North Carolina State University

Department of Electrical and Computer Engineering

ECE 506: Fall 2019

Project 1: Parallel Radix Sort using OpenMP

by

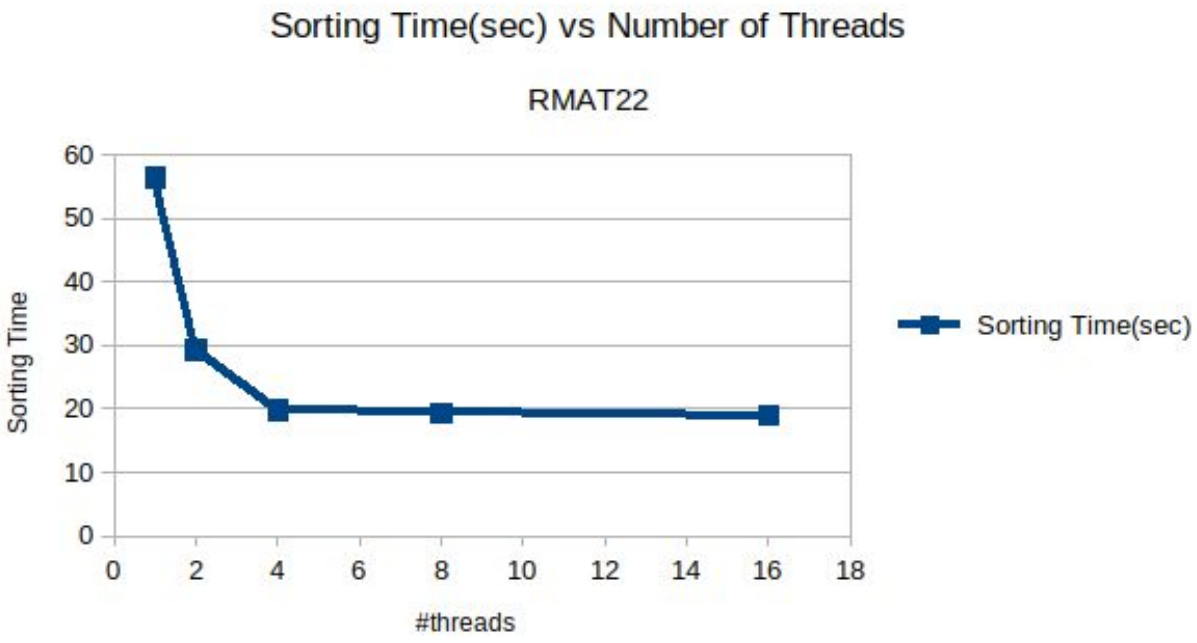
RHUSHIKESH ANAND PRABHUNE

(NCSU ID: 200321627)

Approach:

A serial radix sort algorithm was implemented first by modifying the existing count sort algorithm. The difference was that an extra loop needed to be present in radix sort to ensure sorting numbers with multiple digits. The LSB's of the numbers were used to sort the dataset for the first iteration. The next iteration used the sorted list in the first iteration to sort the list according to the next digit in the number. The final result was a fully sorted list.

The parallel approach used OpenMP constructs to split parts of the program in different number of threads. Loops were exploited for their ability to be parallelized and thus different iterations of the loops were run in parallel to decrease the overall sort time of the program. The program was run with different number of threads and the results were compared. Different datasets were used to compare the sorting times of the program running on 1,2,4,8,16 threads respectively. The serial version of the program was also used and the results compared with the parallel version. Synchronization techniques were used to avoid false sharing and thereby improve the sort time of the program.

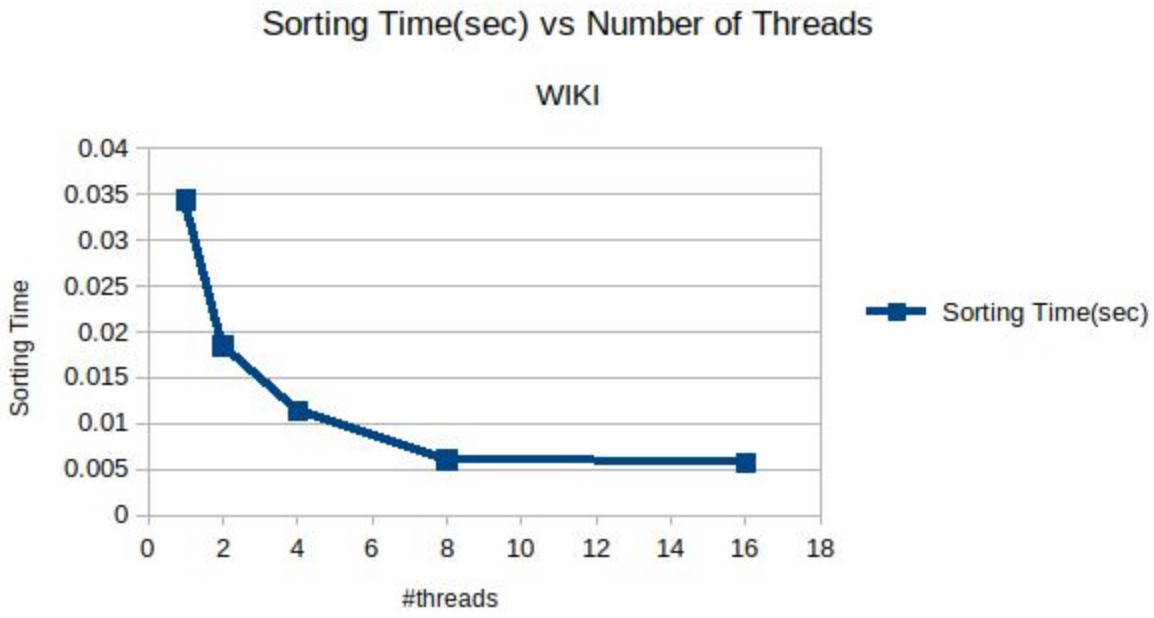


Edges: 67108864 Vertices: 4194303

RMAT22

#threads	1	2	4	8	16
Sorting Time(sec)	56.476551	29.335076	19.846938	19.383128	18.991421

Speedup(numthread=1 compared to numthread=16) = 56.476551/18.991421 = 2.97
The speedup does not change much for 4 and 8 threads since there is a limit to which a program can be parallelized. Therefore, increasing the number of threads after a certain number causes a bottleneck and hence does not improve the system performance. Using more threads than needed will just increase the overhead and therefore can increase the sort time of the algorithm. However, in this data set, many more threads can be used to increase the speedup because of the large number of data in the set.
Speedup(numthread=1 compared to numthread=2) = 56.476551/29.335076 = 1.92
The sort time of a serial radix sort can be lesser than the parallel radix sort because of the increased overhead due to parallel constructs and the bottleneck conditions mentioned above.

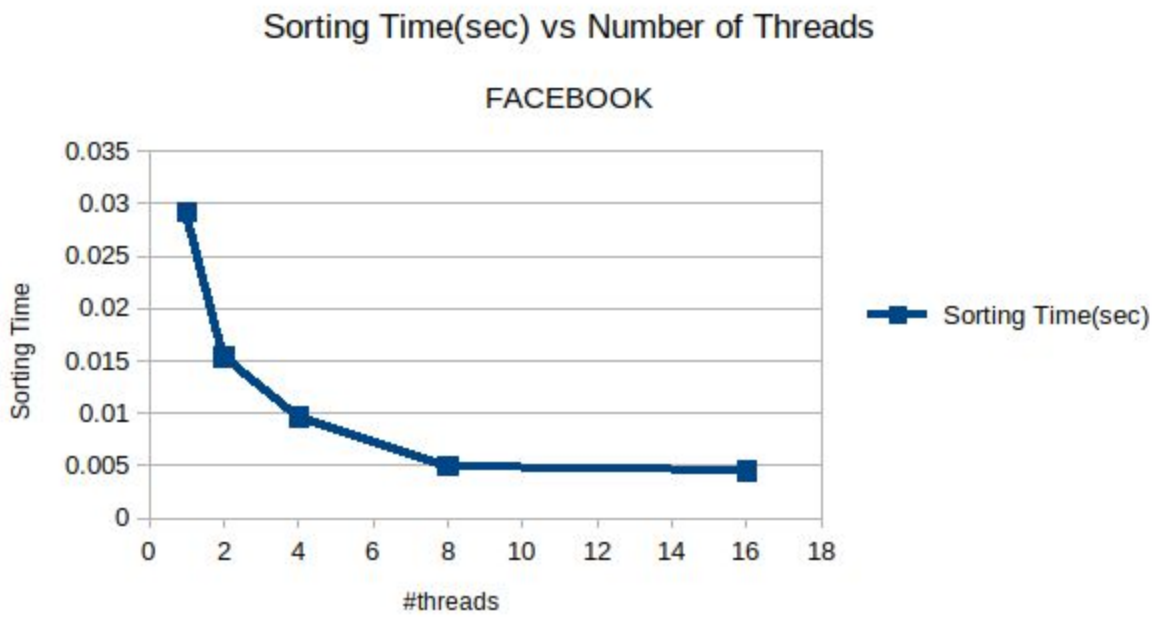


Edges: 103689 Vertices: 8298

WIKI

#threads	1	2	4	8	16
Sorting Time(sec)	0.034382	0.018473	0.011422	0.00606	0.005756

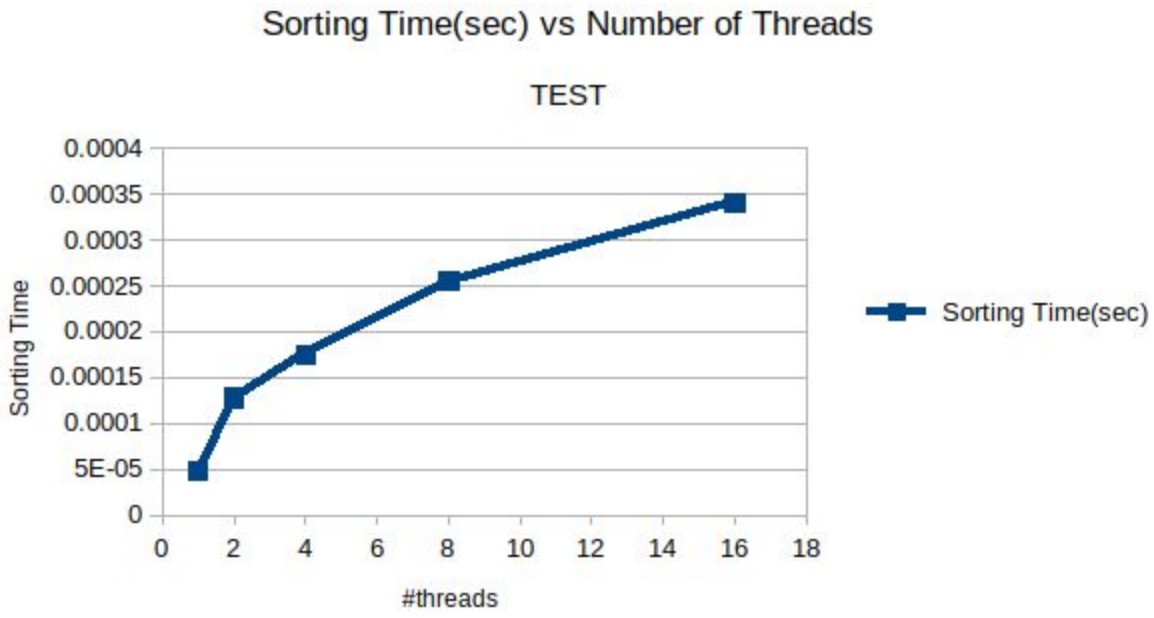
Speedup(numthread=1 compared to numthread=16) =0.034382/0.005756 = 5.97
Speedup(numthread=1 compared to numthread=2) = 0.034382/0.018473 = 1.86
Since the data held in the dataset is large, parallel programming helps reduce the sort time which is seen from the speedup. Larger the data set, larger the time required to complete the task alone. But if split in two, the halved time is also a big number which is what is exploited by parallel programming. Now if we further go on splitting the work, the time keeps splitting but the difference in the time required to complete the task goes on decreasing which is seen as a bottleneck in multi-threading.



Edges: 88234 Vertices: 4039

FACEBOOK					
#threads	1	2	4	8	16
Sorting Time(sec)	0.02925	0.015389	0.009644	0.00495	0.00448

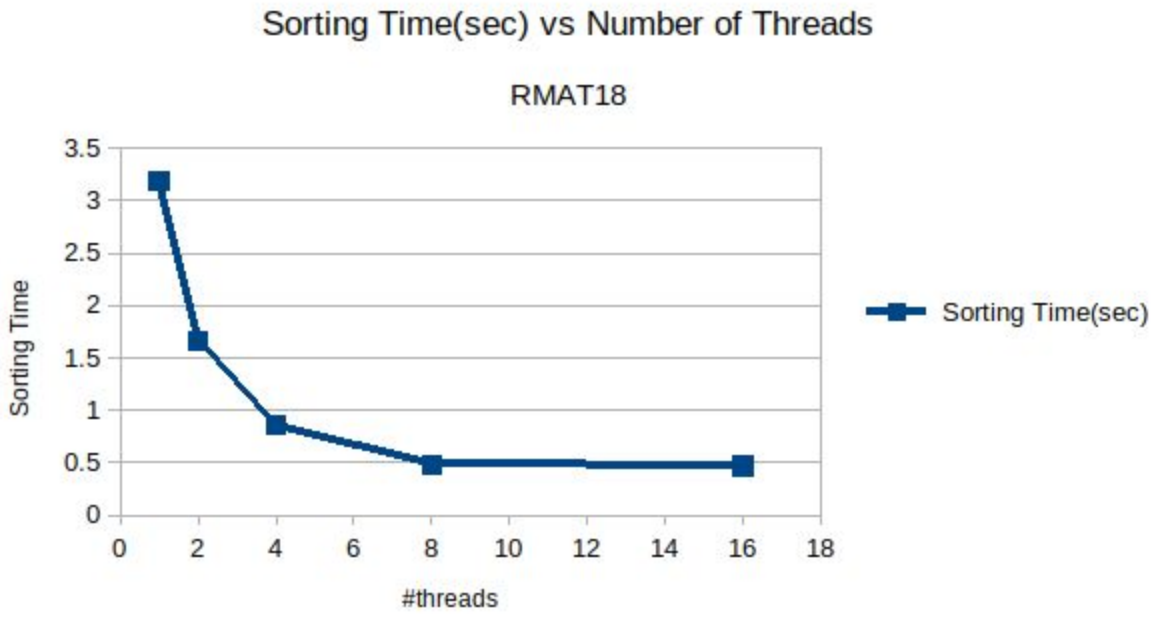
Speedup(numthread=1 compared to numthread=16) = 0.02925/0.00448 = 6.5
Speedup(numthread=1 compared to numthread=2) = 0.02925/0.015389 = 1.9
The same thing is observed here as the previous dataset. The total speedup between 1 thread execution and 16 thread execution is large, but the one between two consecutive threads(especially when the number of threads is already large) is small. We also see that the serial version of the code is faster than the parallel version with 1 thread but slower than the parallel version with 8 and 16 threads. Therefore, selecting the number of threads is essential to ensure an optimal system. On increasing the number of threads to a large extent, the performance will again drop due to overhead.



Edges: 11 Vertices: 31

TEST					
#threads	1	2	4	8	16
Sorting Time(sec)	4.8E-05	0.000128	0.000175	0.000255	0.000341

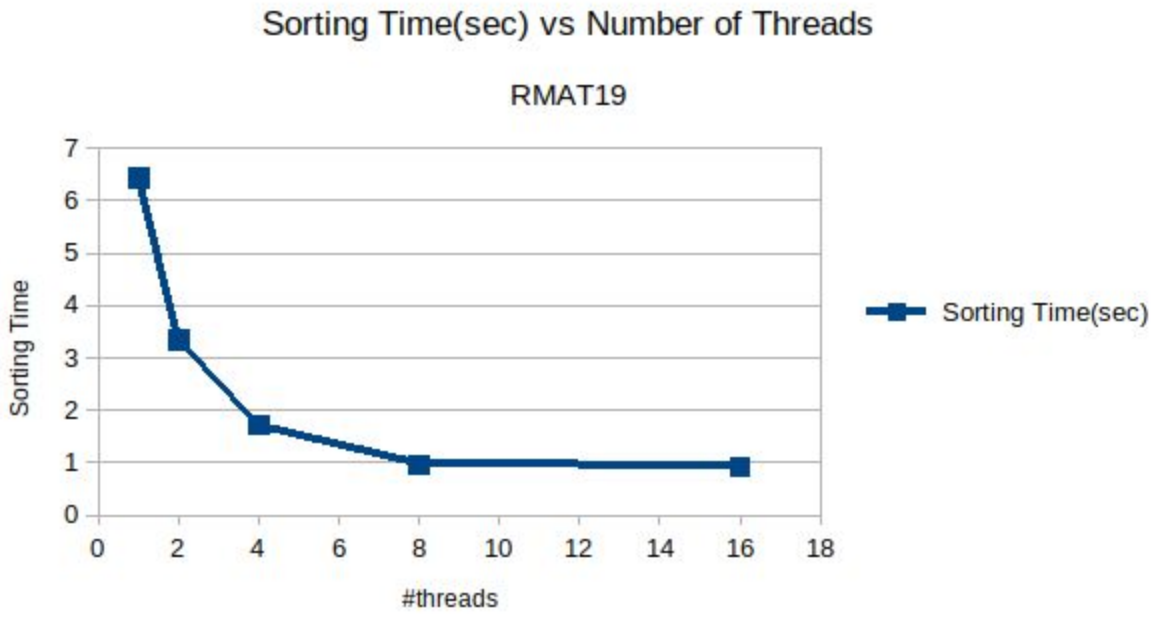
Speedup for TEST dataset comes out to be less than one. The number of data entries in TEST is very low. Because of this, increasing the number of threads just increases the overhead due to parallel programming which slows down the code. On the other hand, increasing the threads does not increase the performance since the bottleneck has already been reached. Multi-threading for a very small data set like above is not a good idea. We can simply use the serial version of the algorithm to attain optimum performance. This explains the nature of the graph.



Edges: 4194304 Vertices: 262144

RMAT18					
#threads	1	2	4	8	16
Sorting Time(sec)	3.193245	1.666796	0.859052	0.480666	0.469954

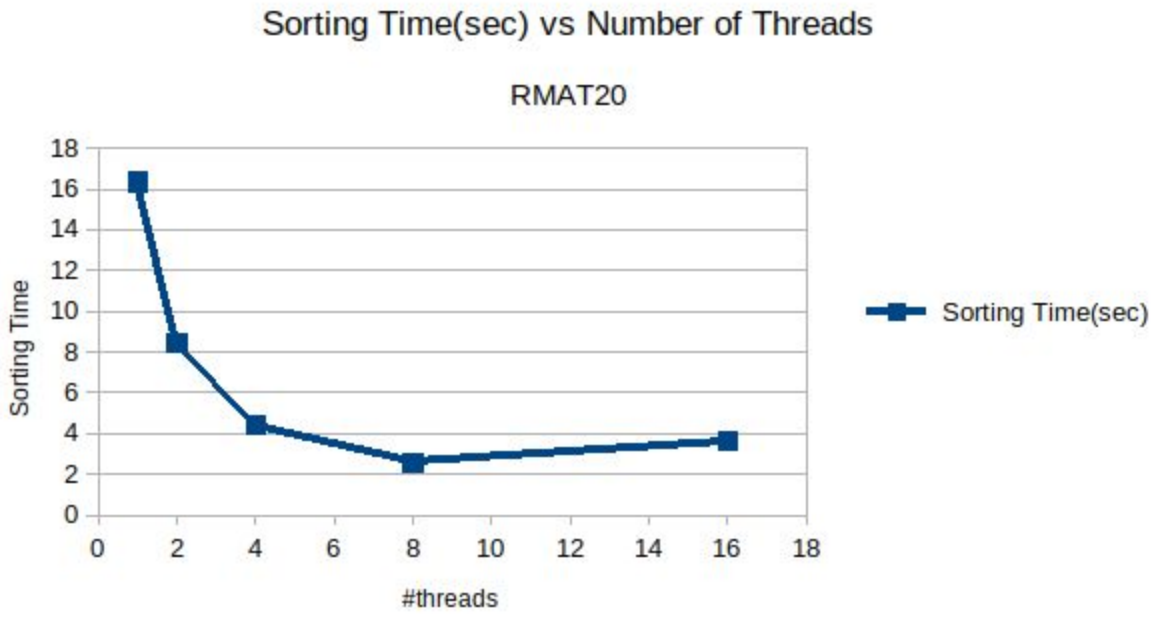
Speedup(numthread=1 compared to numthread=16) = 3.193245/0.469954 = 6.8
Speedup(numthread=1 compared to numthread=2) = 3.193245/1.666796 = 1.91
Unlike the TEST dataset, RMAT18 is very large and needs parallel programming to produce a good performance. Hence, the sorting time drops by almost 2 on increasing one thread. But the rate of drop goes on decreasing as we increase the number of threads. Here, the sorting time with 8 and 16 threads only has a difference of 20 ms as compared to the difference between the sorting time with 1 and 2 thread(s) which is around 1.6 seconds.



Edges: 8388608 Vertices: 524289

RMAT19					
#threads	1	2	4	8	16
Sorting Time(sec)	6.440924	3.339933	1.719646	0.962472	0.924053

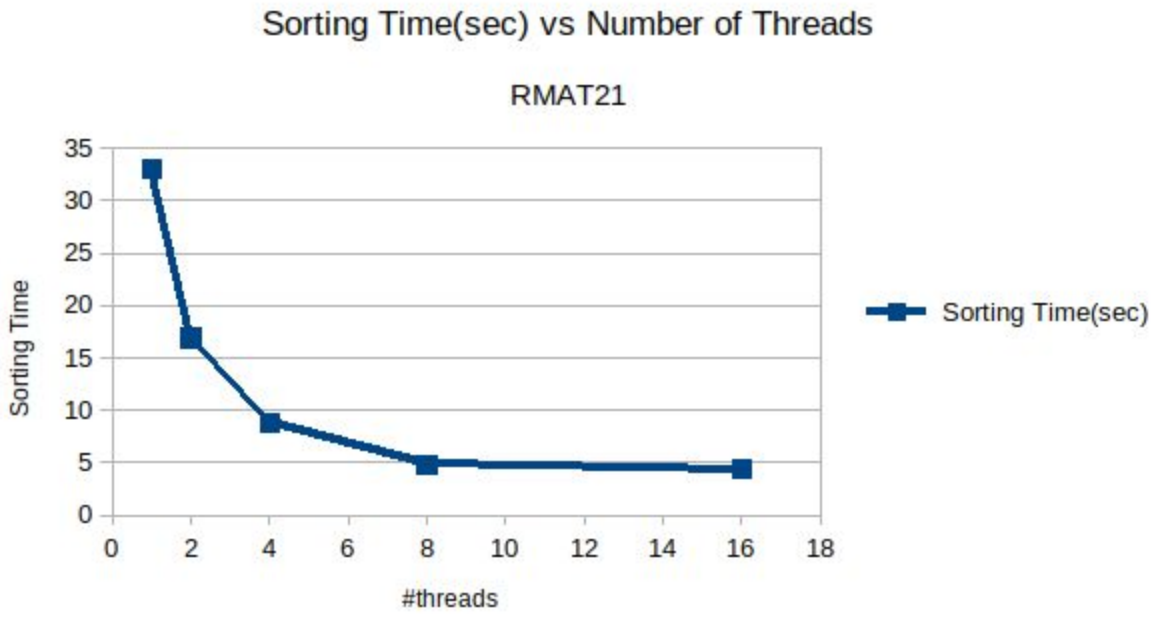
Speedup(numthread=1 compared to numthread=16) = 6.440924/0.924053 = 6.97
Speedup(numthread=1 compared to numthread=2) = 6.440924/3.339933= 1.92
The speedup of this dataset(6.97) is even greater than the previous dataset(RMAT18) because the number of edges in this dataset is much more than RMAT18. Hence, multi-threading helps sorting this data to a larger extent.
Therefore, **bigger the data set, larger are the benefits of multi-threading(greater speedup).**



Edges: 16777216 Vertices: 1048567

RMAT20					
#threads	1	2	4	8	16
Sorting Time(sec)	16.331398	8.448804	4.417057	2.593418	3.638934

Speedup(numthread=1 compared to numthread=8) = 16.331398/2.593418 = 6.29
Speedup(numthread=1 compared to numthread=2) = 16.331398/8.448804= 1.93
Somewhere in between RMAT19 and RMAT21, this dataset can also be considered to be large and be able to utilize multi-threading to a large extent.



Edges: 33554432 Vertices: 2097151

RMAT21					
#threads	1	2	4	8	16
Sorting Time(sec)	33.032485	16.891789	8.81182	4.828459	4.384518

Speedup(numthread=1 compared to numthread=16) = 33.032485/4.384518 = 7.53
Speedup(numthread=1 compared to numthread=2) = 33.032485/16.891789= 1.95
This dataset and has the highest speedup(compared to 16 threads) among all the other data sets. Such a data set could also utilize more than 16 threads efficiently. More than RMAT21, RMAT22 is even larger and can accommodate a larger number of threads efficiently. The difference between the sort times of the 16 thread and 8 thread process is also substantial compared to other data sets which shows that there is still scope for the sort time to decrease.

Remarks:

- Parallel programming can improve or degrade the performance of a system.
- It proves to be beneficial if two conditions are met:
    1) The CPU speed is limiting the performance of the system, not the other resources like memory or disk.
    2) It does not introduce large amounts of additional work(overhead).
- Bottlenecks should be taken into consideration while implementing multi-threading. If not used properly, it will just hamper the performance.
- False sharing occurs when the same cache line is shared by threads which cause the entire line to keep updating as and when required by the individual processes. Synchronization can be used to avoid this and thus improve performance.
- Sometimes a parallel program may execute more slowly than a serial one due to false sharing.
- Loops can always be exploited by parallel programming to improve performance by using API's like OpenMP.