CS4533 Concurrent Programming

Take Home Lab 1

**Group Members**

200045U – Aravinda H.W.K

200742E – Yapa Y.M.C.C.L.C

**Step 1**

**Design Explanation**

The given code is designed to compare serial programming vs parallel operations (membership, insertion, and deletion) on a linked list using mutex locks and read write locks for thread synchronization. Operations are distributed across threads ensuring thread-safe execution before parallel processing as below.

First, the linked list is initialized by inserting n random values, and m random values are generated for operations in array(opr\_values). Then the number of operations (m) is distributed among the number of threads equally. For instance, if there are 10,000 operations (m=10000) and 4 threads (thread\_count=4), each thread will handle 2,500 operations so that each thread works on a specific set of operations.

Inside the ThreadFunction, operations are first categorized based on the ratios specified by the command line arguments (number of member, insert, and delete operations). These operations are stored as characters ('M' for membership, 'I' for insert, and 'D' for delete) in an array.

Then these operations are shuffled using the Fisher-Yates shuffle algorithm. This ensures that the operations within a thread are randomly ordered rather than being executed in a strict sequence as all member operations first, followed by insert and delete.

Finally, each thread iterates through its assigned portion of the opr\_values array. Depending on the operation (M, I, or D), it calls the relevant function. Mutex lock and read write lock will handle the mutual exclusion by locking the shared linked list during each operation.

**Step 3**

**Experimental Results**

Case 1

n = 1,000 and m = 10,000, mMember = 0.99, mIndert = 0.005, mDelete = 0.005

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Implementation | No of threads | | | | | | | |
| 1 | | 2 | | 4 | | 8 | |
| Average | Std | Average | Std | Average | Std | Average | Std |
| Serial | 9312.5 | 2724.8125 |  |  |  |  |  |  |
| One mutex for entire list | 11500 | 2311.462 | 19916.67 | 3175.43 | 17227.27 | 1659.94 | 18171.42 | 1484.93 |
| Read-Write lock | 9931.03 | 952.59 | 9488.37 | 2333.72 | 9955.55 | 3111.23 | 12156.25 | 2315.57 |

Case 2

n = 1,000 and m = 10,000, mMember = 0.90, mIndert = 0.05, mDelete = 0.05

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Implementation | No of threads | | | | | | | |
| 1 | | 2 | | 4 | | 8 | |
| Average | Std | Average | Std | Average | Std | Average | Std |
| Serial | 10885.71 | 1794.95 |  |  |  |  |  |  |
| One mutex for entire list | 16809.16 | 3286.60 | 25222.22 | 1092.91 | 29222.22 | 3308.23 | 25680 | 2322.35 |
| Read-Write lock | 14028.90 | 1365.81 | 20055.55 | 1109.96 | 22900 | 1293.70 | 23891.66 | 3059.07 |

Case 3

n = 1,000 and m = 10,000, mMember = 0.50, mIndert = 0.25, mDelete = 0.25

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Implementation | No of threads | | | | | | | |
| 1 | | 2 | | 4 | | 8 | |
| Average | Std | Average | Std | Average | Std | Average | Std |
| Serial | 22463.41 | 3289.32 |  |  |  |  |  |  |
| One mutex for entire list | 41925.92 | 3440.996 | 71277.78 | 4253.79 | 73565.21 | 3526.85 | 75117.64 | 4284.89 |
| Read-Write lock | 41714.28 | 2492.84 | 79562.50 | 2159.28 | 115000 | 38436.09 | 84900 | 2766.86 |

**Computer used**

* CPU
  + Intel(R) Core(TM) i7-1065G7
  + 4 physical cores
  + 2.40GHz
  + 8 MB Intel® Smart Cache
* Memory
  + size 8GB
  + type DIMM
* Operating system
  + Windows 11
  + 64-bit operating system
* Tools
  + Compiler: gcc
  + Libraries: pthread, sys/time
  + Visual studio code

**Step 5**

Case 1 (mMember = 0.99)

Add graph

In case 1, there are more read-heavy operations compared to write operations. Hence, Read-write locks significantly outperforms mutex locks because, read-write locks allow multiple concurrent reads. Mutex locks, on the other hand, make all threads to wait till one read or write operation ends.

When increasing the number of threads, execution times for read-write lock operations decreases as they scale better compared to mutex locks. Mutex locks cannot differentiate between read and write operations creating create unnecessary contention and serialization even for read-only operations.

Case 2 (mMember = 0.9)

Add graph

In this case, although member operations decrease operations are mostly read operations. But slight increase in insertions and deletions introduces write contentions. Read-write locks still outperform mutexes, but the difference is decreased due to the increased frequency of write operations that require exclusive access to the linked list.

Case 3 (mMember = 0.5)

Add graph

In this case, as write operations are sufficiently large, write operations require exclusive locks, leading to contention among threads trying to perform write operations. Hence, Read-write locks and mutex locks perform somewhat similarly. As the Benefit of concurrency diminishes, increasing the number of threads make more overhead to the overall algorithm resulting in increasing of execution times of high number of threads.