CSCE 735 Homework 3

Name: Pranav Anantharam
UIN: 734003886

1. The revised code (OpenMP-based parallel merge sort implementation) produced the below output for varying list sizes and thread counts:

Spreadsheet Image:

List Size	Number of Threads	Error	Time (in seconds)	Qsort Time (in seconds)
16	2	0	0.4848	0
16	4	0	0.0062	0
16	8	0	0.007	0
1048576	16	0	0.027	0.1782
16777216	256	0	0.5128	3.5706

The error count was reported to be zero for all scenarios tested.

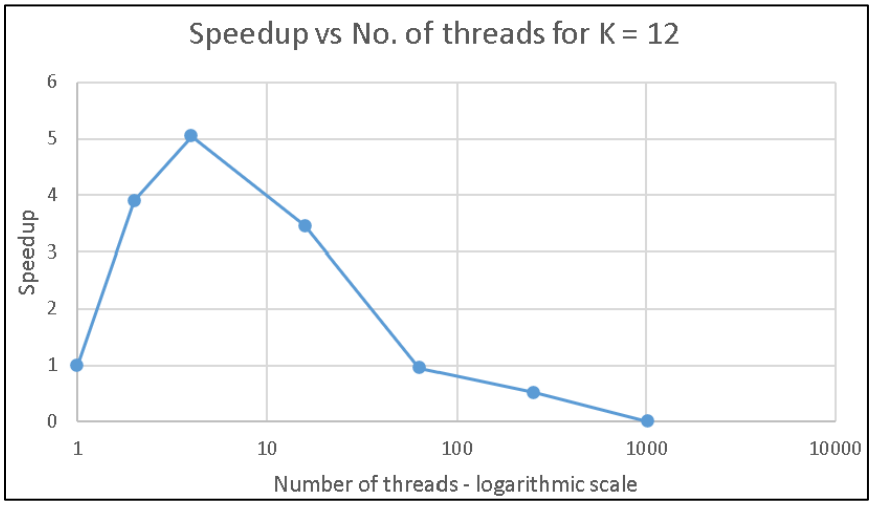
---

2. The plots for speedup and efficiency for varying values of k and q are given below:

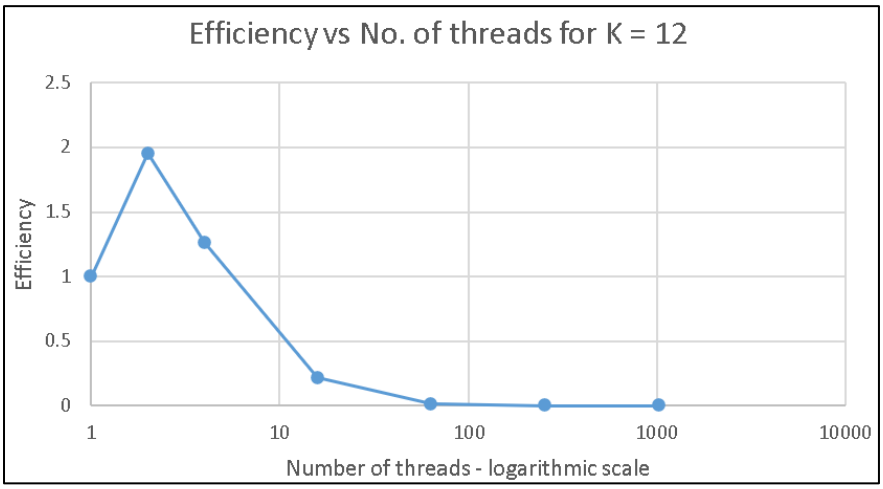
Spreadsheet Calculations:

List Size	Number of Threads	Error	Time (in seconds)	Qsort Time (in seconds)	Speedup	Efficiency
4096	1	0	0.0328	0.0009	1	1
4096	2	0	0.0084	0.001	3.9047619	1.95238095
4096	4	0	0.0065	0.001	5.04615385	1.26153846
4096	16	0	0.0095	0.0007	3.45263158	0.21578947
4096	64	0	0.0346	0.0004	0.94797688	0.01481214
4096	256	0	0.0635	0.0004	0.51653543	0.00201772
4096	1024	0	7.2394	0.0004	0.00453076	4.4246E-06
1048576	1	0	0.191	0.1779	1	1
1048576	2	0	0.1042	0.1774	1.83301344	0.91650672
1048576	4	0	0.0598	0.1775	3.19397993	0.79849498
1048576	16	0	0.0265	0.1777	7.20754717	0.4504717
1048576	64	0	0.0436	0.1778	4.38073394	0.06844897
1048576	256	0	0.2364	0.2977	0.80795262	0.00315606
1048576	1024	0	0.4131	0.2686	0.46235778	0.00045152
268435456	1	0	64.7976	64.7778	1	1
268435456	2	0	32.9627	64.8189	1.96578557	0.98289278
268435456	4	0	16.6634	64.9346	3.88861817	0.97215454
268435456	16	0	4.3026	64.7751	15.0601032	0.94125645
268435456	64	0	2.1007	65.3585	30.8457181	0.48196435
268435456	256	0	2.3488	65.4665	27.5875341	0.1077638
268435456	1024	0	4.4109	65.5471	14.6903353	0.01434603

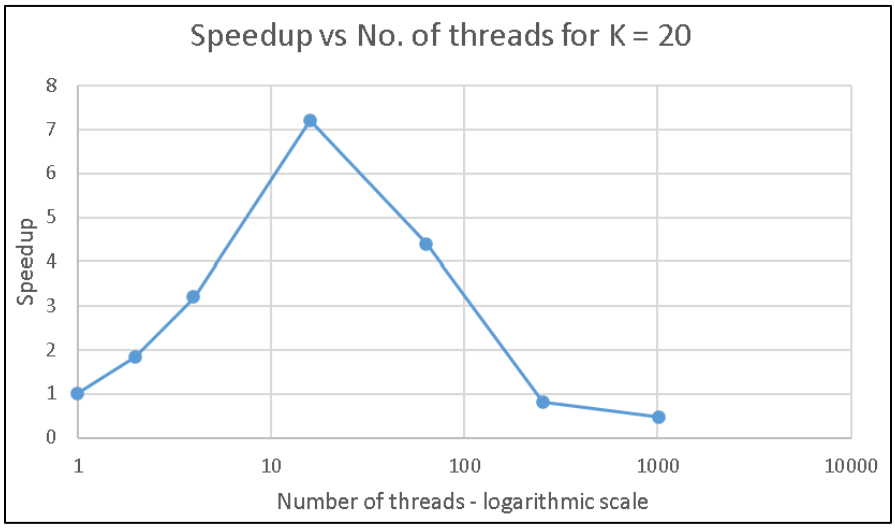
Speedup vs No. of threads for List Size = 4096 elements



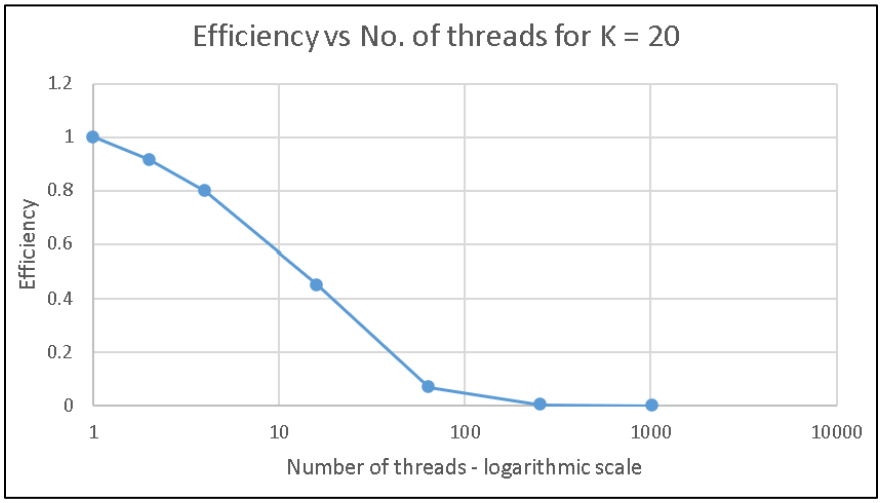
Efficiency vs No. of threads for List Size = 4096 elements



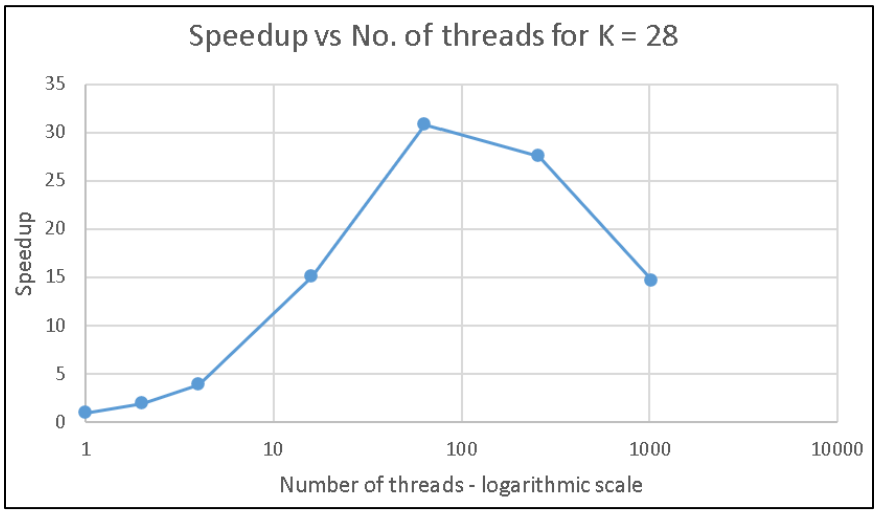
Speedup vs No. of threads for List Size = 1048576 elements



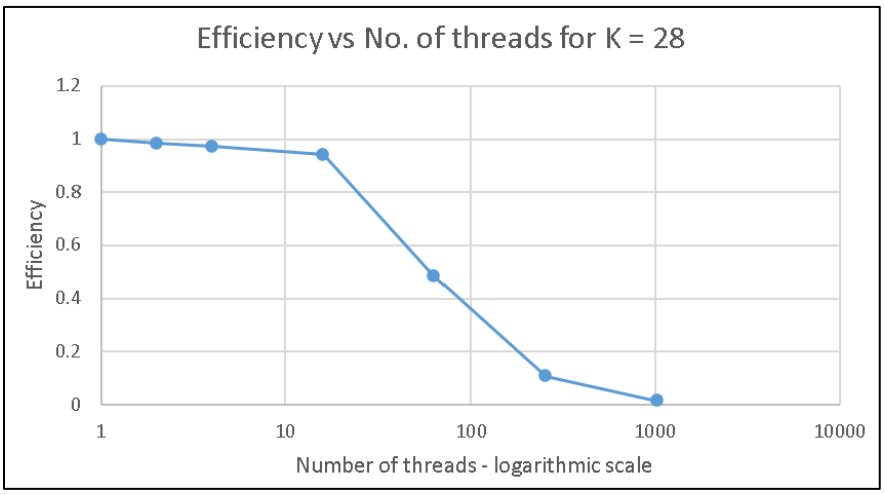
Efficiency vs No. of threads for List Size = 1048576 elements



Speedup vs No. of threads for List Size = 268435456 elements



Efficiency vs No. of threads for List Size = 268435456 elements



The results of the experiment align with my understanding of the expected behavior of the parallelized code.

For small values of K (K=12), the parallel merge sort implementation may take longer to perform sorting compared to the serial merge sort implementation. This can be attributed to the significant overhead involved in scheduling threads in comparison to the time duration taken to perform the sorting. Moreover, there is a significant decrease in efficiency observed as the number of threads increases. For K = 12 and number of threads = 4, we observe good performance where the speedup was calculated to be **5.0461** and efficiency was **1.261**.

For larger values of K (K = 20, 28), speedup increases the number of threads increases. This is expected to happen as more threads will be able to complete more work in parallel and the large list size allows for more parallelism. But the speedup decreases beyond the optimal number of threads. This occurs since the overhead associated with managing threads becomes significant as the number of threads increases.

Moreover, the increase in the number of threads for a constant list size displays a decrease in efficiency. This is since the amount of work assigned to each thread reduces with an increase in the number of threads. And so, the overhead involved in switching threads becomes significant, which results in a decrease in efficiency. Overall, to achieve maximum efficiency, a balance between work assigned to each thread and number of threads is essential.

---

3. The results of experiments for the instance with k = 28 and q = 5 for different choices for OMP_PLACES and OMP_PROC_BIND is given below:

a) OMP_PLACES = "threads"

- OMP_PROC_BIND = "master"

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =  69.3812,
qsort_time =  65.4311
```

- OMP_PROC_BIND = "close"

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =   2.2653,
qsort_time =  64.9583
```

- OMP_PROC_BIND = "spread"

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =   2.2308,
qsort_time =  65.3229
```

b) OMP_PLACES = "cores"

- OMP_PROC_BIND = "master"

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =  69.2167,
qsort_time =  65.0814
```

- OMP_PROC_BIND = "close"

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =   2.2649,
qsort_time =  64.8782
```

- OMP_PROC_BIND = "spread"

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =   2.2393,
qsort_time =  64.9248
```

c) OMP_PLACES = "sockets"

- OMP_PROC_BIND = "master"

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =   4.0433,
qsort_time =  64.8690
```

- OMP_PROC_BIND = "close"

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =   2.2403,
qsort_time =  65.2588
```

- OMP_PROC_BIND = "spread"

```
List Size = 268435456, Threads = 32, error = 0, time (sec) =   2.2362,
qsort_time =  64.8422
```

Using the OMP_PLACE environment variable, the location where all threads will bound to is defined. The options available are – **threads** (single thread of hardware), **cores** (single core that runs all threads) and **socket** (single socket can contains one or more cores).

Using the OMP_PROC_BIND environment variable, the binding policy for threads distributed across the locations specified using OMP_PLACE is defined. The options available are – **master** (all threads are bound to same place as master thread), **close** (all threads are bound to successive places close to place of master thread) and **spread** (all threads will be distributed among places evenly).

From the experiments conducted, the following observations were made:

The highest execution times are observed when OMP_PROC_BIND is set to "master". All the threads will be bound to the same place as master thread, and so the available resources and memory for execution are limited, resulting in higher execution time and large unused cache. When OMP_PLACE is set to ‘threads’, only a single thread is available for execution of the program, hence resulting in the highest execution time of **69.3812 seconds**. The execution time improves when OMP_PLACE is set to ‘cores’ and it improves further when OMP_PLACE is set to ‘socket’ as the program is allowed to make use of more hardware resources for execution.

When OMP_PROC_BIND is set to ‘close’, we observe faster execution times. This is because all threads are bound close to the location where the master thread runs, which allows for faster sharing of data and improves communication. Furthermore, we observe an increase in execution time as we modify the OMP_PLACE variable to ‘cores’ and ‘threads’, since it limits the hardware resources available to execute the program.

When OMP_PROC_BIND is set to ‘spread’, we observe the fastest execution times. This is because the threads are evenly distributed across the hardware leading to optimal usage of cache.