Optimisation Techniques for Parallel K-Means on MapReduce

Sami Al Ghamdi(✉), Giuseppe Di Fatta, and Frederic Stahl

School of Systems Engineering, University of Reading,
Whiteknights, RG6 6AY, Reading, UK
s.a.m.alghamdi@pgr.reading.ac.uk,
{g.difatta,f.t.stahl}@reading.ac.uk

Abstract. The K-Means algorithm is one the most efficient and widely used algorithms for clustering data. However, K-Means performance tends to get slower as data grows larger in size. Moreover, the rapid increase in the size of data has motivated the scientific and industrial communities to develop novel technologies that meet the needs of storing, managing, and analysing large-scale datasets known as *Big Data*. This paper describes the implementation of parallel K-Means on the MapReduce framework, which is a distributed framework best known for its reliability in processing large-scale datasets. Moreover, a detailed analysis of the effect of distance computations on the performance of K-Means on MapReduce is introduced. Finally, two optimisation techniques are suggested to accelerate K-Means on MapReduce by reducing distance computations per iteration to achieve the same deterministic results.

Keywords: K-Means · Parallel K-Means · Clustering · Mapreduce

1 Introduction

Clustering is the process of partitioning data points in a given dataset into groups (clusters), where data points in one group are more similar than data points in other groups. cluster analysis plays an important role in the Big Data problem. For example, it has been used to analyse gene expression data, and in image segmentation to locate objects' borders in an image.

K-Means [1] is one of the most popular and widely used clustering algorithms. K-means has been extensively studied and improved to cope with the rapid and exponential increase in the size of datasets. One obvious solution is to parallelise K-Means. K-Means have been parallelised based on different environments such as Message Passing Interface (MPI) [2] and MapReduce [3].

For a given number of iterations, the computational complexity of K-Means is dominated by the distance computations required to determine the nearest centre for each data point. These operations consume most of the algorithm's run-time because, in each iteration, the distance from each data point to each centre has to be calculated. Various optimisation approaches have been introduced to tackle this issue. Elkan [4] applied the triangle inequality property to eliminate unnecessary distance computations on high dimensional datasets. An optimisation technique based on multidimensional

© Springer International Publishing Switzerland 2015
G. Di Fatta et al. (Eds.): IDCS 2015, LNCS 9258, pp. 193–200, 2015.
DOI: 10.1007/978-3-319-23237-9_17

trees (KD-Trees) [5] was proposed by Pelleg and Moore [6] to accelerate K-Means. Judd et al. [7] presented a parallel K-Means formulation for MPI and used two approaches to prune unnecessary distance calculations. Pettinger and Di Fatta [8, 9] proposed a parallel KD-Tree K-Means algorithm for MPI, which overcomes the load imbalance problem generated by KD-Trees in distributed computing systems. Different approaches have been proposed to improve K-Means efficiency on MapReduce by reducing the number of iterations. However, we intend to accelerate K-Means on MapReduce by reducing distance computations per iteration.

This paper describes the implementation of K-Means on MapReduce with a mapper-combiner-reducer approach and how the iterative procedure is accomplished on MapReduce. In Addition, it presents some preliminary results relative to the effect of distance calculations on the performance of K-Means on MapReduce. Finally, two approaches are suggested to improve the efficiency of K-Means on MapReduce.

The rest of the paper is organised as follows: Sect. 2 briefly introduces K-Means and MapReduce, and presents a detailed description of Parallel K-Means on MapReduce. Section 3 reports the experimental results. Section 4 presents the work in progress. Finally, Sect. 5 concludes the paper.

2 Parallel K-Means on MapReduce

2.1 K-Means

Given a set X of n data points in a d-dimensional space $\mathbb{R}^d$, and an integer k that represents the number of clusters, K-Means partitions X into k clusters by assigning each $x_i \in X$ to its nearest cluster centre, or centroid, $c_j \in C$, where C is the set of k centroids. Given a set of initial centroids, data points are assigned to clusters and cluster centroids are recalculated: this process is repeated until the algorithm converges or meets an early termination criterion. The goal of K-Means is to minimise the objective function known as the Sum of Squared Error $(SSE) = \sum_{j=1}^{k} \sum_{i=1}^{n_j} ||x_i - c_j||^2$, where x is the i^{th} data point in the j^{th} cluster and n_j is the number of data points in the j^{th} cluster. The time complexity for K-Means is $O(nkd)$ per iteration.

2.2 MapReduce

MapReduce [3] is a programming paradigm that is designed to, efficiently and reliably, store and process large-scale datasets on large clusters of commodity machines.

In this paradigm, the input data is partitioned and stored as blocks (or input-splits) on a distributed file system such as Google File System (GFS) [10], or Hadoop Distributed File System (HDFS) [11]. The main phases in the MapRede are *Map*, *Shuffle*, and *Reduce*. In addition, there is an optional optimisation phase called *Combine*. The MapReduce phases are explained as follows:

In the *Map* phase, the user implements a *map* function that takes as an input the records inside each input-split in the form of key1-value1 pairs. Each map function

processes one pair at a time. Once processed, a new set of intermediate key2-value2 pairs is outputted by the mapper. Next, the output is spilled to the disk of the local file system of the computing machine. In the *Shuffle* phase the mappers' output is sorted, grouped by key (key2) and shuffled to reducers. Once the mappers' outputs are transferred across the network, the *Reduce* phase proceeds where reducers receive the input as key2-list(value2) pairs. Each reducer processes the list of values associated to each unique key2. Then, each reducer produces results as key3-value3 pairs, which are written to the distributed file system. The *Combine* phase is an optional optimisation on MapReduce. Combiners minimise the amount of intermediate data transferred from mappers to reducers across the network by performing a local aggregation over the intermediate data.

2.3 Parallel K-Means on MapReduce Implementation

Parallel K-Means on MapReduce (PKMMR) has been discussed in several papers (e.g., [12, 13]). However, in this paper we explain, in details, how *counters* are used to control the iterative procedure. Moreover, we show the percentage of the average time consumed by distance computations. PKMMR with a combiner consists of: *Mapper*, *Combiner*, *Reducer* user program called *Driver* that controls the iterative process. In the following sections, a data point is denoted as *dp*, a cluster identifier as *c_id*, the combiner's partial sum and partial count as *p_sum* and *p_count*.

Driver Algorithm. The Driver is a process that controls the execution of each K-Means iterations in MapReduce and determines its convergence or other early termination criteria. The pseudocode is described in Algorithm-1. The Driver controls the iterative process through a user defined counter called *global_counter* (line 2). The global_counter is used as a termination condition in the while loop. The counter is incremented in the Reducer if the algorithm does not converge or an early termination condition is not met, otherwise, the counter is set to zero and the while loop terminates. Besides configuring, setting, and submitting the MapReduce job, the Driver also merges multiple reducers' outputs into one file that contains all updated centroids.

Algorithm-1: *Driver*

```
1:  Select k initial cluster centroids randomly;
2:  global_counter := 1 //initialised and modified in Reducer (Algorithm-4)
3:  while global_counter > 0 or a termination condition is not met do
4:      Configure and setup a MapReduce job;
5:      Send initial set of centroids to computing nodes,
6:      Run the MapReduce job;
7:      if number of reducers > 1 then
8:        Merge reducers output into one file
9:      end if
10:     global_counter := Counter(global_counter).getValue();
11: end while
```

Mapper Algorithm. Each Mapper processes an individual input-split received from HDFS. Each Mapper contains three methods, *setup*, *map* and *cleanup*. While the map method is invoked for each key-value pair in the input-split, setup and cleanup methods are executed only once in each run of the Mapper. As shown in Algorithm-2, setup loads the centroids to c_list. The map method takes as input the offset of the dp and the dp as key-value pairs, respectively. In lines 4−10, where the most expensive operation in the algorithm occurs, the loop iterates over the c_list and assigns the dp to its closest centroid. Finally, the mapper outputs the c_id and an object consists of the dp and integer 1. Because it is not guaranteed that Hadoop is going to run the Combiner, Mapper and Reducer must be implemented such that they produce the same results with and without a Combiner. For this reason, an integer 1 is sent with the dp (line 11) to represent p_count in case the combiner is not executed.

Algorithm-2: *Mapper*

```
Method    setup ( )
 1:  Load centroids to c_list;
Method    map (key, value)
 1:  Extract dp vector from value;
 2:  c_id := -1;
 3:  min_distance := ∞;
 4:  for i := 0 to c_list.size -1 do
 5:      distance := EuclideanDistance(c_list[i], dp)
 6:      if distance < min_distance then
 7:          min_distance := distance;
 8:          c_id := i;
 9:      end if
 10: end for
 11: output (c_id, (dp, 1));
```

Algorithm-3: *Combiner*

```
Method    setup ( )
 1:  Load centroids to c_list;
Method    reduce(c_id, list<values>)
 1:  p_count := 0,  p_sum := 0;
 2:  for value in values do
 3:      Extract dp vector from value;
 4:      p_sum := p_sum + the vector sum of dps in d-dimensions;
 5:      p_count := p_count + 1;
 6:  end for
 7:  output(c_id, (p_sum, p_count))
```

Combiner Algorithm. As shown in Algorithm-3, the Combiner receives from the Mapper (key, list(values)) pairs, where key is the c_id, and list(values) is the list of dps assigned to this c_id along with the integer 1. In lines 2−6, the Combiner performs local aggregation where it calculates the p_sum, and p_count of dps in the list(values) for each c_id. Next, in line 7, it outputs key-value pairs where key is the c_id, and value is an object composed of the p_sum and p_count.

Reducer Algorithm. After the execution of the Combiner, the Reducer receives (key, list(values)) pairs, where key is the c_id and each value is composed of p_sum and p_count. In lines 2−6 of Algorithm-4, instead of iterating over all the dps that belong to a certain c_id, p_sum and p_count are accumulated and stored in total_sum and total_count, respectively. Next, the new centroid is calculated and added to new_c_list. In lines 9−11, a convergence criterion is tested. If the test holds, then the global_counter is incremented by one, otherwise, the global_counter's value does not change (stays zero) and the algorithm is terminated by the Driver.

Algorithm-4: *Reducer*

Method *setup ( )*

```
1:  Load centroids to c_list; //holds current centroids
2:  global_counter = 0;
3:  Initialise new_c_list; //holds updated centroids
```

Method *reduce(c_id, list<values>)*

```
1:  total_sum, total_count,  new_centroid, old_centroid  = 0;
2:  for value in values do
3:      Extract dp vector from value;
4:      total_sum := total_sum + value.get_p_sum();
5:      total_count := total_count + value.get_p_count();
6:  end for
7:  new_centroid :=  total_sum /  total_count;
8:  add new_centroid to new_c_list
9:  if new_centroid has changed  or a threshold is not reached then
10:     Increment global_counter by 1
11: end if
12: output(c_id, dp)
```

Method *cleanup( )*

```
1:  Write new centroids in new_c_list to HDFS;
```

3 Experimental Results

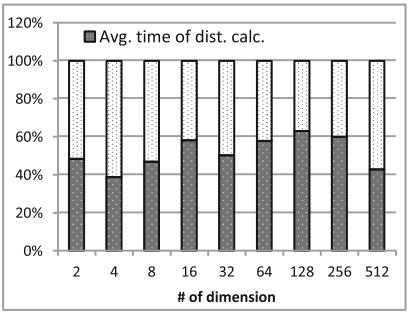
To evaluate PKMMR, we run the algorithm on a Hadoop [14] 2.2.0 cluster of 1 master node and 16 worker nodes. The master node has 2 AMD CPUs running at 3.1 GHz with 8 cores each, and 8 × 8 GB DDR3 RAM, and 6 × 3 TB Near Line SAS disks

running at 7200 rpm. Each worker node has 1 Intel CPU running at 3.1 GHz with 4 cores, and 4 × 4 GB DDR3 RAM, and a 1 × 1 TB SATA disk running at 7200 rpm.

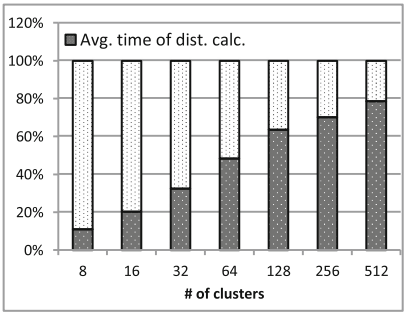
The datasets used in the experiments are artificially generated where data points are randomly distributed. Additionally, initial cluster centroids are randomly picked from the dataset [1]. The number of iterations is fixed in all experiments at 10.

To show the effect of distance calculations on the performance of PKMMR, we run the algorithm with different number of data points n, dimensions d and clusters k. The percentage of the average time consumed by distance calculations in each iteration is represented by the grey area in each bar in the Figs. 1-(a), (b), and (c). The white dotted area represents the percentage of the average time consumed by other MapReduce operations per iteration including job configuration and distribution, map tasks (excluding distance calculations) and reduce tasks.

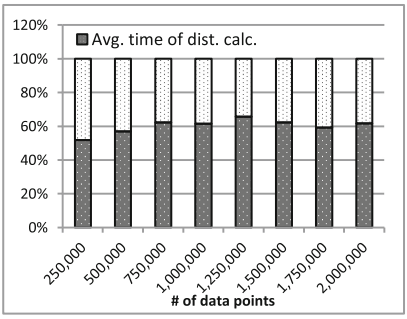
In each run, we compute the average run-time for one iteration by dividing the total run-time over the number of iterations. Then, the average run-time consumed by distance calculations per iteration is computed.



(a) Avg. time consumption with variable number of d. n=1000000, k=128.



(b) Avg. time consumption with variable number of k. n=1000000, d=128.



(c) Avg. time consumption with variable number of n. d=128, k=128.

Fig. 1. Percentage of the average consumed time by distance calculations per iteration with variable number of d, k and n.

We run PKMMR with a varied number of d, while n is fixed at 1,000,000, and k is fixed at 128. Figure 1-(a) shows that 39 % (d = 4) to 63 % (d = 128) of the average iteration time is consumed by distance calculations.

PKMMR is also run with a variable number of k, while n is set to 1,000,000 and d is set to 128. In Fig. 1-(b), it can be clearly seen the tremendous increase in the percentage of consumed time by distance calculations per iteration from 11 % (k = 8) to 79 % (k = 512). In this experiment, distance calculations become a performance bottleneck as the number of clusters increases, which is more likely to occur while processing large-scale datasets.

Figure 1-(c) illustrates the percentage of the average time of distance calculations when running PKMMR with variable number of n, while d = 128 and k = 128. As it can be observed, distance calculations consume most of the iteration time. About 65 % of the iteration time is spent on distance calculations when n = 1,250,000. Therefore, reducing the number of required distance calculations will most likely accelerates the iteration run-time and, consequently, improves the overall run-time of PKMMR.

4 Work in Progress

We intend to accelerate the performance of K-Means on MapReduce by applying two methods to reduce the distance computations in each iteration. Firstly, triangle inequality optimisation techniques are going to be implemented and tested with high dimensional datasets. However, such techniques usually require extra information to be stored and transferred from one iteration to the next. As a consequence, large I/O and communication overheads may hinder the effectiveness of this approach if not taken into careful consideration. Secondly, efficient data structures, such as KD-trees or other space-partitioning data structures [15], will be adapted to MapReduce and used with K-Means. Two issues will be investigated in this approach. First, inefficient performance with high dimensional datasets that has been reported in [6]. Second, load imbalance that was addressed in [8, 9].

5 Conclusions

In this paper we have described the implementation of parallel K-Means on the MapReduce framework. Additionally, a detailed explanation of the steps to control the iterative procedure in MapReduce has been presented. Moreover, a detailed analysis of the average time consumed by distance calculations per iteration has been discussed. From the preliminary results, it can be clearly seen that most of the iteration time is consumed by distance calculations. Hence, reducing this time might contribute in accelerating K-Means on the MapReduce framework. Two approaches are under investigations, which are, respectively, based on the triangle inequality property and space-partitioning data structures.

References

1. Lloyd, S.: Least Squares Quantization in PCM. IEEE Trans. Inf. Theor. **28**(2), 129–137 (1982)
2. Dhillon, I.S., Modha, D.S.: A data-clustering algorithm on distributed memory multiprocessors. In: Zaki, M.J., Ho, C.-T. (eds.) KDD 1999. LNCS (LNAI), vol. 1759, pp. 245–260. Springer, Heidelberg (2000)
3. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, **6**, p. 10. Berkeley, CA, USA (2004)
4. Elkan, C.: Using the triangle inequality to accelerate k-means. In: presented at the International Conference on Machine Learning - ICML, pp. 147–153 (2003)
5. Bentley, J.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
6. Pelleg, D., Moore, A.: Accelerating exact K-means algorithms with geometric reasoning. In: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 277−281, New York, NY, USA (1999)
7. Judd, D., Mckinley, P.K., Jain, A.K.: Large-scale parallel data clustering. IEEE Trans. Pattern Anal. Mach. Intell. **20**, 871–876 (1998)
8. Pettinger, D., Di Fatta, G.: Scalability of efficient parallel K-means. In: 2009 5th IEEE International Conference on E-Science Workshops, pp. 96–101 (2009)
9. Di Fatta, G., Pettinger, D.: Dynamic load balancing in parallel KD-tree K-means. In: IEEE International Conference on Scalable Computing and Communications, pp. 2478–2485 (2010)
10. Ghemawat, S., Gobioff, H., Leung, S.-T.: The google file system. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 29–43. New York, NY, USA (2003)
11. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10. Washington, DC, USA (2010)
12. Zhao, W., Ma, H., He, Q.: Parallel *K*-means clustering based on mapreduce. In: Jaatun, M. G., Zhao, G., Rong, C. (eds.) Cloud Computing. LNCS, vol. 5931, pp. 674–679. Springer, Heidelberg (2009)
13. White, B., Yeh, T., Lin, J., Davis, L.: Web-scale computer vision using mapreduce for multimedia data mining. In: Proceedings of the Tenth International Workshop on Multimedia Data Mining, pp. 9:1–9:10. New York, NY, USA (2010)
14. Apache Hadoop. http://hadoop.apache.org/. Accessed on 03 January 2015
15. Pettinger, D., Di Fatta, G.: Space partitioning for scalable K-means. In: IEEE The Ninth International Conference on Machine Learning and Applications (ICMLA 2010), pp. 319-324. Washington DC, USA, 12–14 December 2010