Project 3 Report

Overview: In this project, I deployed parallel algorithms to solve two computation problems. In the first part of the project, I revisited the picture filtering problem from Project 2. Specifically, I implemented stealing and workBalance algorithms with double ended task queues to parallelize the filtering algorithm. In the second half, I implemented an additional feature named “MapReduce algorithm” to solve the “Single Source Shortest Path problem” along with the executorService technology.

Part 1:

The directories are set up similar to Project 2.

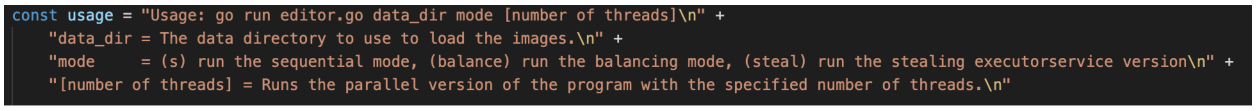
The concurrent folder contains all the concurrent functions, including both stealing and workBalancing executorServices. The unbounded double ended queue is also implemented here in the concurrent folder. Specifically, the unbounded double ended queue is a linked list, where each node is dynamically allocate. The queue struct stores the top and bottom nodes, the size of the queue and a mutex lock, whereas each node contains the Runnable or Callable task, the previous and next node. The struct has three methods: PushBottom(), PopTop() and PopBottom(). PushBottom() is used to add tasks into the queue, while PopTop() and PopBottom() are used to extract tasks from the queue. The locking mechanism here is a coarse-grain method.

In stealing.go, you can find my implementation of the stealing executorService. Each thread has access to its own DEQueue for Runnable/Callable tasks, which it can access through PushBottom and PopBottom. After a task is submitted by the user, the task gets pushed to a thread at random by the executor scheduling feature. After the thread executes a give task, it updates the future which is created during task submission. Idling threads that have empty DEQueues will attempt to steal tasks from other threads one at a time by using PopTop() to access the victim threads. This way, the competing thread and the queue-owning thread access the same queue from different ends.

In balancing.go, you can find my implementation of the balancing executorService. Each thread has the same setup as in the stealing algorithm, except the work balancing features. The fewer tasks each thread has, the more likely this given thread will initiate a work balancing process. The work balancing initiator thread will pick a victim at random. If the victim has over a threshold amount of tasks more than the starving thread, the algorithm performs a balancing step that equalizes the number of tasks in these two threads.

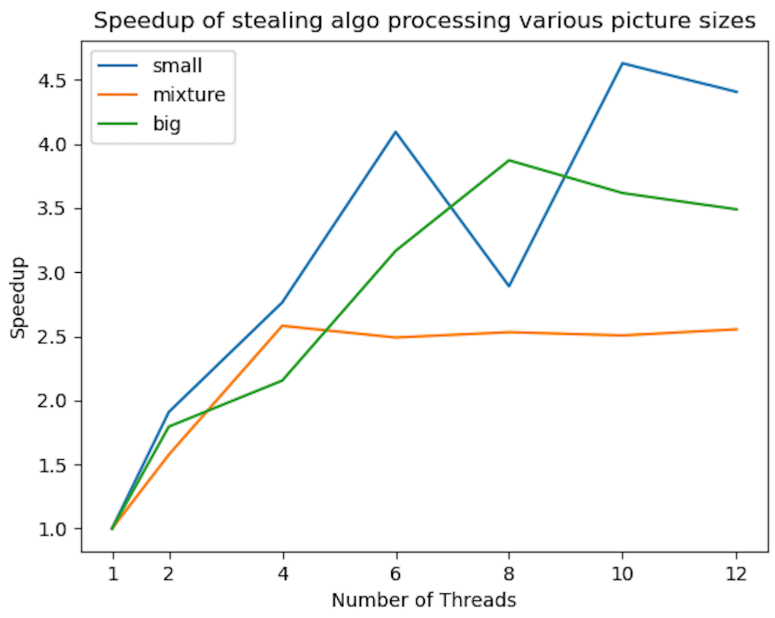
In the scheduler folder, you can find the files associated with the image filtering problem. sequential.go contains the sequential solution to the problem. executors.go contains the parallel solution to the problem. In executors.go, each task is a Runnable, which gets submitted to the executorService for scheduling and execution. Please note two important distinctions here from Project 2. You should only process images from one data directory at a time. Secondly, instead of dividing an image into multiple subtasks and assign them to different mini-workers as in Project 2, each image is considered a standalone task here and gets processes by one thread.

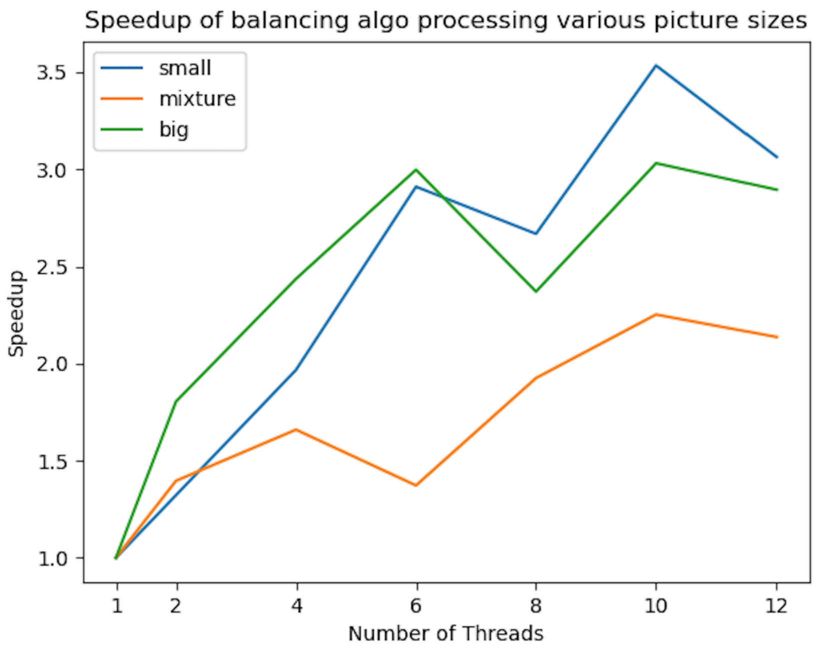
To test the functions, you should import the data directory as in Project 2. The editor folder contains the main function and you can utilize the filtering program by following the following usage:



```
const usage = "Usage: go run editor.go data_dir mode [number of threads]\n" +
    "data_dir = The data directory to use to load the images.\n" +
    "mode     = (s) run the sequential mode, (balance) run the balancing mode, (steal) run the stealing executorservice version\n" +
    "[number of threads] = Runs the parallel version of the program with the specified number of threads.\n"
```

You can also benchmark the algorithm in the benchmark folder. Running **“python image_filtering_time_experiment.py”** will run the program with various threadcounts and data sizes with both stealing and work balancing algorithms, and it will output the runtime. Next, you can run **“python3 output_reader.py”** to calculate the speedup and create beautiful plots as show below.





The above tests were run on MacBook Pro 2017. It has one Dual-Core Intel Core i5 Processor and 2 physical cores with embedded hyper-threading technology. It has 8 GB of memory. We see linearly increasing speedup with increasing number of threads in both versions. This well-behaved speedup is observed with big images. We reach a speedup limit with small images after the thread number is more than 8. This happens because the parallel portion reaches the maximal speedup around 8 as the parallel processing time of small images is not so long. The sequential portion of the program, such as task generation, and increasingly more synchronization overhead during work redistribution outweighs any marginal performance gain after thread number exceeds 8.

As the queue-owning thread and the competing thread access the same queue from opposite ends of the queue during work redistribution (via PopBottom() and PopTop() respectively), theoretically we should not have any contention there. However, due to our coarse-grain locking mechanism, both PopBottom() and PopTop() need to access the same lock, creating a potential hotspot. Additionally, PushBottom() would need to access the same lock too, meaning tasks cannot be added to and extracted from a queue at the same time. The task generation step is also a bottleneck in the problem, as it happens prior to task submission. This coarse-grain locking hotspot is unable to be removed as it is specified in this project write-up. The task generation bottleneck also cannot be removed as it needs to be sequential to provide all necessary components for the task.

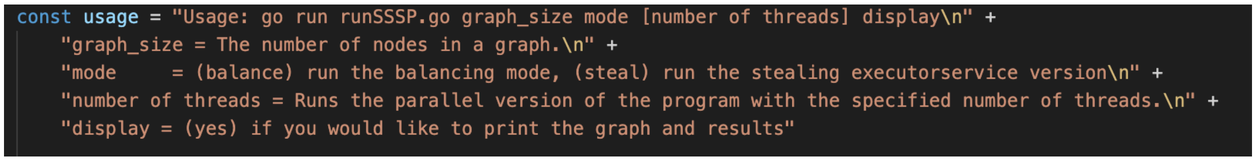
Lastly, we observe numerically better speedup with work-balancing algorithm than stealing algorithm. This is possibly due to the fact that the work-balancing algorithm grabs multiple tasks from the victim thread while the stealing algorithm can only grab one task at a time. This means that the work-balancing algorithm triggers the work redistribution step less frequently and only when the threshold is reached, while the stealing algorithm can trigger work redistribution more often as it only grabs one task even when it’s empty already. Therefore, it is safe to hypothesize that excessive work redistribution steps decreases the performance and limits the speedup due to excessive locking hotspot as discussed earlier.

Part 2:

In part two, I implemented additional feature named “MapReduce” to solve a graph problem — “Single Source Shortest Distance”. The associated code can be found in the mapReduce folder and the main function is in the SSSP folder. The MapReduce implementation spans two files — framework.go and app.go in the mapReduce folder. In framework.go, I provided the framework for mapReduce, the struct and the associated methods. We can set the input, mapperSupplier and reducerSupplier to the mapReduce struct. The Call() method then executes the mapReduce, producing the desired result.

The app.go file applied the framework to solving our problem. The input list to mapReduce is a list of nodes that are known to be reachable from the source. The algorithm then creates a mapper for each node in the input list. The mapper outputs all the nodes reachable from the inputNode and the distance of these nodes from the source node via the inputNode. The algorithm then combines all the mapper outputs into an aggregate map where each key is the target node and the value is a list of distances from the source node to the target node via all possible paths. We create a reducer for each target node and the reducer output the shortest distance to reach this target node. The algorithm is run iteratively, as iteration adds more reachable nodes to the input list. The algorithm terminates when the newly calculated distances are no more than epsilon less than the old distances. mappers and reducers are both callable structs, and they are submitted to an executorService created by user preference either stealing or work-balancing.

To run the program, please cd into the SSSP folder, which contains the main function, and follow the usage instruction.



```go
const usage = "Usage: go run runSSSP.go graph_size mode [number of threads] display\n" +
    "graph_size = The number of nodes in a graph.\n" +
    "mode     = (balance) run the balancing mode, (steal) run the stealing executorservice version\n" +
    "number of threads = Runs the parallel version of the program with the specified number of threads.\n" +
    "display = (yes) if you would like to print the graph and results"
```

For example: “go run runSSSP.go 10 steal 2 yes”

This program first creates a directed graph by building a random adjacency matrix of the input graph size. The program then calls the mapReduce functions to solve the graph.

To benchmark the program, you can also cd into the benchmark folder and run “python mapReduce_time_experiment.py”. This runs the SSSP program with various graph sizes, thread numbers and executors, and it outputs the runtime. We can then use “python3 mapReduce_output_reader.py” to calculate the speedups and plot them.

I did not observe any significant speedup in this program. This is due to the fact that our problem here is a toy example. Each mapper and reducer is only a simple calculate or comparison that takes negligible time to run. However, we have a huge number of mappers and reducers in this program. For each mapper and reducer, we need to generate them, submit them, and aggregate them all sequentially. This means that the actually parallelized step is very small and only a fraction of the sequential step. And the sequential step is further complicated by the number of reducer and mappers here. The sequential overhead here is simply way to big to show any advantage provided by our mapReduce feature.