CS3210 Assignment 3

Part1: Description of algorithm

1. Your program's parallelisation strategy and design

Our program parallelising strategy and design is to assign each process to manage a distinct subset of contiguous stations based on their rank and the total number of processes. Each process will handle all train operations such as loading, unloading and moving of trains onto platform and links within the subset of stations. When there is inter-process communication, it is mainly done by MPI_Send and MPI_Recv to send data over from one process to another. Spawning of trains are mainly done by master and then distribute to the terminal stations in respective processes. The different phases such as train spawning, movement and platform management are managed and synchronised by MPI_Barrier to maintain consistent states. At the end, all the trains will be gathered to the master to sort and print out the stations.

2. How deadlocks/race conditions are resolved in your implementation

Deadlocks are mainly avoided by having proper synchronisation in place, where communication between processes follow a predictable order of having an MPI_Send to always be matched with an MPI_Recv using the same tags and communicators, ensuring that there will be no deadlocks caused by mismatched sends and receives. Circular waits are also avoided because every process first send out all outgoing data before receiving all incoming data, hence ensuring that no deadlock can occur. To ensure that every receive will always be matched with a send, even when there is no data sending, we will send a dummy train over to ensure that the MPI_Recv will definitely receive something and not be stuck forever causing a deadlock. There is also barriers being implemented to synchronise the stages of the stimulation to eliminate the chances of process entering mismatched communication states

We avoided race condition by firstly ensuring that any form of updates such as the updating of stations is within its own subset of stations, since it is not possible for any process to modify the state of local stations or their trains in other processes. In addition, trains crossing boundaries between processes are also explicitly communicated and ensuring that one process is responsible for any given train will prevent the scenario of two processes updatin the same train concurrently, preventing a race condition from happening.

3. The key MPI constructs used to implement this strategy and why they are used

The following MPI Constructs are being used with their rationale:

- MPI_Send and MPI_Recv - This is used when trains move from one process's station to another, the data is send using MPI_Send and received with MPI_Recv. This enforce point to point communication, preventing concurrent updates to the same train.
- MPI_Type_create_struct - This is used to define custom MPI datatypes for the train structure to allow processes to send and receive entire train objects and preserve the structure fo complex data and ensuring consistency on train attributes.
- MPI_Barrier - This is used for synchronisation purposes to avoid inconsistencies and aimed to maintain a clear separation between simulation phases.
- MPI_Gather and MPI_Gatherv - This is used at the end of the simulation for every tick, where all the trains data from the different stations will be gathered at the master

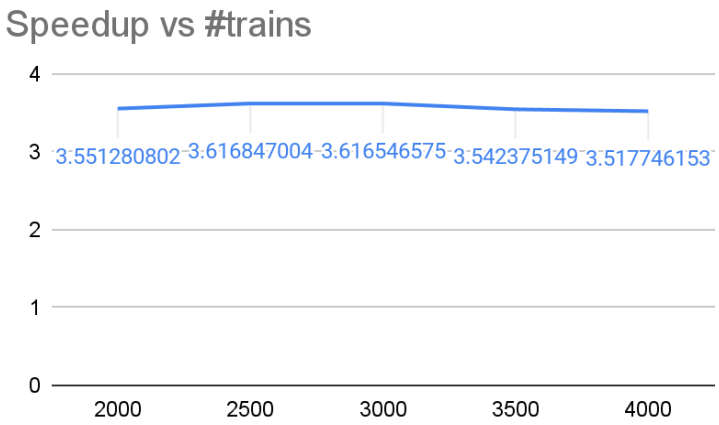
process to be sorted and printed. MPI_Gather is used to gather the size of the data that the various processes is sending and MPI_Gatherv is used to gather all the train data from the various processes, this is mainly designed to efficiently handle variable-sized data because the number of trains at different processes will be different throughout the simulation.

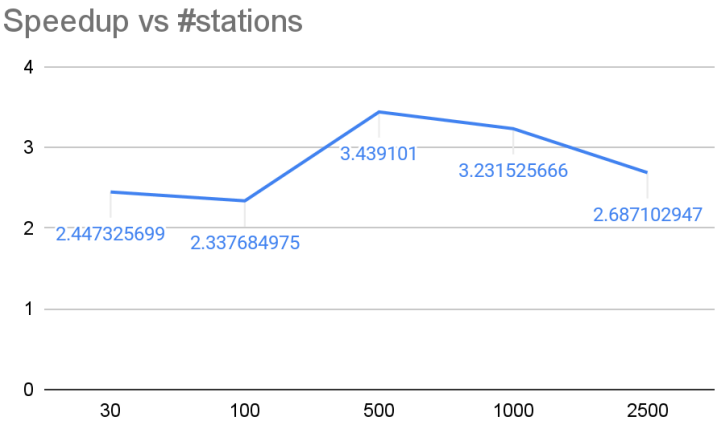
- MPI_Probe - This is used to allow the receiving process to inspect the incoming message before receiving them, to determine the size fo the data, allow the receiving process to handle varying number of trains that is being sent over from the sender, this enables the receiver to allocate sufficient memory for incoming message based on the number of trains being send, providing flexibility and reducing memory overhead.

Part2: Analysis of which parameters affect speedup the most

For this part, we will stablize everything else when testing an effect of a parameter. Also, for each parameter, we will have a line graph showing how and how much it affect the speedup. In the appendix, we will explain how the execution time and speedup is gained by showing how our test cases are generated and what are other settings when running.

Station and Train

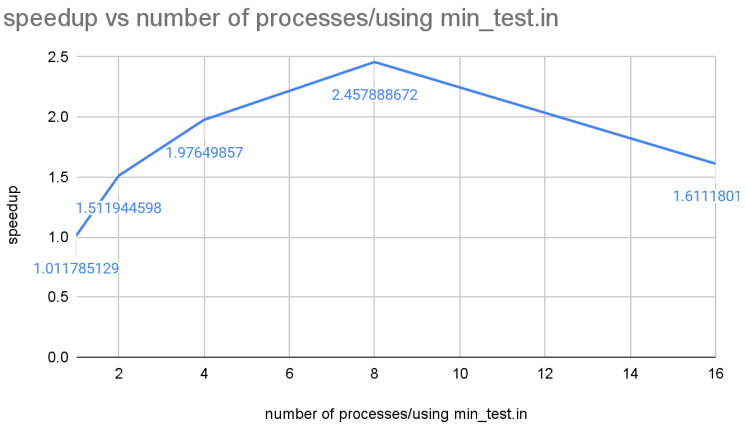


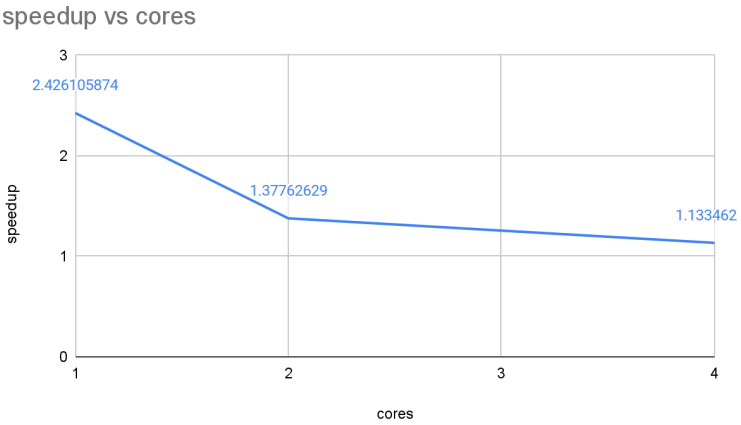


The number of stations affects speedup by determining how evenly the workload is distributed among processes. With a smaller number of stations from 30-100, the parallel implementation underperforms due to underutilization of processes and significant communication overhead between processes as compared to actual computation. As the number of stations increases from 500 - 1000, speedup improves due to better utilisation of processes and the computation outweigh the communication overhead. As the number of stations becomes very large of 2500, the inter-process communication for managing adjacent stations start to dominate which cause the speedup to reduce despite the increase workload.

The number of trains does not lead to significant changes of speedup as compared to running the same workload sequentially, as the number of trains increase, the speedup decreases slightly as the increase in the number of trains can increase dependencies which leads to synchronisations and data exchange overload to increase.

Number of processes and number of cores

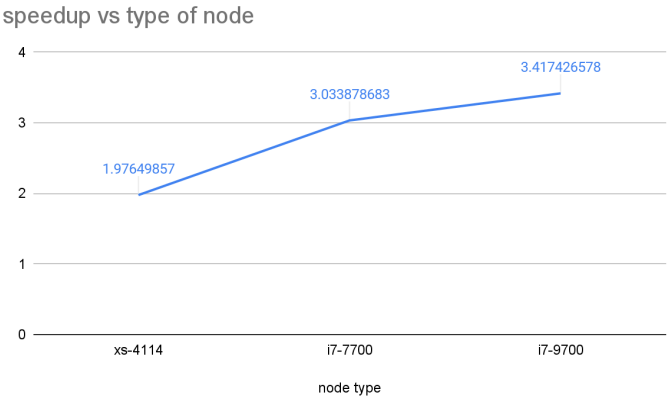


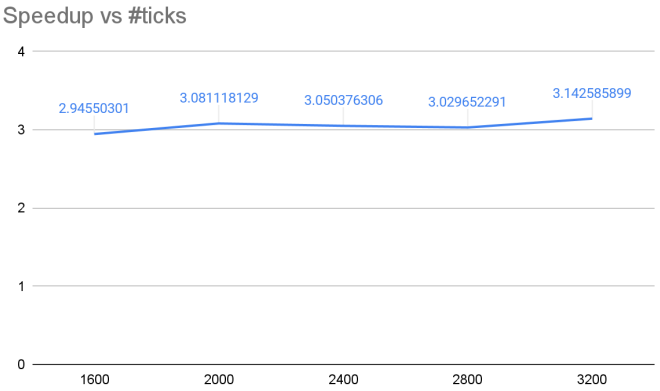


The number of processes determines the granularity of the workload distribution. With too few processes, the workload per process is large, and the benefits of parallelism are limited. As the number of processes increases, the workload is distributed more finely, leading to better utilization of computational resources. However, excessive processes result in increased communication overhead, especially if processes frequently exchange data. For instance, with 8 processes handling a moderate train count, speedup is optimal, but increasing to 16 processes leads to diminishing returns due to higher communication costs and reduced work per process. In addition, in the xs-4114 core, as it has 10 cores and 20 threads, since we are using a single core, by having 16 processes, there will be contention as the processes are competing for the limited number of cores, hence this leads to increased context switching which could result in the speed of the program to decrease, hence leading to the speedup to be adversely affected.

For a single machine, the speedup is the highest of 2.42x, it then decreases to 1.37x for 2 machines and then 1.13x for 4 machines. The reason for the decrease of speedup is that as more machines are used, communication between machines must occur over the network rather than within a single machine, this network communication is much slower than within a single machine, this leads to significant slowdowns, hence reducing the efficiency of parallelism as significant latency is added, hence this limits the scalability of the parallel implementation.

Node type and Ticks





The type of node influences speedup through its hardware characteristics, including core count, clock speed, and interconnect performance. In this assignment, we used three node types: **Xeon Silver 4114**, **i7-7700**, and **i7-9700**.

- **Xeon Silver 4114**: With 10 cores and a lower base clock speed of 2.2 GHz, this node performs well for workloads requiring higher parallelism but suffers in performance when number of processes is lower due to its lower clock speed.
- **i7-7700**: This processor has 4 cores with a higher base clock speed of 3.6 GHz, making it efficient when fewer processes are utilised but leads to a limited speedup when many processes are utilised due to contention of cores.
- **i7-9700**: With 8 cores and a clock speed of 3.0 GHz. Speedup scales better on this node compared to the i7-7700 due to the additional cores and better parallel processing capability despite the lower clock speed as compared to the i7-7700.

The number of ticks affects speedup by determining how long the simulation runs, which directly scales the computational workload.

When the number of ticks is small, the workload is low, and communication overhead between processes dominates. In this case, speedup is minimal because the processes spend more time coordinating than computing. As the number of ticks increases, the computational workload grows proportionally. This reduces the impact of communication overhead, leading to better speedup. Longer simulations allow processes to spend more time on actual computation, making parallelism more effective.

After all our experiment, we find that the number of processes is the most dominant factor affecting the speedup of the program. That is because increasing the number of processes initially improve speedup by effectively distributing the workload, reducing computation time per process. However, beyond a certain point, the benefits diminish due to increased communication overhead and synchronization delays between processes, which can offset the gains from parallelization. The results show that speedup is maximized when the number of processes aligns with the available physical cores, ensuring optimal resource utilization with minimal contention. Thus, the number of processes is a critical parameter that must be carefully balanced to achieve the best performance in parallel implementations.

Appendix

1. How our results are produced

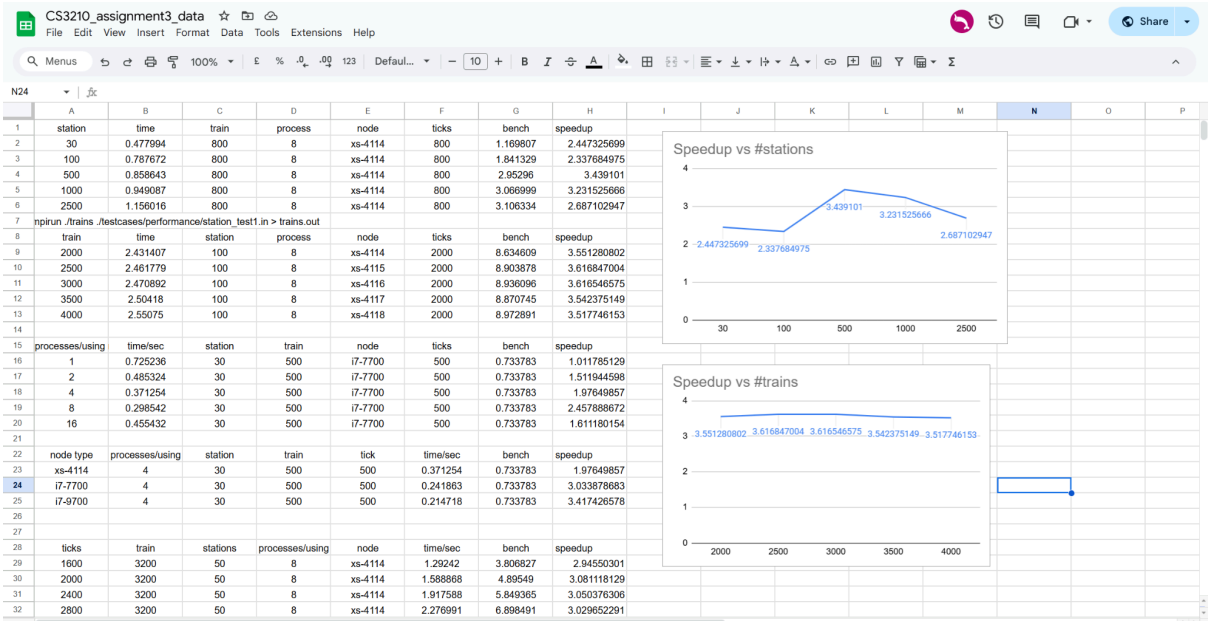
For getting the execution time of our program, we run: salloc --nodes 1 --ntasks 8 --constraint=xs-4114 mpirun ./trains ./testcases/performance/station_test1.in > trains.out

Where the station_testx.in is generated by:
python3 gen_test.py xx 10 10 800 1250 800 > ./testcases/performance/station_testx.in
And xx is where we put the number of varied stations, and it applies to other parameters also.

For getting the execution time of bench program, we run: srun ./bench_seq ./testcases/performance/station_test5.in > bench.out as required in the FAQ

2. Settings when we run the program

We have record all our raw data into an excel file, you can access this by link, below is the screenshot of this



station	time	train	process	node	ticks	bench	speedup
30	0.477994	800	8	xs-4114	800	1.169807	2.447325699
100	0.787672	800	8	xs-4114	800	1.841329	2.337684975
500	0.858643	800	8	xs-4114	800	2.95296	3.439101
1000	0.949087	800	8	xs-4114	800	3.066999	3.231525666
2500	1.156016	800	8	xs-4114	800	3.106334	2.687102947

mpirun ./trains ./testcases/performance/station_test1.in > trains.out

train	time	station	process	node	ticks	bench	speedup
2000	2.431407	100	8	xs-4114	2000	8.634609	3.551280802
2500	2.461779	100	8	xs-4115	2000	8.903878	3.616847004
3000	2.470892	100	8	xs-4116	2000	8.936096	3.616546575
3500	2.50418	100	8	xs-4117	2000	8.870745	3.542375149
4000	2.55075	100	8	xs-4118	2000	8.972891	3.517746153

processes/using	time/sec	station	train	node	ticks	bench	speedup
1	0.725236	30	500	i7-7700	500	0.733783	1.011785129
2	0.485324	30	500	i7-7700	500	0.733783	1.511944598
4	0.371254	30	500	i7-7700	500	0.733783	1.97649857
8	0.298542	30	500	i7-7700	500	0.733783	2.457888672
16	0.455432	30	500	i7-7700	500	0.733783	1.611180154

node type	processes/using	station	train	tick	time/sec	bench	speedup
xs-4114	4	30	500	500	0.371254	0.733783	1.97649857
i7-7700	4	30	500	500	0.241863	0.733783	3.033878683
i7-9700	4	30	500	500	0.214718	0.733783	3.417426578

ticks	train	stations	processes/using	node	time/sec	bench	speedup
1600	3200	50	8	xs-4114	1.29242	3.806827	2.94550301
2000	3200	50	8	xs-4114	1.588868	4.89549	3.081118129
2400	3200	50	8	xs-4114	1.917588	5.849365	3.050376306
2800	3200	50	8	xs-4114	2.276991	6.898491	3.029652291

cores	processes	station	train	tick	time/seq	bench	speedup	node
1	8	30	500	500	0.302453	0.733783	2.426105874	xs-4114
2	8	30	500	500	0.532643	0.733783	1.37762629	xs-4114
4	8	30	500	500	0.647382	0.733783	1.13346214	xs-4114