Assignment 3 Report

Nguyen Son Linh (A0200705X)
Njal Telstoe (A0260770H)

Program Description

Parallelisation strategy

To parallelize the program, we distribute the links across all the processes and have each of them be responsible for updating the state of their links. Since a troon can transit from a link on one process, to a link on another process, we have to communicate this with MPI messages. When a troon finishes its transit and is to be moved to a link on another process, we send the troon information with an MPI message. If the transit is not finished or there is no transit on the connecting link, we still send a message because the recipient will be expecting it. But now it is a special message with signals to the receiving link that there is no troon coming from the sender.

Each link keeps track of their own troons, when it's time to output the troon states, we gather all the troons at the root process and it prints the states.

Avoiding deadlocks and race conditions

We decided to use non blocking sends when sending a transiting troon from one link to another because this avoids deadlock issues. To ensure that there are no race conditions, we wait for all messages to be sent before proceeding to the next tick.

Key MPI constructs

We used MPI_Isend to send troon messages. We used MPI_Irecv to receive messages. To ensure that there would be no race conditions, we used MPI_Waitall to wait for all the sent and received messages to be completed. When gathering all the troons at the root process, we use MPI_Gather first to get the number of troons on each process. This is because there can be different amounts of troons on each process, so the root process needs to know how many troons to expect from each process. Then we use MPI_Gatherv to gather all the troons at the root process.

Impact of input size

Since we do updates on each link for each tick, we would expect the performance to be dependent on the number of stations and the number of ticks. It will also depend on how the stations are connected, because there can be a varying amount of links created depending on if lines share the same links.

The performance should not be very dependent on the number of troons, since we never iterate over them. There will be some performance penalty when gathering all the troons at the root process for printing. So we can expect a slight performance drop for an increased number of troons, and if the number of lines printed is increased.

We took measurements on two network configurations: small and large.

	stations	max_link_weight	max_popularity	max_line_length
small	100	50	50	70
large	10k	500	500	7k

For the small configuration, we varied the number of ticks and number of troons per line, creating 9 sub-configurations. Number of ticks can be 1k, 10k, and 100k, while the number of troons can be 10, 100 and 1000. Regarding the large configuration, due to time constraints, we fixed the number of ticks to 100k, and varied the number of troons per line at 1000, 10000 and 100000.

Partition is fixed to be i7-7700. `perf stat -r 5` is used to measure the time taken for consistency. Our MPI implementation is run on 1 node with 4 tasks.

Small configuration

Ticks/Troons	10	100	1000
1k	0.004	0.007	0.003
10k	0.017	0.047	0.059
100k	0.076	0.258	0.298

Table 1: Speedup on small network with different ticks and troons per line.

Large configuration

Ticks/Troons	1000	10000	100000
100k	1.171	1.122	0.798

Table 1: Speedup on large network with 100k ticks and different troons per line.

It is surprising that when number of troons increase for the large configuration, speedup decreases, which could be due to MPI send and receive overhead.

Performance Optimization

The links are distributed equally across the processes. Rank 0 get's the first chunk of links, rank 1 get's the second chunk and so on. In the original implementation, the links were created in the order they appeared in the adjacency matrix. This resulted in chunks not necessarily containing connecting links, which in turn resulted in many IMP messages having to be sent across the processes because of troons transiting to other chunks.

In our final solution, we changed the order of how links are created in such a way that chunks contained mostly connecting links. Instead of creating the links in the order they appeared in the link matrix, we created the links based on the order they appeared in for each line. Since multiple lines can share the same links, there would be some disjoint links in the chunks, but the improvement was still very significant.

Comparison of execution time for the old solution and the new solution:
Using a large configuration with 100000 troons

Version / Ticks	10	1000
Old	5.8 seconds	62.6 seconds
New	5.2 seconds	5.4 seconds

We observe that when we increase the number of ticks, resulting in more communication across chunks, the performance gains in the new solution become very visible.

Bonus

No change in implementation.
Configuration:

- 1x Xeon Silver 4114 machine and 1x Intel Core i7-7700 machine
- 4 tasks on each node.
- Large network with 100k ticks and 10k troons per line

Parallel execution time: 17.5343s
Sequential execution time: 13.185s

Appendix

Reproduction of results

First, run `make submission`.

In our repository, how to create the small and large networks is included in `gen_performance_tests.md`, and it is advised to adjust the number of ticks and number of troons per line by hand. Afterwards, please run `prepare.sh` to create the folder that will be copied to nfs, which include all the testcases, job scripts along with executables `troons` and `troons_seq`. Then run the following command (replace accordingly for small configuration):
`./run_troons.sh testcases/large.in && ./run_troons_seq.sh testcases/large.in`
This will create two slurm batch jobs for `troons` and `troons_seq` with the same testcase. When these jobs are finished, symlinks are created to the result files and speedup can be calculated.

Please look into the details of `slurm/troons.sh` and `slurm/troons_seq.sh` for details of how perf is used to measure execution time.

For bonus part, there is a separate runner and job used for it, which can be found easily in the repo.

Chart links (data included)

Speedup calculation for small and large configurations