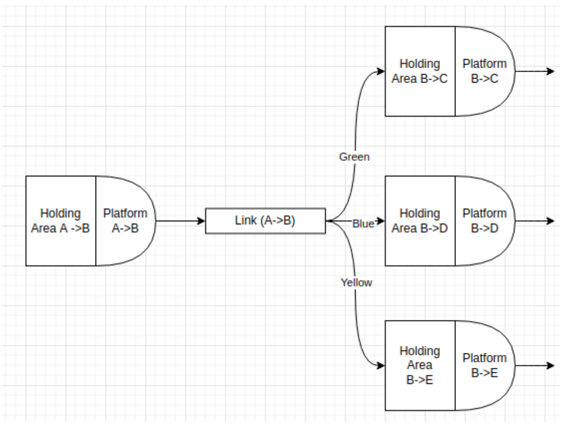
CS3210 Assignment 1 Report

Problem Modelling

In this troon simulation problem, it is clear that it is a graph simulation problem. To model the graph, we decided to have:

1. 2 type of nodes.
    a. The first node is the holding + platform where each node is associated with a source and destination. The source and destination are not assosictive. Hence node from A->B is distinct from node from B->A. Implementation wise, they could be split into two nodes with 1 edge.
    b. The second node is the link. Therefore the link connects one platform to the next waiting areas depending on the line of the troon.
2. An edge just represents the possible movement of the troon from one place to another.

Pictorially, the graph looks like the following picture:



Therefore, in this modelling, we don't have the notion of station. As a high-level implementation overview, we decided to follow this plan:

1. "Troon" is just a data class that contains states such as the link, source and destination.
2. There are 3 classes of HoldingArea (WaitingArea in our code), Platform, and Link. These 3 classes would form the graph as shown above and each of these classes is responsible to move the troon from the source to destination. Hence, it is a **push-based design**.
3. In each tick, we should process all the platforms, waiting area and then the link following this pseudocode

```
  for (size_t tick = 0; tick < ticks; tick++) {
      UpdateAllLinks();
      // in the worst case, the platforms need to do 2 jobs, to push
the current troon to link and take another incoming troon
      PushAllPlatform();
```

```
        // Manage Waiting Area (Holding Area)
        SpawnTroons();
        UpdateAllWA();

        UpdateWaitingPlatform();
        PrintTroonDescriptions();
    }

    CleanAllResources();
}
```

Parallelization Strategy

Upon understanding the pseudocode, then our parallelization strategy is to:
In each step of the processing, the sequential pseudocode would look like

```
for (size_t i = 0; i < size; i++) {
   node[i]->process();
}
```

Therefore, we can parallelize each of these processes to the following pseudocode

```
#pragma omp parallel for
for (size_t i = 0; i < size; i++) {
   node[i]->process();
}
```

There are 4 processes that can be parallelized:

1. Update all the links and push to Holding Area in parallel.
2. Push all the troons from the platform in parallel.
3. Update all Holding Area (i.e. Waiting Area) in parallel such as pushing the troon from the waiting area to the platform, etc.
4. Update all platforms in parallel such as incrementing the waiting time, etc.

Link Contention Resolution

Following the graph model in our solution, there is no link contention because each link only receives from 1 platform at each tick.

Key Data Structures and Synchronization Constructs

There are a few data structures:

1. Firstly, we initialized all the required HoldingArea, Platform and Links and store them into **2D arrays**. For example, HoldingAreaData[i][j] indicates the HoldingArea from i to j where i and j are the indexed station names.
2. After which, we assemble the graph (the HoldingArea, Platform, and Link). We also store each of them into **vectors** of HoldingArea, Platform, and Link so that they can be processed easily based on the pseudocode above.

3. For the holding area, we have a **queue and a priority queue (PQ)** based on troon's id comparison to break the tie when two troons come at the same time.

The only possible race-condition comes from the contention of multiple links pushing the troons to the same pq (to be sorted by id before being pushed to queue), we use a thread-safe priority queue using `pragma omp critical` construct given by the OMP library. We cannot directly used the mutex given by c++11 library because c++11 constructs doesn't work well with OMP and often gives undefined behaviour (source: https://stackoverflow.com/questions/53508921/is-it-safe-to-use-a-c-mutex-in-openmp-code). However another downside of `pragma omp critical` is that that part of the code becomes sequential because if the name is the same for all object instance, then only 1 thread can access the critical region as discussed here: https://stackoverflow.com/questions/28716014/c-pragma-omp-criticalname. One possible optimization is to use `omp lock` construct instead (but interestingly, it makes our program slower slightly for 5000 stations input so we decided to drop it).

Input Factors and Speedup

1. The **number of stations and links** affect speedup the most for our implementation.
2. As mentioned in our parallelization strategy, there are 4 processes that can be parallelized. All of which are related to operations done in links and stations (holding area and platform). In our implementation, we parallelized all these aspects and as the result, as the number of links and stations increase, **our speedup compared to our sequential implementation** also increased.

In all of the experiments, we are using dxs-4114 machine

When we fix the number of stations as shown in Appendix table 1, links, and ticks to 5000, 7500, and 5000 respectively, we can see that the speedup remains almost constant for all the case.

As we increase the tick as shown in Appendix table 2, we cannot observe any increase on speedup. Interestingly, the speed of our parallel implementation if even slower in dxs-4114 machine. We also tried in i7-9700 machines for some of the cases with 8 threads and the parallel give 2x speedup. A possible reason is that because in per tick, the number of tasks that can be parallelized remains the same as we fix the number of traTcains, stations, and links to be 5000, 5000, and 7500 respectively. The increase in time required for the parallel program to run might be because of the overhead imposed when creating parallelization is too big during each for loop which causes the overall code to slow down.

As the stations and links number increased as shown in Appendix table 3, our parallel program also manages to achieve better speedup compared to the sequential version of our program. This is due to in the parallel version, we can process several of the stations and links on parallel. The number of trains and ticks is fixed to be 5000 and 5000.

Performance Optimization Attempt

Actually, We were reading about mastering OpenMP performance (source: https://www.openmp.org/wp-content/uploads/openmp-webinar-vanderPas-20210318.pdf) and encountered that the correct way to use `parallel` construct is like the following:

```
#pragma omp parallel {
    #pragma omp for { <code block 1> }
    #pragma omp for { <code block 2> }
}
// End of parallel region
```

Instead of:

```
#pragma omp parallel for { <code block 1> }
#pragma omp parallel for { <code block 2> }
```

The reason is that in the first block of code, the parallel region cost is only incurred once and in the second block, the parallel region cost is repeatedly incurred. Therefore, applying this optimization to the code similar to pseudocode described in parallelization strategy, I obtained a slight speedup. It is not significant but it is still a speedup.

The results are attached in the *appendix*. We generated a testcase with 1700 stations, 20000 trains per line for 10000 ticks (10% of the input size for performance marking).

It can be seen that in general, the wall clock time reduces by 0.14s in dxs-4114 using 40 threads and 0.04s in i7-9700 using 8 threads. While the xs-4114 using 20 threads show that the average wall clock time increase, the range of values between the not shared and shared parallel in xs-4114 are actually almost completely overlapping. The number and instructions required are generally decreasing for shared parallel compared when we have multiple parallel for. The cost-saving may not be that great because we are only reducing from 4 parallel regions to 2 parallel regions.

Hopefully, with larger number of ticks, stations and trains per line, the difference between the wall clock time for shared parallel and unshared parallel could be more significant.

Appendix

1. To run the performance measurement in a machine manually, pull the repo and run “make compareTimingSeq”. It will create a folder called “result” that contains 3 measurement outputs (our_result.out, troons_seq1_result.out, troons_seq2_result.out).
   a. our_result.out contains some the default perf stat (from the machine setting). In soctf-020, it contains task-clock, cycles, instructions, page-faults, etc of our group troons implementation.
   b. Similarly, troons_seq1_result.out and troons_seq2_result.out are the results from the given executables.

List of Lab Machines

- dxs-4114
- xs-4114
- i7-9700

Input Factors and Speedup

Table 1.

Trains	Parallel speed (40 threads)	Sequential speed	Speedup
1024	2.5142	3.4367	1.37
2048	2.4746	3.37183	1.36
4096	2.5185	3.431	1.36
8192	2.5794	3.4042	1.32

Table 2.

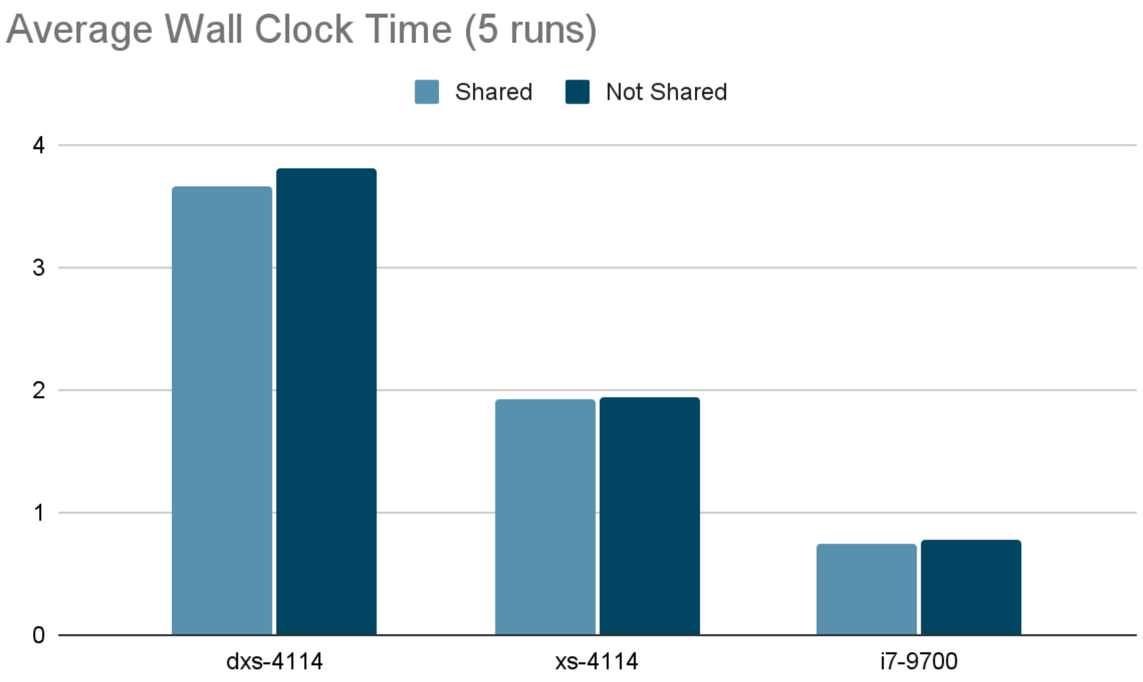
Ticks	Parallel speed (40 threads)	Sequential speed	Speedup
8192	1.8308	1.6668	0.91
16384	1.6257	1.65755	1.02
32768	31.747	18.993	0.598
65356	83.59	39.352	0.47

Table 3.

Stations/Links	Parallel speed (40 threads)	Sequential speed	Speedup
4096/6144	2.1126	2.6456	1.25
8192/12288	5.1807	7.025	1.36
16384/24576	17.187	24.0693	1.4
32768/49152	63.68	91.1	1.43

Performance Optimization Attempt Result

Machine	**wall clock**	**context-switches**	**cycles**	**instructions**
dxs-4114 (not shared)	3.8059 +- 0.0946	3,102 (+- 12.88%)	352,407,124,429 (+- 2.32%)	16,723,018,174 (+- 2.15%)
dxs-4114 (shared parallel)	3.6673 +- 0.0986	3,694 (+- 31.23%)	339,936,885,976 (+- 2.61%)	15,986,960,094 (+- 2.21%)
xs-4114 (not shared)	1.9319 +- 0.0280	989 (+- 47.16%)	86,513,909,164 (+- 1.43%)	9,832,248,701 (+- 0.72%)
xs-4114 (shared parallel)	1.9342 +- 0.0317	853 (+- 54.07%)	86,605,280,035 (+- 1.66%)	9,762,356,647 (+- 0.71%)
i7-9700 (not shared)	0.7817 +- 0.0103	901 (+- 60.15%)	22,791,865,891 (+- 1.10%)	8,047,533,760 (+- 0.16%)
i7-9700 (shared parallel)	0.73993 +- 0.00592	542 (+- 95.35%)	22,330,216,742 (+- 0.66%)	8,013,101,477 (+- 0.15%)



Average Wall Clock Time (5 runs)
Shared
Not Shared
4
3
2
1
0
dxs-4114
xs-4114
i7-9700