Assignment 3

Algorithm Design and Parallelisation

We decided to split the task down to the platform level. Each platform also accounts for its outgoing link. Each platform has a hashmap to store which platform to send its train to based on what line the train is on, as well as an array of input platforms to identify which platform it will be receiving a train from. Each platform stores a map from (platform ids on current rank) to platform pointer in order to quickly send a train into a platform on the local rank. The array of platforms not on the local rank is also stored for easy iteration.

Since each link has a minimum travel time, to reduce requests, we send the earliest tick the link might not be free (the "recheck tick") along with the optional train data between platforms. For each platform, we use a priority queue to store the received recheck ticks and retrieve the platforms with recheck ticks corresponding to the current tick. The recheck tick is also stored to send a message at that tick.

First, the platforms are initialised with the station and line data. There are at most 2*(max_line_len-1)*lines platforms. Then the AUCTION algorithm developed by Khan and Li (1995) is used. This distributes seeds (chosen round-robin from the terminals, or randomly) to each rank, whence they use BFS to assign higher priority to closer platforms. They send their priorities to the root rank using `MPI_Gather`, which assigns each platform to the rank that gave it highest priority and `MPI_Bcast`s the assignments to all ranks. This algorithm ensures platforms are roughly equally distributed and the number of links crossing ranks is approximately minimised.

At each tick, each rank iterates through the platforms it is assigned.

1. If that platform is a terminal platform, and there are trains to spawn, a train is inserted into the platform holding area.
2. Each platform calls `MPI_Isend` for the output platforms on other ranks, if its stored recheck tick has arrived. If a train is actually being output to one of those platforms, it is sent, otherwise an invalid train is sent (id = -1). The new recheck tick is also sent, if the link is now empty, it is now+link_length+1, if it is occupied, it is link_enter_time+link_length. It is guaranteed a train cannot exit before this tick. The recheck tick is stored so the platform can resend messages at that time.
3. Each platform calls `MPI_Irecv` for every input platform on another rank whose recheck tick has arrived, to receive a new recheck tick and optionally train data.
4. The platform checks whether it can push a train from the holding area into the 'platform area'.
5. The newly received recheck ticks are pushed into the priority queue of input platforms along with their platform ids.
6. If the tick needs to be saved, the platform saves the states of all trains in the holding area, in the 'platform area' and in the outgoing link assigned to this platform

After all ticks are simulated, rank 0 gathers all the states from all the other ranks. It first gathers the number of states each rank has with `MPI_Gather`, then calls `MPI_Gatherv` to gather all the states. Then rank 0 sorts the states according to tick and then in ascending lexicographical order, then prints to `stdout` .

How deadlocks, race conditions and synchronisation issues were avoided

As we have no guarantees on the structure of the graph, we used `MPI_Isend` and `MPI_Irecv` to send the trains to different ranks to prevent deadlock. To handle the cases where there is no train to actually send, we maintained the invariant that a platform needs to call `MPI_Isend` for every remote output platform it has for each line, when the recheck tick has arrived. If there is no train that needs to be sent, a `INVALID TRAIN` data is sent instead. Each platform also calls `MPI_Irecv` for every remote input platform it has when its recheck tick arrives, even if there is actually no train to receive. This way, each `MPI_Isend` is paired with a `MPI_Irecv` .

There can be race conditions, as order of receives is not guaranteed. For example, if rank 0 stores platform 0 and 1, and tries to receive a train for platform 0 from rank 1, it might receive from any platform from rank 1 sending to platform 1. Hence for each platform to receive the correct data, we must ensure `tag` corresponds to a distinct (sender platform id, receiving platform id). We can do this by making `tag = sender_platform_id * num_of_platforms + receiver_platform_id`. The platform ids are 0, 1, 2, … `num_of_platforms - 1` . This way, each platform gets the correct train, when its rank calls `MPI_Irecv`. Since MPI guarantees that messages are non-overtaking, it is guaranteed that the sequence of trains sent from any platform to any other does not change, so the simulation is deterministic.

We did not use any explicit MPI construct to ensure synchronisation. Each rank calls `MPI_Waitall` after all its platforms have called `MPI_Isend` and `MPI_Irecv` to ensure the send and receive are completed and the data can be safely accessed.. If a couple of ranks could progress faster through the ticks, this means these ranks form a separate connected component, and progressing faster is still safe. If not, all platforms will wait at `MPI_Waitall` . Hence there is no need for `MPI_Barrier` . After all ranks have simulated all the ticks, they will be synchronised when they call `MPI_Gather` for rank 0 to start collecting all the states.

Changing number of stations and `max_line_len` to compensate

Test Command: `python3 gen_test.py S 10 10 500 S/2 500 > [file]`

The mpi process is run with 4 processes on the i7-7700 partition.

S	bench_seq on i7-7700 partition	trains on i7-7700 partition only, 4 processes	Speed up

	Command: `srun -p i7-7700 ./bench_seq [file] > /dev/null`	Command: `salloc -p i7-7700 -N 1 -n 4 mpirun -np 4 ./trains [file] > /dev/null`	
30	0.493949	0.17812	2.773
50	0.648621	0.265227	2.445
70	0.776215	0.366953	2.115
90	0.828486	0.360048	2.301
100	0.825592	0.376243	2.194

The speed up seems to decrease as the number of stations increased and `max_line_len` increased. This is possibly due to constant overhead on the sequential benchmark that is a lesser proportion of the time as platforms increase. Work is split down to the platform level, and this finer granularity means that percentage of work done by each process is roughly the same even though number of stations and `max_line_len` increased.

Changing number of ticks to simulate

Test Command: `python3 gen_test.py 30 10 10 500 15 N > [file]`

N	bench_seq on i7-9700 partition Command: `srun -p i7-9700 ./bench_seq [file] > /dev/null`	trains on i7-9700 partition only, 8 processes Command: `salloc -p i7-9700 -N 1 -n 8 mpirun -np 8 ./trains [file] > /dev/null`	Speed up
500	0.4370	0.1050	4.161
700	0.6299	0.1495	4.213
900	0.8292	0.2085	3.976
1100	1.0203	0.2463	4.142
1300	1.2237	0.3013	4.061

The speed up did not vary much as the number of ticks increased. This is because the number of ticks does not change the percentage of ‘work’ assigned to each processor. Each processor still has almost an equal number of platforms to simulate.

Comparing speed when running on single node vs on 2 nodes
Test command: `python3 gen_test.py 100 10 10 500 50 500 > [file]`

bench_seq Command: `srun -p xs-4114 ./bench_seq [file] > /dev/null`	Trains, 1 node Command: `salloc -p xs-4114 -N 1 -n 4 mpirun -np 4 ./trains [file] > /dev/null`	trains, 2 nodes Command: `salloc -N 2 -n 4 -p xs-4114 mpirun -np 4 --map-by node ./trains [file] > /dev/null`
1.184	0.529193	0.533936

We chose to run on the xs-4114 partition for our 2 nodes test, to ensure the cpus used are the same, and any difference in timing is not due to the different execution power of the cpu (i.e. higher clock frequency, more cache etc). When run on 2 nodes, communication is slower, as bytes have to pass through a network. On 1 node only, bytes only need to be sent to different cores on the same cpu, which is faster.

Bonus

To support a variable number of lines, we only needed to replace hardcoded constants with the input number of lines and calculate the sequence of lines from the constant LINES.

Our implementation processes link and train failures after trains have been spawned but before they advance, so a train leaving a platform to transit at tick i that fails at tick i will be immediately decommissioned.
Before the simulation starts, the train failures are sorted by tick and the link failures are stored in their respective platforms and then sorted by tick. A pointer is stored for each of these lists. Every tick, for all lists above, the pointer is advanced to find and activate all failures at this tick.
Each link stores whether a link or train failure has happened, to be reset when the train exits. When a link/train failure happens for the first time, the end_time of the link (if occupied) is added by end_time-tick-1, doubling the ticks before end_time arrives. The appropriate amount of maintenance is also added. The link is considered not free even if empty if maintenance is nonzero, and it ticks down every tick.
When a train failure happens, each local platform loops through all present trains to find the train. If it is in holding or platform, it is decommissioned, if it is in the link, it is marked failed and the link is affected as above. The boolean failed has also been added to the scheduled train datatype so failed trains can be transferred as normal. However when a failed train reaches the holding queue of any platform it is immediately decommissioned.
Finally logic was added to enumerate and display decommissioned trains by sending them as states.

Measuring speedup

Test command: `python3 gen_test.py --num_train_lines 10 30 10 10 500 15 500 > [file]`

Followed by manual addition of 10 link failures and 10 train failures

Bonus, 1 node, 2 processes Command: `salloc -n2 -N1 -p xs-4114 mpirun ./bonus [file] > /dev/null`	Bonus, 2 nodes, 8 processes Command: `salloc -n8 -N2 -p xs-4114 mpirun ./bonus [file] > /dev/null`
1.10542	0.48889

There is a speedup of 2.261.

References:

M. S. Khan and K. F. Li, "Fast graph partitioning algorithms," IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing. Proceedings, Victoria, BC, Canada, 1995, pp. 337-342, doi: 10.1109/PACRIM.1995.519538.