Parallel Programming
Spring 2019
Yuqin Wu

Project 2 Report

In this project, I basically implemented a concurrent images effects application. This application can read in a png image file, and apply 4 different effects to the image based on the input parameter. The four effects are grayscale, blur, edge detection, and sharpen. User can use the capital letter to indicate which effect to apply. They will be applied in the given order, and store the result image into user specified address.

In the application, the user input is simulated as a csv file with images. The format of the csv file is:

imageAddress,storeAddress,[effects...]

Usage of the script:

./editor [-p=[num of threads]] <csv file>"

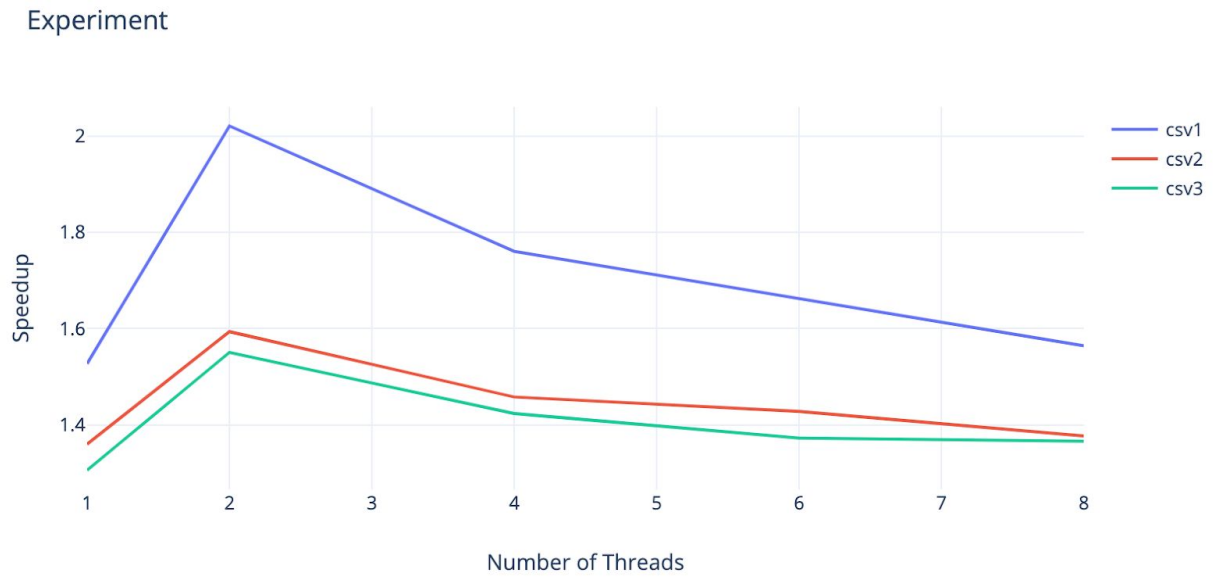
I implement this in two version, one is sequential and the other is parallel. For sequential, it is simply execute and apply each effect in order. For parallel version, I use both functional decomposition and data decomposition to speed up the processing time. First of all, There is a main thread creating the user specified number of worker threads. The threads immediately wait before the main thread start reading the csv file. After creating all worker threads, the main thread start reading the csv file and load the image and the corresponding effects, then push the image and effects into a shared queue. It also store the storeAddress into a local array because this is unnecessary to share. Each time a job is pushed to the shared queue, the main thread will broadcast all worker threads to start working. Each time the worker threads is done with a image, the last thread done it will push the image to a shared result queue. The main thread, after finished reading csv file, will begin check this result queue. Once there is a finished image, this main thread will store it to the storeAddress. This part of job is entirely concurrent with the worker threads processing the image. This is the functional decomposition part of design.

For the data decomposition, it is implemented inside these worker threads. Each worker thread has an ID number which also indicate the order of the worker thread. Each worker thread will only process 1/n size of the image, where n is the number of the worker threads. In order to simplify the calculation, the image is vertically splitted into n parts. For example, if the image width is 100 units and there are 4 worker threads, worker thread with ID = 0 will process the image with width from 0-25 unit, and worker thread with ID =1 will process width from 26-50 unit and so on. This is done by pass the ID and total number of threads to the processImg() function. The processImg() function calculate proper portion of the task to each worker thread. The calculation is carefully designed to handle the non-divisible case and out of bound case, with a math ceiling function and a min function. There is two barriers in the algorithm. The inner one is to ensure all threads will together finish one effect and then begin the next effect. The outer barrier is to ensure all threads will together finish one image and then begin the next image. This is the data decomposition design.

This report aims to provide and analyze the performance of the result of the experiment on different number of threads and three different datasets. This experiment contains the following test cases:

Sequential with csv1/csv2/csv3 datasets
Parallel: the combinations of csv1/csv2/csv3 three datasets and 1/2/4/6/8 worker threads, which has total 15 cases.

We will going to experiment the speed up for parallel programming on this machine(my laptop):
Processor Speed: 1.2 GHz
Number of Processors: 1
Total Number of Cores: 2
L2 Cache (per Core): 256 KB
L3 Cache: 4 MB
Memory: 8 GB



As we can see, first of all, the speedup is obvious. All 15 data points have a speedup larger than 1. This means the parallelism indeed work in this project. Then we focus on analyzing each single line. Horizontally, all three lines shows the same pattern: reaching maximum performance with 2 threads and decreasing with more threads. We can see that the speedup increase drastically from 1 worker thread to 2, but slowly decrease from 2 to 8. This is because my laptop has only 2 cores, so when 2 worker threads are assigned, the performance will be the optimal. With more thread, there will be more context switches, adding the overhead. We also can notice that, even though the speedup decrease as the thread getting more, it does not, at least when 8 threads, fall below the 1 thread condition. Therefore, we may conclude that it is better to have more threads than just one worker thread.

Now vertically, we can see that the speedup goes down with the increasing of task size. However, each dataset is about 13.6MB, 74.3MB, and 159MB. As we can see, even though the size is doubled, the speedup didn't decrease that much. We may conclude that the speedup is reaching to a stable condition, though this may need more tests to confirm.

– 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?

The hotspots is the image process part. It takes the most time to process. While the bottleneck is that only after one image is finished processing, it start to read the next image. These two parts are really suitable to make them parallel. I was able to parallelize them using functional decomposition to read csv file and processing image, and use data decomposition to process image concurrently. Methods are stated above.

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

I think my implementation is a fine-grain granularity because each time one effect is done, all threads have to wait for the last one to finish. This can be regarded as a type of synchronization and it is a little bit frequently. In addition, there is a somehow "dynamic" load balancing. The task are evenly split based on the number of threads.

– Does the image size being processed have any effect on the performance? For example, processing csv files with the same number of images but one file has very large image sizes, whereas the other has smaller image sizes.

Yes, apparently, large image size will incur longer processing time because there will be more pixels to compute for each image. The work of computation proportionally increase with the size of the images**.** My experiment results also proved this. The processing time increase from seconds to minutes with the same amount of images, which is 10.