DS-221 Introduction to Scalable Systems
Assignment - 4

Submitted by - Ocima Kamboj
Serial Number - 06-02-01-10-51-18-1-15899

Parallelization

Contents

1 K-means Clustering Using OpenMP

1.1 Methodology

K-means clustering partitions N elements into K clusters, such that each element belongs to the cluster with the nearest means. This is achieved with the help of an iterative algorithm-

1. Choose K initial cluster means.
2. For all the elements, calculate the distance from the point to all the different means.
3. Assign each element to the cluster with the nearest mean.
4. Calculate the average of the elements in each cluster to obtain K new cluster means.
5. Repeat steps 2-4 until the assignments don't change, or the maximum number of iterations is reached.

Some of the ways in which the initial means can be chosen are as follows -

1. Select some particular elements from the input data as the initial means.
2. Select K elements randomly from the input data with uniform probability.
3. Use kmeans++ Algorithm. This algorithm tries to select points which are far away from each other. It does this by selecting the points randomly, but the probability of selecting them is directly proportional to the square of their distance to the nearest mean (which has already been selected).

This algorithm doesn't always converge to the global optimum. The solution that the algorithm converges to depends on the initial set of chosen means. So, in practice, the algorithm is run multiple number of times with different set of initial points, and the one which gives the minimum total "point-to-cluster mean" distance is chosen as the answer.

1.2 Experimental Setup

The sequential algorithm was coded in C++. After which OpenMP directives were added in this code to parallelize it.
Different ways were experimented with to select the initial points -

1. Select first 20 elements as the initial points.
2. Calculate the mean of every 1000 elements (for the 20,000 data set), and choose these as the initial means.
3. Select them uniformly at random.
4. Use the k-means++ algorithm.

To compare the times between different runs of the program, a set of initial points had to be fixed. To select this set, the program was run many times with uniformly chosen initial points, and the set which was giving the optimal answer in maximum number of iterations was selected.

1.3 Results

1.3.1 20k Data Set

The results are in the following tables -

No of iterations = 75			
Cluster	No of point	Centroid-x	Centroid-y
1	1000	949.918	347.023
2	1000	-336.451	346.249
3	1000	946.56	-352.638
4	1000	-941.625	350.892
5	1000	2754.86	350.439
6	1000	-2148.24	343.911
7	1000	-345.699	-355.499
8	1000	2143.91	349.022
9	1000	-1552.33	-355.163
10	1000	-2750.43	349.204
11	1000	-2153.26	-349.41
12	1000	-1552.65	353.062
13	1000	355.097	-343.376
14	1000	2149.6	-351.743
15	1000	-944.646	-347.193
16	1000	1550.91	-342.147
17	1000	2742.15	-353.587
18	1000	357.817	347.131
19	1000	-2762.36	-348.505
20	1000	1552.86	351.969

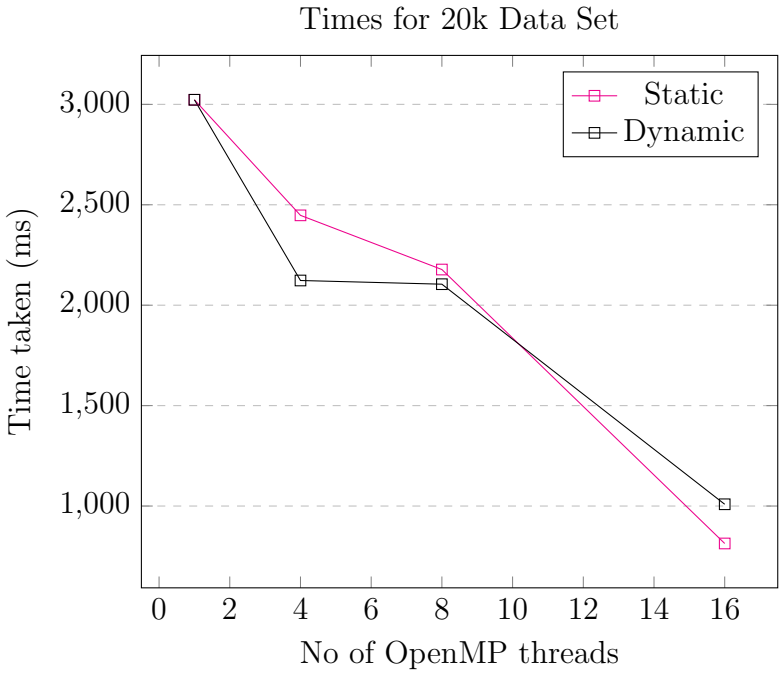
Table 1: Output for 20k Data Set

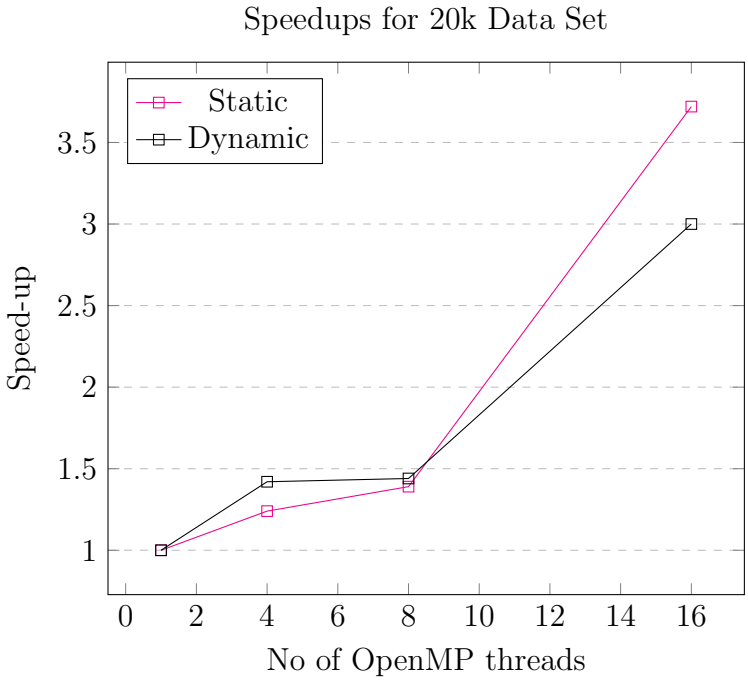
Static Schedule Clause							
No of Proc.	Time-1	Time-2	Time-3	Time-4	Time-5	Avg. Time	Speed up
1 (sequential)	2871	3031	3120	3131	2961	3022.8	1.00
4	2216	2402	1938	2812	2869	2447.4	1.24
8	2567	1762	2316	2412	1831	2177.6	1.39
16	926	773	792	830	747	813.6	3.72

Table 2: Times (ms) and Speed-ups for 20k Data Set

Dynamic Schedule Clause							
No of Proc.	Time-1	Time-2	Time-3	Time-4	Time-5	Avg. Time	Speed up
1 (sequential)	2871	3031	3120	3131	2961	3022.8	1.00
4	2183	2305	1746	2295	2086	2123	1.42
8	1992	2315	1887	2219	2110	2104.6	1.44
16	1090	863	1091	968	1032	1008.8	3.00

Table 3: Times (ms) and Speed-ups for 20k Data Set


Times for 20k Data Set
Static
Dynamic
Time taken (ms)
3,000
2,500
2,000
1,500
1,000
0
2
4
6
8
10
12
14
16
No of OpenMP threads


Speedups for 20k Data Set
Static
Dynamic
Speed-up
3.5
3
2.5
2
1.5
1
0
2
4
6
8
10
12
14
16
No of OpenMP threads

1.3.2 500k Data Set

The results are in the following tables -

No of iterations = 86			
Cluster	No of point	Centroid-x	Centroid-y
1	25000	-20999.3	7604.37
2	7736	19688	-8551.55
3	50000	-41251.1	7598.84
4	25000	-21017.5	-7574.38
5	25000	20986.2	7583.29
6	12353	7487	-8909.29
7	50000	54751.3	-7591.23
8	25000	34495	-7616.55
9	12647	7522.98	-6304.13
10	25000	61504.9	7615.66
11	25000	-47999.3	-7606.15
12	25000	7507.24	7591.2
13	25000	-61518.1	-7600.48
14	25000	-61505.1	7603.93
15	9616	21001.1	-6018.18
16	25000	47995.8	7603.73
17	7648	22362.6	-8573.82
18	25000	-34508.7	-7605.1
19	50000	-7503.1	0.48862
20	25000	34505.7	7590.43

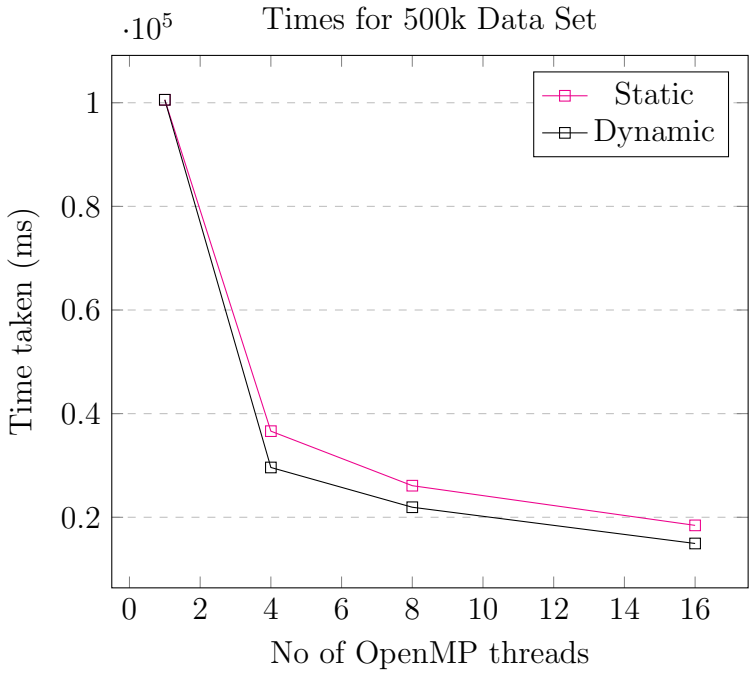
Table 4: Output for 500k Data Set

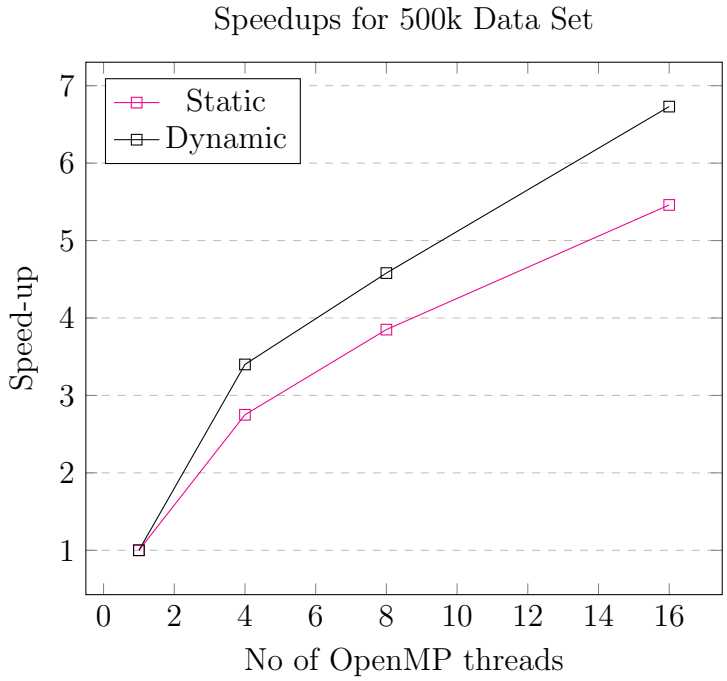
Static Schedule Clause							
No of Proc.	Time-1	Time-2	Time-3	Time-4	Time-5	Avg. Time	Speed up
1 (sequential)	131166	119533	82006	80263	89885	100570.6	1.00
4	38676	34939	34510	36335	38639	36619.8	2.75
8	29799	26895	23799	24671	25311	26095	3.85
16	18116	18055	19323	17217	19471	18436.4	5.46

Table 5: Times (ms) and Speed-ups for 500k Data Set

Dynamic Schedule Clause							
No of Proc.	Time-1	Time-2	Time-3	Time-4	Time-5	Avg. Time	Speed up
1 (sequential)	131166	119533	82006	80263	89885	100570.6	1.00
4	29400	29757	28540	30380	29977	29610.8	3.40
8	21406	21842	21238	22537	22655	21935.6	4.58
16	14422	15679	15667	14337	14602	14941.4	6.73

Table 6: Times (ms) and Speed-ups for 500k Data Set





1.4 Observations

1. Parallelization has decreased the running times of the algorithm in all cases as expected.

2. We'd expect that the dynamic scheduling clause performs better than the static scheduling clause. For the 500k data set, the dynamic scheduling clause performs consistently better than the static scheduling clause. For the 20k data set, for 16 threads, the static

scheduling clause is performing better, which is unexpected. This trend also differed between different runs.

3. In ideal cases, the speed up is equal to the number of processes. But in practical scenarios, it is always less than the number of processes. Our results conform to this.

4. The speed ups for the 500k data set are better than the 20k data set, which is expected because the ratio of parallelizable code to sequential code increased for larger data sets. Thus, more benefit in terms of performance due to parallelzation is expected for larger data sets.

2 MPI: Merge Sort

2.1 Methodology

Merge Sort is a divide and conquer algorithm which recursively divides the input in two halves, sorts them, and then takes this two sorted halves and merges them to give a full sorted array. The algorithm using MPI was implemented as follows -

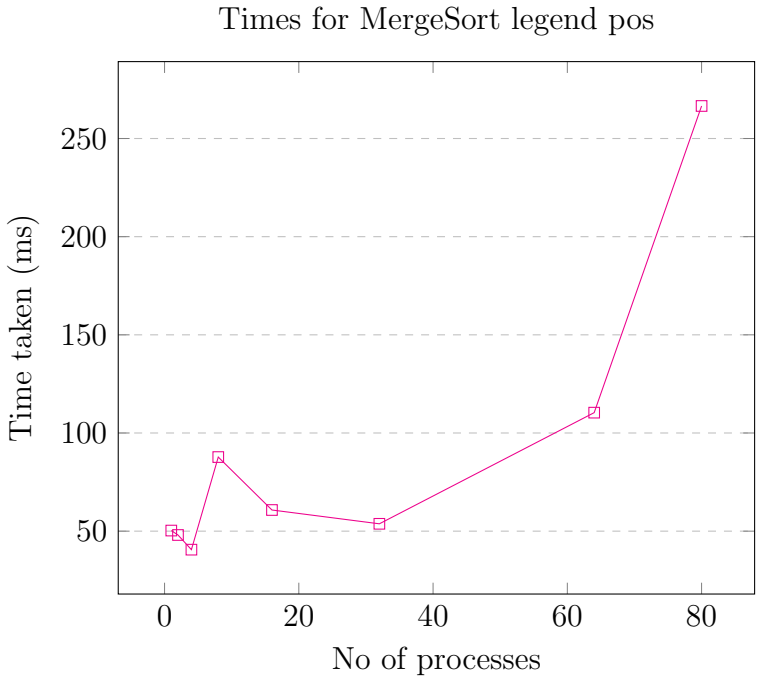
1. The process-0 loads the input array and then divides them among the processes.
2. The processes sort the part of the array that is with them.
3. We then start a tree-like reduction. In the first iteration, all the odd processes send their sorted arrays to the even processes, which then perform the merge routine on this received array and the array already with them. In the next iteration, processes again get together in groups of two to merge the arrays that are with them.
4. After $ceil(log_2 n)$ (n is the number of processes) iterations, the full sorted array is formed in process-0.

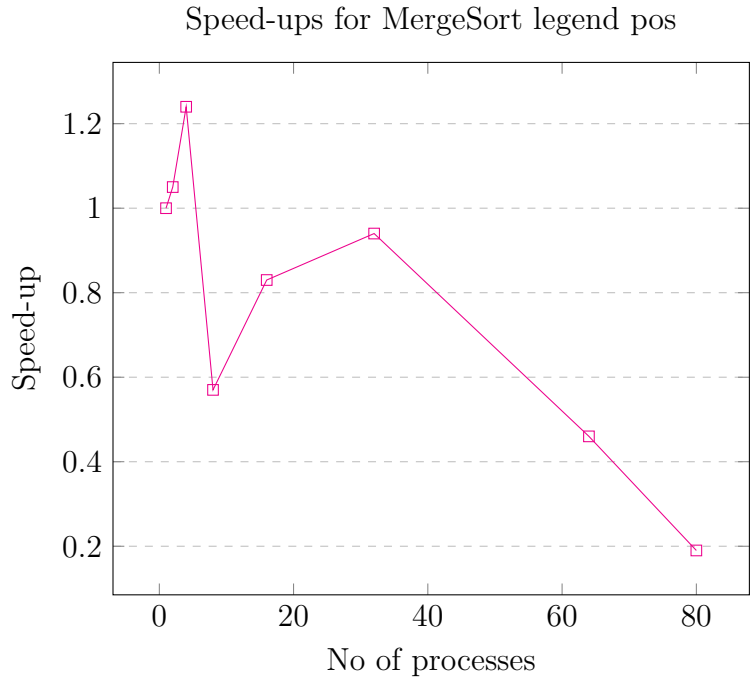
2.2 Results

The execution times and speed ups are in the following tables. These times also include the data loading time.

	QSort	Seq. MSort	2	4	8	16	32	64	80
T-1	49	70	56.84	43.01	80.44	53.49	59.92	65.29	180.24
T-2	49	66	46.61	55.15	118.39	59.08	34.67	85.27	132.21
T-3	44	70	50.88	32.00	82.35	67.61	44.16	124.25	134.17
T-4	45	70	42.26	34.11	102.66	72.49	38.25	103.61	133.74
T-5	49	79	42.45	38.51	58.30	62.62	30.50	100.36	85.04
T-6	48	71	63.20	38.01	64.30	105.25	62.41	162.97	1064.97
T-7	49	73	50.25	37.73	105.65	60.80	79.08	122.25	70.68
T-8	49	71	44.72	39.81	93.52	46.38	55.76	118.37	274.70
T-9	57	64	43.84	41.79	109.74	31.23	74.68	111.65	280.46
T-10	64	67	39.36	45.53	62.03	48.75	57.93	109.55	309.48
Avg Time	50.3	70.1	48.04	40.57	87.74	60.77	53.74	110.36	266.57
Speed ups	1	0.72	1.05	1.24	0.57	0.83	0.94	0.46	0.19

Table 7: Times(in ms) and Speed-ups for MergeSort





2.3 Observations

1. The trend obtained in times and speed-ups is inconsistent. One possible reason may be the time at which the program was run on the machine, because subsequent runs were giving different results. Another reason might be the relatively small size of the data set.

2. As expected, we get speed-ups till 4 processes if we compare with the inbuilt Quick Sort, and till 32 process if we compare with sequential Merge Sort (except at 8 processes, which is showing an unexpected increase in execution time), showing the performance benefit of parallelisation.

3. The speed ups are not much. We might get more benefit of the parallel code in case of a larger data set.

4. The times for larger number of processes are increasing. This may be because of the increased overhead of Sends and Receives.

3 Reduction using CUDA

Output of reduction = 637144.

	Seq	Shared Memory	Without Shared memory
Time	0.148	0.12608	0.137984
Speedup	1	1.17	1.07

Table 8: Times(in ms) and Speed-ups for Reduction