Homework 4

Group 16

Diversity Maximization on a Cloud

Marco Baggio
Matteo Migliorini
Nicolò Lazzaro
Gaia Grosso

May 31, 2018

Diversity Maximization on a Cloud

In this work we will show the performances reached by a Map-Reduce implementation coreset-based for Diversity Maximization problem. Our control parameters will be the final average distance output by the code; the time spent in building the coreset; the time for running the 2-approximation runsequential algorithm on the coreset.

Several variables can modify the results: on one hand we could vary the two free parameters required by the code, namely the number of centers to be found by Farther-First-Algorithm and the number of partitions applied by the map function to run it; one the other hand we could change the architecture on which to implement the code by setting different numbers of workers.

Making use of the Cloud they could manifest also other noisy factors which cannot be controlled by the client. They are due to the shared use of the service. We are going to show how all that have affected our outputs.

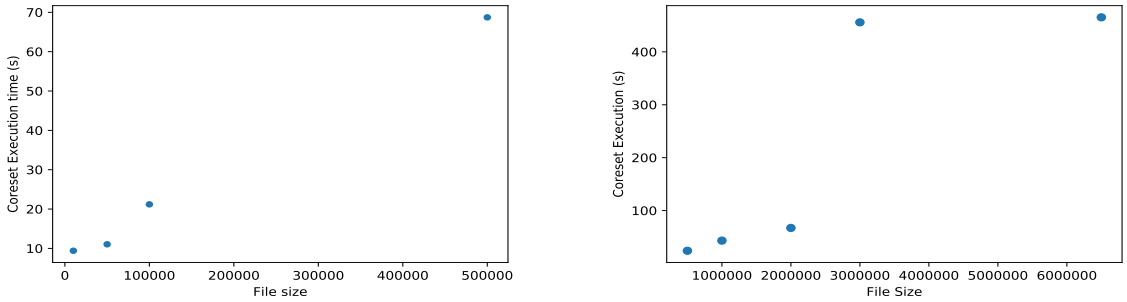
1 Analysis of parameters

1.1 Input dimension

Table 1.1.1: Output results varying the input dimension. The code has been run first in local and then on the Cloud, keeping `NumbBlocks` fixed at 25 and `Kcenters` at 10. As for the Cloud, we fixed the total executor at 16 and the worker at 4.

	|Input|	Average distance	Coreset execution (s)	Final execution (s)
Local	10000	5.7594	9.4023	0.0623
	50000	7.1763	11.0413	0.0607
	100000	8.6376	21.1915	0.0734
	500000	7.5280	68.7200	0.0749
Cloud	500000	7.71218	24.0088	3.1554
	1000000	6.18576	43.20350	3.4388
	2000000	8.04008	67.17744	3.2959
	3000000	8.33509	455.8071	3.4847

1.1.1 Coreset function response to different file sizes



(a) Local **(b)** Cloud

Figure 1.1.1: Analysis of dependence of Coreset Execution on the input data dimension.

From Figure 1.1.1 it is hard to spot a trend or to infer an asymptotic behavior but we can assert that the bigger the input size ($|Input|$) is the more time takes the Coreset building function to be executed.

We know anyway that `Kcenter` algorithm runs in time $O(K\,N)$, where K is the number of centers to be computed and N corresponds to $\frac{|Input|}{\texttt{NumBlocks}}$. In our case, as we varied $|Input|$ but we kept the number of partitions (`NumBlocks`) fixed, we expect that the dominant term in coreset building time is proportional to the size of the file. This is almost evident for executions run in local but not for those run on the Cloud. As we will see in next sections variability in Cloud

output is so high that we cannot say much about the behaviors we observed using the Cloud without statistics.

1.1.2 runSequential function response to different file sizes

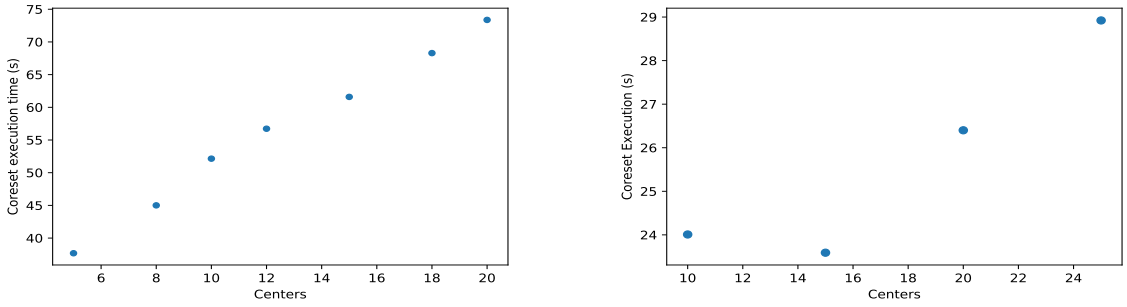
From Figure 1.4.1 is possible to see that even with different file sizes the running time behavior of `runSequential` remains unchanged, therefore we can suppose that the amount of data analyzed does not influence the time taken by the function in order to execute.

1.2 Number of centers (K)

Table 1.2.1: Output results varying the number of centers. The code has been run first in local then on the Cloud, giving as input 500000 points and keeping `NumbBlocks` fixed (`NumbBlocks`=10 in local, `NumbBlocks`=25 on the Cloud). As for the Cloud, we fixed the total executor at 16 and the worker at 4.

	Centers	Average distance	Coreset execution (s)	Final execution (s)
Local	5	7.355579	37.68164	0.04378
	8	6.170594	45.00481	0.20526
	10	7.27188	52.14808	0.43638
	12	7.62739	56.72569	0.71331
	15	7.61347	61.59248	1.26588
	18	7.83821	68.29197	2.33369
	20	7.97064	73.36821	3.61232
Cloud	10	7.7121	24.0088	3.1554
	15	7.4801	23.5904	10.295
	20	7.1687	26.4005	26.291
	25	7.2590	28.9218	47.867

1.2.1 Coreset function response to number of centers changes

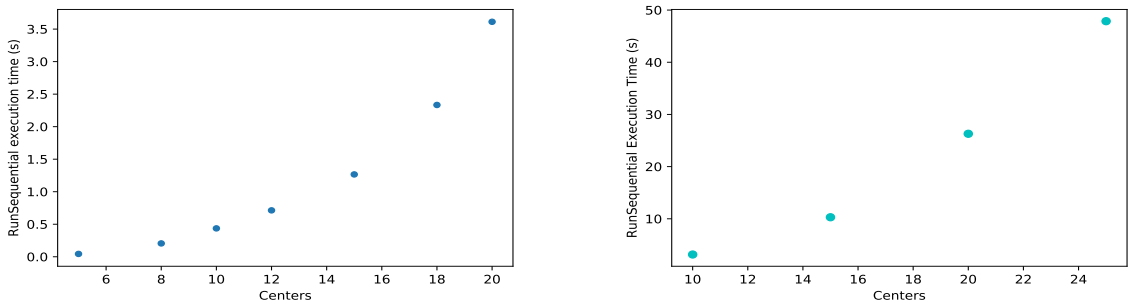


(a) Local **(b)** Cloud

Figure 1.2.1: Analysis of dependence of Coreset Execution on the number of chosen Centers.

As seen in the graphs in Figure 1.2.1 even if the Cloud results are not completely clear, by using the results obtained in the local machine to reinforce our hypothesis, we can safely confirm that there is a direct correlation between Coreset-function time execution and the number of centers to compute.

1.2.2 runSequential response to number of centers changes



(a) Local **(b)** Cloud

Figure 1.2.2: Analysis of dependence of RunSequential Execution on the number of chosen Centers.

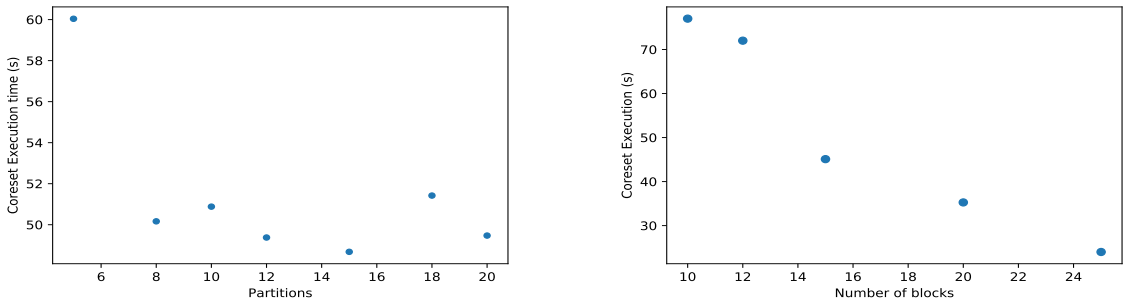
While playing around with the number of centers K required we had the possibility to confirm that the function `runSequential` behaves like a $O(\frac{K^3}{2})$ as shown by the analysis of its complexity and the graph produced shown above.

1.3 Number of partitions (`NumBlocks`)

Table 1.3.1: Output results varying the number of partitions. The code has been run first in local then on the Cloud, giving as input 500000 points and keeping the number of centers fixed at 10. As for the Cloud, we fixed the total executor at 16 and the worker at 4.

	Partitions	Average distance	Coreset execution (s)	Final execution (s)
Local	5	6.81101	60.0399	0.1004
	8	6.81101	50.1655	0.2718
	10	7.2718	50.8818	0.4113
	12	6.8110	49.3761	0.6035
	15	7.9115	48.6785	0.9150
	18	6.0319	51.4214	1.3740
	20	7.9181	49.4714	1.62350
Cloud	20	7.7344	35.2652	2.1530
	15	7.7492	45.1001	1.1944
	10	8.1251	77.0155	1.0959
	25	7.7121	24.0088	3.1554

1.3.1 Coreset building function response to number of partition changes



(a) Local **(b)** Cloud

Figure 1.3.1: Analysis of dependence of Coreset Execution on the number of chosen Partitions.

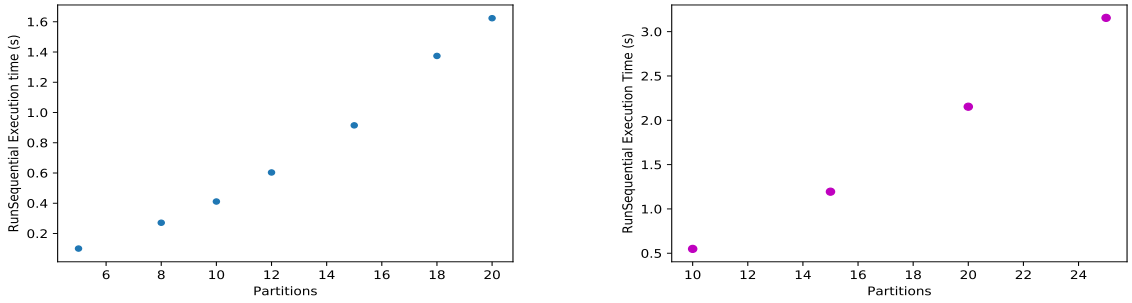
Figure 1.3.1 shows different behaviors for executions run in Local and those run on the Cloud. We expect Coreset building time to depend on the choice of partitions (`NumBlocks`), as this function is built using a Map-Reduce algorithm.

`Kcenters` algorithm we exploited to find the coreset is $O(k\,N)$, where K the number of centers to be selected and N the input size, namely $\frac{|Input|}{\texttt{NumBlocks}}$ when we call the function for each partition. Since input file have been fixed we should see that the Coreset building execution time decreases linearly as the `NumBlocks` size increases and so we expect our data to fit a model like

$$t_{Coreset} = a - b\,\texttt{NumBlocks}$$

This is verified for the Clooud, but no evident dependences are visible in Local runs. This is justified by the fact that running in Local we have only one worker so even if a partition is apllied to the data, each work is then executed by the same core. So in fact Map-Reduce implementation is useless for a single core execution.

1.3.2 runSequential response to number of partitions changes



(a) Local **(b)** Cloud

Figure 1.3.2: Analysis of dependence of RunSequential Execution on the number of chosen Partitions.

The function runSequential takes as input the number of centers K and the Coreset, whose dimension is given by (K `NumBlocks`). As `runSequential` execution time is of the order $O(K\,|Input|^2)$, we finally expect the time of execution to be proportional to (K^3`NumBlocks`^{2}).

$$t_{runSeq} \sim K^3\texttt{NumBlocks}^2$$

As expected, followed the lines of an $O($`NumBlocks`$^2)$ function. We can see that in Figure 1.3.2 representing the results achieved during testing both in Local and in Cloud.

1.4 `NumBlocks` × `K`

1.4.1 runSequential response to a simultaneous increase in both centers and partitions numbers

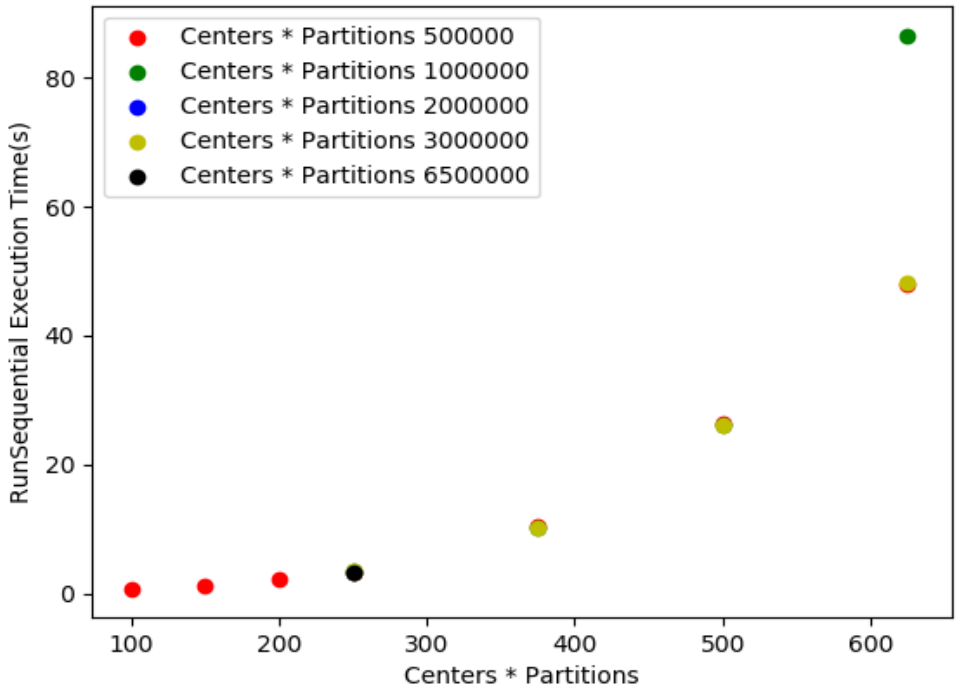


Figure 1.4.1: Analysis of `RunSequential` execution time dependence on K `NumBlocks`

Figure 1.4.1 shows a good proof of the expected behavior of `runSequential` to a simultaneous increase in both centers `k` and partitions `NumBlocks`.

From this graph is also possible to see that even with different file sizes the running time of `runSequential` remains unchanged, therefore we can suppose that the amount of data analyzed does not influence the time taken by the function in order to execute (in the graph there are actually some barely visible overlapping points).

1.5 Core Executors (workers)

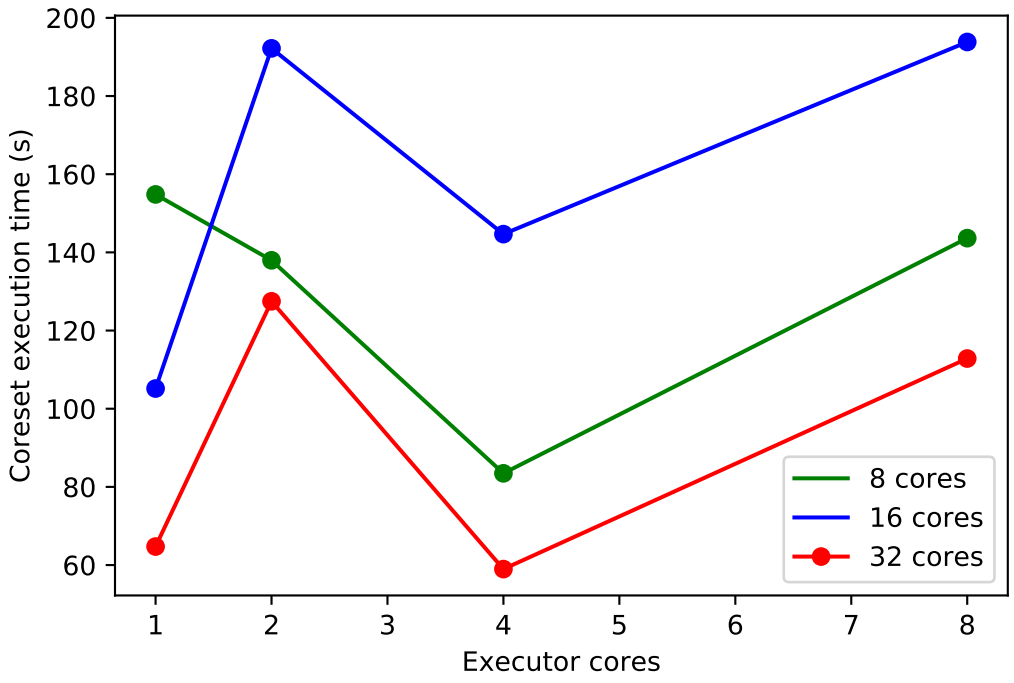


Figure 1.5.1: Analysis of Coreset-building execution dependence on the number of Executor cores.

From the results obtained by varying the total number of cores involved in the job and the cores assigned to each worker (hence the number of workers) it is hard to infer clear informations. While with one core per worker each execution behave like expected (more cores equal more efficiency hence less time taken, when we add more cores to each worker it seems to maintain the expected behavior only for high number of total cores. A strange behavior is also encountered when increasing the number of workers (*total cores/cores per executor*). Even if it seems that increasing the number of workers improves the time taken by the job this performance is not always maintained.

1.6 Robustness and Variance

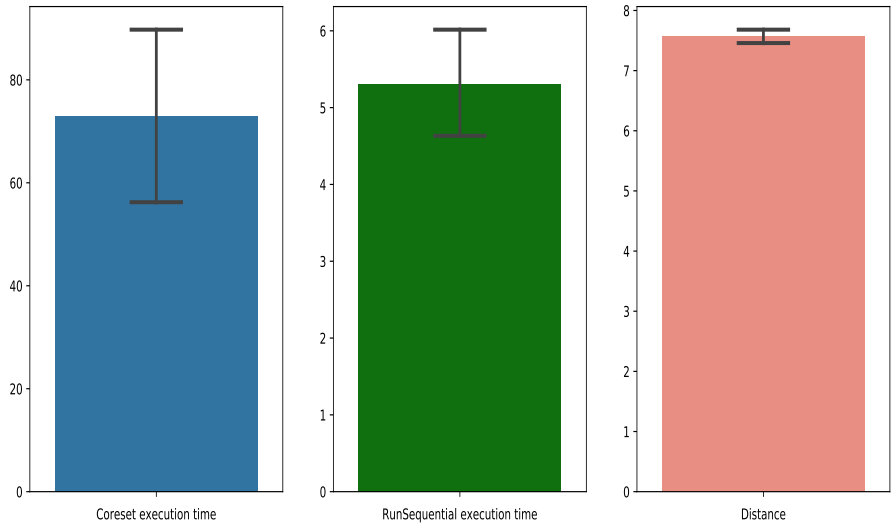


Figure 1.6.1: Analysis of control parameters variability.

Last thing we decided to test the robustness and variance of results obtained via the Cloud. As expected, the computation of the distance, being the most lightweight operation of the job, took far less time than the others and was affected to less fluctuations. Meanwhile, both `runSequential` and coreset-execution took a significant amount of time to be completed. The first one due to order $O(k^3n^2)$ and the other one due to high parallelism.

Moreover both them were more susceptible to the performance of the cores involved, which could vary depending on the resources available at the time we executed the code. In particular we've experienced an higher variance in this jobs when the cloud was overcrowded, specifically in the Coreset-execution function being the most parallelism-heavy operation.