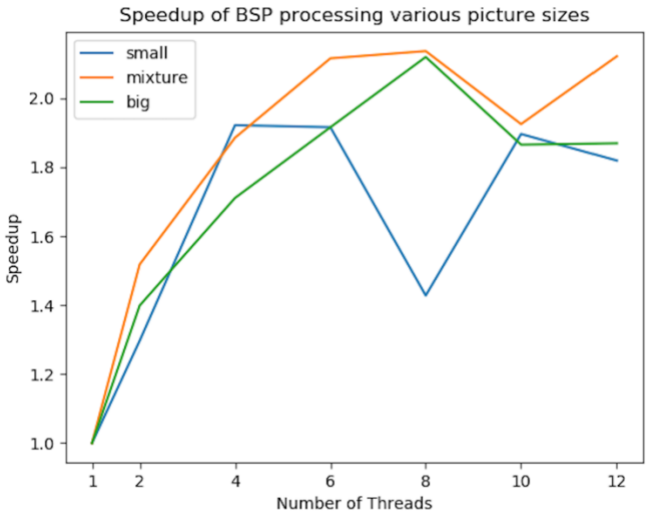
This project creates an image processing system with three different implementations – sequential, pipeline and bulk synchronous parallel model. The image processing system allows users to apply filters to images, and the filters include “sharpen”, “edge detection”, “blurring” and “grayscale”. The sequential version applies the filters onto images with a sequential convolution algorithm with no parallelism.

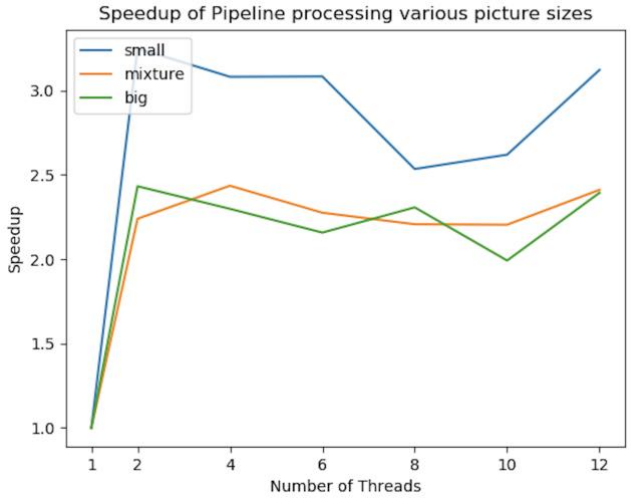
The pipeline implementation deploys a fan-in-fan-out pattern to parallelize the sequential. A user-defined number of threads are spawned to process pictures. A thread is responsible for processing a complete picture, and it completes a request by spawning additional goroutines to work on subdivided portions of the picture. These additional goroutine share equal amount of work. The pipeline implementation utilizes channels to communicate between goroutines and coordinate synchronization. It first spawns a generator that reads in the json file and creates tasks to be put in the channel task queue. The worker routines then grab jobs from the task queue and process jobs. An aggregator then synchronizes all the worker and combine their results to produce desired images.

Bulk synchronous parallel implementation is similar to the pipeline in that they both use parallelism to divide each image task into smaller subtasks. However, BSP contains a fixed number of threads that all go through a super step and global synchronization. There is no generator or aggregator as these tasks are performed in the global synchronization step.

You can either directly run command lines or run sbatch benchmark-proj2.sh in the benchmark folder to test the program.

To evaluate the performance of the program, I wrote two python scripts and a sbatch request to record the runtimes of the program. Each implementation is run with 2, 4, 6, 8, 10 or 12 threads for 5 times. The average runtime is calculated and the speedup compared to sequential version is plotted in the graphs below.





BSP speedup increases to around 2 as thread number increases to 6, after which the speedup plateaus around 2 regardless of picture size. This eventual plateau is expected as BSP has a global synchronization step which create a bottleneck: no matter how fast each super-step is, the global synchronization step dictates the fastest speed-up as the global synchronization step takes constant time, can't be parallelized, and has to happen when all goroutines are paused.

Pipeline speedup, on the other hand, seems uniform across thread numbers. The speedup for big pictures with 12 threads is about the same as the speedup with 2 threads, which is around 2.5. This also make sense because the pipeline implementation creates a total of N*N goroutines for each run, where N is the command line input thread number. In other words, when you put 2 in the command line, the program actually spawns a total of 4 goroutines. This quickly exhausts physical cores and the advantage of physical concurrency, and the context-switching has to be coordinated by the scheduler to handle application concurrency. This scheduling and context switching overhead cannot be parallelized and become the upper bound of the speedup graph.

In the sequential version, the hot-spots and bottlenecks are mainly in the convolution step where multiple calculations have to happen for each individual pixel. This is the slowest and hottest step. Pipeline performs better than BSP. This is because pipeline maintains a relatively more fluid structure, less wait time and less bottleneck. While threads have to remain idle for global synchronization in BSP, threads do not need to wait idle in pipeline as long as task stream is not empty. More work is continuously being done and less wasted resources in Pipeline. The problem size affects the performance in pipeline but not in BSP. As mentioned before, in BSP, the main bottleneck is the globalization step, which is independent of problem size. However, in pipeline, small problem size shows the greatest speedup because the sequential version is quite slow and pipeline speedups the process more dramatically than other problem sizes.
The performance measurements will be different if a 1:1 or N:1 scheduler is used. If 1:1 is used, the overhead of creation, context-switching and deletion are a lot higher, we won't be able to get as good of a speed-up in pipeline, but the plateau will be the same.

We should be able to expect a similar speedup in BSP with 1:1 scheduler, because only a limited fixed number of threads are spawned and maximum concurrency is allowed by kernel threading. If N:1 is used, we shouldn't be able to see any speedup since there is only a single kernel entity back the N threads and there will be no parallel execution of threads.
Based on the topics we discussed in class, both implementations can potentially benefit from a few changes. For example, in the pipeline implementation, the generator creates a bottleneck as all workers must wait for it to generate tasks, or else they will be idle waiting for jobs. If we provide additional parallelization at the generator step, we can reduce this bottleneck. The same goes to the aggregator step. In the BSP implementation, too many tasks happen in the global synchronization step. We can potentially parallelize the synchronization step by a new thread and allow the rest of workers to keep moving along the request list. Additionally, instead of using mutex, it can benefit from a more sophisticated lock-free algorithm.