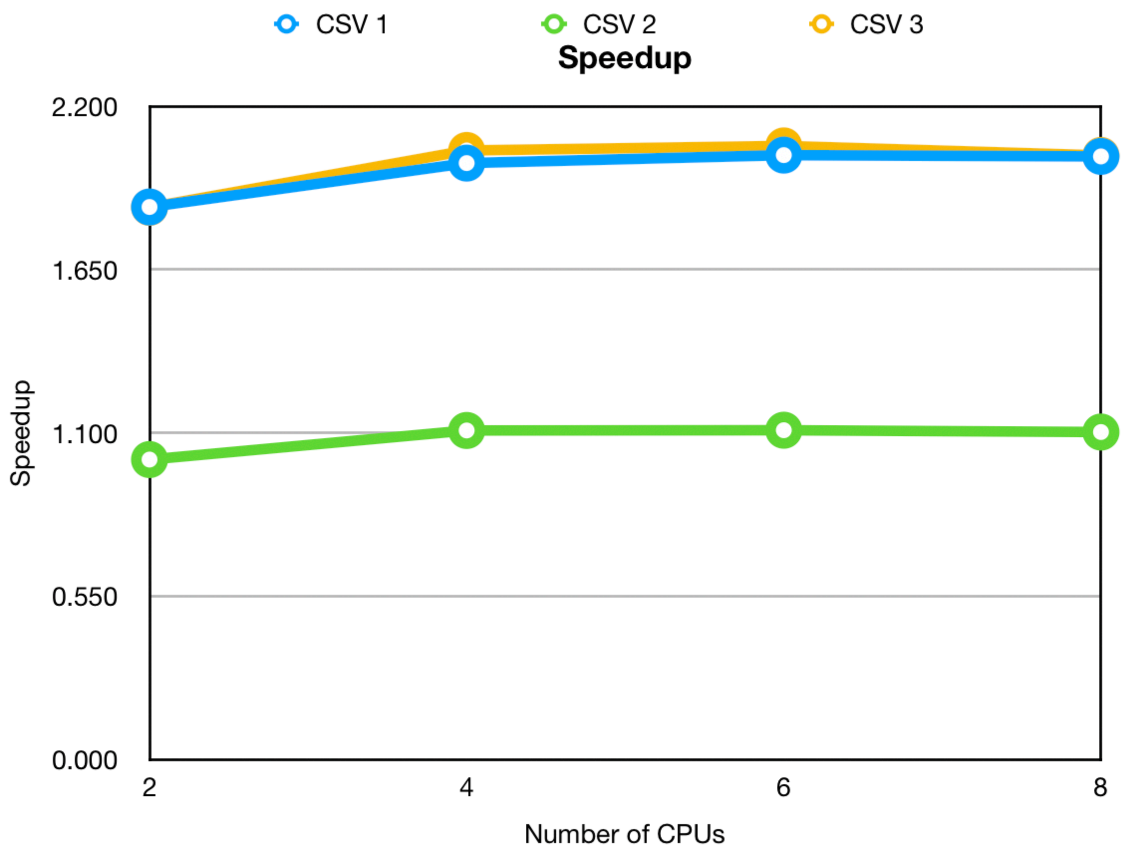
All tests were run using a single shell script which ran the tests in order and outputted the timings to a text file. A csv file from each of the three arrangements was passed to the editor along with the number of threads, unless omitted.

To create an average, each test type was run 5 times. Because a single thread and sequential operation were programmatically identical, the "-p=1" option was omitted from testing.

4 Questions

– What are the hotspots and bottlenecks in your sequential program? Were you able to parallelize the hotspots and/or remove the bottlenecks in the parallel version?

It's clear from the speedup graph that although speed up was achieved over the sequential version, the number of threads had a negligible effect on the speed of the computation. It is my guess that requiring a lock for each layer update caused a bottleneck. To explain the speedup over sequential performance, perhaps the pipeline allowed one thread to compute the convolution while another was updating, thus saving some time, but additional threads showed no further gains.



I believe just by the shear size of the computations involved (m x n pixels for an image by 9 pixels per convolution effect), that the convolution caused the greatest hotspot. Although I did split up each image's color layer, I did not split up the layer. If I could parallelize the computation required for an entire layer, I believe the gains would be measurable.

The reason for the reduced effectiveness of the speedup for the second problem set, CSV 2, escapes me.

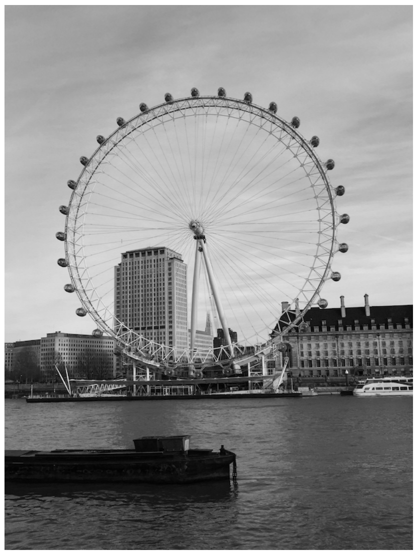
– Describe the granularity in your implementation. Are you using a coarse-grain or fine-grain granularity?

Because I was seeking both functional and data decomposition, I chose a fine-grained granularity for my parallelized implementation. In order to allow multiple threads to operate on the same image, over several effect on the same color layers, I need a way to protect the color layers image data.

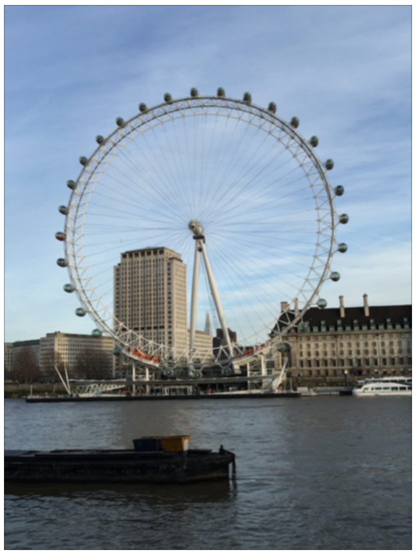
Without locks, a process thread could be computing the blur effect on an image's red layer while another thread was computing the gray effect on that same layer. The result was an image split between different effects, sometimes appearing half gray and half blurred, with colored stripes in the middle.

By locking each color layer every time it was updated, that insured that the image data remained in tact across threads. Although clearly slower from the previous graph, the results visual results achieved were admirable:

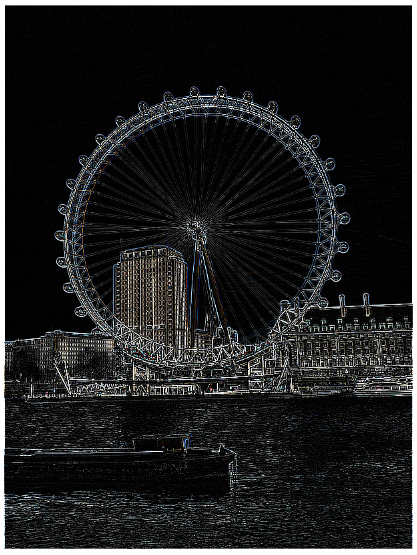
Grayscale:



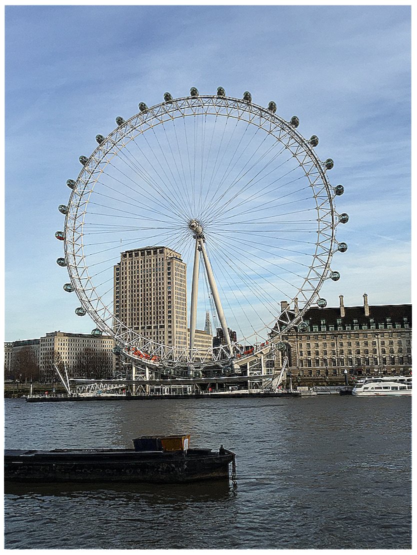
Blur:



Edge Detection:

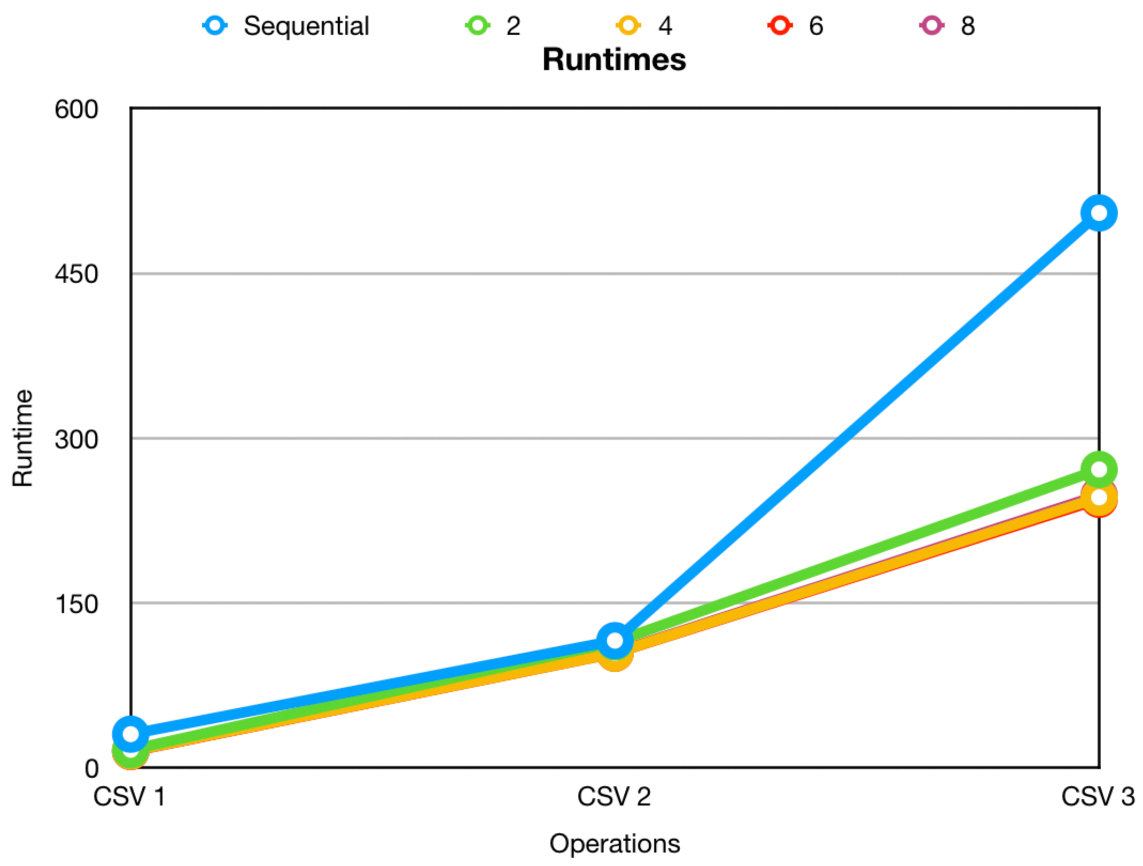


Sharpen:



– **Does the image size being processed have any effect on the performance?**

Tests were ran on three collections of images and their convolutions. CSV 1 constituted images with a rough size of 1 MB. CSV 3 contained the same effects as 1, but with images between 15 and 20 MB in size. CSV 2 added four more effects, and included 4 files that were larger in size, but was otherwise identical to CSV 1.



As demonstrated by the graph, CSV 3 with the larger files had a significant effect on run time. Likewise, the effect of image size was noticeably more pronounced in the sequential version.