Tan Ming Wei - A0238178Y
Yew Wei Quan - A0234911L

Parallelization Strategy

Our program makes use of data parallelism to achieve speedup, with each thread representing a station. Each tick is done sequentially but during a tick, the stations will process the trains in their platform and links in parallel. To ensure correctness, the processing is done in two separate loops with an implicit barrier between them.

1. The first loop progresses the train in the link and queues it to the next station's holding area once it is done traveling
2. The second loop loads passengers into the train and sends a ready train to the link
3. The third loop gets the next train from the holding area and sends it onto the platform.

For each tick, the execution order is as follows:

1. A single thread will spawn new trains if needed
2. Parallel loop 1
    a. Each station will progress the train on each of its links
    b. Each station will add a train from its outgoing links to the holding area of the destination station if the train on the link has completed its journey.
3. Parallel loop 2
    a. Each station will progress the train on each of its platforms, if any.
    b. Each station will add the train on each of its platforms to the corresponding link, if the train on the platform is ready to travel.
4. Parallel loop 3
    a. Each station will dequeue the first train in the holding area queue into the platform, if the platform is unoccupied.
5. A single thread will print the state of all trains, if required.

Resolving Link Contention

Link contention is resolved using a priority queue in the holding area, which limits the number of trains eligible to enter a platform and subsequently the link to only 1.

Trains entering a station will first be added to a priority queue in the holding area, sorted by the arrival time and train ID. The holding area where the trains queue at a particular station will depend on the train's destination station.

When more than one train wants to use the same link (i.e they have the same destination station), they will first queue up at the holding area priority queue. Then, the trains will take turns entering the platform as there can only be one train on the platform at any time. Only after the first train enters the link will the next train in the queue be able to enter the platform. Additionally, only after the first train leaves the link will the next train be able to enter the link.

Key Data Structures and Synchronization Constructs

For our implementation, we made use of OpenMP's work-sharing constructs to achieve parallelisation. We declared a parallel section and partitioned loop iterations among the threads.

The `#pragma omp single` directive was used for creating trains and printing state as our implementations for the functions were not parallelizable. There was also an implicit barrier between each directive by OpenMP, which ensured that tasks in the loop were done in the right order.

```
#pragma omp parallel
  for (int_fast32_t tick = 0; tick < ticks; tick)
{
#pragma omp single
  spawn_trains_if_needed(tick, num_green_trains,
  num_yellow_trains, num_blue_trains, num_line_stations,
  train_threads);
#pragma omp for
    for (int_fast32_t i = 0; i < num_stations; i++)
      stations[i].run_traintrack(tick);
#pragma omp for
    for (int_fast32_t i = 0; i < num_stations; i++)
      stations[i].run_platform(train_threads);
#pragma omp for
    for (int_fast32_t i = 0; i < num_stations; i++)
      stations[i].run_holdingarea(train_threads);
#pragma omp single
    if (ticks - tick <= num_lines)
      print_state(tick);
  }
```

A priority queue is also used to represent a platform's holding area, so that trains could be enqueued and sorted according to arrival tick and train id (if arrival tick is the same). As race conditions might occur due to many threads updating it at the same time, we created a critical section to guard it using OpenMP's mutex. A mutex is created for each platform struct and is used to synchronize access to the priority queue.

```
omp_set_lock(&next_platform->mutex);
next_platform->holding_area.emplace(waiting_train);
omp_unset_lock(&next_platform->mutex);
```

Note: There seems to be a problem with `omp_init_lock` where the memory will still be shown as reachable in valgrind even after destroying the lock.

Structs were used to represent: Trains, Trains waiting in holding area, Links, Platforms, Stations. Each struct stores related data and the station struct contains various methods to move trains around. A vector is used to store all instances of the station struct and multiple unordered_map are used to store the instances of the platform and link structs so that they can be accessed when the program is running.

Input Size and Speedup

The number of stations affected speedup the most for our implementation. When the number of stations was increased from 1000 to 10,000, the increase in runtime in our implementation on the xs-4114 machine was **much less** compared to the increase in runtime in the sequential implementation, correspondingly, the speedup of our implementation increased significantly from 2.68 to 4.10.

The reason is that our program models stations as threads and we have reasonable suspicion that troons_seq2 did the same. When the number of stations increased, the number of parallelizable loop iterations in our code also increased. As the sequential version had to run each iteration sequentially, there is a significant speedup for our implementation.

1000 stations, 20,000 trains for each line, 0 lines printed Our implementation: 8 threads, 32 chunk size						
	xs-4114			**i7-7700**		
	troons_seq2	**ours**	**Speedup**	**troons_seq2**	**ours**	**Speedup**
Total ticks	**Wall time (s)**	**Wall time (s)**		**Wall time (s)**	**Wall time (s)**	
1,000	0.09	0.12	0.82	0.07	0.09	0.84
10,000	0.68	0.41	1.65	0.55	0.28	1.94
100,000	8.97	3.22	2.79	7.16	2.67	2.68

100,000 ticks, 1000 stations, 0 lines printed Our implementation: 8 threads, 32 chunk size						
	xs-4114			**i7-7700**		
	troons_seq2	**ours**	**Speedup**	**troons_seq2**	**ours**	**Speedup**
Trains per line	**Wall time (s)**	**Wall time (s)**		**Wall time (s)**	**Wall time (s)**	
2000	7.87	2.90	2.71	6.32	2.31	2.73
20,000	8.98	3.17	2.83	7.19	2.61	2.76
200,000	9.15	3.81	2.40	7.17	3.21	2.23

100,000 ticks, 20,000 trains for each line, 0 lines printed Our implementation: 8 threads, 32 chunk size		
	xs-4114	**i7-7700**

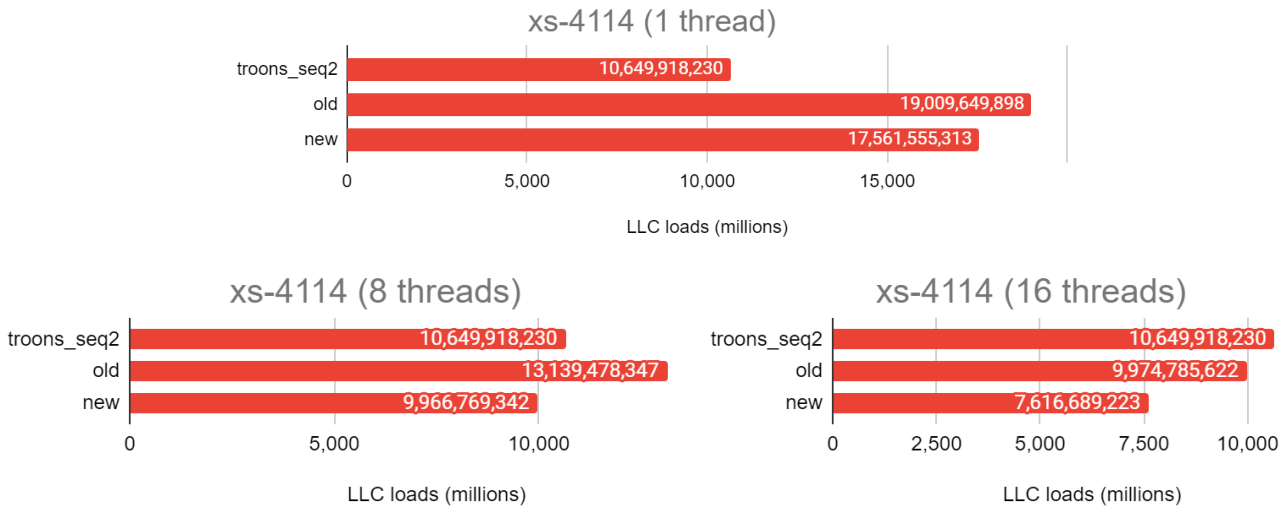
	troons_seq2	ours	Speedup	troons_seq2	ours	Speedup
No. of stations	**Wall time (s)**	**Wall time (s)**		**Wall time (s)**	**Wall time(s)**	
100	0.94	0.73	1.30	0.69	0.50	1.39
1,000	8.90	3.32	2.68	7.10	2.57	2.76
10,000	115.32	28.14	4.10	102.80	60.61	1.66

Performance Optimization

After comparing with the sequential version, we realized that our version (on a single thread) had more than twice the LLC loads compared to the sequential version. The number of LLC loads decreased when the number of threads increased, which means that having more control flows allowed the program to make better use of memory locality. Therefore, we tried to optimize our parallel implementation by further optimizing memory locality and decreasing the number of LLC loads.

To improve memory locality, we grouped related data together by creating an Edge struct to store a platform and its corresponding outgoing link. Then for each station, we stored the pointer to the next station and the pointer to the corresponding Edge together in a tuple.

The new implementation with the Edge struct led to lower LLC loads and a slightly shorter wall clock time for our program to run. As expected, there was a greater decrease when multithreading was used. However, in terms of wall clock time, the Edge struct benefitted our single-threaded version more than the multi-threaded version.



Note: The same input is used for the measurements made here: 100,000 ticks, 20,000 trains per line, 10000 stations, and 0 lines printed. Additionally, another graph illustrating the L1 cache miss rate in the old vs new implementation is included in the appendix. The number of L1 cache loads remained around the same for the old and new implementations.

Bonus

We are using the optimized implementation previously in our report. Our implementation uses OpenMP and with each thread modeled as a station. The key optimizations we attempted were in regard to memory locality.

Firstly, we changed all arrays and vectors from 2D to 1D to improve spatial locality. Secondly, we grouped all required data (next station, platform, link) for computation, so that the first memory load for computation will also bring the required data to the cache line.

Appendix

How to reproduce our results:

- Generate inputs using the generate_sample.py script located in the testcases folder in our repository. Edit the stations count, number of ticks, number of trains and lines to print before generating a sample using `python3 generate_sample.py`
- `make submission` to generate the “troons” executable
- `make bonus` to generate the “troons_bonus” executable
- `troons <input file>`
- `troons_bonus <input file>`

Testing was done on the i7-7700 and xs-4114 partitions on the distributed computing lab and measurements were acquired with the `perf stat` command with the relevant flags

Number of threads vs Chunk size

100,000 ticks, 200,000 trains per line, 1000 stations

i7-7700					
	Chunksize				
Threads	**1**	**4**	**16**	**64**	**256**
1	343.2	343.1	340.0	340.7	343.6
4	130.4	130.4	130.3	130.8	130.7
16	144.2	144.2	144.3	144.1	143.7
64	164.6	164.9	165.3	165.5	165.4
256	256.1	255.8	255.7	255.5	255.6

xs-4114					
	Chunksize				
Threads	**1**	**4**	**16**	**64**	**256**
1	409.3	407.2	402.2	399.7	401.8
4	116.9	117.5	118.5	118.6	118.2
16	46.5	46.9	46.2	47.2	46.3
64	195.6	195.8	198.8	198.2	199.9
256	527.6	528.4	528.7	529.3	530.7

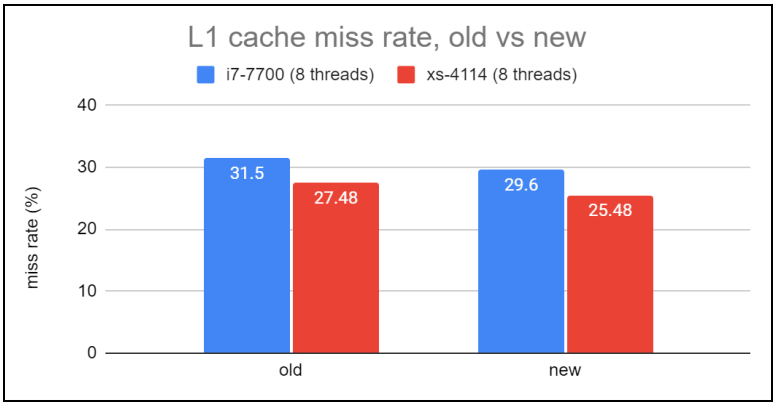
Cache loads comparison

100,000 ticks, 20,000 trains per line, 10000 stations

xs-4114 (troons_seq2)				
wall time (s)	LLC loads	LLC miss %	L1 cache loads	L1 miss %
122	10,649,918,230	0.96	42,972,901,649	42.05

xs-4114 (our implementation)										
	old					**new**				
# of threads	**wall time**	**LLC loads**	**LLC miss**	**L1 cache loads**	**L1 miss**	**wall time**	**LLC loads**	**LLC miss**	**L1 cache loads**	**L1 miss**

	(s)		%		%	(s)		%		%
1	183	19,990,649,898	3.71	105,873,103,361	28.9	154	17,561,555,313	0.83	105,338,007,844	26.8
8	27.9	13,139,478,347	0.28	106,295,239,520	28.9	26.4	9,966,769,342	0.11	105,656,508,002	26.9
16	22.2	9,974,785,622	0.16	106,830,705,427	29.0	20.2	7,616,689,223	0.09	106,189,275,933	27.1



Graph related to performance optimization attempted. The input used for the measurements is 100,000 ticks, 200,000 trains per line, and 1000 stations.

100,000 ticks, 200,000 trains per line, 1000 stations

I7-7700 (troons_seq2)				
wall time (s)	**LLC loads**	**LLC miss %**	**L1 cache loads**	**L1 miss %**
7.16	816,542,904	0.02	4,013,945,035	51.87

I7-7700 (our implementation)										
	old					**new**				
# of threads	**wall time (s)**	**LLC loads**	**LLC miss %**	**L1 cache loads**	**L1 miss %**	**wall time (s)**	**LLC loads**	**LLC miss %**	**L1 cache loads**	**L1 miss %**
1	3.14	997,740,598	0.09	11,934,199,508	31.56	2.99	834,254,906	0.11	11,880,286,331	29.52
2	3.05	987,710,348	0.09	11,951,243,416	31.53	3.06	836,150,731	0.11	11,863,495,215	29.56
4	3.21	975,274,642	0.10	11,951,032,812	31.55	3.03	865,740,854	0.10	11,843,237,348	29.58
8	3.05	996,859,063	0.09	11,940,754,696	31.50	3.06	858,575,697	0.11	11,847,336,323	29.60
16	3.11	989,221,967	0.09	11,965,264,186	31.50	3.09	855,233,065	0.10	11,846,175,713	29.58
32	3.12	992,659,011	0.09	11,951,299,262	31.51	2.98	855,099,884	0.10	11,846,269,702	29.58

xs-4114 (troons_seq2)				
wall time (s)	LLC loads	LLC miss	L1 cache	L1 miss %

		%	loads	
9.16	59,277,116	0.16	4,015,981,278	50.44

xs-4114 (our implementation)										
	old					new				
# of threads	wall time (s)	LLC loads	LLC miss %	L1 cache loads	L1 miss %	wall time (s)	LLC loads	LLC miss %	L1 cache loads	L1 miss %
1	3.95	145,134,459	0.93	11,920,300,713	27.50	3.87	148,302,047	0.88	11,822,512,712	25.51
2	3.87	144,687,147	0.90	11,924,529,598	27.48	3.90	148,248,621	0.88	11,822,329,480	25.50
4	3.76	144,827,303	0.89	11,927,599,357	27.48	3.94	148,054,675	0.87	11,835,048,302	25.48
8	3.92	144,954,539	0.90	11,928,008,049	27.48	3.90	148,482,984	0.87	11,828,259,336	25.48
16	3.98	144,783,275	0.93	11,926,115,692	27.48	3.90	148,179,612	0.92	11,830,737,459	25.48
32	3.82	144,637,783	0.91	11,919,658,846	27.49	3.78	148,686,116	0.90	11,838,120,418	25.46