Parallelisation Strategy and Design

There is exactly 1 platform per outgoing link regardless of how many different station lines share this link, each with their own corresponding holding area queue. Hence, we can designate platforms as the smallest unit of task and divide all the platforms evenly among the number of processes available. The number of platforms per process is rounded up so that the last process would not get significantly more work than the others.

Each platform would be in charge of the holding area queue and the outgoing link, and would only need to inform the other platforms of trains that left that link, minimising the communication needed. However, we are unable to simply send the trains to a destination station with this design and would have to send trains to the correct platform based on their colour. Hence, each station communicates with exactly the number of station line colours its link has for outgoing trains, and communicates with this same number of stations for incoming trains. It would not be more than the number of colours in the link since it is impossible to send/receive trains of other colours.

As there will likely be many more platforms than processes available, I have assigned the platforms amongst processes such that platforms that are linked together (platform_1 can travel to platform_2) are grouped within the same process as much as possible. This minimises the number of communications needed with other processes, and maximises the number of faster "self-communication". This was done by allocating to each process as many platforms within a single station line (same colour) in the forward direction, followed by its backwards direction, and followed by the next station line as possible. However, duplicates need to be taken care of since different lines can share the same platform.

How Deadlocks/Race Conditions are resolved

Since there is guaranteed to be cyclic dependency (each station line is a loop, and there may be self-sending processes), the blocking MPI_Send() would result in a deadlock if all platforms send before receiving. Hence I used the non-blocking MPI_Isend() for all sends instead, making such a deadlock no longer possible. This is likewise used for self-sending on rank 0 for printing of outputs, which would otherwise deadlock.

Since there is no way for the receiving platform to know whether the sending platform has any trains to send, it would always have to try to receive from the sending platform. The sending platform however knows whether there is a train to send and could choose not to send. However, this would result in a deadlock since the receiving platform will forever be trying to receive. Hence, a train number of "-1" is sent instead to indicate that there is no train to send, fixing this deadlock.

Each platform receives incoming trains using the non-blocking MPI_Irecv() and then sorts and modifies the values in place. However, this could result in a race condition since there could be interleavings of receiving values and modifying/sorting values. Moreover, receiving of values could even go on to the next tick if the receives were too slow, causing 2 different tick's receives to interleave with the same receive buffer. Hence, the receive requests are checked using MPI_Waitall() to ensure all receives are done before modifying/continuing.

Key MPI Constructs used to implement this strategy and why they are used

The non-blocking MPI_Isend() and MPI_Irecv() are used to overlap communication with computation to speed up to program.

MPI_Waitall() is used to wait for all the MPI_Irecv() to be done before reading and modifying the data, preventing data races.

The blocking MPI_Recv() is used for receiving trains to print in rank 0 since there is no more computation to be done by rank 0, and it would have to immediately wait.

Since rank 0 is receiving 2 types of values from other platforms: incoming trains and trains to print, there could possibly be interleavings between the 2 types of values, which is especially bad if any platforms were too fast (Eg: platform_3 sends both outgoing trains and trains to print to rank 0 before platform_4 starts sending outgoing trains to rank 0), causing rank 0 to be unable to differentiate between the 2 types. MPI_Barrier() is hence used for all platforms to send and receive trains completely before sending trains to print.

MPI_Barrier() is also used at the end of each tick to synchronise all processes to the same tick, preventing erroneous states such as receiving trains from the same platform twice.

Analysis of which Parameters affect Speedup the most

Note: The following input file parameters were measured with various numbers of nodes, tasks, and CPUs. Graphs and details for the generated test cases can be found in the appendix.

Number of Stations

As the number of stations increases, generally the time taken gradually increases. This is likely because as the number of stations grow, the adjacency matrix that needs to be iterated through to find the number of platforms becomes much bigger (grows at $O(n^2)$). However, as this is only done once by each process and does not involve communication, it does not significantly impact the time taken.

Max Line Length

As the max line length increases, the time taken generally stays the same. This is likely because even though increasing the max line length increases the number of platforms that needs to communicate with each other, since I have assigned the platforms such that those in the same line are highly likely to be grouped together, increasing the max line length only increases the number of “self-communication”, and the amount of communication with other ranks generally remains the same. Hence, the max line length does not affect the time taken by much.

Max Number of Trains

As the max number of trains increases, generally the time taken gradually increases. This could be because my code iterates through all the future spawned trains in rank 0 for sorting purposes, and all other processes would have to wait for it. Hence, increasing the number of trains spawned would increase the time taken. However, this does not involve any communication and does not significantly impact the time taken.

Another possible reason is because every platform also has to iterate through and find the colours of all spawned trains in that specific tick for every process, as well as spawning and queuing trains for their own platforms. Hence, increasing the number of trains spawned would increase the number of ticks for which all of these would have to be calculated, increasing the time taken. However, this once again does not involve any communication, and does not significantly impact the time taken. Hence, increasing the max number of trains does not greatly increase the time taken for either possible reason.

Number of Ticks

As the number of ticks increases, the time taken increases linearly. This is because every platform would have to communicate with all of its incoming and outgoing platforms for every tick, and would further have to wait for all platforms to be done with the current tick before moving on to the next tick for synchronisation purposes. Hence, increasing the number of ticks linearly increases the amount of communication needed, greatly increasing the time taken since communication is the bottleneck for the program.

Number of Tasks

Comparing all 4 input file parameters used, as the number of tasks increases, the time taken decreases. This is because increasing the tasks increases the number of processes the platforms can be distributed amongst, resulting in less overall work per process since they are evenly distributed. This thus results in a decrease in time taken.

Number of Nodes

Comparing all 4 input file parameters used, as the number of nodes increases, the time taken increases, even if a higher ntask is used. This is likely because even though the work is distributed amongst more processes, these processes are split amongst a separate node and is no longer within the same processor. Hence, the physical distance that data has to travel becomes significantly higher, causing a much higher overhead during communication and exceeding the benefits of having more ntasks. Hence, increasing the number of nodes increases the time taken for the relatively small amount of work done by the code.

CPU Type

Between the 2 CPUs tested: i7-13700 and xs-4114, the i7-13700 on average took about half as long to run when comparing all 4 parameters tested. This is likely because the i7-13700 has a higher max turbo frequency of 5.20GHz as compared to the 3.00GHz max turbo frequency of xs-4114. This allows many more clock cycles to be done per second, resulting in all calculations being done faster.

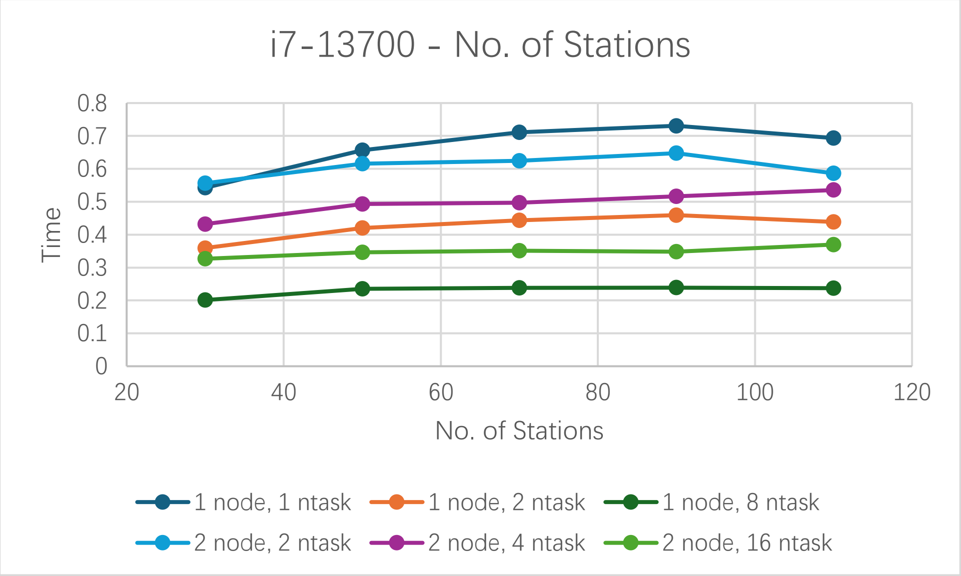
In conclusion, the number of ticks affects speedup the most when generating the input file, as the other parameters do not affect time taken by much. Comparing the hardware configurations used, the number of tasks affects speedup the most since it decreases the time by about half to a third as compared to changing from xs-4114 to i7-13700, which decrease the time by about half.

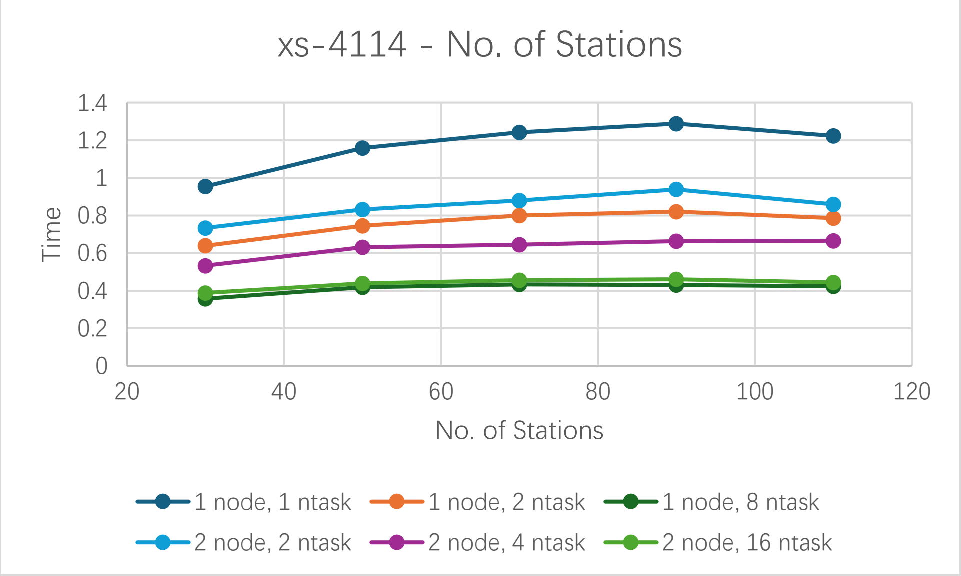
Appendix

All the code used to generate and run the tests can be found in the files “generated_tests.sh” and “run_generated.sh” respectively.

All the data for this section can be found in “Data.xlsx”.

Number of Stations

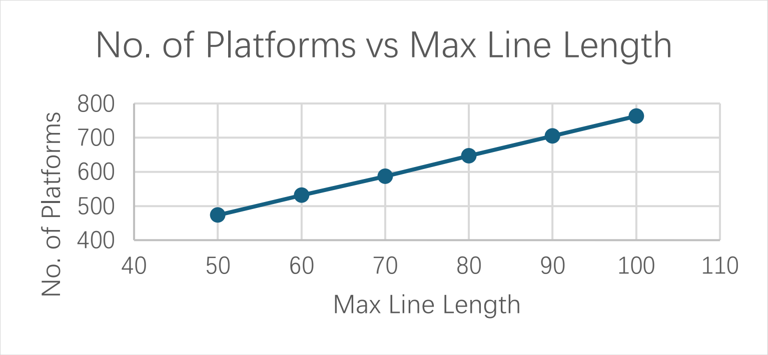




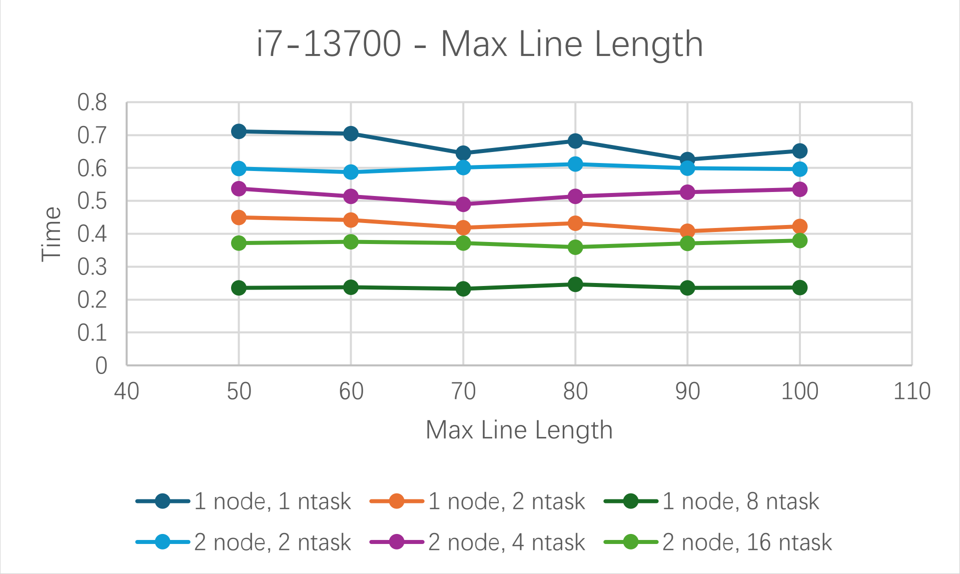
```
python3 gen_test.py <stations> 10 10 500 55 500
```

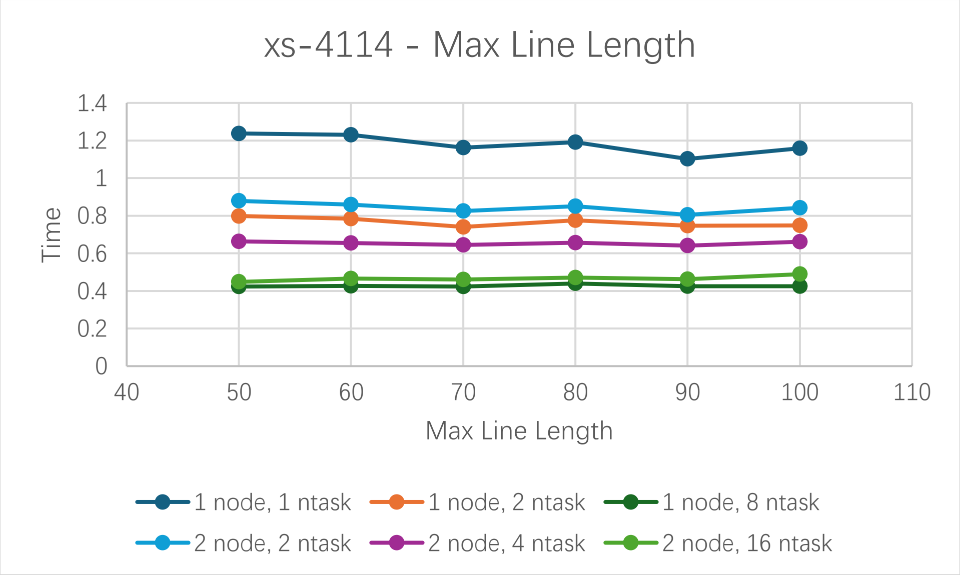
where `<stations>` is 30, 50, 70, 90, 110

Max Line Length



No. of Platforms increases linearly with the Max Line Length.

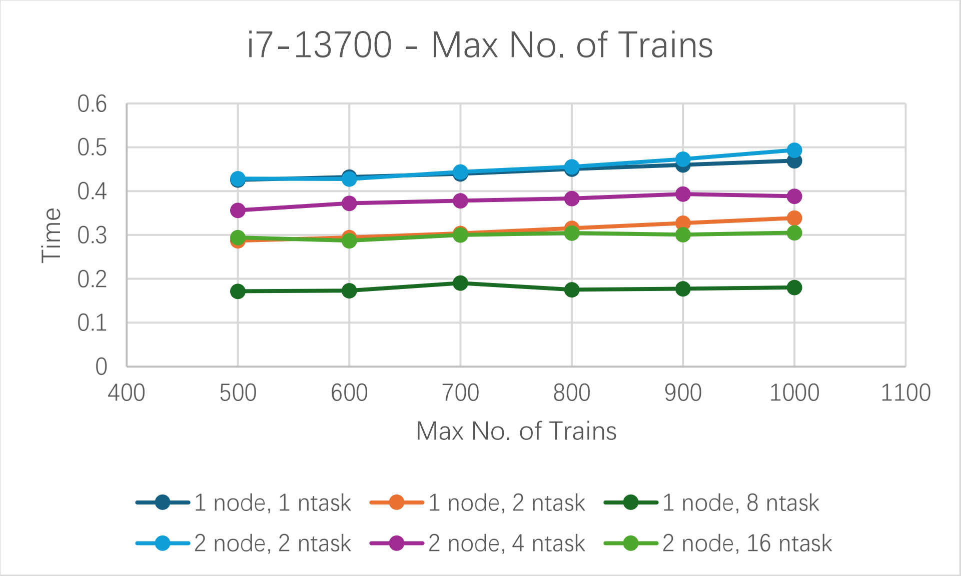


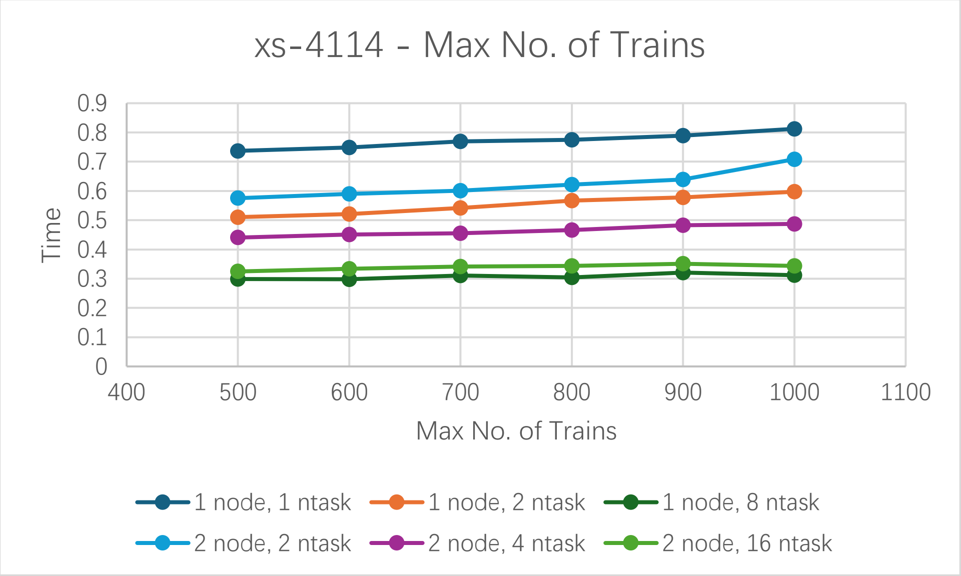


```
python3 gen_test.py 100 10 10 500 <max_line_len> 500
```

where `<max_line_len>` is 50, 60, 70, 80, 90, 100. Line length above the number of stations S=100 is not tested since there cannot be more than S=100 stations in a line.

Max Number of Trains

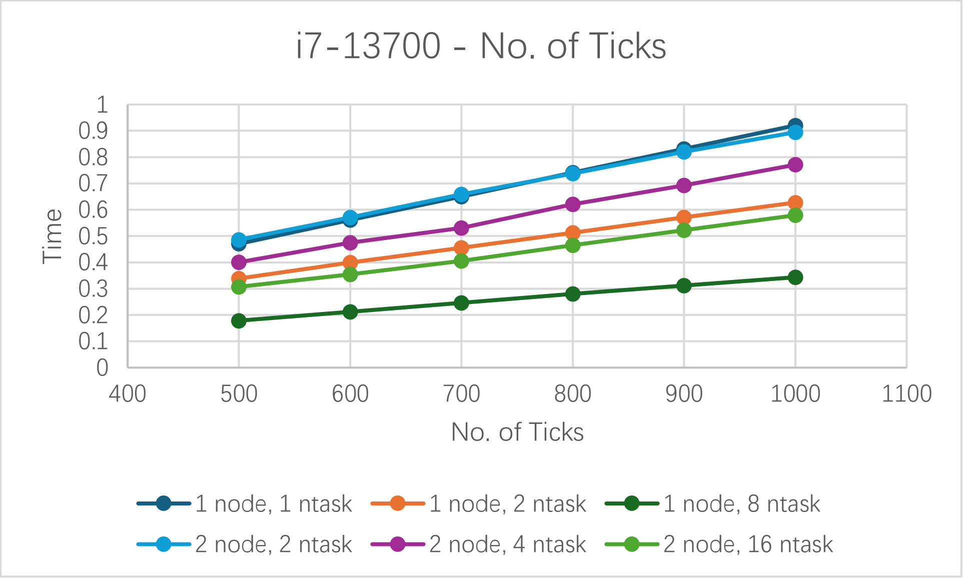


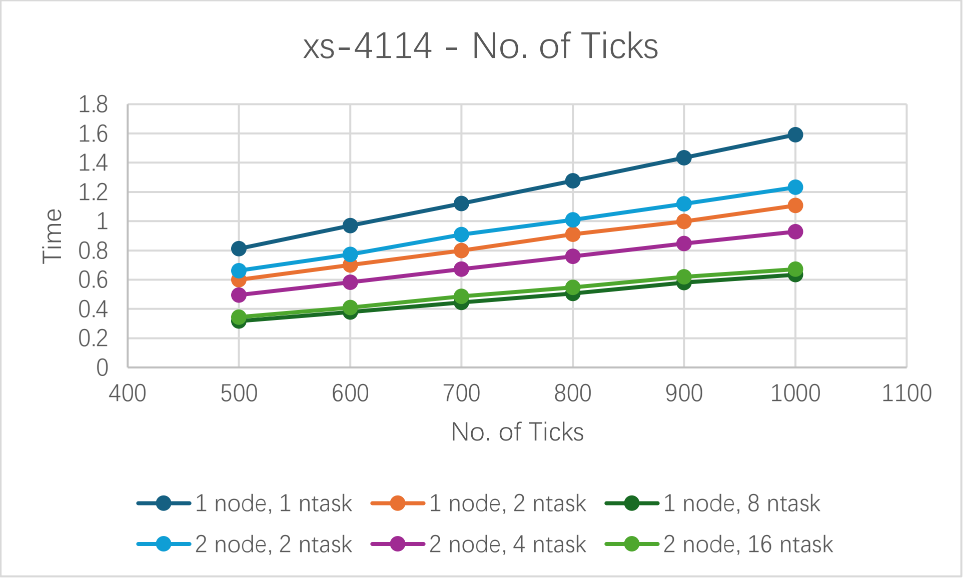


```
python3 gen_test.py 30 10 10 <max_num_trains> 15 500
```

where `<max_num_trains>` is 500, 600, 700, 800, 900, 1000

Number of Ticks





```
python3 gen_test.py 30 10 10 1000 15 <N>
```

where `<N>` is 500, 600, 700, 800, 900, 1000