CS69001 Computing Laboratory – I

Assignment No: C3

Date: 28–October–2019

Let $G = (V,E)$ be an undirected graph with N vertices. We number the vertices as $V = \{0,1,2,\ldots,N-1\}$. We want to do a breadth-first (BFS) traversal starting from Vertex 0. A standard implementation of BFS is given below. A *visited* array of size N, and a queue Q of vertices are used. Since Q may contain at most N vertices, we use an array of size N to store Q.

```
1. Set $Q[0] = 0$ and $F = B = 0$.
2. Set $visited[0] = 1$ and $visited[i] = 0$ for $i = 1,2,\ldots,N-1$.
3. While $(F \leqslant B)$ /* $Q$ is not empty */, repeat:
    (a) Dequeue: Set $u = Q[F]$, and increment $F$.
    (b) For all vertices $v$ such that $(u,v) \in E$ and $visited[v] = 0$, repeat:
        i. Enqueue: Increment $B$, and set $Q[B] = v$.
        ii. Mark $visited[v] = 1$.
```

In this assignment, you write a multi-process program for implementing this BFS-traversal algorithm. For simplicity, assume that the edge information is provided in the form of the $N \times N$ adjacency matrix. This matrix (along with its dimension N) resides in the shared memory. Moreover, the *visited* array, the queue Q, and the two queue end indices F and B are shared by the processes, and should also reside in the shared memory. Some additional information needed for synchronization and communication are to be stored in the shared memory. This includes (i) the number p of child processes involved in the traversal, (ii) a variable called n_{done}, and (iii) a chunk-definition array C of size $p \times 2$. Each individual item in these data types can be taken as an integer, so the shared memory may be attached as an `int` pointer.

A total of $p+1$ processes are to be involved. The process you launch by running your code is called the *parent process*. It spawns p *child processes* which actually perform the BFS traversal. The parent process is needed to properly synchronize the child processes.

Part 1: Initialize the BFS traversal

The parent process creates a shared-memory segment M for storing all the shared data explained above. You may assume $N = 1000$ and $p = 4$ throughout this assignment. The parent process initializes the adjacency matrix to store a random graph G. Each edge (u,v) is present in G with probability 0.005. After this graph generation, the parent process prints the graph (see sample output for the printing format). Finally, the parent process initiates the BFS traversal by performing Steps 1 and 2 of the BFS algorithm.

The parent process also creates four semaphore arrays. These arrays with their semaphore counts and initial values are listed below.

Semaphore-array name	Number of semaphores	Initial value(s)
S_{LS} (the level-start semaphore)	1	0
S_{LE} (the level-end semaphore)	1	0
S_Q (semaphore for mutual exclusion of the queue Q)	1	1
S_V (semaphores for mutual exclusion of the *visited* array)	p^2	1 (each)

Part 2: Creating the child processes

After completing the initial book-keeping task, the parent process spawns p child processes. These child processes perform Step 3 of the BFS algorithm in parallel. The parent process stays alive for coordinating the work of the p child processes. When all p child processes exit, the parent process removes the shared-memory segment M and all semaphore arrays created by it, and exits itself.

Part 3: Synchronizing the levels

The BFS traversal should proceed level by level. The parent process initializes the level 0 (the root of the BFS tree). The p child processes then explore the other levels until no further level can be added to the BFS tree. The processing of no level may start before the previous level completes.

In order to see why this synchronization is necessary, suppose that the child processes are working at level l of the BFS tree. Let two different child processes Q and Q' be in charge of two vertices u and u' at level l. Suppose also that u has an unvisited neighbor v, and u' has an unvisited neighbor v', and (v, v') is an edge in E. Q enqueues v at level $l+1$, but before Q' gets a chance to enqueue v' at level $l+1$, Q finds v' as an unvisited neighbor v' of v. Q would then add v' at level $l+2$. This is wrong.

The level-wise synchronization is achieved as follows. Each child process waits on the level-start semaphore S_{LS}. The parent process finds the current Q segment $[F, B]$. It breaks the segment into p (almost) equal-sized chunks, and writes the start and the end indices of the p chunks in the chunk array C in the shared memory. After this, the parent process wakes up all the child processes by making p signal calls to S_{LS}. The parent then waits on the level-end semaphore S_{LE}. Before the parent goes to wait, the count n_{done} should be zero.

Each child process, after waking up, processes its chunk of the queue (see the next two parts). When a child process is done with its chunk, it increments n_{done}, and goes to wait on S_{LS} for starting the next level. The last process to finish its chunk at the current level sees p as the incremented value of n_{done}. Before it goes to wait on S_{LS}, it wakes up the parent process by signaling the semaphore S_{LE}. Before the next level begins, n_{done} should be reset to 0 either by the parent process or by the last child process to finish the current level.

Part 4: Mutual exclusion of the queue *Q*

Each child process, while processing its chunk at the current level, discovers the new unvisited neighbors from the vertices of its chunk. The child process accumulates these BFS-tree links in a private array. When all vertices in the chunk are explored, this private information is copied to the shared array Q. Enqueuing involves incrementing the index B and writing a vertex number at $Q[B]$. Two different child processes must not be allowed to enqueue at the same time. The semaphore S_Q should be used for mutually exclusive access of Q by the different child processes. That is, a child process acquiring the lock on Q should be allowed to finish writing its entire private data to Q before another child process can start its own copying process.

Part 5: Mutual exclusion of the *visited* array

When an unvisited neighbor v of a vertex u is located by some child process (at some level), that process marks (u, v) as a link of the BFS tree, so that no other child process can discover v again as an unvisited neighbor. This is done by setting $visited[v] = 1$ in the shared memory. This may lead to a race condition.

If the entire $visited$ array is guarded by a single semaphore, no parallel access of this array is possible. If two child processes want to access two different locations in this array, this does not lead to a race condition. However, if we create N semaphores for guarding individual entries of the $visited$ array, the number of semaphores will be huge (like $N = 1000$). A practical trade-off between these two extreme situations can be achieved as follows.

The entire $visited$ array is guarded by p^2 semaphores in S_V. The entry at index i is guarded by the j-th semaphore in the semaphore array S_V, where $j = i \text{ rem } p^2$. At any point of time, there can be at most p concurrent accesses of $visited$ array entries. There are p^2 locks in total. Assuming that the access patterns (indices) are random, the probability that there is a collision (contention on the same lock) is less than $1/2$, by the birthday paradox. As a result, most of the accesses can proceed in parallel without any race condition.

Sample output: A verbose sample output file is separately linked from the lab website.

Submit a single C/C++ source file. Do not use global/static variables.
Do not use STL queues or vectors. The shared arrays $visited$ and Q are to be managed by you.
