

## CSCE 735 Spring 2023

### HW 3: Parallel Merge Sort Using OpenMP

Name: Shweta Sharma

UIN: 433003780

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`

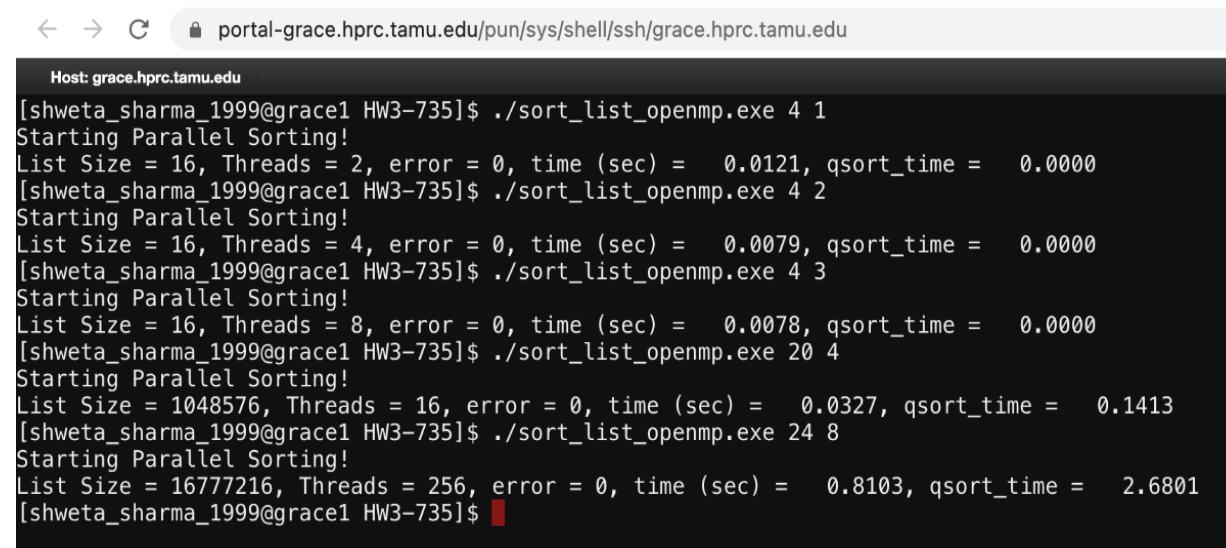
`./sort_list_openmp.exe 4 3`

`./sort_list_openmp.exe 20 4`

`./sort_list_openmp.exe 24 8`

Ans: I have revised the code to execute the merge sort in parallel via OpenMP. Attaching the screenshot here of the output of the code on Grace portal. I have added one parallel region by adding `#pragma omp parallel` directive, and I have put the code to sort the list in that region. Throughout the code to sort a sublist by a thread, i have added barriers by adding `#pragma omp barrier` directive to introduce some amount of synchronization.

Following are the output in The Grace portal:



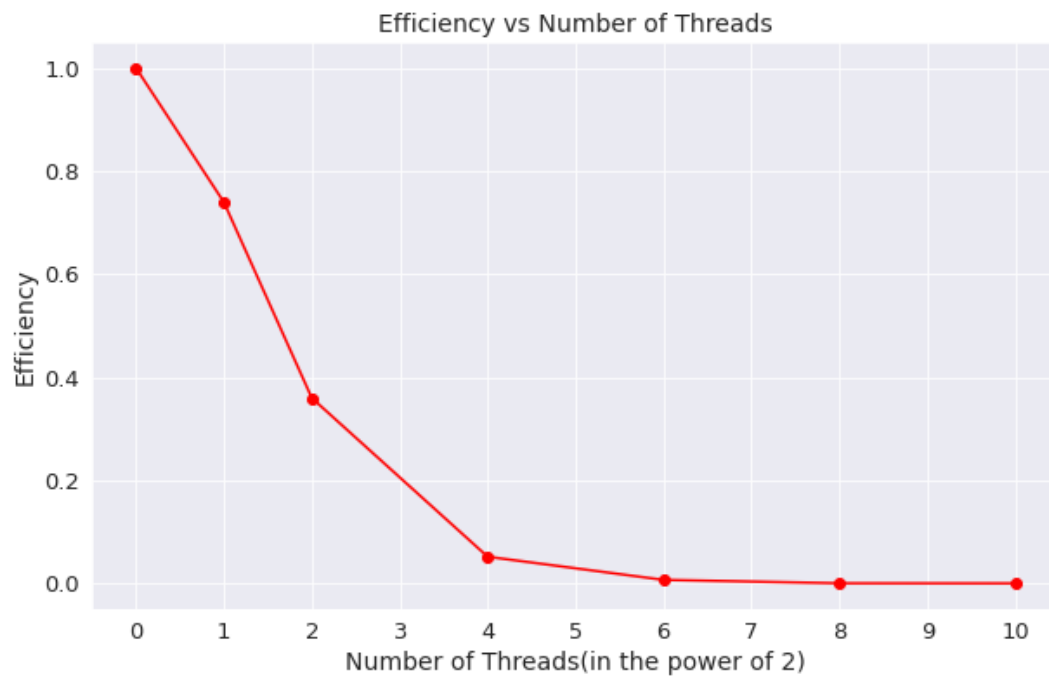
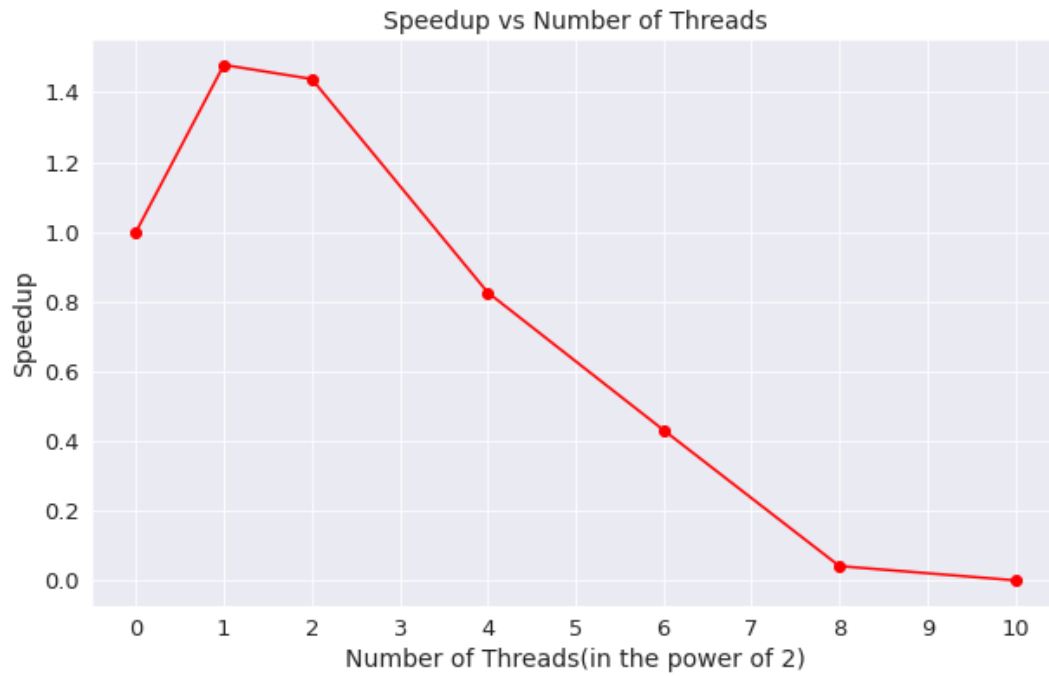
```
portal-grace.hprc.tamu.edu/pun/sys/shell/ssh/grace.hprc.tamu.edu
Host: grace.hprc.tamu.edu
[shweta_sharma_1999@grace1 HW3-735]$ ./sort_list_openmp.exe 4 1
Starting Parallel Sorting!
List Size = 16, Threads = 2, error = 0, time (sec) = 0.0121, qsort_time = 0.0000
[shweta_sharma_1999@grace1 HW3-735]$ ./sort_list_openmp.exe 4 2
Starting Parallel Sorting!
List Size = 16, Threads = 4, error = 0, time (sec) = 0.0079, qsort_time = 0.0000
[shweta_sharma_1999@grace1 HW3-735]$ ./sort_list_openmp.exe 4 3
Starting Parallel Sorting!
List Size = 16, Threads = 8, error = 0, time (sec) = 0.0078, qsort_time = 0.0000
[shweta_sharma_1999@grace1 HW3-735]$ ./sort_list_openmp.exe 20 4
Starting Parallel Sorting!
List Size = 1048576, Threads = 16, error = 0, time (sec) = 0.0327, qsort_time = 0.1413
[shweta_sharma_1999@grace1 HW3-735]$ ./sort_list_openmp.exe 24 8
Starting Parallel Sorting!
List Size = 16777216, Threads = 256, error = 0, time (sec) = 0.8103, qsort_time = 2.6801
[shweta_sharma_1999@grace1 HW3-735]$
```

All the runs executed successfully with zero error.

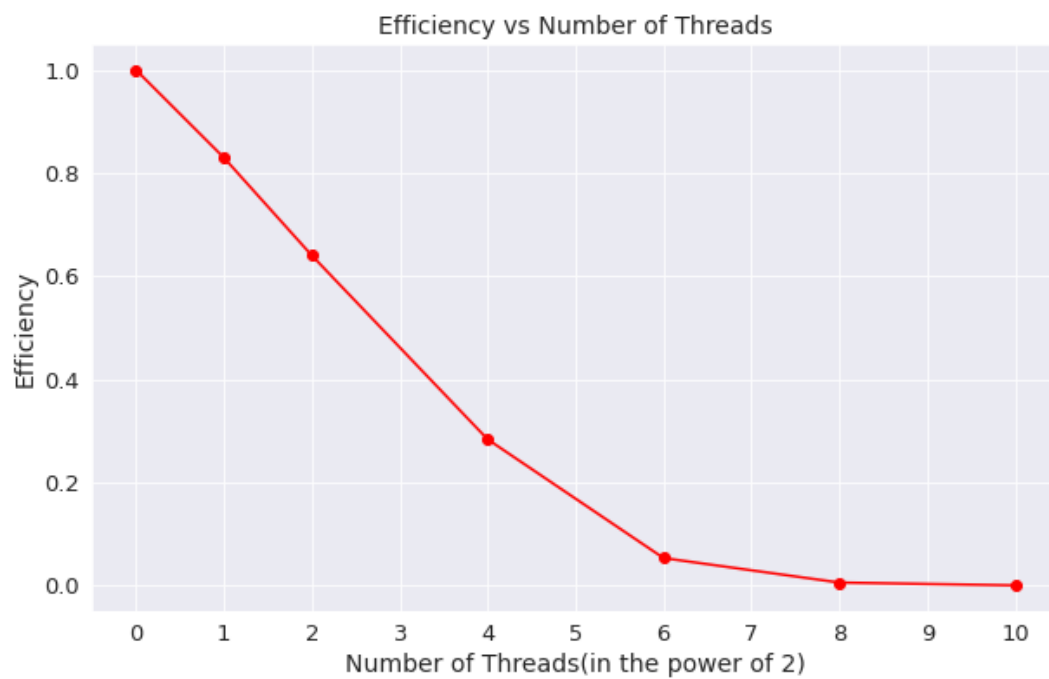
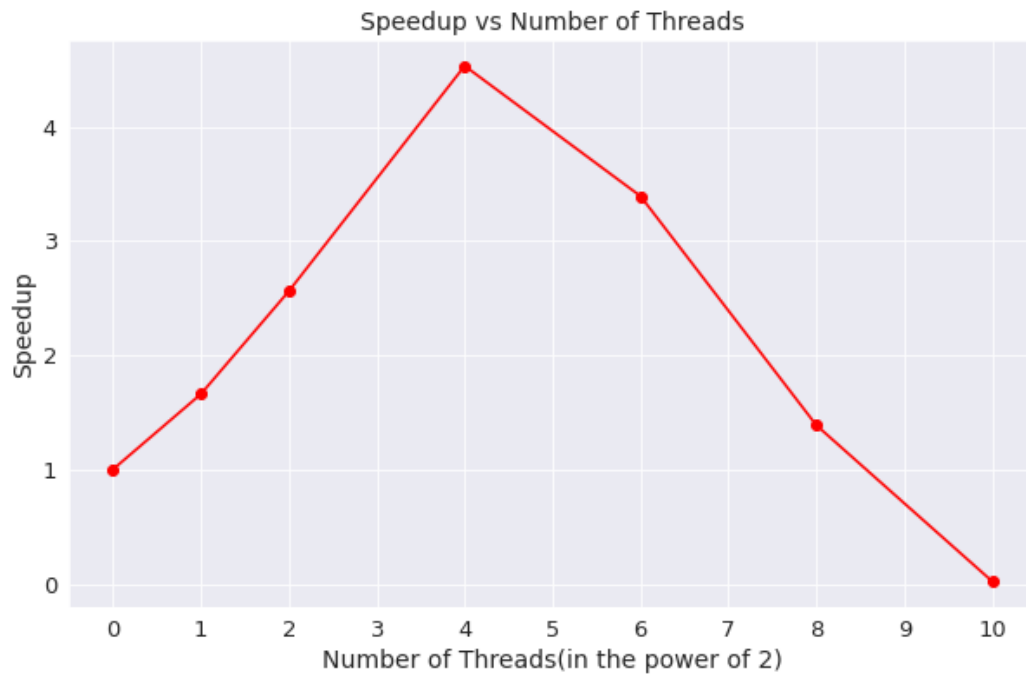
2. (20 points) Plot speedup and efficiency for all combinations of k and q chosen from the following sets: k = 12, 20, 28 ; q = 0, 1, 2, 4, 6, 8, 10. Comment on how the results of your experiments align with or diverge from your understanding of the expected behavior of the parallelized code.

Ans:

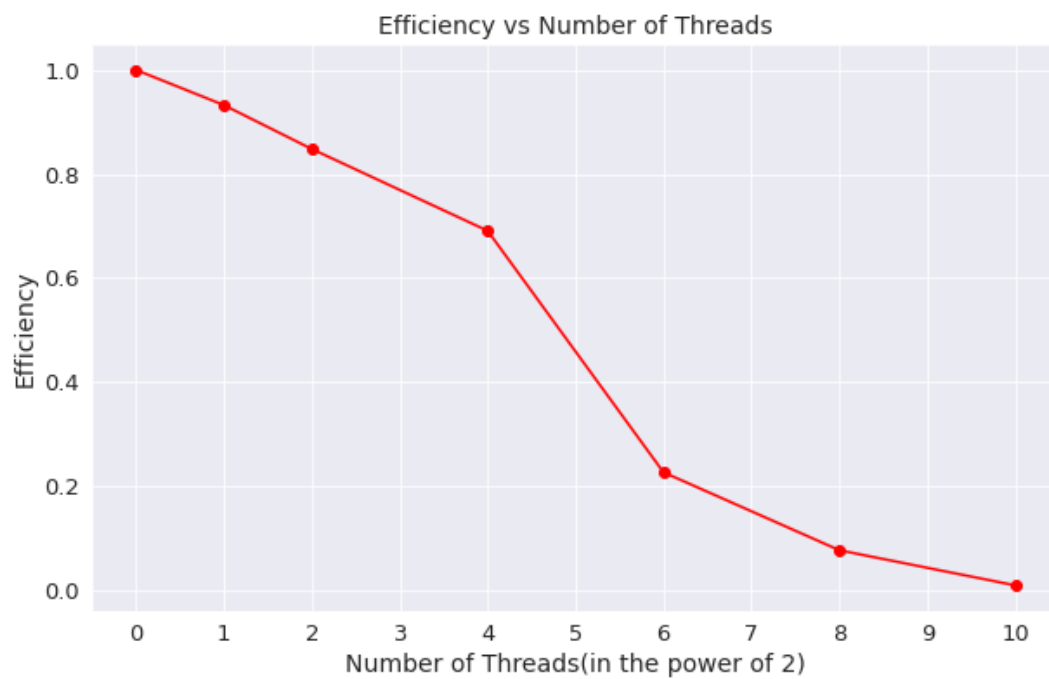
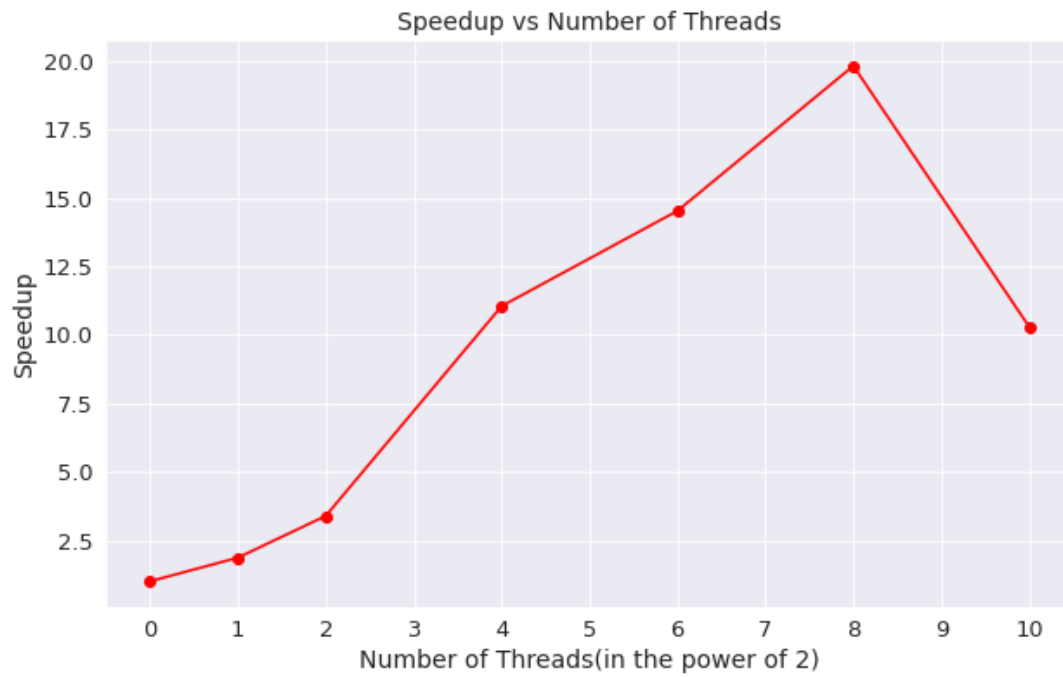
For  $k=12$ , we have the following curves:



For  $k=20$ , we have the following curves:



For  $k=28$ , we have the following curves:



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 resources in a node, which was 48 cores. It achieved the highest speedup when utilizing 16 threads and dropped significantly when 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 close to 1 before it employed all the 48 cores ( $q < 6$ ). The two values then went down quickly since a 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. My guess is that threads had more strict rules of thread modulation, yet OpenMP maintained the flexibility of thread revoking, which sacrificed a little bit of speedup here. But in conclusion, these two mechanisms both obtained extraordinary optimization performing instructions in parallel.

3. (10 points) For the instance with  $k = 28$  and  $q = 5$  experiment with different choices for OMP\_PLACES and OMP\_PROC\_BIND to see whether the parallel performance of the code is impacted, and if so, how the performance impacted. Explain your observations.

Ans:

Execution Time(s)	OMP_PLACES=threads	OMP_PLACES=cores	OMP_PLACES=sockets
OMP_PROC_BIND = master	56.009	55.0541	3.8716
OMP_PROC_BIND = close	2.1425	2.1358	2.1142
OMP_PROC_BIND = spread	2.1499	2.1286	2.1541

Above is the table of execution time in different situations. Clearly could we see that it takes relatively higher time if OMP\_PROC\_BIND = master in threads and cores. It therefore implied that the master thread/core might not have sufficient space to let multiple threads operate at the same time. Thus, they should wait and line up until others are completed. However, the execution time in sockets did not have a huge gap compared to the 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.