Indian Institute of Technology, Kharagpur
Department of Computer Science and Engineering

CS39002 : OPERATING SYSTEMS LAB

ASSIGNMENT 3: HANDS-ON EXPERIENCE OF USING SHARED MEMORY

27th Feb, 2023

Group Number : 24

Shivam Raj
20CS10056

Kushaz Sehgal
20CS30030

Rushil Venkateswar
20CS30045

Jatin Gupta
20CS10087

Contents

1. The Main Process

1.1 Description

Loads a graph into a shared memory. Note that the graph is dynamic - new nodes and new edges will be added later. It then spawns the producer and consumer processes. For simplicity, assume all edges of the graph to have weight 1.

1.2 Implementation

The first step is the creation of a Graph class which would store the dataset in the shared memory segment. This Graph class has a central pool of nodes (stored as an array). For each edge in the dataset, we create two nodes in the graph (since the graph is undirected) and update the node_to_head and node_to_tail arrays.

1.2.1 Description of the Graph class

Data members :

- DNode **pool[MAX_LIST]** : central pool of nodes of size MAX_LIST. Each edge in the graph uses 2 nodes out of the pool
- size_t **node_to_head[MAX_NODE]** : stores the index of the head of the adjacency list of each node
- size_t **node_to_tail[MAX_NODE]** : stores the index of the tail of the adjacency list of each node
- int **degree[MAX_NODE]** : stores the degree of each node
- size_t **npool** : pointer to the next free index in the pool
- int **num_of_nodes** : total number of nodes in our Graph

Member functions :

- void **init()** : initializes the data members of Graph class
- DNode * **dnode_alloc()** : returns a pointer to a node if there are unused nodes in the pool
- DNode * **dnode(size_t index)** : returns a pointer to a node in the pool based on the index
- DNode * **dnode_next(const DNode *node)** : returns a pointer to the next node to the current node
- void **add_dnode(DNode *node, int a, int b)** : adds node b to the pool and sets b to be the tail of the adjacency list of node a
- void **dnode_push(int a, int b)** : allocate the nodes a and b in the pool and adds a to b's adjacency list, and vice versa
- void **dijkstra(vector<int> sources, vector<int> &dist, vector<int> &parent)** : multi-source dijkstra's algorithm called by each consumer process
- void **optimized_dijkstra(vector<int> sources, vector<int> &dist, vector<int> &parent)** : multi-source dijkstra's algorithm implemented level-wise

- void **propagate_new_nodes(vector<int> &dist, vector<int> &parent, vector<int> new_nodes)** : updates the non-source nodes (for a particular consumer) among the new nodes added

The main process reads the data and creates the graph in the shared memory. Once that is done, it forks and the child calls the producer process. Similarly, we fork 10 more times and call the consumer process in each child process.

2. The Producer Process

2.1 Description

Wakes up every 50 seconds, and updates the graph in the following way: A number m in the range $[10, 30]$ is chosen randomly, and m new nodes are added to the graph. For each newly added node, select a number k in the range $[1, 20]$ at random; the new node will connect to k existing nodes. The probability of a new node connecting to an existing node of degree d is proportional to d (in other words, a popular node is more likely to get connections from new nodes). Assume all newly added edges to have weight 1.

2.2 Implementation

The producer process launches an infinite loop in which it sleeps for 50 seconds and then selects a random value m. For each of these m new nodes to be added to the graph, we use the discrete_distribution function of random class to ascertain which nodes are to be connected to the new node.

3. The Consumer Process

3.1 Description

These processes wake up once every 30 seconds, and count the current number of nodes in the graph (which is in shared memory). Each consumer process gets mapped to a designated set of nodes. This "mapped set" contains 1/10 of all nodes in the graph. There should be a mapping of the first 1/10 nodes to consumer-1, the second 1/10 nodes to consumer-2, and so on.
Then each consumer process runs the Dijkstra's shortest path algorithms considering all nodes in its mapped set as the source nodes.
A consumer process runs Dijkstra's algorithm as stated above and writes (appends) the set of edges depicting the shortest path to each reachable node from the source node (along with the source node) to a file; then it goes to sleep, to wake up 30 seconds later and repeats the process. Note that there should be 10 such files, one per consumer process. Each consumer thread should know, by some mechanism, its mapped set of nodes, and which file it should append outputs to.

3.2 Implementation

Each consumer is assigned 1/10 of the total number of nodes sequentially as its mapped-set. These consumers then each run the dijkstra function on their mapped-set and generate shortest path for each node in the graph. These paths are written to a file and the consumer process then goes to sleep for 30 seconds.

4. Optimization on Dijkstra

4.1 Description

Note that the producer will add only a limited number of nodes and/or edges to the (much larger) network, which might not change many shortest paths. Design and implement a strategy to optimize the process of re-computing shortest paths. This version will run with a "-optimize" flag to your code.

4.2 Implementation

For each consumer process, $1/10th$ of the new nodes created by the producer are new source nodes whereas the rest are the new non-source nodes.
We provide an optimized_dijkstra function which is implemented level wise so that all nodes which are present in the priority queue at the same time (i.e. the same level) are processed together. This allows us to terminate the function early if there are no updates to the distance values at a particular level.
Along with the optimized_dijkstra function, there is also the propagate_new_nodes function which optimizes the shortest path calculation of non-source nodes. For each of these non-source nodes, we find the neighbour with the least distance from the source set and update the distance of the non-source node. This causes the distances of other neighbours to also be updated, and we continue updating all the neighbours level-wise.

These optimizations allow us to handle the new graph more efficiently, avoiding unnecessary computations. If the '-optimize' flag is used, the faster optimized_dijkstra function is used on the new source set and the new non-source nodes have their distances updated using the propagate_new_nodes function.