CS4532

Concurrent Programming

Lab 1

Group Members

190601D - W. B. Somarathne
190077A - L. A. R. L. Bandara

Contents

System specifications

CPU

- Intel® Core™ i7-8750H
- Total Cores 6.
- Total Threads 12.
- Max Turbo Frequency 4.10 GHz.
- Intel® Turbo Boost Technology 2.0 Frequency‡ 4.10 GHz.
- Processor Base Frequency 2.20 GHz.

Memory

- 16 GB

Operating system

- Ubuntu 21.04.3
- 64-bit

Tools

- Compiler: gcc
- Libraries: pthread, sys/time
- VS Code

1. Design

In order to generate m_{Member} , m_{Insert}, and m_{Delete} operations of each type using given number of threads we used an array based implementation. An array was created containing 0,1 and 2 in counts m_{Member}, m_{Insert}, and m_{Delete} respectively. Then to ensure randomness in the operations we shuffled the list using Fisher-Yates shuffle algorithm. Next this array was passed to all the threads and the starting and ending indexes of each thread operation set was generated based on the rank of the thread and the total number of threads. This way each thread got an equal partition from the total number of operations to be done. Next, each thread was run with thread-safe features like read-write locks and mutexes. A switch case was used to run each linked-list operation based on the integer value received from the array.

2. Approach

A linked list is implemented as a:
a) Serial program
b) Parallel program (based on Pthreads) with one mutex for the entire linked list
c) Parallel program (based on Pthreads) with read-write locks for the entire linked list

Each of the programs has been run 385 times[1] to obtain an accuracy of ±5% and a confidence level of 95%. For each of the given 3 cases, 10,000 operations(Insert, Delete, or Member) have been done on a linked list with 1000 elements and the average and standard deviation of the runs have been taken and presented below.

3. Experiment Results

[1] - Selecting the number of Samples

The requirement was to obtain results within an accuracy of ±5% and 95% confidence level.

$$n = \left(\frac{100zs}{r\bar{x}}\right)^2$$

n = number of samples
z value = 1.96 (assuming normal distribution)
r = accuracy = 5
x = sample mean (obtained by considering different samples)
s = sample standard deviations (obtained by considering different samples)

In order to achieve this, we used the following equation.

The sample mean, sample standard deviation was obtained by executing the program for different sample sizes. Using this we obtained a suitable sample size of 385.

Case 1:

n = 1,000 and m = 10,000, m_{Member} = 0.99, m_{Insert} = 0.005, m_{Delete} = 0.005

Implementation	**Number of Threads**							
	1		**2**		**4**		**8**	
	Avg	**Std**	**Avg**	**Std**	**Avg**	**Std**	**Avg**	**Std**
Serial	21415.14	698.86						
One mutex for entire list	21515.49	635.49	35762.47	1062.01	35074.32	1302.59	36845.17	1039.95
Read-Write Lock	22182.39	621.40	13010.66	653.12	10149.85	623.08	10942.38	687.07

Case 2:

n = 1,000 and m = 10,000, m_{Member} = 0.90, m_{Insert} = 0.05, m_{Delete} = 0.05

Implementation	**Number of Threads**							
	1		**2**		**4**		**8**	
	Avg	**Std**	**Avg**	**Std**	**Avg**	**Std**	**Avg**	**Std**
Serial	26954.69	638.07						
One mutex for entire list	27132.56	600.16	41500.75	831.71	41463.91	1177.09	42815.88	902.71
Read-Write Lock	27660.43	858.27	21991.62	532.67	22800.95	744.13	26451.89	935.93

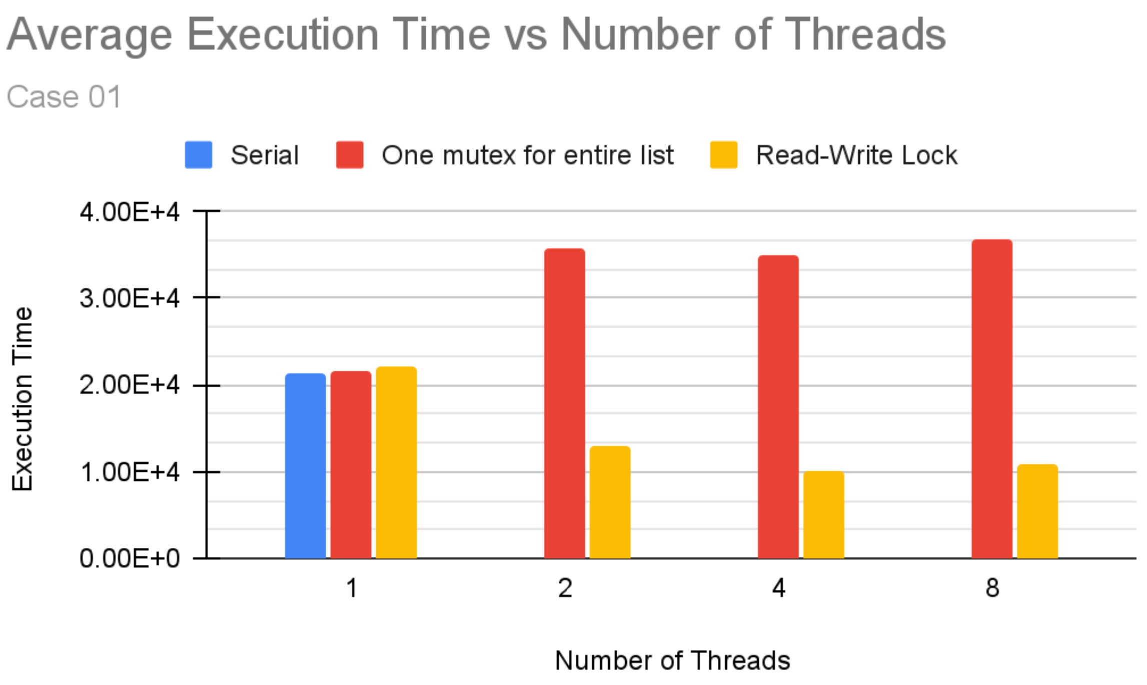
Case 3:

n = 1,000 and m = 10,000, m_{Member} = 0.5, m_{Insert} = 0.25, m_{Delete} = 0.25

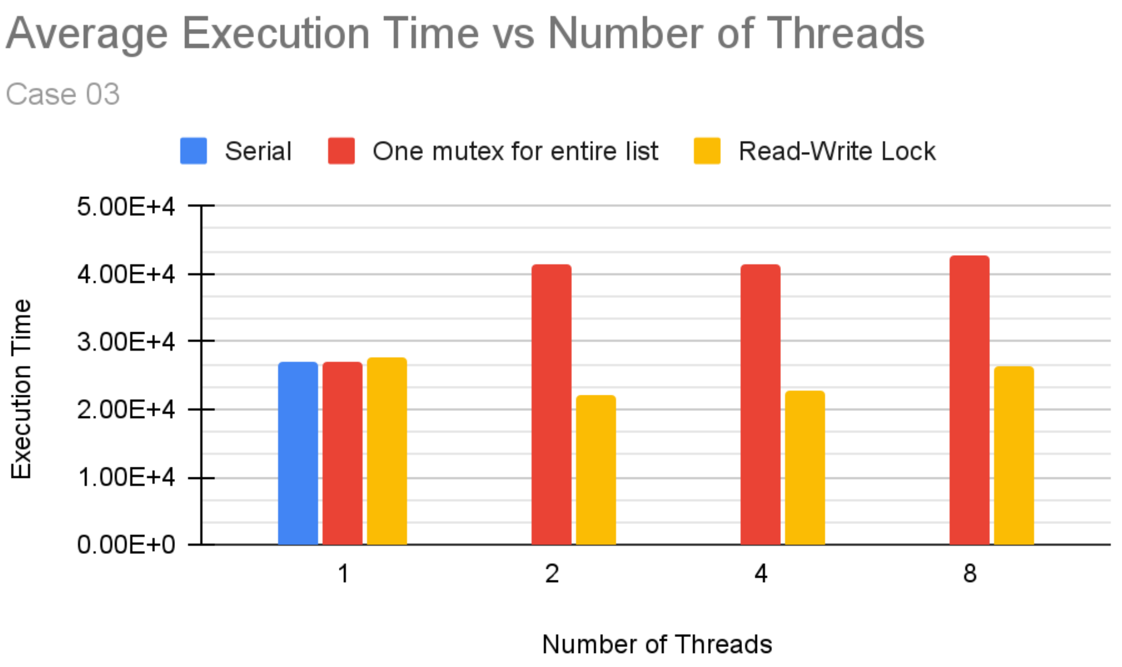
Implementation	**Number of Threads**							
	1		**2**		**4**		**8**	
	Avg	**Std**	**Avg**	**Std**	**Avg**	**Std**	**Avg**	**Std**
Serial	54512.68	4163.45						
One mutex for entire list	54229.62	4148.64	71170.31	3899.38	74258.74	4627.50	80206.00	7494.58
Read-Write Lock	57934.04	5991.31	72864.71	3418.78	76788.86	3797.54	84046.54	4008.66

4. Average Execution Time Graphs

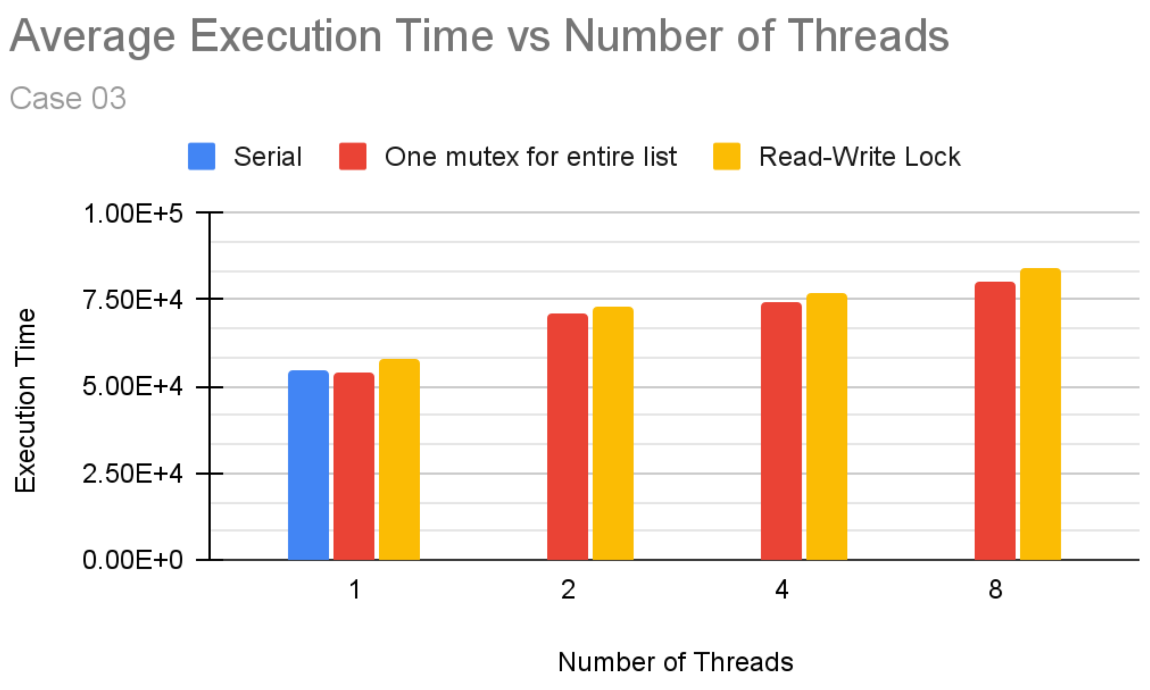
Case 01:



Case 02:



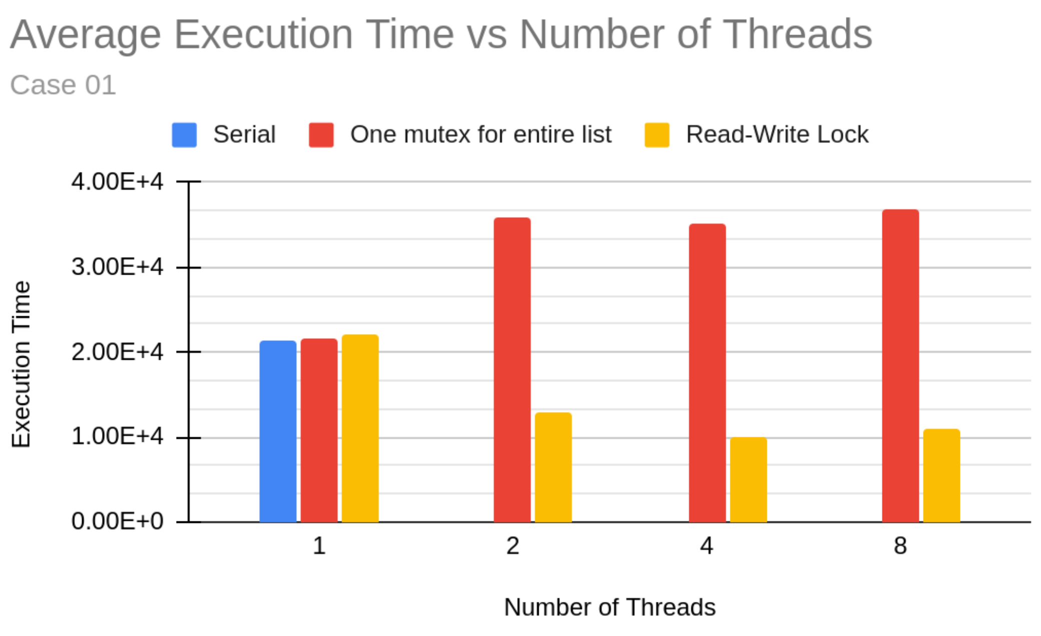
Case 03:

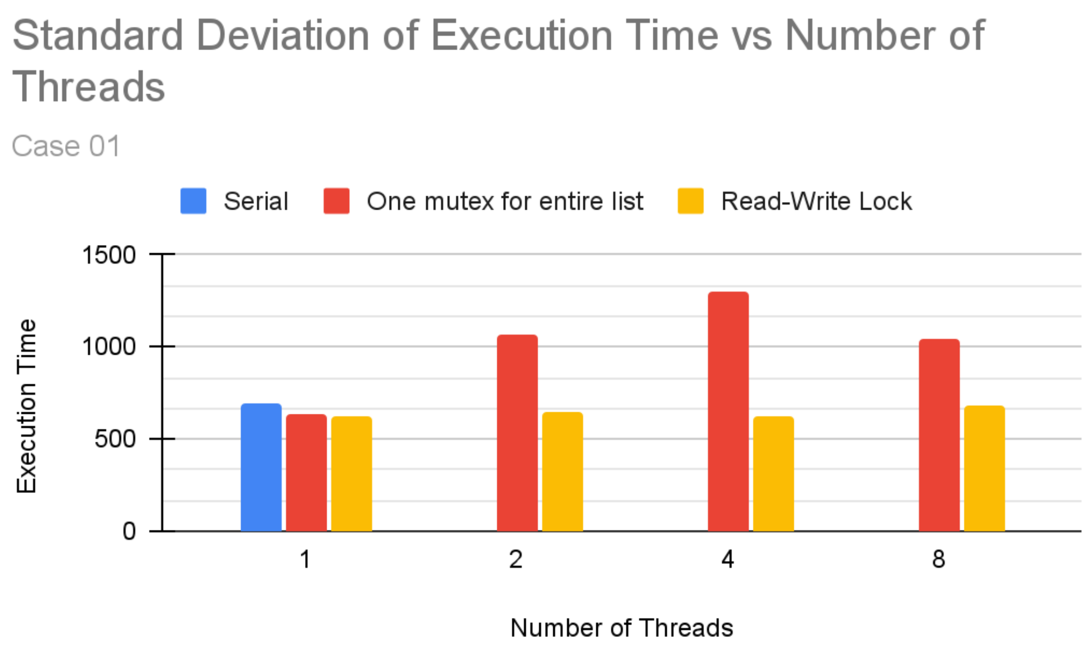

Average Execution Time vs Number of Threads
Case 03
Serial
One mutex for entire list
Read-Write Lock
Execution Time
1.00E+5
7.50E+4
5.00E+4
2.50E+4
0.00E+0
1
2
4
8
Number of Threads

5. Analysis

Case 01:

Charts:





Serial Implementation:

Average Execution Time: 21415.14 microseconds
Standard Deviation of Execution Times: 698.86 microseconds

The serial implementation serves as our baseline. It is the simplest approach with no parallelization. As expected, it has the lowest execution time because there is no overhead associated with synchronization between threads. The standard deviation is relatively low, indicating that the execution times are consistent across multiple runs.

Parallel Implementation with One Mutex for the Entire List:

Average Execution Times:

- 1 Thread: 21515.49 microseconds
- 2 Threads: 35762.47 microseconds
- 4 Threads: 35074.32 microseconds
- 8 Threads: 36845.17 microseconds

Standard Deviation of Execution Times:

- 1 Thread: 635.49 microseconds
- 2 Threads: 1062.01 microseconds
- 4 Threads: 1302.59 microseconds
- 8 Threads: 1039.95 microseconds

In this parallel implementation, all threads share a single mutex to access the linked list. As the number of threads increases, the execution time also increases. This is likely due to increased contention for the mutex, leading to threads waiting for access, which reduces parallelism. The standard deviation remains relatively low, indicating consistent performance across multiple runs.

Parallel Implementation with Read-Write Lock for the Entire List:

Average Execution Times:

- 1 Thread: 22182.39 microseconds
- 2 Threads: 13010.66 microseconds
- 4 Threads: 10149.85 microseconds
- 8 Threads: 10942.38 microseconds

Standard Deviation of Execution Times:

- 1 Thread: 621.40 microseconds
- 2 Threads: 653.12 microseconds
- 4 Threads: 623.08 microseconds
- 8 Threads: 687.07 microseconds

In this parallel implementation, a read-write lock is used to allow multiple threads to read the linked list concurrently while ensuring exclusive access during writes. The execution times are slightly better than the one-mutex approach, especially as the number of threads increases. The reduced contention on reads due to the read-write lock helps maintain better parallelism. The standard deviation remains relatively low, indicating consistent performance.

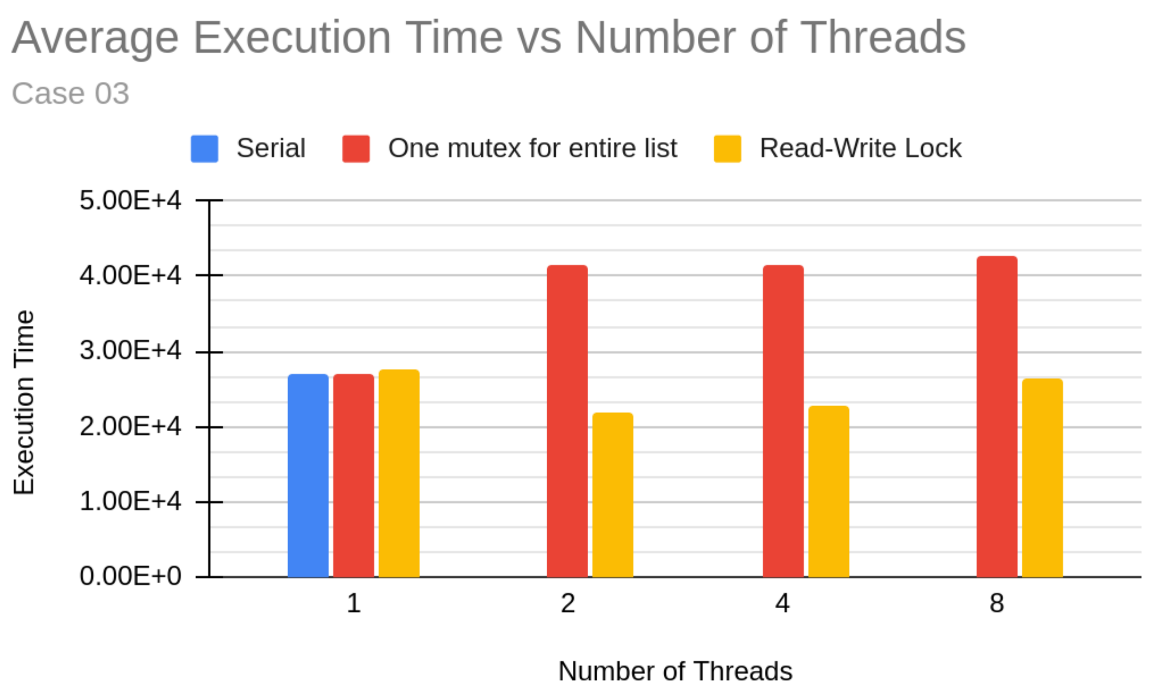
Observations and Evaluation:

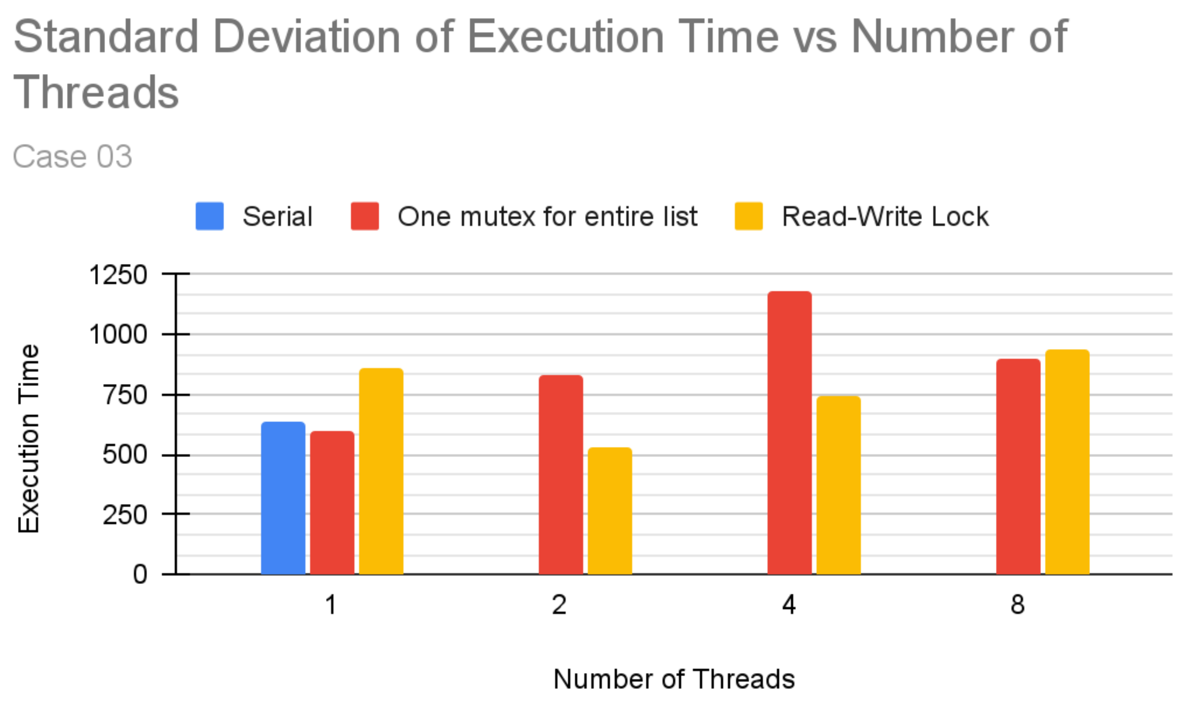
- As expected, the serial implementation outperforms the parallel implementations in terms of execution time because it doesn't have the overhead of synchronization.
- Both parallel implementations exhibit increasing execution times as the number of threads increases. This phenomenon is likely due to contention for shared resources (mutex or read-write lock) among threads. Adding more threads initially increases parallelism but eventually leads to diminishing returns and increased contention.
- The parallel implementation with a read-write lock generally performs better than the one-mutex approach, particularly as the number of threads increases. This is because the read-write lock allows multiple threads to read concurrently, reducing contention for read access and improving parallelism.
- The standard deviations for all cases are relatively low, indicating consistent performance across multiple runs, which is a positive outcome.

In conclusion,, the findings suggest that when implementing a parallel linked list, the choice of synchronization mechanism can significantly impact performance. The read-write lock approach tends to perform better in scenarios with multiple read-heavy operations, while the one-mutex approach may be suitable for scenarios with more balanced read and write operations. However, the diminishing returns with increasing threads should be carefully considered when choosing the level of parallelism for a given workload.

Case 02:

Charts:





Serial Implementation:

Average Execution Times: 26954.6883 microseconds
Standard Deviation: 638.07 microseconds

In the serial implementation, there is only one thread, so it performs all the operations sequentially. As with dataset 1, this approach serves as a baseline for comparison. The average execution time is consistent, and the standard deviation is relatively low, indicating that the execution times are relatively stable.

Parallel Implementation with One Mutex for Entire List:

Average Execution Times:

- 1 Thread: 27132.56 microseconds
- 2 Threads: 41500.75 microseconds
- 4 Threads: 41463.91 microseconds
- 8 Threads: 42815.88 microseconds

Standard Deviation:

- 1 Thread: 600.16 microseconds
- 2 Threads: 831.71 microseconds
- 4 Threads: 1177.09 microseconds
- 8 Threads: 902.71 microseconds

Similar to dataset 1, the execution times for the parallel implementation with one mutex show an unexpected trend. As the number of threads increases, the execution times also increase. This suggests that there is contention for the mutex, causing threads to spend more time waiting.
The standard deviation also increases with more threads, indicating more variability in execution times.

Parallel Implementation with Read-Write Lock for Entire List:

Average Execution Times:

- 1 Thread: 27660.43 microseconds
- 2 Threads: 21991.62 microseconds
- 4 Threads: 22800.95 microseconds
- 8 Threads: 26451.89 microseconds

Standard Deviation:

- 1 Thread: 858.27 microseconds

- 2 Threads: 532.67 microseconds
- 4 Threads: 744.13 microseconds
- 8 Threads: 935.93 microseconds

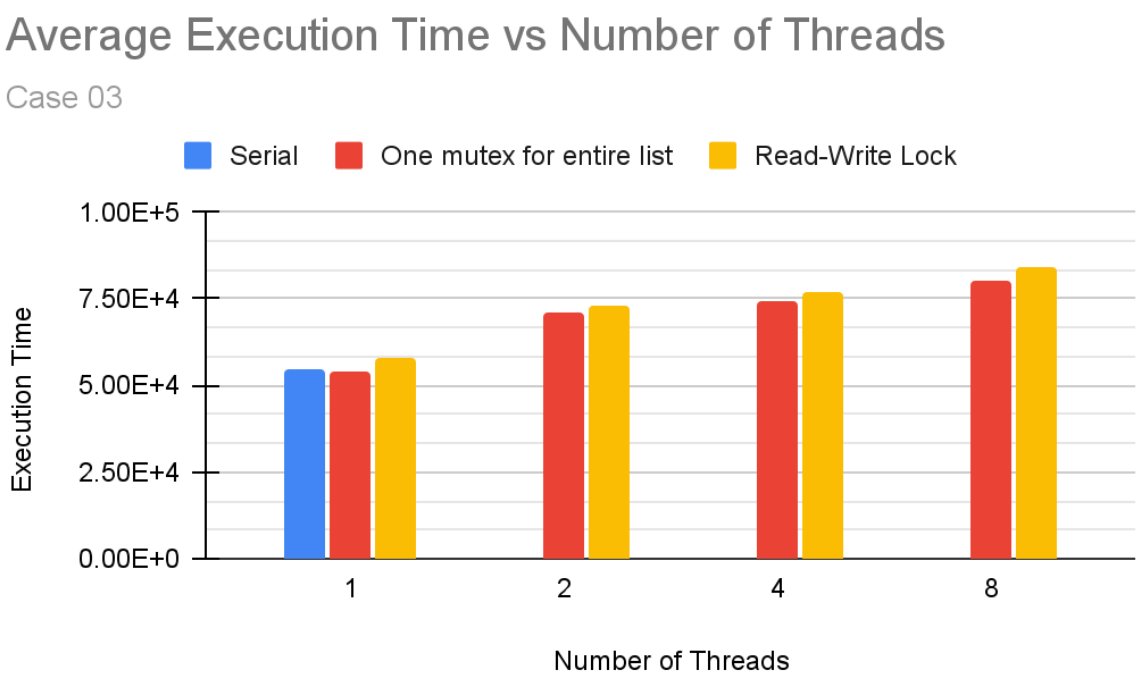
The parallel implementation with read-write locks exhibits better scalability than the mutex-based approach for this dataset. Execution times decrease as the number of threads increases, up to a certain point.
Similar to dataset 1, there is a slight increase in execution times with more than 4 threads, indicating potential contention for the read-write locks.
The standard deviation is relatively stable, indicating consistent execution times.

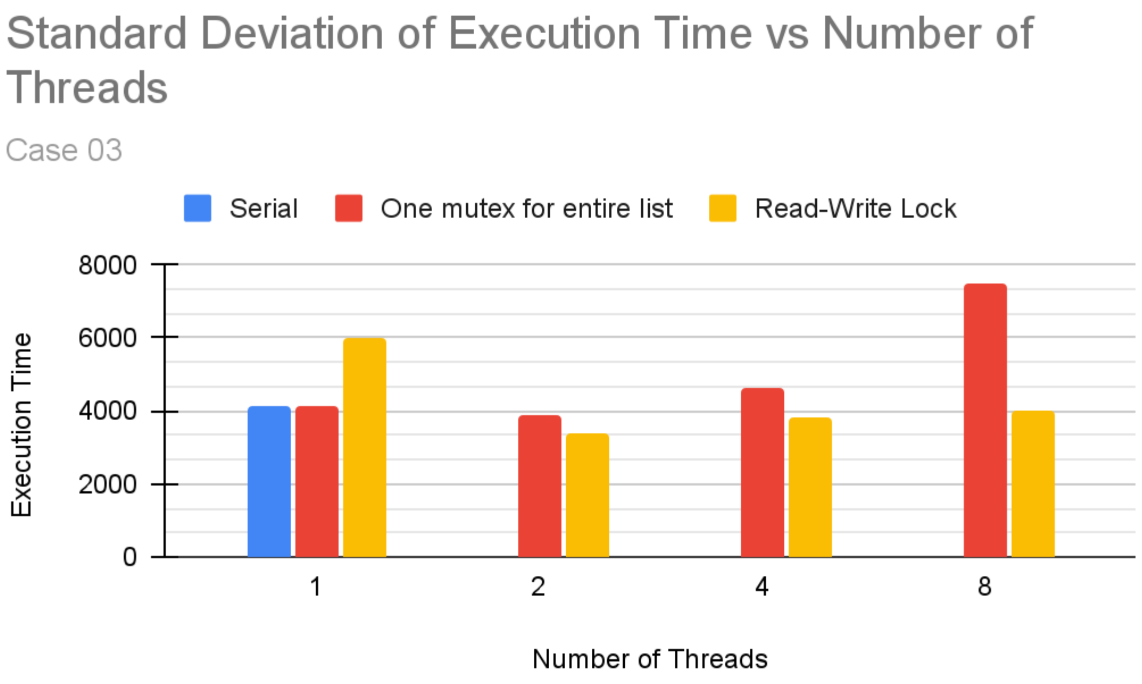
Observations and Evaluation:

The serial implementation again has the lowest execution time because it doesn't involve any thread synchronization overhead.
The parallel implementation with one mutex for the entire list exhibits a similar performance degradation as observed in dataset 1. Contention for the mutex leads to slower performance as the number of threads increases.
The parallel implementation with read-write locks shows better scalability than the mutex-based approach for this dataset. However, it also experiences a slight increase in execution times with more than 4 threads, suggesting possible contention for the locks.
Standard deviations provide insights into the variability of execution times. Lower standard deviations indicate more predictable performance.
In summary, the choice of synchronization mechanism continues to have a significant impact on performance. Read-write locks appear to perform better than a single mutex for this dataset, but further optimizations may be needed to handle larger numbers of threads effectively. These observations reinforce the need to carefully select synchronization strategies based on the specific characteristics of the dataset and hardware configuration.

Case 03:

Charts:





Serial Implementation:

Average Execution Times: 54512.68 microseconds
Standard Deviation: 4163.45 microseconds

As in the previous datasets, the serial implementation serves as a baseline for comparison. It executes all operations sequentially, and the average execution time is relatively consistent. The standard deviation is moderate, indicating some variability in execution times.

Parallel Implementation with One Mutex for Entire List:

Average Execution Times:

- 1 Thread: 54229.62 microseconds
- 2 Threads: 71170.31 microseconds
- 4 Threads: 74258.74 microseconds
- 8 Threads: 80206.00 microseconds

Standard Deviation:

- 1 Thread: 4148.64 microseconds
- 2 Threads: 3899.38 microseconds
- 4 Threads: 4627.50 microseconds
- 8 Threads: 7494.58 microseconds

Similar to previous datasets, the parallel implementation with one mutex exhibits increasing execution times as the number of threads increases. This suggests that contention for the mutex is causing threads to spend more time waiting.
The standard deviation also increases with more threads, indicating greater variability in execution times.

Parallel Implementation with Read-Write Lock for Entire List:

Average Execution Times:

- 1 Thread: 57934.04 microseconds
- 2 Threads: 72864.71 microseconds
- 4 Threads: 76788.86 microseconds
- 8 Threads: 84046.54 microseconds

Standard Deviation:

- 1 Thread: 5991.31 microseconds
- 2 Threads: 3418.78 microseconds
- 4 Threads: 3797.54 microseconds
- 8 Threads: 4008.66 microseconds

Similar to previous datasets, the parallel implementation with read-write locks exhibits better scalability than the mutex-based approach for this dataset. Execution times decrease as the number of threads increases, up to a certain point.
There is still a slight increase in execution times with more than 4 threads, suggesting potential contention for the read-write locks.
The standard deviation is relatively stable, indicating consistent execution times.

Observations and Evaluation:

The serial implementation again has the lowest execution time due to the absence of thread synchronization overhead.
The parallel implementation with one mutex for the entire list shows the same performance degradation pattern observed in previous datasets, with increased contention for the mutex leading to slower performance as the number of threads increases.
The parallel implementation with read-write locks continues to perform better than the mutex-based approach for this dataset. However, there is still a slight performance degradation with more than 4 threads, indicating potential contention for the locks.
Standard deviations provide insights into the variability of execution times. While they are relatively stable, they do show some increase with more threads.
In summary, the choice of synchronization mechanism remains critical for performance in dataset 3. Read-write locks continue to outperform a single mutex but may still face contention issues with a large number of threads. Further optimizations, such as fine-grained locking or lock-free data structures, may be necessary to achieve better parallel performance. These observations underline the importance of considering the dataset characteristics and hardware configuration when designing and selecting synchronization strategies.