01.118 High Performance Computing Project Report

Improvement on MRT Simulation Program via Multi-threading

1005154 Yong Zhe Rui Gabriel

1 Introduction

On my countless MRT rides across the sunny island of Singapore, I have transited at the stations Serangoon and Bishan and always wondered, why are there so many people at these stations?

This has been a question that has been on my mind since my days in secondary school and now that I am armed with the ability to simulate Singapore's MRT system with code, we can use the simulation to model this complex system and figure out how we can optimize the MRT system so that Serangoon and Bishan will have less congestion.

1.1 Objective

Thus, the goal of this project is to utilize the concept of creating a digital simulation to allow us to understand how we can optimize the MRT system and reduce congestion in the system.

Ideally, the model that is created through this project will be able to accept larger systems and easily scale to other metro systems. So simulating something like the entire Japanese train system should be able to be done by simply changing the input into the model.

1.2 Trivially Parallelizable

The almost laughable aspect of this project is that to get a good idea of the average congestion at stations, we want to run a Monte Carlo experiment with the simulation.

This means that the project is trivially parallelizable because we can just have different threads each run their own simulation run and collate the data from the various runs afterwards. This method of parallelization would result in immediate improvements.

However, in the spirit of this course, I instead look towards improving the performance of a single run and how one might utilize high-performance computing concepts to speed up a single run.

2 Model

2.1 Understanding the Agents

A metro system comprises of a few key components. They are the Stations, Trains and Commuters.

A metro system comprises Stations and Trains which pass Commuters between each other Commuters enter and exit the metro system via stations, Trains travel between Stations and Stations are places that are connected to other Stations via train tracks.

These components will each have their own class in the program to represent these objects in the metro.

2.2 Choice of Simulation Paradigm

The initial model is important as it decides the overall speed of the program. If the initial model is slow, no matter what we try to optimize at a lower level, the effects will not be significant as having a better model.

Hence, an appropriate choice of simulation paradigm would allow for a faster-performing simulation. There are the following paradigms we could choose from for this purpose.

- Discrete-Event Simulation (Process-Oriented)
- Agent-Based Simulation (Object-Oriented)

Further details about Discrete-Event Simulation and Agent-Based Simulation are detailed in Appendix 6.2

Due to the nature of a metro system, where events mainly occur when trains arrive and depart stations, the program was approached via a Discrete Event Paradigm since it allows the simulation to move between time steps quickly and hopefully progress faster.

2.3 Components of the Program

The program comprises of two main phases: initialization and simulation.

Initialization involves parsing the input data (CSVs) into structures and classes that the program will work on. Initialization only needs to be performed once at the start of the program. Subsequent runs can then make use of the initialized values without incurring the initialization cost. This means we will not necessarily see significant speed ups by improving our initialization methods. Hence, it will not be the focus of this project and can be explored as further improvement. (Brief details provided in Appendix 6.3)

Simulation is the main portion of the program that needs to be repeated and run many times. As mentioned in Section 1.2, there is a trivial way to parallelize this. However, we will examine how we could improve the performance of a single run.

2.4 Data used in the program

The data used in this simulation is gathered from the existing Singapore MRT System's travel times and the expected number of trips between two stations in the simulation is based on the data provided by LTA[1]. Further details and assumptions made of the simulation are detailed in Appendix Section 6.4

[1] Passenger Volume by Origin Destination Train Stations dynamic dataset for the month of February 2023

3 The Initial Approach

3.1 Event Graph for Discrete-Event

Figure 1 is an abbreviated version of a full event graph, where each station has a commuter spawning loop and each train has its own train movement loop.

As a quick summary, we see there are two main event loops, one of commuter spawning and another of trains arriving and leaving stations. When these events occur, we then have a sequence of other events that are triggered like boarding commuters or alighting commuters.

Based on this plan, we see that there are 4 main events that the program is built around, spawning and terminating commuters as well as trains arriving and leaving a station.

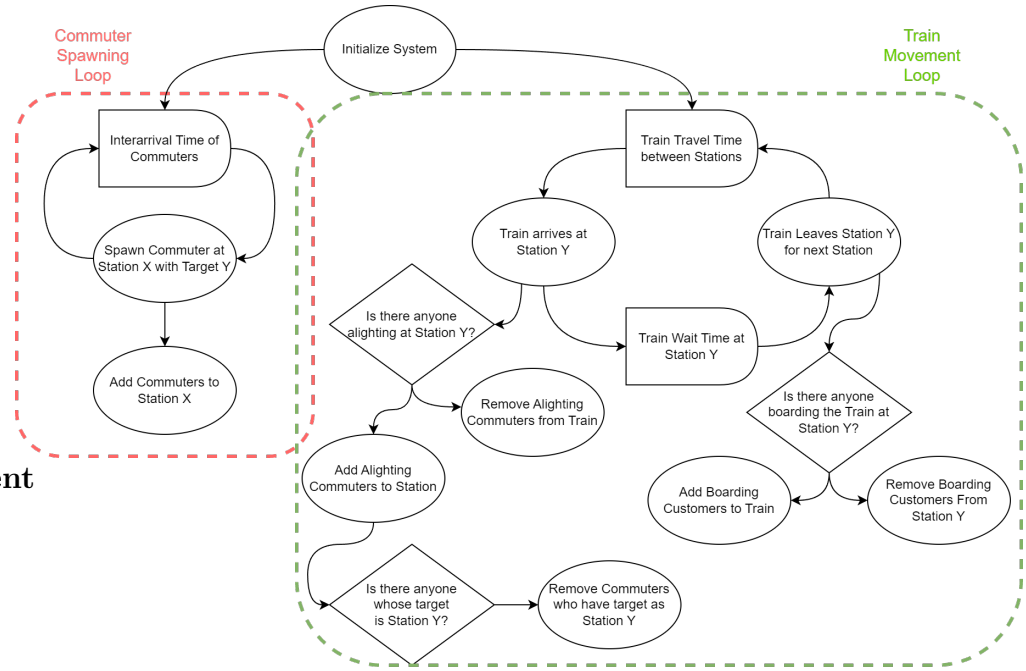

Figure 1: Event Graph for Initial Model

3.2 Binary Min-Heap Event Queue

In a Discrete-Event Simulation, the event queue is what maintains the execution order of events in the simulation. The event queue thus needs to have `insertion`, `extract_min` and `peek`(see the time of the earliest event) operations be fast. This is because these are the only operations that are done on the event queue.

We see that we want to use a min-priority queue abstract data structure for the event queue. One logical choice is thus a binary min-heap. It performs `insertion` and `extract_min` in $O(\log(n))$ time and `peek` in $O(1)$. Hence, we initially implement the event queue as such.

3.3 Code

The code for the single-threaded version can be found on GitHub at this link

3.4 Results

Table 1 contains the data collected from running the various single-threaded versions

As we can see from the trial times taken from running the simulation with a binary heap implementation of the event queue, the program is presently taking about 75s per run.

From examining the flame graph (Appendix 6.1 Figure 5), maintaining the binary heap seems to be taking up quite a bit of execution time ($\sim$ 13.25% of execution time)

	Timings (s)			
Type	**Trial 1**	**Trial 2**	**Trial 3**	**Avg**
Binary Heap	73.967	78.832	73.174	75.324
PQ from Datastructures.jl	51.85	51.18	51.331	51.454

Table 1: Results of Discrete-Event Simulation Version

3.5 Priority Queue from DataStructures.jl

There is a package `DataStructures.jl` that already implements the priority queue abstract data type. Thus, in an attempt to improve this initial iteration of the program, the binary min-heap was replaced with the priority queue from `DataStructures.jl` and performance was measured again. The results are shown in Table 1

We see that there was a significant improvement from 75s to 51s by simply using a pre-existing implementation of the priority queue.[2]

[2]If there is a library or package that we can call upon to do the job, we should consider it first before attempting to create our own programs to improve performance.

3.6 An Extension: Fibonacci Heap?

One possible extension to explore would be to utilize a Fibonacci Heap[3] for the event queue. The Fibonacci Heap is said to have an *amortized cost* of $\theta(1)$ for `insert` and `peek` as well as $O(\log(n))$ for `extract_min`. This data structure, while having in theory, good time complexity is said to only be beneficial at a large scale. It would be interesting to see if that holds true (and the Fibonacci Heap is not efficient) in this use case.

4 Moving to Multi-Threading

After executing the initial approach, it is noted that Discrete Event Simulation is very serial in nature due to the singleton event queue that the system abides to. This makes the initial Discrete Event approach effectively almost impossible to parallelize simply since it is unclear which events in the queue can be processed in parallel.

To parallelize the model means performing a redesign.

4.1 The Redesign

Using a mix of concepts from Agent-Based Simulation and Discrete-Event Simulation, we can come up with a revised model to program.

Figure 2 illustrates the steps each station will take in a single time step[4].

Noting that each station is effectively independent of each other, we are able to then consider having threads run station operations synchronously.

However, when trains arrive/leave stations, the stations need to communicate with each other, which means the stations need to run in tandem and move forward in time as a system.

This means that after running station operations for a certain period, the stations need to sync up to know which trains are arriving and leaving the stations (Phase 2). This concept is effectively Agent-Based Simulation design, except that stations are unconventionally thought of as agents sending trains (events) between each other.

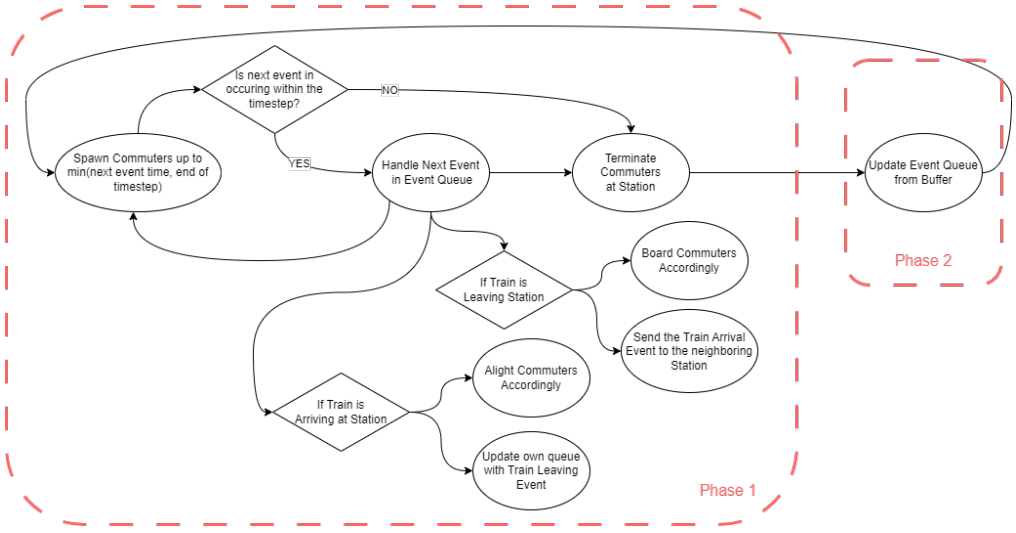

Figure 2: Station Simulation Step

Focusing on each station's operations, we still treat them as a Discrete-Event Simulation and have each station have its own event queue of train arrivals and departures. Through this method, we do not need to update the location of trains since we know where a train is through the event queue of stations.

The process of Commuter spawning and termination is treated more as part of the Agent-Based Simulation Paradigm where for each timestep, we will simply spawn a certain number of commuters based on a random number drawn from a Poisson distribution, instead of having events in the event queue to determine spawning of commuters. As for termination, we will terminate commuters at the end of each timestep.

This is to allow for a smaller event queue size which would allow for faster operations because we noticed that maintaining an event queue is a costly operation. It is more effective to simply use a loop instead of having these events be part of the event queue.

4.2 Managing the Event Queue

Since a single station can have multiple neighbours, at any timestep, it could be possible that multiple neighbours want to send trains over into the station's event queue. This might lead to race conditions occurring and trains not being sent over to the station's event queue properly due to overwriting.

Thus, an event buffer is created for each station for the sole purpose of neighbours writing to the buffer. Each neighbour is assigned a slot for them to write to, thus they will not overwrite any other neighbours' events.

Each station updates its queue from the buffer at the end of each timestep to receive the new events (in Phase 2). Now, with the buffer, race conditions will be avoided. This concept is illustrated in Figure 3

[3]Interesting Video on Fibonacci Heaps at this link

[4]The code does not implement Figure 2's sequence exactly. It did not have the cycle between spawning and event handling due to an oversight.

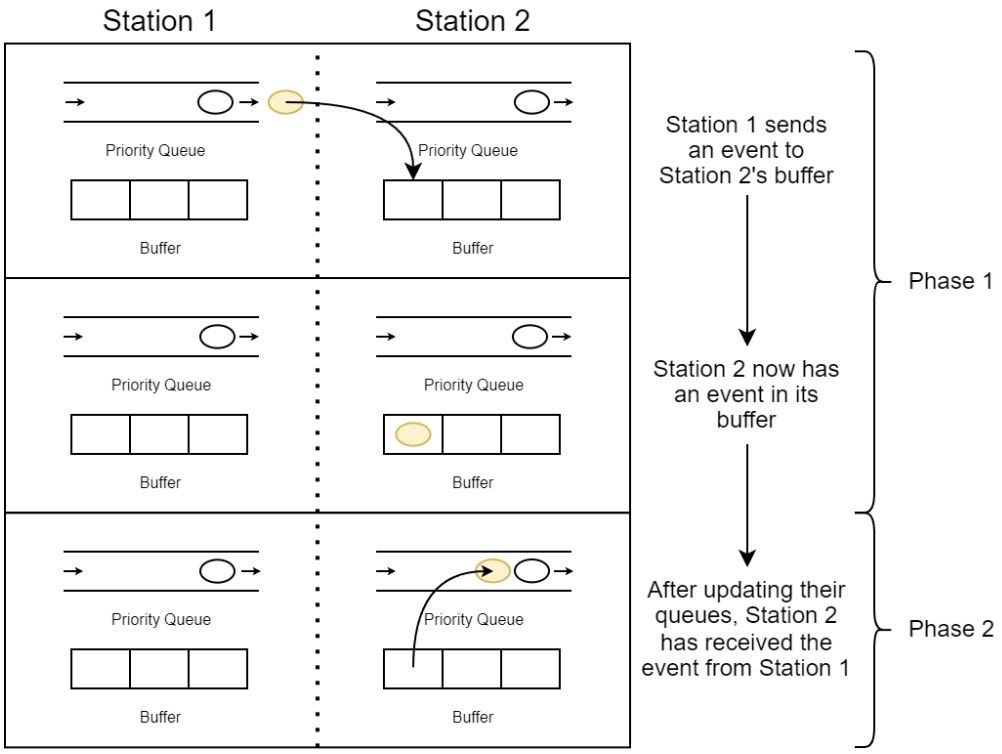

Figure 3: Stations Communicating to each other

4.3 Code

The code for the multi-threaded version can be found on GitHub at this link

4.4 Results

Table 2 contains the data collected from running the redesigned program. The following trials utilized 4 threads. The timestep is the number of minutes in simulation time before stations update their event queue.

We see that by transitioning to a threadable version of the simulation, we see a gain in performance to 36s when our timestep is 0.5. This is a 29% decrease in time taken from the best single-thread version we had.

This program does not benefit from simply being able to be threaded, we can also extend the duration of each timestep to gain performance (at the cost of some precision). Given a large enough duration, the simulation could progress faster by having to move through lesser timesteps. We see this improvement when we increase our timesteps to 1 and 2 as seen in Table 2.

	Timings (s)			
Timestep	**Trial 1**	**Trial 2**	**Trial 3**	**Avg**
0.5	36.781	35.241	37.168	36.397
1	24.385	31.191	24.073	26.550
2	20.089	19.235	19.663	19.662
3	Runs into event overwriting error			

Table 2: Results of Multi-Threaded Version with different timesteps

However, it is noted that the duration cannot be too long as stations need to sync up with each other after a certain duration. (e.g. Timestep 3 in Table 2)

4.5 Improving Commuter Spawning

By examining the flame graph (Appendix 6.1 Figure 6) we see that `event_spawn_commuters` is a process that takes up a relatively significant portion of time ($\sim$ 27.68% of execution time). Hence, we work on improving that function first.

Looking at Figure 6, we see that The operation `ijl_apply_generic` takes up $\sim$ 48% of the spawn commuters function and it is related to the loop that adds the commuters to the station. Using a simple for loop for such a task seems to be inefficient (See Appendix 6.7 Figure 10 for code).

	Timings (s)			
Method	**Trial 1**	**Trial 2**	**Trial 3**	**Avg**
For loop	36.781	35.241	37.168	36.397
`fill!`	30.086	32.635	31.247	31.323

Table 3: Results of Different ways to add spawned commuters

After running a brief experiment (Details in Appendix 6.8) to consider how the new commuter should be spawned, it seems that the `fill!` function works the fastest and thus is the chosen function to speed up `event_spawn_commuters`

Using 4 threads and having the timestep as 0.5, we attain the result in Table 3

We see that there appears to be some improvement from the original 36s to about 31s.

4.6 Data Storage

Another point noted was that accessing the dictionaries that were used to store input parameters like spawn rate and interchange paths seem to take up significant time in the flame graph.

Thus, an idea was to change these dictionaries to matrices or vectors of vectors to provide direct access to the data without having to perform hashing. The results are shown in Table 4 and the trials done with 4 threads and 0.5 timestep.

However, after attempting to change it for the spawning parameters, the results seem to not make much difference to the timing. This is possibly due to matrices and vectors of vectors requiring contiguous spaces in memory and perhaps because of that the time spent allocating memory they did not result in any performance improvements in contrast to using a dictionary.

	Timings (s)			
Method	**Trial 1**	**Trial 2**	**Trial 3**	**Avg**
Dictionary	32.428	35.252	20.858	29.513
Matrices	28.223	30.239	30.018	29.493
Nested Vectors	33.71	35.655	35.253	34.873

Table 4: Results of Multi-Threaded Version with different methods of storage

4.7 Different Number of Threads

A very natural step would then be to vary the number of threads used. The results are shown in Table 5 and the timestep for each trial was set as 0.5.

While there is clearly an improvement from having 4 threads instead of 1, beyond that we do not seem have any significant gain in performance. This seems un-intuitive at first but it could be due to an issue of different workloads between threads. Since each station has to handle a varying number of commuters, inefficient job scheduling for the various threads will not allow for adding more threads to have significant improvements.

	Timings (s)			
Threads	**Trial 1**	**Trial 2**	**Trial 3**	**Avg**
1	42.463	44.270	44.659	43.797
4	20.931	21.880	21.401	21.404
8	20.238	20.885	21.609	20.911
12	19.207	19.495	19.352	19.351

Table 5: Results of Multi-Threaded Version with different thread count[6]

5 Conclusion and Possible Improvements

5.1 Job Scheduling

One issue that can be improved is that given each station has different workloads with respect to handling commuters (some stations are busier than others). By using `@threads`, there is no specified workload assignment to each thread. The thread scheduler is not aware of the workload that each station might have.

Hence, a possible improvement would be to pre-assign stations to each thread, this way, each thread is assigned a similar amount of work and no thread is left idle without any work.

Some things to note about this improvement would be that over a single day, station's workload will vary and the scheduling of jobs will need to change depending on the time of the day the program is simulating.

If the job scheduling takes too long, this might become inefficient. However, with the use of greedy algorithms like Longest-Processing-Time first scheduling that runs in $O(n)$, this might not be too impossible.

5.2 Parallelizing Station Operations

Another improving the performance would be to parallelize the operations each station needs to take. This way, the parallelization is station-agnostic and we do not need to worry about job scheduling

5.3 GPU Usage

We should not eliminate the possibility of using the GPU to accelerate the program. Given that there is the existing FLAME GPU framework that runs on CUDA GPUs, it seems definitely possible that there exists some way to reimagine the model for usage on the GPU which could potentially accelerate the program significantly by making use of the many ALUs that a GPU has.

5.4 More Rigorous Testing

Something else to consider would be to test the programs more rigorously. Given the time, different scales of metro systems should be experimented on, ranging from 10s of stations to millions of stations to truly understand the performance benefits of the design of the program.

[6]These trials were run on a separate day as the other trials and seem to have a 33% improvement for differences in computer conditions

6 Appendix

6.1 Flame Graphs

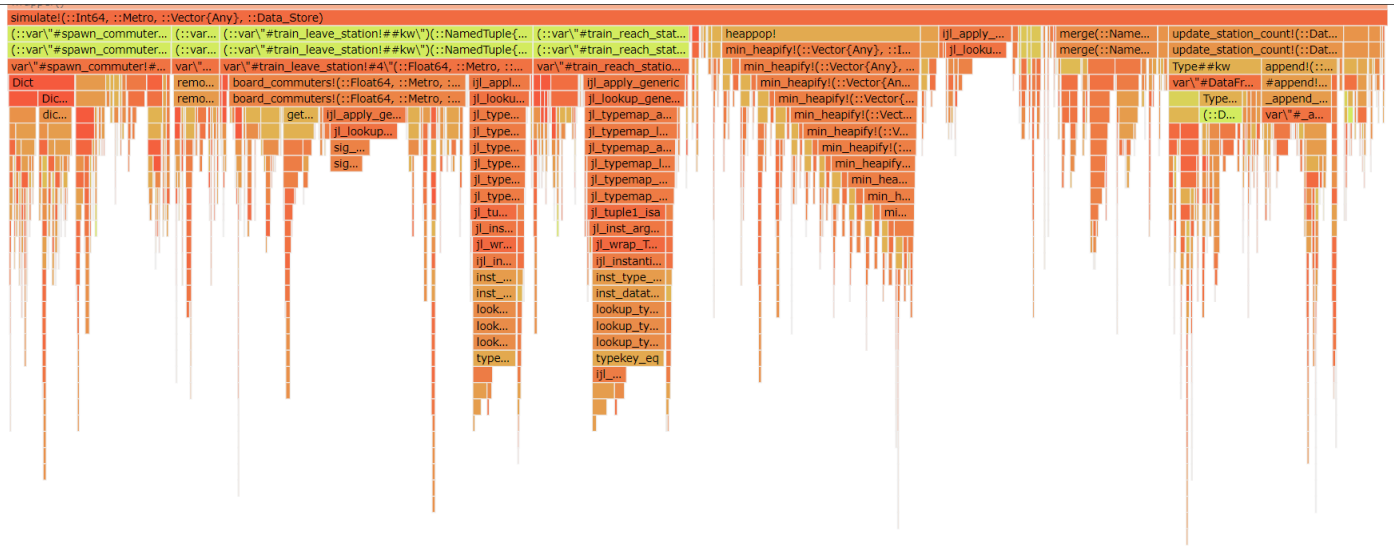

Figure 4: Flame Graph with Binary Heap Event Queue

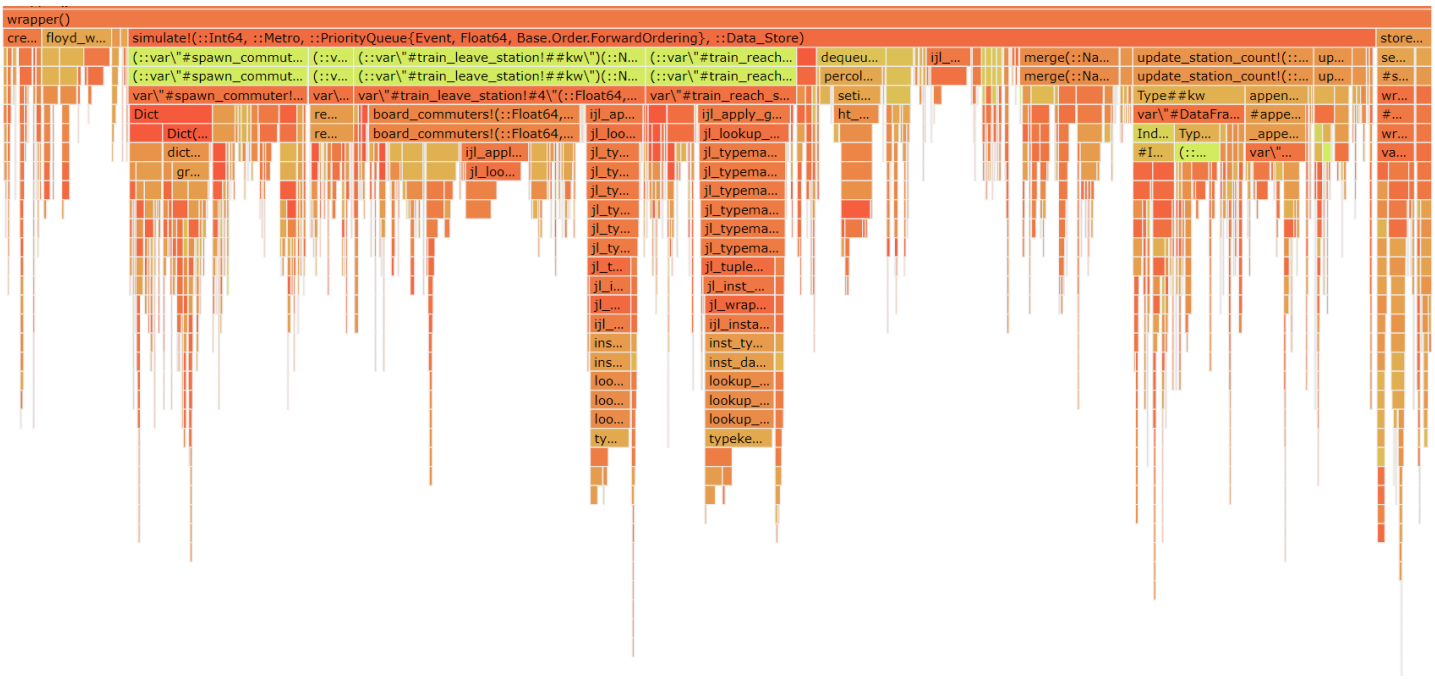

Figure 5: Flame Graph with Event Queue using DataStuctures.jl Priority Queue (right)

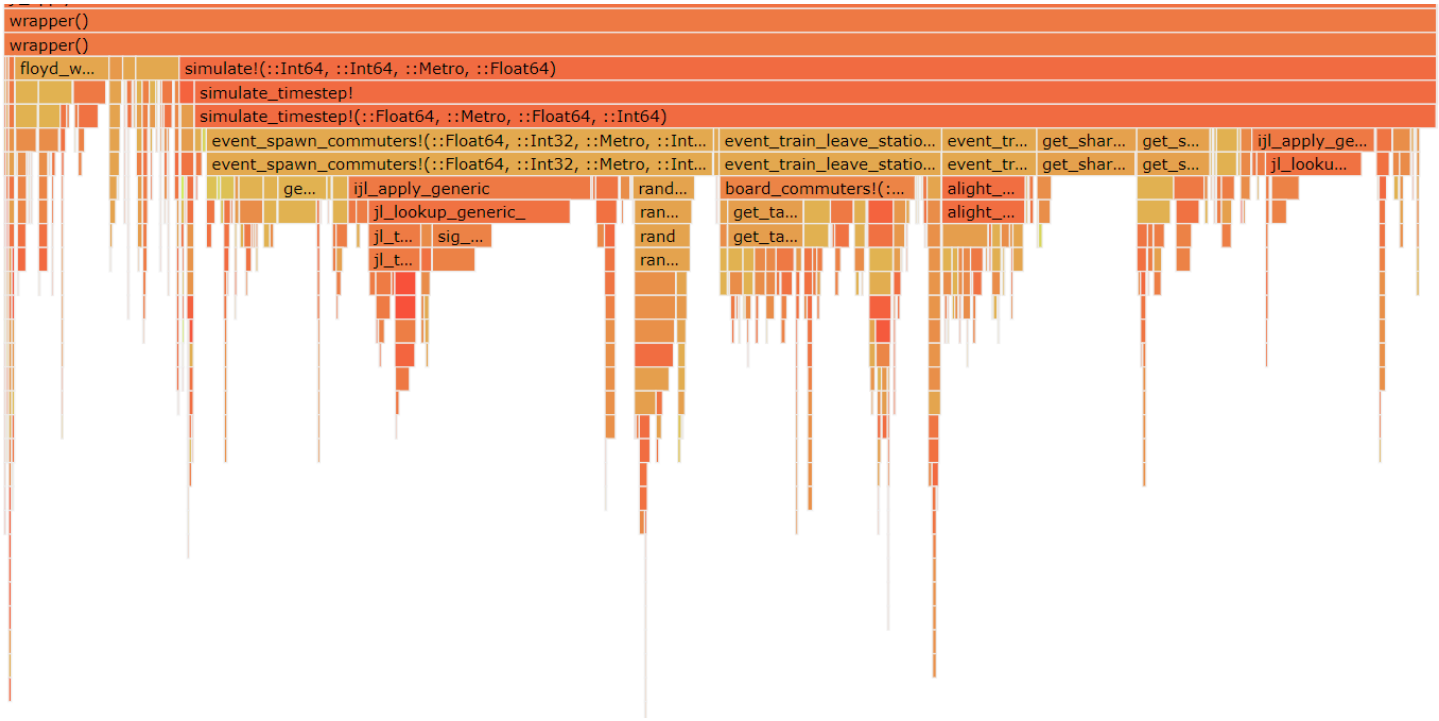

Figure 6: Flame Graph of Multi-Thread version

6.2 Explanation of Simulation Paradigm

The following section briefly describes what Discrete Event Simulation and Agent Based Simulation is.

6.2.1 Discrete Event Simulation

The key features of Discrete Event Simulation are as follows

- Overall system is described by a state
- Events allow for the systems state to change
- There are intervals between events called delays
- Events trigger other events depending on conditions on the state

A key point of Discrete Event Simulation is that it is able to jump in time from one event to another. If nothing is going on in a particular timestep, we can simply move to the next timestep.

This way, if events happen relatively infrequently, we are able to progress through the simulation quickly.

6.2.2 Agent Based Simulation

The key features of Agent-Based Simulation are as follows

- Captures interactions and progress by tracking where every agent is at any point in time
- Can capture more complex structures and dynamics
- Almost anything can be modelled with Agent-Based
- Computationally intensive since it updates at every single time step which could mean a lot of updates

This paradigm of simulation can be a lot more detailed and but for each timestep, it could be wasting a lot of time updating states of agents that are not doing much in the simulation.

6.3 Initialization Details

The initialization involves various steps such as constructing the metro struct as well as creating the paths for the Commuters to follow.

6.3.1 Construction

Construction steps are parallelizable since each station or train are created with a unique ID that then can be stored in their own slot. This is a portion of initialization that can be parallelized.

6.3.2 Path Finding

Using an efficient all-pairs shortest path algorithm, Floyd-Warshall Algorithm, that runs in $O(n^3)$ where n is the number of stations. The place that this part of initialization can improve on is the way path reconstruction is done for the commuters. Currently, the implementation of path reconstruction is done naively and can be part of future improvements to the program.

6.4 Simulation Assumptions and Rules

The simulation has the following asssumptions and rules

- Commuters only arrive at stations between 6am and 12am
- Commuters do not shy away from interchanging, in fact they think that interchanging only takes 6 seconds
- Commuters only take the shortest path between their location and their destination
- Trains are still running until 1am and do not move away to any depot, they are left on the metro system at the end of the day
- The interarrival times of commuters are exponentially distributed
- The expected number of commuters going from one location to another is determined based on the sample data provided by LTA for the month of February 2023
- Trains have an interarrival time of 4 minutes
- Trains also have a capacity of 900 commuters
- Trains do not break down or have delays during the day
- Commuters enter the system through stations
- Commuters only exit the system through stations
- Commuters will board the first train they can board (be it based on destination/capacity)
- Commuters will wait in line orderly until it is their boarding turn (no cutting queue)
- Commuters do not miss their target stop

6.4.1 MRT map

The metro system that the simulation is based upon is follows the map below

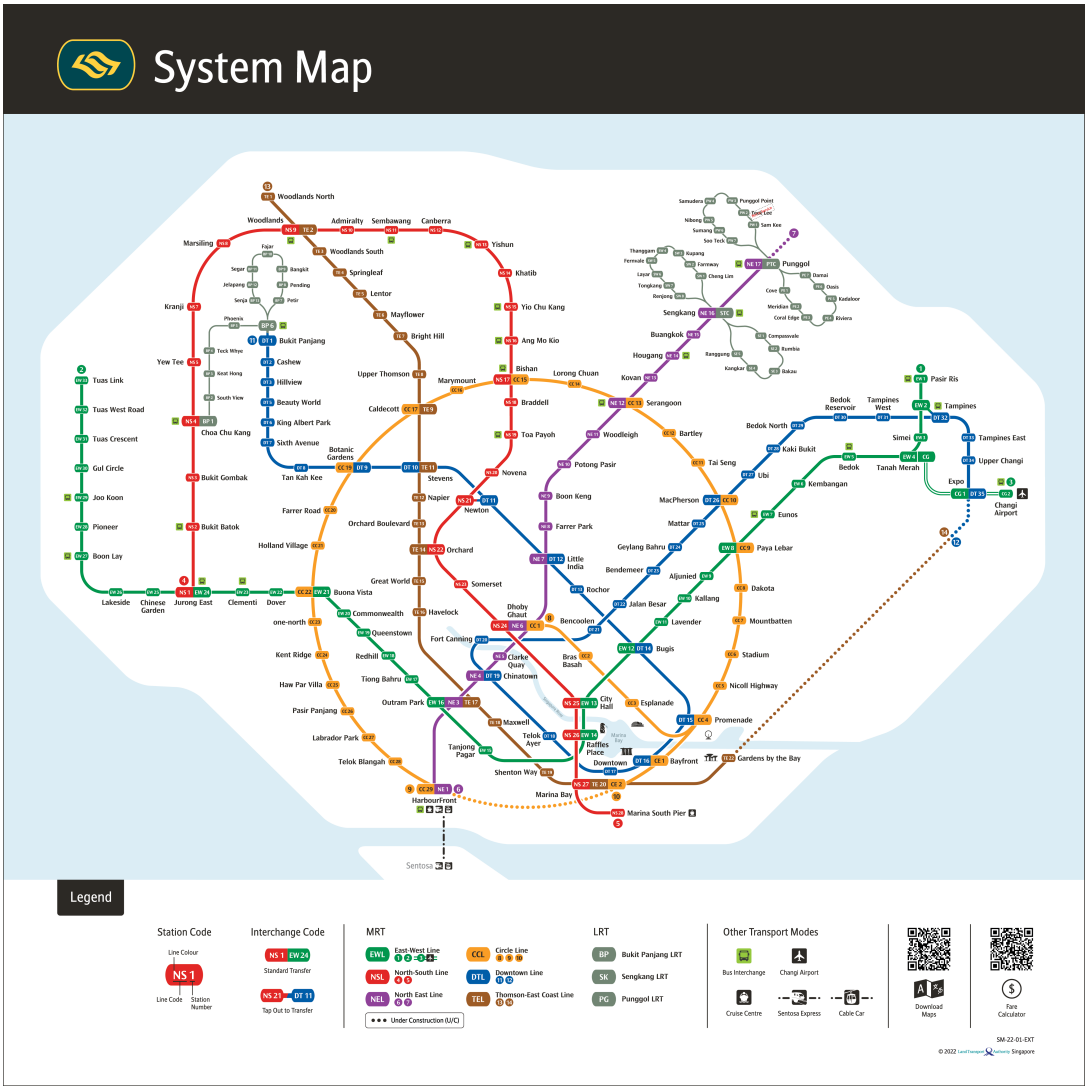

Figure 7: MRT Map of Singapore

6.5 Correctness of programs

The main check when scaling the program up was with the sanity check of whether the expected number of people spawned and terminated tallied with the data that was referenced.

6.5.1 Single Thread

For the single thread program, the program was first developed on a smaller scale with only 3 stations, a, b and c, on a single line. The execution sequence was then printed and tallied to check for correctness.

One such example is in Figure 8

```
C:\Users\gabes\Documents\HighPerformanceComputing\Project\HPC_Project>julia test_simul.jl
time 0: Train 1 reaching Station a
time 0: 0 Commuters alighting Train 1 at Station a
time 0: 0 Commuters boardng Train 1 at Station a
time 0: spawning commuter at Station a
time 1: spawning commuter at Station b
time 2: spawning commuter at Station c
time 2: Train 1 leaving  Station a
time 3: Train 1 reaching Station b
time 3: 0 Commuters alighting Train 1 at Station b
time 3: 0 Commuters boardng Train 1 at Station b
time 5: Train 1 leaving  Station b
time 6: spawning commuter at Station a
time 7: Train 1 reaching Station c
time 7: 0 Commuters alighting Train 1 at Station c
time 7: 1 Commuters boardng Train 1 at Station c
time 7: spawning commuter at Station b
time 8: spawning commuter at Station c
time 9: Train 1 leaving  Station c
time 11: Train 1 reaching Station b
time 11: 0 Commuters alighting Train 1 at Station b
time 11: 1 Commuters boardng Train 1 at Station b
time 12: spawning commuter at Station a
time 13: Train 1 leaving  Station b
time 13: spawning commuter at Station b
time 14: Train 1 reaching Station a
time 14: 2 Commuters alighting Train 1 at Station a
time 14: 2 Commuters boardng Train 1 at Station a
time 14: spawning commuter at Station c
time 16: removing 2 commuters from Station a
time 16: Train 1 leaving  Station a
time 17: Train 1 reaching Station b
time 17: 0 Commuters alighting Train 1 at Station b
```

Figure 8: Execution sequence on simple metro system

6.5.2 Multi Thread

Similarly, logical checks were also done with the execution sequence.

However, at this point in time, I was recommended to try and unit test my functions for correctness. he details of the unit tests can be found at the link here on Github

```
julia> include("tests/tests.jl")
Test Summary:     | Pass  Total  Time
Create Class Test |   47     47  5.4s
Test Summary:           | Pass  Total  Time
Get Neighbour Functions |    4      4  0.0s
Test Summary: | Pass  Total  Time
Event Passing |   76     76  2.5s
Test Summary: | Pass  Total  Time
Add Commuter  |   19     19  0.3s
Test Summary:   | Pass  Total  Time
Remove Commuter |   12     12  0.1s
Test Summary:              | Pass  Total  Time
Utility Get Board targets |    4      4  0.2s
Test Summary: | Pass  Total  Time
Board Train   |   10     10  1.3s
Test Summary:    | Pass  Total  Time
Alight Commuters |   10     10  0.1s
Tests Done
```

Figure 9: Sample unit test output

6.6 Cross check with ground truth

Ideally, given more time, the programs data output should be cross-checked for each station with the ground truth data to check for the correctness of the model.

6.7 Initial way to Spawn Commuters

```
for i in 1:number_spawn
        new_commuter = Commuter(
                station,
                target,
                time,
                time,
                0
        )
        s.commuters["waiting"] = add_commuter_to_station(s.commuters, "waiting", new_commuter)
end
```

Figure 10: Initial for loop to add commuters

6.8 Experiment to Fill Commuters

For Commuter spawning, we experiment with three methods

- using `fill!`
- using for loop and `@threads`
- using just a for loop

The results using `@btime` are as shown below in the image

```
In [30]: x = Vector{Commuter}(undef, 100);

In [31]: @btime fill!(x, new_commuter);
  150.365 ns (0 allocations: 0 bytes)

In [32]: @btime @threads for i in 1:10
             new_commuter = Commuter(
                 i,
                 2,
                 3,
                 3,
                 0
             )
             x[i] = new_commuter
         end;
  5.120 µs (83 allocations: 8.27 KiB)

In [33]: @btime for i in 1:10
             new_commuter = Commuter(
                 i,
                 2,
                 3,
                 3,
                 0
             )
             x[i] = new_commuter
         end;
  231.423 ns (10 allocations: 640 bytes)
```

Figure 11: Code depicting the 3 different ways

We see that using `fill!` is clearly the fastest method