Assignment 1 Report

Julius Putra Tanu Setiaji (A0149787E), Chen Shaowei (A0110560Y)

24 October 2018

1 Program Design

This train simulation is implemented in OpenMPI. Primary design considerations are:

- Each edge is simulated by one process as required in the specifications.
- There is a master process responsible for the following:
  - Distribution of information (map, lines) to slave processes.
  - Keeping track of train states (travelling/stationary).
  - Keeping track of station door open/close times.
  - Synchronising time across all slaves.
- Slave processes hold queues of trains and send them to each other.

Assumptions

- Only one train can open its doors at each at any one time, regardless of direction.
- Train stations have infinite capacity for waiting trains.
- Time units are discrete and can have no subdivisions
  - **Implication**: It is sufficient to store all time units as integers instead of floating point numbers
- Trains must open their doors for at least 1 unit of time.
  - **Implication**: We round every randomly generated door open time up to the nearest integer

2 Points to Note / Implementation Details

- Current simulation time needs to be shared across all processes. In addition, time can only be advanced after all threads have completed the actions to be done in the current tick.
  - **Implication**: Explicit synchronisation messages need to be sent between master and slaves before and after the advancement of time.
- Each slave process maintains two queues of trains. The first queue contains the trains that are waiting to enter the edge. The second queue contains the trains that are waiting at the next station but have yet to close their doors.

- The logic for each slave process at each tick is as follows:
  - If a current train is occupying the edge and it can leave at the current time, the process will query master to find out when this train will finish closing its doors. The edge will then be marked as available.
  - If there are trains waiting to access the edge, the process will dequeue the first one and set it as the current train.
  -
- Each train is a finite state machine with 4 states: `OPEN_DOOR`, `CLOSE_DOOR`, `DEPART` and `ARRIVE`.
- Each train keeps track of its next action time (time for it to change to the next state), and actions to be completed within the current tick are performed in a `while` loop.
- There is a `#pragma omp barrier` to ensure that all threads exit the `while` loop before time is advanced.
- The advancement of time is done in a `#pragma omp single` block to ensure that it is only performed by one thread. In addition, `#pragma omp single` has an implicit barrier at the end of the block so this prevents other threads from entering the next iteration of the `while` loop until the advancement of time is complete.
- Certain resources i.e. train tracks, door-opening rights are limited, and only one train may access them at any time.
  1. One way to implement this is to have threads block when waiting to access such resources. However, this would interfere with the way we have chosen to implement time within this simulation. Blocked threads would prevent code after a `#pragma omp barrier` statement from being executed. We therefore decided **not** to go with this implementation.
  2. An alternative way to implement is to have a queue of trains requesting access to such resources. At each tick, each train would check whether it is at the head of the queue, and if so, it will be able to access the resource. We realised that it will also be necessary to store the time at which a train would be able to gain access to the resource, since our simulation time advances in integer increments, but door open time may have a fractional value. Upon further consideration, we realised that this design could be simplified.
  3. The key insight we arrived at is that any train waiting for access to a resource only needs to know the next time said resource will be available. Since this system does not permit a train to give up waiting for a resource, this can be implemented simply with a thread-safe timekeeper object. When a train requests access to a resource, it tells the timekeeper how much time it will occupy the resource for. The timekeeper will then inform the train of the time when the train can access the resource, and update its internal next available time. This is the implementation we decided to go with. In other words, we are implementing an implicit queue for a First-Come-First-Serve (FCFS) scheduling policy.
- The next consideration is ensuring that system statistics are reported correctly. Since we assume that trains cannot open its doors for 0s, only one train can open its doors at each station per tick. There is therefore no potential race condition in the update of statistics. Nevertheless, we protected their critical sections with a `#pragma omp critical` block for additional safety.
- Since print statements are not atomic, we wrapped them in a `#pragma omp critical` block to ensure that only one print operation executes at any one time.
- Thread safe timekeeper objects have their critical sections protected by a `#pragma omp critical` block.

3 Execution Time

3.1 Testcase Used

We use a map generated by a Ruby script which can be found in appendix. The map generated will have at least 4 vertices with degree = 1, and all the paths forming the train lines are at least of length 4. However, these values are configurable.

We ran the same map for 100, 1000, and 10000 time-ticks with all possible different combinations of numbers of trains in each line as long as the total number of trains is between 1 and 64 inclusive. This results in around 47,900 testcases for each time-tick size. As such, we will only include the scatter diagram of the results in the report. However, should the need arise, csv files containing the raw data is attached together with the report.

Below you can find a sample input and visualisation of the adjacency matrix and the train lines. The parameters that we used are number of stations = 10, maximum distance between stations = 10.

Do also note that to facilitate more accurate execution time analysis, we disabled per-tick status output for the following tests.

4 Discussion

It can be observed that the wall clock time taken for the simulation to complete increases as number of threads increases. This is because as number of threads increases, there is more contention of resources – in our case contention for each train (thread) in using links (edges) between stations (vertices) and contention for each train in opening door at each station. We are managing this using an implicit queue through implementing a timekeeper to track the next allowed event to occur, protected by marking that section as critical. As such, for each link (edge) or station (vertex), only one train (thread) is able to register itself to the timekeeper at any given time – others will have to wait.

However, we have an interesting observation as well. Notice that the execution time behaves very differently when the number of threads is beyond the number of logical cores (20 cores). For small input size (100 ticks), when the number of threads is beyond the number of logical cores, the execution time actually falls. However, as input size gets larger (1,000 and 10,000 ticks), execution time increases. This can be explained that when the number of threads are below the number of logical cores, all the threads are running concurrently, resulting in more lock contention (not to be confused with resource contention). However, when number of threads is above the number of logical cores, the threads take turns to wake up and do work, resulting in less lock contention. For smaller input size, the lock contention time actually outweighs the execution time, resulting in the fall in execution time. However, for larger input size, there is a large overhead in context-switching which outweighs the effect of lock contention time. This is supported by the data we collected on number of context-switches for 100 ticks and 1,000 ticks.

We also observe another trend – that the variance in execution time falls when the number of threads exceed the number of logical cores. We currently have no explanation on this, but we suspect that the compiler does an optimisation when the number of threads exceed the number of logical cores.

5 Bonus

Starvation will never occur in the simulation program that we wrote. This is because to decide which train to open door or to be allowed to use a link next, we are using an implicit queue to implement First-Come-First-Serve (FCFS) scheduler. As such, every train is assured access to the link or permission to open door after a long enough time. The assumptions are that no train open its doors or travel using the links indefinitely (which we believe are fair to make).

6 Appendix A: Ruby script used to generate test cases

Below is the code listing of the ruby script used to generate test cases. Essentially, it does:

1. Create a random adjacency matrix with diagonal = 0
2. Find the MST of the random graph created
3. Ensure that there are enough vertices with degree = 1, else go back to step 1
4. Enumerate the 2-combinations of the vertices with degree = 1, and pick 3 randomly.
5. For each of the three 2-combinations, assign them to be the termini of each line.
6. Using breadth-first-search, find the path between the two vertices for each pair of termini.
7. Ensure that the path is long enough, else go back to step 4.

```ruby
# SOME CODE
```