Parallel Radix Sort based on Count Sort using OpenMP

Objective:

- To perform Radix sort of given datasets using count sort in serial and to parallelize the loops used in the sorting function.
- To report the time taken for sorting operation in serial and parallel and to compare values to find speed-up achieved on parallelizing the sorting function.

Introduction:

Count Sort:

Count-sort is a key-based algorithm which works by counting the number of significant key values in an array and using that key value to find the position of the corresponding element in the sorted array based on count value of the element.

Radix Sort:

Radix sort is an algorithm based on count-sort but executes in terms of digits and not the whole number. It sorts data with integer keys for each digit starting from the Least-significant-digit to the Most-significant-digit.

Steps – followed:

- Count-sorted the given set of elements in a dataset in serial.
- Performed serial radix-sort by executing count sort for ‘n’ times where ‘n’ is the number of digits of the largest number in the dataset.
- Parallelized all possible loops in count sort by changing the number of threads executing the program.
- Used the parallelized count-sort to perform radix sort of ‘n’ digits starting from the LSD to MSD.
- Measured time taken for each sorting algorithm and calculated speed-up achieved in parallelizing the function.

Values obtained:

	Count Sort Time (seconds)		Radix Sort Time (seconds)	
	Serial	**Parallel**	**Serial**	**Parallel**
RMAT 18	0.034572	0.024099	0.124486	0.076064
RMAT 19	0.078136	0.065703	0.248293	0.151435
RMAT 20	0.204064	0.195469	0.495806	0.305066
RMAT 21	0.512640	0.499711	0.999090	0.610166
RMAT 22	1.322170	1.248106	2.241038	1.344055

Serial: Number of Threads = 1

Parallel: Number of Threads = 2

It is found that the time taken to execute the sorting algorithm in parallel is lower than that to execute in serial.

Once the loops are parallelized, the array to be sorted in divided into parts and each part is allocated to a thread to execute individually, i.e., If 2 threads are used and if the number of elements in the input array is 7, 3 elements are allocated for the 1st thread, 3 for the 2nd thread and the remaining offset elements are allocated for the last thread to execute. Among the four steps performed in count sort, 3 of them could be parallelized such that every thread will execute individually at the same time, thus reducing the latency achieved in serial sorting to complete the process. Each thread would have its unique thread id, key values and count array to perform the sorting operation and the final sorted array is obtained from parallel sorting of input data based on count values.

For every test case, it is found that the time taken to execute in parallel is lower than that in serial for both count sort and radix sort. Also, as the size of dataset, the time taken to execute increases since the code needs to traverse through a higher number of vertices.

Speed-ups achieved:

RMAT 18:

Count Sort – 0.034572 / 0.024099

= 1.43

Count Sort Speed-up = 1.43x

Radix Sort – 0.124486 / 0.076064

= 1.64

Radix Sort Speed-up = 1.64x

Similarly, the speed-ups are found for the remaining datasets,

RMAT 19:

Count Sort – 1.19x

Radix Sort – 1.64x

RMAT 20:

Count Sort – 1.04x

Radix Sort – 1.63x

RMAT 21:

Count Sort – 1.03x

Radix Sort – 1.64x

RMAT 22:

Count Sort – 1.06x

Radix Sort – 1.67x

Thus, it is found that the radix sort achieves a better speed-up for every testcase.

RMAT 18 - Dataset:

Number of Threads	Count Sort Parallel Time (seconds)	Radix Sort Parallel Time (seconds)
1	0.034572	0.124486
2	0.024099	0.076064
3	0.025263	0.066334
4	0.026563	0.064044
6	0.039588	0.058335
8	0.048973	0.055213
16	0.056946	0.049354
32	0.080461	0.048268
64	0.131286	0.049032
128	0.230457	0.051756
256	0.430393	0.054863

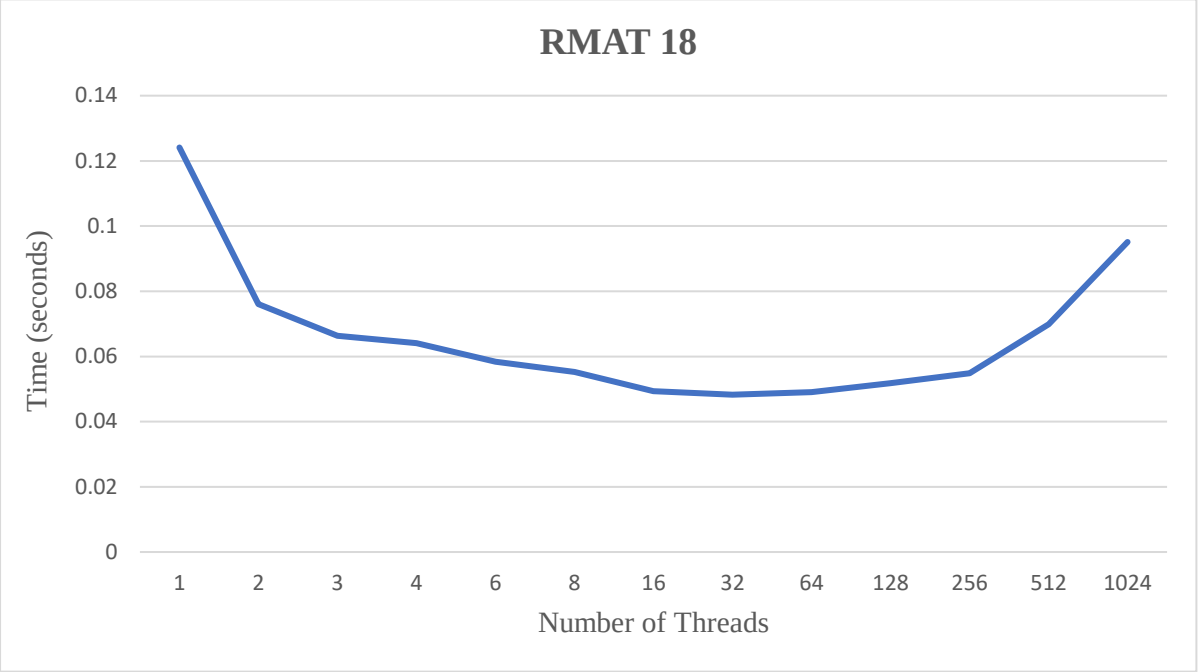
It is found that the count sort initially performs better than radix sort in serial, but as we increase parallelization by increasing the number of threads, the radix-sort emerges with a higher speed of execution than count sort.

Radix-sort parallel:

Number of Threads	Radix Sort Parallel Time (seconds)
1	0.124486
2	0.076064
3	0.066334
4	0.064044
6	0.058335
8	0.055213
16	0.049354
32	0.048268
64	0.049032
128	0.051756
256	0.054863
512	0.069900
1024	0.095064

A graph is plotted for parallel radix-sort based on the values obtained in-order to understand the performance of the function.

Graph:



Time Values are found for Radix sort – Parallel by changing the number of threads. From the graph, it is found that the time taken to perform sorting decreases as we increase the number of threads, running in parallel, to an extent. Beyond a value, as we increase the number of threads the time for sorting doesn't increase, rather increases. This may be due to the following reasons:

1. Using too many threads for a single program makes too little work for each thread and the overhead for starting and terminating of threads increases.
2. Also, overhead may rise from the way the threads share finite shared resources.

Also, cache pollution occurs when too many threads run a program sharing a cache, each may evict blocks from cache at each step which may be required by other threads and the time to fetch the data increases thus increasing the overall time to execute.

Also, as the number of context switching increases, the latency too increases, thus degrading the performance.

Similarly, the performance of count/radix sort is found for every dataset given.

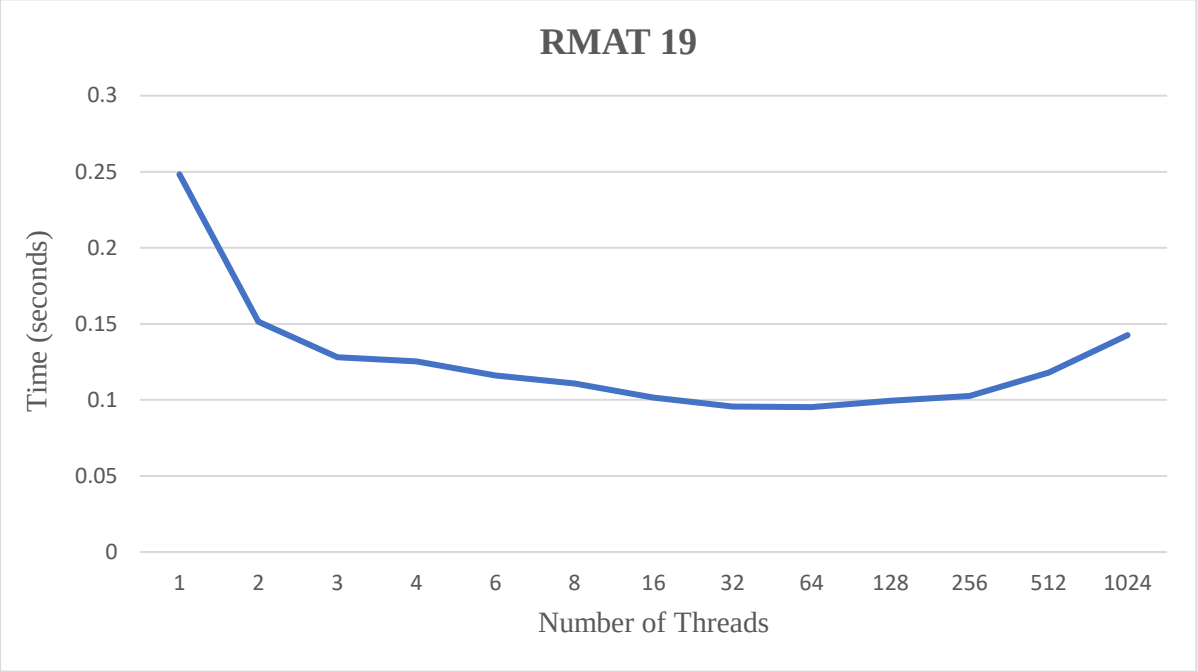
RMAT 19 - Dataset:

Number of Threads	Count Sort Parallel Time (seconds)	Radix Sort Parallel Time (seconds)
1	0.078136	0.248293
2	0.065703	0.151435
3	0.079006	0.127984
4	0.083430	0.125332
6	0.120720	0.115982
8	0.113998	0.110786
16	0.152348	0.101526
32	0.200481	0.095739
64	0.297140	0.095251
128	0.492197	0.099421
256	0.873727	0.102567

Radix-sort Parallel:

Number of Threads	Radix Sort Parallel Time (seconds)
1	0.248293
2	0.151435
3	0.127984
4	0.125332
6	0.115982
8	0.110786
16	0.101526
32	0.095739
64	0.095251
128	0.099421
256	0.102567
512	0.117847
1024	0.142604

Graph:



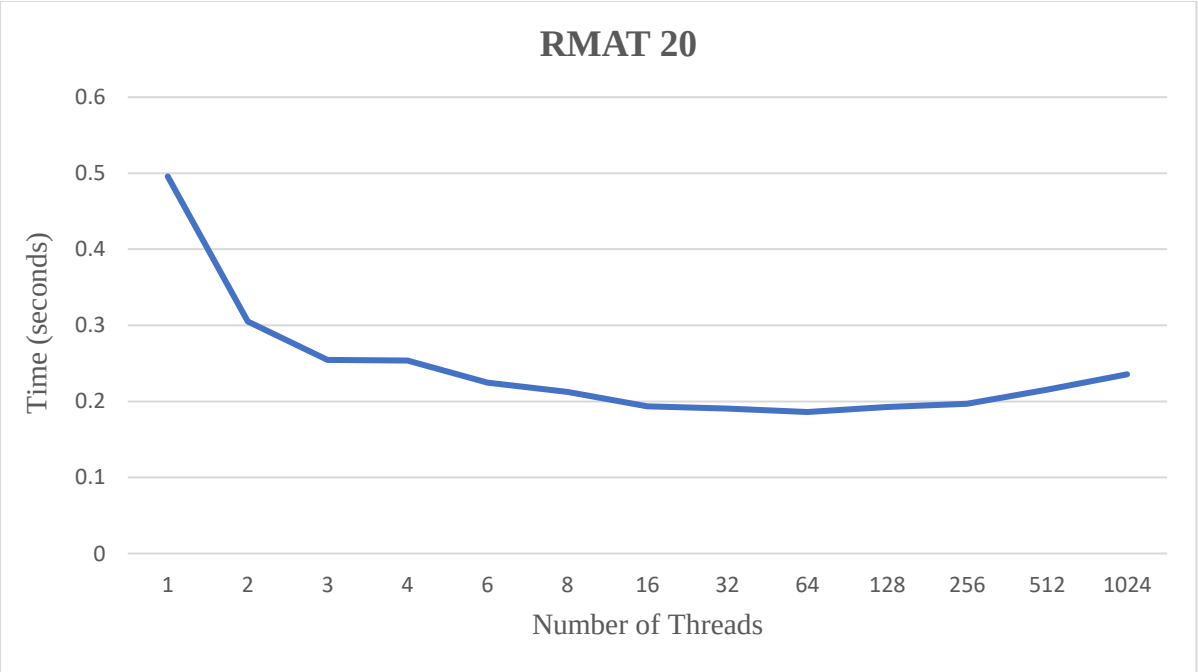
RMAT 20 - Dataset:

Number of Threads	Count Sort Parallel Time (seconds)	Radix Sort Parallel Time (seconds)
1	0.204064	0.495806
2	0.195469	0.305066
3	0.221810	0.254445
4	0.256893	0.253765
6	0.338035	0.224828
8	0.334384	0.212361
16	0.448798	0.193399
32	0.579631	0.190727
64	0.877574	0.186073
128	1.708815	0.192534
256	8.750523	0.197021

Radix-sort Parallel:

Number of Threads	Radix Sort Parallel Time (seconds)
1	0.495806
2	0.305066
3	0.254445
4	0.253765
6	0.224828
8	0.212361
16	0.193399
32	0.190727
64	0.186073
128	0.192534
256	0.197021
512	0.215536
1024	0.235667

Graph:



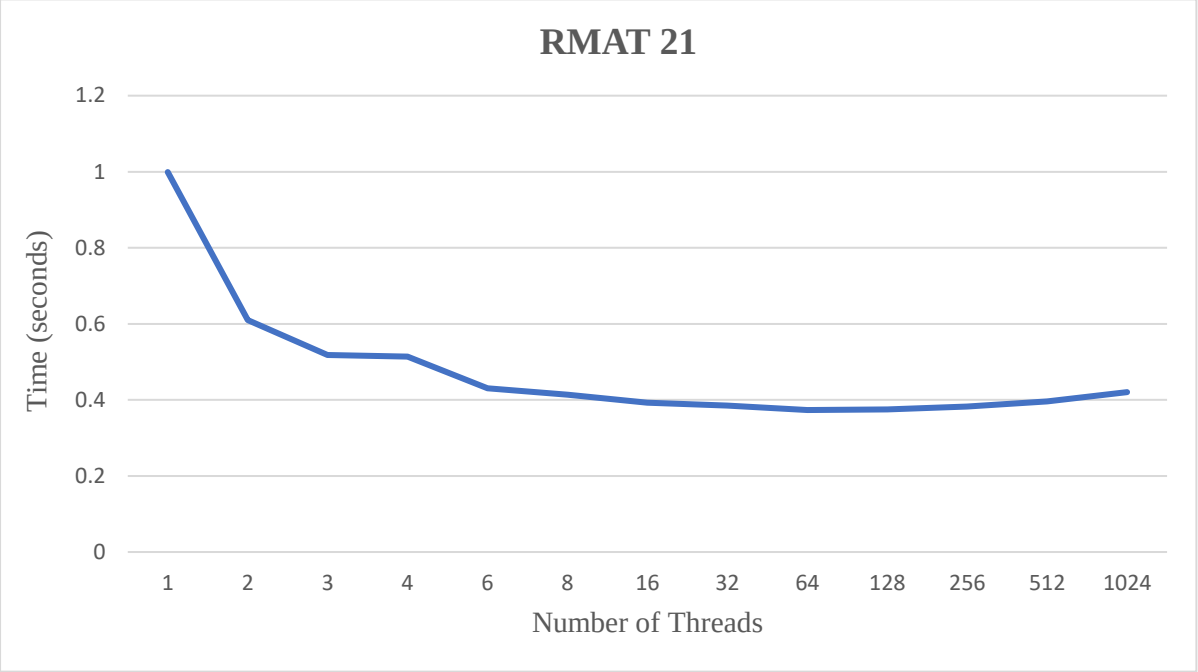
RMAT 21 - Dataset:

Number of Threads	Count Sort Parallel Time (seconds)	Radix Sort Parallel Time (seconds)
1	0.512640	0.999090
2	0.499711	0.610166
3	0.586133	0.517891
4	0.649996	0.514277
6	0.722416	0.430858
8	0.763011	0.413834
16	0.839342	0.392459
32	0.929054	0.385300
64	1.455976	0.373355
128	2.239064	0.374604
256	3.711823	0.382373

Radix-sort Parallel:

Number of Threads	Radix Sort Parallel Time (seconds)
1	0.999090
2	0.610166
3	0.517891
4	0.514277
6	0.430858
8	0.413834
16	0.392459
32	0.385300
64	0.373355
128	0.374604
256	0.382373
512	0.396152
1024	0.420514

Graph:



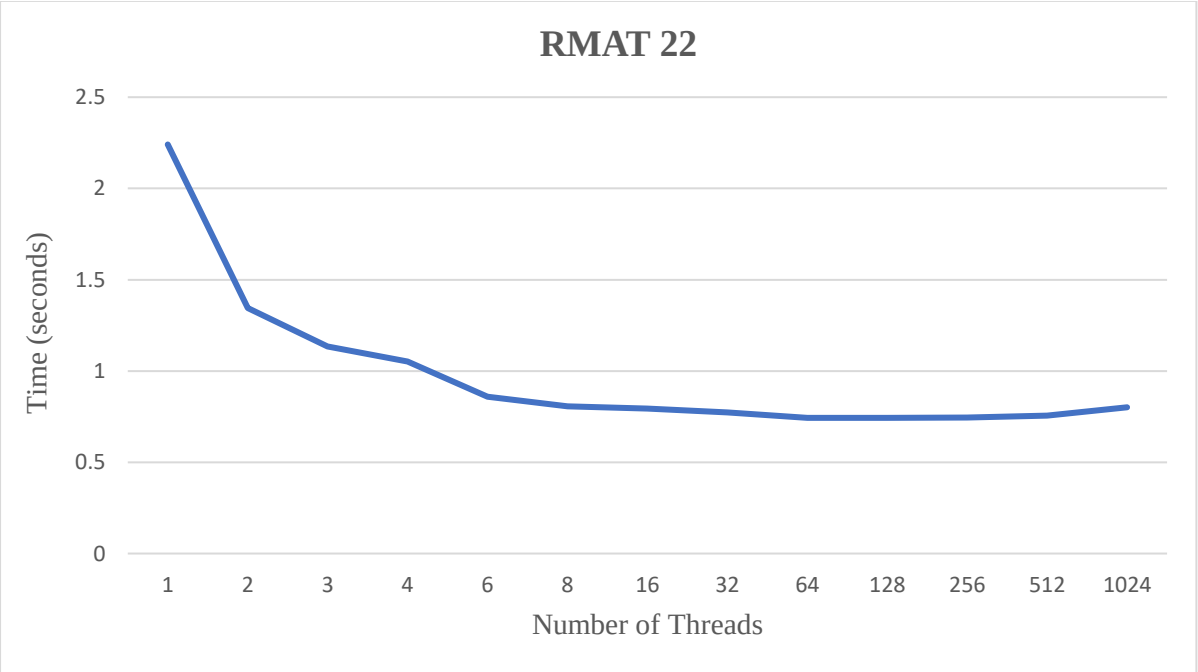
RMAT 22 - Dataset:

Number of Threads	Count Sort Parallel Time (seconds)	Radix Sort Parallel Time (seconds)
1	1.322170	2.241038
2	1.248106	1.344055
3	1.388403	1.133530
4	1.496439	1.052792
6	1.740123	0.858330
8	1.746560	0.806267
16	2.043553	0.794635
32	2.421686	0.773790
64	3.090759	0.743288
128	4.668924	0.744106
256	7.837487	0.745350

Radix-sort Parallel:

Number of Threads	Radix Sort Parallel Time (seconds)
1	2.241038
2	1.344055
3	1.133530
4	1.052792
6	0.858330
8	0.806267
16	0.794635
32	0.773790
64	0.743288
128	0.744106
256	0.745350
512	0.755580
1024	0.801780

Graph:



Inference:

Thus, for every dataset, it is found that lower execution time is achieved for parallel sort than serial sort as we increase the number of threads to an extent. Beyond that point, the time complexity again increases due to many threads executing a single program causing an increase in overhead due to context switching, cache pollution.

Conclusion:

- Thus, radix sort for given datasets using count sort is performed in serial and the loops used in the sorting function are parallelized.
- Time taken for sorting operation in serial and parallel are reported and speed-up achieved on parallelizing the sorting function is found.