Performance Measure of Synchronisation Constructs

ARPIT SINGH

111601031

COMPUTER SCIENCE AND ENGINEERING

System Specifications

System Specifications on which the performance of a program typically depends

Architecture	x86_64
CPU (s)	4
Model Name	Intel(R) Core(TM) i5-7200U CPU @2.50GHz
CPU max MHz	3100.0000
CPU min MHz	400.0000
Gcc version	7.4.0

Outline

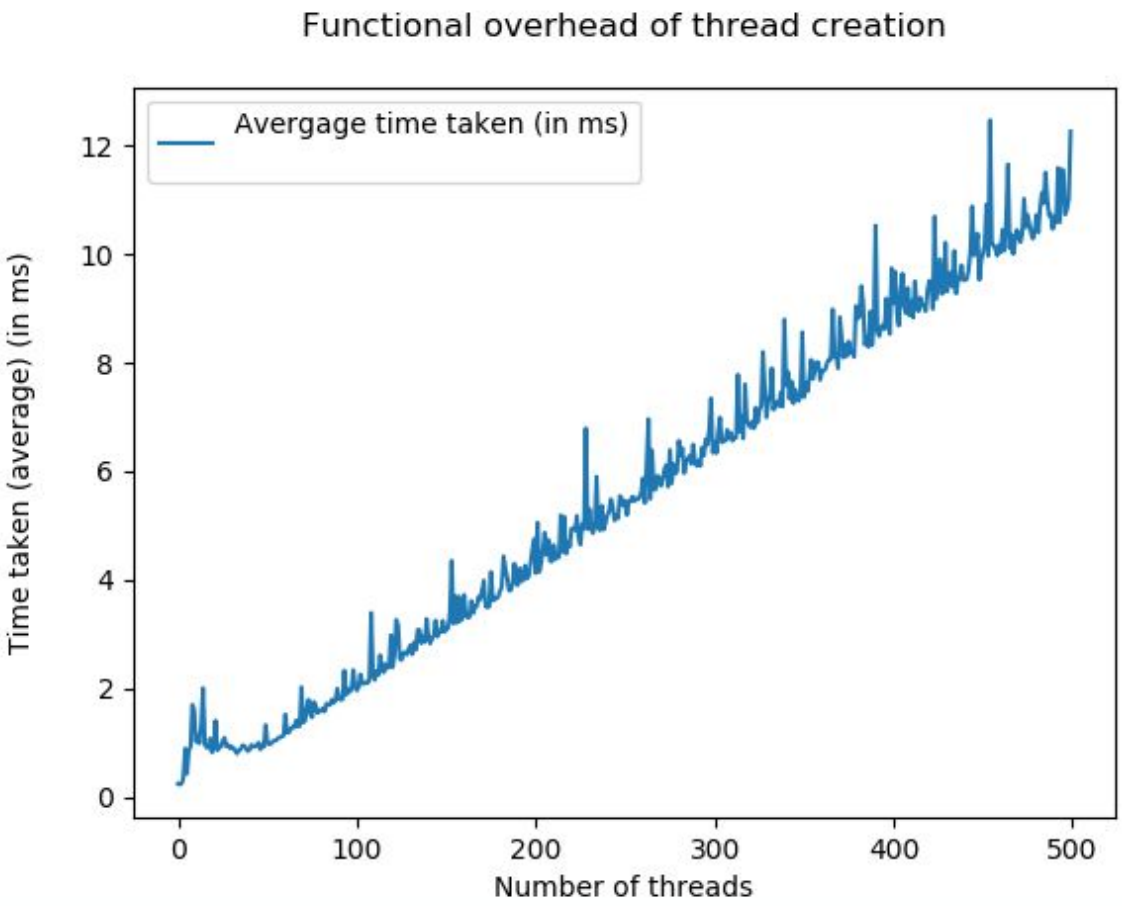
In order to measure the performance of synchronisation constructs, several programs are made and tested. So this report is going to contain the description of the programs, their results and the insights that can be drawn from those result.

Program 1 : pthread_overhead.c

This program measure the functional overhead of various pthread function and synchronisation constructs.

Pthread_create and pthread_join (for n threads)

In this we measure the time taken for pthread_create and pthread_join by spawning n threads which do not do any work, just return NULL. For measurement of time interval, start point is just before the pthread_create of all n threads and end point is just after the pthread_join of all n threads.



Insight

We can see the effect of context switching here. It may be noted that the average time taken for thread creation and termination keep on increasing as the number of threads increases because of context switching. As the number of threads spawned increases there will be increase in the context switch, so this additional overhead results in the increase of the average.

For synchronisation constructs

Readings (in nanoseconds)	Pthread_create and join	Mutex lock and unlock	Sem lock and unlock	Rwlock (rdlock and unlock)
Reading-1	33889	999	489	961

Reading 2	34874	1060	590	1185
Reading-3	34729	1101	720	1297
Reading-4	35197	1048	676	1066
Reading-5	38741	1070	822	1379
Minimum	33889	999	489	961
Maximum	3871	1101	822	1379
Average	35486	1055.6	659.4	1177.6

The program used to measure the overhead is written in c++ since using chronos give the time to the precision of nanoseconds.From the statistics above, it is clear that the functional overhead for creating and joining thread is largest (almost of the order of 100 microseconds). Functional overhead for Rwlock and mutex (lock and unlock) is almost the same (of the order of few microseconds).Functional overhead for semaphores is also of the order of microseconds and smaller as compared to the sem_lock and sem_unlock.

Since these statistics just give the functional overhead, so to measure the actual performance of synchronisation constructs we need programs that compare them.

Program 2: histogram

This program takes the n numbers and assign them to the BIN number of bins. We can adopt to approaches. First we can have an array of hist (to contain the counts) that can be shared between the threads or we can create local hist for each thread and then calculate the global hist from the local hist. In the first approach since we have only one global hist which is shared resulting in critical sections so we can use mutex or the busy wait. So the performance measure of both approaches are given below.

The performance of a program depends on the value of n and n_threads

N =100000000, n_threads = 1

Readings	Shared (mutex)	Shared (busywait)	Local
Reading 1	1888.128	1085.545	393.57500
Reading 2	1904.626	1086.346	357.944
Reading 3	1886.67	1074.364	344.795
Reading 4	1850.9170	1073.325	343.514
Reading 5	1913.604	1089.845	366.959
Minimum	1888.128	1073.325	343.514
Maximum	1913.604	1089.845	393.575
Average	1888.6938	1081.885	361.26

N =100000000, n_threads =2

Readings	Shared (mutex)	Shared (busywait)	Local
Reading 1	9242.606	8527.147	262.555
Reading 2	8863.15	8480.376	281.698
Reading 3	8194.878	8581.548	289.675
Reading 4	7886.391	8438.95	198.737
Reading 5	7732.3373	8811.069	176.323
Minimum	7732.3373	8438.95	176.323
Maximum	9242.606	8581.548	289.675
Average	8838.401	8567.818	241.7976

Time taken for n_threads = 2 is much more as compared to n_threads = 1 with the same value of n. This is because n_threads = 1, there is only one thread, so a lot of time is saved from critical section where in other case more than one thread is executed, so lot of time goes in waiting.

N =100000000, n_threads =3

Readings	Shared (mutex)	Shared (busywait)	Local
Reading 1	8171.204	9525.592	215.373
Reading 2	7836	8434.913	212.978
Reading 3	8835.589	8454.102	195.392
Reading 4	8835.916	8458.598	260.857
Reading 5	8586.441	8522.459	242.300
Minimum	7836	8434.913	195.392
Maximum	8835.913	9525.592	260.857
Average	8453.03	8571.818	225.38

N =100000000, n_threads =4

Readings	Shared (mutex)	Shared (busywait)	Local
Reading 1	7757.131	15418.411	237.76
Reading 2	9947.298	10197.557	198.138
Reading 3	10050.889	9897.61	196.309
Reading 4	9252.398	11057.591	229.191
Reading 5	9242.814	11609.107	210.54
Minimum	7757.131	9897.61	196.309

Maximum	10050.889	15418.411	237.76
Average	9250.106	11636.0552	214.3876

N =10000000, n_threads =2

Readings	Shared (mutex)	Shared (busywait)	Local
Reading 1	803.864	912.861	23.548
Reading 2	943.878	884.102	23.324
Reading 3	831.915	869.095	25.0730
Reading 4	758.766	857.123	27.7111
Reading 5	946.8900	872.251	22.823
Minimum	758.766	857.123	22.823
Maximum	946.89	912.861	27.711
Average	857.0626	879.0864	24.4958

N =10000000, n_threads =3

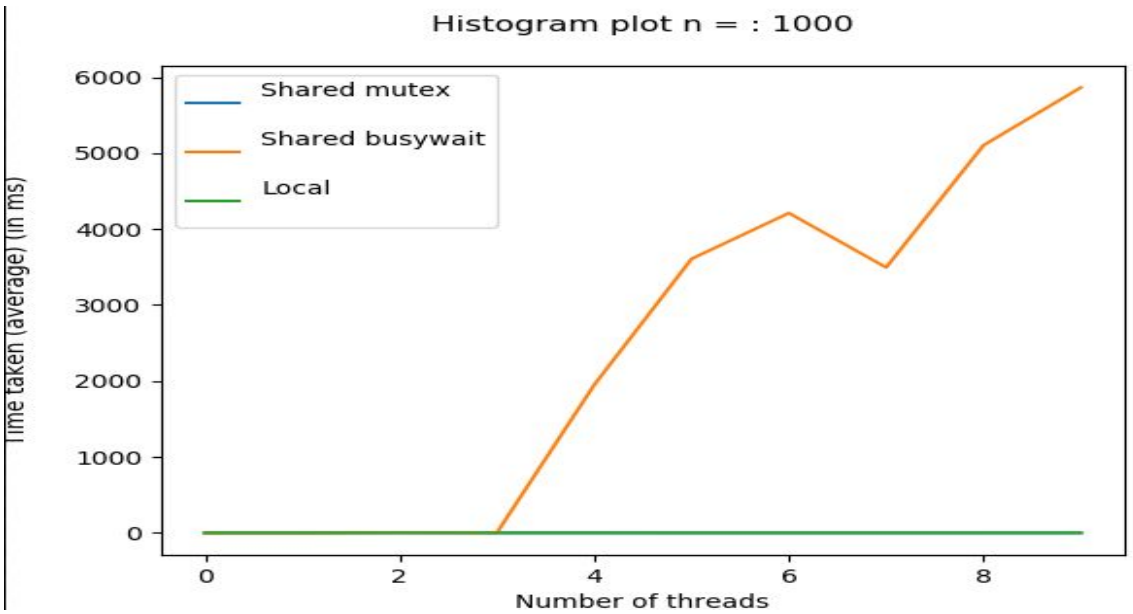
Readings	Shared (mutex)	Shared (busywait)	Local
Reading 1	682.656	790.447	26.223
Reading 2	842.87	796.7900	25.142
Reading 3	853.933	794.933	21.373
Reading 4	712.344	811.493	25.181
Reading 5	704.881	808.883	21.2900
Minimum	682.656	790.447	21.2900
Maximum	853.933	811.493	25.181
Average	759.3368	800.5092	23.8418

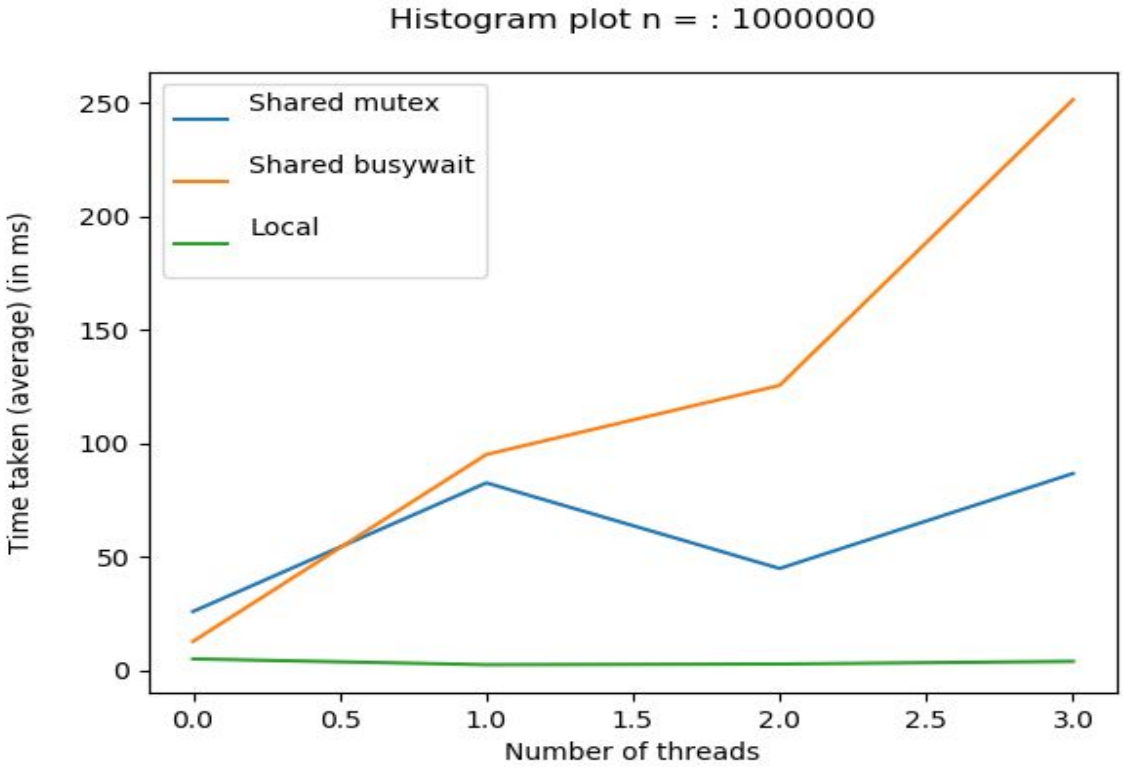
N =10000000, n_threads =4

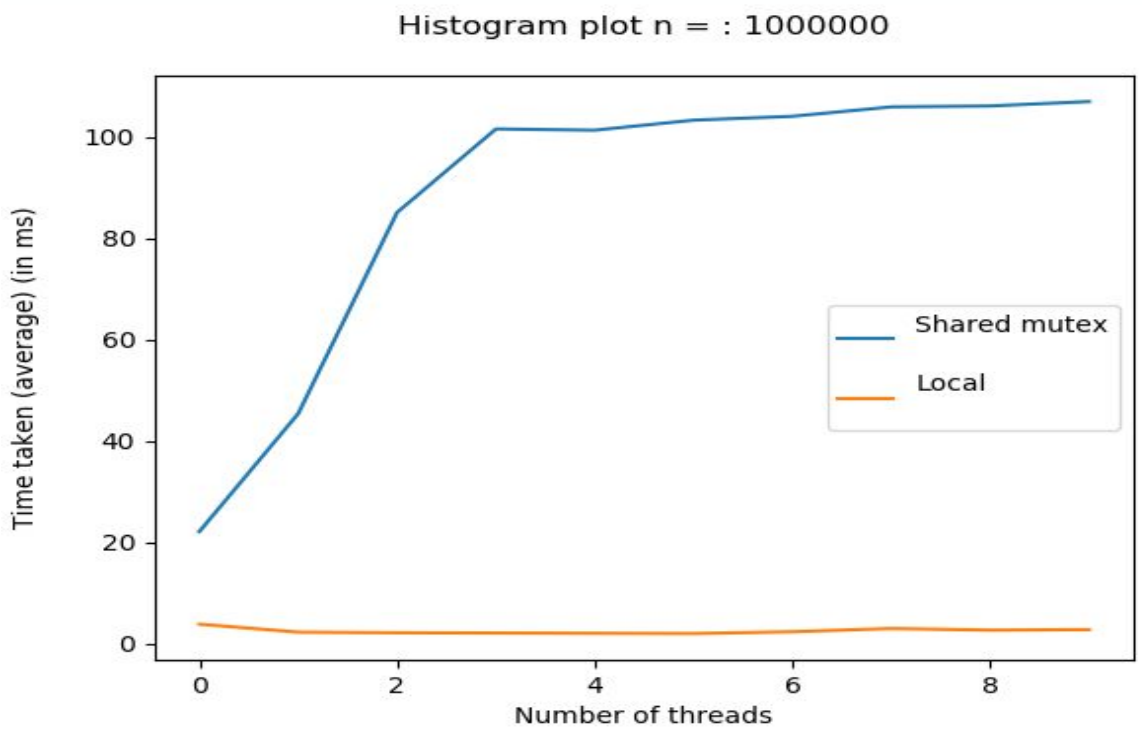
Readings	Shared (mutex)	Shared (busywait)	Local
Reading 1	930.88	1211.563	19.81
Reading 2	998.563	975.563	20.991
Reading 3	856.967	863.315	23.061
Reading 4	878.0311	875.726	22.683
Reading 5	883.175	848.177	20.984
Minimum	856.967	848.177	19.81
Maximum	930.88	1211.563	23.061
Average	909.523	954.8688	21.5058

N =10000000, n_threads =1

Readings	Shared (mutex)	Shared (busywait)	Local
Reading 1	186.603	109.523	36.409
Reading 2	187.603	109.092	36.568
Reading 3	193.635	108.494	36.100
Reading 4	186.784	108.882	35.912
Reading 5	186.753	108.451	35.896
Minimum	186.603	108.451	35.896
Maximum	193.635	109.523	36..568
Average	188.2756	108.884	36.177


Histogram plot n = : 1000
Shared mutex
Shared busywait
Local
Time taken (average) (in ms)
Number of threads
0
1000
2000
3000
4000
5000
6000
0
2
4
6
8


Histogram plot n = : 1000000
Shared mutex
Shared busywait
Local
Time taken (average) (in ms)
Number of threads
0
50
100
150
200
250
0.0
0.5
1.0
1.5
2.0
2.5
3.0



From the first plot it is clear that the shared busywait takes much more time as compared to shared mutex or having the local variable. One thing to note is that, **when the number of threads increases beyond the number of cores then the time for busy wait shoots very high**. This is because when number of threads is less than the number of cores than there is one core available for each thread but when more threads are included then only some of the threads are on the core, remaining are scheduled later. So, a thread run for some amount of time and then other threads are scheduled. It may happen that when the thread is scheduled then it is in busy wait, but as soon as other threads are scheduled then this thread is not in busy wait, but since it is not scheduled so it will wait for longer time until it turns come. Moreover, busywait is **computation extensive** i.e. even when the thread is in busy wait then it is using cpu resources and other threads which actually require to do some work are not scheduled.All these factors account for large computation time when the number of threads is more than the number of cores.

Having a local variable is much less computationally expensive as compared

to having a shared variable.Since having a shared variable results in critical section and the execution of critical section is essentially serial. As the number of

threads increase then the time taken is increasing instead of decreasing. The reason for this is that majority of the code in case of shared variable is in critical section (only one thread can execute that piece of code), so there is large overhead in calling synchronisation constructs and waiting, resulting in increased time as the number of threads increase.

Program 2: Barrier

In this program, barriers are implemented using mutexes , semaphores and condition variables. Then each thread call the barrier multiple times to see the performance.

Threads = 1

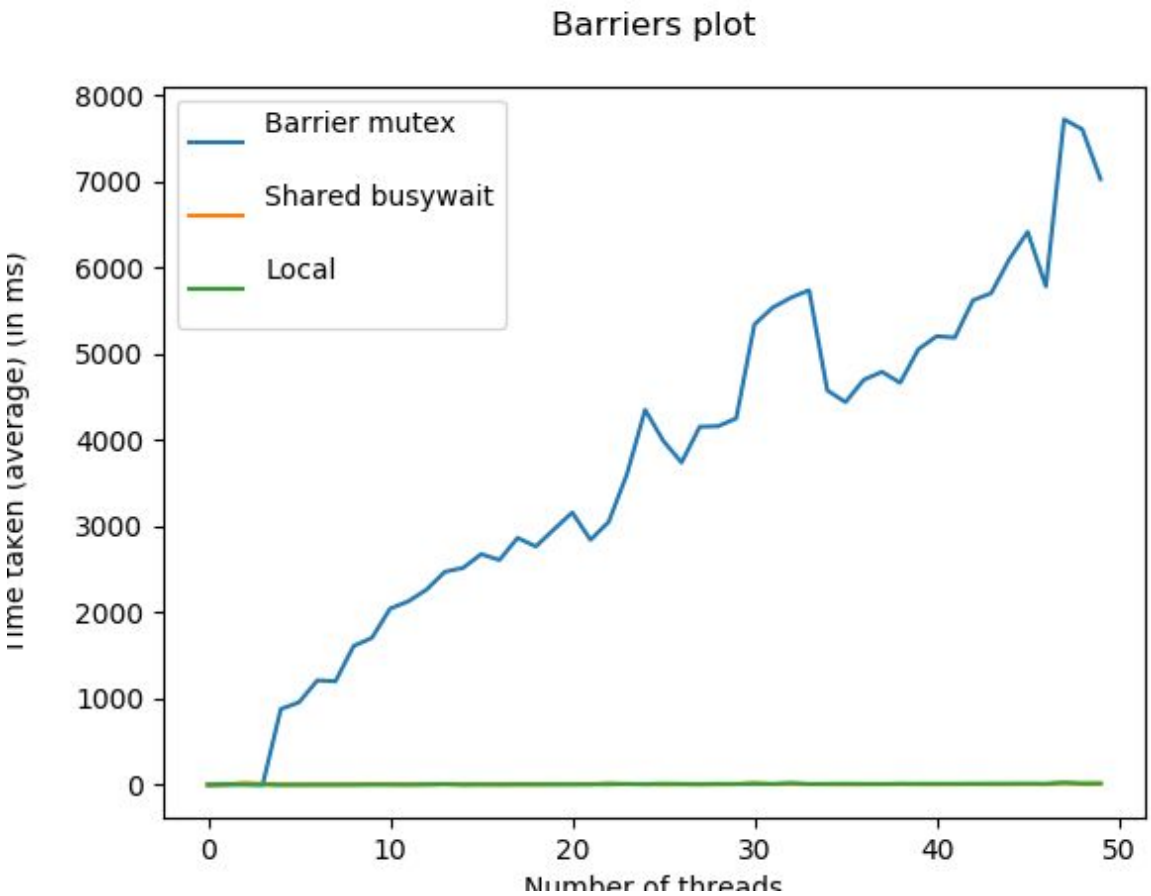
	Mutex barrier	Semaphore barrier	Barrier conditional
Min	0.097	0.141	0.148
Max	0.191	0.176	0.206
Average	159.3	0.147	0.178

Threads = 5

	Mutex barrier	Semaphore barrier	Barrier conditional
Min	1050.425	0.682	1.077
Max	1178.33	0.684	1.224
Average	1108.802	0.68366	1.153

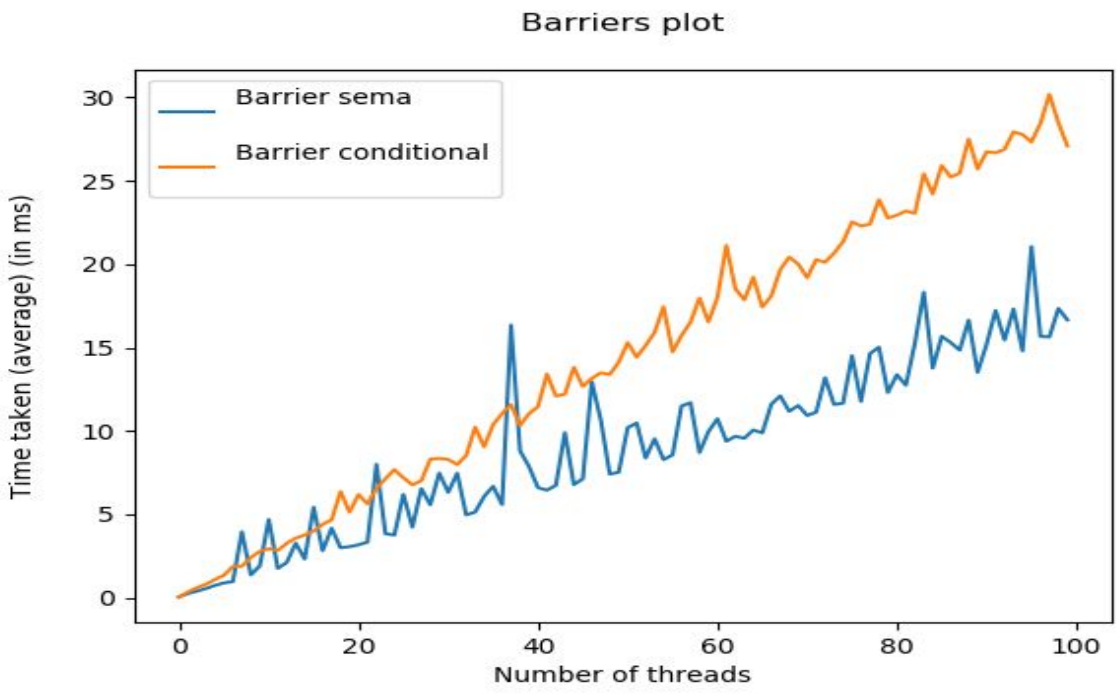
Threads = 10

	Mutex barrier	Semaphore barrier	Barrier conditional
Min	1684.54	1.440	2.319
Max	1800.4941	1.481	2.398
Average	1745.903	1.466	2.351



When we are implementing barrier using busy wait, it performs comparable to other two implementations (barrier using semaphores and barrier using conditional variables), until the number of threads is less than the number of cores. Because each thread is assigned to one of the core and all the threads reach the barrier without much significant delay as there is not much wait. But as the number of threads increase more than the number of cores, then because of context switching (thread that can be close to the barrier may be scheduled out resulting in large delay) and busy wait involved the computation time takes a

large jump. As the number of threads increase, the time increases because of increased contention as more number of threads have to reach at the barrier.



Barrier using the sema and barrier using the conditional variables perform much much better as compared to the implementation using mutex and busy wait.

However, these two implementations perform almost the same,however, barrier semaphore having slightly better performance due to low functional overhead of semaphore post and wait.

The performance of barriers using semaphores and performance of barriers using conditional variables is almost the same, however barriers using conditional variables is better because using the conditional variable, we can implement barrier multiple times in the program. However in case of semaphores, barrier_sem need to be reset after each barrier.

Program 4 : Pi calculation

In this program, value of pi is calculated using the formula

$$\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

As the value of n increases, the value of pi becomes more and more accurate.In the parallel version, different threads are computing the value for different portion of the sum. Each thread is assigned n/p portion of the series, where n is the number of elements considered in the series, while p is the number of threads.The four approaches used are having a local sum that adds up to the global sum in which this operation is protected using busy wait and mutex.The other two approaches are adding directly to the global sum and protecting this using busy wait and mutex.

Thread = 4 , n = 10000

	Mutex local time	Mutex global time	Busy local time	Busy Global time
Min	0.069	0.132	0.069	0.663
Max	0.115	0.151	0.09	4.389
Average	0.0887	0.143	0.081	2.242

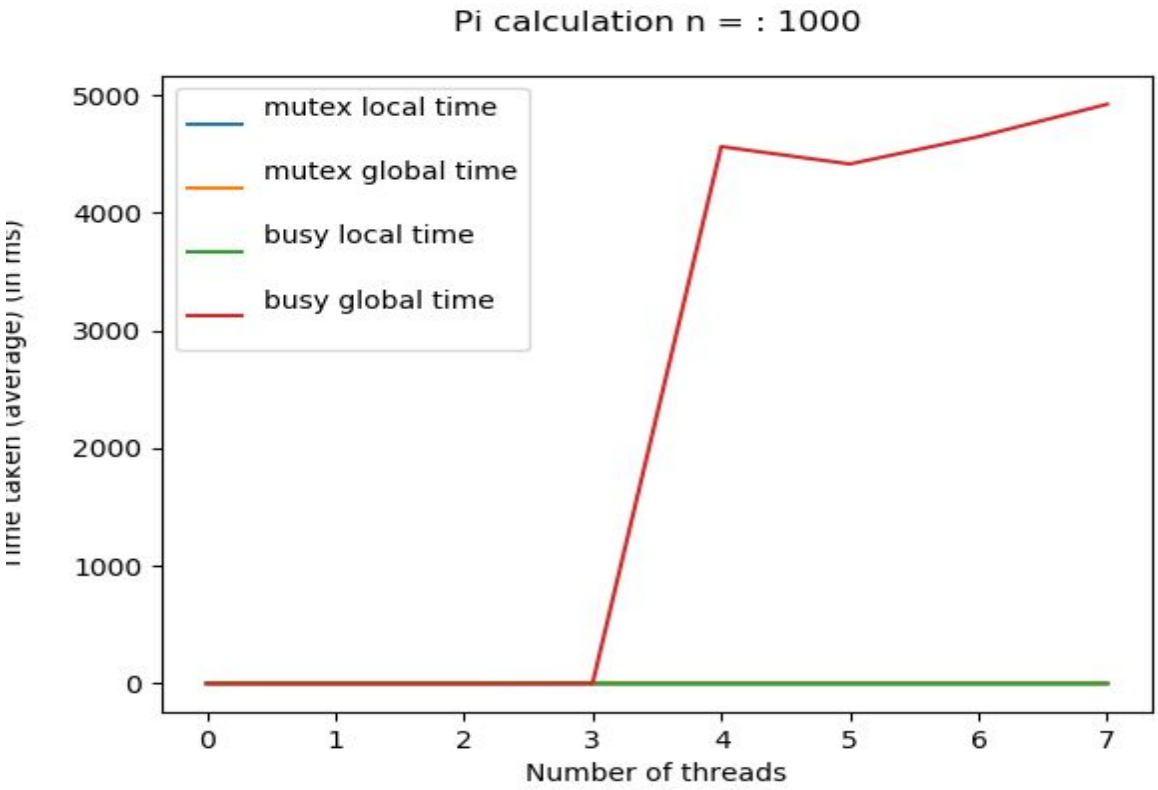
Thread = 5 , n = 10000 (Motive is to show too much time taken by busy global)

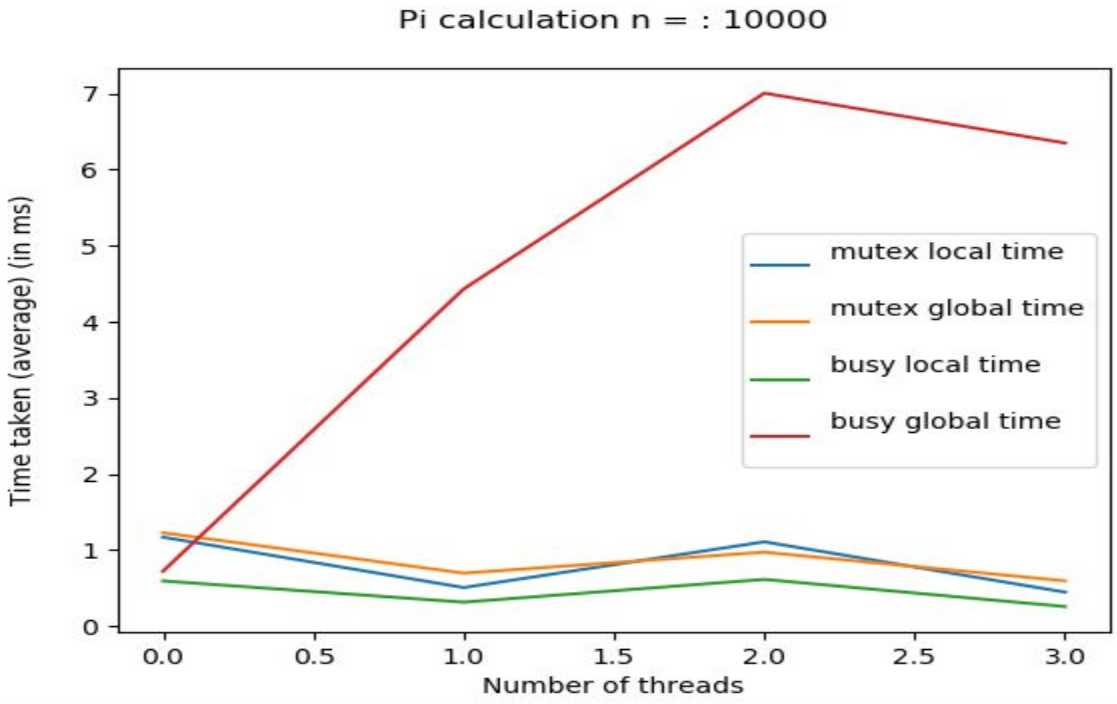
	Mutex local time	Mutex global time	Busy local time	Busy Global time
One reading	0.093	0.159	0.1110	45710.375

Thread = 5 , n = 10000000

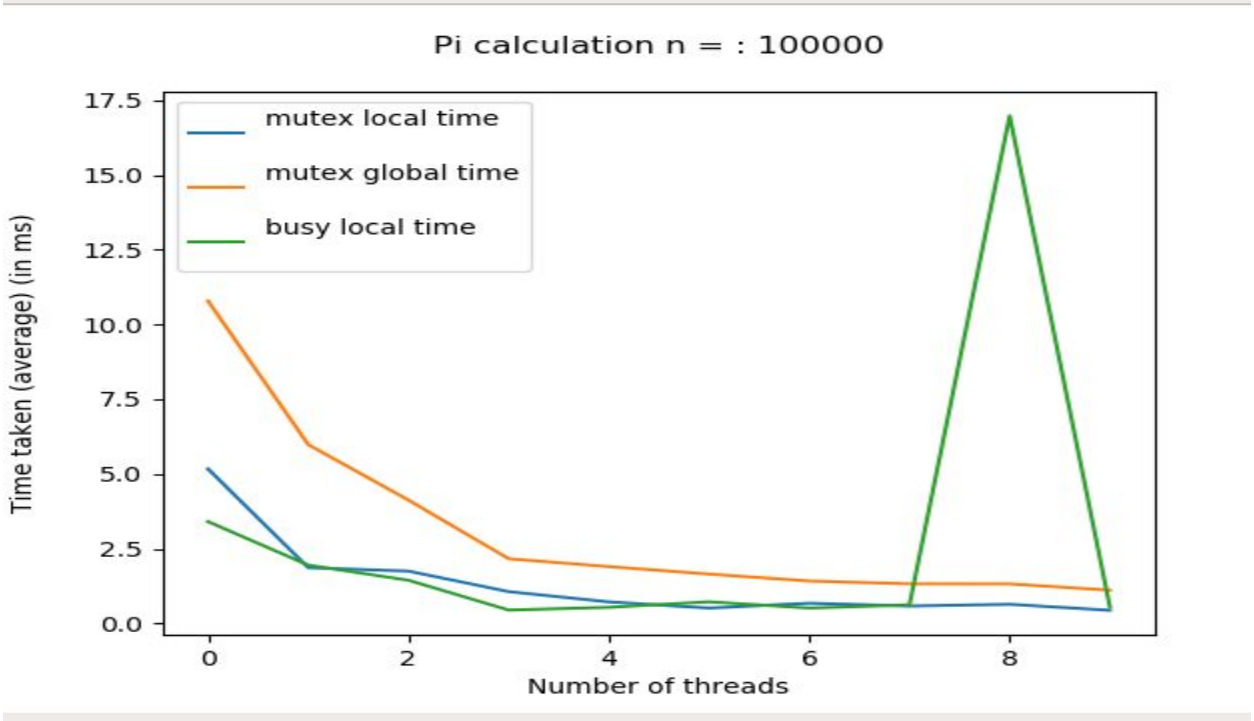
	Mutex local time	Mutex global time	Busy local time
Min	22.841	102.407	22.042
Max	23.506	117.158	26.783
Average	23.2103	112.11	24.616

The plots below show the results :

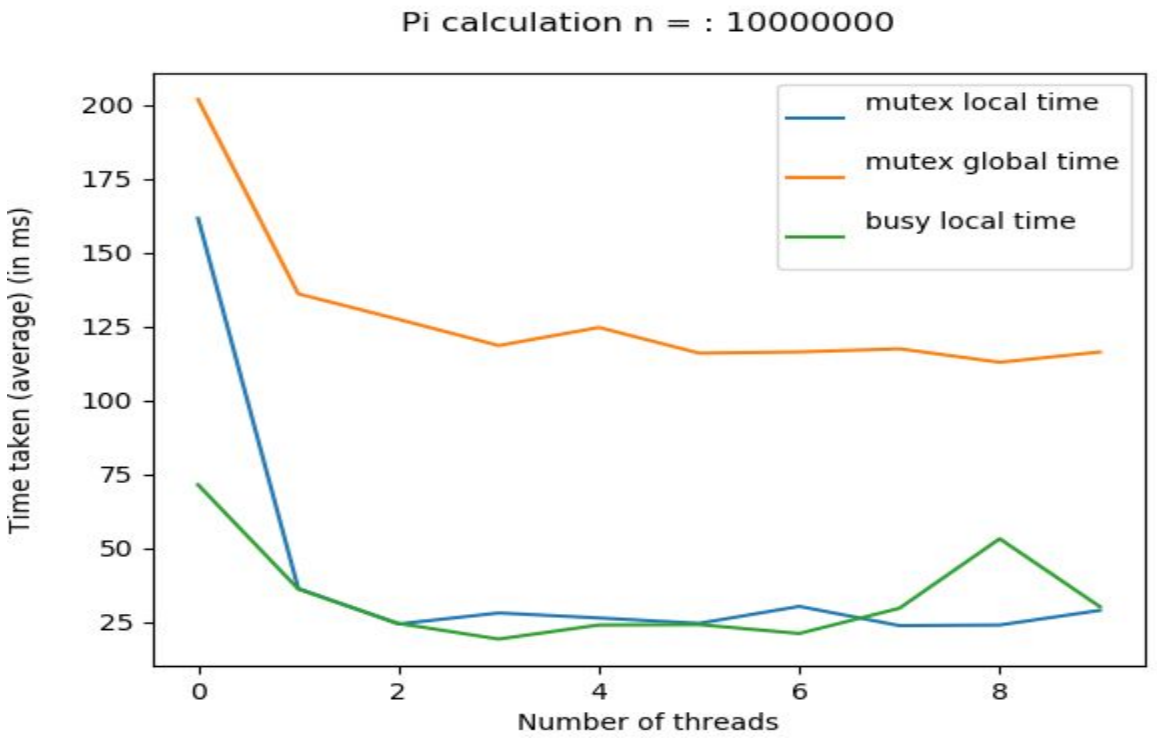


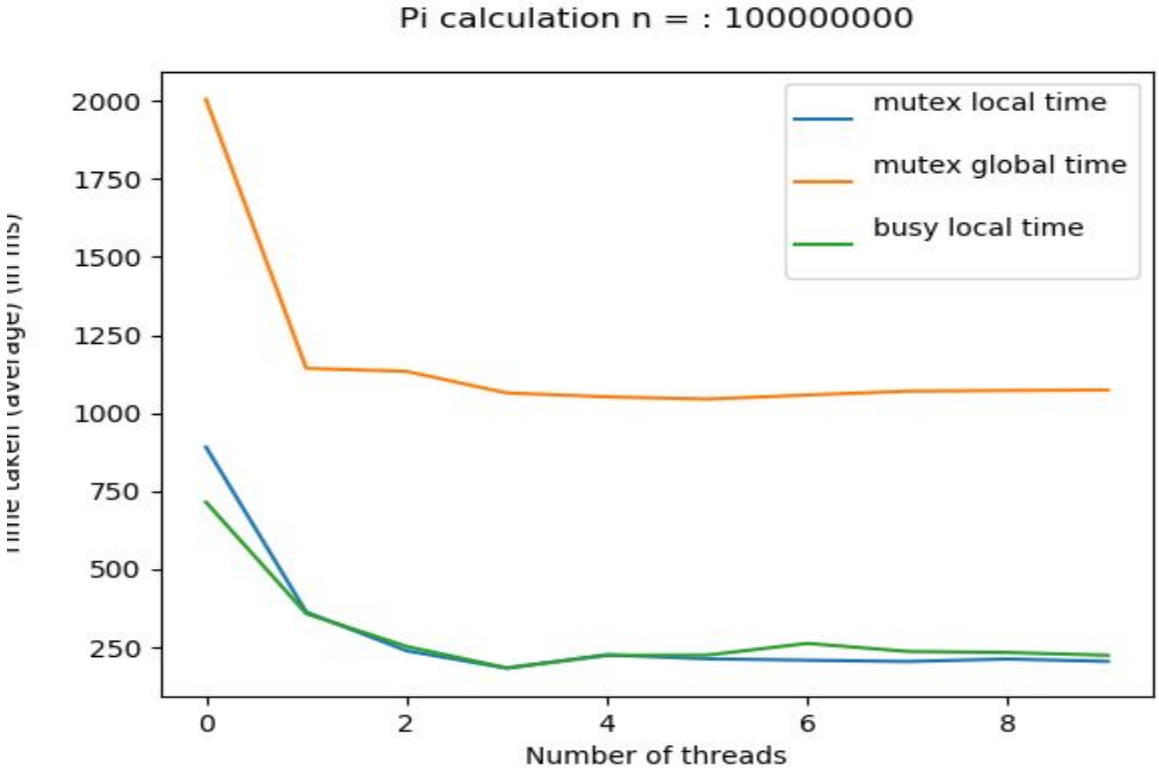


In both the cases when n = 1000 and n = 10000, computation time in having only the global sum and using only busy wait is very high. This makes sense because having only the global sum, drastically increases the critical section area and protecting them using busy wait result in lot of overhead.



Unexpected spikes in case of busy wait with local variable is because of context switching overhead when the number of threads is significantly large..


Pi calculation n = : 10000000
mutex local time
mutex global time
busy local time
Time taken (average) (in ms)
200
175
150
125
100
75
50
25
0
2
4
6
8
Number of threads


Pi calculation n = : 100000000
mutex local time
mutex global time
busy local time
2000
1750
1500
1250
1000
750
500
250
0
2
4
6
8
Number of threads

Curve becomes almost constant as the number of threads increases.It is clear from the above plots that as the number of threads increase then the computation time decreases due to parallelisation. So the **speedup** and **efficiency** increases as the number of threads increase. But after a certain limit, the time taken is almost the same, do not decrease with the increase in the number of threads.So speedup becomes almost constant while efficiency decreases after having the large number of threads.

In this case, having a local sum variable and protecting it using mutex and busy wait takes almost the same time. This is because the critical section is relatively very small in both the cases, mutex lock and unlock have functional overhead, so the time in case of both the mutex and busy wait is almost the same.

Program 5 : Link list

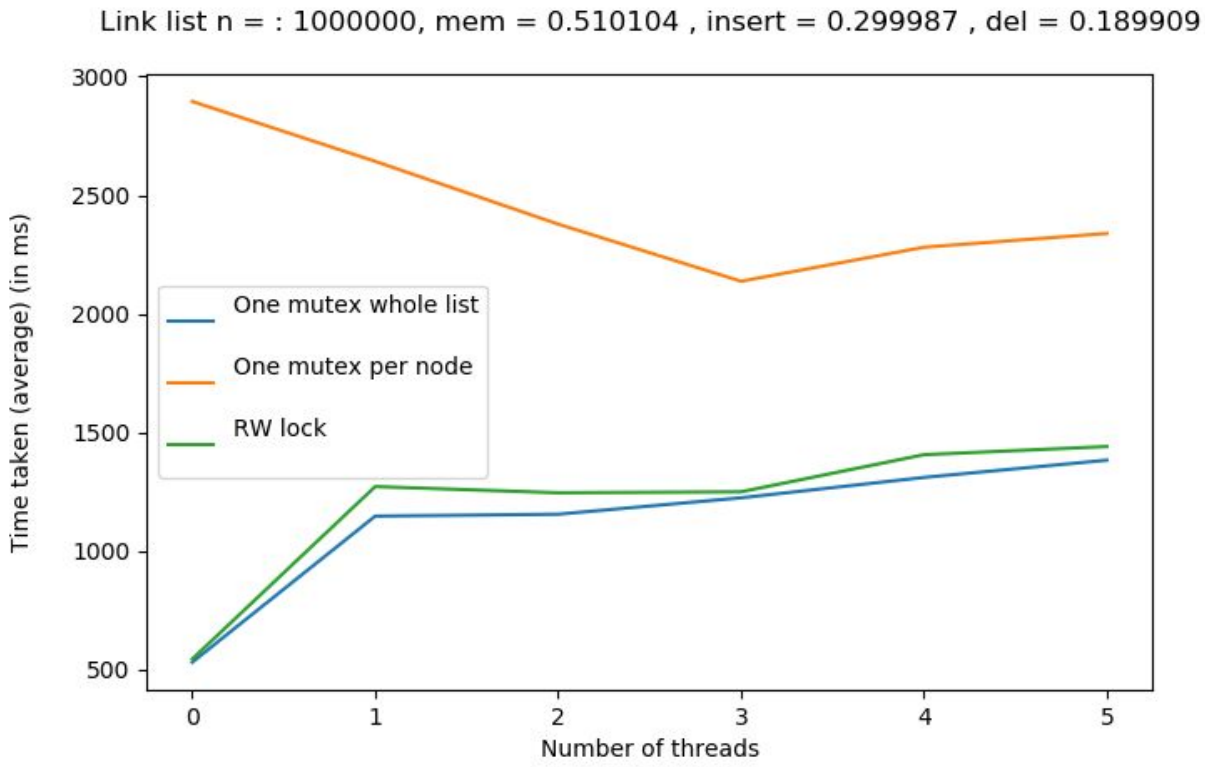
In this program, a link list (having elements in increasing order) is shared across multiple threads.Since the link list is shared it can be protected using **mutex on whole list**, **mutex on each node** and having a **read/write lock on the whole list**. In case of read/write locks, multiple find operations (isMember) can be done in parallel while write operation (delete and insert) must be exclusive. In case of one mutex on whole list, all the operations are exclusive.

Thread = 1 , n = 10000000, mem = 90% , insert = 5% , del = 5%

	Mutex one whole list	Mutex per node	Rwlock on whole list
Min	454.104	2549.155	466.241
Max	459.052	2645.344	471.958
Average	456.484	2590.218	469.5033

Thread = 5 , n = 10000000, mem = 90% , insert = 5% , del = 5%

	Mutex one whole list	Mutex per node	Rwlock on whole list
Min	1036.374	2051.356	572.367
Max	1132.605	2094.149	661.936
Average	1133.271	2077.2066	607.153



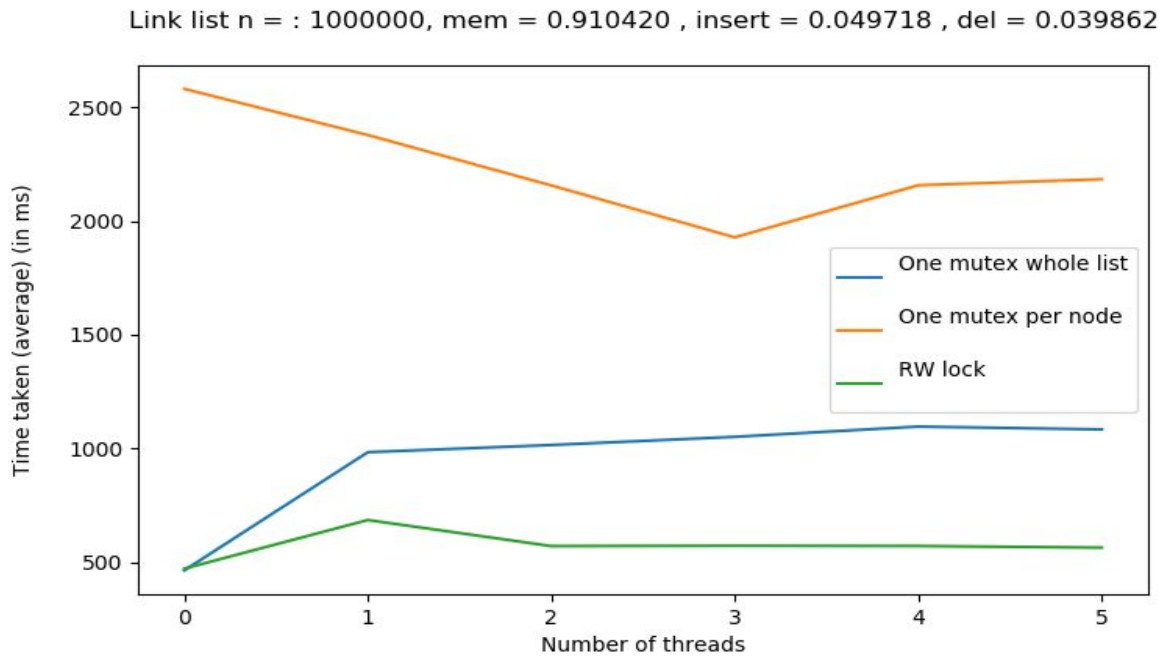
Here the number of operations done are 10^6, in which almost 50% operations are lookup operations, 30% are insert operations and remaining 20% are delete operations with 500 keys.

Having **one mutex per node performs very badly**, because of the functional overhead of mutexes. Since each operation involves multiple lock and unlock operations, so there is very large overhead of mutex lock and unlock functional call in addition to the wait. So that's why one mutex per node performs very badly.

The inferiority of the implementation that uses one mutex per node persists when we use multiple threads. There is far too much overhead associated with all the locking and unlocking to make this implementation competitive with the other two implementations.

In case of one thread, time taken is significantly very small because there is no contention in accessing the list because of other threads.

Since the number of read operations (lookup) is almost the same as number of insert and delete operations so having one mutex on whole list and having one read/write lock on whole list is almost the same.



When the number of lookup operations is very large as compared to insert and delete operations, as in this case, read/write lock performs much better as compared to mutex implementation because the read/write lock allows for the parallel read access (lookup operation is a read operation). On the other hand, if there are a relatively large number of Inserts and Deletes (for example, 10% each), there's very little difference between the performance of the read-write lock implementation and the single-mutex implementation. Thus, **for linked list operations, read-write locks can provide a considerable increase in performance, but only if the number of Inserts and Deletes is quite small.**

Note, when number of thread = 1, then single-mutex implementation and read/write lock implementation takes very less time. But after that, as the number of threads increase the time taken for both the implementation is almost the same. This is because, as the number of threads increase , **contention increase, but number of operations/per thread decrease , so the effective time for computation remains almost the same.**

Conclusion

There are various synchronisation constructs and there performance depends on various factors including problem type, number of threads, number of cores in the system, system on which the code is run etc. So this reported presenter the broad overview of the general synchronisation constructs the kind of problem they are suited for and the problem associated with it.