ECE 506 – PARALLEL COMPUTER ARCHITECTURE

PROJECT - 1

PARALLELIZATION OF RADIX SORT ALGORITHM USING OPENMP AND PERFORMANCE EVALUATION FOR THE IMPLEMENTATION WITH DIFFERENT BENCHMARKS

Submitted By,

Vignesh Kumar Ramaraj

200310417

Aim and Objective:

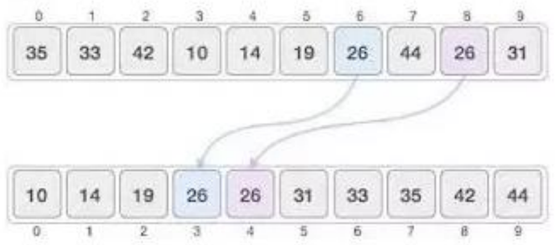
- To implement stable sorting (Count Sort, Radix Sort) algorithm that can be used in the preprocessing of applications that rely on graph data structures with large data sets.
- Since sorting alone takes up to 80% of the preprocessing time, the goal is to reduce the time it take to perform sorting of large data sets by exploiting the concurrency in the proposed Radix Sort algorithm
- After identifying the points that can be executed concurrently, make use of the OpenMP libraries and directives to parallelize sections of the Radix Sort algorithm.
- Observe and report the behavior of the implemented parallelized Radix Sort algorithm

Introduction:

Graph processing is becoming an ubiquitous application, and the cost of building a graph data structure for a given data set dominates the execution of the graph functions. It is extremely important to research the possibilities to optimize existing preprocessing steps in building a graph structure. Since data in a graph structure is represented as a set of vertices, and edges represent the links between them, it is crucial to preserve the stability of the data sets after sort and then feed to BFS traversal stage. Count Sort and Radix Sort are the most commonly employed stable sorting algorithms.

Stable Sort:

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in the sorted output as they appear in the unsorted input. Whereas a sorting algorithm is said to be unstable if there are two or more objects with equal keys which don't appear in same order before and after sorting.



Count Sort:

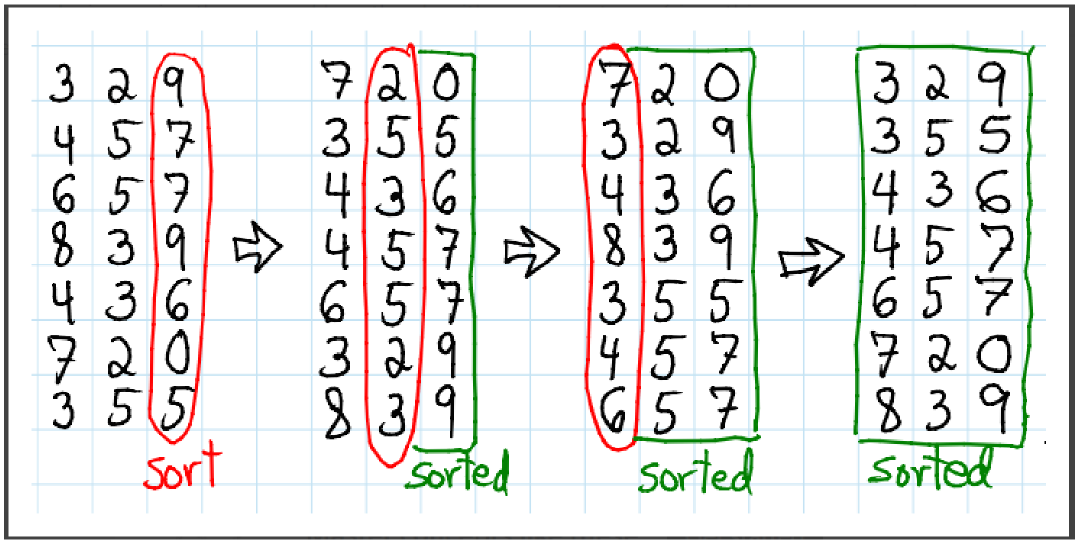
Counting sort is a non-comparative sorting technique that can perform better than O(nlogn) time. It is based on keys between a specific ranges. It counts the number of objects that has distinct key values (kind of hashing). Then it does some arithmetic to calculate the position of each object in the output sequence.

Radix Sort:

Radix sort is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value (place value). Radix sort uses counting sort as a subroutine to sort an array of numbers. Because radix sort is not comparison based, it is not bounded by Ω(nlogn) for running time — in fact, radix sort can perform in linear time. Radix sort incorporates the counting sort algorithm so that it can sort larger, multi-digit numbers without having to potentially decrease the efficiency by increasing the range of keys the algorithm must sort over (since this might cause a lot of wasted time).

Implementation:

1. Provided with the count sort function and other framework that read, parse and perform count sort for the data set.
2. Making using of the framework provided, written a function that performs radix sort by iterating over a loop that performs count sort for each digit of the vertices.



3. After implementing the radix sort, I noted the points that can be run concurrently
4. Employed **SMPD** parallel programming method and rewritten the radix sort algorithm which now can be parallelized without data dependencies
5. Included OpenMP libraries, made use of the parallel constructs and directives available to execute loop iterations parallely.
6. Simulated the written sorting algorithm with different number of thread and for different benchmarks.

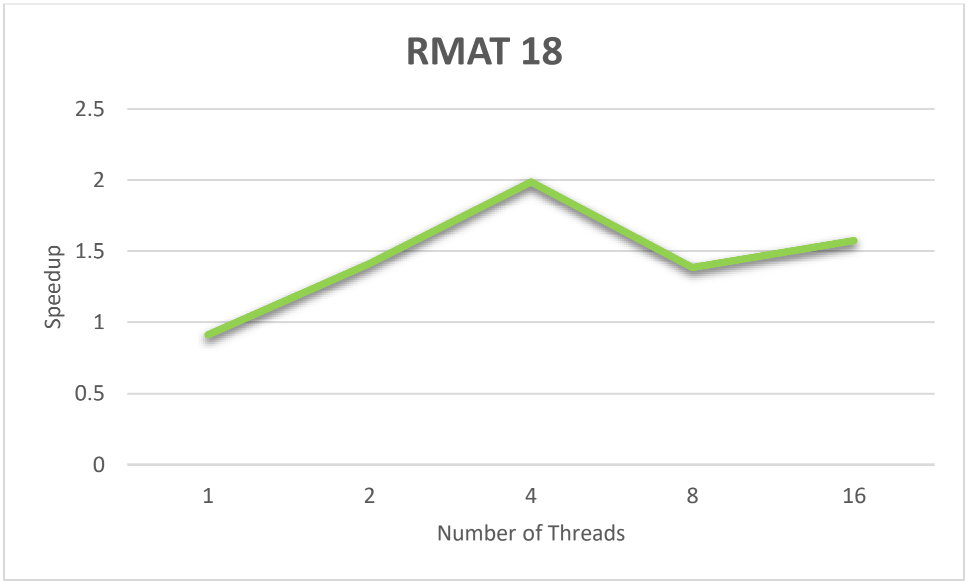
Tabulations and Plots

Simulated the algorithm for different data sets and regressed with different number of threads and tabulated the results.

Note: Serial column represent the algorithm that was written without any OpenMP parallel constructs and Thread = 1 represents algorithm that include parallel constructs but run with NUM_THREADS=1

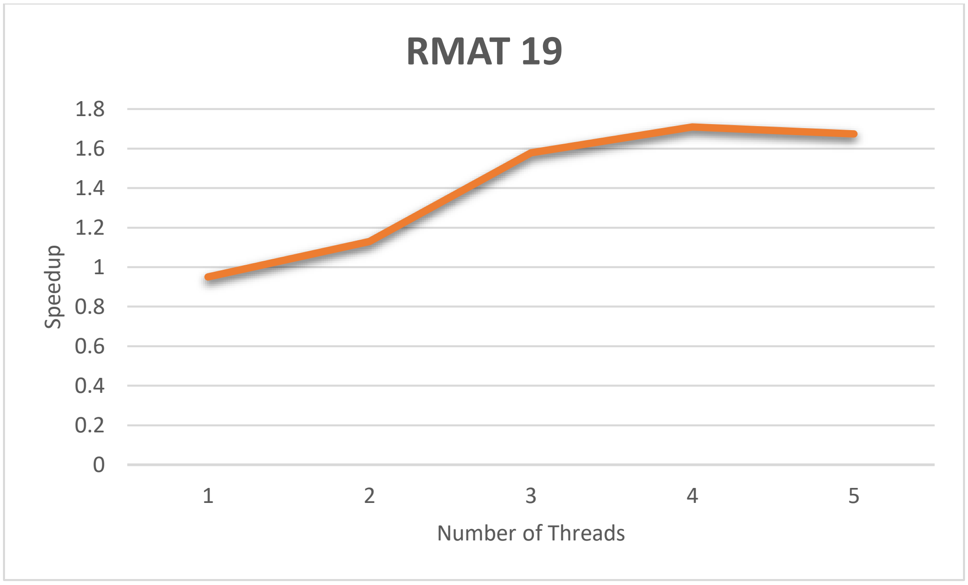
1. RMAT 18 TRACE

Number of threads	Serial	Parallel	Speedup
1	0.899607	0.985095	0.913218522
2	2.164265	1.531376	1.413281258
4	1.758028	0.884645	1.98726947
8	0.910621	0.657352	1.385286726
16	0.91025	0.5783	1.574010029



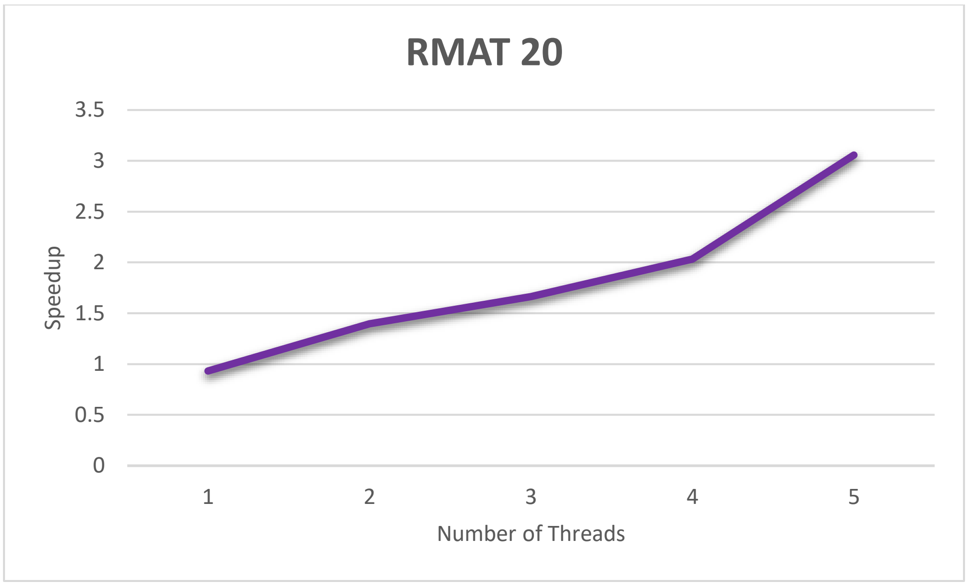
RMAT 19 TRACE

Number of threads	Serial	Parallel	Speedup
1	1.772184	1.864601	0.950436045
2	3.058118	2.705806	1.130205935
4	2.158458	1.367068	1.578895856
8	2.322646	1.358834	1.709293409
16	2.488935	1.485994	1.674929374



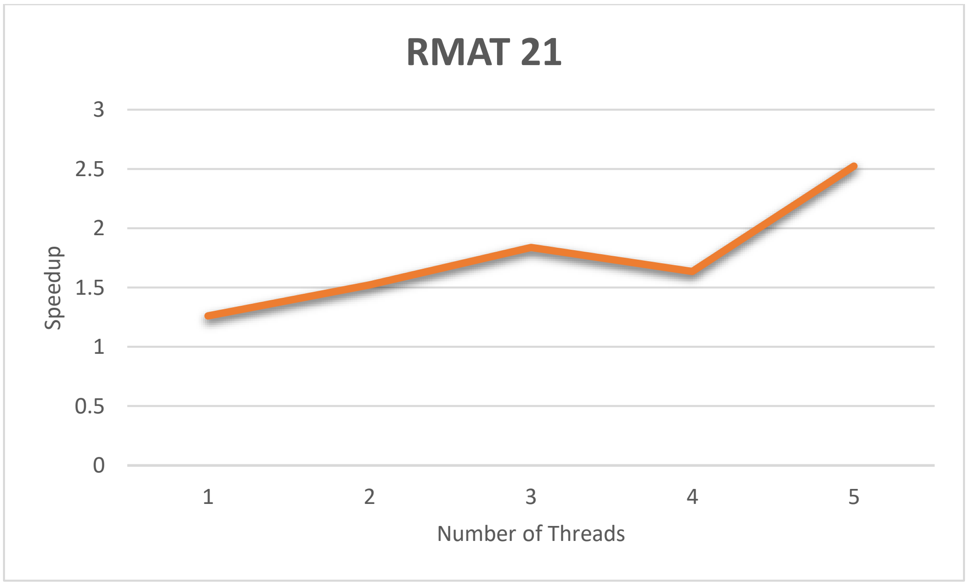
RMAT 20

Number of threads	Serial	Parallel	Speedup
1	4.27907	4.590361	0.932185944
2	3.963606	2.841007	1.39514123
4	5.928564	3.56427	1.663331902
8	5.342563	2.629636	2.031673966
16	4.791572	1.567566	3.056695539



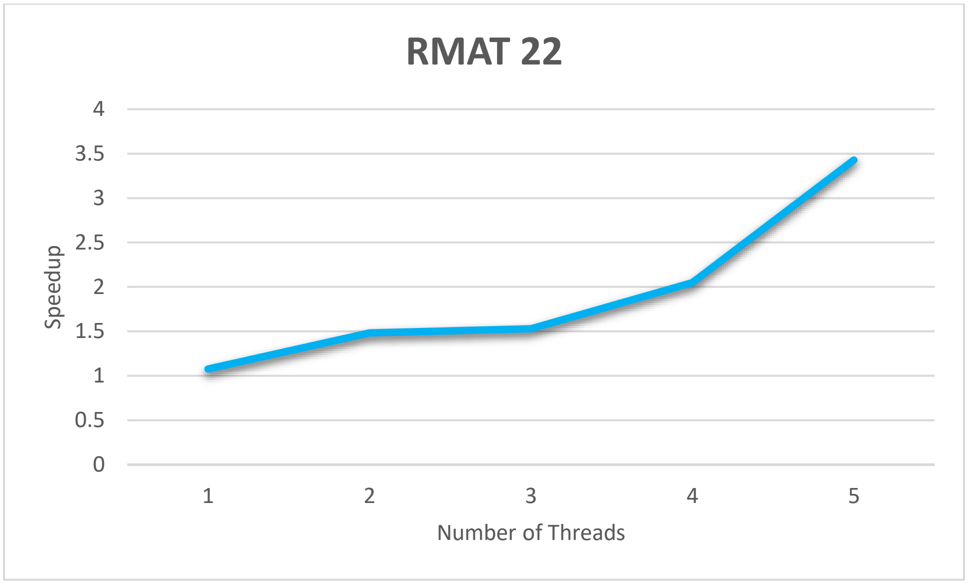
RMAT 21

Number of threads	Serial	Parallel	Speedup
1	13.894478	11.016586	1.261232654
2	11.857101	7.786954	1.522687947
4	14.226955	7.740207	1.838058724
8	8.640455	5.280053	1.636433384
16	9.253954	3.665517	2.524597212



RMAT 22

Number of threads	Serial	Parallel	Speedup
1	22.204021	20.632145	1.076185777
2	19.077764	12.862875	1.483164845
4	16.085935	10.512632	1.530152963
8	19.65376	9.598461	2.047594922
16	18.547706	5.410536	3.428071821



Observations

Plotted are the values of speedup Vs number of threads. As we can see from each of the graphs, initially for the smaller traces, trace with less number of vertices and edges, the serial execution seems to perform better than the corresponding parallel executions. But as we increase the size of data set, the parallel implementation show better results and speedup the sorting execution upto 3 times with 16 threads.

Why small data sets are executed better in Serial Radix Sort Implementation?

This is because for parallel implementation we have included set of parallel constructs and there are some synchronization points that increase the overhead for the processor. In case of smaller data set, these overheads included for parallelism dominate the total execution time. Thus our aim to exploit parallelism doesn’t implement any significance in these cases.

Overhead in maintaining parallel threads

Race Conditions!!!

We need to be careful including parallel constructions, because improper implementation leads to race conditions or affect the order of update of a variable which ripple through the whole program and leads to erroneous results.

To avoid race conditions I have used the basic synchronization constructs in OpenMp

#pragma omp barrier

#pragma omp critical

False Sharing!!!

Since each thread work separately on a different data, there are large possibilities of cache pollution. Based on the cache block size, the cache coherency protocol has degrading effects to cache working by invalidating the blocks when a particular data is updated by a particular thread. All these increase the cache miss rate and these are overheads for the parallel implementation.

Speedup increases as we increase the data set size!!!

From the graphs plotted for the RMAT traces, we can observe that as we run larger data sets, we are reaping the benefits of including constructs in the program. Since, here the loop iterations we carryout is large when we execute in serial, parallelizing them and assigning them to different thread improve the speed of execution of the program.

We are basically increasing the work that each thread perform, which is significantly higher compared with the overhead in maintaining the threads.

Conclusion

From the results of various benchmarks, we can say that simply executing a loop with multiple parallel threads does not improve the performance. With parallelism comes complexity, data dependencies and overheads in maintaining synchronization between the threads. But once these overheads becomes negligible when comparing with the useful work done by each threads, the performance improvement by distribution of work among threads becomes handy.