Assignment 1 Report

Nguyen Son Linh (A0200705X)
Njal Telstoe (A0260770H)

Program Description

Parallelization strategy

To parallelize the program, we create one thread per link in one direction. Each tick, we iterate over all the links and update their state.

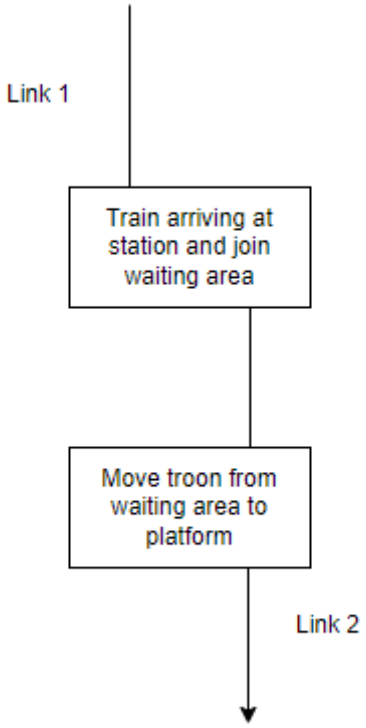
Each link keeps track of all the troons which are on the platform and separates them by which state they are in. The link is responsible for updating its state by updating each troon and moving them to the next state.

Each troon can be in 1 of 3 states.

1. Waiting in holding area
2. On platform (Open door, let passenger on, close doors, wait for link)
3. Transiting on link

The link will begin by updating the troon transiting on the link. It will move the troon to the waiting platform of the outgoing link at the destination station if it is finished transiting. Secondly it moves the troon that's finished on the platform to the transit area if it's available. Lastly it will move the first troon in the holding area to the platform. Notice that the states are updated in reverse order, this is because each state is reliant on the next state being finished, the platform must be available for the waiting area troon to move to the next state for example.

This is parallized by simple using the `#pragma omp parallel for` directive when iterating over all the links. Some synchronization is needed to resolve link contention and guarantee program correctness. We iterate over all the links twice, to resolve some synchronization issues. The first for loop will update the links where there is a transiting troon, whilst the second for loop will update links with troons on the platform and process the waiting area . We had to do this because when a troon arrives at a station, it can immediately go to the platform of the next link to pick up passengers or join the holding area. The following diagram illustrate this situation:



Basically, Link 2 may skip the task of moving a newly arrived train from Link 1 to platform.

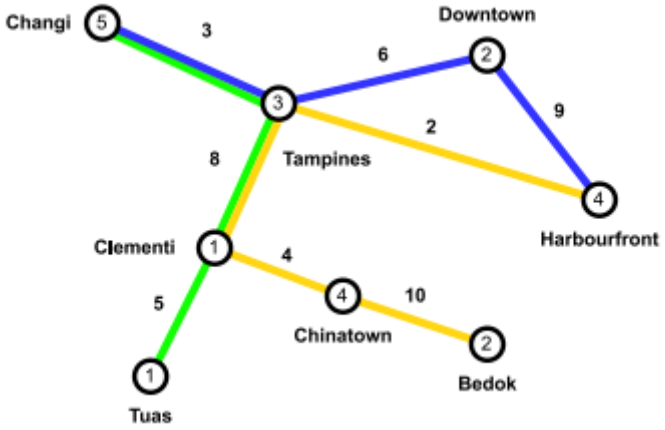
How link contention was resolved

To account for link contention, each link keeps track of the troon that is on the platform and the troon that is transiting on the link. One key observation is that contention only happens when troons arrive at a station, those that have waited longer get the priority, and id is used as a tiebreaker. A priority queue is perfect for this task and was employed. After deciding which troon goes to the platform, this troon is automatically the next one to transit. When to begin transit can be calculated easily by checking whether the link is free.

Key data structures and synchronization constructs

Vectors are used to create lists of all the basic object types (Links, Troons, Stations). We use a two dimensional matrix to represent the troon network. The matrix contains pointers to the links, and is positioned such that entry [i][j] will contain a pointer to the link between station i and j, or nullpointer if there is no link between them. This makes it efficient to find link information if given a start and end station.

As each link has a priority queue(PQ) to maintain the order that troons go onto the platform, a lock is required when pushing into this PQ. We used omp_lock for this. Using the example train network, which is reproduced below, Tampines->Clementi link can have incoming troons from Harbourfront->Tampines and Changi->Tampines. Using a lock guarantees there is only one push to the priority queue at any single moment during program execution.



There is also a need for a barrier to solve the issue of moving an arriving troon to the waiting area. This is done implicitly by `#pragma omp parallel for` directive.

What aspect of the input size affects speedup most?

We observe that the speedup is mainly affected by the number of stations and number of ticks. This is in line with our implementation. We iterate over all the links in the network in parallel. The number of links are dependent on the number of stations, such that if we increase the number of stations the parallel fraction of the program will increase, thus increasing the speedup by executing a bigger chunk of the program in parallel. We also observe that for a low amount of ticks, the sequential solution executes faster than the parallel solution. This is reasonable because for a low amount of ticks, the sequential part of the program which includes initializing the network and printing the troon states will make up the majority of the program. The parallel part will not be big enough to make up for the overhead needed for creating threads, context switching and thread synchronization. Thus making the parallel solution less efficient. The speedup seems to be quite independent from the number of troons, we observe that the execution time is almost constant when increasing the troon count from 30 000 to 300 000. This is also in line with our solution, we only iterate over the number of links to update the network state. There will be some extra overhead for printing the extra troons, spawning more troons, and some increased time for adding and removing troons from the waiting platform because of more contention. But in comparison to the total execution time, this overhead becomes insignificant.

Performance Optimizations that were made

At first, we decided to use 3 different `#pragma omp parallel for` in order to deal with links that have transiting troon, links that have an on-platform troon, and move troons from waiting area to platform. Afterwards, we realized the second and third loop can be combined without breaking the program and proceeded to do so. This is a small optimization aiming to reduce overhead of thread creation. Performance difference is summarised in the following table

using an input of 17000 stations, 200000 troons per line running for 100000 ticks and requiring output of the last 5 ticks. Architecture: xs-4114:

Number of `#pragma omp parallel for`	Time (seconds)
2	25.0970
3	26.2752

By doing this, we achieved a **4.4% increase** in performance which is minor in the context of this assignment but may prove to be useful in other circumstances. It also shows that the overhead of thread creation and joining in openmp is considerable.

Appendix

How to reproduce results

In our repository, there are two scripts run_troons.sh and troons.sh, which we used to measure performance. In order for this to work, these two scripts must be copied to the user home directory. Also, please clone our repo to the home directory. The following commands are sufficient:

```
cd ~
git clone
https://github.com/nus-cs3210/cs3210-a1-a1-a0200705x_a0260770h.git
cd cs3210-a1-a1-a0200705x_a0260770h/
cp run_troons.sh troons.sh ~/
```

To measure performance, please run:
`./run_troons.sh <exec> <input> <partition>`
where `exec` is the executable, `input` is the input file in the `testcases` directory of the repo and `partition` is the partition that is used to measure performance. This will create a symlink `<exec>-<partition>-latest.log` in the home directory that links to the stored log file in nfs For example,
`./run_troons.sh troons example.in xs-4114` will run the `troons` executable using `example.in` on partition `xs-4114` and create `troons-xs-4114-latest.log` in the home directory.

All of our execution time measurements were done using `perf`, with the option `-r 5` used to reduce uncertainty.

“create_input.py” is a simple script that was used to create input files for the troons program. It takes 7 parameters:
NUM_STATIONS
NUM_TICKS

NUM_GREEN_TROONS
NUM_YELLOW_TROONS
NUM_BLUE_TROONS
NUM_LINE_OUTPUT
SHARE_PROBABILITY (A value from 0 to 1 which will increase the chance of multiple stations sharing the same links)

Lab machine nodes used for testing and performance measurement

soctf-pdc-012 was used as the main development machine and to test for correctness. Slurm was used to correctly measure performance on both i7-7700 and xs-4114 architectures. Since slurm may allocate any of the machines matching the architecture, there is no point in listing all of them.

Relevant performance measurements:

Type	Architecture	Stations	Troons	Ticks	Number of lines	Share probability	Node	Execution time
Sequential	xs-4114	15K	300	100K	10	1.0	6	34.083s
Parallel	xs-4114	15K	300	100K	10	1.0	6	17.621s
Sequential	xs-4114	15K	30K	100K	10	1.0	6	58.727s
Parallel	xs-4114	15K	30K	100K	10	1.0	5	19.7507s
Sequential	xs-4114	15K	300K	100K	10	1.0	5	60.671s
Parallel	xs-4114	15K	300K	100K	10	1.0	5	20.3618s
Sequential	xs-4114	1K	300K	100K	10	1.0	5	5.7936s
Parallel	xs-4114	1K	300K	100K	10	1.0	6	1.8765s
Sequential	xs-4114	5K	300K	1K	10	1.0	6	13.4477s
Parallel	xs-4114	5K	300K	1K	10	1.0	5	16.66s
Sequential	xs-4114	15K	300K	10K	10	1.0	5	16.0297s
Parallel	xs-4114	15K	300K	10K	10	1.0	6	15.7851s