Amazon Logistics Simulation

Now with optimized performance and 3.1 gigabytes of output!

I’m serious. I went a bit overboard on this, as usual.

I don’t think you can get much faster than I got for the simulation size (in java). More details in the report.

Efficiency Definition

Efficiency is generally defined as the percent performance relative performance to an optimum. However, determining the optimal performance of this system is out-of-scope for this project. Therefore, we instead define efficiency as the percent utilization of the trucks cargo space. Even an optimal system is unable to reach 100% on this metric. However, as systems grow more optimal, they do get closer to 100%, and as they get further, which means that it is a reasonable model for the efficiency. Obviously, a system could be designed with extremely good efficiency by this metric, but poor overall performance, by loading up the trucks and then sending them in circles: as such, we will also compare the number of ticks needed to evaluate each simulation run.

For the sake of having an actual comparison, I stepped outside the project boundaries to some degree, designing a system that allowed for more than one router class. Of course, at that point I went off the rails to some degree: more details on that piece of art can be found in the Code Structure section.

Performance Notes

A very noticeable fact about the simulation is that it is extremely fast: my most recent execution of it using my potato-based laptop finished in 6:17, only half of which was actually spent running code. That number should be fairly consistent on other machines (CPU scheduling and run initializations may cause some variation): each iteration tick is paired with a call to Thread.sleep() that ensures that the tick took at least a certain amount of realtime to execute. This number, set at the end of each configuration file, is quite small in the versions I am sending in. If you wish to see a version of the simulation that allows a human to actually see more than just teleporting dots, please increase that value (it is in nanoseconds). The smaller

simulations can be quite lovely to watch, when set to lower speeds: I highly recommend giving them a good look, especially the ones which use the smarter routing algorithm.

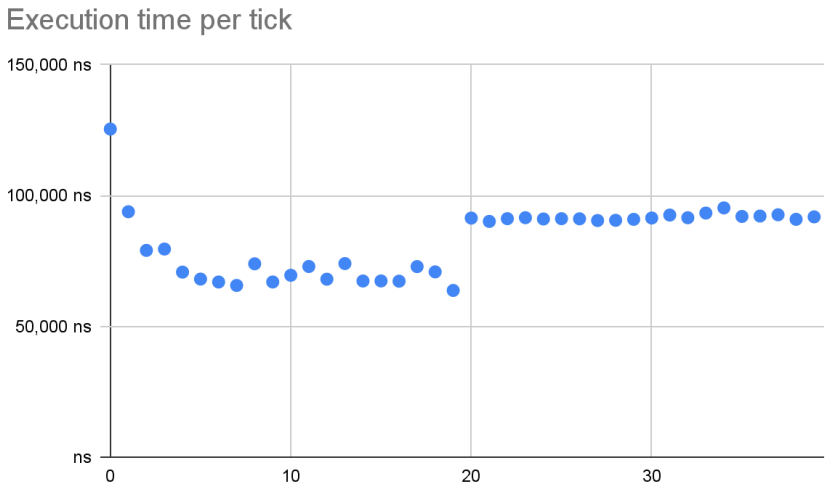
Of course, it isn't enough just to complete the project: I wished to challenge myself, to do harder things. As such, in addition to the fairly small and short "basic-config" and "faster-config", the files you received also include a "big-slow-config" and a "big-fast-config". These big configs each describe a simulation that is about as large as I think they reasonably should be: each truck is given 100 orders to fill (the completely serious maximum you specified in the FAQ), a canvas that is about as large as it can be without making the trucks invisible on my laptop monitor, and the same number of trucks. While adding more trucks is easily possible, doing so will substantially increase the computational complexity for each tick: further, since I forgot to do that for the data I analyzed, it wasn't done in the config files sent in with the project. I feel that changing the number of warehouses and the canvas size is sufficiently different between the two different configuration types.

Profiling

As part of the fun, I ran the simulation under the watchful eyes of a profiler. The results were enlightening: the slowest portions of the code were not the complex mathematics of calculating new points (though, admittedly, that is fairly simple in practice), or the graphics (which do most of their work on a separate thread) but rather the assembly and storage of strings! I describe 2 of the optimizations I did to reduce this cost in "code structure and optimizations", however, there are still major inherent costs to the operation. 15.7% of the CPU time is spent building the Truck status strings, compared to only 2.6% being spent actually carrying out the Truck actions, and 1.6% for both the action and the statuses of warehouses. Tight loops appear to have gone somewhat unoptimized: 14.5% of the CPU time is spent in the actual applyFunctionToList method itself.

However, all those costs are dwarfed by the library costs required for the graphics and logging portion. Writing files requires a whopping 36.5% of our CPU time: and that is despite trying to minimize that cost by assembling the string for each tick using the StringBuilder, a class that Java's JIT has shown it can optimize very effectively.. 16.9% of our time is spent just directing the graphics threads to repaint the screen: actually repainting it is a continuously occurring operation. These costs, while they could be reduced somewhat, are fundamental to the program requirements. The operation I was worried would be time consuming (the routing of packages) turned out to be a mere 1% of CPU time: indeed, the routing doesn't even appear on the profiler, as it is completely inlined!

This shows the importance of profile-guided optimization: had I wanted to improve runtime, I doubt I would have gone so far as to write my own rounding implementation for the toString() function of the Point class. Further, it shows some of the issues with Java's JIT: despite the deterministic nature of the program, there are substantial run-to-run differences in the percent of time consumed by various steps, and variation in what functions are inlined. Some functions, called with constant parameters, don't seem to be significantly improved. That being said, the JIT's effect can be easily seen: the following graph shows the per-tick execution time for all the runs. It is plainly visible how the initial few model runs are much slower, and how they are steadily optimized. It is also clear how the larger canvas, additional warehouses, and longer order queues impacts the model's execution speed: surprisingly little.



Code Structure and Optimizations

In the functional code review, you requested that I explain my use of lambda expressions in the report. This section is dedicated to that explanation, as well as a few other details of my implementation that I consider to be needlessly complex. This section won't go into the unholy mess that is the class graph of this program: that is a result of classes needed in many places (Point, DeQueue, and ShipmentOrder) all existing in the same package as little-used classes (Router and Executer). Additionally, I have spent some time running the code through a profiler: the optimizations I made as a result of that (and some consultation with third-party benchmarks) are described. Some of these notes impact many locations of many files, meaning that you might run into them without any comments explaining why. Other notes only impact one location in one file: as such, they are merely a more formal description of what motivated the unusual structure, rather than the terse and functionality-driven perspective taken on by the code's comments.

Use of Lambda Expressions

In the code, I make use of lambda expressions many times; 36, to be precise. At least 10 of those times are in the context of a call to the applyFunctionToList() method of the DoublyLinkedList class. This function is, in many ways, a replacement for a for-each loop: it applies the provided lambda to the data of every node in the class. Of course, it is very inefficient: most users of it don't really care about the data returned by the function, so the new list is assembled for no reason. More broadly, I believe that this method makes it impossible for the compiler to vectorize the loop that it describes: this slows it down, resulting in a fairly high runtime for the code-running method itself, rather than any actual calculations. This inability to vectorize stems from the inherently serial nature of appending to a linked list: writes that the compiler may not be able to identify as unnecessary. A better method would be to implement the Iterable interface: don't be surprised if you see custom classes with that interface in the next project.

The Router Classes and Their Use

As mentioned above, the Truck class is designed to be used with different Routers, that evaluate their routings in different ways. There are many ways to implement such a system: perhaps the most obvious is dependency injection, when the objects a class depends on are constructed separately and supplied to it through the constructor. Another, slightly less flexible method is to not use a named class at all, and just have a lambda function do the routing. Even passing a lambda function that produces Router objects would be a reasonable solution. I chose none of those: I decided that the simplest and cleanest route was to pass in a class object representing the router to be used, and then use an exception-throwing and certainly-not-intended-for-this-usage class that calls the specified constructor of the specified class. This method is a part of java's reflection (other languages use the far more accurate term

“introspection”) features. These are fascinating to me, and I decided I kind of to give them a whirl. From now on, rather than invoking methods or modifying variables directly, I will exclusively be using reflection to do so.

The use of StringBuilder

A StringBuilder is a member of the Java.lang namespace, however it is very specialized: it simply concatenates a series of strings (with additional insertion and removal features). One might assume that such a class would be slower than concatenating strings with +: however, that is not the case. Concatenating strings with + requires a memory allocation for each additional string: for non-constant strings, those allocations add up quickly. StringBuilder is capable of avoiding all such intermediate allocations and their subsequent garbage collection costs by re-using the same character array for multiple strings: instead of deleting the array, we just tell the Builder to ignore all the data past a certain point. We use this in Truck, where it builds up the status string, and again in Executer, where it combines them all together. For even more efficiency, we could return the StringBuilder, and not convert into a string until the end: however, that would break the API requested even more than I already have by converting status() from a void method that does the logging to a method that returns the string (A change that I think is very justified, given the cost of file.write() calls, which necessitated that I use the Executer StringBuilders in the first place).

The Point Class’s toString() method

This single method is one of the hottest spots in code. It is called by the Truck’s status() method, which is another extreme hot spot. I originally implemented it in a very simple and obvious way: using the Formatter class to assemble the strings describing each point. It was a two-line method: however, it was also consuming 15% of my CPU time, before I had even

started really calling it or using the values it returned. This was a substantial problem: it turns out that the Formatter class is abysmal when it comes to speed. Every call of toString() method would create a new Formatter, have that formatter parse the format string, and then assemble the string piece-by-piece. Even allocating the Formatter as a static object does little to alleviate this. My next choice was the DecimalFormat class: a far-more special purpose class, that does most of it's parsing up-front. However, this was still too complex for the JIT to optimize out completely: and even the minute cost of a function call and some number-twiddling adds up incredibly fast when it must occur numerous times for every object on every tick. Thus, I added a manual implementation of the string printing code. It is now no longer possible to measure the performance of Point.toString, because it has been inlined into its callers, reflecting a nearly-zero cost. As a final optimization, we ensure that points will remember their string representations, taking advantage of their immutable nature (while at the same time turning them into mutable objects). This final optimization reduces the number of times that the number twiddling must occur substantially: Truck.status() now only makes up a small part of the runtime.

Efficiency

The ultimate goal of this report was not to evaluate the runtime speed of the simulation, which I have done in great detail, but instead the efficiency. What difference does map size and router design make in the use efficiency of the trucks? We earlier defined efficiency of the simulation to be equal to the average percentage of truck cargo slots that were filled at any given moment: as such, let's examine a brief table that describes the system's performance

Config File	AVERAGE of Execution Time	AVERAGE of ticks evaluated	AVERAGE of cargo full percentage	AVERAGE of execution time / tick	AVERAGE of routing time / tick	AVERAGE of time spent routing
basic	513,929,177 ns	6,480 ns	14.40%	79,007 ns	145 ns	891038 ns
faster	182,476,449 ns	2,623 ns	49.92%	69,375 ns	1,755 ns	4527141 ns
big slow	5,196,689,171 ns	57,141 ns	14.43%	90,942 ns	74 ns	4252142 ns
big fast	2,033,060,646 ns	22,010 ns	62.76%	92,368 ns	1,957 ns	43029863 ns

This table describes how each configuration performed on each metric. As you can see, the runs using the BetterRouter class performed better on every metric except routing time than the BadRouter class. That includes metrics such as overall execution time! Because the BetterRouter is cable of accomplishing the same series of orders in about a third of the ticks, and status logging takes up so much time for every evaluated tick, the hundredfold increase in per-tick routing time is irrelevant! A small note: the gathering of this kind of high-granularity performance data of a real-world program is very difficult. Determining the precise time is difficult, and actually takes longer than the measured task for some of these programs. That means that our numbers for routing time can't really be relied upon for their precision and accuracy: it is likely that the true value for the BadRouter class is much lower, given that the "routing" is simply the fetch and return of a single memory address.

For our purposes, the BetterRouter makes more efficient use of its resources: while the BadRouter consistently is capable of filling about ⅛th of it's cargo units, the BetterRouter is able to fill half when there are 20 orders and 2/3rds when there are 100. This increase is due to the additional orders increasing the possibility that the truck is capable of picking up or dropping off multiple pieces of cargo at the same location. By our definition of efficiency, the BetterRouter is therefore far superior to the BadRouter: moreover, it becomes even better for increasing order counts (provided they don't increase to the point that the O(N) cost of iterating through the order manifest and calculating distances doesn't outweigh the time saved by reducing the required iterations), while the BadRouter performance remains constant. While the BetterRouter could certainly be improved from its current, incredibly simple state, such as by teaching it to resolve ties on pickup in favor of the next stage's distance, those improvements are out-of-scope for this project.

Conclusion

It took a whole lot of simulating and optimizing to show that the BetterRouter class is better at routing than the BadRouter class.