Parallel Image Processing Project Report

Ruochen Wang

Description of the project

In this project, I implemented an image editor that applies image effects on series of images using 2D image convolution.

In png package, ImageTask struct is defined in png.go, and effects.go provides various functions to apply sharpen, edge detection, blur, and grayscale filtering effects to the image based on 2D image convolution.

Scheduler package provides sequential and parallel (pipeline, BSP) ways to implement image processing jobs on series of images. The program will read images from a series of JSON strings, apply the effects associated with an image, and save the images to their specified output file paths. Specifically, the pipeline model implements the fan-in/fan-out scheme. ImageTaskGenerator produces ImageTasks and dumps them into an imageTask channel, workers try to grab ImageTasks from the channel. A worker is solely responsible for performing all effects for one image and it will spawn mini-workers which apply effects on a slice of the image. A resultsAggregator aggregates the ImageResult from workers' own ImageResults channels and returns a single channel of ImageResult structs, the ImageResult will be saved into the output path. For the BSP model, 1 image and 1 effect is implemented in one super step, each worker applies the effect to a slice of that image. If all workers finish their work, this super step is finished, then they go for the next super step. scheduler.go provides configuration, and acts as the entrance of the program.

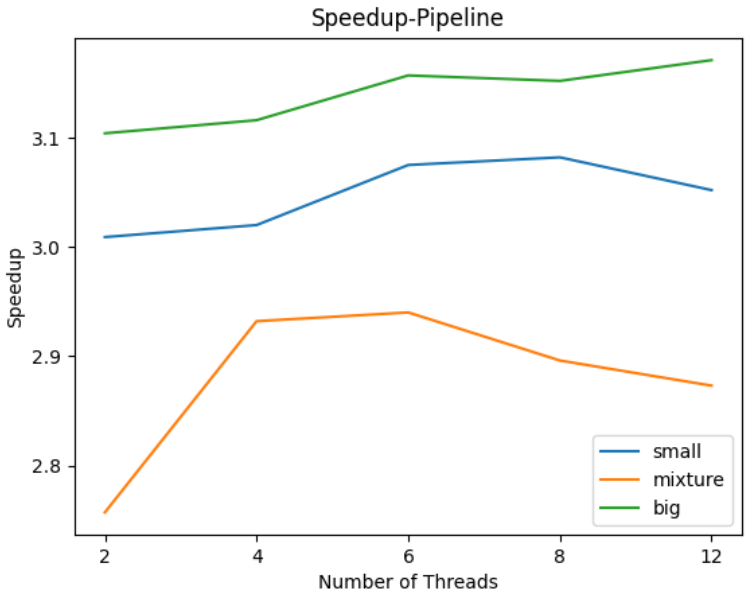
How to run testing script

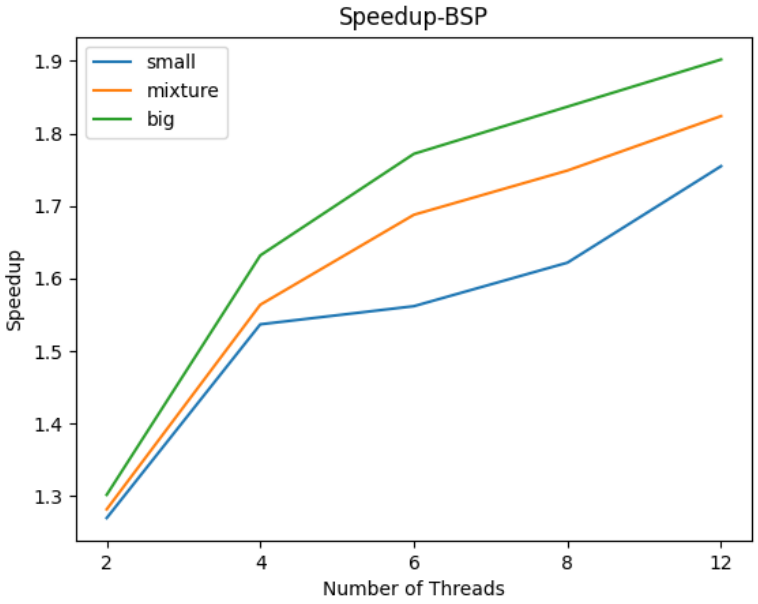
To run a single job, go to the scheduler directory and run scheduler.go, which is the entrance of the problem (go run editor data_dir [mode] [number_of_threads]).

To create speedup graphs, go to benchmark directory, and run the runs.sh file. runs.sh does experiment, stores results in speedup_pipeline.csv and speedup_bsp.csv, and calls makeGraphs.py to make speedup graphs based on results in csv files. After running the runs.sh file, you will find speedup_pipeline.csv, speedup_bsp.csv, speedup-pipeline.png, and speedup-bsp.png in the benchmark directory.

Performance analysis

The speedup graphs I generated are as follows.





In the sequential program, the hotspots and bottlenecks are likely to be the image loading, image processing operations, and image saving are all performed sequentially one after another, which lacks work division and fails to maximize the use of computing resources.

In my experiment, the pipeline model performs better. Firstly, in pipeline model, ImageTaskGenerator goroutine produces ImageTasks, resultsAggregator goroutine aggregates the ImageResult and saves images, they both work in parallel with the workers. While in my BSP implementer, the master goroutine is responsible for task generation and image saving. During this period, all workers are in the condition of waiting, causing a waste of resources. Secondly, I think pipeline model has a larger parallelism degree. The pipeline model has more workers to perform image processor jobs than BSP model. Pipeline model

has n works and each spawns n mini-workers, so it's n^2 workers in total, while my BSP model only has n workers.

The larger problem size results in better performance in BSP model, the speedup increases as the problem size increases, while this trend is not obvious in the pipeline model, speedups are similar among different problem sizes and they don't have specific relationship. I think one possible reason is that in my BSP model, the sequential portion of the larger problem size is smaller than small problem size. In my BSP implementation, task generation and image saving are not parallelized because they are done by the master goroutine and during this period workers are in the condition of waiting. In a larger problem size, image processing takes longer, so the time portion of task generation and image saving are smaller. However, the pipeline model does not encounter this problem, because task generation and image saving are processed in parallel with image processing, the model is well parallelized.

The performance measurements will be different if Go runtime scheduler uses a 1:1 or N:1 scheduler. With a 1:1 scheduler, each Go routine would be executed on a separate operating system thread. This would increase the overhead of thread creation and management, especially when dealing with many goroutines. It could potentially result in decreased performance due to excessive context switching and resource contention. With an N:1 scheduler, multiple goroutines would be mapped to a single operating system thread. This would limit parallelism and concurrency, as only one Go routine can execute at a time on a given operating system thread. It may lead to suboptimal performance, especially when dealing with CPU-bound tasks that could benefit from parallelism.

Potential improvement

I find 3 areas in my implementation that could hypothetically see increases in performance. Firstly, I think we can make workers finish applying all effects to its slice of the image together instead of waiting until all goroutines finish applying effect e1 to begin applying e2. Since each worker is responsible for a different slice of image, this change will not introduce safety issues but can reduce synchronization between goroutines. Secondly, in the pipeline model, I think we can have sharpen worker, edge detection worker, blur worker, and grayscale filtering worker instead of general workers to perform specific image processing task respectively. Once an effect is applied to one image, it will be sent to another channel, and add the next effect by another worker. This increases operations in the pipeline model, which helps take advantage of the pipeline pattern, and increases degree of parallelism. Thirdly, we can introduce a work-balancing mechanism between goroutines to avoid waiting when a worker finishes applying the effect on its own slice of image.