CS 426 Parallel Computing

Project 2 Report

Alp Ege Baştürk

21501267

07.04.2019

Detailed Explanation of the k-reduce function

The k-reduce function initially creates 2 arrays to store ids and values called tmpValStorage and tmpIDStorage. These are allocated and used as buffers only by the root process. Root process makes 2 MPI_Gather() calls in order to gather local ids and values calculated in parallel by nodes prior to reduce. After the calls are made to MPI_Gather(), master has all parts in its buffers of length k * world_size. Then master packs these parallel arrays into a single struct pack array to allow use of qsort. After sort, this array is truncated to get rid of values after first k ones, then unpacked to get ids and values back into the leastk and values.

Explanation of the Main program

- Initialization & file read

The main program starts by passing arguments to program. Then master process initializes the buffer and reads documents by calling readDocuments() and query by calling readQuery() functions. Then it sends single integer values and query to all processes by using MPI_Send() calls. This is determined by if (rank == 0) check. Meanwhile receivers receive from the root using MPI_Recv() in the else part.

- Calculations for data sizes to be sent/received

After distributing global values and the query, each node initializes a dataPortionLengths array. Then makes calculations to find data amount each process will get. If size is not divisible, then remainder is distributed to the portions array. This calculation is redundant for most values but it's simple arithmetic and all nodes can calculate once to prevent communication later.

- Distribution of documents from master

After calculating data portions to be received/send, non-root nodes initialize buffers named myDocumentPartMatrix according to the data sizes they will receive, while master sets this pointer to the first element of the documentMatrix, which is the whole read file. Master sends data row by row to other processes according to calculated data portion lengths using MPI_Sends since collective communications are forbidden at this point.

- Local Calculations

At this point, each process has its part to be calculated in its myDocumentPartMatrix buffer. Everynode calls calculate similarity function for each row they have with the query. This fills a similarities array with the similarities of each row to the query.

After calculating similarities locally, each node calls findLocalLeastk(similarities, myDocumentPartMatrix, dataPortionLengths[rank], k, &My_ids, &My_vals);. This function takes similarities and myDocumentPartMatrix which stores ids at row beginnings. It packs these parallel values to struct pack array, then calls qsort on them with custom comparison function compareFunc, which compares based on values. Then it truncates values after first k values then unpacks it to return My_ids and My_vals respectively.

- Kreduce, print & cleanup

At this point local calculations are completed and each node calls k-reduce function to reduce their results, which was explained above. Then master prints results, all processes make their cleanup and exit.

Data & Results

Tests were done on Intel i7-4720HQ CPU with 4 cores at 2.6 GHz.

- **Effect of Processor Count & Effect of Document Count**

The following data show effects of increasing processor count and document count (size) can be seen

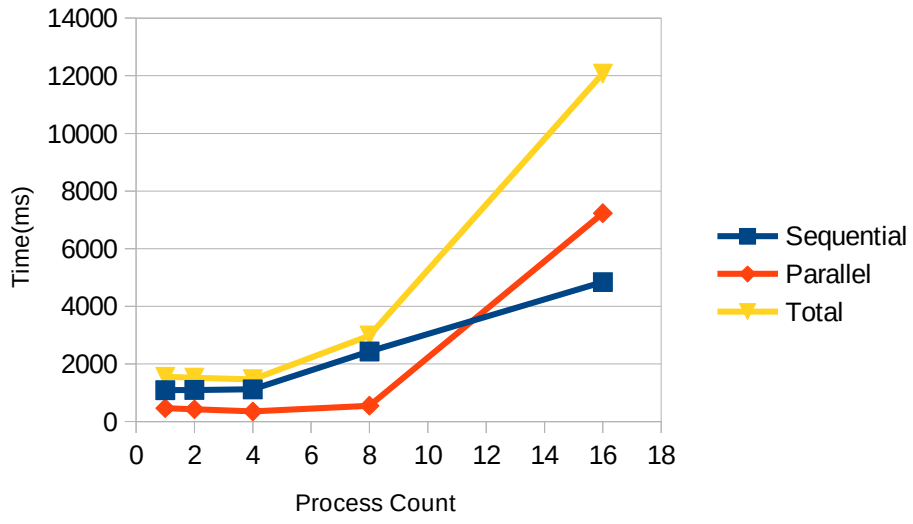
DocSize	1000000
Dictionary Length	4
k	4

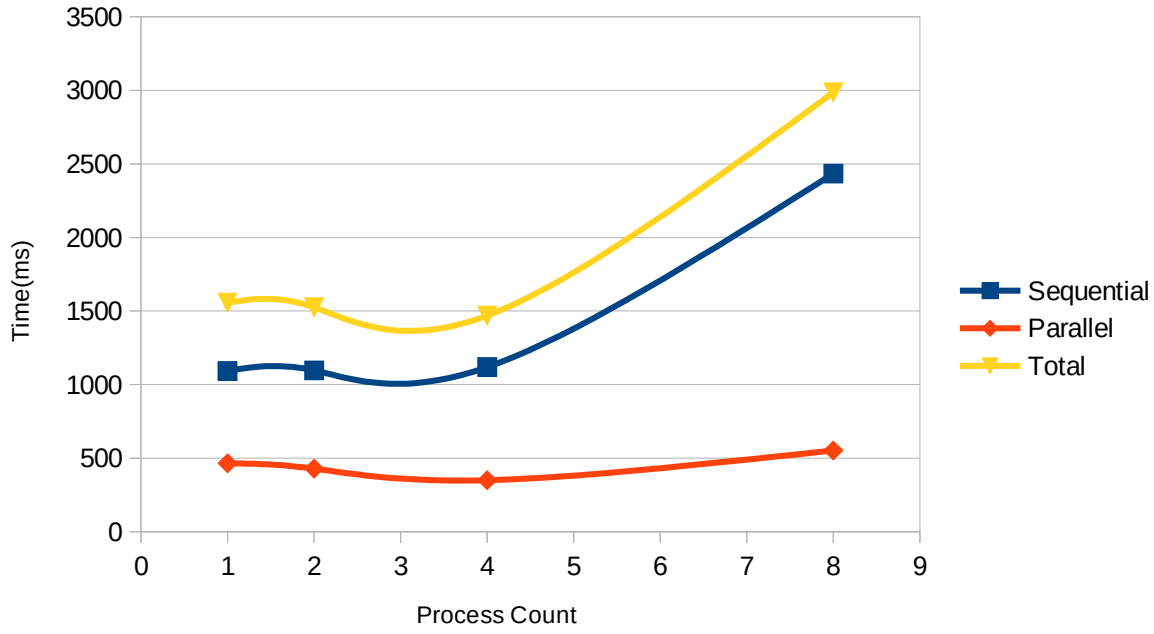
Proccessor count	Sequential	Parallel	Total
1	1091.813326	464.624405	1556.442976
2	1096.49992	428.498507	1525.004148
4	1118.232965	350.189686	1468.429804
8	2434.082031	551.880598	2985.969067
16	4842.767477	7232.811928	12075.587988

DocSize	2000000
Dictionary Length	4
k	4

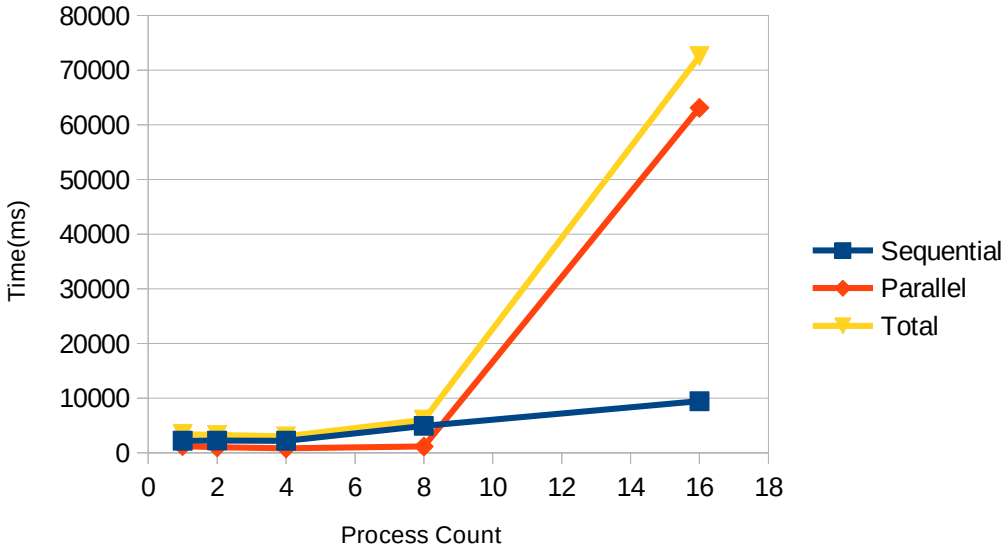
Proccessor count	Sequential	Parallel	Total
1	2206.599474	1212.07881	3418.683052
2	2237.687588	1003.763199	3241.455793
4	2207.108498	803.728819	3010.843515
8	4910.181046	1153.900862	6064.092636
16	9430.687666	63110.571384	72541.264534

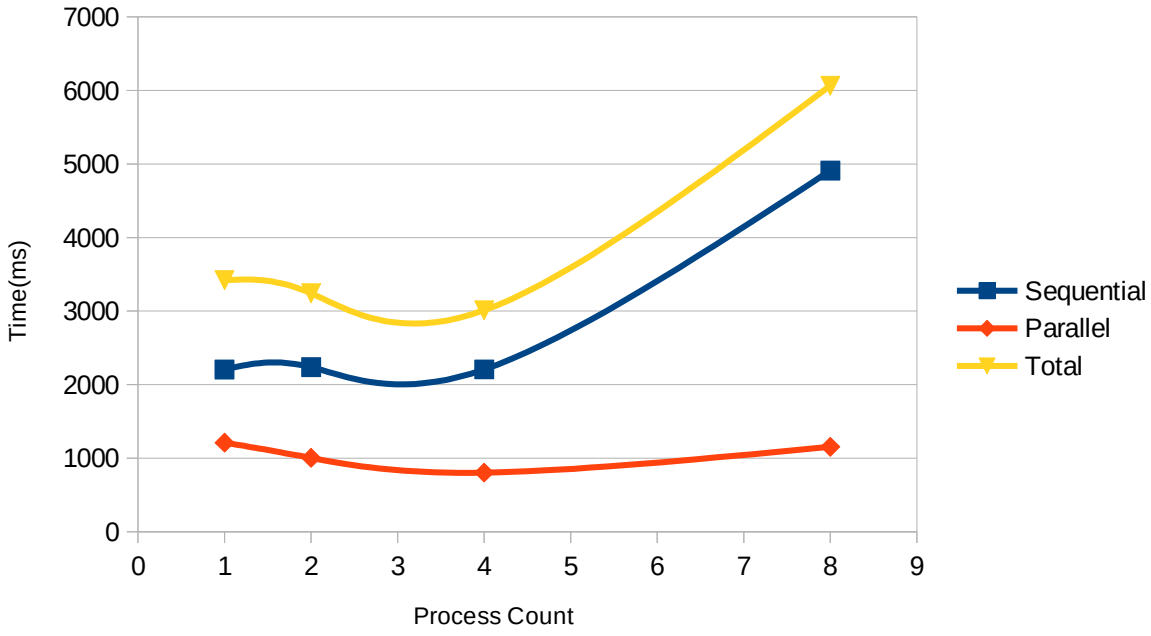
Processor Count Relation for Document Count 1M





Processor Count Relation for Document Count 2M





- **Discussion About the Processor Count**

As it can be seen from the graphs, increase in processor count improves parallel computation performance and overall performance up to a point. This point is 4 cores in the test machine. After this point additional processes cannot run in parallel on the test machine because there are no physical cores left to map and additional costs like message passing and context switching costs deteriorates the performance considerably.

- **Discussion About the Document Count**

Comparing the graphs of 1M and 2M documents, it can be seen that the computation time increases. Increase is greater than 2 times, which might be caused by additional overheads of processing more documents. This increase is expected as row counts are doubled and reading and processing them takes more time.

- **Effects of selecting different K values**

DocSize	1000000
Dictionary Length	8
k	2

Proccessor count	Sequential	Parallel	Total
1	1811.793566	818.03441	2629.833221
2	1815.045118	635.652542	2450.702906
4	1892.768621	503.243446	2396.016359
8	4050.93503	687.002897	4737.944603
16	8189.244986	32581.519842	40770.774364

DocSize	1000000
Dictionary Length	8
k	4

Proccessor count	Sequential	Parallel	Total
1	1812.492371	818.365812	2630.864143
2	1816.076517	641.795874	2457.877398
4	1907.280922	478.472233	2385.762215
8	4130.916357	708.434343	4839.356184
16	8227.512598	20204.510212	28432.030678

DocSize	1000000
Dictionary Length	8
k	8

Proccessor count	Sequential	Parallel	Total
1	1825.470448	818.930387	2644.405365
2	1814.378977	638.250351	2452.634096
4	1829.786062	484.972954	2314.775467
8	4039.84952	683.339119	4723.193884
16	8058.221102	32959.721804	41017.949343

Derived values from the tables above:

Proccessor count	1

K	Sequential	Parallel	Total
2	1811.793566	818.03441	2629.833221
4	1812.492371	818.365812	2630.864143
8	1825.470448	818.930387	2644.405365

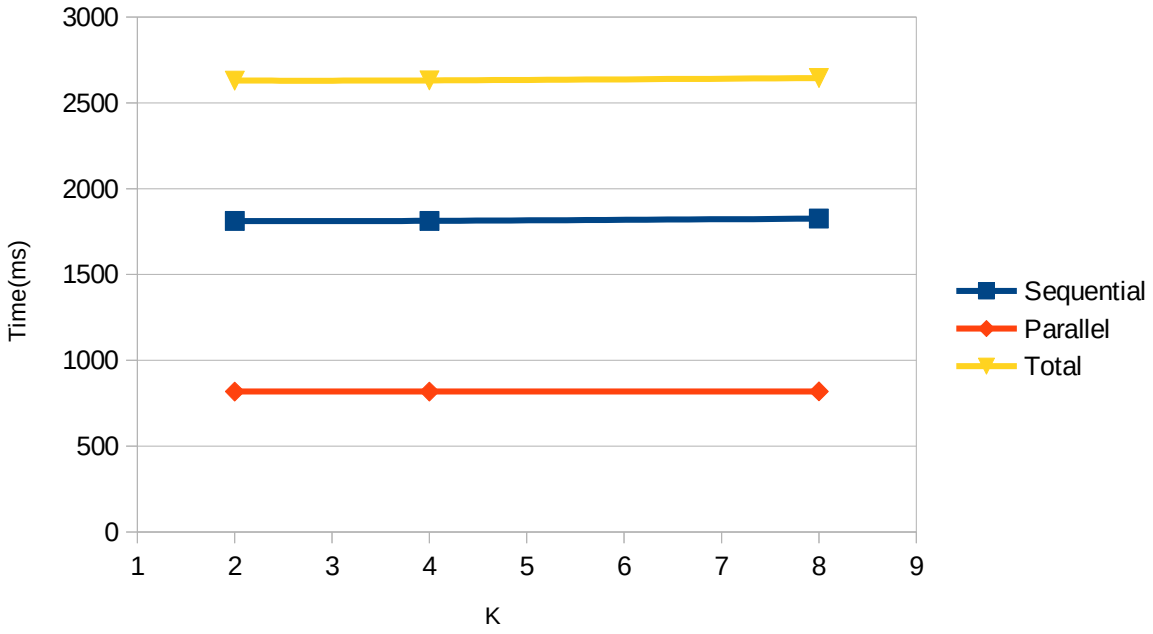
Proccessor count	2

K	Sequential	Parallel	Total
2	1815.045118	635.652542	2450.702906
4	1816.076517	641.795874	2457.877398
8	1814.378977	638.250351	2452.634096

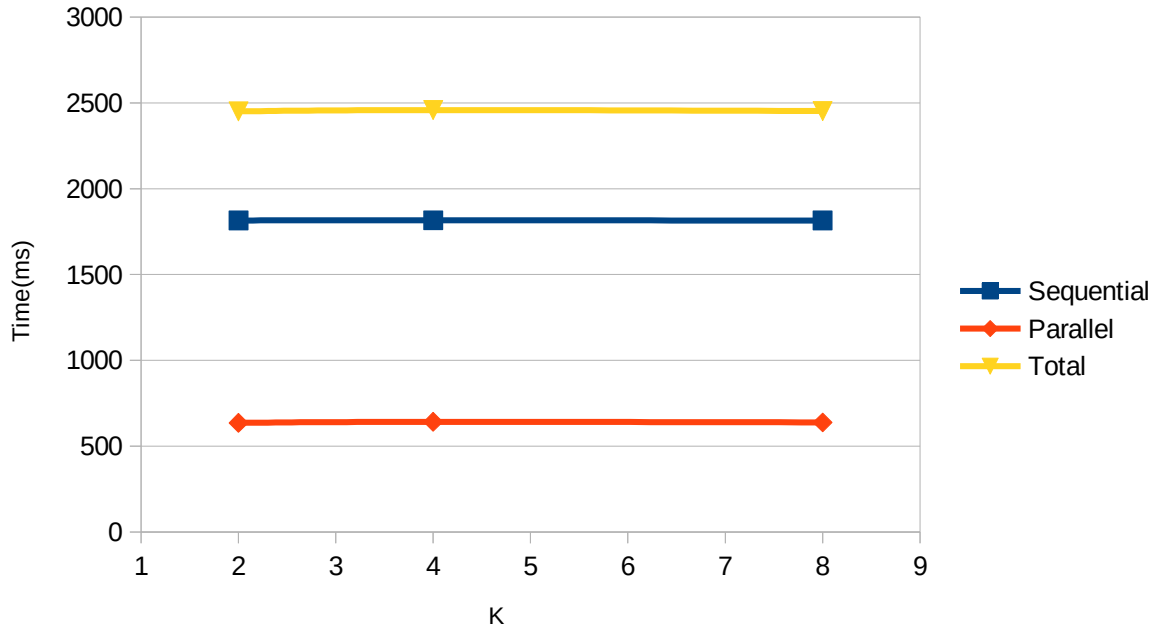
Proccessor cour	4

K	Sequential	Parallel	Total
2	1892.768621	503.243446	2396.016359
4	1907.280922	478.472233	2385.762215
8	1829.786062	484.972954	2314.775467

Graph for 1 Processor



Graph for 2 Processors



- **Discussion About selection of K**

As it can be seen selection of K doesn't change the execution time considerably. Although it's not drawn as a graph, increase in processor counts show the same pattern as explained before. Parallel (red) line is lower for 2 processes as it can be seen in the graphs above.

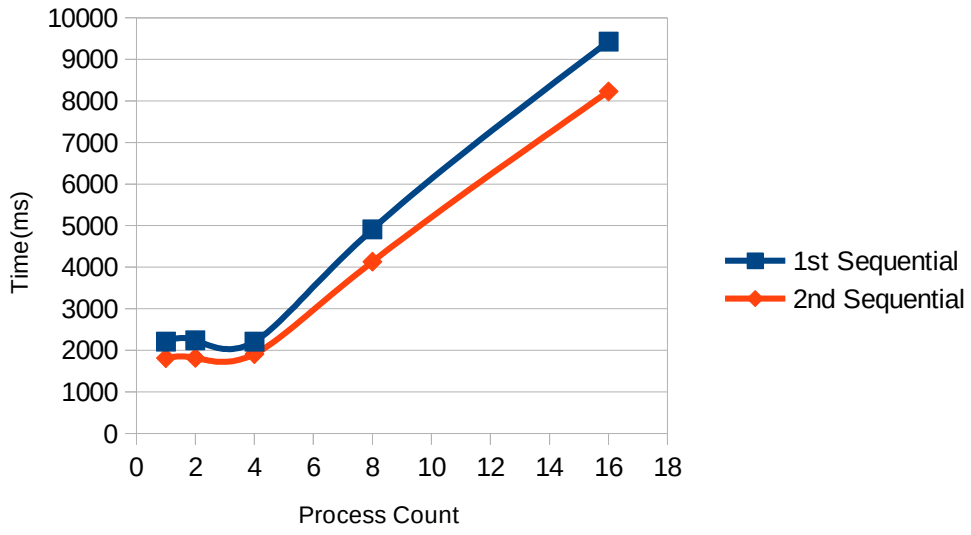
- **Effects of Dictionary Length and Document Count on the Parallel Execution**

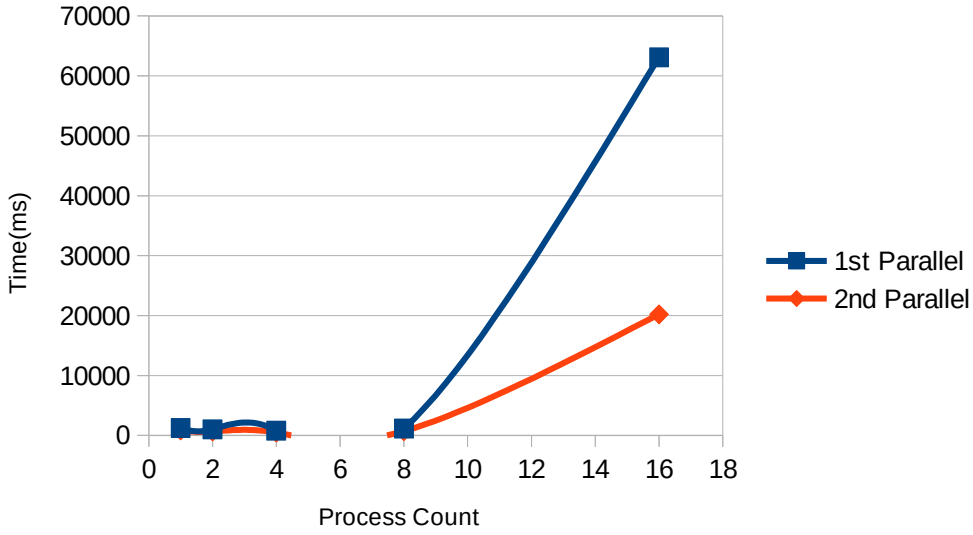
Similar to document count, this increases amount of data to be read and processed. Document count x2 increases row counts by 2 times, dictionary length x2 increases row size by 2 times. They have a similar effect on data size. Therefore following data, with document size 2M with dictionary length 4, and document size 1M with dictionary length 8 is compared.

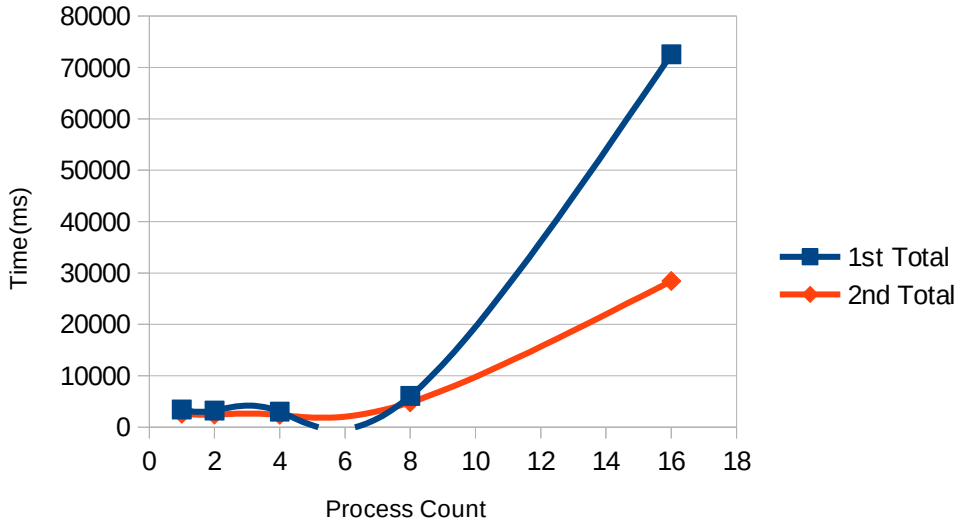
DocSize	2000000		
Dictionary Length	4		
k	4		
Proccessor count	Sequential	Parallel	Total
1	2206.599474	1212.07881	3418.683052
2	2237.687588	1003.763199	3241.455793
4	2207.108498	803.728819	3010.843515
8	4910.181046	1153.900862	6064.092636
16	9430.687666	63110.571384	72541.264534

DocSize	1000000		
Dictionary Length	8		
k	4		
Proccessor count	Sequential	Parallel	Total
1	1812.492371	818.365812	2630.864143
2	1816.076517	641.795874	2457.877398
4	1907.280922	478.472233	2385.762215
8	4130.916357	708.434343	4839.356184
16	8227.512598	20204.510212	28432.030678

In the following graphs, 1st one refers to the data in the first table with DocSize = 2M and Dictionary Length = 4, while 2nd one refers to the second one.







- **Discussion About Effects of Dictionary Length and Document Count on Parallel Run-time**

Both changes increase the data size by approximately the same amount (1st one has more ID fields as overhead). Sequential run-time doesn't change much as it can be seen from the first graph. However Parallel run time improves significantly for the 2nd one. This shows that parallel algorithm works better when same amount of data is distributed more in the rows than in the columns.