CS3210 Assignment 3 Vijayaraghavan Vishnuprasath E0958101

1) Description.

- Parallelisation strategy and design

I parallelised based on stations. So, each process is responsible for a set number of stations and all the outgoing links from these stations. To simplify things, I assigned station i to the (i%total_processes)'th process. This way all process get an equal amount of stations.
Each tick has four broad stages.

1) Updating links. This includes reducing time left for any train on the link. Placing any train that has finished travelling the link into a temporary buffer of the destination station.
2) Updating platforms. This includes reducing loading/unloading time left for any train left on the platforms for the station, pushing the train into the corresponding link if the train is done loading/unloading and the link is free.
3) Spawning of trains. This includes spawning necessary trains at terminal stations and placing them in the buffers of that station. Then, sorting the buffer of each station based on train_id.
4) Updating the holding area of each platform. This includes inserting all the trains from the buffer of that station into the corresponding queue(holding area) for their platform. Then, if platforms are free, remove the front element from that platform's queue(holding area) and place it on the platform.

Communication between processes is only required in step 1 when the process responsible for a link places the train into the buffer of another station in another process.

In step 3, each process might need to know ids of trains spawned in stations that process is not responsible for. This might need communication with other process. To avoid this, each process calculate all the trains spawned in every tick and then based on this, the process assigns appropriate ids to train spawned in stations that process is responsible for.

All other steps don't require communication between processes.

To collate the results about each train in the end, each process gathers information about all the trains in the platforms/links/holding area it is responsible for into space separated string. Then, process of rank 0 gathers all these strings from all the processes and splits based on space, sorts them and outputs them.

- Deadlocks/Race conditions

Deadlocks are only possible during MPI communication calls.

In Step 1, sll the stations in all the processes loop through their neighbours that are in the same process and update the link connecting them if necessary. This part needs no inter process communication. So, there is no deadlock here.
After that every process loops through every link that has endpoints in different processes, and MPI Isend is called from the process where the link begins and the process where the link ends calls MPI Irecv. Since there is matching number of send and receive and non-blocking calls are used, deadlocks are avoided.

When process of rank 0 is collating results about each train in the end, process 0 initiates p-1 recv calls while every other process initiates 1 send call each, where p is number of process. So, there is matching number of sends and receives and each process only sends or receives. So, there is no deadlock even with blocking calls.

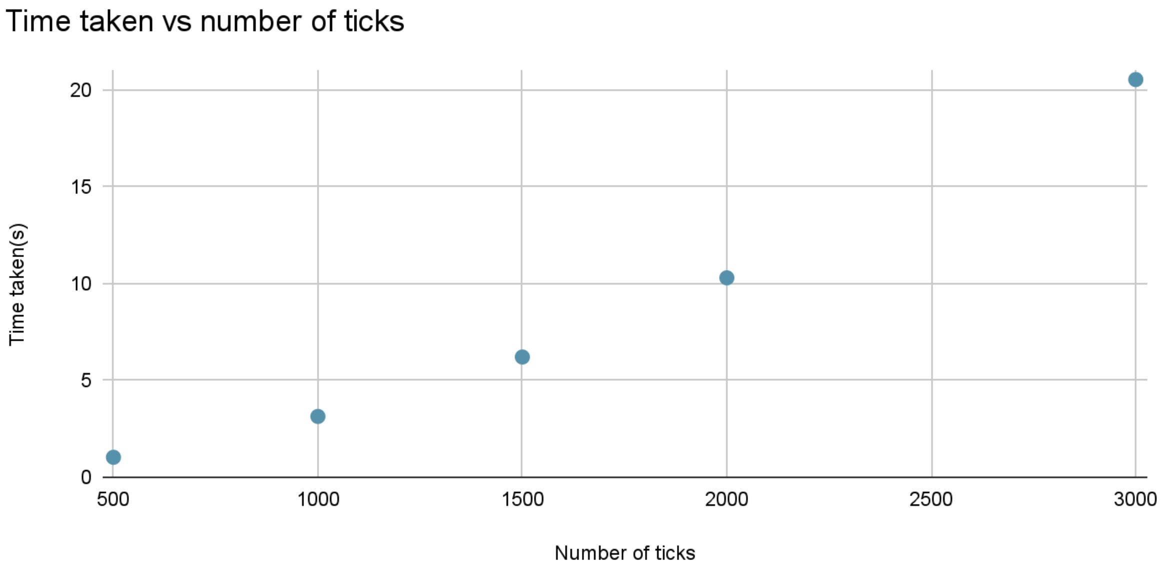
Race condition is only possible when different process modify the same variable. In this program, this is only possible when multiple process attempt to shift the train from their links into buffer of the same station in the same tick.
To avoid this, when trains are received into the buffer of a destination station, they are stored in the order in which source stations of these trains appear in the neighbours list of the destination station. This way exactly one train is placed each place in the buffer of a station avoiding race conditions.

- Mpi constructs

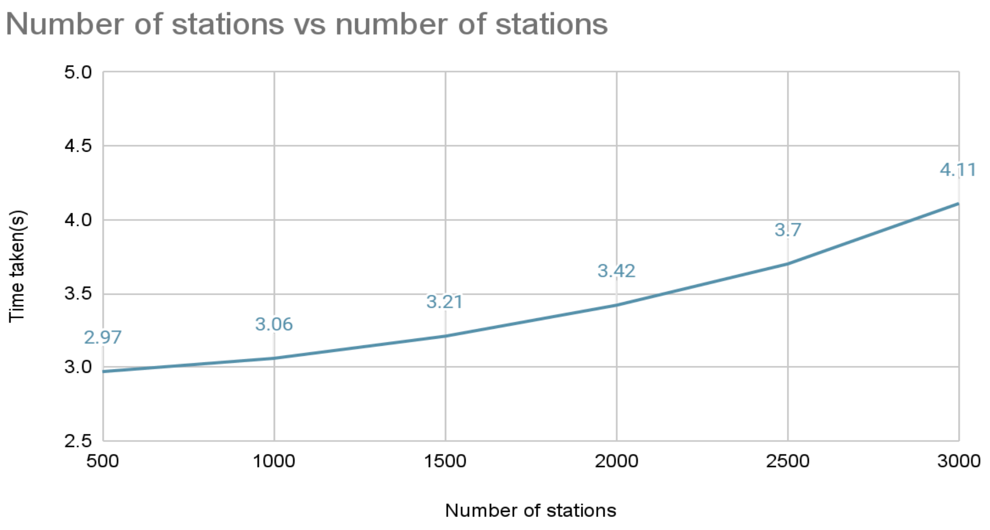
In step 1, Mpi Isend is used to push trains from links in one process into buffer of stations in another process. Mpi Irecv is used to receive these. As explained previously these help to prevent deadlocks. MPI_waitall was also used to ensure all the communications were finished before moving onto next steps.
Since each process is responsible for multiple stations, there might be multiple send/receive calls each corresponding to a station-station pair between two processes. So tags were used during MPI calls. The tag was set to source*num_stations + destination. This way each station-station communication gets an unique tag ensuring correct information is put into the buffer.

When rank 0 process is gathering train information from other process at the end, two MPI calls are used. Since, each process might have variable length of string to send to process 0, the first send/receive call transmits the length of the string to process 0. The second send/receive call transmits the actual string itself to process 0.
For this I only used blocking Send and Recv MPI calls since there is no danger of deadblocking as explained previously.

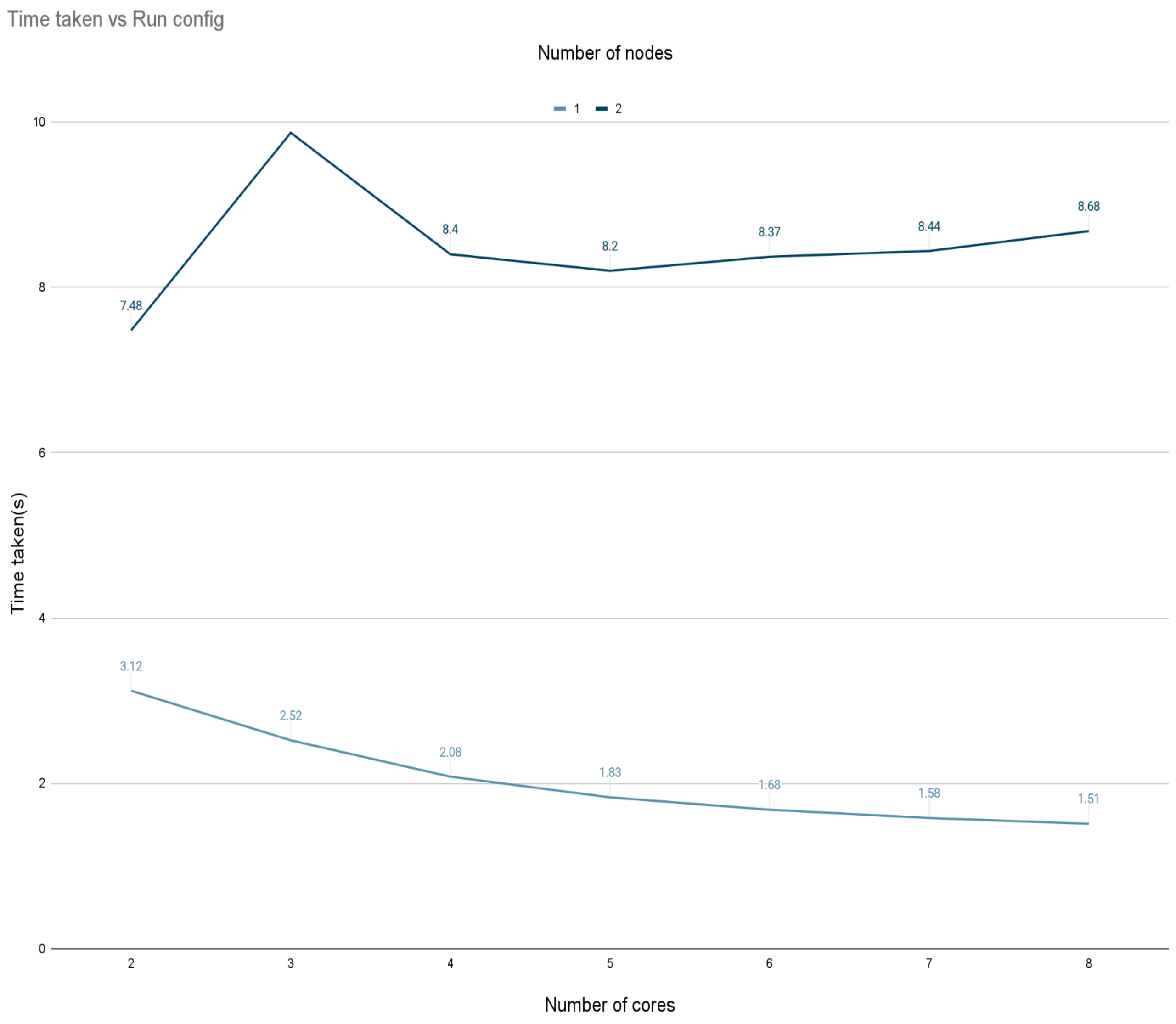
2) Analysis



The total time taken approximately doubles for every 50% increase in number of ticks. This is because only number of ticks was varied in test cases. Max platform waiting time was kept constant. So, number of platform waiting time computations rise more than linearly compared to number of ticks. Adding a counter to check how many times waiting time is generated, we get, 22000,81000,16000 times respectively for 1000,2000,3000 ticks. The ratio of these closely follow the ratios of time.
And since computation of waiting time is intensive, this results in total time also rising more than linearly.



In general, time taken increases with the number of stations but the increase is much weaker than when the number of ticks increased.



In general, as the number of cores increases the time taken decreases for single node configuration. This is expected as computation heavy tasks like platform waiting time generation can be spread over many cores which reduces total time taken despite the communication overhead. However, this was not true when running with 2 nodes as the computation might not be enough to justify extra cost of inter-node communication.

For the same number of total cores, 1 node configuration took shorter time than 2 node configuration. Running mpi with --mca btl_base_verbose 100 reveals that when one node is used, vader(shared memory message transport) is used while for more than one node only tcp is used which relies on the slower interconnects between nodes. So, communication time between processes is longer, resulting in longer computation time.

Appendix:

Raw data

Number of ticks vs time taken

Number of ticks (x)	Time taken/ s
500	1.03
1000	3.14
1500	6.21
2000	10.30
3000	20.54

Run command: salloc -p xs-4114 --nodes 1 --ntasks 2 mpirun trains test_file
Test generate command: python3 gen_test.py 500 10 10 500 500 x,   where x is number of ticks

Number of stations vs time taken

Number of stations	Time taken /s
500	2.97
1000	3.06
1500	3.21
2000	3.42
2500	3.70
3000	4.11

Run command salloc -p xs-4114 –nodes 1 -ntasks 2 mpirun trains test_fil
Test generate command python3 gen_test.py x 10 10 500 x/2 1000,   where x is number of stations.

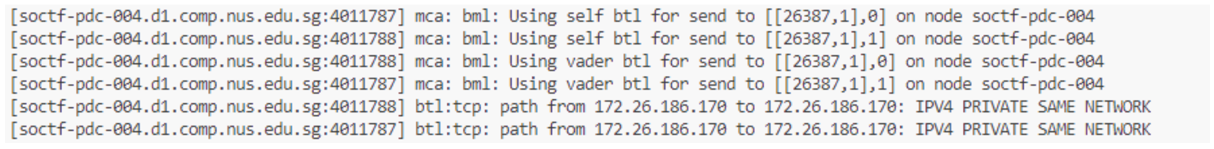
Run configuration vs time taken

(number of nodes, total number of cores)	Time taken (s)
(1,2)	3.12

(1,3)	2.52
(1,4)	2.08
(1,5)	1.83
(1,6)	1.68
(1,7)	1.58
(1,8)	1.51
(2,2)	7.48
(2,3)	9.87
(2,4)	8.40
(2,5)	8.20
(2,6)	8.37
(2,7)	8.44
(2,8)	8.68

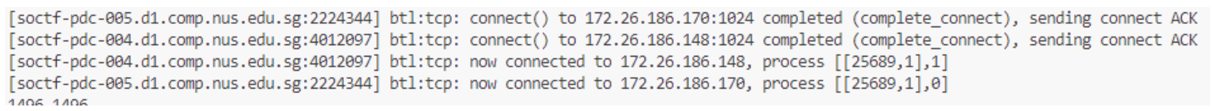
Run test command:
salloc -p xs-4114 --nodes x --ntasksyy mpirun trains test_file, where x and y are number of nodes and total number of cores respectively.

Test generate command: python3 gen_test.py 500 10 10 500 500 100
For salloc -p xs-4114 --nodes 1 --ntasks 2 mpirun --mca btl_base_verbose 100 trains test_file



```
[soctf-pdc-004.d1.comp.nus.edu.sg:4011787] mca: bml: Using self btl for send to [[26387,1],0] on node soctf-pdc-004
[soctf-pdc-004.d1.comp.nus.edu.sg:4011788] mca: bml: Using self btl for send to [[26387,1],1] on node soctf-pdc-004
[soctf-pdc-004.d1.comp.nus.edu.sg:4011788] mca: bml: Using vader btl for send to [[26387,1],0] on node soctf-pdc-004
[soctf-pdc-004.d1.comp.nus.edu.sg:4011787] mca: bml: Using vader btl for send to [[26387,1],1] on node soctf-pdc-004
[soctf-pdc-004.d1.comp.nus.edu.sg:4011788] btl:tcp: path from 172.26.186.170 to 172.26.186.170: IPV4 PRIVATE SAME NETWORK
[soctf-pdc-004.d1.comp.nus.edu.sg:4011787] btl:tcp: path from 172.26.186.170 to 172.26.186.170: IPV4 PRIVATE SAME NETWORK
```

For salloc -p xs-4114 --nodes 2 --ntasks 2 mpirun --mca btl_base_verbose 100 trains test_file



```
[soctf-pdc-005.d1.comp.nus.edu.sg:2224344] btl:tcp: connect() to 172.26.186.170:1024 completed (complete_connect), sending connect ACK
[soctf-pdc-004.d1.comp.nus.edu.sg:4012097] btl:tcp: connect() to 172.26.186.148:1024 completed (complete_connect), sending connect ACK
[soctf-pdc-004.d1.comp.nus.edu.sg:4012097] btl:tcp: now connected to 172.26.186.148, process [[25689,1],1]
[soctf-pdc-005.d1.comp.nus.edu.sg:2224344] btl:tcp: now connected to 172.26.186.170, process [[25689,1],0]
1496 1496
```