Bisecting k-Means Clustering with Thrill

Oleg Schell

Abstract

The k-Means algorithm is a relatively old but still one of the most popular and most used methods to solve clustering problems. Although it does not always terminates in the optimal solution, the inherent speed and simplicity reason its fame. By taking the advantages of hierarchical clustering methods and combining them with the k-Means approach, the shortage in output quality can be diminished. The resulting bisecting k-Means algorithm [Stein00] promises to deliver results of equal quality to hierarchical algorithms by still being simple and fast. Thrill [Thr16], a new and emerging big data processing framework provides an example for the native k-Means algorithm as part of the tutorial. In order to investigate the impact of the new and enhanced procedure, we extended the existing example by adding the possibility to perform the bisecting k-Means method on the input data. In this context we compared the result quality as well as the execution time of both algorithms using the Thrill framework showing that the result quality can be enhanced for cluster sizes close to the ground truth number of clusters by sacrificing a little speed.

1 Introduction

Clustering algorithms have been designed to group n data points $\in \mathbb{R}^d$ with a high similarity into a cluster. In the case of k-Means we choose and iteratively relocate k centroids $\in \mathbb{R}^d$ so, that the total distance of each data point to its closest centroid in minimal. Generally, finding the optimal solution for this certain problem is of theoretical complexity which is NP-hard. Nevertheless, in 1982 Stuart P. Lloyd [Lloyd82] developed an algorithm generating good clusters while being both fast and simple. After having chosen k uniformly random data points as the initial centroids, each remaining input point is assigned to its nearest one. The centroids for the next iteration are then calculated by averaging the positions of all data points related to each centroid of the previous iteration. These two steps of assigning the data points and calculating the new median points are repeated a fixed number of times or until the centroids do not change anymore. Since the total square distance of all points is monotonically decreasing and the number of possible solutions is bound to k^n, the algorithm always terminates. This approach is also known as Lloyds algorithm.

Algorithm 1 Thrill k-Means

```
1: function KMeans(points, d, k, iter)
2:     i ← 0
3:     centroids ← select k random centroids from points
4:     while i < iter do
5:         Calculate closest centroid for each point
6:         centroids ← determine new centroids
7:         i ← i + 1
8:     return centroids
```

2 Available k-Means implementation in Thrill

In times of Big Data, the number of dimensions and the amount of instances increased dramatically. Nonetheless, the k-Means algorithm is still in use due to its high scalability. Further, the assignment of points to the centers is data independent and thus fully parallelizable. Numerous parallel approaches and implementations for this method exist, one of them realized in the Thrill framework. Thrill is an open-source C++ framework for algorithmic distributed batch processing using template meta-programming. By providing a library of scalable algorithmic primitives like *Sort*, *Map* or *ReduceByKey*, more sophisticated algorithms can be developed and implemented. The k-Means example provided by Thrill implements the above stated Lloyds approach and can be seen in Algorithm 1. The parameters of the function are the input data *points* having dimension d, the number of desired clusters k and the fixed number of iterations *iter*. The distributed processing units firstly select k initial centroid from the data points and exchange them. Afterwards each unit independently processes a part of the input by assigning each point to the centroid with the least squared distance. With Thrills *Map*-routine the number of assigned points for each centroid is counted which is then used to determine the new centroids by averaging the clustered points with Thrills *Reduce*-method. After *iter* iterations the determined centroids are returned to the calling function.

3 Bisecting k-Means - General approach

Besides the iterative clustering techniques like k-Means there are also hierarchical methods which determine the result in another manner. They establish a tree-like structure with one cluster containing all instances as the root cluster and individual data points as the leaves. By either splitting clusters in a top-down approach or merging the individual points into bigger cluster in a bottom-up approach, intermediate clusters of the tree can be determined. This kind of clustering yields a better solution but on the other hand has a quadratic time complexity. In comparison, the k-Means approach is relatively fast generating a passable result. The authors of [Stein00] discovered that the combination of both methods result in a fast algorithm with results comparable to hierarchical clustering. The elaborated algorithm was called **Bisecting k-Means** due to its way of working. At the beginning, all data points are considered to be in one huge root cluster. By performing the basic k-Means with $k = 2$ on a chosen cluster, it is split into two smaller ones. As they tested different metrics for which cluster to split, they decided to take the one with the highest number of assigned points. But before a determined solution is accepted in an iteration, the splitting procedure is performed a predefined number of times with taking the solution with the most similarity. This bisecting step of splitting the biggest cluster is performed $k - 1$ times in order to determine k clusters.

Algorithm 2 Our bisecting k-Means implementation

```
1:  function BisecKMeans(points, d, k, iter)
2:      size ← min(k, 2)
3:      centroids ← KMeans(points, d, size, iter)
4:      while size < k do
5:          Calculate closest centroid for each point
6:          bigC ← determine biggest cluster
7:          fPoints ← points of biggest cluster
8:          centroids ← centroids \ bigC ∪ KMeans(fPoints, d, 2, iter)
9:          size ← size + 1
10:     return centroids
```

4 Bisecting k-Means implementation in Thrill

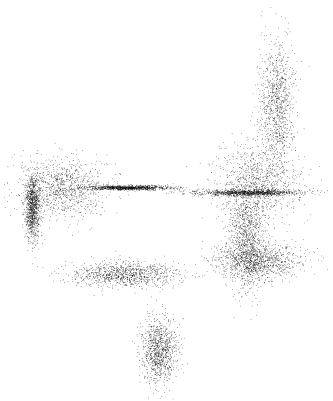
The above described method of bisecting k-Means was taken and implemented with functions from the Thrill library. Since a k-Means example already exists in the framework, we decided to additively add the new approach. The user then can choose between the normal and bisecting k-Means by passing the appropriate argument at program start. Due to the fact that normal k-Means is an integral part of the bisecting method and the calling structure is the same for both approaches, we decided to keep the available signatures and design our bisecting implementation with respect to the existing functions and structures. As for our implementation we set the number of performed refinement steps to one in order to be even more efficient for large input sets compared to native k-Means. Here, we save time since only the points in the biggest cluster are evaluated as opposed to comparing the whole data set to all centroids. The procedure of our Thrill implementation is outlined in Algorithm 2. The implementation receives the same parameters as the existing k-Means algorithm and firstly splits the cluster containing all data points into two new cluster if $k >= 2$. Since we now need to know which of the existing cluster the biggest one is, we have to assign all data points to the current centroids. Here, we use the already existing helping functions provided by the example. Afterwards, the biggest cluster is determined by counting the number of assigned data points for each centroid with the *ReduceByKey*-function and broadcasting the cluster identifier with the most instances to all processing units with *AllReduce*. In the next step, all points belonging to the biggest cluster are extracted in parallel using the libraries *Filter*-method. For these points representing the current biggest cluster the native k-Means algorithm is executed with $k = 2$ in order to split it and determine two new clusters with two new centroids. By adding them into our centroid set and removing the centroid of the old biggest cluster, we managed to increase our set by one. Overall, this iteration is repeated $k - 1$ times in order to get our final k centroids representing the result of this procedure.

Algorithm 3 Thrill k-Means with threshold

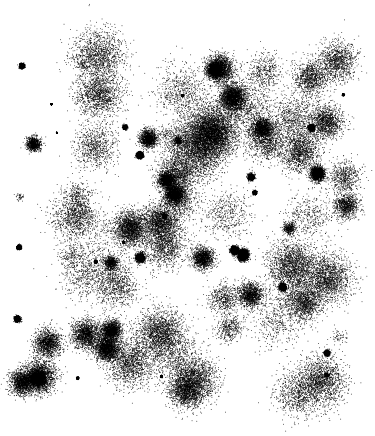
```
1:  function KMeans(points, d, k, iter, epsilon)
2:      i ← 0
3:      running ← true
4:      centroids ← select k random centroids from points
5:      while i < iter && running do
6:          oldCentroids ← centroids
7:          Calculate closest centroid for each point
8:          centroids ← determine new centroids
9:          Sort centroids
10:         running ← false
11:         for all centroids do
12:             if Distance(centroids, oldCentroids) > epsilon then
13:                 running ← true
14:         i ← i + 1
15:     return centroids
```

5 Evaluation

We evaluate our implementation on an artificial point cloud generated by us and the BIRCH3 and BIRCH2 data sets presented in [Zhang97]. All three data sets are 2-dimensional, as seen in Fig. 1, enabling a better visualization of the determined result. Further we ran the clustering algorithms on an Intel Core i5-5200U DualCore processor running at 2.7Ghz on four logical



(a) Our generated data set with $n = 10240$ data point in $c = 10$ cluster.



(b) Birch3 data set with $n = 100000$ data points in $c = 100$ cluster.

(c) Birch2 data set with $n = 100000$ data points in $c = 100$ cluster.

Figure 1: Data sets used for the evaluation of k-Means and bisecting k-Means.

processors and a Thrill configuration with four local workers. In order to compare the speed of both implementations we had to adjust the k-Means algorithm in Thrill. Originally, the native method is executed a fixed number of times predefined at program start. This in turn leads to the fact that bisecting k-Means would always be a magnitude $k-1$ worse in execution time since the native method is called $k-1$ times. Therefore, we added the possibility to set a threshold *epsilon* for an earlier termination. If the position difference of any centroid from iteration t and the same centroid in iteration $t-1$ is bigger than the threshold, an additional iteration is performed. In our tests we set the threshold equal to zero meaning that we accept the centroids of an iteration if they do not change anymore. The adapted k-Means algorithm is outlined in Algorithm 3. Note that the new centroids are determined in a distributed manner in line 8. Since the performed *AllGather*-operation for centroid exchange among the processing units does not maintain their order, each unit has to sort them locally based on their identifier. For the evaluation, we both tested the execution time of the algorithms and the average distance from a single point to its closest centroid. As for the parameters we set the number of fixed iterations to 500 with an *epsilon* threshold of 0. Further, we tested the algorithms for different k, four times for each configuration with a subsequent averaging of the results. The number of clusters were chosen in respect to the ground truth number of clusters. The results for the measured execution time can be seen in figures 2-4. For all three data sets the bisecting method was slower than the native method due to the hierarchical procedure. By using more clusters the difference in execution time worsens. However, the quality of the result represented by the average distance from a point to its nearest centroid is better in the bisecting algorithm. As figures 5-7 show, the average distance for the ground truth number of clusters is smaller in all three data sets when the bisecting method is used. It should be noted that this value is averaged for only one instance. Bearing in mind that we operate on huge data sets, the total difference of the distance would also be enormous. Especially for even structures like Birch2 the gap is huge as seen in figure 7.

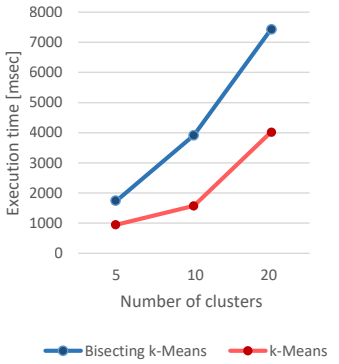


Figure 2: Execution time for our data set.

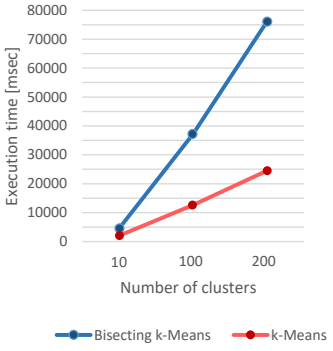


Figure 3: Execution time for the Birch3 data set.

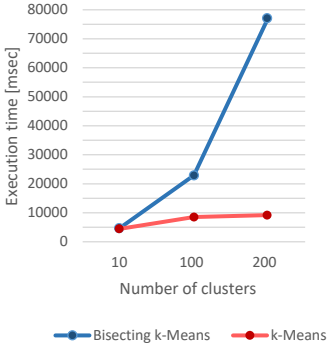


Figure 4: Execution time for the Birch2 data set.

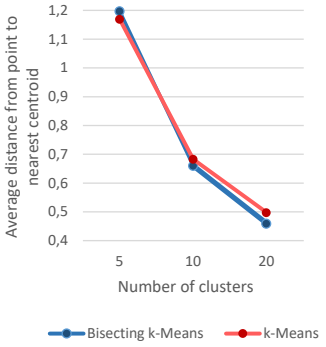


Figure 5: Average point to nearest centroid distance for our data set.

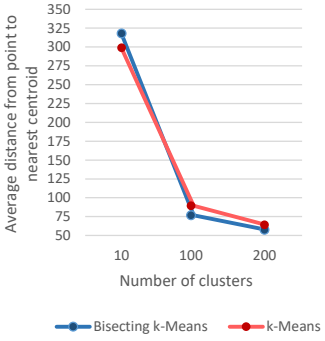


Figure 6: Average point to nearest centroid distance for the Birch3 data set.

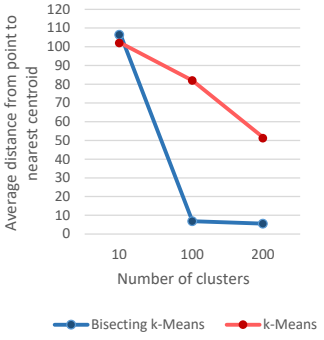


Figure 7: Average point to nearest centroid distance for the Birch2 data set.

6 Conclusion

We have implemented the bisecting k-Means method in the Thrill framework and have compared its performance concerning the execution time and the result quality to the already existing native k-Means approach. In order to get correct and comparable results we had to add a threshold termination for the existing methodology. The results show that the bisecting method is slower than the native k-Means implementation, but exceeds in result quality for cluster sizes close to the ground truth number of clusters.

References

[Thr16] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm and Peter Sanders: *Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++.*

[Stein00] Michael Steinbach, George Karypis and Vipin Kumar:*A comparison of document clustering techniques.*

[Lloyd82] Stuart Lloyd:*Least squares quantization in PCM.*

[Zhang97] Tian Zhang, Raghu Ramakrishnan and Miron Livny:*BIRCH: A new data clustering algorithm and its applications.*