Proj3_Report

How to run

From proj3/ directory run

```
sbatch run_analysis.sh
```

Outline

My program helps you make assumptions on how a stock will do at a future point in time. For example, it helps you predict what the probability is of the SNP going up by more than 5% in the next 3 days.

The query parameters are:

- stock symbol
- number of days into the future
- threshold (up or down %x)
- number of montecarlo iterations (the larger the iteration the more 'accurate', but slower)

Useful Information

- JSON response from the API endpoint can be found in data/rawdata
- as long as the stock symbols are listed in the rawdata file, you can modify the values in the forecaster/input.txt to submit your own requests
- all responses are to the requests are collected in forecaster/output.txt
- in log/app.log you can see the log outputs from the server running, this may help you see what is going on behind the scenes.
- Here are some examples to test single runs. Make sure you are in the proj3 directory!

```
go run forecaster/forecaster.go p 2 w < forecaster/small_input.txt
go run forecaster/forecaster.go p 2 < forecaster/small_input.txt
go run forecaster/forecaster.go s < forecaster/small_input.txt
```

Project Details

Inside the program, this is what happens:

1. Checks database to see if stock data is up to date → construct the pool of data from which to run monte carlo
    a. If not, it will make an api request to pull data (daily movement of a stock, refreshed up to today's date) which will give information such as open, high. low, close values for each date for the past 20+ years.

b. The response comes in a json format, which will be cleaned to return the daily movement **(close-open)/open *100** for each date so that we have a large pool of data on the daily movement of the stock

c. From this pool of data on daily movement of a stock, we calculate the cumulated change from the request <insert function for that> find the mean and standard deviation, which allows us to construct a normal distribution on the sample. I have already loaded some sample data for the purposes of this project, this will be available in data/rawdata

d. Sample is used to construct a monte carlo simulation,

   i. A random number is chosen from a uniform distribution from 0 to 1

   ii. Use the norm inverse function on the normal distribution we created in the previous stage.

2. Run monte carlo

   a. Each request (as seen in proj3/forecaster/input.txt) is formatted in json and includes query parameters described above.

   b. In the producer thread, these requests are packaged into individual tasks, where each tasks runs a db check outlined in part 1

   c. **[Parallelized]** Each task specifies the number of montecarlo simulations to run, and this number is typically very large. We simulate a large pool of data, then get the mean and stdv from that dataset in order to find the probability that the daily movement will cross the hypothesis threshold.

3. Return results

   a. For each task, we run the procedure and return the calculated probability along with the original request in json format. This can be found in proj3/forecaster/output.txt.

Parallelization

I used parallelization to (A) process each request as a task by splitting up the work in different threads, then (B) again when running the montecarlo simulation in each thread.

(A) : Use producer/consumer model to package each request into a task and send into MotherQueue. The primary thread will pop from the back, while thiefs will steal from the head.

(B) : check server/server.go line 127-129

Motivation for pipelining

1. Given multiple requests, each can run concurrently so by distributing it across multiple threads we can help balance the load of the total task and make efficient use of resources

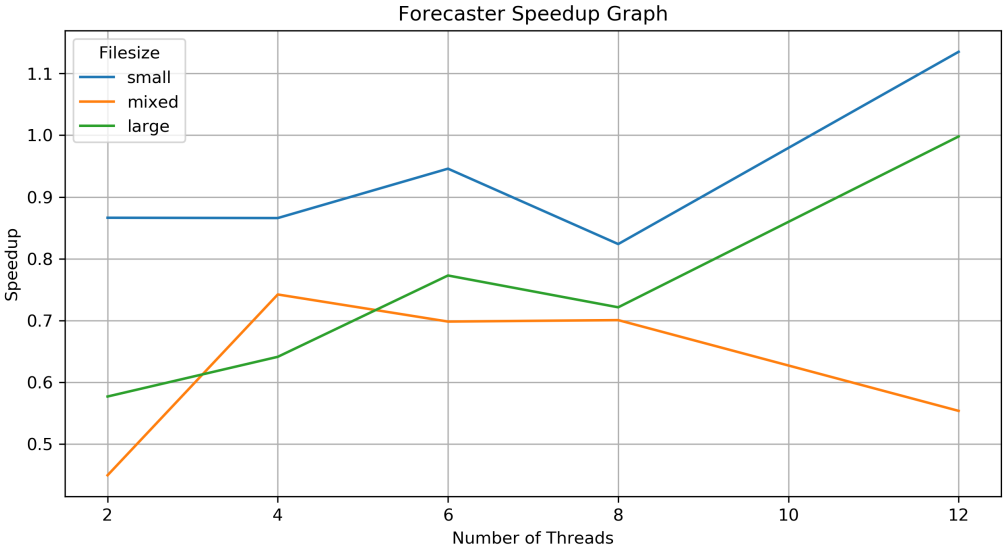
2. Pipelining can significantly improve throughput by allowing multiple stages to process concurrently. This reduces the idle time for each processing unit and ensures that the overall latency from input to final output is minimized.

Challenges and Limitations

A **bottleneck** that I faced was to find an efficient way to calculate the variance of the dataset. Once I had the pool of datapoints, I first had to calculate the mean, then use the mean to calculate the distance from the mean for each datapoint in order to get the variance. This meant I had to run through the same stream of data twice. The alternative is to calculate the mean and variance in

batches then later adjust the difference in the variance per batch. But this is still using two runs to the database, so I decided to stick with the first method.

First, I spawned a single channel to create the simulations and send it off to the channel. A **hotspot** was generating during this stage, since these could be independently generated.



This is comparing the sequential version (1 thread) with a multi-threaded approach where there is no parallelism in the monte-carlo part of the simulation. In other words, the only difference is the dividing of the requests in the (approximately) N/T requests over T threads. In most cases, this implementation performed worse than the sequential implementation.

Then I tried spawned multiple processes to generate values and feed it into a single channel. This seemed to speed up the process by 30% over all file sizes. A further challenge here is that there was only one channel receiving the values, which could potentially be parallelized so that there are multiple threads receiving values which will then map all the values back to one big pool.

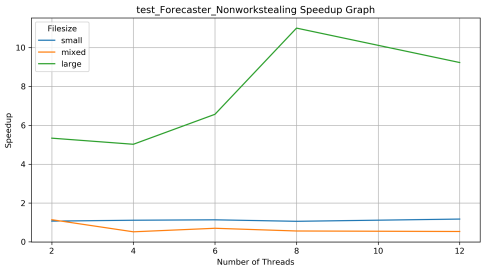
Another challenge was the size of the simulations. First I tried to run it with the following parameters:

[Test 1]

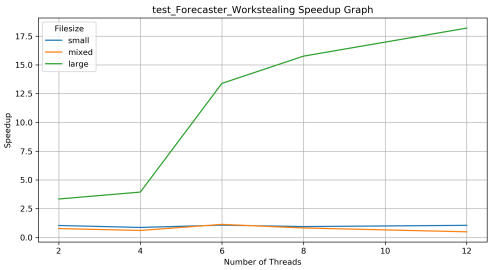
small: 50 iterations

mixed : 2000-1 million iterations

large: 1million iterations



Graph 1 : Singlethreaded vs. Multithreaded, both non-workstealing



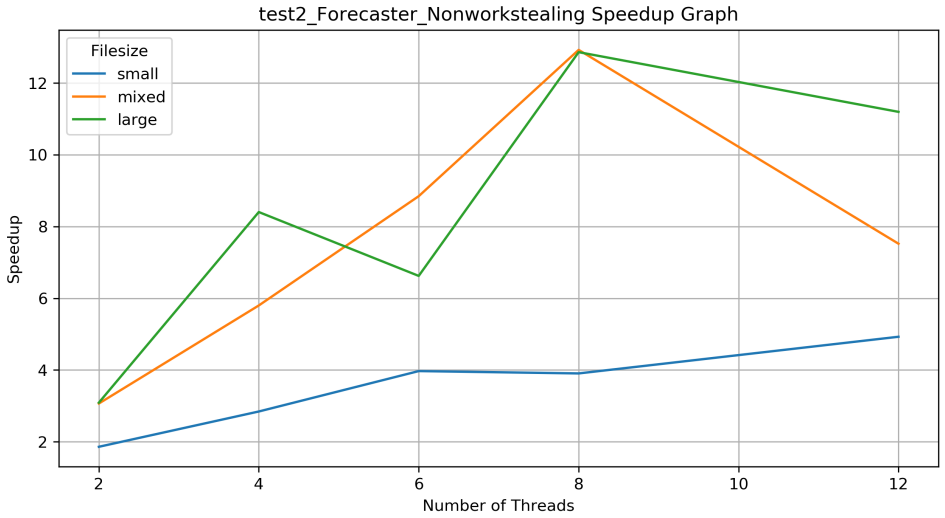
Graph 2: Singlethreaded vs. Multithreaded and workstealing

The small and mixed files had worse performance than the sequential version because the iteration sizes were too small so I increased the simulation size for both to get the final result, outlined in the next section.

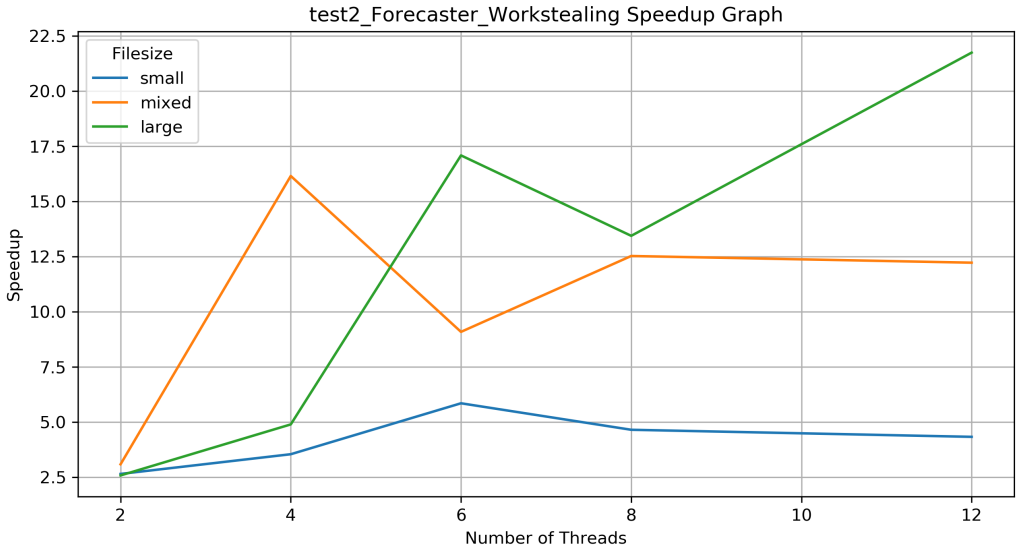
These yielded better performance, which better reflected the actual performance of parallelized monte carlo simulations, since the simulation sizes should generally be much larger than the Test 1 parameters.

Comparing Parallel Implementations

The difference bewteeen small, mixed and large is in the number of simulations the request is asking for.



Graph 1 : Singlethreaded vs. Multithreaded, both non-workstealing



Graph 2: Singlethreaded vs. Multithreaded and workstealing

[Test 2]

small: 5000 iterations

mixed : 100,000-1 million iterations

large: 1million iterations

When it comes to a large number of simulations, the speedup is significant for both. The workstealing case improves performance significantly, especially in large numbers of monte carlo iterations.

All simulations seemed to hit an optimal threshold after which adding more threads leads to worse performance. This is likely due to the bottleneck of having to process the simulation data pool as explained in the **Challenges and Limitations** section