CS4533 Concurrent Programming

Take Home Lab 1

**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 4**

Case 1

In Case 1, with 99% of the operations being Member and the rest 0.5% each being Insert and Delete, it comprises only read operations. If the programs are run with a single thread for all the three implementations the serial, mutex-based, and read-write lock-based all of them work sequentially. Thus, their running times become almost the same. But the mutex and read-write lock implementations still bring more extra overheads with their locking and unlocking. This additional overhead implies that the execution times of the mutex and read-write lock programs are slightly greater than that of the serial program, as can be seen in Graph 1.

Contention in a multi-threaded setting is provided by using a mutex to guard the linked list; this is despite threads reading the list, which could have been done concurrently. This contention raises execution time because threads must wait. Consequently, the execution time for the mutex-based program increases when more threads are used because of increased contention and blocking. In contrast, the read-write lock implementation does allow the reads to proceed in parallel, and hence it reduces execution times with an increasing number of threads being used. However, this improvement in performance diminishes when the ratio of Insert and Delete operations increases.

Graph 1: Execution Time vs. Program for Case 1

Case 2

In Case 2, with 90% operations being Member, and both Insert and Delete being 5%, all programs show increased execution time compared to Case 1. This increase is due to the higher proportion of Insert and Delete operations, which are more time-consuming because of node creation and destruction.

As in Case 1, the performance of Program B (mutex-based) deteriorates with a growing number of threads, but the slope is less steep. This could be because there are fewer chances for parallel execution owing to the reduced proportion of Member operations, and hence less contention over mutexes. Further, Program C—with a read–write lock—takes additional advantage from parallel read operations. However, the improvement in performance is less significant than in Case 1 owing to a higher number of Insert and Delete operations as compared to Case 1.

Graph 2 - Execution Time vs. Program for Case 2

Case 3

In Case 3, where the operations are shared equally among Member, Insert and Delete (50% Member, and 25% each Insert and Delete), all the programs significantly increase their execution times. This is because both Insert and Delete are more time-consuming operations when compared to the Member one.

In this case, due to the high extent of blocking of threads, the mutex-based program, that is Program B, performs worse than the serial one. The gap widens with the decrease in number of threads, since there would be less blocking effect when fewer threads compete for the same resource. Program C (read-write lock) also degrades compared to both previous cases. This is because of the high fraction of write operations, in which the advantage coming from parallel read operations is reduced and the whole parallel programs, B and C, are degraded.

Graph 3 - Execution Time vs. Program for Case 3

In summary, parallel implementations of linked list operations are most effective when the workload is dominated by read operations. As the proportion of write operations increases, the performance of these parallel implementations tends to degrade due to increased thread contention and blocking.