ECE 421

Group Project 1: Stock Market Monitor

Tymoore Jamal
Andrew Williams
Glenn Rhodes
Micheal Antifaoff

Program Design

When using a stream to calculate the highest share price below a passed maximum value we first mapped the stock symbols to their ShareUtil class and retrieve the current price. The elements are then filtered keeping only stocks whose price is below the desired maximum value. The remaining stocks are then sorted by descending stock prices and the first stock (the highest price) is then found and the ShareUtil object is returned.

To accomplish this the stream functions used were map, filter, sorted, findFirst, and get. These functions were chosen as they were all stream safe and would accomplish our goal in a minimal set of steps and keep type casting minimal as well. Changing types within the stream were avoided after the initial creation of ShareInfo objects to reduce unnecessary work and allow the returned value to contain all information about the stock.

The APIFinance file was also modified to accommodate for the API limiting calls to 5 per minute and invalid stock symbols. If the API is not returning values due to the limit being reached the program will continue to send queries until the API returns a result. In the case where an invalid stock symbol has been supplied, an exception will be thrown notifying the user which stock symbol is invalid, then the program will then continue execution and return the highest valid stock under the desired price. In the case where a connection is not able to be made with the API the program will throw an IOException and notify the user of the failure sending request.

Known errors/faults/defects:

The API limits the number of calls to five per minute, this severely slows the execution time of the function as 10 prices are desired in the example used. This leads to waiting for the API limit to reset to allow more queries and will cause the program to scale very poorly since for every additional 5 values a minute of idle time will be required.

Additional Testing

Once our program began making API requests, we noticed a large degree of variability in how long the API required our program to wait before making additional requests. Our initial idea was to put the program to sleep for 60 seconds, then wake it up once the lockout period had elapsed. However, once we realized a full 60 seconds was not always needed, we instead designed our program to make API requests, then determine whether the stock price had been returned correctly. If the stock price had been provided by the API, we return the price, otherwise, we request the price from the API again until a price is provided. This uses slightly more resources than simply putting the processor to sleep, but often results in a much faster total program time.

Another form of testing our program performs is that it checks whether or not the provided symbol is an actual stock; if not, the program throws an exception and continues to get the price for the remaining symbols. If there is no internet connection available, or if the API is down for some reason, our program will throw an IOException to let the user know that an error has occurred.

Determining Execution Time for stream() and parallelStream()

In order to determine execution time, we ran our program first using a regular stream(), and then secondly using parallelStream(). Our results determined that parallelStream() is much faster in both execution/system time and in terms of total time.

Time and CPU Utilization for PickShareFunctional

PickShareFunctional Time	stream()	parallelStream()
User (seconds)	2.77	2.41
System (seconds)	0.31	0.22
Total (seconds)	57.297	10.276
CPU Utilization (%)	5	25

Execution time was highly variable as we ran the code multiple times. Even though we waited for a full minute before running the program each time, the API was highly variable in how many share prices it provided before enforcing a timeout for reaching the request limit. The API also enforced a varying lockout period: occasionally, it made the program wait for a full minute after performing five requests (as stated in the API documentation), but sometimes with parallelStream(), the program was able to get seven share prices right away, and after waiting just over a second, get the next three as well. This introduced a high degree of variability into our timing results, however, parallelStream() was definitely quicker because it was able to get results faster from the API, and it used a greater portion of the CPU in order to minimize execution time.

Question B

PickShareFunctional performs three main operations: first, it creates a stream of ShareInfo filled with the price for each of the symbols in Shares. Then, it trims down this list to a list of shares whose prices are under $500, before returning the highest-priced share. Out of these three steps, the step which takes the most time is step 1: creating the map which creates a list of ShareInfo filled with the price for each of the symbols in Shares. This step is time-intensive because it involves a call to the API, which is an external resource that requires network access and has a limit of 5 requests per minute. As a result, once 5 share prices have been acquired, getPrice() then continually makes requests while waiting for the API to respond with a valid price. During the request timeout period, the API returns a message notifying the user they have reached the limit for any request until the request timeout period has passed. getPrice() takes a significant amount of time because the API has a 5 requests per minute limit, so getPrice() must wait while the API has locked out requests. In contrast, filtering and returning happens very quickly because the system is able to perform these operations on device (and in the case of parallelStream(), at the same time using multiple threads). Since getPrice() is used to create the list of ShareInfo, and since getPrice() needs to wait for the API, creating a list of ShareInfo filled with the price for each of the symbols in Shares is the step which takes the most time.

Question C

Execution times between stream() and parallelStream() are different because while both use streams, parallelStream() makes use of the multiple cores found in modern processors today. stream() is very similar to a forEach() loop, where each element in the stream is run sequentially over one core. In contrast, parallelStream() divides the provided task into many different, smaller tasks and then, using multiple different threads, assigns tasks to each thread. The benefit of parallelStream() is that each of the multiple threads can be run on a different core, allowing us to make use of multi-core architectures and maximize the performance of any stream-based operations. This means parallelStream() will perform on device operations much faster than stream(), although parallelStream() requires more CPU resources than stream(). In our program, we found the parallelStream() variant of our code to have an execution time (or system time) of 0.22 seconds, while the regular stream() variant had an execution time of 0.31 seconds. While this is a fairly small difference, it becomes amplified as programs and data get larger; running programs in parallel allows us to maximize the use of our CPU while saving ourselves time in the process. This is one significant advantage of stream programming in Java: we can parallelize code just by changing stream() into parallelStream().

There are two additional reasons that the total time for parallelStream() is much quicker: with multiple threads requesting at nearly the same time, it appears the AlphaVantage API was too slow counting requests, allowing for multiple threads to get 6, or even 7, share prices before the API began to enforce its five requests per minute limit. Also, the multithreaded parallel version sends many more requests per second and it appears that this results in the API returning values before its one-minute lockout period has elapsed. For these two reasons parallelStream() has a much lower total time than stream().