The platform used for this question is **COE System**. Below are the system details for **COE:**

Model name	Intel(R) Xeon(R) CPU X5650
Frequency	2.67GHz
Vendor ID	GenuineIntel
CPU cores	24
Architecture	x86_64
OS	CentOS Linux release 7.4.1708 (Core)
Cache	L1 data cache: 32K L1 instruction cache: 32K L2 cache: 256K L3 cache: 12288K
RAM	47.2 GB
CPU op-mode(s)	32-bit, 64-bit
Thread(s) per core	2
Core(s) per socket	6
Socket(s):	2
NUMA node(s)	2
CPU MHz	2660.023
BogoMIPS	5320.04
Virtualization	VT-x

For this question, I have implemented a parallel version of the Merge Sort algorithm to sort 5000 random integers ranging from 1 - 100,000. Default optimization was used while compiling.

Overview:

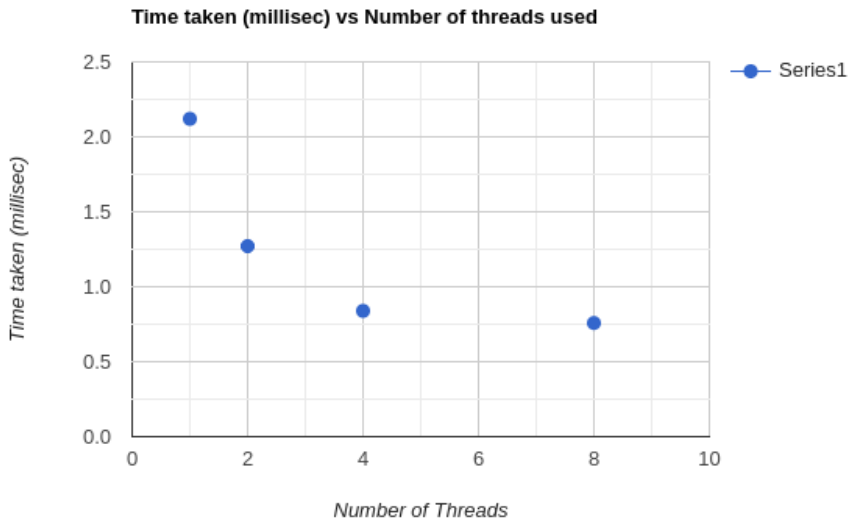
- Below is the flow of the algorithm implemented
    1. Divide the array equally into n (number of threads) and allocate to each thread based on thread ID.
    2. Perform merge sort locally for each allocated array.
    3. Once all allocated arrays are sorted, merge them to get the final sorted array.

Part a:

- I have run the program with 1, 2, 4 and 8 threads. The program was run 5 times for each configuration. The time taken to execute the program was calculated by taking the minimum time across these 5 runs.

Number of Threads	Time taken (milli seconds)
1	2.120
2	1.271
4	0.839
8	0.758

- Below graph shows the reduction in time taken to run the program as we increase the number of threads. As the number of threads increased, the difference between consecutive time taken reduced. As the number of threads increases, the time taken for the following tasks increases.
    - To spawn all the pthreads and join them
    - Time taken to merge all the locally sorted arrays
    - Time taken to create new array and copy elements while merging



Part b:

- I faced difficulties while merging the locally sorted arrays to get the final sorted array. I had to spend some time coming up with an algorithm which is generic to the number of threads used.
- I also faced issues when I was trying to pass more than one argument to the function *thread_merge_sort* which is called by *pthread_create*. Finally I created a struct (containing multiple attributes) and passed the struct as a single argument. I was able to compile and run the code, but when the number of threads were greater than 1, the answers were wrong. So I had to declare some variables as global variables as a work around.

Part c:

- **Strong Scaling:**

  For strong scaling, I kept the number of integers fixed(5000) and varied the number of threads from 1 to 256. Below table contains time taken to run the program for each configuration. The program was run 5 times for each configuration. The time taken to execute the program was calculated by taking the minimum time across these 5 runs.

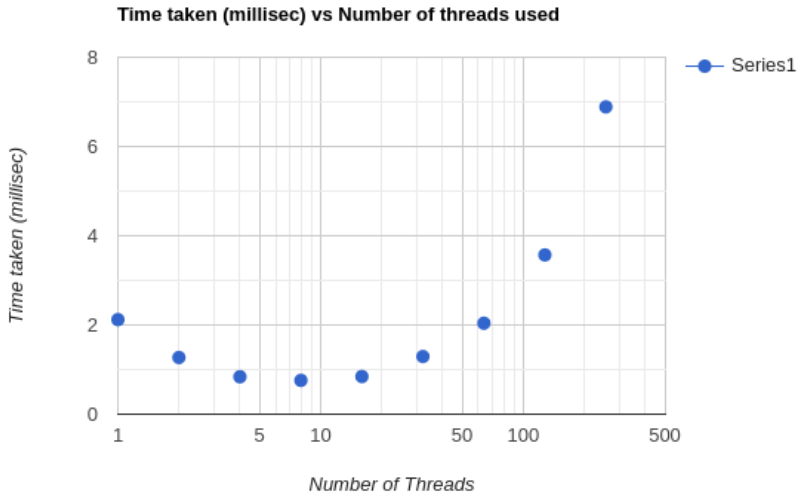
  T1 = time taken to run a program using 1 thread

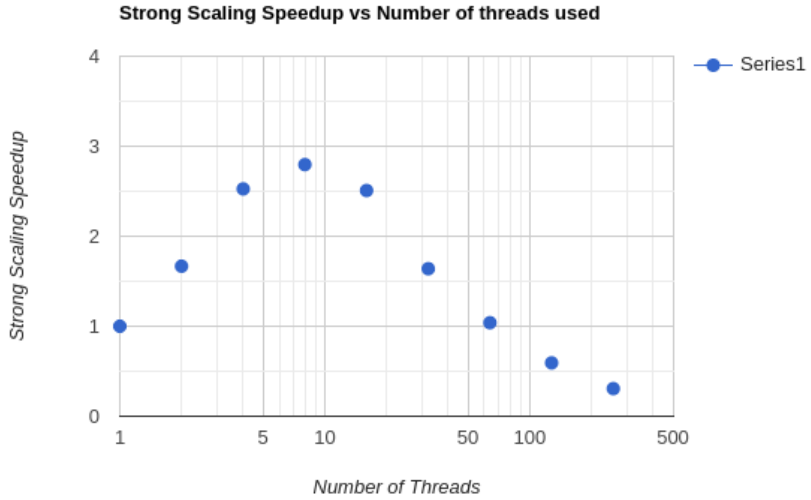
TN = time taken to run a program using N thread

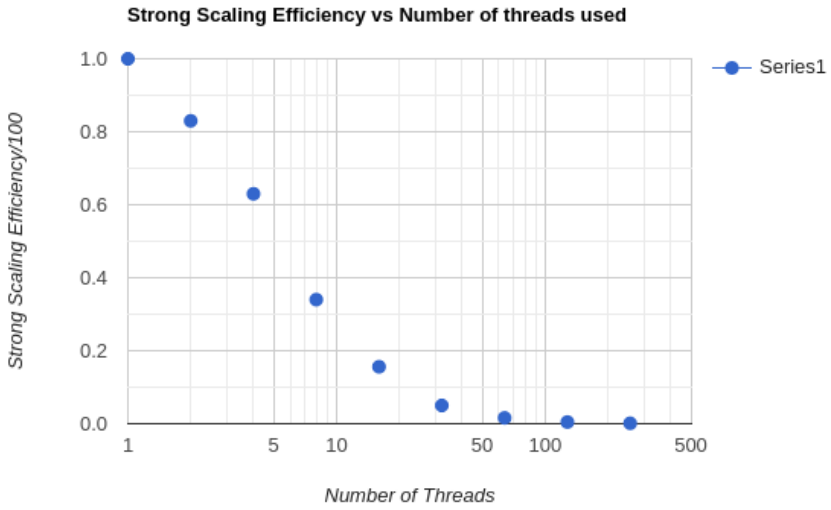
Strong scaling speedup (S) = T1/TN
Strong scaling efficiency (E) = (S/N)*100%

Number of threads	Time taken (milli seconds)	Strong scaling speedup (S)	Strong scaling efficiency (E)
1	2.120	1	100%
2	1.271	1.667	83.39%
4	0.839	2.526	63.17%
8	**0.758**	**2.796**	**34.96%**
16	0.845	2.508	15.6%
32	1.293	1.639	5.12%
64	2.039	1.039	1.62%
128	3.5723	0.593	0.46%
256	6.888	0.307	0.12%

- Below graph shows the variation in time taken to run the program as we increase the number of threads. We get the maximum speedup when num_threads = 8. The efficiency reduces with increase in the number of threads. For num_threads greater than 8, the time taken to run the program increases as num_threads increases. As the number of threads increases, the time taken for the following tasks increases significantly.
    - To spawn all the pthreads and join them
    - Time taken to merge all the locally sorted arrays
    - Time taken to create new array and copy elements while merging


Time taken (millisec) vs Number of threads used
Series1
Time taken (millisec)
8
6
4
2
0
1
5
10
50
100
500
Number of Threads


Strong Scaling Speedup vs Number of threads used
Series1
Strong Scaling Speedup
4
3
2
1
0
1
5
10
50
100
500
Number of Threads



- **Weak Scaling:**

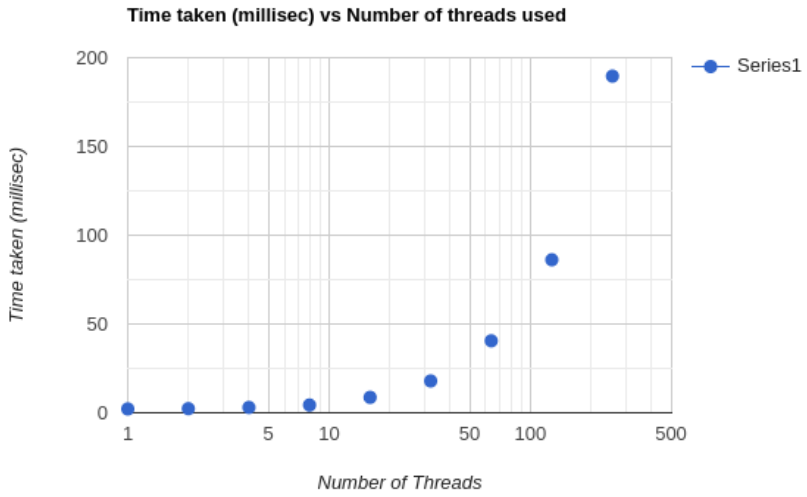
  For weak scaling, I doubled the number of integers and the number of threads simultaneously where the num_threads varied 1-256. Below table contains time taken to run the program for each configuration. The program was run 5 times for each configuration. The time taken to execute the program was calculated by taking the minimum time across these 5 runs.

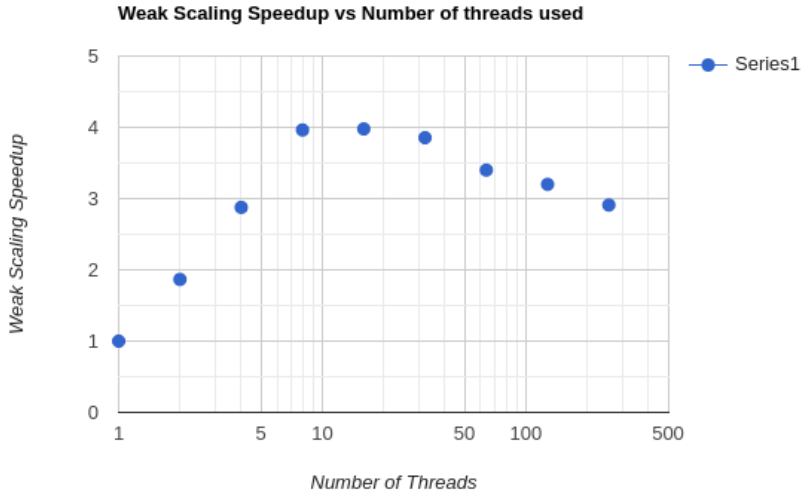
  T1 = time taken to run a program whose input size is X using 1 thread
  TN = time taken to run a program whose input size is N * X using N thread

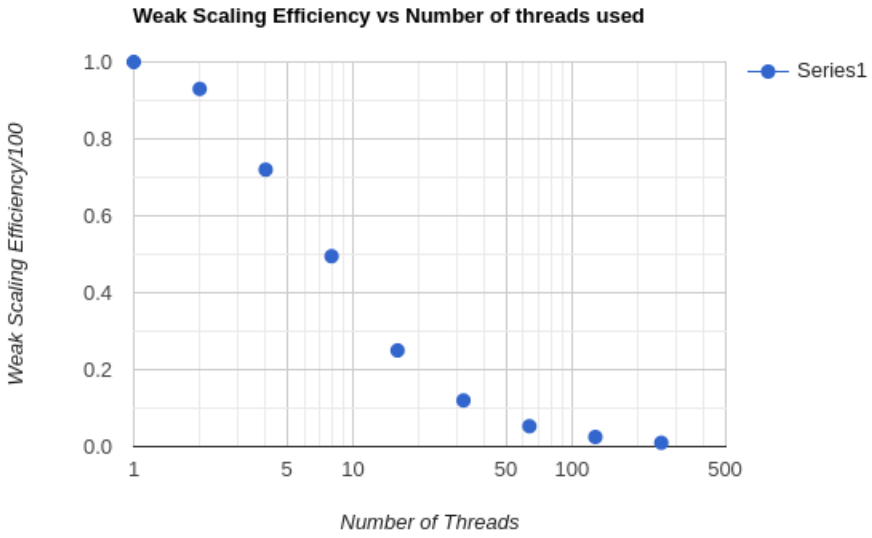
  Weak Scaling Speedup(S) = N*(T1/TN)
  Weak Scaling Efficiency(E) = (T1/TN)*100%

Number of threads	Input size	Time Taken(milli seconds)	Weak Scaling Speedup	Weak Scaling Efficiency
1	5000	2.159	1	100%
2	10000	2.314	1.866	93.30%
4	20000	3.001	2.877	71.94%
8	40000	4.358	3.963	49.54%
16	**80000**	**8.683**	**3.978**	**24.86%**
32	160000	17.920	3.855	12.04%
64	320000	40.536	3.4	5.32%
128	640000	86.176	3.2	2.50%
256	1280000	189.618	2.91	1.1%

- Below graph shows the variation in time taken to run the program as we increase the number of threads and the input size. We get the maximum speedup when num_threads = 16. For the number of threads greater than 16, speedup reduces .
- Usually, there is a decrease in efficiency with the increasing number of processors. This decrease is because as the problem size increases with the increase in the number of processors, more data needs to be communicated between the processors. This increases the latency in communication for the limited bandwidth of the communication network.


Time taken (millisec) vs Number of threads used
Series1
200
150
100
50
0
Time taken (millisec)
1
5
10
50
100
500
Number of Threads


Weak Scaling Speedup vs Number of threads used
Series1
5
4
3
2
1
0
Weak Scaling Speedup
1
5
10
50
100
500
Number of Threads



Reference:

1. Few code samples were taken from here: https://malithjayaweera.com/2019/02/parallel-merge-sort/