CS4532 - Parallel and Concurrent Programming

Take Home Lab 1

Ranasinghe W.M.D.S.S.
Wijesinghe U.M.

Step 1

Design a solution such that you can generate mMember, mInsert, and mDelete operations of each type using a given number of threads. Briefly explain your design.

Consider the following code:

```
void generate_operation_order(int *operations, double M_member, double
M_insert, double M_delete)
{
   // Calculate number of operations for each type
   int member_ops = M * M_member;
   int insert_ops = M * M_insert;
   int delete_ops = M * M_delete;

   // Fill array with exact number of each operation
   int index = 0;
   for (int i = 0; i < member_ops; i++)
       operations[index++] = 0; // 0 represents member

   for (int i = 0; i < insert_ops; i++)
       operations[index++] = 1; // 1 represents insert

   for (int i = 0; i < delete_ops; i++)
       operations[index++] = 2; // 2 represents delete

   // Shuffle the array to randomize the order of operations
   for (int i = 0; i < M; i++)
   {
       int j = rand() % M;
       int temp = operations[i];
       operations[i] = operations[j];
       operations[j] = temp;
   }
}
```

What we do here is, we denote an operation type by a given digit (0, 1, 2). Then according to the fraction, we fill an array with the required number of digits corresponding to each operation type. Finally, we can shuffle the list to randomise the order of operations.

We can then pass this array as an argument and obtain the operation type corresponding to the i^{th} operation by doing something like the following code:

```
int *operations = args->operations;
   int *values = args->values;
   int ops_per_thread = args->ops_per_thread;

   // Perform the operations based on the pre-made operations array and
random values array
   for (int i = 0; i < ops_per_thread; i++)
   {

       if (operations[i] == 0)
       {
           int value = rand() % 65536; // Random value between 0 and
2^16-1
           member_rwlock(value);
       }
       else if (operations[i] == 1)
       {
           int value = values[i]; // Use the pre-generated random value
           insert_rwlock(value);
       }
       else
       {
           int value = rand() % 65536; // Random value between 0 and
2^16-1
           delete_rwlock(value);
       }
   }
```

In fact, we can extend this approach for our random values too, as seen under the second if condition in the code block.

Step 2

Implement the link list using the above three approaches.

Implementation available in the code (zip file).

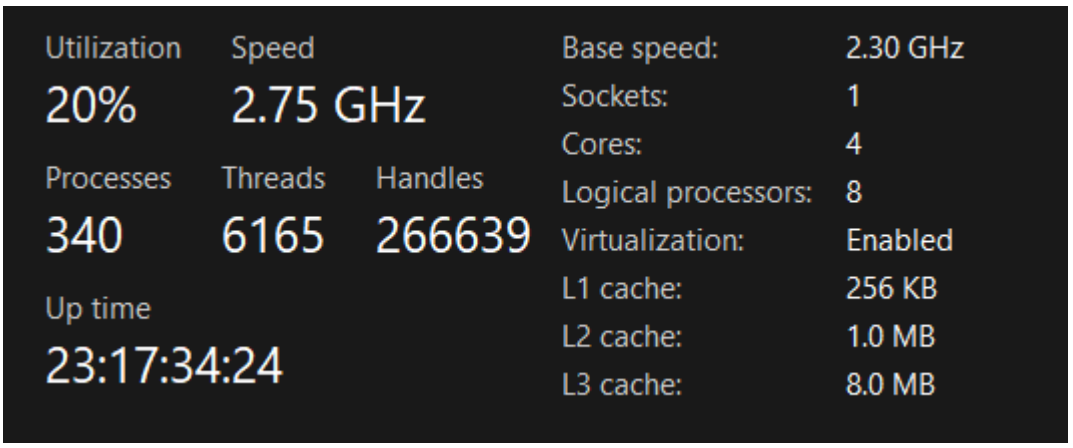
Step 3

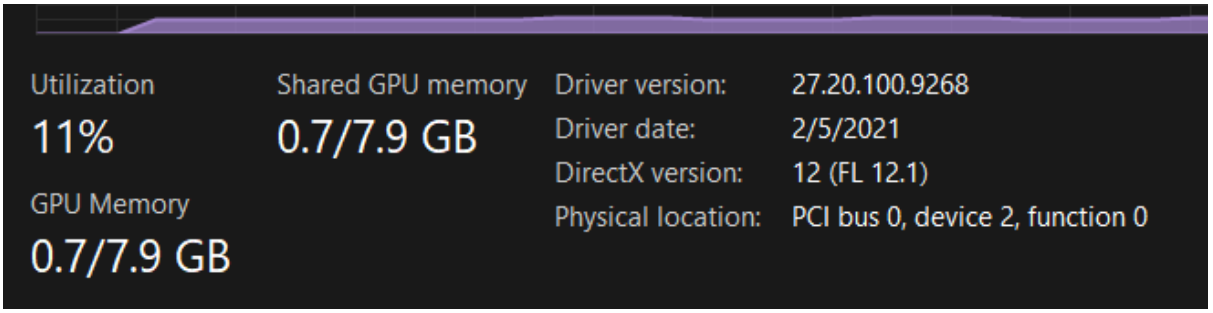
Run your program a sufficient number of times under the following three cases, and then fill up the given tables using the execution time of your programs. While determining the execution time, consider only the time taken to perform m operations and ignore other overheads such as initially populating the link list.

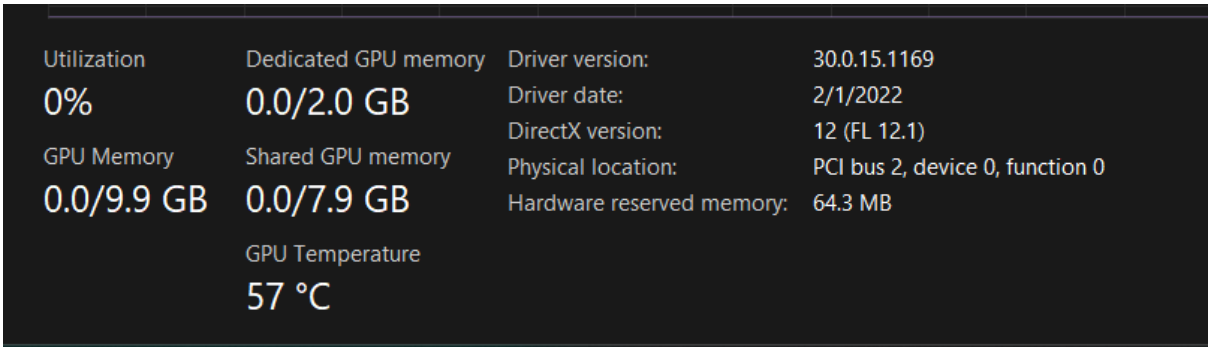
Specs of the machine used

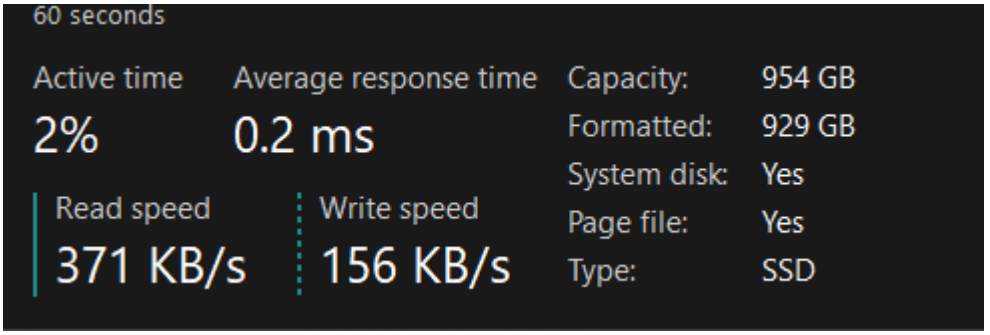
Device name	LAPTOP-4AOFPDB3
Processor	Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz
Installed RAM	16.0 GB (15.9 GB usable)
System type	64-bit operating system, x64-based processor
Pen and touch	No pen or touch input is available for this display

Edition	Windows 11 Home Single Language
Version	23H2
Installed on	4/6/2023
OS build	22631.4169
Experience	Windows Feature Experience Pack 1000.22700.1034.0







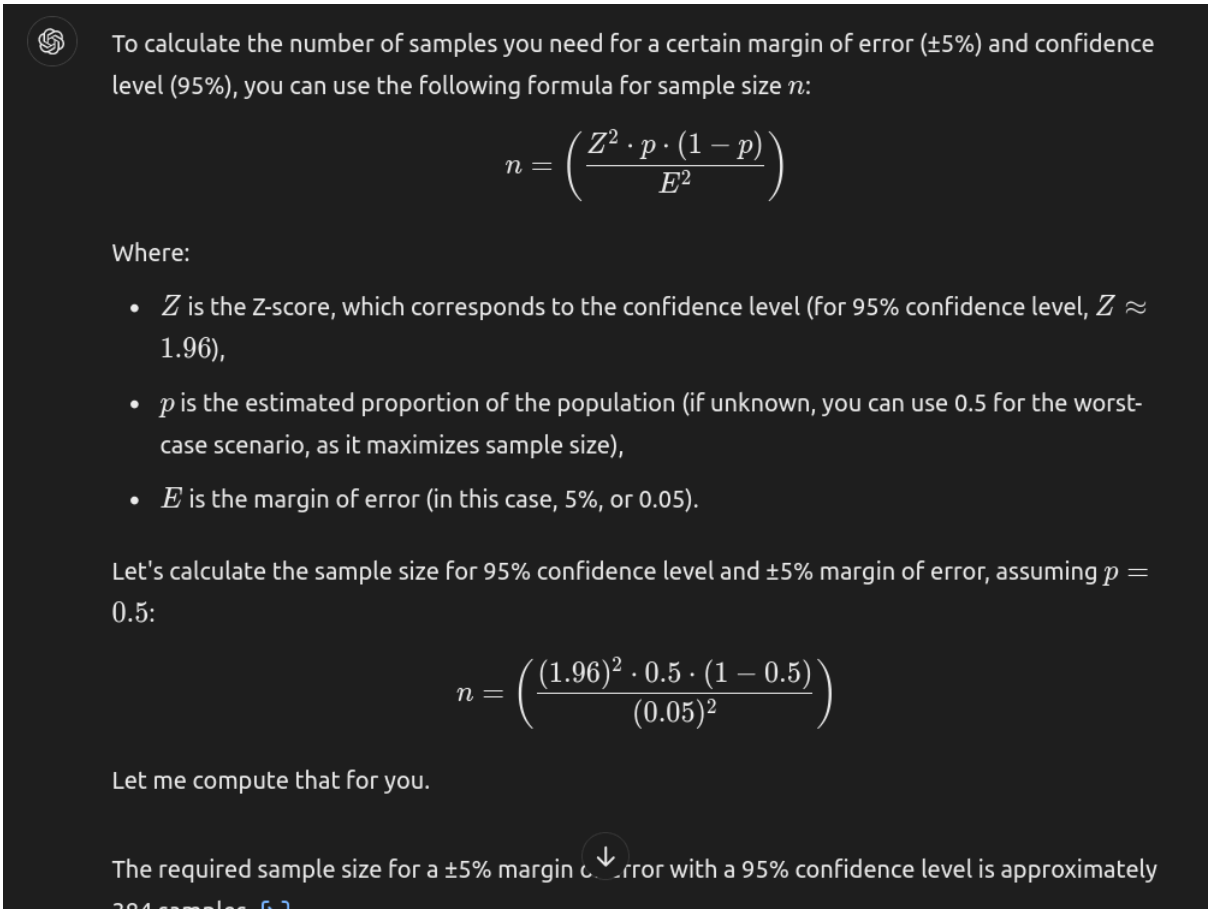


Determining sample size

Consider the following formula we obtained by running a maths source through chatGPT, as well as the pre-calculated table of values we found on a separate maths source. Based on these, we found the number **385** to be required for a confidence interval of 95% with an accuracy of +-5%.

There is an interesting trade-off between the level of confidence and the sample size that shows up here when considering the cost of sampling. The table below shows the appropriate sample size at different levels of confidence and different margins of error, assuming $p = 0.5$. Looking at each row, we can see that for the same margin of error, a higher level of confidence requires a larger sample size. Similarly, looking at each column, we can see that for the same confidence level, a smaller margin of error requires a larger sample size.

Required Sample Size (90%)	Required Sample Size (95%)	Margin of Error
1691	2401	2%
752	1067	3%
271	384	5%
68	96	10%



Performance Tables

Case 1

n = 1000, m = 10000, mMember = 0.99, mIndert = 0.005, mDelete = 0.005

Implementation	No of Threads							
	1		2		4		8	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
Serial	0.041958	0.006542						
One mutex	0.041696	0.007086	0.063327	0.00942	0.069481	0.01418	0.071169	0.009621
Read Write lock	0.041652	0.004326	0.025719	0.004007	0.018852	0.007239	0.016675	0.00359

Case 2

n = 1000, m = 10000, mMember = 0.90, mIndert = 0.05, mDelete = 0.05

Implementation	No of Threads							
	1		2		4		8	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
Serial	0.053418	0.007078						
One mutex	0.053034	0.006115	0.077901	0.01262	0.079457	0.009774	0.081569	0.012634
Read Write lock	0.054478	0.007456	0.05113	0.00864	0.044636	0.006535	0.050457	0.010017

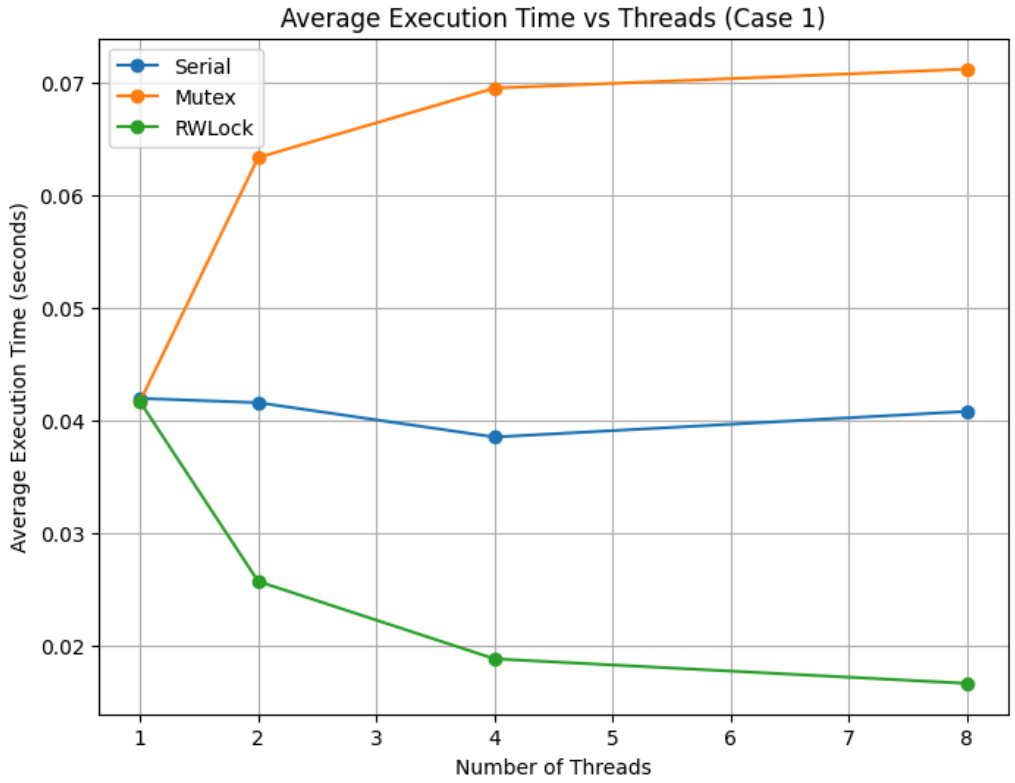
Case 3

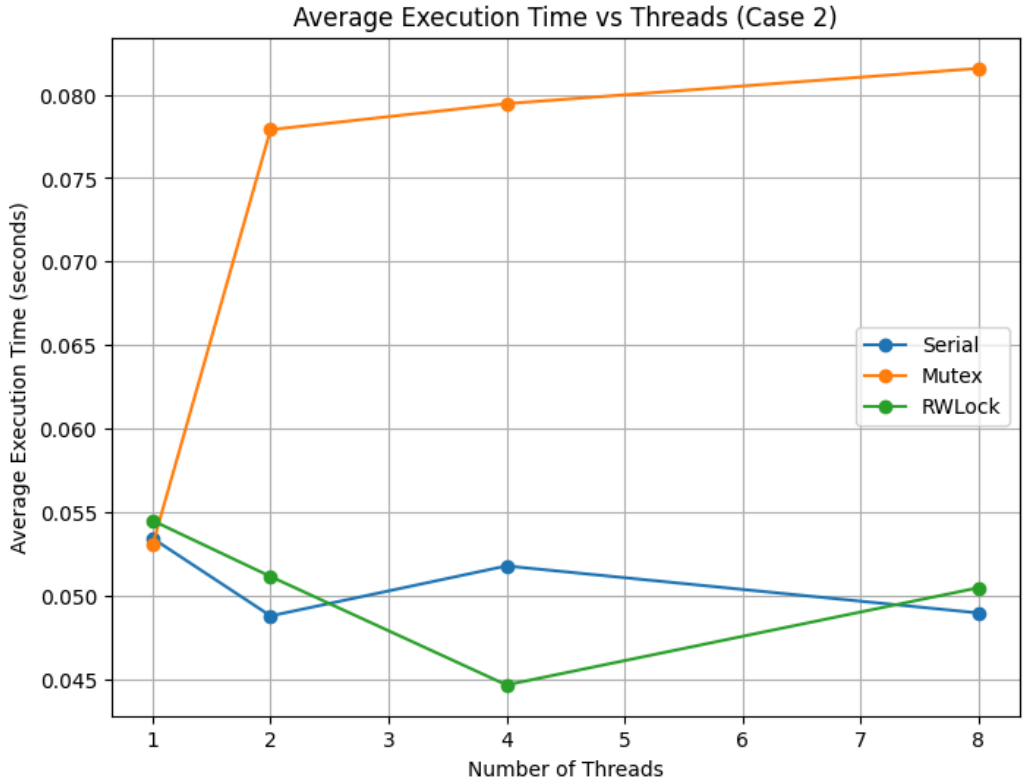
n = 1000, m = 10000, mMember = 0.50, mIndert = 0.25, mDelete = 0.25

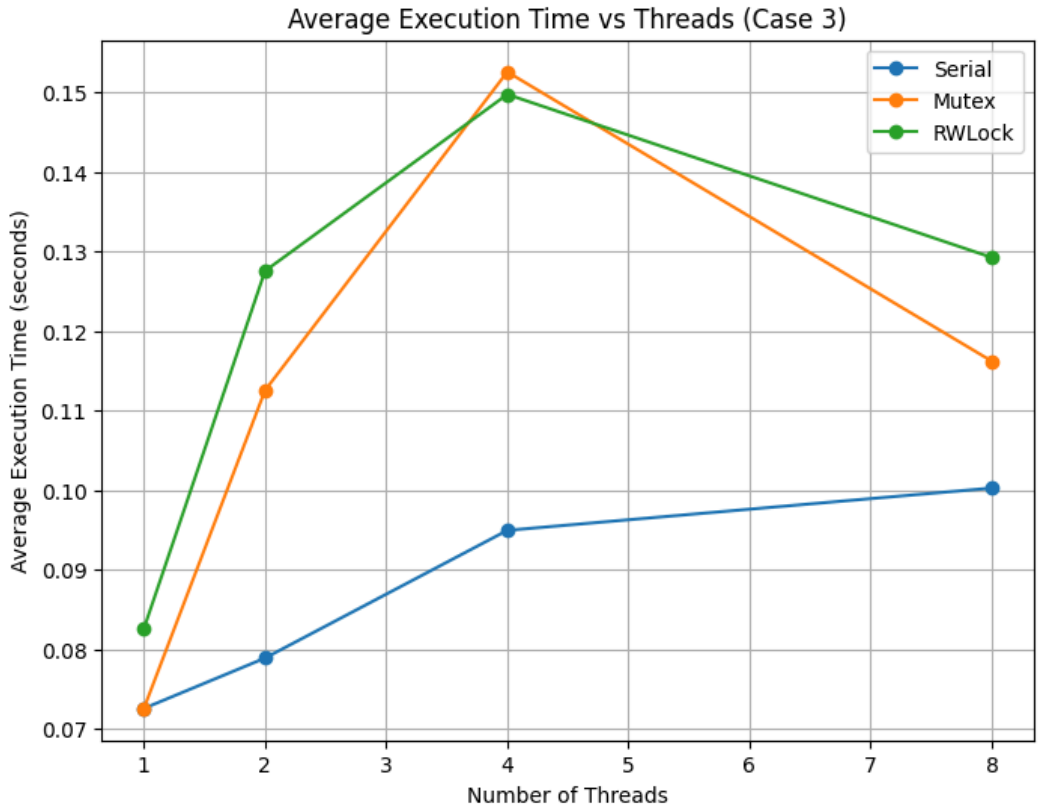
Implementation	No of Threads							
	1		2		4		8	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
Serial	0.072595	0.010607						
One mutex	0.072603	0.008715	0.112556	0.0248	0.152564	0.048895	0.116221	0.021932
Read Write lock	0.082631	0.01573	0.127556	0.023525	0.14974	0.027703	0.129249	0.027284

Step 4

Plot the average execution time against the number of threads for each case.







Step 5

Comment your observations while critically evaluating your findings. You may plot additional graphs to support your discussion.

Based on the initial graphs we can see that the average execution time for each method when the number of threads are 1 is almost the same, except for case 3, where the overhead associated with the read-write lock for write operations slightly elevates it. Taking a look at the times taken for the other two methods in comparison to the serial method, we can make the following observations.

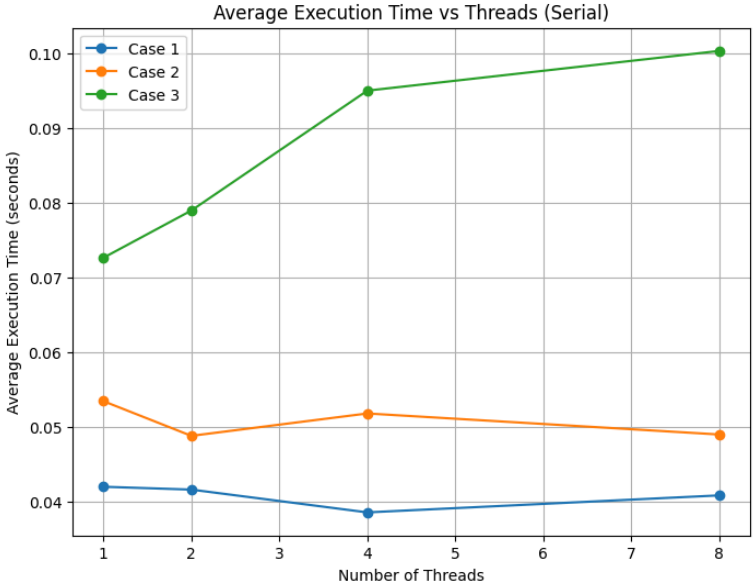
In case one, where the overwhelming majority of the operations are read operations, we can see that the time taken by the mutex method increases considerably before flattening with the number of threads. This is explained since there is no real gain for the mutex method since no two threads can actually work at the same time, taking at least as much time as the serial method just for the operations, which is further exacerbated by the overhead coming both with thread creation and possibly context switching as well as overheads for mutex acquisition times, and possibly even lower level factors such as memory contention and L1 caching in the cpu cores.

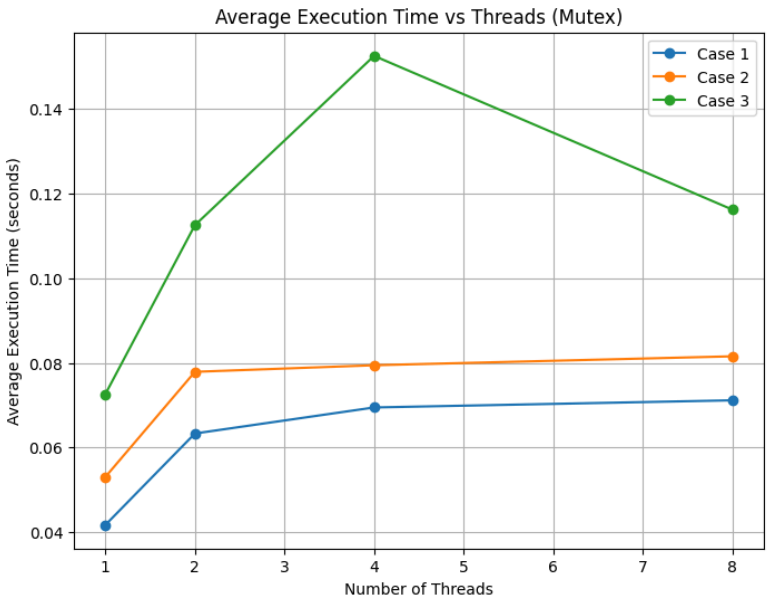
In contrast, we see the rw-lock method significantly decreasing in runtime with the number of threads before flattening out, which is explained by the parallelism which happens for read operations, since multiple threads can acquire the lock to read at the same time. While

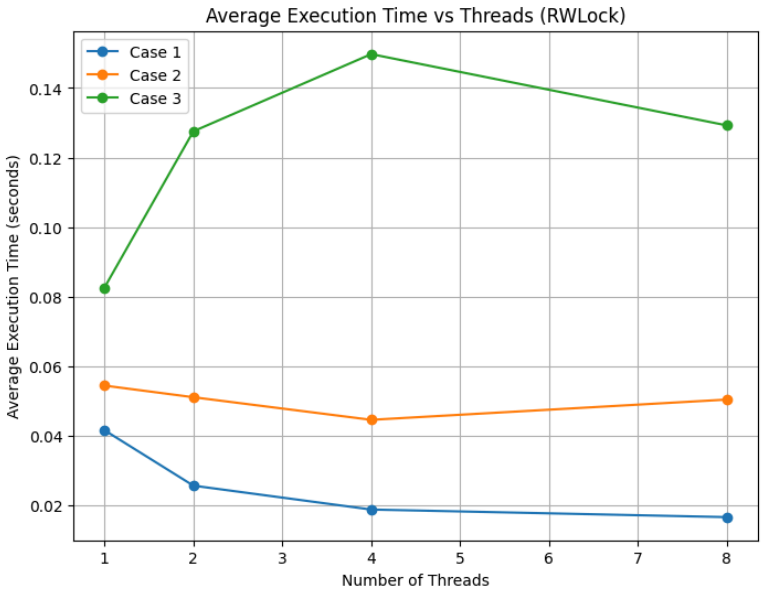
theoretically the performance gain should increase with the number of threads consistently, the flattening out is explained by the practical overheads of thread creation, thread context switching as well as lock acquisition times and again possibly even lower level factors such as memory contention and L1 caching in the cpu cores.

In cases two and three, we can see that the performance advantage that the rw-lock method had has reduced as the fraction of write operations increases. This is explained by the fact that only the read operations can be done in parallel by multiple threads at the same time, whereas the write lock can only be held by a single thread. In fact, we see that for most thread values, rw-lock method takes slightly more time than the mutex method for case 3, indicating a slightly higher overhead for write lock acquisitions under the rw-lock method. It may be necessary to verify the idea of a higher overhead by increasing the fraction of write operations even more (maybe to 0.9) (We haven't done this in this assignment).

In addition we can attribute the extra variation within the plots to system resource availability and other system related factors. For example, there is a slight and sometimes considerable variance for the time taken to run the serial operations which shouldn’t exist theoretically. It may be necessary to reaffirm this conjecture by creating a separate program which only runs one method at a time (serial, mutex or rw-lock) (Again, we haven't done this in this assignment).




Average Execution Time vs Threads (Mutex)
Case 1
Case 2
Case 3
Average Execution Time (seconds)
0.04
0.06
0.08
0.10
0.12
0.14
1
2
3
4
5
6
7
8
Number of Threads


Average Execution Time vs Threads (RWLock)
Case 1
Case 2
Case 3
Average Execution Time (seconds)
0.02
0.04
0.06
0.08
0.10
0.12
0.14
1
2
3
4
5
6
7
8
Number of Threads