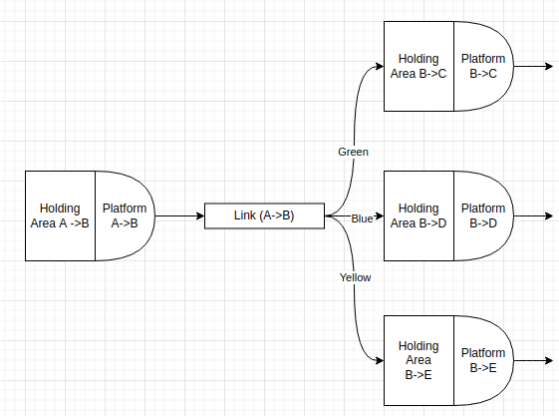
CS3210 Assignment 3 Report

Problem Modelling

Similar to assignment 1, we have the same problem of modeling a troons simulation problem which can be described in the following picture.



However, instead of separating HoldingArea, Platform, and Link which is done in assignment 1, these 3 classes is combined into a vertice. Moreover, instead of representing the edge between the vertices as a pointer, the edge is an integer representing the index of the next vertice in the array as shown in the following data structure.

```
vector<staticLinkState> graphState;
vector<dynamicLinkState *> graphStateDynamic; // to be initialized in each node
```

StaticLinkState are information that are fixed to the link which is static across all nodes. While DynamicLinkState contains information that can change over time such as the waiting area, platform counter, etc which is different between each node. Both of them make a vertice in the graph, which we will call ***link unit*** from now on. Each of these vertices is responsible to manage troons, a data class that contains the troon states such as the location, link, source, and destination.

This design is chosen because MPI runs on a distributed memory environment, meaning pointers cannot be passed from one node/process to another.

Parallelisation Strategy

Our strategy is to use block data distribution between the nodes and to parallelize the link unit instead. Each node will be responsible to handle the **~|E| / N** link units where |E| is the total number of link units and N is the total number of nodes/processes used by OpenMPI.

Therefore, each link unit is currently responsible to process the link, processing the platform, waiting area, etc as shown in the following code snippet. Moreover, after processing the link and should there be some troons that should be managed by other link units in other nodes instead, the exchangeTroons method will allow each node to pass the cross-nodes troons to each other. Moreover, to print the states of the troons, all the nodes must send all their troons to 1 process/node which will sort the troons lexicographically and print it out to stdout.

Deadlocks and Race Conditions Resolution

To avoid deadlocks in exchangeTroons methods, we use non-blocking asynchronous send and receive (i.e. MPI_Irecv and MPI_Isend). As these calls are both non-blocking and asynchronous, it will prevent the program to be in a deadlock condition (unless during the MPI_Isend, all the nodes perceive the MPI and kernel buffers to be full which is very unlikely as the troons do not consume very huge memory).

Moreover, we ensure that all the processes have finished the troons exchanges by calling MPI_Waitall. The exact code is shown here

```
  for (int i = 0; i < nprocs; i++) {
       ...
       MPI_Isend(buffer_send, toSendSize, mpi_troon_type, i, 0, MPI_COMM_WORLD,
&req[i * 2]);
       ...
       MPI_Irecv(troon_buffer, toReceiveSize, mpi_troon_type, i, 0, MPI_COMM_WORLD,
&req[i * 2 + 1]);
   }

   MPI_Waitall(nprocs * 2, req, MPI_STATUS_IGNORE);
```

After the troon exchanges, the troons is inserted into the waiting area sequentially by each node and hence there should not be any race condition here.

Lastly, during the printTroons method, all the nodes except the master/original node will send their troons to the master who will receive it in a blocking, synchronous call (i.e. MPI_Send and MPI_Recv). As this process is done in a round robin fashion and there is no circular dependency between the nodes, hence there should not be any deadlock here. Moreover, each troon is processed sequentially in the master node, so there can't be any race condition either.

Key MPI Constructs

These are the key MPI Constructs that are being used:

1. MPI_Alltoall
   This method is called within the exchangeTroons method to update all of the nodes regarding the number of troons to be received by other nodes.
2. MPI_Isend + MPI_Irecv

These methods are used inside the exchangeTroons method to asynchronously exchange the troons information between the nodes.

3. MPI_Gather
   This is called inside the printTroons by all of the nodes so the master process can gather the number of troons to be received from each of the nodes.
4. MPI_Send + MPI_Recv
   These methods are used inside the printTroons method to pass all the troons from all the nodes except the master node to the master node so that the troons can be sorted and printed.

Input Factors and Speedup

1. The **number of stations and links** affect the speedup the most for our implementation.
2. As mentioned in our parallelisation strategy and shown in the code snippet, all of the process that are being parallelised is related to the operations in the link. As shown in Table 1, as we **increase the number of stations and links correspondingly a greater speedup is achieved**. We want to check with more stations and links, however when we increase the number of stations into 25600 slurm

In table 1, 2, and 3 experiments, we are using the i7-7700 node. In table 4, we are using 1 i7-7700 node and another xs-4114 node. In table 5, we are using 1 xs-4114 node to check scalability (varying the number of cores). The number of links being discussed here is the total number of bidirectional links between stations.

When we fix the number of trains and ticks as shown in Appendix table 1 and table 4 to 100 and 100 repectively. Our code is actually slower than the sequential implementations with these number configurations, however, we see an increasing speedup as we increase the nu,ber of stations and links.

As we increase the number of trains as shown in Appendix table 2, we observe little speedup but not that significant. In this scenario, the numbers of stations, links, and ticks are fixed to 3200, 4800, and 16000.

As we increase the tick as shown in Appendix table 3, we observe little speedup but not that significant. This actually makes sense as we never parallelise any operations across different ticks. In this scenario, the number of stations, links, and ticks are fixed to 6400, 9600, and 4000 respectively.

As shown in Appendix table 5, where we fix the number of stations, links, trains, and ticks to 30000, 45000, 1000, and 1000 respectively and we vary the number of cores, we can see an increase on performance. We can achieve an even higher performance by increasing the number of stations and links, however, slurm does not allow us to run a very big job.

Performance Optimisation Attempt

One performance optimization that we did is that instead of using synchronous and blocking call in printTroons method, we are changing them to asynchronous and non-blocking call (i.e. MPI_Irecv and MPI_Isend).

Our hypothesis is that using the non-blocking and async call, the original node can overlap the receiving and memory allocation computation which makes it faster overall as shown in the following code snippet (The exact code is in the branch called *optimizing*)

```
for (int i = 0; i < nprocs; i++) {
   ...
   auto troon_buffer = new Troon[troons_counters[i]];
   troons_recv_buffer[i] = troon_buffer;

   MPI_Irecv(troon_buffer, troons_counters[i], mpi_troon_type, i, 0,
MPI_COMM_WORLD, &req[i]);
}

MPI_Waitall(nprocs, req, MPI_STATUS_IGNORE);
```

To support this hypothesis, we are running an experiment in i7-7700 node with 4 processes for 6400 stations, 9600 bidirectional links (i.e. 19200 total unidirecitonal links), and 28800 troons for T ticks and we try to print for all the ticks too. T is to be varied because T should be the main factor affecting the speedup here. Even though the requirement is only for the last 5 ticks, we hope to exaggerate the printing to magnify the effect.

The result is shown in Table 6. The result is here is the average of 3 run times to reduce the variance of the result. As the result shows, there is no clear result that the async and non-blocking calls result in a more performant result.

One possible explanation about this is that the cost of memory allocation is becoming less and less significant at the ticks increase possibly due to compiler optimization to avoid the repeated memory allocation. This causes the performance between the synchronous blocking and asynchronous blocking to be fairly similar. Moreover, it is also possible that the cost to synchronize and wait for all the non-blocking, async calls to be done might be higher when it is done a lot of times which increase the time overall.

Appendix

Commands:

- table 1, 2, and 3:

```
srun -n8 -N1 -p i7-7700 /nfs/home/${USER}/troons
/nfs/home/${USER}/generatedInput.in
```

- table 4:

```
srun -n8 -N2 --constraint="i7-7700*1 xs-4114*1" /nfs/home/${USER}/troons
/nfs/home/${USER}/generatedInput.in
```

- table 5

```
srun -n{NUMBER OF CORES} -N1 -p xs-4114 /nfs/home/${USER}/troons
/nfs/home/${USER}/generatedInput.in
```

- table 6

```
srun -n4 -N1 -p i7-7700/nfs/home/${USER}/troons
/nfs/home/${USER}/generatedInput.in
```

Lab Machines:

1. i7-7700
2. xs-4114

Results:

Table 1.

Stations/Links	MPI (s)	Sequential (s)	Speedup
800/1200	0.424616	0.027401918	0.06453340901
1600/2400	0.578677	0.096268156	0.16635905
3200/4800	1.24675	0.369404562	0.296294014
6400/9600	3.6613	1.458808025	0.3984399052
12800/19200	13.6062	5.842230701	0.4293800401

Table 2.

Trains	MPI (s)	Sequential (s)	Speedup
1000	2.23876	1.753242634	0.7831311235
2000	2.67943	2.496862199	0.9318631944
4000	3.25538	3.607854329	1.108274404

8000	4.80012	5.622211667	1.171264816
16000	18.87617	16.23706318	0.8601884373

Table 3.

Ticks	MPI (s)	Sequential (s)	Speedup
4000	5.24616	2.892500713	0.5513557941
8000	5.60412	4.856510164	0.8665963905
16000	6.48874	6.291761667	0.9696430535
32000	8.36953	9.169678918	1.095602611
64000	12.1426	16.73357798	1.378088546

Table 4.

Stations/Links	MPI (s)	Sequential (s)	Speedup
800/1200	0.547739	0.061848684	0.1129163415
1600/2400	0.701327	0.096446243	0.1375196492
3200/4800	1.10424	0.368905054	0.3340805024
6400/9600	3.18833	1.4870197	0.4663945388
12800/19200	11.2865	5.856595268	0.5189026951
25600/38400	44.451	23.15413487	0.5208912031

Table 5.

Cores	MPI (s)	Sequential (s)	Speedup
1	56.5275	16.17445435	0.2861342594
2	55.8883	16.17445435	0.2894068051
4	53.0789	16.17445435	0.3047247465

Table 6.

Ticks	MPI Optimized (s)	MPI Not Optimized (s)	Speedup
500	3.424	5.932	1.732476636
1000	7.849	7.1475	0.910625557
5000	227.78	236.87	1.040272288