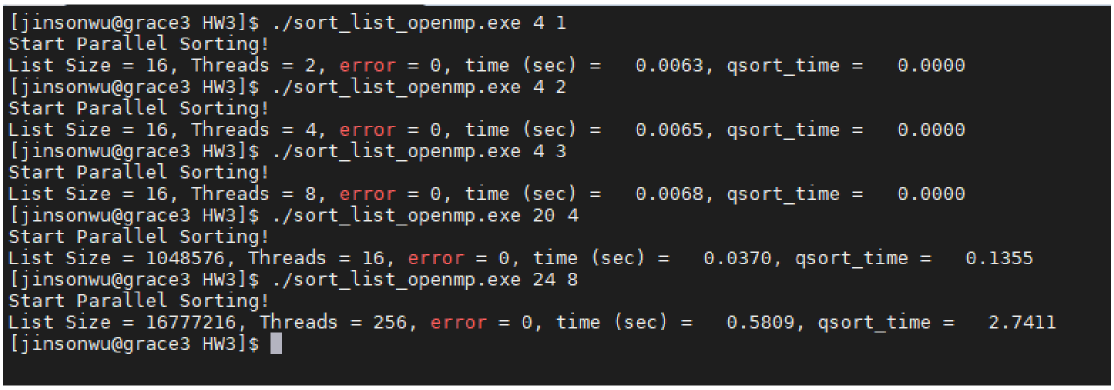
CSCE 735 Parallel Computing – HW3

832001192
Chun-Sheng Wu

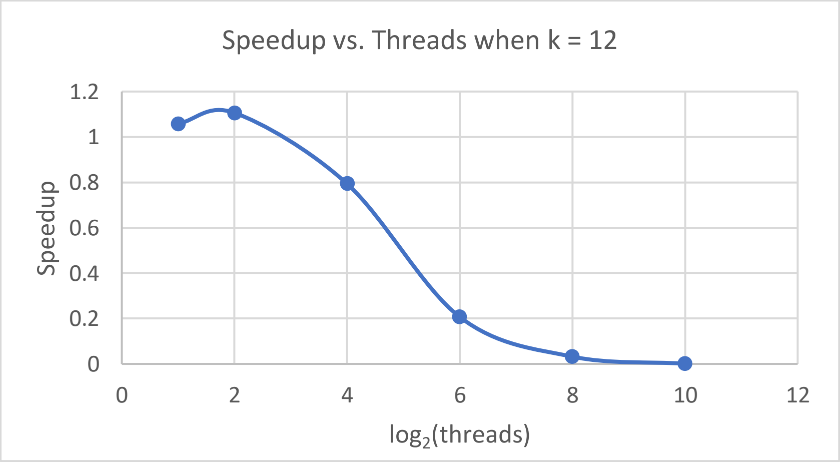
1. Revise the code and via OpenMP

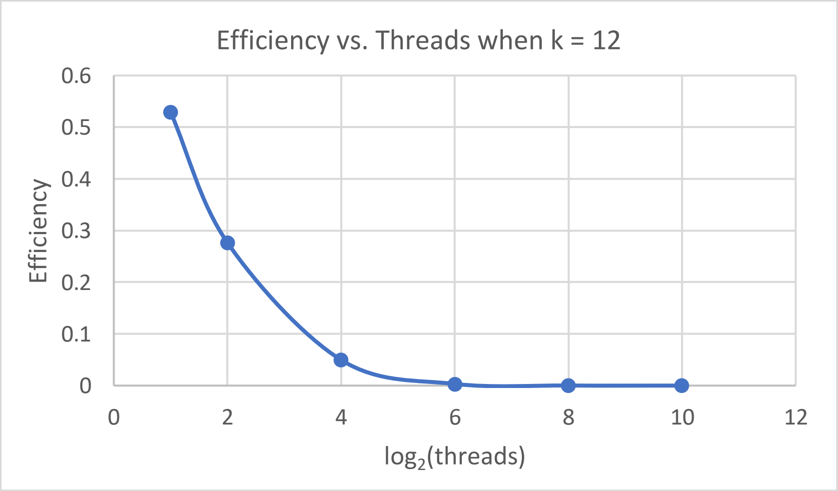
I modified the code with **#pragma omp parallel** region including one **#pragma omp single** to implicitly contain a barrier inside the main execution. I also set several **#pragma omp barrier** in the sorting subroutine to make sure data update and allocation works correctly.

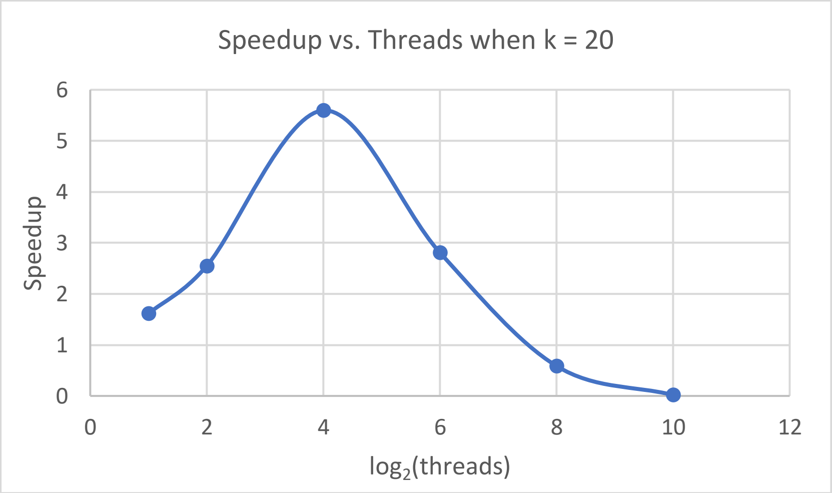


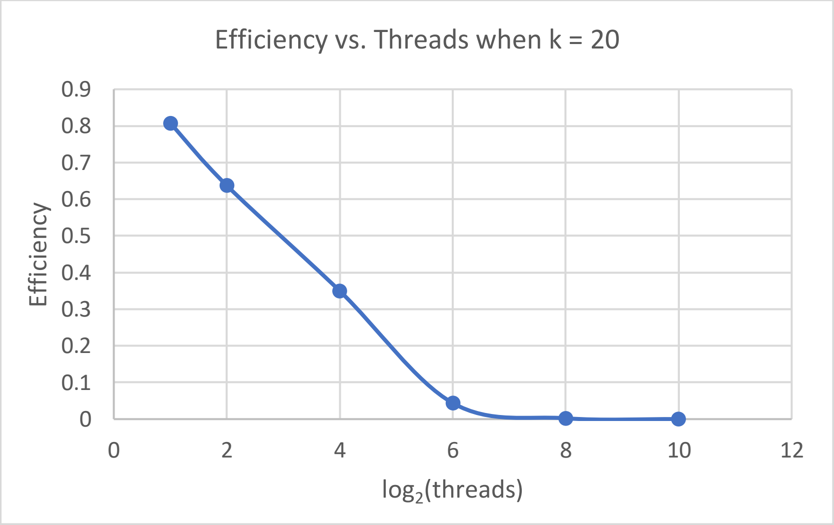
```
[jinsonwu@grace3 HW3]$ ./sort_list_openmp.exe 4 1
Start Parallel Sorting!
List Size = 16, Threads = 2, error = 0, time (sec) =   0.0063, qsort_time =   0.0000
[jinsonwu@grace3 HW3]$ ./sort_list_openmp.exe 4 2
Start Parallel Sorting!
List Size = 16, Threads = 4, error = 0, time (sec) =   0.0065, qsort_time =   0.0000
[jinsonwu@grace3 HW3]$ ./sort_list_openmp.exe 4 3
Start Parallel Sorting!
List Size = 16, Threads = 8, error = 0, time (sec) =   0.0068, qsort_time =   0.0000
[jinsonwu@grace3 HW3]$ ./sort_list_openmp.exe 20 4
Start Parallel Sorting!
List Size = 1048576, Threads = 16, error = 0, time (sec) =   0.0370, qsort_time =   0.1355
[jinsonwu@grace3 HW3]$ ./sort_list_openmp.exe 24 8
Start Parallel Sorting!
List Size = 16777216, Threads = 256, error = 0, time (sec) =   0.5809, qsort_time =   2.7411
[jinsonwu@grace3 HW3]$ 
```

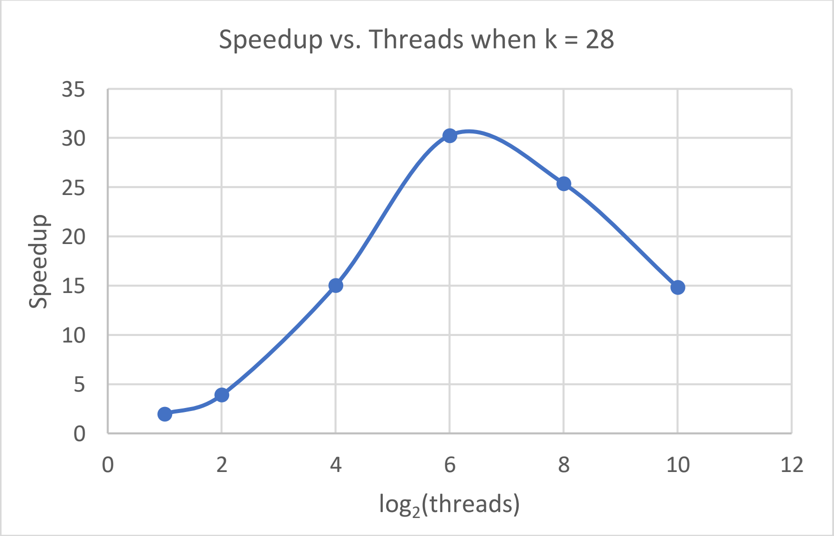
2. Plot speedup and efficiency with k = 12, 20, 28 along with q = 0, 1, 2, 4, 6, 8, 10 and comment

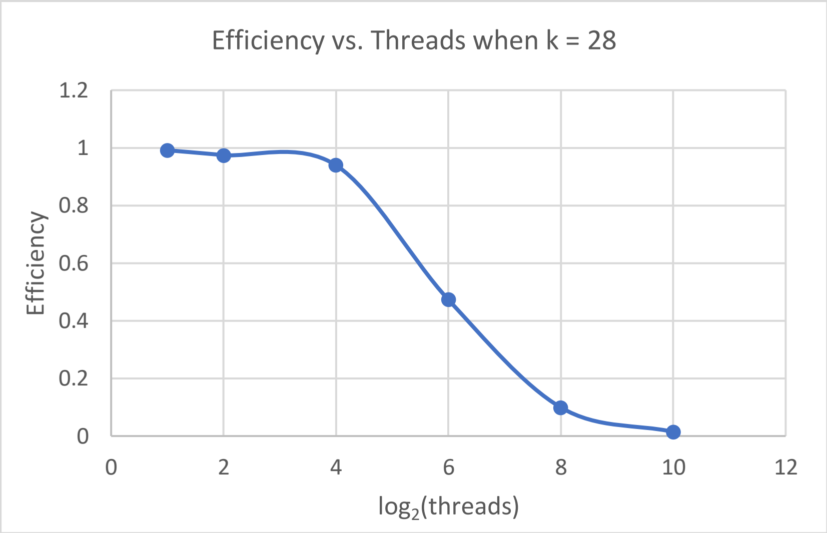





Speedup vs. Threads when k = 20
Speedup
6
5
4
3
2
1
0
0
2
4
6
8
10
12
log$_2$(threads)


Efficiency vs. Threads when k = 20
Efficiency
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
log$_2$(threads)


Speedup vs. Threads when k = 28
Speedup
35
30
25
20
15
10
5
0
0
2
4
6
8
10
12
log$_2$(threads)



From the graphs, we could observe that the speedup and efficiency did not behave well with k=12. This might be because the computing loading was not heavy enough to manifest the advantage of task parallelization.

As we increased k to 20, both values raised but still not reached the desired performance before we revoked all the hardware resource in a node, which was 48 cores. It achieved the highest speedup when utilizing 16 threads and dropped significantly after more threads were involved. Efficiency decreased almost linearly if we put more and more threads in the execution stage.

Curves met with our expectation while k was set to be 28. The speedup was close to the portion we attempted to be parallelized. Meanwhile, the efficiency kept flat with 1 before it employed all the 48 cores (q<6). The two values then went down quickly since there was thread stall happened by requesting more threads than it intrinsically had on the board.

In overall, the optimization of task parallelization took place when the computing loading was sufficient for the machine to be fully executed. Additionally, I realized that the speedup with OpenMP was slightly worse than with threads (thread_t). Guess that threads had more strict rules of thread modulation, yet OpenMP maintained the flexibility of thread revoking, which sacrificed a little bit speedup here. But in conclusion, these two mechanisms both obtained extraordinary optimization performing instructions in parallel.

3. Exhibit the experiment with different OMP_PLACES and OMP_PROC_BIND and show your observation.

Execution Time (s)	*PLACES = threads*	*PLACES = cores*	*PLACES = sockets*
BIND = master	67.6108	67.9025	3.9632
BIND = close	2.2234	2.2282	2.1853
BIND = spread	2.1961	2.1919	2.2015

Above is the table of execution time in different situations. Clearly could we see that it cost relatively considerable time if **OMP_PROC_BIND = master** in threads and cores. It implied that master thread/core might not have sufficient space to let multiple threads operating at the same time. Thus, they should wait and line up until others completed. However, the execution time in sockets did not have huge gap compared to other two bind strategies. Socket itself was a complex of threads/cores sharing the local memory, just like the grid structure in CUDA. Accordingly, it would not undergo severe performance loss working with only one socket since it could adopt other threads in the socket as well. Except for above-mentioned situations, other cases did not have significant discrepancies. Spread was slightly better than close in threads and cores due to well-aligned distribution with limited resources. Close performed slightly better than spread in sockets because of spatial advantage in higher hierarchy alignment.