Computing Laboratory I (CS69011)
Autumn Semester 2023
Assignment - 8

Topic: MultiThreading and Synchronization
Assignment given on: 16.10.2023

In this assignment you need to use the functionalities of the pthread library. You learnt the usage of pthreads in the previous assignment part F. pthread is quite powerful, it lets you parallelize the computation among multiple threads. However, just like any multithreading system, pthread also comes up with a cost—handling race conditions.

What is a "race condition"? Well, that is when two or more threads update a shared data structure simultaneously. Naturally, that will leave the shared data structure in an unpredictable state. So, what is the solution? You use synchronization primitives. We will consider two synchronization primitives in this situation—mutex locks and semaphores.

- What are mutex locks, see here
- What are semaphores? See here, here, and here.

Note that semaphores can also be used for process synchronization. Now, armed with this knowledge, you need to write code for solving the following problem.

Problem: Finding approximate shortest path in a social graph

Problem setup

- Download the edgelist format of a real-world undirected graph (each line is one edge expressed as a set of two nodes) given here: https://snap.stanford.edu/data/loc-brightkite_edges.txt.gz
- The information about the graph is here: https://snap.stanford.edu/data/loc-Brightkite.html
- Now, this forms your base graph. Like any real world graph, edges should get formed and deleted in this graph over time—so you need to implement dynamic random addition and deletion of edges (exact task is below, read on).
- In this dynamic graph, your job is to find shortest paths between pairs of two nodes. However, there are some constraints:
    - You will set up 50 random static nodes and 50 highest degree nodes in the graph (called *landmarks*). You will find the shortest paths between the landmarks and store them. You have to do this again and again over time, as the graph might change.

    - You will divide the nodes into 100 random partitions at the beginning and assign each partition to one landmark. Note that even over time, no nodes are deleted or added, only edges will be deleted or added.
    - In each partition, find shortest paths to the relevant *landmark*.
    - Now you are all set to find approximate shortest paths in the given large graph (58,228 nodes and more than 214K edges). For any given pair of nodes (a, b), find path from a to its landmark (say $landmark_a$) and then b to its landmark (say $landmark_b$) and finally path from $landmark_a$ to $landmark_b$.
    - Combining these three paths you can get your shortest path.
    - Once you find a path, write the path in a file and remove the path (i.e., the edges from the graph).
    - If there is no path, print NO PATH between a, b.
    - However, you need to be smart about creating your own data structure and using mutex locks / semaphores on the data structures. Too many locks/semaphores and your program is really slow, too few locks/semaphores and you will find non-existent paths.

With this in mind, implement the following.

Main Process

- Creating graph data structure and any additional data structures
    - First create the graph data structure from the edgelist given here.
    - Create all necessary data structures / locks / semaphores as needed.
- Creating node pairs for finding approximate path
    - Create a set of 10,000 random pairs of nodes, you need to find paths between them. Write these random pairs of nodes in the "path_to_find.log" file.
- Choose landmark nodes
    - Randomly choose 50 landmark nodes and 50 highest degree landmark nodes.
    - Partition the nodes of the graph in 100 random partitions and assign each partition to one random landmark node.
    - In landmark.log file write your assignment where each line is <landmark node number> <sets of nodes in the assigned partition>
- Creating threads
    - Then create three types of threads—5 *graph_update* threads, 20 *path_finder* threads, 10 *path_stitcher* threads. The threads will exit where there are no more node pairs for finding an approximate path.

graph_update threads

- It will simply update the graph data structure by first randomly tossing a coin. With probability 0.8 it will remove edge and with probability 0.2 it will add edge.
- In the remove edge case, it will find a random edge in the graph (say node0, node1) and remove it. In the "update.log" file, the thread will write, <REMOVE> <node0, node1> <unix_timestamp>

- In the add edge case, it will find a random pair of nodes (say node0, node1) and add an edge, if such an edge exists, then it will do nothing. In the "update.log" file, the thread will write, <ADD> <node0, node1> <unix_timestamp>
- While printing, remember to ensure that the print is not jumbled up due to multiple threads writing.
- If there is a contention of any edge between these threads and any other types of threads then graph_update will have priority.

path_finder threads

- Each thread will just keep on finding paths between landmarks and for each partition find paths between nodes in that partition and the landmark node. Use Dijkstra's algorithm.
- You have to do it smartly, note that a path_finder thread will need to find paths between a node pair either because (i) paths between the node pairs are not computed, or (ii) edges in the paths it found are updated (due to other threads).
- Also you need to make sure the locks/semaphores are used properly.

path_stitcher threads

- Each thread will take a node pair (e.g., <a, b>) where the approximate path is not calculated, and find the approximate path between the nodes using the following algorithm: find path from a to its landmark (say $landmark_a$) and then b to its landmark (say $landmark_b$) and finally path from $landmark_a$ to $landmark_b$. Combining these three paths you can get your shortest path.
- Use the latest paths found by path_finder threads (in the current graph) as much as you can. However, ensure that you are not taking stale paths — paths which are not valid anymore.
- *path_stitcher* threads should have priority over *path_finder* threads.
- Once you find a path the thread will remove those edges in the path from the graph.
- Then the thread will write the following in "path_found.log". It should print "PATH_FOUND <A,B> <A → node1 → node2 … → B > <unix_timestamp>" and then in the next line "PATH_REMOVED <A,B> <A → node1 → node2 … → B > <unix_timestamp>"
- Ensure that you are using the locks/semaphores properly to avoid changes by other threads and ensure not jumbling up the printing
- If no path is found simply print "PATH_NOT_FOUND <A, B> <unix_timestamp>".

Exit condition

- All threads and the main process will exit when all the 10000 paths are processed.
- Before exiting the main process should print the final graph in "final_graph.edgelist" in an edgelist format (just like the original graph was written).

Testing correctness

- Now parse the logs to show the correctness of your code using python (create test.py).
- Show that
    - All PATH_NOT_FOUND are due to either not having a path in the original graph OR due to an edge update event, identify the edge update event (which edges are how updated).
    - Verify if there are PATH_FOUND events which were not possible in the original graph, but happened due to an edge update.

Writing a report

Answer the following questions in details and include a pdf

- What is the data structure used to store the graph? How much RAM did it use?
- What are the additional data structures? Why did you use them
- What are the locks and semaphores used (if any). Specify their use and explain how that prevented race conditions while providing parallelism.
- Finally comment on the errors you encountered and how you debug them.

To submit

- Submit a zip file named <your_roll_no>_Assignment8.zip
- It should include all .c or .cpp files and a README file on how to compile and run the code.
- Include the log files.
- Include the test.py file.
- Include your report.pdf