1:

```
[sedupuganti@grace4 HW2-735]$ ./sort_list.exe 4 1
List Size = 16, Threads = 2, error = 0, time (sec) =   0.0002, qsort_time =   0.0000
[sedupuganti@grace4 HW2-735]$ ./sort_list.exe 4 2
List Size = 16, Threads = 4, error = 0, time (sec) =   0.0004, qsort_time =   0.0000
[sedupuganti@grace4 HW2-735]$ ./sort_list.exe 4 3
List Size = 16, Threads = 8, error = 0, time (sec) =   0.0008, qsort_time =   0.0000
[sedupuganti@grace4 HW2-735]$ ./sort_list.exe 20 4
List Size = 1048576, Threads = 16, error = 0, time (sec) =   0.0313, qsort_time =   0.1494
[sedupuganti@grace4 HW2-735]$ ./sort_list.exe 24 8
List Size = 16777216, Threads = 256, error = 0, time (sec) =   0.2876, qsort_time =   2.6388
[sedupuganti@grace4 HW2-735]$
```

The parallelized thread implementation is indeed working based on this output, and no errors were found.

2:

K = 12

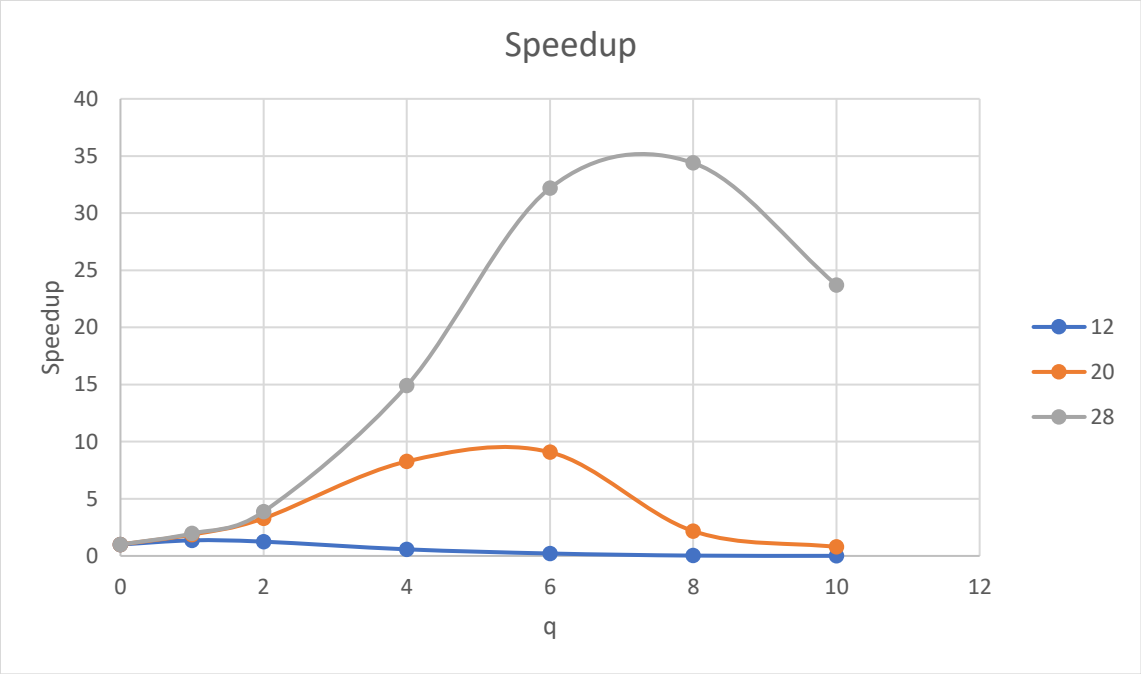
k	q	threads	time	speedup	efficiency
12	0	1	0.0015	1	1
12	1	2	0.0011	1.363636	0.681818
12	2	4	0.0012	1.25	0.3125
12	4	16	0.0026	0.576923	0.036058
12	6	64	0.007	0.214286	0.003348
12	8	256	0.0516	0.02907	0.000114
12	10	1024	0.2163	0.006935	6.77E-06

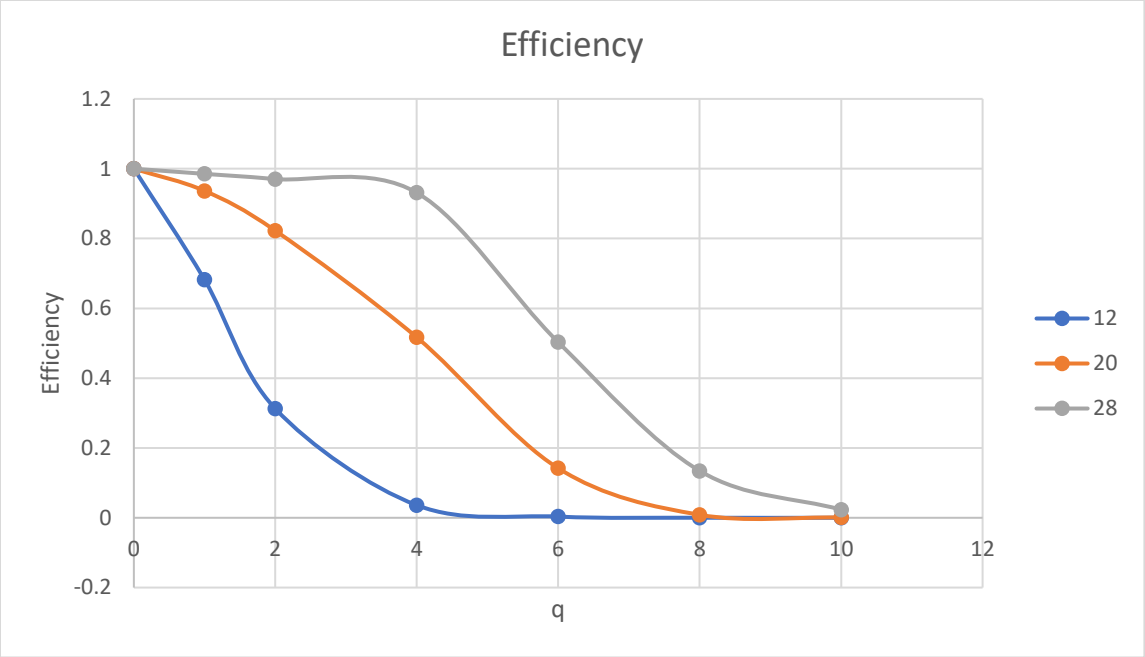
K = 20

k	q	threads	time	speedup	efficiency
20	0	1	0.177	1	1
20	1	2	0.0945	1.873016	0.936508
20	2	4	0.0538	3.289963	0.822491
20	4	16	0.0214	8.271028	0.516939
20	6	64	0.0195	9.076923	0.141827
20	8	256	0.0814	2.174447	0.008494
20	10	1024	0.2259	0.783533	0.000765

K = 28

k	q	threads	time	speedup	efficiency
28	0	1	63.1685	1	1
28	1	2	32.0576	1.970469	0.985234
28	2	4	16.2762	3.881035	0.970259
28	4	16	4.2392	14.90104	0.931315
28	6	64	1.9621	32.19433	0.503036
28	8	256	1.8369	34.38864	0.134331
28	10	1024	2.6645	23.70745	0.023152

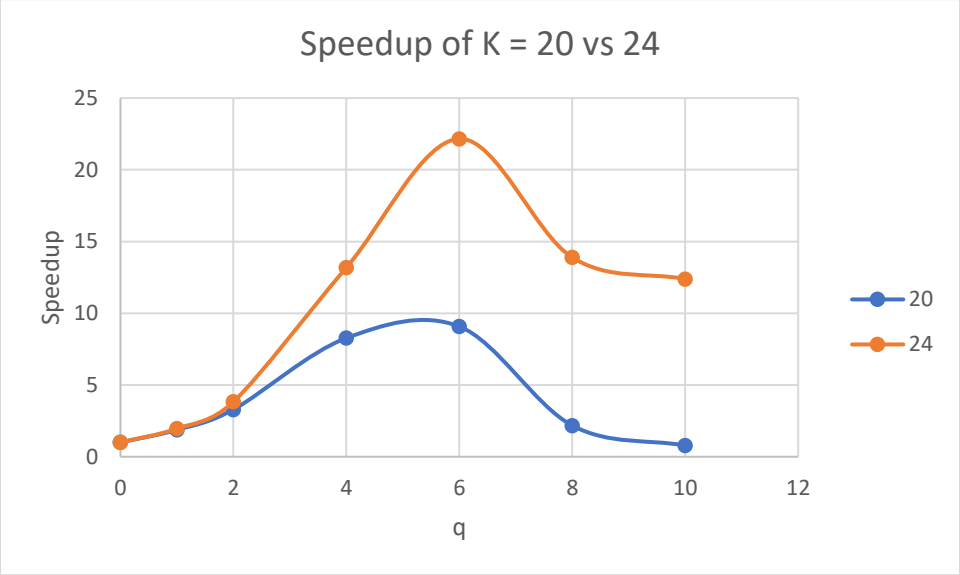


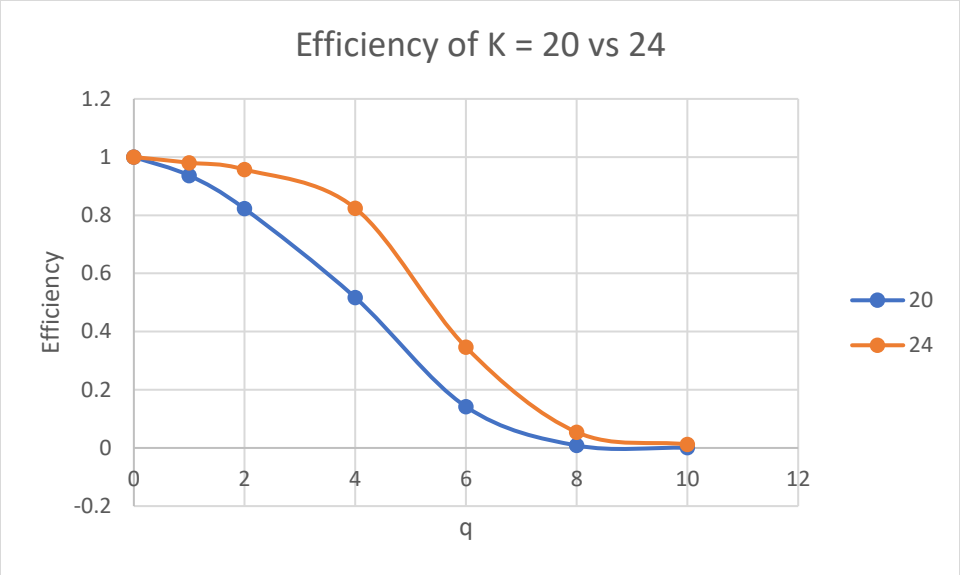


Based on the results of the experiment, it seems that the output does indeed align with the typical behavior of parallel programming. The idea of parallel programming is that multiple tasks are being ran at the same time, which is exactly what Is being displayed here. As the list size increases (k = 28), there is a larger speedup, especially when the number of threads increase to 256 threads, before dipping back down. The reason the speedup dips down by the time it reaches 1024 threads is due to the overhead of having so many threads. The threads slow down due to the amount. If we look at the efficiency graph, we can see that as the number of threads increased, the efficiency decreased for all 3 values of k. When reaching 16 threads, the line for k = 28 begins to dip, as it seems due to the increase in threads, not all threads are able to remain efficient. Therefore, the efficiency decreases despite there being an increase in speedup. The threads are being underutilized. Due to the increase in threads, they are not all able to stay efficient. These graphs do align with the principles of parallel programming.

3:

k	q	threads	time	speedup	efficiency
20	0	1	0.177	1	1
20	1	2	0.0945	1.873016	0.936508
20	2	4	0.0538	3.289963	0.822491
20	4	16	0.0214	8.271028	0.516939
20	6	64	0.0195	9.076923	0.141827
20	8	256	0.0814	2.174447	0.008494
20	10	1024	0.2259	0.783533	0.000765
24	0	1	3.346	1	1
24	1	2	1.7061	1.961198	0.980599
24	2	4	0.8739	3.828813	0.957203
24	4	16	0.2538	13.18361	0.823976
24	6	64	0.151	22.15894	0.346233
24	8	256	0.2409	13.88958	0.054256
24	10	1024	0.2701	12.388	0.012098





The values chosen to compare were 20 and 24. These two values are interesting to see as speedup is very different between the two. There is an increase in speed when k = 24. This is due to the increase in the list size, which causes the speedup to increase. Despite their being a speedup in the threads, the efficiency decreases for both values, as speedup increase. This does indicate that the threads are not being used efficiently despite using a parallelized thread implementation. This is interesting as the whole idea of parallel programming is to improve the time and efficiency of a program. I believe this indicates that up to a certain point is the number of threads good to use otherwise the efficiency of each thread may not be up to par despite their being a speedup. The threads are fully utilized initially (especially up until q = 2), especially as the list size increases, however due to the number of threads, the efficiency decreases, as each thread is not being fully utilized. Overall, this does showcase how the parallelized thread implementation is working properly.