CS4533 Parallel and Concurrent Programming
Lab 1

by
200392E - Mendis D.V.N.
200538J - Ratnayake R.M.P.G.C.K.

Step 1

Three distinct programs are designed to handle each of the following operations on a linked list:
(a) a serial program,
(b) a parallel program utilizing a single mutex for synchronization
(c) a parallel program based on Pthreads with read-write locks.

Each program takes as input the **initial linked list size**, **total number of operations**, **proportion of member, insert, and delete operations** (m_member, m_insert, m_delete), and the **number of threads** to be used.

Each program calculates the total number of member, insert, and delete operations required based on the input proportions and the total number of operations. Then 4 arrays are created with the sizes being equal to the initial linked list size, total number of insert operations, total number of delete operations and total number of member operations. Unique random numbers (in the range of 0 to $2^{16}-1$) are then generated to populate the array for the insert operations and also the array for the initial linked list elements. This is to satisfy the condition that a new value inserted into the list cannot be a value already in the list. Then arrays for delete and member arrays are populated with random numbers without considering their uniqueness.

Then, a linked list is initialized with the values from the initial array. Each thread is assigned a specific rank and provided with arrays corresponding to insert, delete, and member operations. Based on the rank of each thread, it calculates the range of operations it will perform from the total number of insert,delete, and member operations. Threads execute the operations: insert, member, and delete on the linked list using the respective set of values in the arrays.

Step 3

To achieve an accuracy of ±5% and 95% confidence level, it is determined that using 20 samples is sufficient.
Execution time in the below tables are given in ms.

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	18.494	0.253						
One mutex for entire list	19.854	0.417	43.135	1.173	42.894	1.014	45.217	0.837
Read-Write lock	19.380	0.472	14.037	0.222	12.738	0.329	12.801	0.323

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	35.255	0.718						
One mutex for entire list	36.415	0.950	63.523	1.834	62.538	1.224	65.695	2.044
Read-Write lock	36.018	0.393	28.138	0.740	24.714	1.563	26.243	1.518

Case 3

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

Implementation	No of Threads							
	1		2		4		8	
	Average	Std	Average	Std	Average	Std	Average	Std
Serial	73.279	1.507						
One mutex for entire list	78.167	3.530	108.195	4.433	106.816	2.795	109.004	2.241
Read-Write lock	74.912	1.186	98.585	2.009	98.999	1.989	104.517	2.855

System Specification

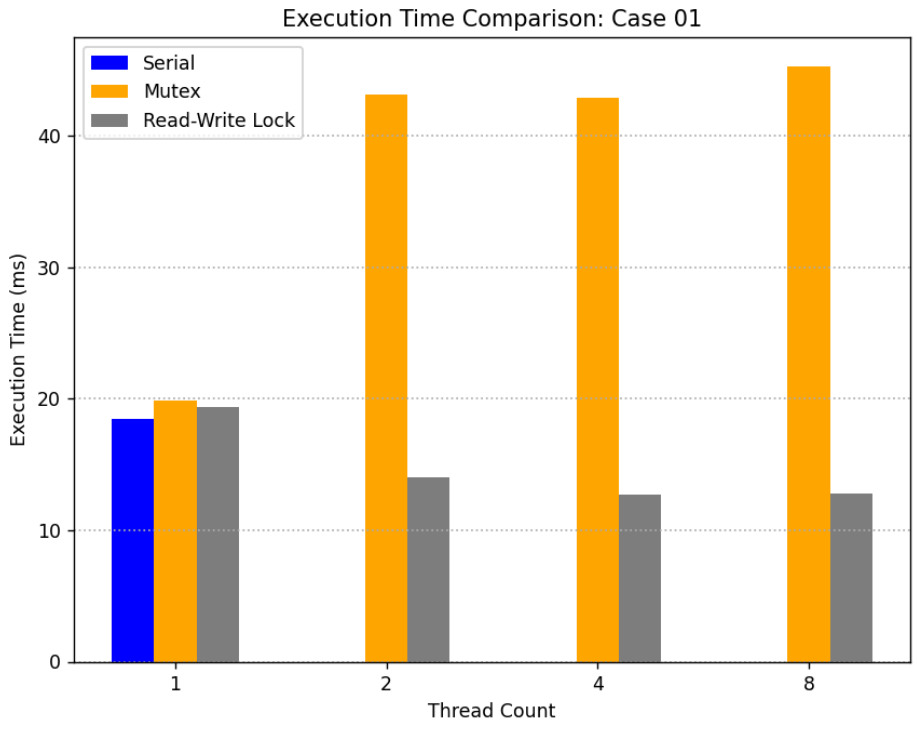
The program was executed in the Google Cloud Platform with the following specifications.

OS	Ubuntu 22.04.4 LTS
OS Version	22.04
OS Config agent version	20240320.00-0ubuntu1~22.04.1
Machine type	n1-standard-32
CPU Platform	Intel Haswell
Architecture	x86/64
vCPU	32
Memory	120 GB
Storage	2800 GB
GPUs	1 x NVIDIA T4

Step 4

Case 1

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



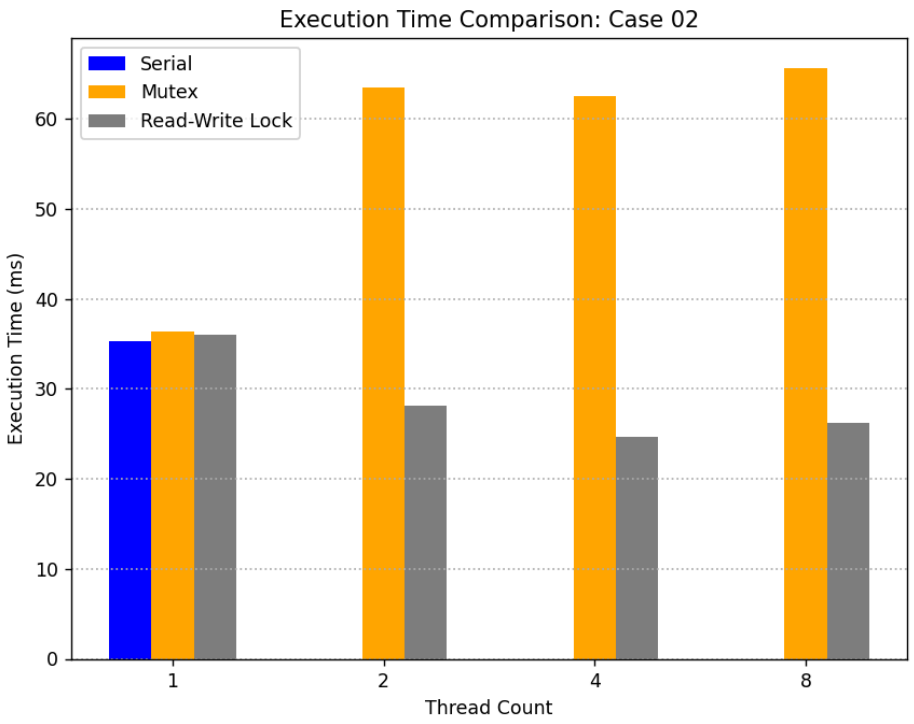
In the single-threaded case, since all three programs are running with only one thread, they essentially behave as serial programs. As a result, there are no significant differences in their execution times. However, the program using a mutex and the program using a read-write lock show slightly higher runtimes compared to the purely serial execution. This is due to the additional overhead caused by locking and unlocking the mutexes and read-write locks, even though these operations are unnecessary in this case, as only a single thread is involved. Consequently, both programs experience slightly longer execution times compared to the program without any synchronization mechanisms.

When using 2, 4, or 8 threads, the program with a mutex exhibits significantly higher execution times, more than double the time taken in the single-threaded case. This is because when one thread acquires the mutex, it blocks all other threads, even during read operations, leading to inefficient thread usage and increased contention for the lock. This lock contention becomes more severe as the thread count increases, causing execution times to remain consistently high regardless of the number of threads used.

In this case, since 99% operations are read operations, the program with a read-write lock shows a significant improvement in execution time as the number of threads increases. This is because read-write locks allow multiple threads to read concurrently, reducing contention during read operations. As shown in the graph, the execution time decreases with higher thread counts. This improvement is primarily due to the ability of the read-write lock to scale read operations efficiently with higher thread counts, enabling better performance in read-heavy workloads.

Case 2

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



In the single-threaded program, execution times for all three techniques (Serial, Mutex, and Read-Write Lock) are very similar. The mutex and read-write lock techniques add some overhead, resulting in slightly higher execution times than the purely serial program. However, because there is only one thread and there is no lock contention, these overheads are minimal. In this case, the synchronization techniques are almost redundant, which accounts for the marginal difference in execution times.

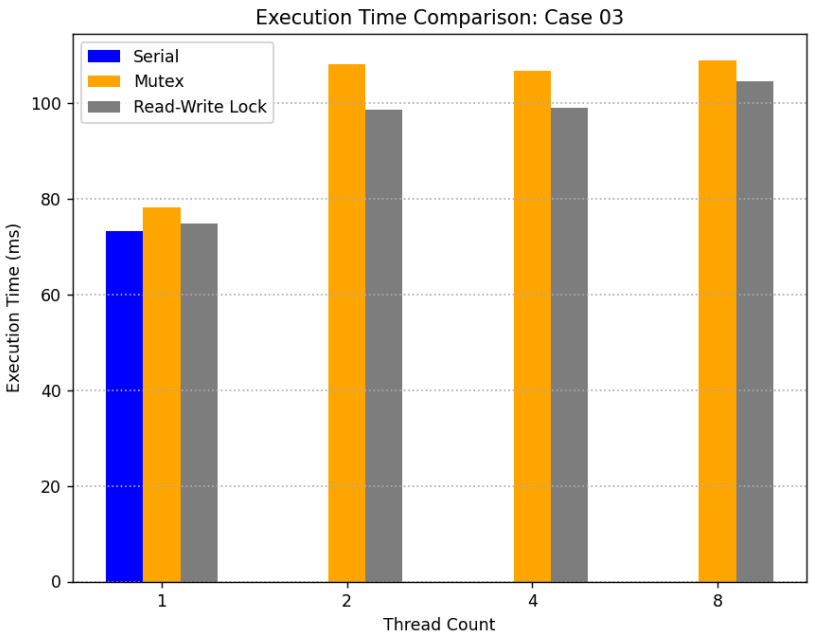
When using 2, 4, or 8 threads, the program with the mutex shows a large increase in execution time as the thread count increases. In this case, the mutex locks the entire critical section, not just for write operations (inserts and deletes) but also for read operations. This means that whenever any thread tries to perform an insert, delete, or read operation, all other threads are blocked from accessing the shared resource until the current thread releases the lock . As the thread count rises, this leads to increased contention, as more threads are forced to wait for access to the critical section, creating a significant bottleneck. Consequently, the program becomes less efficient as more threads are added, with execution times increasing due to the heavy contention for the mutex.

Read-write lock mechanism performs better than the mutex, especially as the thread count increases. This is because the read-write lock allows multiple threads to perform read operations concurrently, which make up 90% of the operations in Case 02. While there are still 10% of operations involving writes (which cause some locking and prevent concurrent execution during writes), the program benefits significantly from the high percentage of read operations.

However, in both the mutex and read-write lock approaches, the increased proportion of write operations (10%) in Case 02 results in a slight increase in execution time compared to Case 01.

Case 3

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



In the single-threaded example, the execution times for all three methods (Serial, Mutex, and Read-Write Lock) are almost identical, as in cases 1 and 2. When only one thread is active, Mutex and Read-Write Lock methods have low overhead as there is the absence of lock contention. Similar to Case 1 and Case 2, the synchronization algorithms add a little bit of overhead to the purely serial version, but this difference is negligible due to the lack of thread contention.

When using 2, 4, or 8 threads, the program with the mutex shows a large increase in execution times as the thread count increases just as similar to case 1 and 2. Here also, despite the proportion of each operation type, the mutex locks the critical section at each operation which forces the other threads to be waiting until the mutex unlock the critical section. So this lock contention creates a significant bottleneck by increasing the execution time.

Despite the increased proportion of write operations, the read-write lock mechanism outperforms the mutex based approach. The read-write lock lets multiple threads conduct read operations concurrently, accounting for 50% of the effort in this case. However, because the remaining 50% of operations involve write operations, which require exclusive access to the shared resource, the program faces higher contention than in Case 2.