CSCE 735 Fall 2022

Name: Rohan Chaudhury
UIN: 432001358

HW 3: Parallel Merge Sort Using OpenMP

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

ANSWER:

Following are the logs obtained from executing the above mentioned commands:

List Size = 16, Threads = 2, error = 0, time (sec) = 0.0059, qsort_time = 0.0000

List Size = 16, Threads = 4, error = 0, time (sec) = 0.0063, qsort_time = 0.0000

List Size = 16, Threads = 8, error = 0, time (sec) = 0.0066, qsort_time = 0.0000

List Size = 1048576, Threads = 16, error = 0, time (sec) = 0.0261, qsort_time = 0.1715

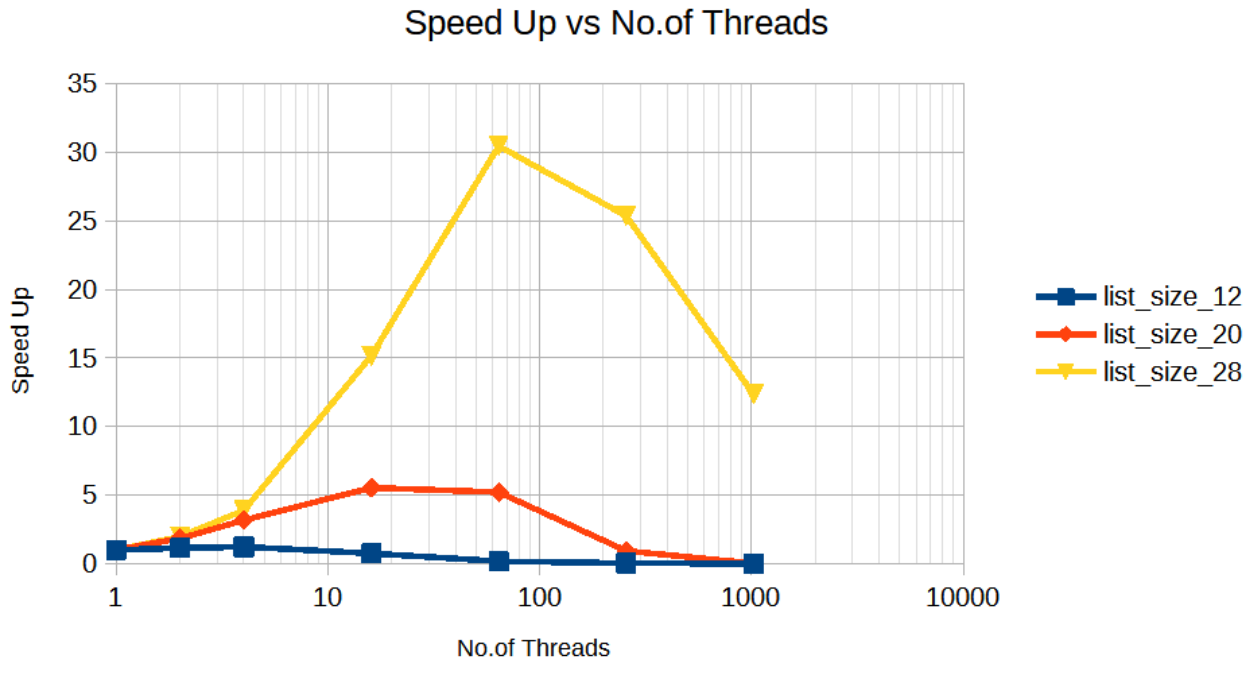
List Size = 16777216, Threads = 256, error = 0, time (sec) = 0.5213, qsort_time = 3.4749

List Size	16	Threads	2	error	0	time (sec)	0.0059	qsort_time	0
List Size	16	Threads	4	error	0	time (sec)	0.0063	qsort_time	0
List Size	16	Threads	8	error	0	time (sec)	0.0066	qsort_time	0
List Size	1048576	Threads	16	error	0	time (sec)	0.0261	qsort_time	0.1715
List Size	16777216	Threads	256	error	0	time (sec)	0.5213	qsort_time	3.4749

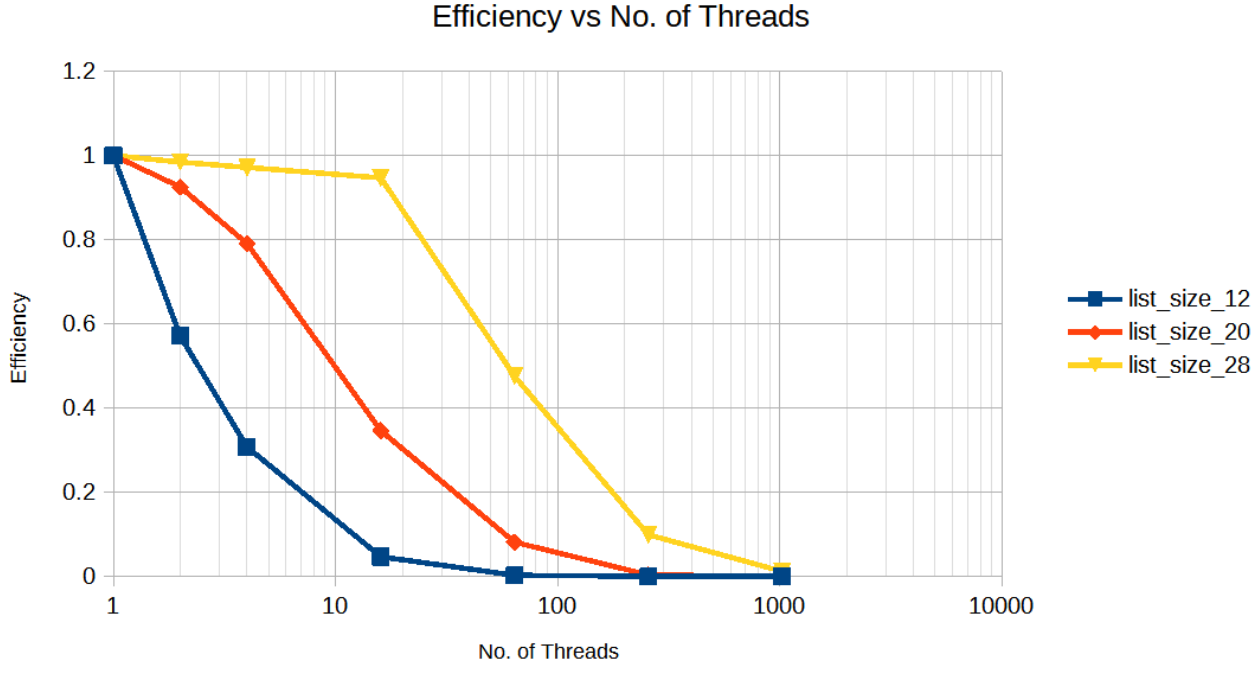
As can be observed, the error returned for all the aforementioned tests is 0. Furthermore, for smaller list sizes, parallelization has no advantage over single threaded execution. The anticipated speedup is noticeable over single threaded processing for large list sizes 2^20 and 2^24.

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.

ANSWER:



Plot: Speed Up vs No. of threads for k in [12, 20, 28]



Plot: Efficiency vs No. of threads for k in [12, 20, 28]

Data obtained from experiments for k = 12

List Size	4096	Threads	1	error	0	time (sec)	0.008	qsort_time	0.001	Speedup	1	Efficiency	1
List Size	4096	Threads	2	error	0	time (sec)	0.007	qsort_time	0.001	Speedup	1.142857143	Efficiency	0.571428571
List Size	4096	Threads	4	error	0	time (sec)	0.0065	qsort_time	0.0007	Speedup	1.230769231	Efficiency	0.307692308
List Size	4096	Threads	16	error	0	time (sec)	0.0107	qsort_time	0.0007	Speedup	0.747663551	Efficiency	0.046728972
List Size	4096	Threads	64	error	0	time (sec)	0.0458	qsort_time	0.0004	Speedup	0.174672489	Efficiency	0.002729258
List Size	4096	Threads	256	error	0	time (sec)	0.2179	qsort_time	0.0004	Speedup	0.036714089	Efficiency	0.000143414
List Size	4096	Threads	1024	error	0	time (sec)	8.7422	qsort_time	0.0006	Speedup	0.000915101	Efficiency	8.93654E-07

Data obtained from experiments for k = 20

List Size	1048576	Threads	1	error	0	time (sec)	0.1828	qsort_time	0.1704	Speedup	1	Efficiency	1
List Size	1048576	Threads	2	error	0	time (sec)	0.0988	qsort_time	0.1718	Speedup	1.850202429	Efficiency	0.925101215
List Size	1048576	Threads	4	error	0	time (sec)	0.0578	qsort_time	0.1722	Speedup	3.162629758	Efficiency	0.790657439
List Size	1048576	Threads	16	error	0	time (sec)	0.033	qsort_time	0.1724	Speedup	5.539393939	Efficiency	0.346212121
List Size	1048576	Threads	64	error	0	time (sec)	0.0351	qsort_time	0.1729	Speedup	5.207977208	Efficiency	0.081374644
List Size	1048576	Threads	256	error	0	time (sec)	0.1963	qsort_time	0.2646	Speedup	0.931227713	Efficiency	0.003637608
List Size	1048576	Threads	1024	error	0	time (sec)	8.3809	qsort_time	0.2632	Speedup	0.0218115	Efficiency	2.13003E-05

Data obtained from experiments for k = 28

List Size	268435456	Threads	1	error	0	time (sec)	62.5407	qsort_time	62.5141	Speedup	1	Efficiency	1
List Size	268435456	Threads	2	error	0	time (sec)	31.7764	qsort_time	62.4964	Speedup	1.968149318	Efficiency	0.984074659
List Size	268435456	Threads	4	error	0	time (sec)	16.0932	qsort_time	62.6693	Speedup	3.886156886	Efficiency	0.971539222
List Size	268435456	Threads	16	error	0	time (sec)	4.1292	qsort_time	62.5148	Speedup	15.14596048	Efficiency	0.94662253
List Size	268435456	Threads	64	error	0	time (sec)	2.054	qsort_time	62.5975	Speedup	30.44824732	Efficiency	0.475753864
List Size	268435456	Threads	256	error	0	time (sec)	2.4663	qsort_time	62.7916	Speedup	25.35810729	Efficiency	0.099055107
List Size	268435456	Threads	1024	error	0	time (sec)	5.0651	qsort_time	63.3243	Speedup	12.34737715	Efficiency	0.012057985

In the graphs shown above (plotted using the data obtained), the blue, red, and yellow lines represent speed up versus thread count across different values of k, that is, the size of the list to sort. We can see that when compared to k = 20 (red line) or k = 28 (yellow line) the speed up tends to be non-existent at the lowest value of k, that is, k = 12 (blue line). This is to be expected since when the input size/data to be operated on is small, using multiple threads will increase execution time due to increased thread management overhead cost in comparison to single threaded execution. This is the circumstance in which the thread management overhead is almost equivalent to the time required to complete the task in a single threaded environment. However, as k increases, single threaded execution takes substantially longer time. So in this scenario, the thread management overhead times in the parallelization implementations will not cause the parallelized execution times to outweigh single threaded execution times. We can also observe similar kind of behaviour for k = 28 and k = 20 in the above plots, where increasing the number of threads improves speed up until an optimum value is reached, beyond which speed up begins to decrease because of the aforementioned thread management overheads.

From the plots, we can also observe that as the value of k (size of the list) increases, the optimum speed up is achieved at greater thread counts. We can explain this with the concepts of work division. The number of work units that a thread can process has an optimal value at a certain point, beyond which it may need many context switches or substantial memory accesses to handle its tasks. This scenario results in longer execution times. Also, if the number of work units to be processed by the threads is below this optimal work unit load for the threads, the CPU will spend longer times on thread handling instead of executing the task of the thread. On the basis of this we can expect that higher thread counts would result in optimal performance for larger list sizes.

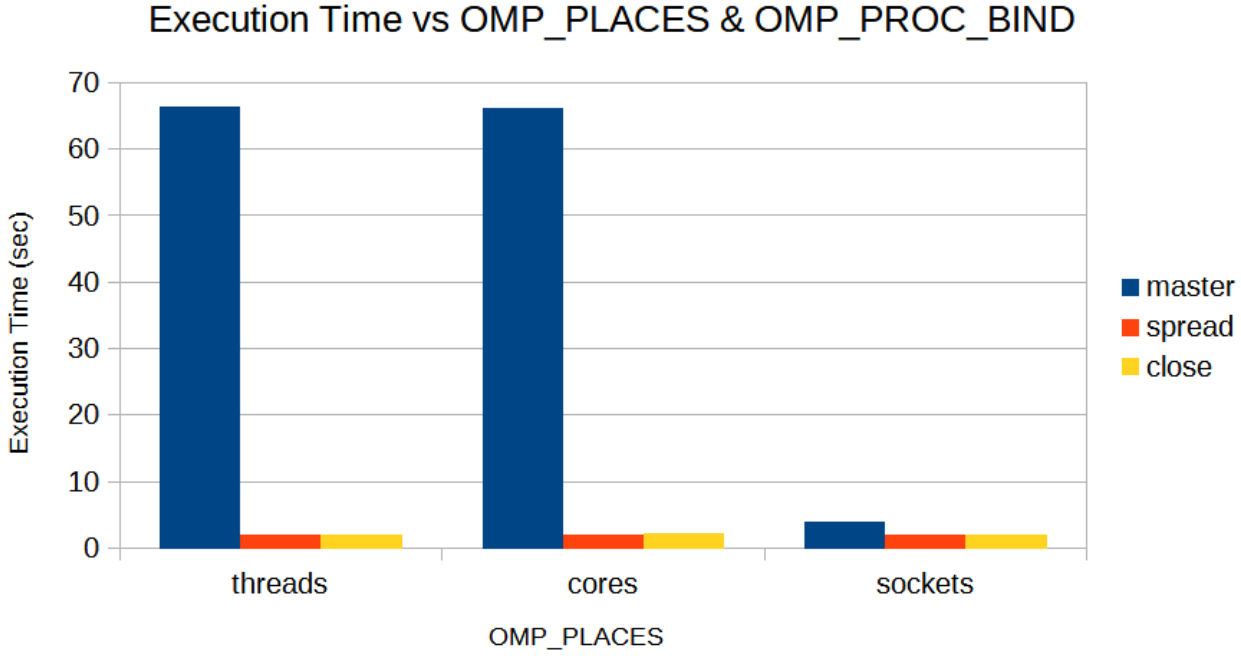
Also, from the efficiency vs. number of threads plot, we can see that the efficiency declines as the number of threads increases. This is caused because of thread synchronization and thread contention over computational tasks.

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 how the parallel performance of the code is impacted. Explain your observations.

ANSWER:

The table below shows the results of the experiments with k = 28 and q = 5 and with different choices for OMP_PLACES and OMP_PROC_BIND to see how the parallel performance of the code is impacted.

OMP_PROC_BIND	FALSE	OMP_PLACES	N/A	List Size	268435456	Threads	32	error	0	time (sec)	2.8711	qsort_time	62.9875
OMP_PROC_BIND	master	OMP_PLACES	threads	List Size	268435456	Threads	32	error	0	time (sec)	66.4989	qsort_time	63.278
OMP_PROC_BIND	master	OMP_PLACES	cores	List Size	268435456	Threads	32	error	0	time (sec)	66.1356	qsort_time	62.8494
OMP_PROC_BIND	master	OMP_PLACES	sockets	List Size	268435456	Threads	32	error	0	time (sec)	3.9329	qsort_time	62.6182
OMP_PROC_BIND	spread	OMP_PLACES	threads	List Size	268435456	Threads	32	error	0	time (sec)	2.1386	qsort_time	62.6808
OMP_PROC_BIND	spread	OMP_PLACES	cores	List Size	268435456	Threads	32	error	0	time (sec)	2.1368	qsort_time	62.6591
OMP_PROC_BIND	spread	OMP_PLACES	sockets	List Size	268435456	Threads	32	error	0	time (sec)	2.1381	qsort_time	62.6594
OMP_PROC_BIND	close	OMP_PLACES	threads	List Size	268435456	Threads	32	error	0	time (sec)	2.1645	qsort_time	62.6461
OMP_PROC_BIND	close	OMP_PLACES	cores	List Size	268435456	Threads	32	error	0	time (sec)	2.1714	qsort_time	62.6575
OMP_PROC_BIND	close	OMP_PLACES	sockets	List Size	268435456	Threads	32	error	0	time (sec)	2.1421	qsort_time	62.5838



Plot: Execution time with variations in OMP_PLACES and OMP_PROC_BIND

From the above plots we can see that when OMP PLACES is set to either 'Threads' or 'Cores', the execution times are significantly higher for OMP_PROC_BIND= "master" (as seen in the first and second cluster from the left). When OMP_PROC_BIND is set to 'Master,' the whole group of logical threads will be scheduled in the same location as the master thread. Because a core in Grace portal has only 1 hardware thread, setting OMP_PLACES to either 'Cores' or 'Threads' with OMP_PROC_BIND set to 'Master' would result in the same performance. When OMP_PLACES is set to 'sockets,' the execution time for the identical OMP_PROC_BIND

(="master") configuration is substantially lower than for when OMP_PLACES is set to either 'Threads' or 'Cores'. This is because a socket in Grace portal has 24 hardware threads available, allowing it to execute many logical threads in parallel and thus improving the performance.

In general, if the application is operating on shared data sets, setting OMP_PROC_BIND to 'Close' will improve performance and if the application operates on disjoint groups of data, setting OMP_PROC_BIND to 'Spread' would result in higher performance. According to the figure above, the execution time between 'Spread' and 'Close' does not change much for all the 3 values of OMP PLACES. This is because, in the program the threads are initially run on disjoint sets of data, but as iterations go, more threads begin to work on shared bigger chunks of data, finally culminating to sharing one massive list. As a result, the software is neither working decisively on separate sets of data nor totally working on shared sets of data. Because of this ambiguity, both 'Spread' and 'Close' would produce similar outcomes because either setup will assist the other part of the merge sort process, that is, either the preliminary portion in which each thread works on the smallest subsets or the final portion in which each thread works on the shared set of larger subsets in the main list.