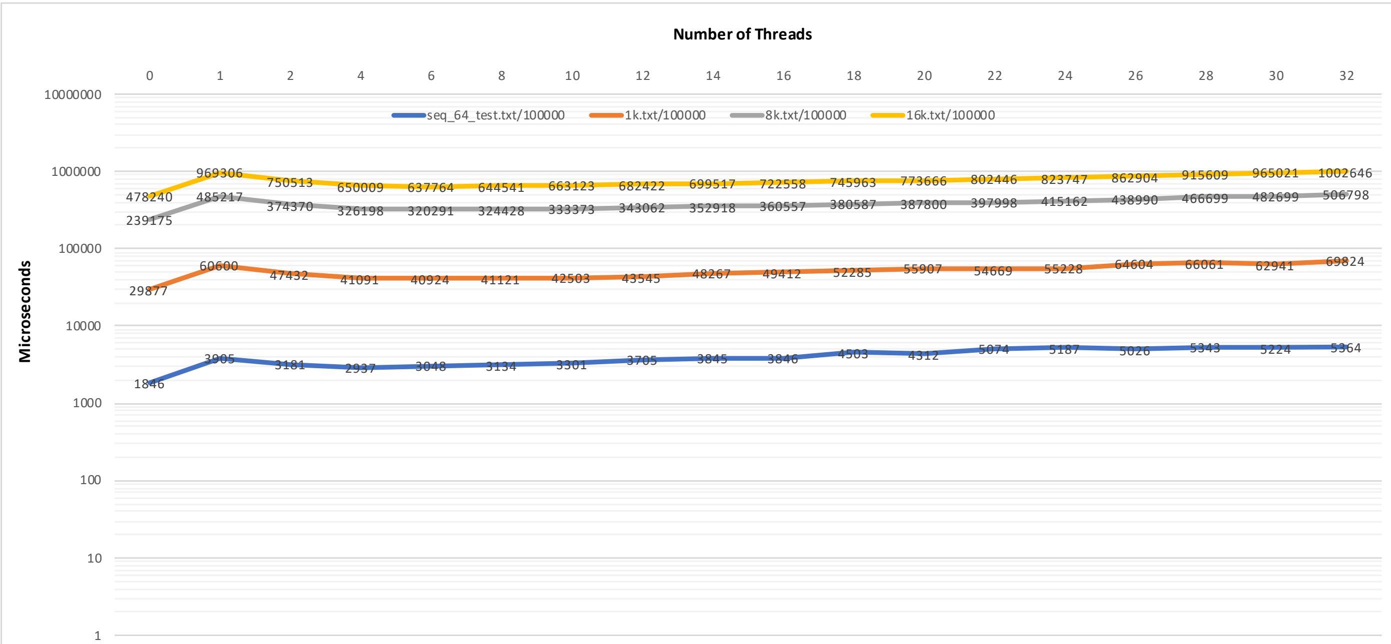
Srinivas Prasad Prabhu
UT EID - sp55629
Machine Used – Intel Xeon – 16 cores, 2 GHz + 64 GB RAM
CS380P – Lab 1 – Prefix Scan and Barriers

1) Recall that prefix scan requires a barrier for synchronization. In this step, you will write a work-efficient parallel prefix scan using pthread barriers. For each of the provided input sets, set the number of loops of the operator to 100000 (-l 100000) and graph the execution time of your parallel implementation over a sequential prefix scan implementation as a function of the number of worker threads used. Vary from 2 to 32 threads in increments of 2. Then, explain the trends in the graph. Why do these occur? From here: "A prefix sum algorithm is work-efficient if it does asymptotically no more work (add operations, in this case) than the sequential version. In other words, the two implementations should have the same work complexity, O(n).

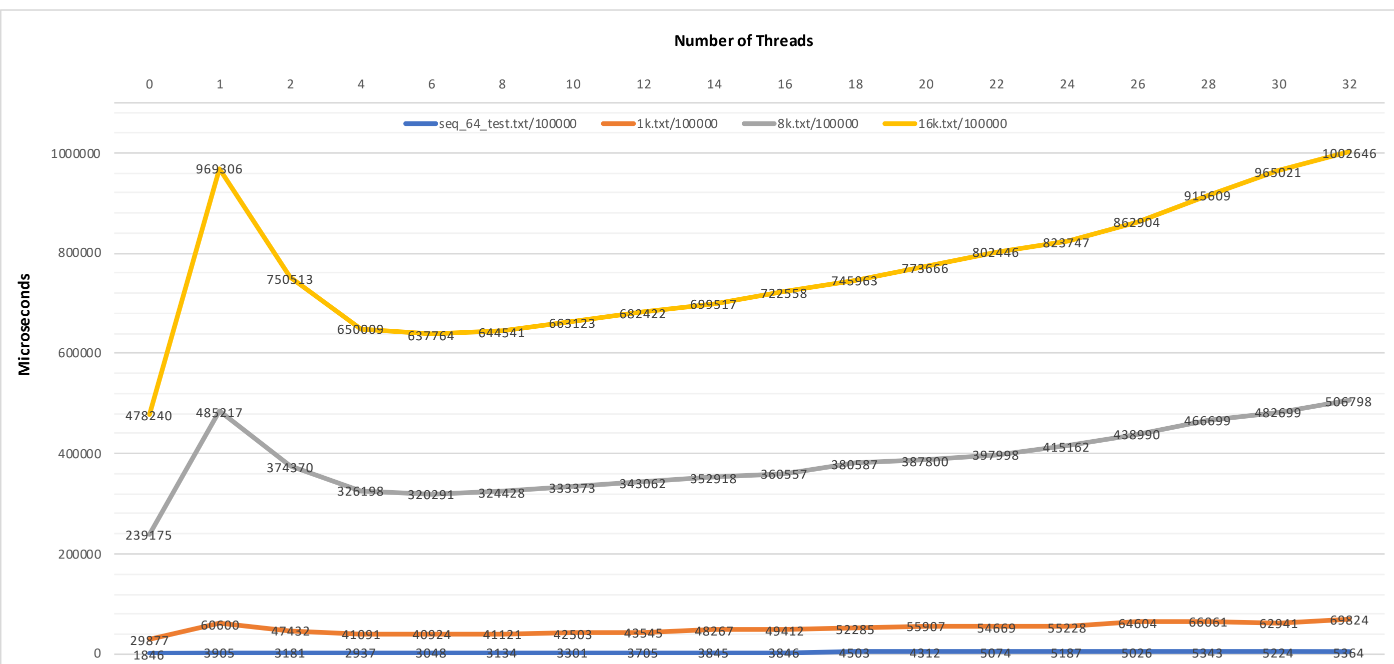
Ans –
Please find below the graph of the execution times of various inputs with -l 10E5 vs varying number of worker threads from 0-32. The 2 graphs below plot the same values of microseconds in linear and logarithmic scale.

Parallel Prefix Scan execution time (usec) vs Number of Threads - Loops - 100000

usec	0	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
seq_64_test.txt/100000	1846	3905	3181	2937	3048	3134	3301	3705	3845	3846	4503	4312	5074	5187	5026	5343	5224	5364
1k.txt/100000	29877	60600	47432	41091	40924	41121	42503	43545	48267	49412	52285	55907	54669	55228	64604	66061	62941	69824
8k.txt/100000	239175	485217	374370	326198	320291	324428	333373	343062	352918	360557	380587	387800	397998	415162	438990	466699	482699	506798
16k.txt/100000	478240	969306	750513	650009	637764	644541	663123	682422	699517	722558	745963	773666	802446	823747	862904	915609	965021	1002646



Execution time of Parallel Prefix Scan with varying number of threads (0 thread count means sequential) - Logarithmic Scale - Loops - 100000



Execution time of Parallel Prefix Scan with varying number of threads (0 thread count means sequential) - Loops - 100000

Observations -

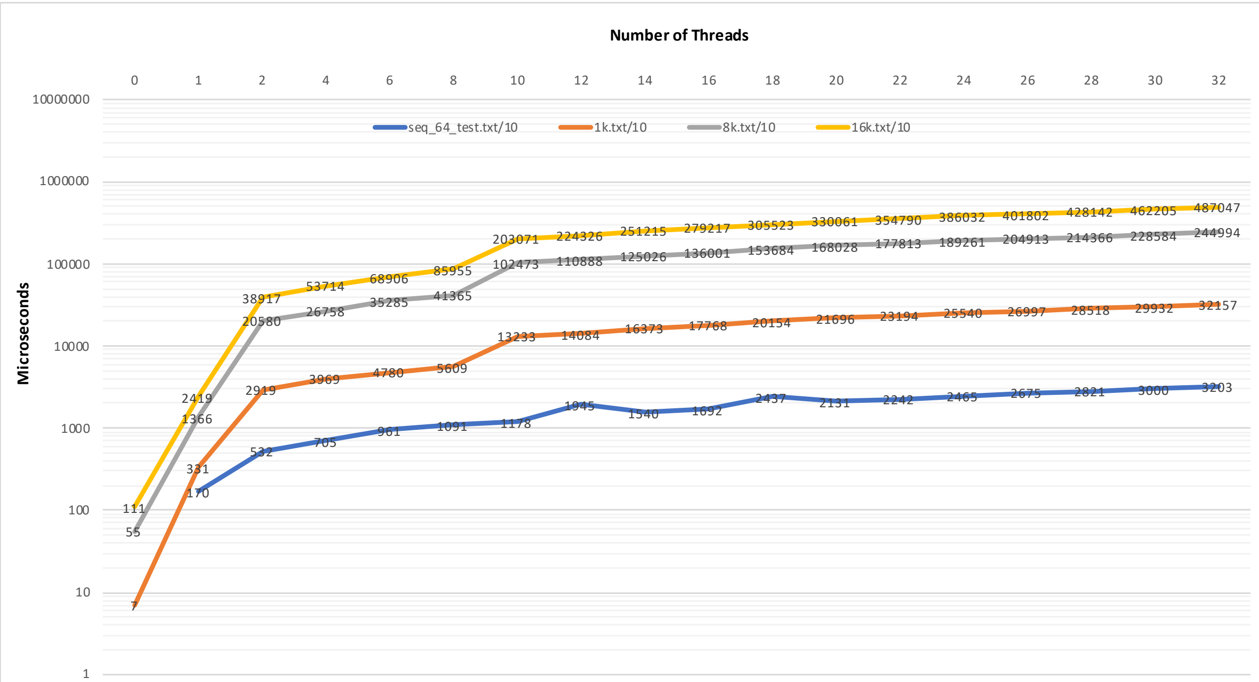
1) It can be observed from the graph that a parallel implementation with just 1 thread is highly in-efficient and consumes exponentially higher time. This is because of the overheads involved in thread creation and the context switch time at each of the barriers.
2) As the number of threads increases, the work starts getting more evenly divided among the threads and we can see exponential reduction in times compared to a single thread.
3) From the logarithmic graph it can be observed that beyond 8-10 threads, the time consumed starts increasing higher than the sequential implementation even though the growth is slow.
4) As the number of threads increases, the size of the blocks of input on which each thread works on decreases. This leads to very high rate of cache invalidations. This reduces the actual speedup obtained by the parallel implementation.
5) As the number of threads increases beyond the number of physical cores, a lot of time is spent in context switching between the various threads. This again leads to reduction in the speedup obtained by the parallel implementation.
6) As all the worker threads are working on the same data, the need for barrier synchronization also reduces the performance of the parallel implementation as multiple threads are waiting for longer at the same barrier.
7) Due to points 4-6 above just increasing the number of threads doesn’t yield a high-performance solution. Partitioning the data optimally so that different threads are working on different independent blocks of data (which fit into a cache line), which in turn reduces the need for barrier synchronization will improve the performance of parallel prefix scan.

2) Now that you have a working and, hopefully, efficient parallel implementation of prefix scan, try changing the amount of loops the operator does to 10(-l 10) and plot the same graph as before. What happened and why? Vary the -l argument and find the inflexion point, where sequential and parallel elapsed times meet (does not have to be exact, just somewhat similar). Why can changing this number make the sequential version faster than the parallel? What is the most important characteristic of it that makes this happen? Argue this in a general way, as if different -l parameters were different operators, afterall, the -l parameter is just a way to quickly create an operator that does something different.

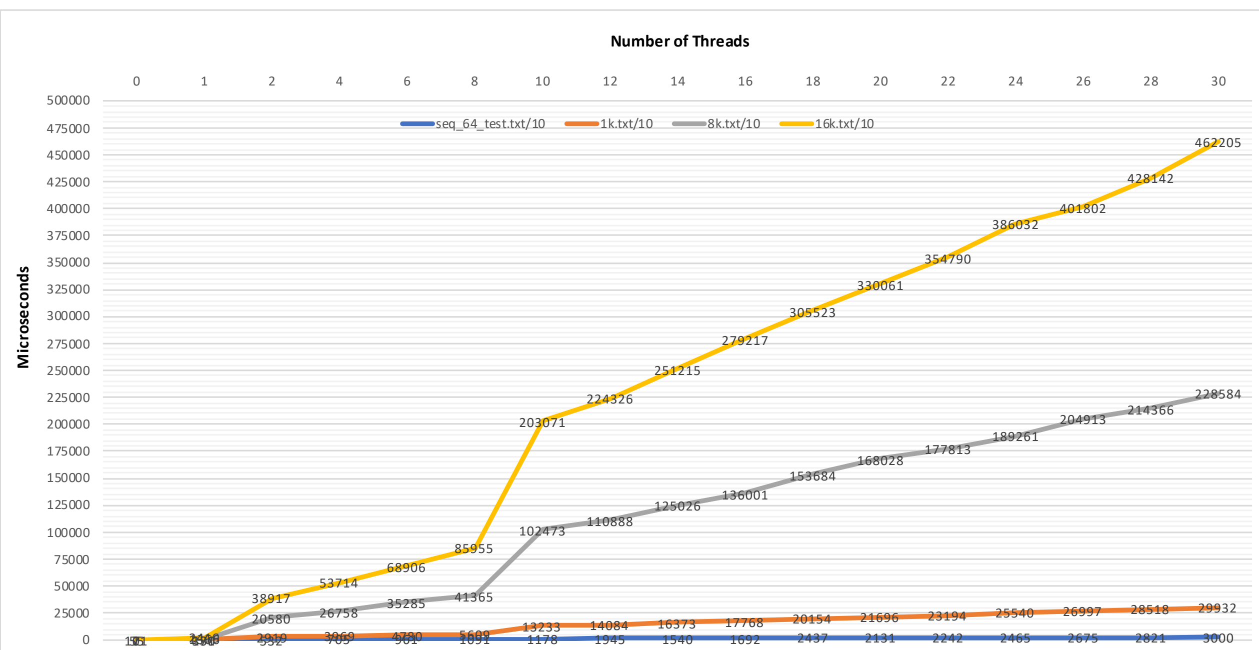
Ans –
Please find below the graph of the execution times of various inputs with -l 10 vs varying number of worker threads from 0-32. The 2 graphs below plot the same values of microseconds in linear and logarithmic scale.
Second, graphs - various inputs with different loop numbers from 1 – 10E7 vs execution time is plotted on a logarithmic scale of time, with different fixed number of threads.

Parallel Prefix Scan execution time (usec) vs Number of Threads - Loops - 10

usec	0	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
seq_64_test.txt/10	0	170	532	705	961	1091	1178	1945	1540	1692	2437	2131	2242	2465	2675	2821	3000	3203
1k.txt/10	7	331	2919	3969	4780	5609	13233	14084	16373	17768	20154	21696	23194	25540	26997	28518	29932	32157
8k.txt/10	55	1366	20580	26758	35285	41365	102473	110888	125026	136001	153684	168028	177813	189261	204913	214366	228584	244994
16k.txt/10	111	2419	38917	53714	68906	85955	203071	224326	251215	279217	305523	330061	354790	386032	401802	428142	462205	487047



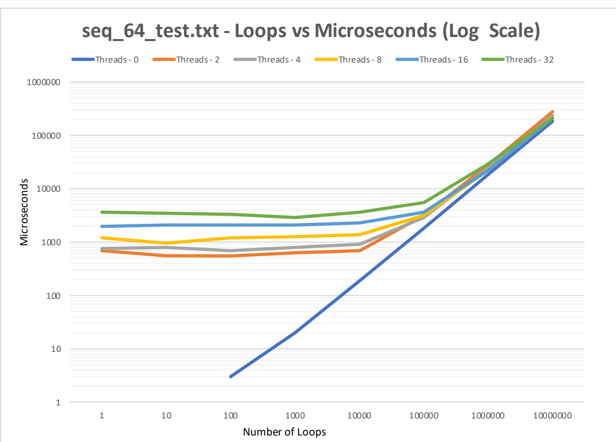
Execution time of Parallel Prefix Scan with varying number of threads (0 thread count means sequential) - Logarithmic Scale - Loops 10

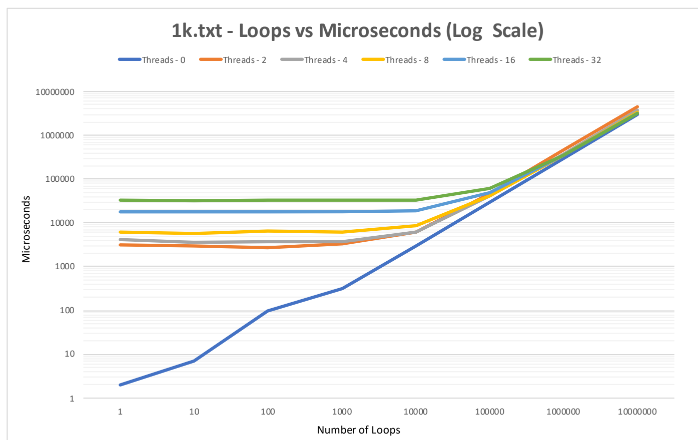


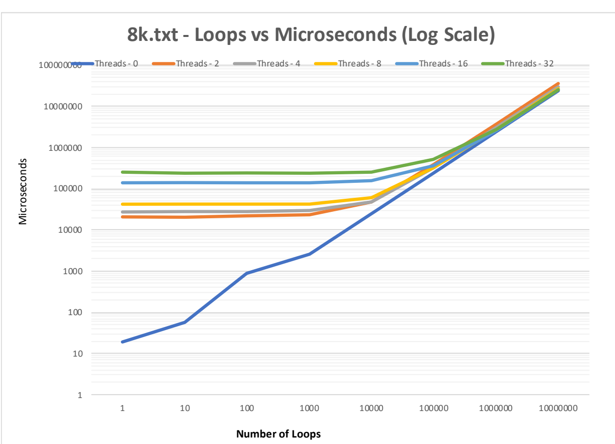
Execution time of Parallel Prefix Scan with varying number of threads (0 thread count means sequential) - Loops 10

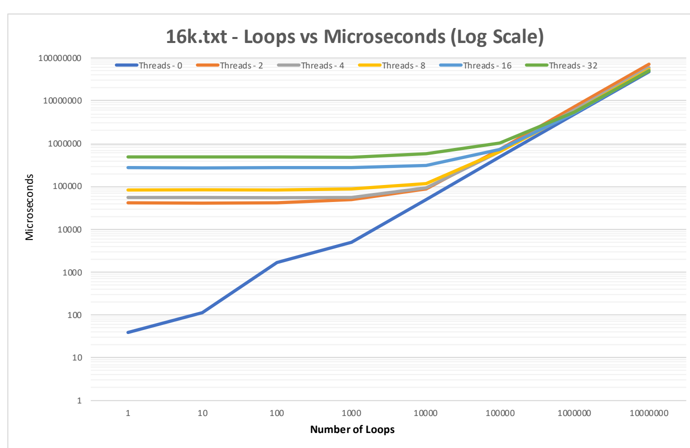
Execution time (usec) of different inputs with varying loops and fixed number of threads - pthread barrier

seq_64_test.txt							1k.txt						
	Threads - 0	Threads - 2	Threads - 4	Threads - 8	Threads - 1	Threads - 32		Threads - 0	Threads - 2	Threads - 4	Threads - 8	Threads - 1	Threads - 32
1	0	676	749	1184	1970	3639	**1**	2	3121	4057	6118	18243	32389
10	0	554	786	955	2048	3530	**10**	7	2925	3595	5727	18182	32065
100	3	561	691	1196	2047	3391	**100**	97	2713	3625	6449	18053	32473
1000	20	620	812	1278	2069	2891	**1000**	312	3404	3800	6164	17977	32897
10000	186	676	900	1351	2256	3607	**10000**	3020	6147	6266	8589	19274	33044
100000	1838	3236	2910	3188	3608	5541	**100000**	29933	47451	41317	41232	48316	62250
1000000	18427	28080	23944	22155	21895	28541	**1000000**	298449	450203	377349	342555	338415	346548
10000000	183823	277998	234599	214082	208098	210820	**10000000**	2984304	4483120	3737520	3369506	3189994	3157642
8k.txt							16k.txt						
	Threads - 0	Threads - 2	Threads - 4	Threads - 8	Threads - 1	Threads - 32		Threads - 0	Threads - 2	Threads - 4	Threads - 8	Threads - 1	Threads - 32
1	19	20388	27213	42133	141403	245891	**1**	39	42572	56311	85131	273836	485115
10	56	20314	27255	42300	140593	242263	**10**	111	40822	55396	83718	270694	486128
100	862	21793	27593	42775	141141	241676	**100**	1694	41313	54693	85090	271880	481648
1000	2540	23911	28847	43586	141103	243439	**1000**	5084	49517	56945	89148	276208	480844
10000	24065	47271	46686	59236	153481	258310	**10000**	48159	90224	94689	114670	302259	596191
100000	239121	376196	328149	324021	363584	510425	**100000**	478251	746777	654386	644787	723187	1013534
1000000	2389581	3599748	3012040	2734501	2618149	2698534	**1000000**	4779271	7193906	6018217	5508610	5240268	5399959
10000000	23893812	35850970	29907518	26967498	25475604	25129539	**10000000**	47790761	71721033	59777722	54195151	50935196	50020808









Observations –

1) Just like previously seen, it can be observed from the graph that a parallel implementation with just 1 thread is highly in-efficient and consumes exponentially higher time. This is because of the overheads involved in thread creation and the context switch time at each of the barriers.
2) Sequential implementation seems to be much faster than parallel implementation for low values of loops. This is because the parallel implementation needs barriers where threads are waiting repeatedly. The parallel implementation also has overheads of context switching and maintaining cache coherence. This causes increased time in parallel implementation compared to the sequential implementation.
3) As we vary the number of loops, from the logarithmic graphs we can observe that the sequential and parallel implementation almost start taking the same time for loop values greater than 10E5 -10E6.
4) Loops value indicates a certain amount of work done during each operation. If the loop value is low, the sequential implementation is exponentially faster. This is because of no overhead of thread creation and barrier synchronization. As we increase the loops value, it indicates more work being done in each operation.

   Since the sequential version must do this work at each of the O(n) operations, its time increases proportionally to the increase in loops value.

   In the parallel case, the algorithm is still doing the O(n) order operations (work-efficient). However, there are more processing elements (PE's) which are doing this work in parallel. This implies that with increased work the parallel implementation will start yielding higher efficiency compared to a sequential implementation. Parallel algorithms still have the overhead of cache coherence and barrier synchronization which prevent them from achieving ideal speedup. However, this efficiency can be improved by increasing the number of processing elements (PE's) and better barrier primitives.

   To achieve high efficiency and scalability parallel algorithms should be able to break the problem dataset into independent parts which can fit in caches. Better spatial and temporal locality in the caches can lead to higher efficiency.

3) In this step you will build your own re-entrant barrier. Recall from lecture that we considered several implementation strategies and techniques. We recommend you base your barrier on pthread's spinlocks but encourage you to use other techniques we discussed in this course. Regardless of your technique, answer the following questions: how is/isn't your implementation different from pthread barriers? In what scenarios would each implementation perform better than the other? What are the pathological cases for each? Use your barrier implementation to implement the same work-efficient parallel prefix scan. Repeat the measurements from part 2, graph them, and explain the trends in the graph. Why do these occur? What overheads cause your implementation to underperform an "ideal" speedup ratio?
How do results from part 2 and part 3 compare? Are they in line with your expectations? Suggest some workload scenarios which would make each implementation perform worse than the other.

Ans –
My implementation of the spin barrier uses one spin lock and 2 atomic variables.
Pseudocode

```
spin_lock()
Incr waiters()
Incr waitingThds()

If (waiters() == expected())
{
   decrement waiters()
    do {
        // sleep for 1 ns
    } loop until waiters() == 0;
    atomic compare and exchange waiting threads = 0
    spin_unlock()
}
else
{
   spin_unlock()
   do {
        // sleep for 1 ns
    } loop until waiting threads == expected
   decrement waiters()
}
```

This implementation is different from the pthread_barrier_t –

a) Each thread is constantly polling to check if the waitingThreads is equal to expected instead of waiting (scheduling out) until notified.
b) Having a while loop waiting for the condition simulates a spin lock.
   The busy wait condition (without the nanosleep) performs better than the pthread_barrier when thread count is less than the PE's. This is because it eliminates the overhead of context switching.
   The sleep of 1ns was added to improve the performance compared to a busy wait for the case of higher thread count compared to processing elements (PE's).
   I was observing an exponential decrease in performance without the ns sleep when the number of threads exceeds the number of process elements (PE's). This was because all the PE's were occupied with threads spinning and doing the check. Any extra threads had to wait until the current running thread is scheduled out after utilization of its time slice. Addition of the 1ns sleep helps when the number of threads exceeds the number of PE's, as this leads to scheduling out this thread and giving an opportunity for another thread to run. This provides better time performance on the barrier despite having more threads than PE's.
c) Usage of volatile, atomic variables and CAS commands ensures that memory barriers are applied implicitly without any re-ordering.

Pthread_barrier's will perform better when there are large number of threads as the barrier schedules' out each thread and puts it in a wait state. This ensures optimal utilization of the compute resources. The pathological case for pthread_barrier's is having just 1 thread. This leads to lot of un-necessary work related to context switches.
My implementation using loops (without the sleep) will perform better for lesser number of threads as it will avoid the context switches overhead involved with waiting approach. The pathological case for my implementation is just having many threads (more than the number of PE's). This is because with many threads each thread must get a chance to be scheduled, to check for the validity of the condition in the while loop.

My implementation is unable to achieve the ideal speed up ratio, due to constantly polling (busy waiting) on the condition as the number of threads increase.

Please find below, the graphs - various inputs with different loop numbers from 1 – 10E7 vs execution time is plotted on a logarithmic scale of time, with different fixed number of threads. This data is from the implementation using the 1ns sleep.

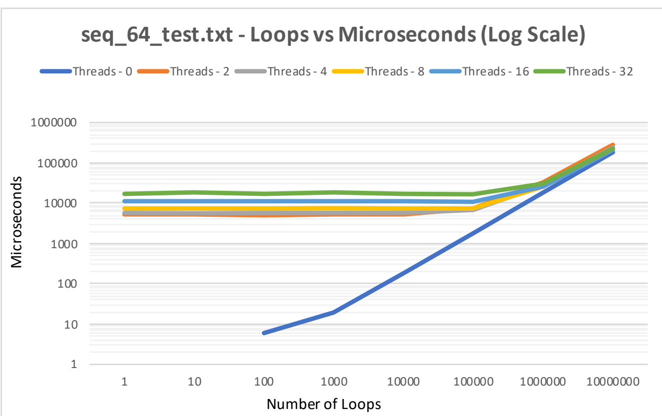
Execution time (usec) of different inputs with varying loops and fixed number of threads - custom barrier

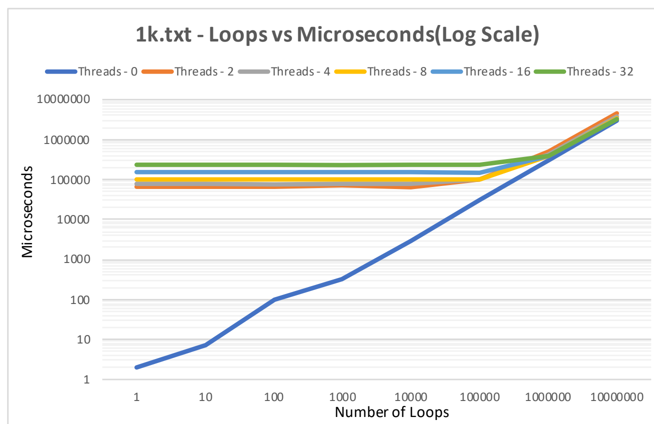
seq_64_test.txt						
	Threads - 0	Threads - 2	Threads - 4	Threads - 8	Threads - 16	Threads - 32
1	0	5204	5678	7514	11282	17666
10	0	5111	5610	7525	11383	18310
100	6	5009	5657	7535	11386	17351
1000	19	5295	5657	7500	11379	17774
10000	185	5260	5716	7528	11458	17266
100000	1837	7153	6665	7549	10811	16660
1000000	18386	32090	27714	25302	26776	30933
10000000	183753	281463	237706	217398	210724	224571

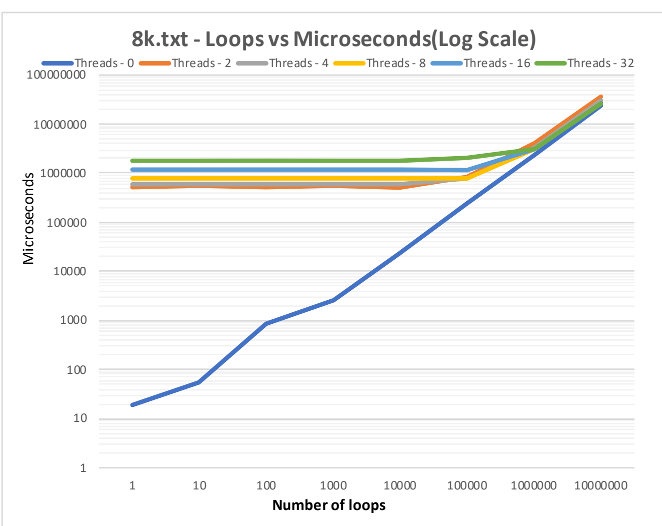
1k.txt						
	Threads - 0	Threads - 2	Threads - 4	Threads - 8	Threads - 16	Threads - 32
1	2	67543	75872	101955	151947	231222
10	7	67862	75684	102119	152494	238919
100	98	68818	75681	100913	152678	231828
1000	317	71268	76243	101296	151237	230651
10000	3004	64355	76539	102166	152529	231025
100000	29863	103895	96848	100785	147736	241421
1000000	298430	507190	434486	395261	376384	384329
10000000	2983840	4539134	3791887	3423677	3257409	3226492

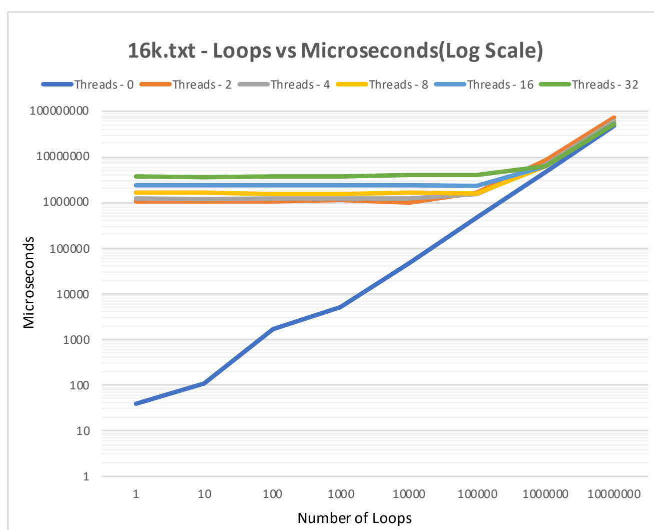
8k.txt						
Loops	Threads - 0	Threads - 2	Threads - 4	Threads - 8	Threads - 16	Threads - 32
1	19	530338	595179	795172	1199599	1817060
10	55	538973	596763	796952	1198131	1851915
100	863	537213	608401	796520	1195260	1800884
1000	2539	562390	596912	792735	1189516	1841030
10000	24079	509717	601212	797461	1184087	1837847
100000	239085	823499	769921	785795	1159571	2000040
1000000	2390882	4050687	3453623	3158836	3011857	3048802
10000000	23890584	36309775	30351306	27415624	26905760	26036879

16k.txt						
	Threads - 0	Threads - 2	Threads - 4	Threads - 8	Threads - 16	Threads - 32
1	39	1076162	1200775	1586758	2393754	3644613
10	111	1069594	1192479	1595985	2385366	3574238
100	1699	1078322	1192650	1584767	2390199	3629868
1000	5051	1134219	1194626	1574738	2384035	3641775
10000	48148	992072	1200598	1604468	2383671	4100936
100000	478160	1646741	1539809	1567827	2310668	3941819
1000000	4778723	8099783	6908327	6313233	6062059	6117996
10000000	47782810	72609133	60659209	55492481	54051200	51632190









Observations –

1) Just like previously seen, it can be observed that the sequential implementation seems to be much faster than parallel implementation for low values of loops. This is because the parallel implementation needs barriers where threads are waiting repeatedly. The parallel implementation also has overheads of context switching and maintaining cache coherence. This causes increased time in parallel implementation compared to the sequential implementation.
2) With the increase in the number of loops and threads, my barrier implementation is polling in the loops and trying to check for the condition. This leads to more work done than a pthread_barrier for large number of threads.

   This is evident in the convergence (sequential vs parallel) loop count. We can see the sequential time converging with parallel implementation for loop counts of approximately 10E6 or more compared to 10E5 or more in the pthread_barrier case. (Part 2)

   This is in line with the expectations of my barrier implementation.