Parallel BIRCH Clustering on Spark

Xingyu Zhou
Computer Science Department
University of California, Los Angeles
Email: xingyuhit@gmail.com

***Abstract*—BIRCH is a popular clustering algorithm which excels at clustering data with only single pass of input data in condition of limited memory. This paper presents an implementation of parallel BIRCH clustering algorithm on Spark which takes advantages of BIRCH and Spark parallel engine, resulting in scalable performance without compromising its accuracy.**

Keywords—BIRCH; parallel; clustering algorithm; Spark

I. INTRODUCTION

Clustering algorithms have been studied for decades, yet we see recent improvements with new emergent technology and faster processors. BIRCH[1, 2] (Balanced Iterative Reducing and Clustering using Hierarchies), is an excellent algorithm in terms of processing for very large datasets and requiring only one (or two) passes of data IO. In addition, while the study and application of parallel computing has become increasingly popular these years, MapReduce and Spark[3] for instance, clustering algorithms have been through the same revolution by embracing parallel computing. Although KMEANS, a classical clustering algorithm, has been already transplanted to parallel platform, there are plenty of other clustering algorithms which are as good but yet to take the advantage of it. BIRCH is one of them which has great potential to leverage the parallel computing to improve its performance without compromising its accuracy.

A parallel BIRCH clustering algorithm implemented on Spark is presented in this paper. It is an improved version of PBIRCH[4], and instead of using arbitrary parallel models, Spark is used as the parallel platform due to its popularity and convenience. The rest of the paper is organized as follows. Section 2 briefly introduces relevant background including BIRCH and Spark. Section 3 elaborates the algorithm and architecture of the parallel BIRCH and the way it works on Spark. Section 4 presents the experimental results. Finally the conclusion is presented in Section 5.

II. RELEVANT STUDY AND BACKGROUND

A. BIRCH Algorithm

BIRCH is created to solve the problem of previous clustering algorithms which might require multiple scan of databases due to memory limit. BIRCH uses the concept of Cluster Feature (CF) to condense the information of subcluster of points without compromising its accuracy of computing. There are four phases in this algorithm. In phase 1, a height-balanced tree is built with each CF node has multiple CF entry, and a CF entry has a child CF

node. The leaf CF entries are the objective condensed data that could be utilized in the next process. Phase 2 is optional which is only used to prune the nodes in order to prepare for some algorithms that has certain constraints in phase 3. Previous one (or two) steps solve the problem of space constraints, and now all the data are in memory in condensed form. In phase 3, theoretically any clustering algorithm could be used here to cluster the leaf entries as ordinary data. Phase 4 further corrects of centroids using the whole data (instead of the condensed data) and is optional.

A CF entry contains the number of points, linear sum and square sum, which are sufficient to compute other measures such as distances between entries. There are several parameters in this algorithm, such as the maximum number of branches (number of entries a node could have), the distance threshold that two entries can be merged, memory limit and memory check frequency. These parameters are important in the process of training. For instance, the memory limit indirectly determines the distance threshold since lower memory limit means larger distance threshold and resulting in less entries by merging entries. These can largely affect the accuracy and even the performance of this algorithm.

B. Spark

Spark is an open-source cluster computing framework that provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance. Due to its fast speed (100x faster than Hadoop MapReduce in memory, or 10x faster on disk), ease of use and generality, it has gained immense popularity in the area of the parallel

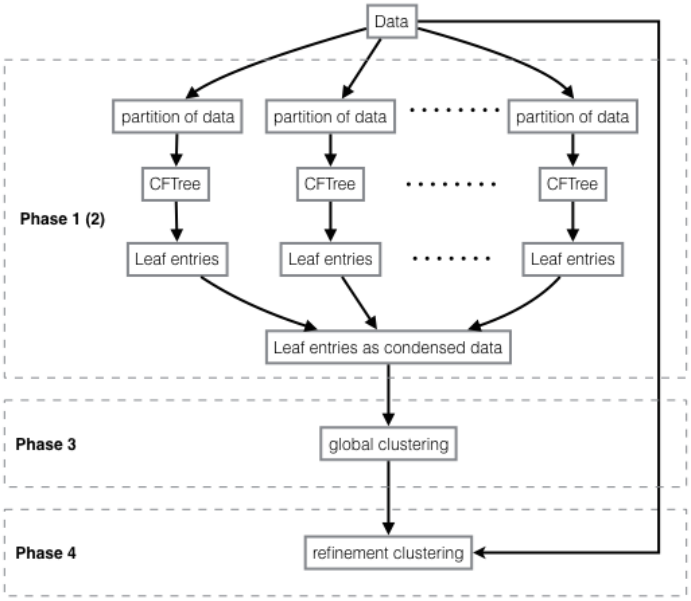

Figure 1. Parallel BIRCH on spark work flow

computing. Spark itself has implemented the parallel version of KMEANS on Spark in MLlib. But that is the only clustering algorithm implemented officially. Two options are given for choosing the initial centroids, namely random and kmeans||. Kmeans||[5] is an advanced algorithm to choose initial centroids. It is based on kmeans++[6], but adapts it to parallel computing environment and makes it more efficient.

III. PARALLEL BIRCH ON SPARK

The outline of whole algorithm is shown in Figure 1. As we can see, the algorithm takes four steps (or phases). The first two steps are the main part of BIRCH, which builds the CF (clustering feature) trees using a complex strategy to condense the data without losing the necessary information. In the parallel case, multiple trees are built instead of one in the original paper so that less data shuffling and network communication need in the process. Phase 3 uses the condensed data generated in the previous steps to do global clustering such as

agglomerative clustering or kmeans (kmeans is used here). Phase 4 is optional, which is to refine the clustering using the original data.

A. Phase One (Two)

Phase 1 is the essence of BIRCH algorithm. The data is loaded into memory and then used to build a CF tree. The parallel way of doing this is, first partitioning the data, sending them to clusters of machines, then for each partition of data, building each CF tree separately and in parallel. In this way, we have multiple CF trees instead of one. The discussion of whether using one CF trees or multiple CF trees is in the discussion part of this section.

The key of BIRCH is to condense data. In other words, a CF entry virtually incorporate multiple data points into one through merging process. The key of merging is the distance threshold. By increasing the distance threshold, the more chance entries are merged, resulting in less memory consumption. But overly high distance threshold may affect the accuracy of clustering results. To appropriately adapt the distance threshold to the memory limit is crucial to the balance of accuracy and memory consumption of this algorithm. The discussion of how to adapt the distance threshold dynamically and is properly presented in the discussion part of this section.

Phase 2 is merely to do the extra pruning and merging for the CF tree in order to prepare for particular conditions that might need in phase 3. In this case, it is optional and not implemented in this project.

B. Phase Three

After the CF trees have been built, a global clustering algorithm is used in this step. The CF leaf entries can be considered to be the condensed data, which will be clustered using KMEANS (it can be other global clustering algorithms, but KMEANS used here).

In the parallel environment, KMEANS is used differently from single-machine environment. Assume we have the initial centroids for K clusters (we will discuss how to choose initial centroids later). First, data is partitioned into multiple parts for clusters of machines and broadcast the initial centroids to these clusters. After that, for each partition, we should calculate the distance of each point to the cluster centroids, and assign each point to a cluster to which it has the minimum distance. Now we need to recalculate the centroids based on the points with newly-assigned cluster numbers. Since the points within the same clusters are crossing multiple clusters at this moment, there will be data shuffling (or centroids shuffling, which can be more efficient) so that data within the same cluster are in the same partition, and centroids can be calculated. These calculated centroids are then again broadcast to all the partitions and repeat the steps until the requirement is met (threshold is met, or reach maximum iterations).

A special design in this implementation is, if the number of CF entries is less than K, then the number of CF leaf entries will become the K, and there will be no need of training. In another special case when K <= 0, which means no presumed K is provided, then the number of CF leaf entries will be the K.

As for choosing initial centroids, there are at least two approaches, one is randomly choosing K points from data, the other is kmeans||. The kmeans|| is chosen for its high accuracy and fewer iterations of later steps. As this paper mentioned before, KMEANS along with kmeans|| algorithm has been

implemented in MLlib of Spark library, which is convenient. The detailed discussion of these two methods can be seen in the discussion part of this section.

C. Phase Four

Phase 4 needs an additional pass of the whole data, which fortunately is optional. But this pass may improve the quality of clustering. Basically, KMEANS is used to cluster the data again but with the initial centroids inherited from phase 3. The accuracy of centroids may be improved in this step.

D. Discussion

The first question is whether we build a CF tree or build multiple CF trees in this parallel algorithm. According to [7], there is only one copy of CF tree in memory, while different processors / cores / GPU units are building the tree alternately. But when it spans multiple machines, it is nearly impossible to do so due to the large amount of shuffling data to keep one copy of CF trees. And more importantly, it fails to use the multiple memory resources in this process. Thus, I choose to build multiple CF trees for each partition of data instead of keeping only one copy of it.

Another question is how to adapt distance threshold dynamically and properly through the CF tree building process. First, it is important to know that the memory limit parameter is the final factor that determines the quality of adapting process. But it is still important to do it carefully since it is easy to let it go wrong. For instance, I used the average distance of closest pair entry of all nodes to be the estimate of expected distance threshold since the closest pair is the most likely to be merged. But it could be wrong in the parallel environment, as the data are more sparse and with more arbitrary order. In this case, the distance of closest pair could be very large, thus make the distance threshold unreasonably large. To avoid this, I computed the reference distance threshold in the process by comparing the current memory usage and the memory limit parameter, and then use it to constrain the maximum distance to be the double of the reference one. On the other hand, when the distance of closest pair entries is too small, the reference distance can be used to be the lower boundary of it.

The third question is whether to randomly choosing initial centroids or to use kmeans||. The answer is undoubtedly in favor of the latter. It is by instinct that kmeans|| is better than the randomly choosing, but by experiments in this paper it is proven to be true. The result is that the accuracy is either much better or slightly better depending on the luckiness of random choosing process and the quality of data.

IV. Experimental Study

Both performance and accuracy of this parallel BIRCH on spark implementation are tested. The time cost for each phase or of the total is used for measuring performance. As for accuracy, the Adjusted Rand Index[8] (ARI) is used to measure the similarity between the ground-truth labels and the computed labels.

The experimental study is conducted in both local environment and cluster environment (clusters of machines).

A. Influence of Noise Ratio of Data

In order to know the influence of noise ratio of input data on performance and accuracy, experiments with different noise ratio data have been done using 100,000 points of data with K (# of

Table 1. Test with different noise ratio

NOISE RATIO	ACCURACY (ARI)	TIME (SEC)
0%	0.999973	12.39
1%	0.989914	12.60
5%	0.949538	11.89
10%	0.895934	10.70

clusters in the data) = 4, Dim (# of dimension) = 2, number of partitions = 8.

From the Table 1, we can see that the results are desirable since as the noise ratio raised from 0% to 10%, the accuracy dropped approximately 10%. Surprisingly, The performance is increasing when there is more noise added. It may result from more data are merged due to the increasing of distance threshold as noise data are usually irregular and scattered.

B. Influence of Number of Clusters (K) for Data

This is the experiments with different number of clusters (K) for data have been done using 100,000 points of data with noise ratio = 5%, Dim = 2, number of partitions = 8.

Table 2 shows that as the K increases from 4 to 128, the accuracy drops from 0.95 to 0.7. One of the possible explanations is that data are more cramped due to more clusters of data, and the boundary becomes more vague as the K increase. The influence of K on (time) performance is trivial from this result.

C. Influence of Number of Dimensions for Data

Next are the experiments for input data with different number of dimensions. 100,000 points of data with noise ratio = 5%, K = 4, number of partitions = 8.

Table 2. Test with different K

K	ACCURACY (ARI)	TIME (SEC)
4	0.949538	11.89
16	0.874591	14.47
64	0.761516	14.95
128	0.705336	15.17

As we can see from Table 3, the dimension affects the accuracy and performance in an unexpected way where from d = 2 to d = 8 the accuracy and performance decreased but from d = 8 to d = 16 they are slightly increased.

Table 3. Test with different dimension

DIMENSION	ACCURACY (ARI)	TIME (SEC)
2	0.949538	11.89
8	0.674310	14.63
16	0.705198	13.13

D. Influence of Number of Cores for Spark

Table 4 and Figure 2 are for the experiments on local machine to test the influence of number of cores on performance, and Table 5 and Figure 3 are on accuracy.

Table 4. Performance result with different # of cores

# OF CORES	PHASE 1 (2)	PHASE 3	PHASE 4	TOTAL
1	11.00	9.83	4.47	25.29
2	7.58	8.06	3.66	19.31
3	8.99	7.36	0.47	16.82
4	7.06	7.09	3.41	17.56

Table 5. Accuracy result with different # of cores

# OF CORES	ACCURACY (ARI)
1	0.784676
2	0.830251
3	0.875787
4	0.828223

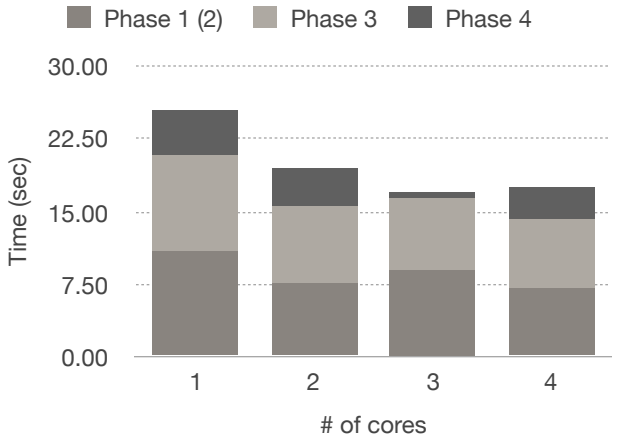

Figure 2. Performance result with different # of cores

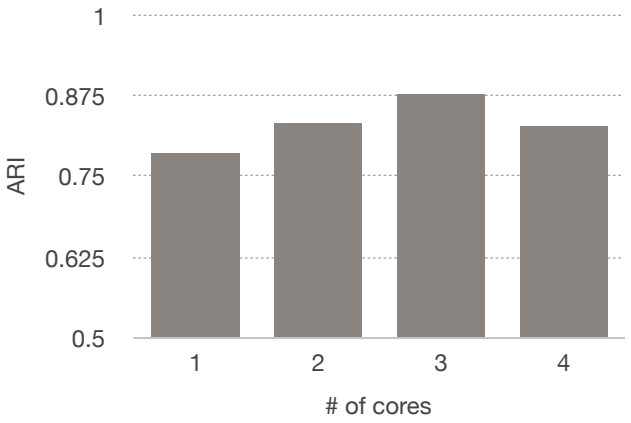

Figure 3. Accuracy result with different # of cores

The performance result is as expected, i.e. the time cost goes down as more cores are involved in. The accuracy varies from 0.78 to 0.88, which is acceptable fluctuation of randomness inside the algorithm, but it's interesting to notice that the case of 3 cores has the most accurate result.

E. Influence of Number of Partitions for Data

The result of experiments on different number of partitions is shown in Table 6 and Figure 4 for performance, and Table 7 and Figure 5 for accuracy. These experiments are done using a local machine with 4 cores.

As we can see, the number of partitions has huge impact on performance. For a 4 cores machine, at least 4 partitions are needed in order to achieve good

Table 6. Performance result with different # of partitions

# OF PARTITIONS	PHASE 1 (2)	PHASE 3	PHASE 4	TOTAL
1	110.20	3.29	2.65	116.13
2	108.61	4.22	2.25	115.08
4	9.06	5.30	2.53	16.90
8	6.95	6.84	3.40	17.19
12	6.43	10.01	3.61	20.05
16	7.23	9.86	0.63	17.72
20	7.30	12.92	2.65	22.88
24	7.69	13.48	5.10	26.27

Table 7. Accuracy result with different # of partitions

# OF PARTITIONS	ACCURACY (ARI)
1	0.784955
2	0.829051
4	0.828504
8	0.784479
12	0.828946
16	0.785177
20	0.784344
24	0.785036

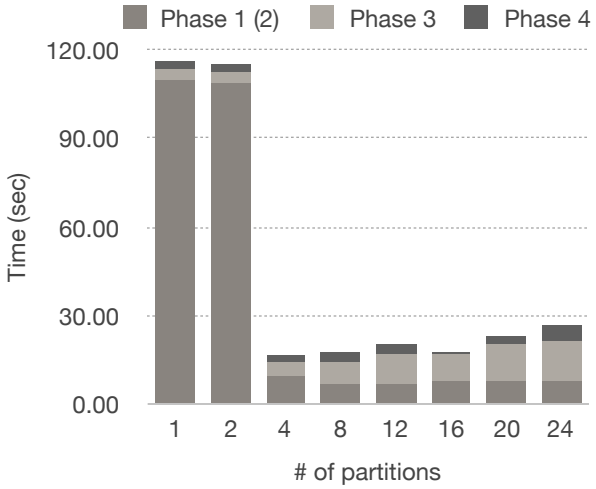

Figure 4. Performance result with different # of partitions

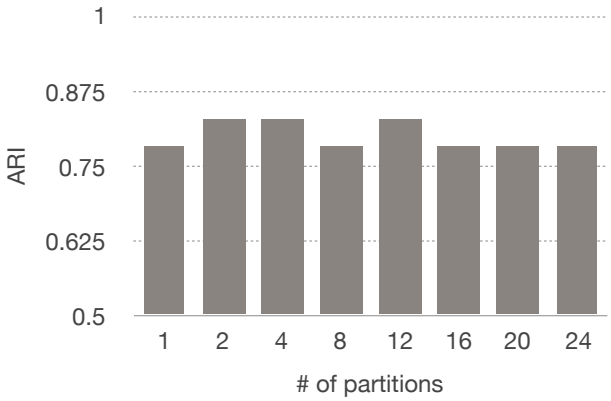

Figure 5. Accuracy result with different # of partitions

performance according to the results. However, its influence on accuracy is not obvious.

F. Influence of Memory Reference Limit Per Tree

The result of experiments on local machine (4 cores) is shown in Table 8 and Figure 6 for performance, and Table 9 and Figure 7 for accuracy.

As we can see, the performance increase as the memory reference limit for each CF tree increase. Since more memory available, less tree rebuilding processes needed, thus less time spent. The accuracies for various memory limit are relatively stable. Therefore, it is safe to say that memory limit does not affect accuracy as expected.

Table 8. Performance result with different memory per tree

MEMLIM	PHASE 1 (2)	PHASE 3	PHASE 4	TOTAL
16M	32.81	6.95	2.54	42.31
32M	25.09	6.16	2.71	33.96
64M	8.40	6.95	3.21	18.56
128M	6.61	7.16	2.42	16.19
256M	6.66	6.66	1.04	16.39
512M	7.31	8.09	3.17	18.57
1G	6.75	6.94	2.70	16.39

Table 9. Accuracy result with different memory per tree

MEMLIM	ACCURACY (ARI)
16M	0.828368
32M	0.829844
64M	0.783339
128M	0.830310
256M	0.782063
512M	0.829711
1G	0.784598

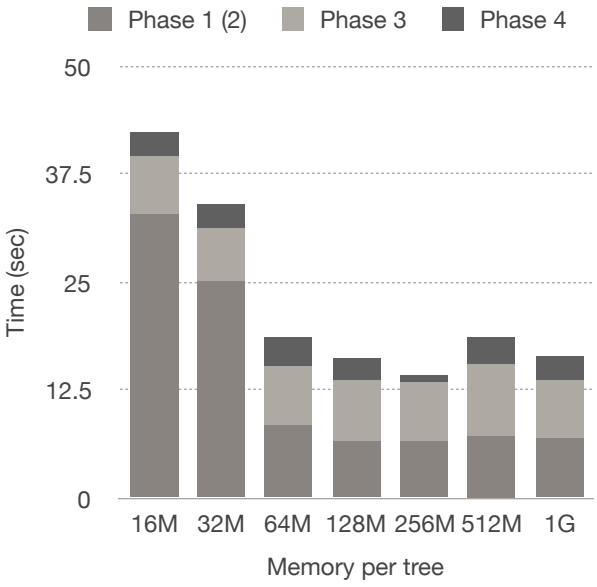

Figure 6. Performance result with different memory per tree

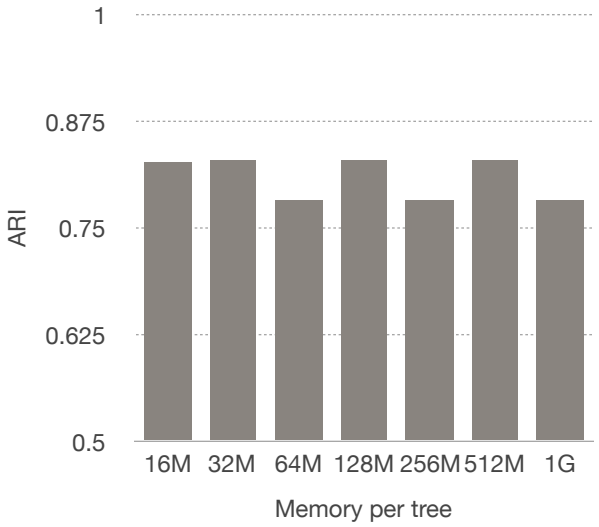

Figure 7. Accuracy result with different memory per tree

G. Scalability

The most important test should be how this algorithm scale with different number of machines used since it is a parallel algorithm implemented on Spark. First, we can see how it scales using different number of points running with different number of machines. Next, we can see if the scalability is sensitive with dimensionality, number of clusters (K), or noise ratio of input data. 10 million datasets (K = 16, noise ratio = 5%, dimension = 8) are used here except for specific setting for tests (for instance, 1 million and 100 thousand points are used in this scalability test w.r.t different number of points, and dimension = 2 and dimension = 16 are used in sensitivity test for dimensionality, and etc.)

Table 10 and Figure 8 shows the scalability result over different number of points.

Basically, the time cost goes down when there are more machines running. However, since there are more overheads when more machines involved, the performance speedup is affected. There is nearly no speedup for 100,000 points (in fact, time cost goes up slightly), a big speedup on 1,000,000 points from 1 machine to 2 machines, which stops at 4 machines,

Table 10. Scalability over # of points w.r.t # of machines

# OF POINTS	1 MACHINE	2 MACHINES	4 MACHINES	8 MACHINES	15 MACHINES
100000	12.33	13.41	15.75	21.67	27.98
1000000	516.59	152.28	24.53	25.45	32.16
10000000	893.98	794.86	651.31	576.35	413.63

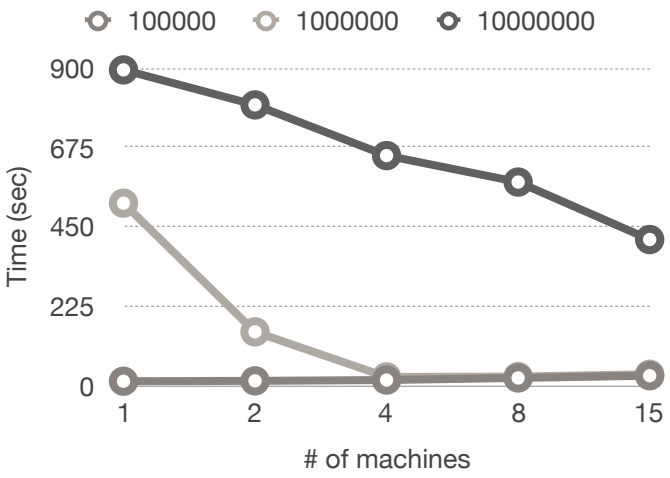

Figure 8. Scalability over # of points w.r.t # of machines

and stable but slower speedup for 10,000,000 points. Therefore, the result shows that larger number of points has a great potential to be fast enough when there are unlimited number of machines given, and smaller data sets can quickly reach the optimal performance point with small amount of machines.

Table 11 and Figure 9 shows the scalability result of sensitivity test for dimensionality.

Table 11. Sensitivity to dimensionality w.r.t # of machines

DIMENSION	1 MACHINE	2 MACHINES	4 MACHINES	8 MACHINES	15 MACHINES
2	691.19	413.00	251.52	191.32	57.75
8	893.98	794.86	651.31	576.35	413.63
16	1205.05	670.98	528.69	396.77	305.00

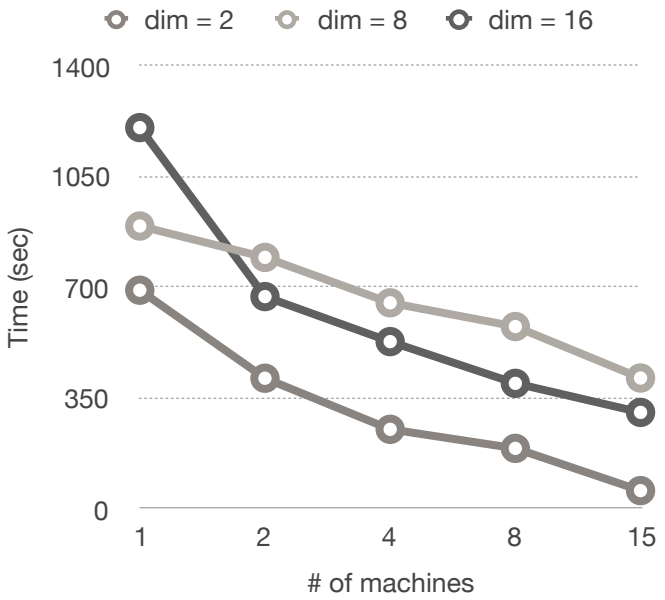

Figure 9. Sensitivity to dimensionality w.r.t # of machines

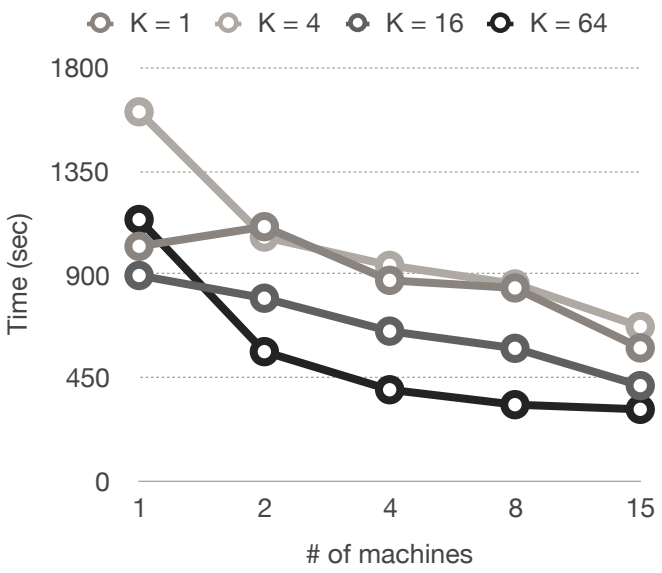

Figure 10. Sensitivity to # of clusters (K) w.r.t # of machines

The result shows that the scalability remains to be good for data with different dimensions. It is worth noticing that with 15 machines, 2 dimensional points speedup nearly linearly with the number of machines if communication overheads are not taken into consideration.

Table 12 and Figure 10 shows the scalability result of sensitivity test for K (# of clusters). As we can see, the scalability is not affected by the number of clusters.

Table 13 and Figure 11 shows the scalability result of sensitivity test for dimensionality. As it is

Table 12. Sensitivity to # of clusters (K) w.r.t # of machines

K	1 MACHINE	2 MACHINES	4 MACHINES	8 MACHINES	15 MACHINES
1	1021.33	1107.05	872.86	839.71	578.15
4	1608.10	1061.71	938.10	857.33	671.35
16	893.98	794.86	651.31	576.35	413.63
64	1138.26	561.84	395.67	330.54	310.23

Table 13. Sensitivity to noise w.r.t # of machines

NOISE	1 MACHINE	2 MACHINES	4 MACHINES	8 MACHINES	15 MACHINES
0%	821.61	636.45	580.07	538.08	368.66
1%	953.78	689.64	619.88	525.66	406.36
5%	893.98	794.86	651.31	576.35	413.63

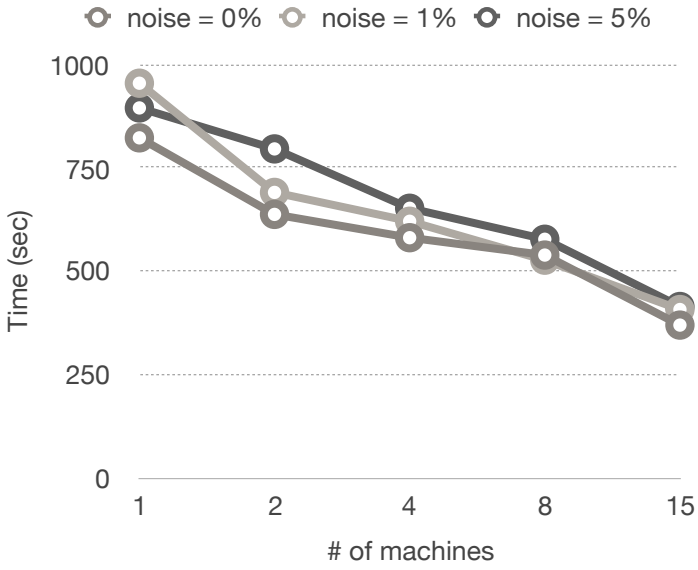

Figure 11. Sensitivity to noise w.r.t # of machines

shown, there are few differences among different noise ratio in terms of performance and scalability.

In conclusion, this parallel BIRCH clustering algorithm implemented on Spark scales with multiple machines, and the scalability does not sensitive to noise ratio, dimensionality and K (number of clusters in the data).

V. Conclusion

This paper presented parallel BIRCH on Spark, which combines the advantages of both, i.e. only one (or two) passes over data with limited memory and noise resistance in BIRCH, and scalable performance in parallelism and Spark platform. This results in a desirable performance without compromising its accuracy. This project fills the blank that no parallel BIRCH clustering implementation on Spark currently exists. In addition, a wide range of experiments has been done and the results are promising.

Acknowledgment

I would like to express my gratitude to my advisor Prof. Carlo Zaniolo for the support of my master project. Furthermore I would like to thank Max Mazzeo for introducing me to the topic as well as for the support on the way. I also would like to thank Matteo Interlandi for helping me do the experiments with clusters of machines. Last but not least, I would like to thank everyone who helped me through the process and gives review to my project paper.

References

1. Birch, Zhang T. "an efficient data clustering method for very large databases/T. Zhang, R. Ramakrishnan, M. Livny." Proc. of the ACM SIGMOD Conf. on Management of Data.–Montreal: ACM Press. 1996.
2. Zhang, Tian, Raghu Ramakrishnan, and Miron Livny. "BIRCH: A new data clustering algorithm and its applications." Data Mining and Knowledge Discovery 1.2 (1997): 141-182.
3. "Apache Spark." Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc. 8 March 2016. Web. 8 March 2016. <https://en.wikipedia.org/wiki/Apache_Spark>
4. Garg, Ashwani, et al. "PBIRCH: a scalable parallel clustering algorithm for incremental data." Database Engineering and Applications Symposium, 2006. IDEAS'06. 10th International. IEEE, 2006.
5. Bahmani, Bahman, et al. "Scalable k-means++." Proceedings of the VLDB Endowment 5.7 (2012): 622-633.
6. Arthur, David, and Sergei Vassilvitskii. "k-means++: The advantages of careful seeding." Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2007.
7. Dong, Jianqiang, Fei Wang, and Bo Yuan. "Accelerating BIRCH for clustering large scale streaming data using CUDA dynamic parallelism." Intelligent Data Engineering and Automated Learning–IDEAL 2013. Springer Berlin Heidelberg, 2013. 409-416.
8. Santos, Jorge M., and Mark Embrechts. "On the use of the adjusted rand index as a metric for evaluating supervised classification." Artificial neural networks–ICANN 2009. Springer Berlin Heidelberg, 2009. 175-184.