Simulation of MRT Network with MPI

Implementation Overview

Parallelisation Strategy

To achieve parallelism, we distributed the computational workload associated with station simulations across multiple MPI processes. Each station's state and operations can be processed independently, which makes this a good choice.

Stations are assigned to MPI processes in a round-robin manner. This minimizes workload imbalance by ensuring a balanced distribution.

Each process would be responsible for simulating the subset of stations assigned to it, which includes managing train movements, queuing, and departures.

Station data and train states are localized to each process, avoiding shared memory conflicts. Inter-process interactions occur strictly via MPI communications.

A logical clock system ensures all processes progress in synchronized time steps, preventing inconsistencies due to out-of-sync updates.

Parallelisation Design

Train-related data is stored in a contiguous 1D array in the root process. This simplifies memory management and reduces fragmentation, enabling efficient data access and transfer during communication.

Each process performs local simulations for its subset of stations. When Inter-process communication is required, it would be communicated to the root node first, which would then scatter it to the respective stations in the next time step.

Trains that move between stations handled by different processes trigger inter-process communication. For this, MPI constructs like `MPI_Gather` and `MPI_Scatter` are used to transfer train state information between processes.

Handling deadlocks and race conditions

Local Copies of Shared Data:

- To avoid race conditions, local copies of shared resources (e.g., train buffers) are maintained within each process. These are only updated by the process itself.
- Any data required by another process is explicitly communicated, ensuring no simultaneous read/write conflicts.

Using MPI_Scatter and MPI_Gather:

- At each simulation step, processes send local train and station updates to the root process using `MPI_Gather`. The root process consolidates these updates and redistributes the necessary information back to the processes using `MPI_Scatter`. This ensures that data exchange is limited to root-process interactions, simplifying communication patterns and eliminating peer-to-peer dependencies.
- These collective communication operations are implicit barriers. This eliminates the need for explicit barriers and ensures all processes are synchronized at each timestep.

Avoiding Circular Dependencies

- Since all communication occurs through the root process, the system avoids circular dependencies between processes. This design inherently prevents deadlocks.

Key MPI constructs

MPI_Comm_rank and MPI_Comm_size

- MPI_Comm_rank: identifies the rank of a process, determining its role in the simulation (root or worker)
- `MPI_Comm_size`: Provides the total number of processes in the communicator, enabling calculation of variables for even distribution of workloads.

MPI_Gather and MPI_Scatter

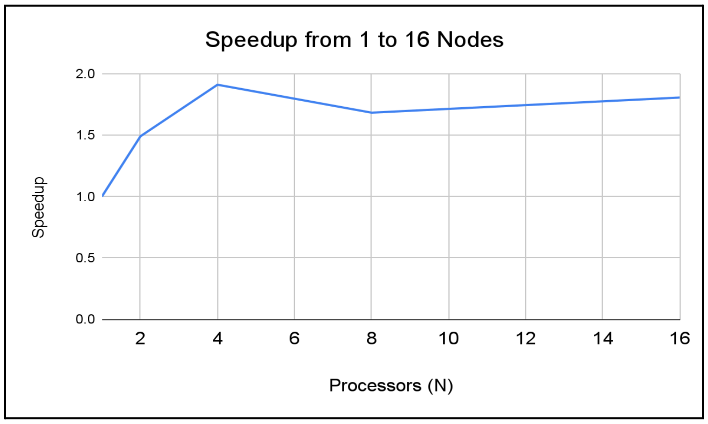
- MPI_Gather: Collects data from all processes to root for consolidation
- MPI_Scatter: Distributes processed or updated data from the root process to all other processes.
- They ensure efficient and synchronised communication, avoiding the complexity of direct peer-to-peer communication and avoiding deadlocks arising from circular dependencies.

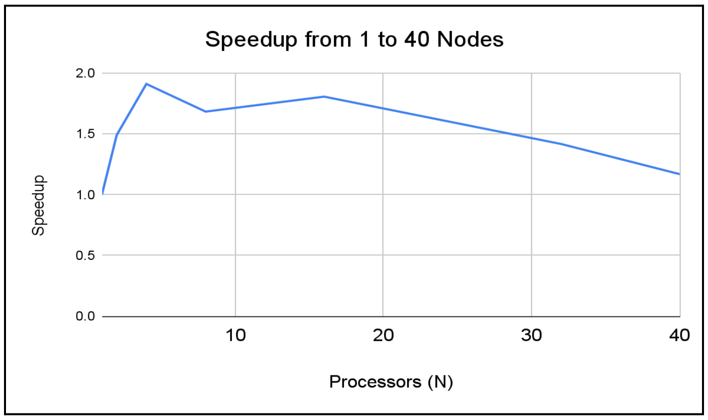
MPI_Datatype

- Creates a custom datatype to reduce communication overhead by transferring only relevant fields of a train's data instead of sending the entire object over.

Analysis of Input Factors on Runtime & Explanations

Number of Nodes: N = 1 to N = 4 is run on a single node. N > 4 uses a multi node configuration.

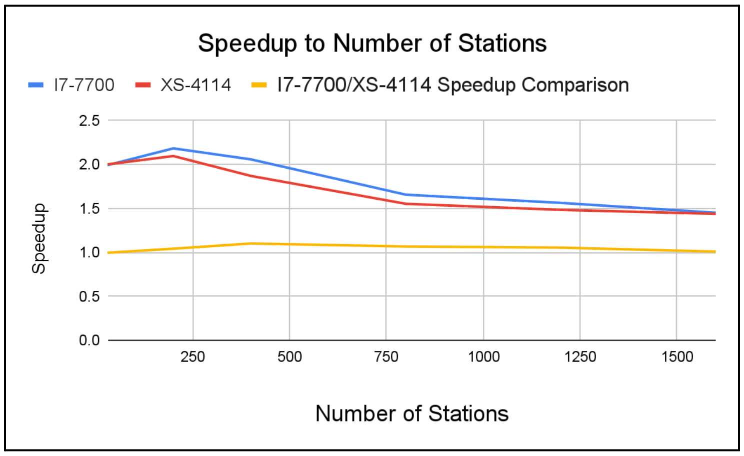




Speedup increases as the number of processors increases from 1 to 4. The highest speedup (~1.91) is observed at 4 processors, indicating this is where the parallelism is most effective. Beyond 4 processors, the speedup starts to diminish with a slight dip and then increases as we increase to 16 processors. Above 16 processes, the speedup gradually decreases.

This is due to the workload not being intensive enough to outweigh the cost of cross-node MPI communication.

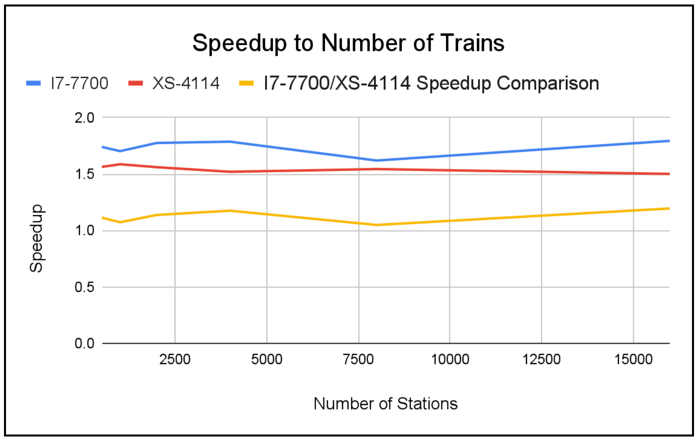
Number of Stations and Links (Half the amount of stations and Raw Data here)



Difference between CPU:
The performance difference can be largely attributed to their architectural characteristics and how they handle the train simulation workload. The i7-7700 with a higher clock speed at 3.6Ghz and lower inter-core latency displays better performance in the MPI-Based Simulation. This advantage is particularly evident in the speed up number where a 2.18 speedup was achieved with 200 stations, while the Xs-4114 only had 2.09. The faster cache access and better single core performance that the i7-7700 has proved crucial for handling the frequent train state updates and MPI communication effectively.

Scalability As the Number of Stations Increases: However, as the number of stations scales up from 30 to 1600, both processors show degrading performance but with different patterns. This degradation is primarily due to increasing communication overhead, as a larger number of stations results in more frequent MPI communications between processors - particularly when trains transfer between stations managed by different processors. Despite this scaling challenge, both processors maintain notable speedup even at 1600 stations, with the i7-7700 achieving 1.44x speedup and the XS-4114 achieving 1.43x speedup, demonstrating that parallelization benefits persist even under heavy communication loads. This sustained performance benefit suggests that the simulation's workload distribution still outweighs the communication overhead, even at large station counts.

Number of Trains (Raw Data here)

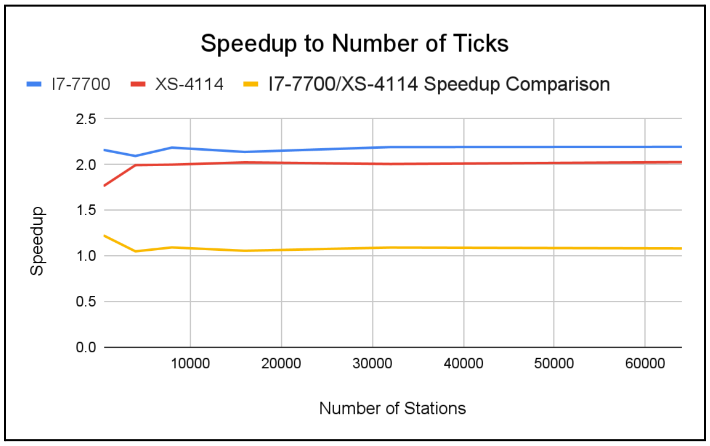


Difference Between CPU:
The performance comparison between I7-7700 and XS-4114 nodes when scaling train numbers from 500 to 1600 shows interesting patterns. While the I7-7700 maintains more consistent performance with execution time ranging narrowly between 0.54-0.57 across all train counts. Its speed up ratio shows a slight improvement up to 4000 at its peak (1.78x) before gradually decreasing. The XS-4114 on the other hand shows overall slower execution times around 0.81-0.83 seconds but still maintaining stable performance across different train loads.

Scalability as the Number of Trains increases: Both processors show better stability with increasing train counts compared to increasing station counts. This suggests that the simulation handles train scaling more efficiently than station scaling. This is likely because adding trains primarily increases computational within processors rather than increasing inter-processor communication. This is further supported by how the i7-7700 having a higher clock speed (3.6GHz>2.2Ghz) shows significantly better performance. But the performance gap between the two processors remains relatively constant, unlike with station scaling where the gap widened more significantly. Therefore, showing how train count scaling is less dependent on the architectural difference between the processes, or more particularly their inter-core communication characteristics.

.

Number of Ticks (Raw Data here)



Scalability as the Number of Ticks increases:
The scaling behavior with increasing tick counts reveals a more linear relationship between execution time and workload for both processors. As seen from the graph the i7-7700 shows consistently better speedup ratios ranging from 2.16x at 500 ticks to 2.19x at 64000 ticks, demonstrating excellent scalability with increased simulation duration. Similarly the XS-4114 exhibits similar scaling characteristics but with lower absolute speedup values, ranging from 1.76x to 2.02x. Notably, both processors maintain their speedup ratios even as execution time increases substantially as shown on the yellow line, suggesting that temporal scaling of the simulation is well balanced between computation and communication. **Temporal Scaling Success, Enhance Performance at higher tick count:** In the graph, both processors actually show slightly improved speedup ratios at higher tick counts, with their best performance at the highest tick tested. This contrasts with the scaling pattern seen for trains and stations. Hence, this behavior indicates that the simulation's parallelization strategy is particularly effective for temporal scaling, likely because increasing ticks only adds computation within existing communication patterns rather than introducing additional computation overhead. Nevertheless the i7-7700 superior clock speed advantage remains evident throughout, but both processors still demonstrate that parallel implementation becomes more efficient with longer simulation durations.

Appendix

Commands:

Generation of test cases:

Min_test:

```
python3 gen_test.py 30 10 10 500 15 500 > testcases/performance/min_test.in
```

Varying Stations & Num_links:

```
python3 gen_test.py 30 10 10 500 15 500 > testcases/stations_30.in
python3 gen_test.py 800 10 10 500 400 500 > testcases/stations_800.in
python3 gen_test.py 1600 10 10 500 800 500 > testcases/stations_1600.in

python3 gen_test.py 200 10 10 500 100 500 > testcases/stations_200.in
python3 gen_test.py 400 10 10 500 200 500 > testcases/stations_400.in
python3 gen_test.py 1200 10 10 500 600 500 > testcases/stations_1200.in

python3 gen_test.py 3200 10 10 500 1600 500 > testcases/stations_3200.in
python3 gen_test.py 6400 10 10 500 3200 500 > testcases/stations_6400.in
python3 gen_test.py 12800 10 10 500 6400 500 > testcases/stations_12800.in
```

Varying max_num_trains:

```
python3 gen_test.py 800 10 10 500 400 500 > testcases/trains_500.in
python3 gen_test.py 800 10 10 1000 400 500 > testcases/trains_1000.in
python3 gen_test.py 800 10 10 2000 400 500 > testcases/trains_2000.in
python3 gen_test.py 800 10 10 4000 400 500 > testcases/trains_4000.in
python3 gen_test.py 800 10 10 8000 400 500 > testcases/trains_8000.in
python3 gen_test.py 800 10 10 16000 400 500 > testcases/trains_16000.in
```

Varying Ticks:

```
python3 gen_test.py 30 10 10 500 15 500 > testcases/ticks_500.in
python3 gen_test.py 30 10 10 500 15 4000 > testcases/ticks_4000.in
python3 gen_test.py 30 10 10 500 15 8000 > testcases/ticks_8000.in
python3 gen_test.py 30 10 10 500 15 16000 > testcases/ticks_16000.in
python3 gen_test.py 30 10 10 500 15 32000 > testcases/ticks_32000.in
python3 gen_test.py 30 10 10 500 15 64000 > testcases/ticks_64000.in
```

Execution for varying Station, Num_Train, Ticks:

```
salloc -p i7-7700 --nodes 1 --ntasks 4 mpirun ./trains testcases<testcase>.in > outputs/<testcase>.txt
salloc -p xs-4114 --nodes 1 --ntasks 4 mpirun ./trains testcases<testcase>.in > outputs/<testcase>.txt
```

Lab Machines used:

1. I7-7700
2. xs-4114

Results:

[Excel Sheet with Raw Data and graphs]