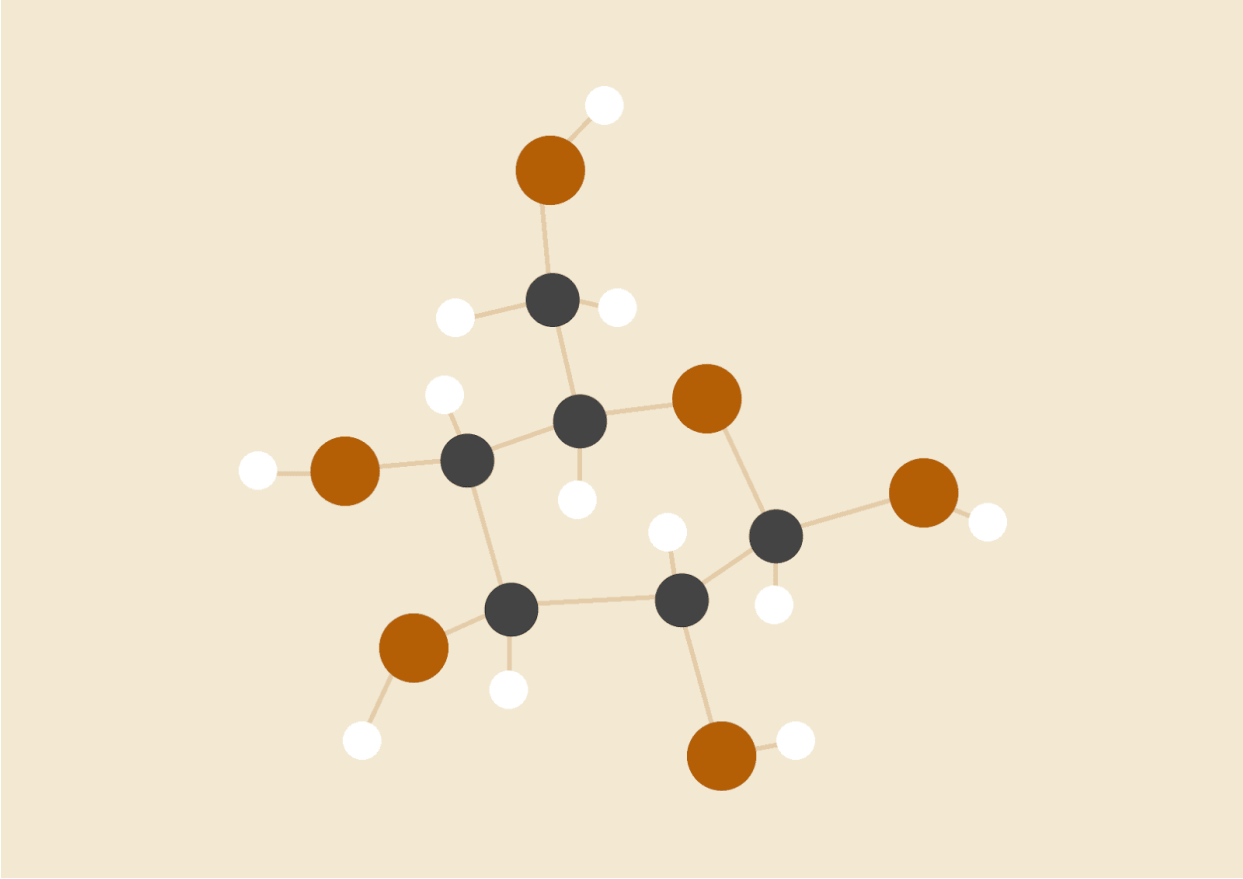
PERFORMANCE PROFILE REPORT



Amit Vikram Singh

111601001

23.02.2019

6th Sem, B.Tech

GLOBAL SUM:

Problem: compute sum of i where $1 \leq i \leq n$.

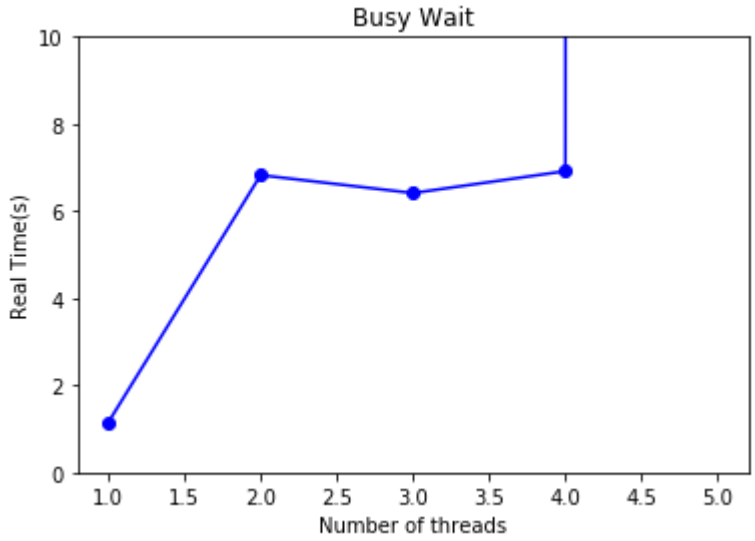
METHOD 1: Each thread gets a part of data and increments global sum in thread safe way.

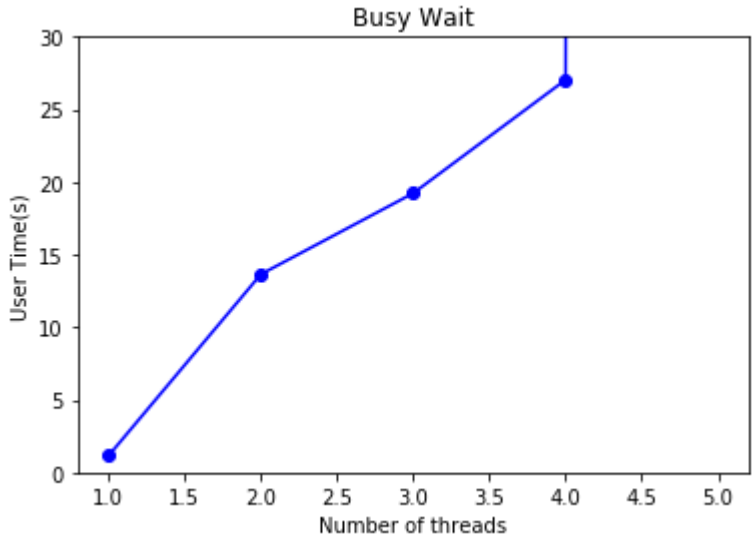
BUSY WAIT

Procedure: **Create k number of threads. Create a shared variable g_sum which will store the final sum. Data is divided equally among threads. Each thread proceeds in following way:**

- ITERATE OVER DATA
    - Lock g_sum by applying busy_wait of flag variable.
    - Increment g_sum.
    - Change flag variable so that another thread is executed next.

Number of Threads	Number of Calculation per Thread	Average Real Time Taken	Minimum Real Time Taken	Maximum Real Time Taken	Average User mode combined Time Taken among all CPUs
1	1e8	1.147	1.137	1.165	1.152
2	5e7	6.824	6.476	7.12	13.646
3	3.33333333e8	6.413	6.055	6.631	19.230
4	2.5e7	6.916	6.0452	8.175	27.039
5	2e7	More Than an hr	More than an hr	More than an hr	More than an hr



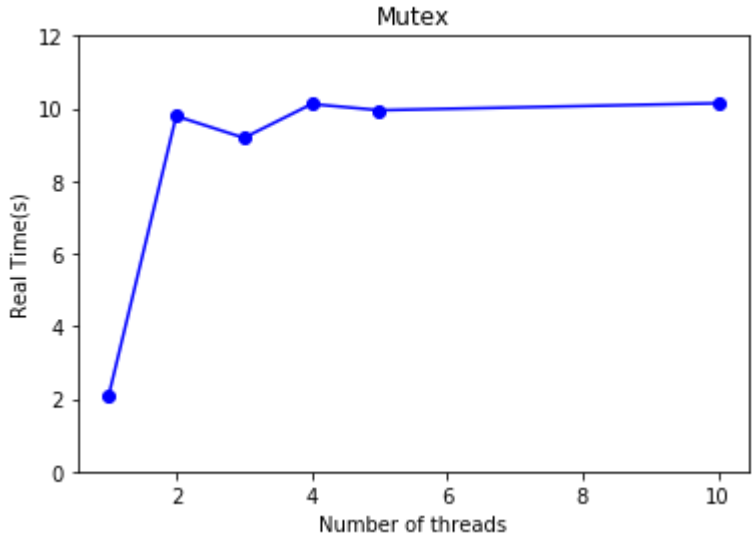


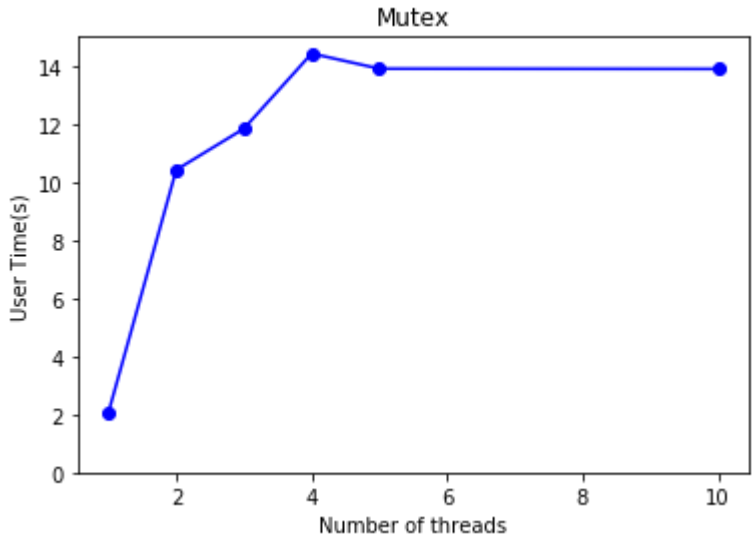
MUTEX

Procedure: **Create k number of threads. Create a shared variable g_sum which will store the final sum. There is a mutex g_sumMtx to ensure thread safe access of g_sum. Data is divided equally among threads. Each thread proceeds in following way:**

- ITERATE OVER DATA
    - Lock mutex.
    - Increment g_sum.
    - Unlock mutex.

Number of Threads	Number of Calculation per Thread	Average Real Time Taken	Maximum Real Time Taken	Minimum Real Time Taken	Average User mode combined Time Taken among all CPUs
1	1e8	2.068	2.13716	1.9801	1.152
2	5e7	9.789	13.707	7.821	10.417
3	3.33333333e8	9.189	9.878	7.434	11.839
4	2.5e7	10.120	10.837	8.754	14.425
5	2e7	9.950	10.703	9.012	13.9015





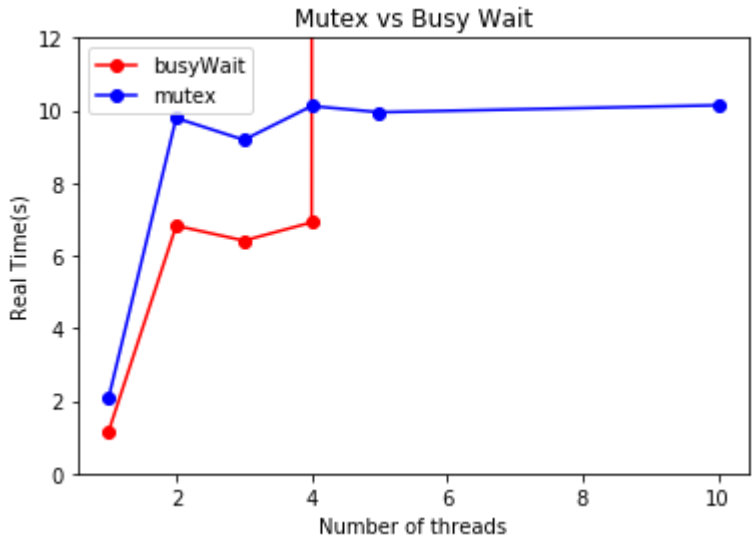
OBSERVATION:

1. **Number of threads and busy_wait:** Best performance is gained when single thread is used. Using more than 1 thread takes roughly 6 times more time compared to single thread. While using only one thread busy_wait is not effective at all because flag variable is always equal to myRank of thread hence busy_wait loop breaks immediately. Hence there is no overhead because of busy_wait when using single thread. When using more than one thread, thread execute in order and after incrementing g_sum once thread has to wait until all other threads increment g_sum once. Busy_wait is CPU extensive. This can be seen by comparing the average real time taken and combined time taken by all cpu. Since thread execute in order so at one time only one thread is doing the computation So user time should roughly be equal to real time taken. But even though threads are not doing computation, most of the time they might be looping in busy_wait and hence combine time over all cpu is roughly equal to number of threads times real time taken. When number of thread(=5) is more than number of cores(=4) it takes more than an hour to complete the computation. This is so because at most 4 threads can be running at at time. Hence at least two thread will execute sequentially(pigeon hole principle). Note that this sequential execution is completely different from thread executing in order. When thread are running on different cores, as a thread goes in busy_wait next thread running on different core immediately comes out of busy_wait. While in sequential execution when one thread goes in busy_wait until this particular thread is not blocked other thread can not run.

2. Number of threads and Mutex: Best performance is gained when single thread is

used. Using more than 1 thread roughly takes 7-8 times compared to single thread. While using only one thread thread always succeeds in locking the mutex. But when using more than one thread, a thread might be blocked while locking the mutex. Now while a thread is blocked OS has to give some to do backup and scheduling which might take few hundred CPU cycles. Thread can not be unblocked until other thread unlocks the mutex. This is the reason why using more than one thread takes more time compared to single thread. However mutex is not CPU extensive. As depicted from data real time taken is roughly equal to the combined time taken by all CPUs because when a thread is not doing computation it's blocked unlike busy_wait which keeps looping.

3. Busy_wait vs mutex: As discussed busy_wait is CPU extensive while mutex is not. When using number of thread less than number of cores mutex takes more time(real time) compared to busy_wait. This is because thread lock and unlock is expensive than other instructions. When thread successfully locks the mutex then change of flag is done using few instructions but when thread gets blocked OS has to do backup and scheduling which might take few hundreds CPU cycles. When using number of thread more than number of cores then busy_wait performance decreases manifolds while mutex performance doesn't change much.



METHOD 2: Each thread gets a part of data and then calculate mySum. After calculating mySum, thread increments g_sum by mySum in thread safe way.

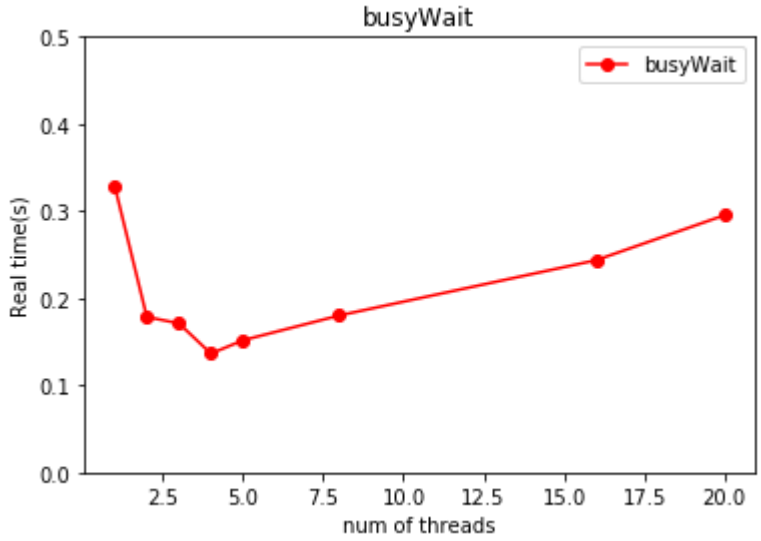
BUSY_WAIT

Procedure: **Create k number of threads. Create a shared variable g_sum which will store the final sum. Data is divided equally among threads. Each thread also contains a local variable mySum in which it stores the sum of data assigned to that thread. Each thread proceeds in following way:**

- ITERATE OVER DATA
    - Increment mySum.
- Lock g_sum by applying busy_wait of flag variable.
- Increment g_sum by mySum.
- Change flag variable so that another thread is executed next.

Number of Threads	Number of Calculation per Thread(s)	Average Real Time Taken(s)	Minimum Real Time Taken(s)	Maximum Real Time Taken(s)	Average User mode combined Time Taken among all CPUs(s)
1	1e8	0.328	0.3054	0.334	0.331
2	5e7	0.178	0.1599	0.1835	0.358
3	3.33333333e	0.171	0.159	0.181	0.477

	8				
4	2.5e7	0.136	0.123	0.149	0.490
5	2e7	0.151	0.146	0.161	0.4874
8	1.25e7	0.180	0.1542	0.1997	0.665
16	6.25e6	0.243	0.197	0.282	0.915
20	5e6	0.295	0.229	0.367	1.115



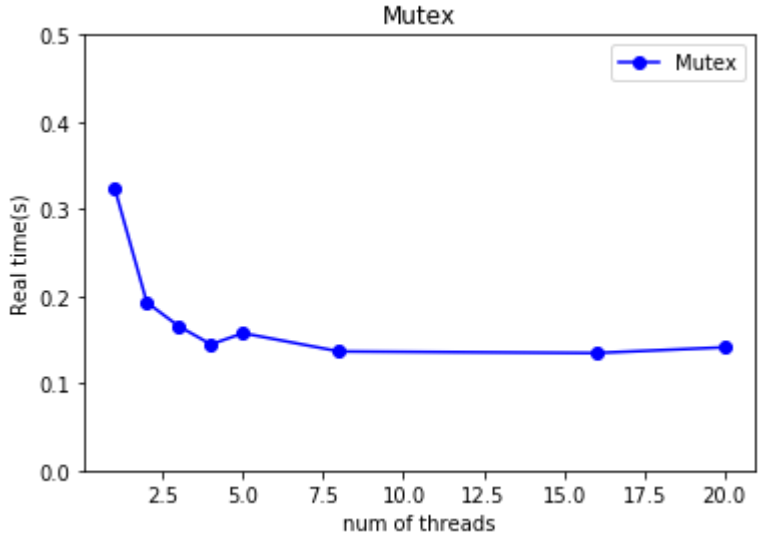
MUTEX

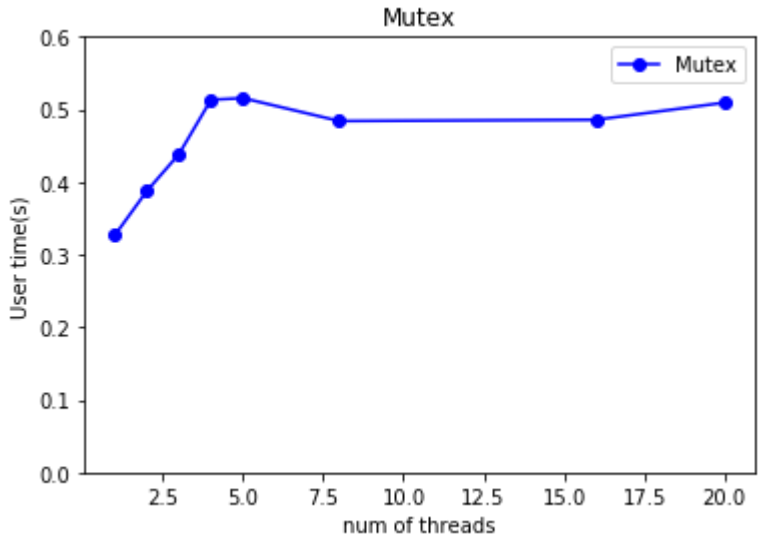
Procedure: **Create k number of threads. Create a shared variable g_sum which will**

store the final sum. There is a mutex g_sumMtx to ensure thread safe access of g_sum. Data is divided equally among threads. Each thread also contains mySum in which it stores the sum of data assigned to that thread. Each thread proceeds in following way:

- ITERATE OVER DATA
    - Increment mySUm
- Lock mutex.
- Increment g_sum by mySum.
- Unlock mutex.

Number of Threads	Number of Calculation per Thread	Average Real Time Taken(s)	Maximum Real Time Taken(s)	Minimum Real Time Taken(s)	Average User mode combined Time Taken among all CPUs(s)
1	1e8	0.3238	0.3088	0.3294	0.3265
2	5e7	0.192	0.179	0.21	0.387
3	3.33333333e8	0.1654	0.145	0.176	0.438
4	2.5e7	0.144	0.142	0.149	0.513
5	2e7	0.157	0.147	0.171	0.515
8	1.25e7	0.1365	0.118	0.145	0.484
16	6.25e6	0.134	0.118	0.144	0.485
20	5e6	0.141	0.136	0.144	0.509





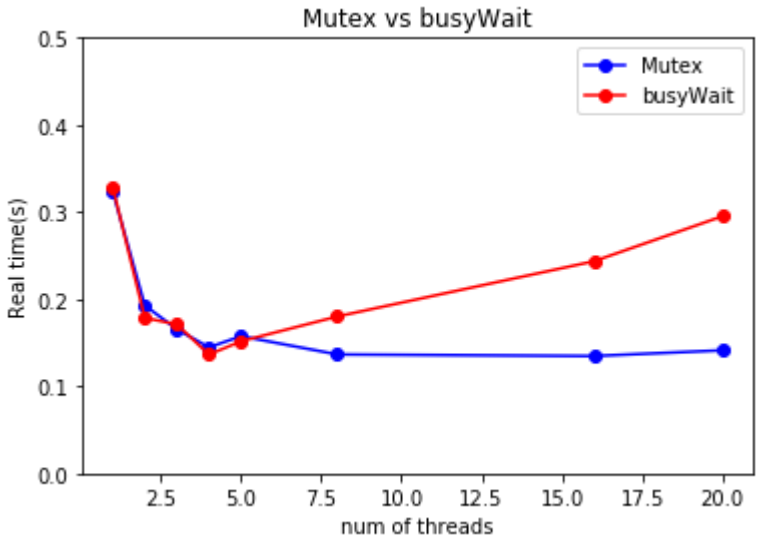
OBSERVATION

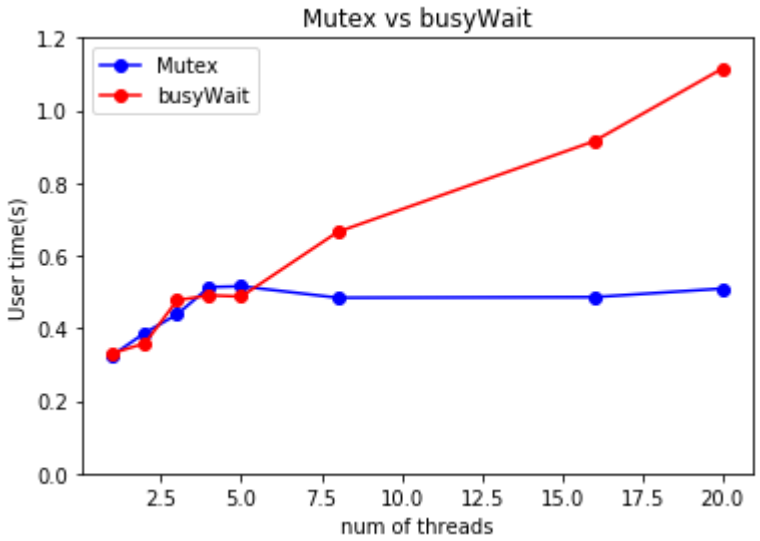
1. **Busy Wait:**
    a. Real time taken decreases as we increase the number of thread upto the number of cored. This is so because computation is distributed among threads and since computation in one thread is completely independent of others, most probably they will run parallely. However when number of thread is increased beyond number of cores then real time taken starts increasing. This is so because busy_wait forces strict ordering and one thread might have to wait for other thread to get descheduled. Example: suppose flag = 2 and thread 1, 4, 5 6 are spinning in busy wait. Now until one of the thread 1, 4, 5, 6 is deschedule no other thread can proceed.

b. User time taken is roughly number of thread times real time taken if number of threads is less than or equal to number of cores because we can expect them to run parallely hence the real time taken will be approximated time taken by one single thread. However if number of thread is more than number of cores then real time taken is not the approximate time taken by one thread and hence the user time is no more equal to number of threads times real time instead user time will approximately be time taken by one thread times number of thread.

2. Mutex:
    a. Real time taken decreases as we increase the number of thread upto the number of cored. This is so because computation is distributed among threads and since computation in one thread is completely independent of others, most probably they will run parallely. However when number of threads is increased beyond number of cores then real time taken remains constant. To explain this let's when number of thread was k(=number of cores) and each thread take approximate x seconds to complete its computation so the real time taken will be approximately x seconds. Now we double the number of threads to be 2k and since now load on one thread will be halved each thread take approximate x/2 seconds. Now maximum threads which can run in parallel are k but since time taken by each thread is also halved last thread will still complete computation after x seconds.
    b. Arguments are similar as in case of busy wait.
3. Mutex vs busy wait:
    a. As we can see from the graph both mutex and busy wait implementation real time decreases with number of threads and both almost take same time. However when number of threads increases beyond number of cores, real time taken by mutex version remains same while real time taken by busy wait version increases with number of threads. As discussed above(busy wait section) busy wait forces strict ordering and hence some time one thread which is in busy wait has to wait until another thread(which is also in busy wait) is descheduled.
    b. For both mutex and busy wait version User time taken is roughly number of thread times real time taken if number of threads is less than or equal to number of cores. However if number of thread is increased beyond number of cores then user time increase for busy wait version and remains constant for mutex version. In case of MUTEX version load on one thread is

proportional to 1/(num Of Thread) and user time as expected is proportional to (num Of Thread) and load on one thread = (num of thread * (1/ num of thread) = constant). In case of BUSY WAIT version same situation is expected but only problem is that busy wait forces ordering and some time one thread has to wait until other thread get descheduled which increase user time.

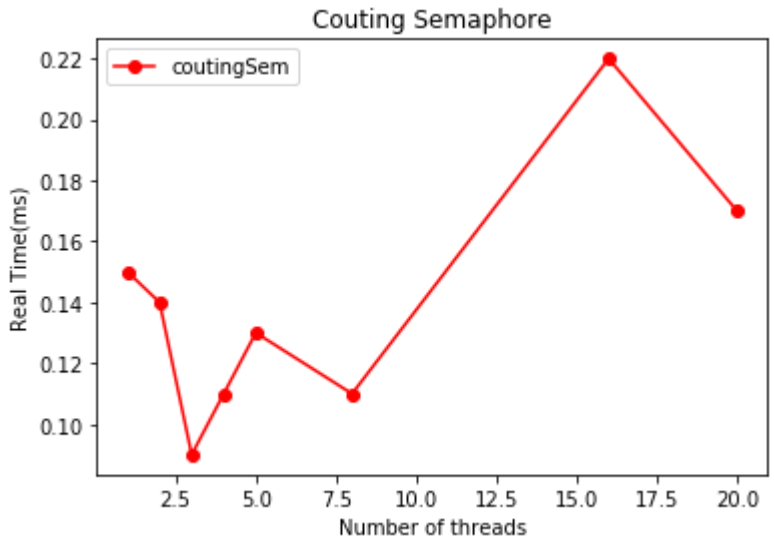




THREAD BARRIER

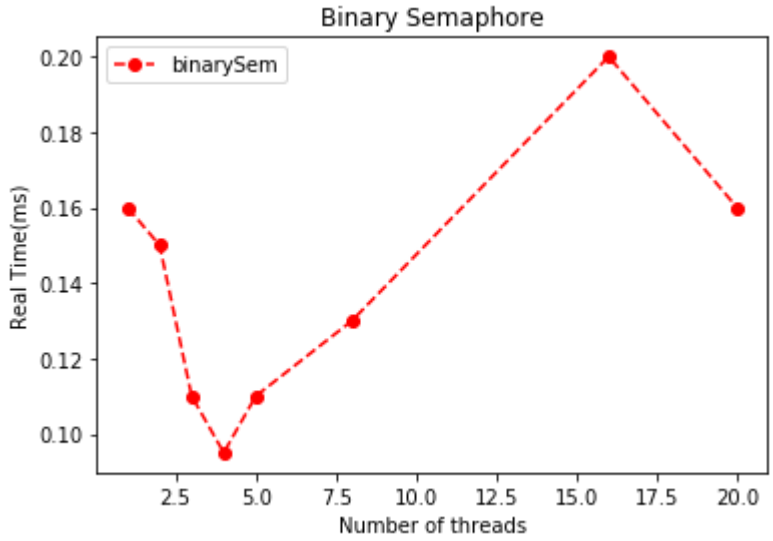
1. Using Counting Semaphore:

Number of Threads	Average Real Time Taken(ms) per thread	Maximum Real Time Taken(ms) per thread	Minimum Real Time Taken(ms) per thread
1	0.15	0.28	0.08
2	0.14	0.18	0.11
3	0.09	0.16	0.07
4	0.11	0.13	0.069
5	0.13	0.16	0.10
8	0.11	0.16	0.09
16	0.22	0.49	0.10
20	0.17	0.50	0.10



2. Using Binary Semaphore(although there is nothing like binary semaphore in pthread):

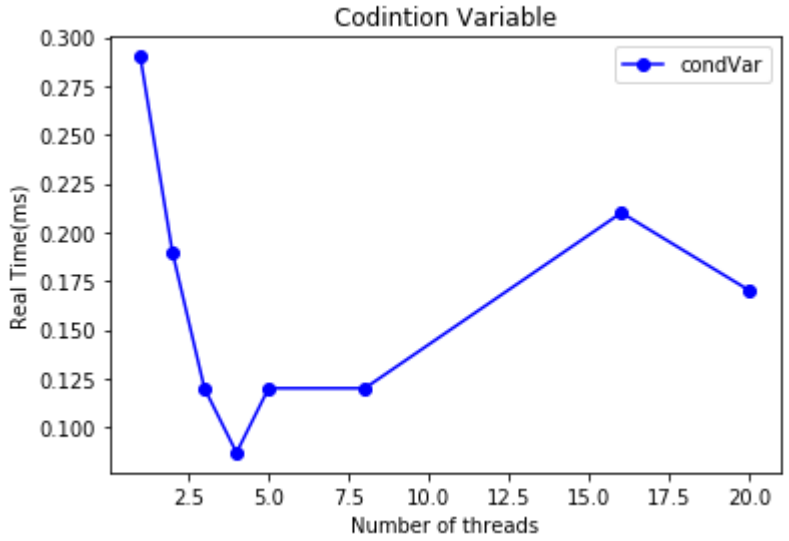
Number of Threads	Average Real Time Taken(ms) per thread	Maximum Real Time Taken(ms) per thread	Minimum Real Time Taken(ms) per thread
1	0.16	0.26	0.11
2	0.15	0.21	0.12
3	0.11	0.16	0.09
4	0.095	0.14	0.061
5	0.11	0.14	0.087
8	0.13	0.16	0.10
16	0.20	0.7	0.11
20	0.16	0.3	0.11



3. Using Condition Variable:

Number of Threads	Average Real Time Taken(ms) per thread	Maximum Real Time Taken(ms) per thread	Minimum Real Time Taken(ms) per thread
1	0.29	0.43	0.11
2	0.19	0.27	0.15

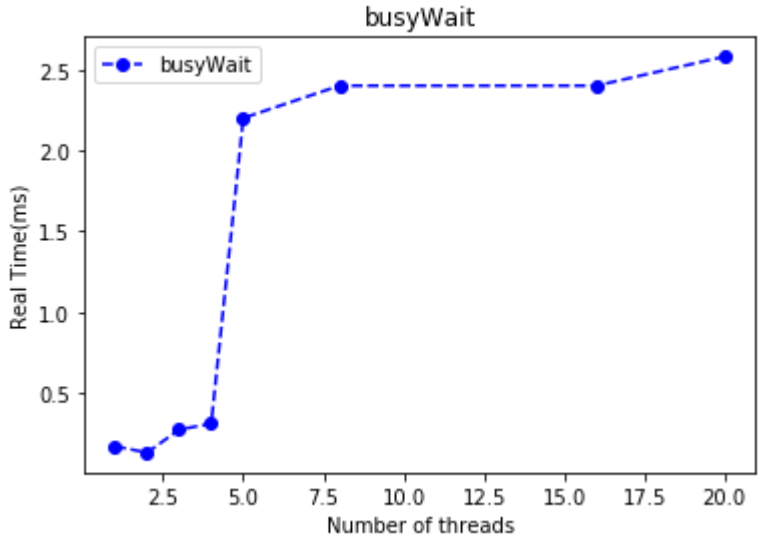
3	0.12	0.20	0.092
4	0.087	0.16	0.069
5	0.12	0.16	0.10
8	0.12	0.15	0.11
16	0.21	0.67	0.11
20	0.17	0.38	0.10



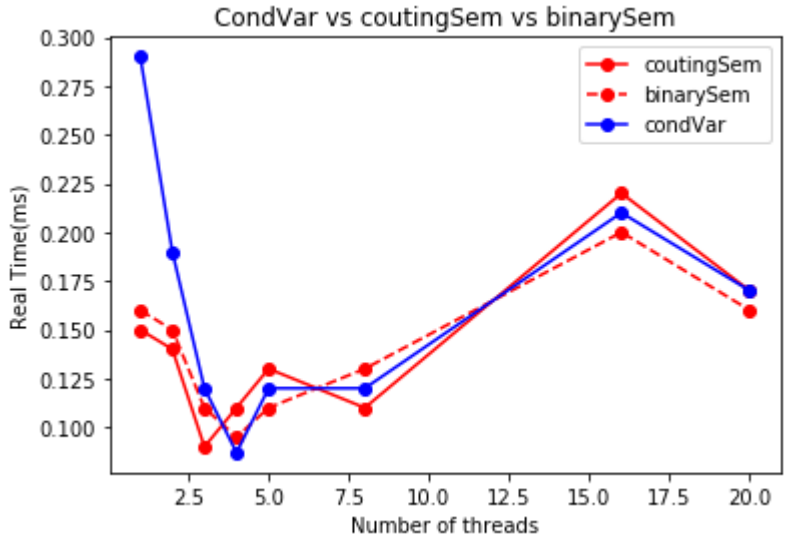
4. Using Busy Wait and a Mutex:

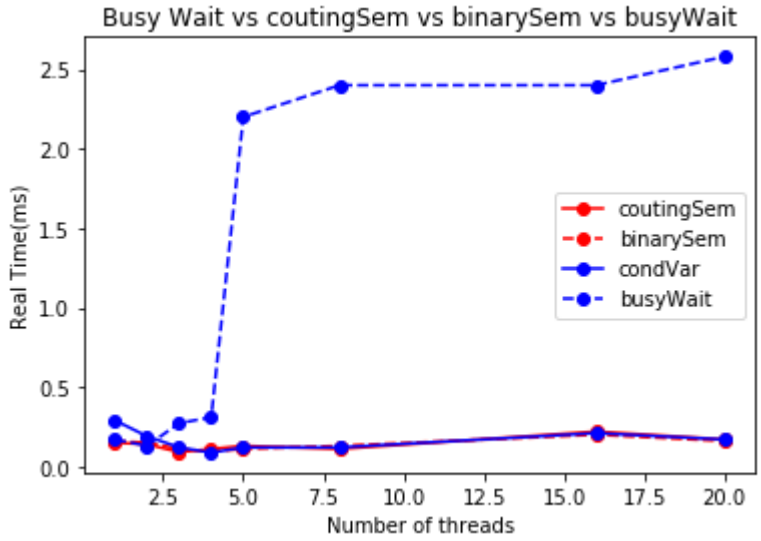
Number of Threads	Average Real Time Taken(ms) per thread	Maximum Real Time Taken(ms) per thread	Minimum Real Time Taken(ms) per thread
1	0.17	0.3	0.09
2	0.13	0.2	0.09
3	0.272	1.11	0.09
4	0.31	1.2	0.086
5	2.2	2.6	1.8
8	2.4	2.9	1.7

16	2.4	4.4	1.3
20	2.58	4.4	1.53



COMPARISONS:





OBSERVATIONS:

1. All the methods perform equally well if number of thread is upto number of cores. However if number of thread goes beyond number of cores then busy wait performance decreases rapidly while other method's performance decreases slowly.

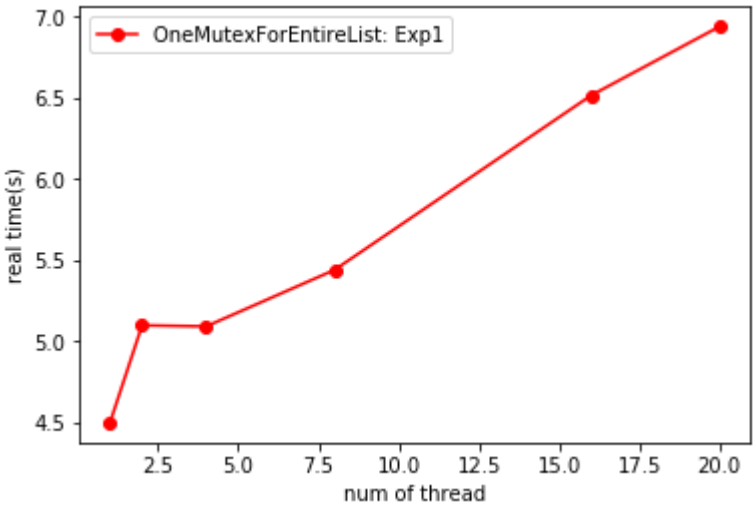
MULTI-THREADED LINKED LIST:

One Mutex for Entire List:

Experiment 1:

1000 initial keys, 100000 operations: 80%member, 15% insert, 5% delete

Number of Threads	Average Real Time Taken(ms)	Maximum Real Time Taken(ms)	Minimum Real Time Taken(ms)
1	4.51	4.63	4.35
2	5.098	5.63	4.93
4	5.091	5.33	4.97
8	5.439	5.508	5.39
16	6.515	6.73	6.324
20	6.94	6.99	6.87

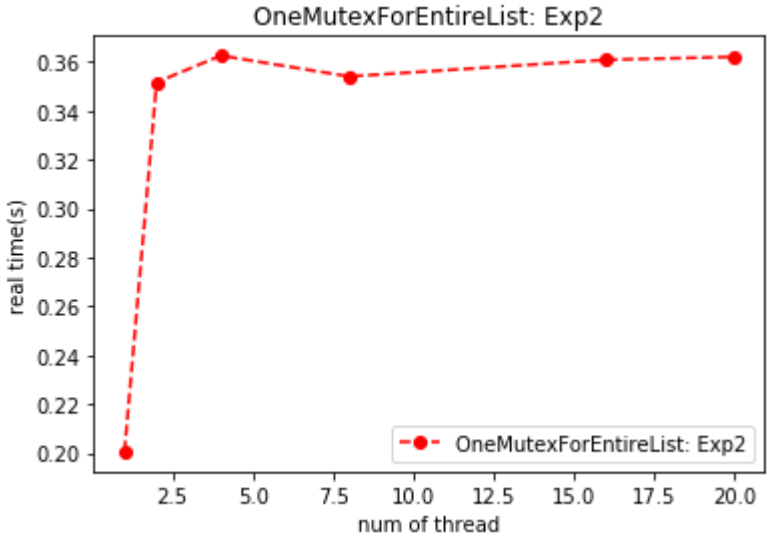


Experiment 2:

1000 initial keys, 100000 operations: 99.9%member, 0.05% insert, 0.05% delete

Number of Threads	Average Real Time Taken(ms)	Maximum Real Time Taken(ms)	Minimum Real Time Taken(ms)

1	0.206	0.202	0.2
2	0.351	0.365	0.342
4	0.362	0.368	0.355
8	0.354	0.358	0.35
16	0.360	0.364	0.358
20	0.362	0.367	0.354



OBSERVATION

1. When we increase number of insert operation time taken increases. This is because list size will increase with number of insert operations which will subsequently increase time taken for any time of operations. Also there is overhead related with memory allocation/deallocation which increases with number of insert/delete operations.

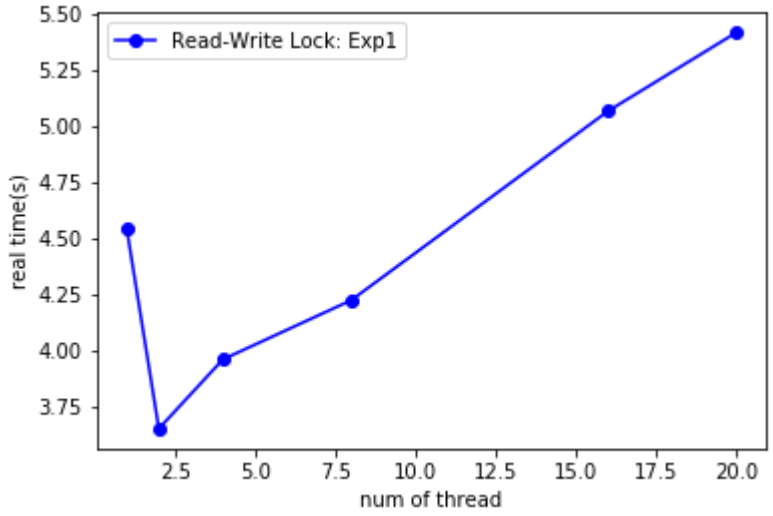
One READ-WRITE Lock for Entire List:

Experiment 1:

1000 initial keys, 100000 operations: 80%member, 15% insert, 5% delete

Number of Threads	Average Real Time Taken(ms)	Maximum Real Time Taken(ms)	Minimum Real Time Taken(ms)

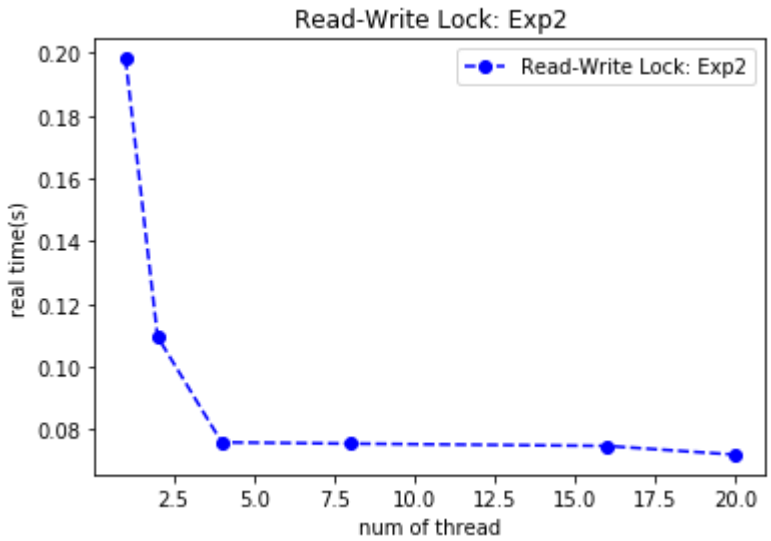
1	4.540	4.656	4.46
2	3.647	3.75	3.59
4	3.957	4.08	3.871
8	4.22	4.36	4.17
16	5.06	5.12	5.03
20	5.41	5.53	5.15



Experiment 2:

1000 initial keys, 100000 operations: 99.9%member, 0.05% insert, 0.05% delete

Number of Threads	Average Real Time Taken(ms)	Maximum Real Time Taken(ms)	Minimum Real Time Taken(ms)
1	0.198	0.2	0.197
2	0.109	0.122	0.105
4	0.075	0.079	0.074
8	0.0755	0.0766	0.0744
16	0.0748	0.078	0.073
20	0.072	0.079	0.068

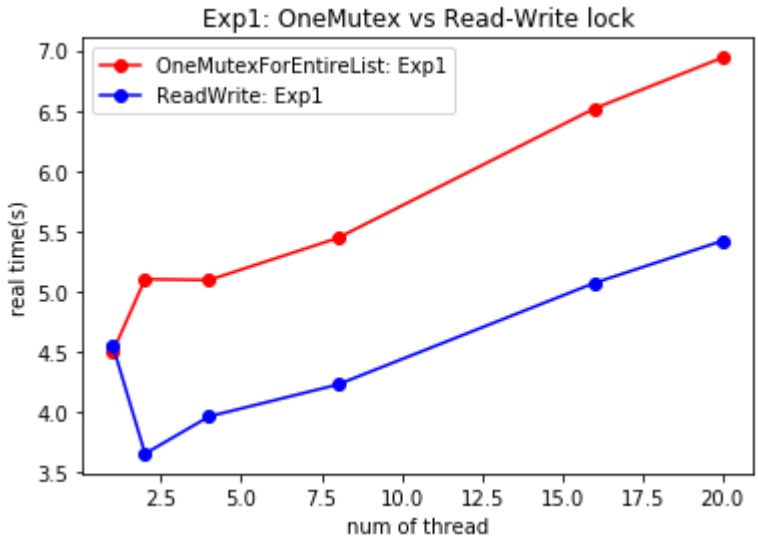


OBSERVATION

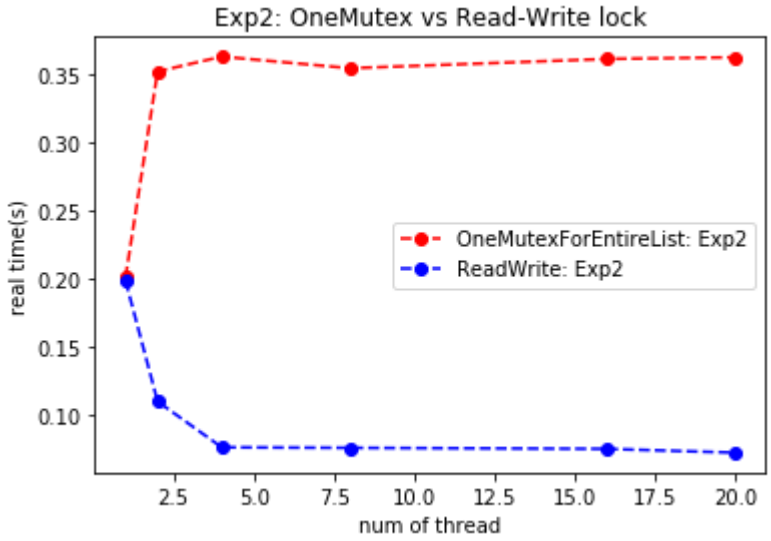
When we increase number of insert operation time taken increases. This is because list size will increase with number of insert operations which will subsequently increase time taken for any time of operations. Also there is overhead related with memory allocation/deallocation which increases with number of insert/delete operations.

PERFORMANCE COMPARISON

1. **1000 initial keys, 100000 operations: 80%member, 15% insert, 5% delete**



2. 1000 initial keys, 100000 operations: 99.9%member, 0.05% insert, 0.05% delete



1. When there is only one thread for both implementation(READ-WRITE lock and One mutex for entire list) operations are serialized hence time taken is same for both.

2. We can see that when number of read operation is much more compared to number of write operations then *READ-WRITE lock* is much more efficient than *One mutex lock for entire list*. This is so because *READ-WRITE lock* allows parallel read operations hence if there are multiple thread reading linked list then it can be done in parallel. This is also the reason why time taken decreases with number of thread for *READ-WRITE lock*. However maximum number of thread reading the linked list in parallel is limited by

number of cores hence if we increase number of thread beyond number of cores then time taken remains constant.