Assignment 1
Train Network Simulation using Parallel Programming

CS3210 – 2018/19 Semester 1

Learning Outcomes

This assignment is designed to enhance your understanding of parallel programming with shared-memory (OpenMP) and message passing (MPI). You will apply synchronization constructs you learned in class to solve a real world problem.

Problem Scenario

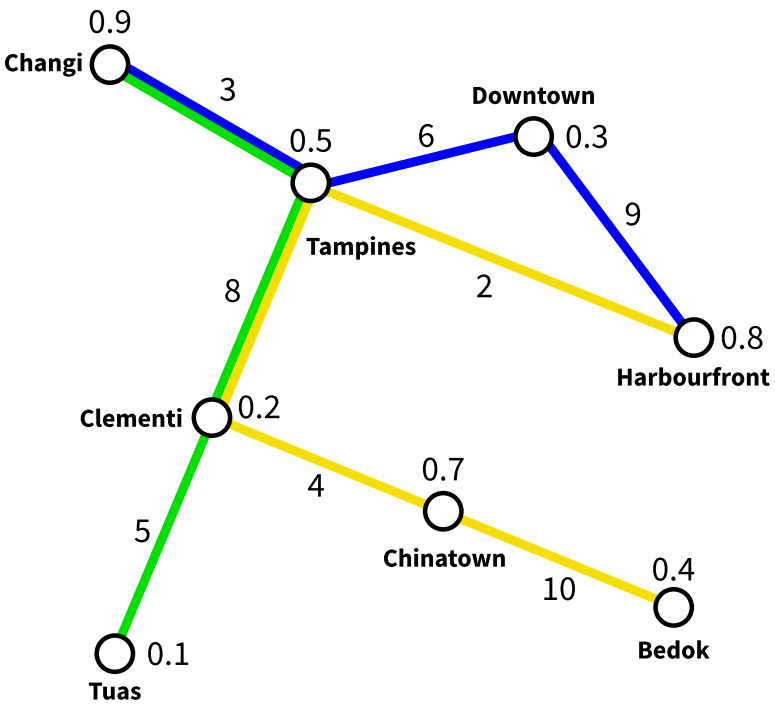

Figure 1: Example Train Network

The above shown graph in Figure 1 is a hypothetical train network in Singapore (MRT network). The nodes in the graph are stations and edges are links connecting stations. The weight of each edge represents the length of the link they are associated with and the weight of each node represents the popularity of the train station. Distances are measured in kilometers (km). Assume each edge consists of two parallel one-directional links dedicated for traveling in opposite directions.

There are three train services operating in this network: Green Line, Yellow Line and Blue Line throughout the day in both directions. The stations serviced by each train service are as follows:

- **Green:** Tuas → Clementi → Tampines → Changi (and reverse direction)

- **Yellow:** Bedok → Chinatown → Clementi → Tampines → Harbourfront (and reverse direction)
- **Blue:** Changi → Tampines → Downtown → Harbourfront (and reverse direction)

As illustrated in the graph, some links (edges) are shared by multiple train services. For example the link between Clementi and Tampines stations is shared by both Green Line and Yellow Line. However, there can only be a **maximum of one train** on a given link in one direction between two stations at any given time. Trains may wait at a station until the next link is available (assume that there is enough stationing space in the train stations). All trains travel at a **uniform velocity** of 1 km/time-unit (tick), and there is no acceleration or deceleration time (stopping and starting is instantaneous). Every train must stop at every stop on its route for at least the loading time (time-units) defined by the following formula:

$$loading\_time\_at\_station\_i = popularity\_of\_i \times r$$

where $r \in [1, 10]$. r should be obtained using `rand()` or `srand()` function calls in C.
Even if the train is stationing for a longer time at the station, passengers are able to board the train only during the loading_time_at_station. Doors will be closed for the rest of the stationing time. Assume that once a train reaches the end of a line, it starts traveling in the opposite direction.

Your task is to write parallel programs in (i) OpenMP and (ii) MPI to simulate this train network without encountering deadlocks and starvation. At the end of the execution, output the average waiting time for each line, given a number of trains that run on each line as input. The waiting time for a station on a line is computed as the time between doors of two consecutive trains (on that line) close and open. For example, on the yellow line at Clementi station, a train loads passengers and closes doors. The next next train arrives after 3 time-units and opens doors immediately. In this case, the waiting time was 3 time-units. The average waiting time for each line is computed by averaging the waiting time for each station on a line throughout the duration of the simulation.

(i) OpenMP Implementation:
In the OpenMP implementation, you must use threads to represent trains. (One thread per train)

(ii) MPI Implementation:
In the MPI implementation, you must use MPI processes to represent links between stations. (One MPI process per one link in one direction) Furthermore you should do a performance comparison between the two implementations based on their execution times for same input.

- **Starvation** occurs when a threads/process in your program is always denied from gaining access to a resource and, as a result, cannot make progress. In starvation, the resource becomes available from time to time, but the thread/process will not gain access to it (other thread/processes will gain access and make progress).
  In our problem scenario, starvation occurs when a particular train is waiting infinitely because it cannot gain access to the next link as other trains gain access to the link.
- **Deadlock** occurs when a thread/process is blocked in a waiting state because a requested resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process. The resource will never become available (as the other thread/process holding the resource is blocked).

The train network depicted above is just one possible scenario. Your program should be able to handle different problem scenarios (different graphs with different weights) and different configuration for the three train lines, based on the input as described below. The number of lines will always remain the same across all scenarios (i.e. 3 lines – green, yellow and blue)

Inputs and Outputs

Input file strictly follows the structure shown below:

1. S – Number of train stations in the network (size of the adjacency square matrix representing the network graph).
2. L – the list of stations.
3. $M - S$ consecutive lines of 2-dimensional adjacency square matrix representing the graph. Elements in the matrix are positive integers denoting the weight of the edge (distance between two stations). A '0' means there is no edge. The ordering of the columns/rows of M is same with the ordering of L.
4. P – a list containing popularity of each station in order of L.
5. G – list of stations serviced by green line.
6. Y – list of stations serviced by yellow line.
7. B – list of stations serviced by blue line.
8. N – number of time ticks (time-units) in the simulation.
9. g, y, b – number of trains per each line (green, yellow, blue)

Sample Input

```
8
changi,tampines,clementi,downtown,chinatown,harborfront,bedok,tuas
0 3 0 0 0 0 0 0
3 0 8 6 0 2 0 0
0 8 0 0 4 0 0 5
0 6 0 0 0 9 0 0
0 0 4 0 0 0 10 0
0 2 0 9 0 0 0 0
0 0 0 0 10 0 0 0
0 0 5 0 0 0 0 0
0.9,0.5,0.2,0.3,0.7,0.8,0.4,0.1
tuas,clementi,tampines,changi
bedok,chinatown,clementi,tampines,harborfront
changi,tampines,downtown,harborfront
10
10,10,10
```

Assume that trains are introduced on each line one at a time and they never exit the network (they start working in the opposite direction once they reach the end of the line).

Output file strictly follows the structure shown below:

1. At each time-unit (tick), your program should output the positions of every train. It should clearly show whether a particular train is waiting at a station or in transit between two train stations. A train is identified by the prefix of the line (g, y, b) and the index of the train (0, 1, 2, 3,...). A station is identified by prefix s followed by the index in the list of stations (L). In our problem scenario, $s0$ represents changi, $s1$ represents tampines, etc.

2. Given the number of trains that run on each line as input, display the average waiting time, the average longest waiting time, and the average shortest waiting time on each line (in time-units).

    - **Average waiting time** for each line is computed by averaging the waiting time for each station on a line throughout the duration of the simulation.
    - **Average longest waiting time** for each line is computed by averaging the longest waiting time at each station on a line.
    - **Average shortest waiting time** for each line is computed by averaging the shortest waiting time at each station on a line.

</>

Sample Output

```
0: g0-s7, g1-s0, y0-s6, y1-s5, b0-s0, b1-s5,
1: g0-s7->s2, y1-s5->s3, g1-s0->s1, b0-s0, b1-s5, g2-s7, g3-s0, y0-s6->s4,
y2-s6, y3-s5,
2: ...
..
..
..
9: ...

Average waiting times:
green: 10 trains -> 6.0, 12.1, 2.1
yellow: 10 trains -> 7.6, 14.5, 4.2
blue: 10 trains -> 9.3, 15.2, 6.7
```

FAQ

Frequently asked questions (FAQ) received from students for this assignment will be answered in this file. The most recent questions will be added at the beginning of the file, preceded by the date label. Check this file before asking your questions.

If there are any questions regarding the assignment, please post on the IVLE forum or email Sunimal (sunimalr@comp.nus.edu.sg).

Assignment Submission Instructions

You are allowed to work in groups of maximum two students for this assignment. You can discuss the assignment with others as necessary but in the case of plagiarism both parties will be severely penalized. Assignment submission will be done in in two rounds.

1. **Mon, 1 Oct, 12pm – Assignment 1 – part 1 [12 marks]**: OpenMP implementation and testcases with report about your implementation design and assumptions. Analyze the execution time of the OpenMP program and briefly discuss the advantages of using simulation (simulation time-units versus execution time).
2. **Mon, 22 Oct, 12pm – Assignment 1 – part 2 [13 marks]**: MPI and OpenMP implementations and testcases with report including implementation details, assumptions, and performance comparison between MPI and OpenMP implementations.

Bonus [3 marks]: Use a script to find the maximum number of trains that can run on each train line (on each implementation) without encountering starvation. In other words, find out how many trains can run on a line at the saturation point (for a specific input graph and line configuration). Add your observations to the report under a separate section titled Bonus.
Your report should include:

- A brief description of your programs design (how they work) and implementation assumptions.
- Any special consideration or implementation detail that you consider non-trivial.
- Details on how to reproduce your results, e.g. inputs, execution time measurement etc.
- Execution time measurement when varying the number of threads/processes (1-64) and the number of cores (1-36). For these measurements, run your implementations on node 2 in our lab (Dell Precision 7820 with Intel Xeon).
- Analyze your measurement results and explain your observations. Performance comparison between MPI and OpenMP implementations should be included in your final report (part 2). State your assumptions.

Submit your assignment before the deadline under IVLE Files. Each student must submit one zip archive named with your student number(s) (A0123456Z.zip - if you worked by yourself, or A0123456Z_A0173456T.zip - if you worked with another student) containing:

1. Your C code along with three sample input files, and the output produced by running your code on these files.
2. `README` file, minimally with instructions on how to execute the code
3. Report in PDF format (a1_report.pdf)

Note that for submissions made as a group, only one most recent submission (from any of the students) will be graded, and both students receive that grade.
Penalty of 5 marks per day for late submissions will be applied.