1. (70 points) Revise the code to implement parallel merge sort via OpenMP. The code should compile successfully and should report `error=0` for the following instances:

```
./sort_list_openmp.exe 4 1
./sort_list_openmp.exe 4 2
./sort_list_openmp.exe 4 3

./sort_list_openmp.exe 20 4
./sort_list_openmp.exe 24 8
```

Solution:

The code given in sort_list_openmp.c was revised incorporating the OpenMP directives to compile and execute the merge sort successfully. The results for the above-given instances are as follows:

```
[namith03@grace1 HW3]$ cat HW3_Q1
List Size = 16, Threads = 2, error = 0, time (sec) =   0.2286, qsort_time =   0.0000
List Size = 16, Threads = 4, error = 0, time (sec) =   0.0052, qsort_time =   0.0000
List Size = 16, Threads = 8, error = 0, time (sec) =   0.0060, qsort_time =   0.0000
List Size = 1048576, Threads = 16, error = 0, time (sec) =   0.0274, qsort_time =   0.1693
List Size = 16777216, Threads = 256, error = 0, time (sec) =   0.4491, qsort_time =   3.4488
```

As required, error = 0 was reported for all the instances

2. (20 points) Plot speedup and efficiency for all combinations of k and q chosen from the following sets: k=12,20,28;q=0,1,2,4,6,8,10.Comment on how the results of your experiments align with or diverge from your understanding of the expected behaviour of the parallelized code.

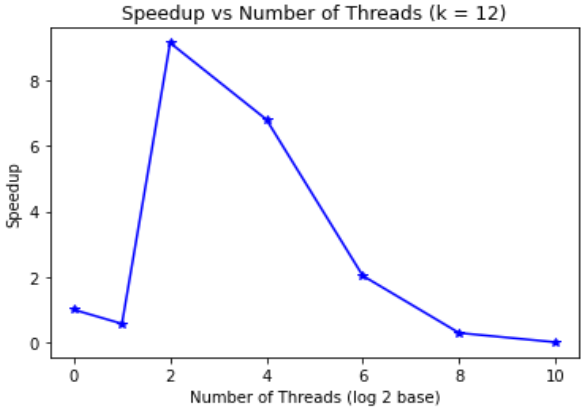
Solution:

The modified code is executed for the following scenarios

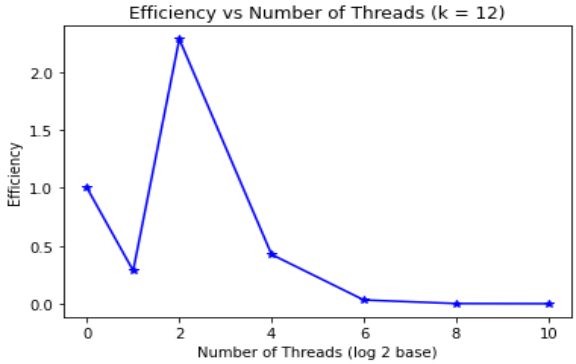
For k = 12 ; q = 0,1,2,4,6,8,10

```
[namith03@grace1 HW3]$ cat HW3.2.1
List Size = 4096, Threads = 1, error = 0, time (sec) =   0.0558, qsort_time =   0.0010
List Size = 4096, Threads = 2, error = 0, time (sec) =   0.0977, qsort_time =   0.0010
List Size = 4096, Threads = 4, error = 0, time (sec) =   0.0061, qsort_time =   0.0007
List Size = 4096, Threads = 16, error = 0, time (sec) =   0.0082, qsort_time =   0.0009
List Size = 4096, Threads = 64, error = 0, time (sec) =   0.0274, qsort_time =   0.0004
List Size = 4096, Threads = 256, error = 0, time (sec) =   0.1923, qsort_time =   0.0004
List Size = 4096, Threads = 1024, error = 0, time (sec) =   7.2339, qsort_time =   0.0004
```

Plot for Speedup vs Number of threads:



Plot for Efficiency vs Number of threads:

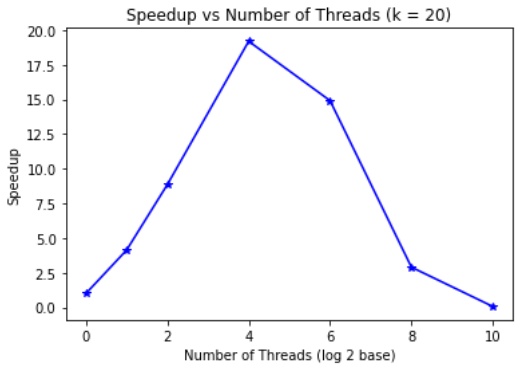


For k = 20 ; q = 0,1,2,4,6,8,10

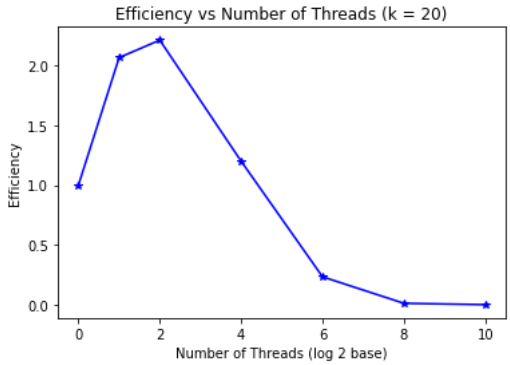
The results are as follows:

```
[namith03@grace1 HW3]$ cat HW3.2.2
List Size = 1048576, Threads = 1, error = 0, time (sec) =   0.5011, qsort_time =   0.1712
List Size = 1048576, Threads = 2, error = 0, time (sec) =   0.1212, qsort_time =   0.1691
List Size = 1048576, Threads = 4, error = 0, time (sec) =   0.0566, qsort_time =   0.1697
List Size = 1048576, Threads = 16, error = 0, time (sec) =   0.0261, qsort_time =   0.1701
List Size = 1048576, Threads = 64, error = 0, time (sec) =   0.0336, qsort_time =   0.2598
List Size = 1048576, Threads = 256, error = 0, time (sec) =   0.1729, qsort_time =   0.2602
List Size = 1048576, Threads = 1024, error = 0, time (sec) =   8.1282, qsort_time =   0.2040
```

Plot for Speedup vs Number of Threads:



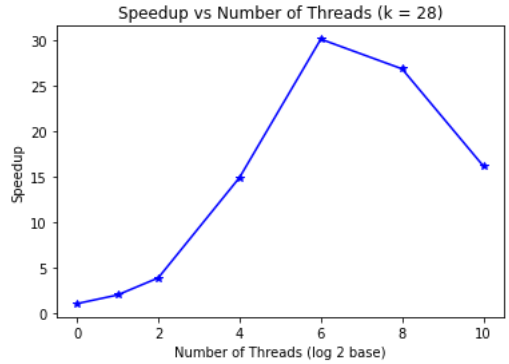
Plot for Efficiency vs Number of Threads:



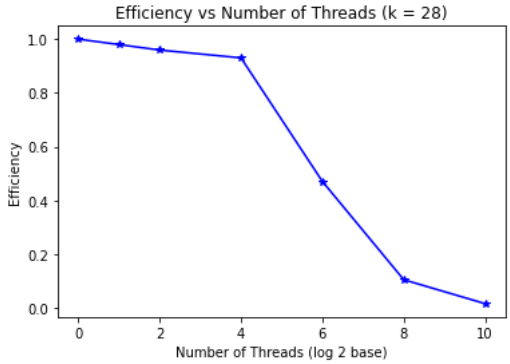
For k = 28 ; q = 0,1,2,4,6,8,10

The results are as follows:

Plot for Speedup vs Number of Threads:



Plot for Efficiency vs Number of Threads:



Observation: From the plots and results, it can be observed that speedup increases with an increase in the number of threads. But we can observe that after a certain number of threads, the Speedup starts to drop with the increment in the number of threads. The effect of parallelization in the code only is observed until a certain number of threads depending upon

the size of the list. Beyond a certain number of threads, the increase in the number of threads results in overhead associated with it which results in a decrement in the Speedup.

3. (10 points) For the instance with k = 28 and q = 5 experiments with different choices for OMP_PLACES and OMP_PROC_BIND to see how the parallel performance of the code is impacted. Explain your observations.

Solution:

OMP_PLACES = 'threads'

OMP_PROC_BIND=true

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =
2.1161, qsort_time =  51.6718
```

OMP_PLACES = 'threads'

OMP_PROC_BIND=master

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =
56.6969, qsort_time =  51.9165
```

OMP_PLACES = 'threads'

OMP_PROC_BIND=close

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =
2.0966, qsort_time =  49.5693
```

OMP_PLACES = 'threads'

OMP_PROC_BIND=spread

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =
2.1136, qsort_time =  51.8452
```

OMP_PLACES = 'cores'

OMP_PROC_BIND=true

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =
2.0786, qsort_time =  50.1452
```

OMP_PLACES = ‘cores’

OMP_PROC_BIND=master

```
List Size = 268435456, Threads = 32, error = 0, time (sec) = 56.0853, qsort_time =  51.4559
```

OMP_PLACES = ‘cores’

OMP_PROC_BIND=close

```
List Size = 268435456, Threads = 32, error = 0, time (sec) = 2.1300, qsort_time =  49.4142
```

OMP_PLACES = ‘cores’

OMP_PROC_BIND=spread

```
List Size = 268435456, Threads = 32, error = 0, time (sec) = 2.1187, qsort_time =  50.7423
```

OMP_PLACES = ‘socket’

OMP_PROC_BIND=true

```
List Size = 268435456, Threads = 32, error = 0, time (sec) = 2.1653, qsort_time =  51.3794
```

OMP_PLACES = ‘socket’

OMP_PROC_BIND=master

```
List Size = 268435456, Threads = 32, error = 0, time (sec) = 56.3392, qsort_time =  50.9575
```

OMP_PLACES = ‘socket’

OMP_PROC_BIND=close

```
List Size = 268435456, Threads = 32, error = 0, time (sec) = 2.0786, qsort_time =  49.0065
```

OMP_PLACES = ‘socket’

OMP_PROC_BIND=spread

```
List Size = 268435456, Threads = 32, error = 0, time (sec) = 2.0650, qsort_time =  47.5324
```

It can be observed that the execution time is more for OMP_PROC_BIND = master in comparison with the other options. Because when OMP_PROC_BIND = master is employed, all the threads in the team are assigned to places closer to the master thread (single thread). Hence this does not utilize much parallelization and results in more time or nearly equal time to the serialized sort which does not employ parallelization.

For other combinations of OMP_PROC_BIND and OMP_PLACES, all the threads are assigned parallelly in the system and hence parallelization is utilized here which results in much lesser time in comparison with the serial sort which does not employ parallelization.