Future Generation Computer Systems 102 (2020) 84–94

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Fast and effective Big Data exploration by clustering

Michele Ianni [a], Elio Masciari [b,*], Giuseppe M. Mazzeo [c], Mario Mezzanzanica [d], Carlo Zaniolo [e]

[a] DIMES, University of Calabria, Rende, Italy
[b] DIETI, Federico II University, Naples, Italy
[c] Facebook, Menlo Park, USA
[d] DISMEQ, Milano Bicocca University, Milano, Italy
[e] Computer Science, UCLA, Los Angeles, USA

ARTICLE INFO

Article history:
Received 9 February 2019
Received in revised form 1 July 2019
Accepted 31 July 2019
Available online 6 August 2019

Keywords:
Big Data
Clustering
Data exploration

ABSTRACT

The rise of Big Data era calls for more efficient and effective Data Exploration and analysis tools. In this respect, the need to support advanced analytics on Big Data is driving data scientist' interest toward massively parallel distributed systems and software platforms, such as Map-Reduce and Spark, that make possible their scalable utilization. However, when complex data mining algorithms are required, their fully scalable deployment on such platforms faces a number of technical challenges that grow with the complexity of the algorithms involved. Thus algorithms, that were originally designed for a sequential nature, must often be redesigned in order to effectively use the distributed computational resources. In this paper, we explore these problems, and then propose a solution which has proven to be very effective on the complex hierarchical clustering algorithm CLUBS+. By using four stages of successive refinements, CLUBS+ delivers high-quality clusters of data grouped around their centroids, working in a totally unsupervised fashion. Experimental results confirm the accuracy and scalability of CLUBS+ on platforms tailored for Big Data management.

1. Introduction

The current era of *Big Data* [1–3] has forced both researchers and industries to rethink the computational solutions for getting useful insight on massive data produced in a wide variety of real life scenarios. In fact, a great deal of attention has been devoted to the design of new algorithms for analyzing information available from Twitter, Google, Facebook, and Wikipedia, just to cite a few of the main big data producers. Although this massive volume of data can be quite useful for people and companies, it makes analytic and retrieval operations really time consuming due to their high computational cost. A possible solution relies upon the possibility to cluster big data in a compact but still informative version of the entire dataset. Obviously, such clustering techniques should produce clusters (or summaries) that are meaningful and as much compact as possible. Clustering algorithms could be beneficial in several application scenarios such as cybersecurity, user profiling and recommendation systems, to cite a few.

Although all textbooks describe clustering as the quintessential "unsupervised learning task" all the major algorithms proposed so far require significant guidance from the user, such as the number of clusters to be derived for *k*-means++ and the termination condition for hierarchical clustering. In [4] we proposed CLUBS$^+$, a clustering algorithm that manages to be very effective without any user guidance by (i) operating in multiple stages whereby the techniques used at one stage compensate for the shortcomings of the previous stages, and by (ii) applying carefully selected and well-tuned criteria to assure the quality of the results produced at each stage. In particular, the benefit of (i), is illustrated by the fact that while over-splitting represents a very difficult issue for most divisive algorithms, it is much less of concern here, because a certain degree of over-splitting can be tolerated since it will be undone in the successive stages that include an agglomerative step. Combining the divisive and agglomerative phases is an idea taken from its predecessor CLUBS [5] which has demonstrated its usefulness in several applications [6]. CLUBS$^+$ provides major improvements with respect to its predecessor CLUBS [5], including the addition of the two refinement steps, that improve (i) the detection of outliers, which is critical in distinguishing regions of high-density noise from low-density clusters, and (ii) the creation of well-rounded ellipsoidal clusters around the centroids.

* Corresponding author.
E-mail addresses: mianni@dimes.unical.it (M. Ianni), elio.masciari@unina.it (E. Masciari), mazzeo@cs.ucla.edu (G.M. Mazzeo), mario.mezzanzanica@unimib.it (M. Mezzanzanica), zaniolo@cs.ucla.edu (C. Zaniolo).

https://doi.org/10.1016/j.future.2019.07.077
0167-739X/

In this paper we present a parallel version of CLUBS$^+$, which we call CLUBS-P. Like CLUBS$^+$, CLUBS-P, scales linearly w.r.t. the size of the dataset. Furthermore, experiments show that the speed-up increases almost linearly as more machines for running the algorithm are available. These two features make CLUBS-P a very effective algorithm for clustering large amount of data. Finally, a nice feature of CLUBS-P is that once the dataset to be clustered has been distributed across the available machines, there is no need to move data, as CLUBS-P requires the machines to exchange only summary information.

We provided two different implementation of our parallel clustering algorithm. The first one is based on an ad-hoc approach, based on message passing while the second one is based on the widely used Big Data framework Apache Spark [7,8]. The goal of this dual implementation is to evaluate the opportunity of using Big Data frameworks against ad hoc implementation based on the data features. The rationale for this kind of analysis is explained in what follows. The implementation of a distributed algorithm calls for suitable solution to some crucial problem in a distributed programming environment such as: load balancing and fault tolerance. A very popular solution proposed in recent years for implementing parallel algorithms tailored for Big Data is Apache Spark.[1] Indeed, it offers very useful functions which relieves the programmers from the explicit management of process assignment and memory management. In this respect, we implemented our CLUBS-P algorithm based on Spark. On the other side, the use of general framework like Spark, may limit the control over data manipulation, thus limiting the possibility to push on applying very specific optimizations for the problem at hand. To this end we also implemented a message passing based version of the algorithm for the purpose of comparing to which extent a solution could overcome the other and the experimental settings that would suggest the use of a solution against the other.

Plan of the paper. In Section 2 we briefly discuss some related work on clustering in order to focus on main features of this class of algorithms. In Section 3 we provide an overview of CLUBS$^+$, while in Section 4 we first discuss the crucial points of CLUBS$^+$ with a view to its parallelization, that lead to devising CLUBS-P, for which we present two implementations: one based on the message passing paradigm (Section 5), and another based on Apache Spark (Section 6). In Section 7 we present the experimental results and, finally, in Section 8 we draw our conclusions.

2. Related work

The advent of Big Data has rekindled the interest of researchers in the data mining fundamental tools, such as clustering. Clustering algorithms with super-linear computational complexity, in fact, are not suitable in the context of Big Data. Several approaches have been proposed for overcoming the complexity of clustering techniques, both for the single- and the multiple-machines scenario [9].

The approaches for making possible clustering of large datasets on single-machines are typically based on sampling and dimensionality reduction techniques. CLARA [10] and CLARANS [11], for instance, largely improves PAM [12] performances using sampling techniques to reduce the solution search space. In [13] a method using Random Projection has been proposed to reduce the dimensionality of data while trying to maintain the pairwise distance between points. Instead, in [14] a method using global projection has been proposed, with the objective of maintaining each projected point as close as possible to the original one. As regards density based approaches, in [15], the authors present NG-DBSCAN, an approximate density-based clustering algorithm that operates on arbitrary data and any symmetric distance measure. Moreover, in [16] a parallel DBSCAN algorithm is proposed that combines the disjoint-set data structure and Parameter Server framework in order to minimize communication cost.

Although the above methods can significantly reduce the computational times of clustering on large datasets, the growth of data size has been steeper than the growth of the amount of data that single machines can handle in a given amount of time using the above techniques. This make necessary the use of parallel approaches, that pose serious challenges, due to some crucial aspects, like coordination, communication, sharing of resources and fault-tolerance capability. Examples of clustering algorithms with a parallel implementation are DBDC [17], ParMETIS [18], and DBSCAN based on message passing [19].

Recently, a number of classical algorithms have been adapted to the popular MapReduce paradigm, including PK-Means [20] and MR-DBSCAN [21]. In [22] an automatic method is proposed that optimizes the disk accesses and network communication, while providing a pluggable general framework for clustering with MapReduce.

A very successful variation of k-means has been proposed in [23], where k-means‖, a parallel version of k-means++ has been shown to outperform k-means++ also in the single-machine scenario, thanks to the very efficient seeding algorithm. The implementation of this algorithm is available for Spark,[2] for this reason, and since k-means is the most popular around-centroid algorithm, it is the best candidate for a comparative analysis with our implementations of CLUBS-P.

3. An overview of CLUBS$^+$

In [4] we recently introduced CLUBS$^+$. It is a parameter-free around-centroid clustering algorithm based on a fast hierarchical approach, that combines the benefits of the divisive and agglomerative approaches. The first operation performed by CLUBS$^+$ is the definition of a binary space partition of the domain of the dataset D, which yields set of coarse clusters that are then refined. The next operation is an agglomerative procedure performed over the previously refined clusters, which is followed by a final refinement phase. In the refinement phases, outliers are identified and the remaining points are assigned to the nearest cluster. In our running example of Fig. 1, we show how the successive steps of CLUBS$^+$ applied to the two-dimensional data distribution of Fig. 1(a), produce the results shown in Fig. 1(b–e).

In order to make this paper-self contained, in the following we provide some details about the four steps of CLUBS$^+$ allowing a full comprehension of the parallel algorithm implemented in this paper, while a more exhaustive description of the algorithm phases can be found in [4].

3.1. The divisive step

The *divisive step* of CLUBS$^+$ performs a top-down binary partitioning of the dataset to isolate hyper-rectangular blocks whose points are as much as possible close to each other: this is equivalent to minimizing the *within clusters sum of squares* (*WCSS*) of the blocks, an objective also pursued by k-means and many other algorithms.

Since finding the partitioning which minimizes a measure such as the *WCSS* is NP-hard even for two-dimensional points [24], CLUBS$^+$ uses a greedy approach where the splitting hyperplanes are orthogonal to the dimension axes, and the blocks

[1] http://spark.apache.org/.

[2] http://spark.apache.org/docs/latest/mllib-clustering.html.

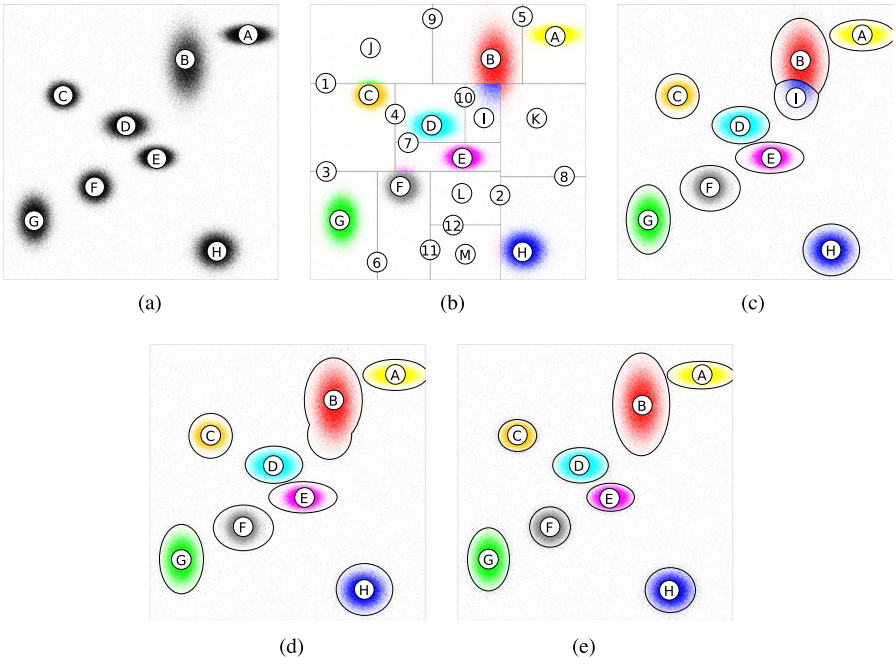

Fig. 1. A two-dimensional dataset with 8 clusters (a), its partitioning after the divisive step (b), the intermediate refinement (c), the agglomerative step (d), and the final refinement (e).

are recursively split into pairs of clusters that optimize specific clustering criteria that will be discussed later.

Given a input dataset, the algorithm begins with a single cluster S corresponding to the whole dataset. S is entered into a priority queue, Q. Q contains blocks of the partition that is iteratively built over the dataset. While Q is not empty, a block B is removed and split into a pair of blocks. If the split is effective, the pairs of blocks replace B in Q, otherwise B becomes a 'final' block for this phase.

In order to efficiently find the best split for a block, the marginal distributions of the block must be computed. In particular, for each dimension i of a d-dimensional block B we must compute the functions $C_B^i : \mathbb{R} \to \mathbb{N}$ and $LS_B^i : \mathbb{R} \to \mathbb{R}^d$, defined as follows

$$C_B^i(x) = |p \in B \wedge p[i] = x| \tag{1}$$

$$LS_B^i(x) = \sum_{p \in B \wedge p[i]=x} p \tag{2}$$

where we sum up all the values of the points p belonging to the block B having the value of the ith coordinate equal to x. These functions can be represented as maps, or, assuming that the coordinates of the points are integers, as arrays. In [4] we showed that the split that minimizes the *WCSS* can be found through a linear scan of these maps/arrays.

We conducted an extensive evaluation of the effectiveness of criteria proposed in the literature for estimating the quality and naturalness of a cluster set [25], and we concluded that the Calinski–Harabasz index, *CH-index* for short, is the most suitable to our needs [26]. Briefly, after each step, we compute the new *CH-index*, and if it is increased by the split, then we consider the split effective and continue the divisive phase. Otherwise, we check a "local" criterion, based on the presence of a 'valley' in the marginal distribution. In fact, even though a split could not decrease the overall *CH-index*, a very large local discontinuity could justify the split of a block anyway.

3.2. The intermediate refinement

When the divisive phase is completed, the overall space is partitioned into (a) blocks containing clusters, and (b) blocks that only contain noise points. In this *intermediate refinement* phase CLUBS$^+$ seeks to achieve the objectives of (I) separating the blocks that contain clusters from those that do not, and (II) generating well-rounded clusters for the blocks in the previous group.

Task (I) is performed by using the observation that (a) the density of blocks containing only noise is low, and (b) the density of such blocks is rather uniform because of the random nature of noise.

Thus, CLUBS$^+$ checks condition (a) first by sorting the densities of blocks and detecting jumps in density between contiguous blocks. The first jump in the sequence of blocks, sorted by increasing density, determines the subset of noise-block candidates. Then, we test the candidates using condition (b), i.e., a small hypercube is taken around the centroid of each block being checked for condition (b), and if the density in the hypercube is significantly larger than the density of the whole block, this is classified as an outlier block; otherwise it is classified as a cluster block. For instance, these tests applied to our running example in Fig. 1(b) have classified blocks "J", "K", "L" and "M" as outlier-blocks.

Once, the cluster-blocks have been found, we can now proceed with task (II). To this end, we measures the distance of each points $\mathbf{p} = (p_1, \ldots, p_d)$ from each cluster-blocks C with centroid $(c_1, \ldots, c_d)$ as

$$dist^\star(\mathbf{p}, C) = \sum_{i=1}^{d} \left(\frac{\|p_i - c_i\|}{r_i} \right)^2 \tag{3}$$

where r_i is the radius of the cluster along the ith dimension, estimated using the usual criterion based on the maximum $3\times\sigma$ distance for separating core points from outliers, in particular: $r_i = 3 \cdot \sqrt{\frac{WCSS_i(C)}{n}}$, where n is the number of points inside C and $WCSS_i(C)$ is the variance of the ith coordinates of the points in C. Observe that $dist^\star(\mathbf{p}, C) = 1$ defines an ellipsoid whose center is the centroid of C and whose radius on the ith dimension is r_i.

This step produces well-rounded clusters that restore the natural clusters that might have been shaved-off by rigid partitioning scheme leveraged in the divisive step. For instance, the block of data in Fig. 1(b), which contains the points of the cluster "E" in Fig. 1(a), contains also some points of the original cluster "F" which is now reassembled about its centroid.

3.3. The agglomerative phase

In the *agglomerative step*, the clusters produced by the previous step are merged whenever this improves the overall clustering quality. The process of merging is symmetric to that of splitting: at each step we find the pair of clusters whose merge produces the least increase of *WCSS* and, if *CH-index* resulting from their merge increases, the merge is actually performed by replacing the two clusters by their union. If we determine that merging these two clusters will decrease the *CH-index*, the merge is not performed and the whole agglomerative phase ends, since no merging of other cluster pair can improve *CH-index*.

We point out that, we do not impose any other constraint on the choice of pairs of clusters to be merged. This is unlike the divisive step, where clusters are bound to be hyper-rectangles and each split yields two hyper-rectangles whose union is the hyper-rectangle corresponding to the split node: in the agglomerative step there is no constraint on the shape that a cluster resulting from a merge can have. In practice, however, the only clusters considered for merging are the contiguous ones, since their merge yields a smaller increase of the *WCSS* (i.e., a larger increase of the *CHindex*). Fig. 1(d) shows that the cluster "B" is obtained as the merge of the clusters "B" and "I" of Fig. 1(c).

The agglomerative step is a very fast step, as does not require to access data, since it is based only on previously computed summary information.

3.4. The final refinement

The *agglomerative step* often produces clusters of slightly irregular shapes due to the approximate and greedy criteria used in the previous steps. For instance, Fig. 1(d) shows that cluster "B" has been obtained by merging clusters "B" and "I" of Fig. 1(c). The evident asymmetry is due to the fact that the block "K" of Fig. 1(b) has been considered an outlier-block during the intermediate refinement, and the points at its upper-left corner have been asymmetrically adsorbed (due to their different position and radius) by blocks "B" and "I".

The final refinement step improves the quality of clusters and also identifies the final outlier points.

The algorithm used to perform final refinement is basically the same as the one used for task (II) of the intermediate refinement (see [4] for further details).

We can now state the complexity of CLUBS as follows:

Proposition 3.1. *Algorithm* CLUBS *works in time* $\mathcal{O}(n \times k)$ *where n is the number of points and k is the number of clusters automatically detected after the divisive step.*

Proof. The complexity of CLUBS is determined by the steps with the largest complexity, i.e. the refinement steps as the other steps work on smaller portion of the data. Indeed, the initialization of the divisive step requires a scan of the whole dataset and performs operations with constant complexity for each point,[3] therefore its complexity is $\mathcal{O}(n)$.

Assuming that k clusters are produced, their generation requires $2 \times k - 1$ iterations.[4] The procedure which computes the best splits requires a linear scan of all the data in the cluster, thus its complexity is $\mathcal{O}(n)$. Also the procedure which evaluates the effectiveness of the split has $\mathcal{O}(n)$ complexity (the number of iterations needed to evaluate the effectiveness of the split is equal to the number of distinct coordinates, which is usually much smaller than n).

Since in practical cases $k \ll n$, the dominant operation of each iteration is the computation of the best split. Thus, we can say that the complexity of the divisive step is $\mathcal{O}(n \times k)$. The computation of the cumulative marginal distribution can be performed through a linear scan of the data inside the cluster. Furthermore, after choosing the best split, it is necessary to compute the cumulative marginal distribution for just one of the two blocks obtained from the split,[5] since the cumulative marginal distributions for the other block can be obtained by difference.

The size of the cumulative marginal distribution never exceeds the number of points and it is often smaller than the size of the dataset, since many points have the same coordinate in a given dimension.[6] □

4. CLUBS-P: the parallelization of CLUBS$^+$

We now analyze the steps of CLUBS$^+$ with the aim of pointing out the issues to be tackled in order to parallelize its execution.

We assume that n workers (i.e., different machines) with equivalent hardware features are available. The workers are coordinated by a master machine. The dataset is distributed over the worker nodes that store locally a portion of the dataset, which can be, for instance stored, in a distributed file system, or even previously split over the local file systems of the machines.

It is quite intuitive that the critical operations that can benefit from the availability of multiple workers are those requiring to access the original data: (1) computation of the marginal distributions during the divisive phase; (2) computation of the local density of outlier blocks candidates during the intermediate refinement; (3) assignment of points to the closest centroid during the refinement phases.

The parallelization of the operations above is discussed in the following. The other operations, which involves computations on summary information, instead, can be handled by the master, using the same approach as in the original version of CLUBS$^+$.

4.1. Parallel computation of the marginal distributions

As remarked in previous section, the computation of the marginal distributions, more specifically of the vectors C and LS, is the only operation that requires to access the data during the divisive step. This operation can be efficiently performed in parallel thanks to the associativity and commutativity of the sum and count operations, which are the only mathematical operations involved in the computation. Therefore, given a block partition of the data B, $\{B_1, \ldots, B_k\}$, those vectors can be independently computed on each B_i (thus in parallel) and the vector for the whole block B can be obtained by simply summing the vectors obtained on all the B_i. According to the Map-Reduce paradigm, each block B_i is mapped to a set of pairs of vectors, representing its C and LS for each dimension, and these vectors are subsequently reduced by summing vectors of the same dimension.

[3] Actually, for each point computations with complexity linear in the number of dimensions are performed, but we are interested in the complexity w.r.t. the size of data.

[4] k clusters are generated by $k - 1$ splitting steps that replace a cluster by a pair; then, the failure of splitting conditions is tested for each of these k clusters.

[5] Clearly, it is better to compute the marginal distribution for the block containing less points.

[6] This may not be the case for continuous values. However, a quantization can be easily performed without compromising the quality of the results.

4.2. Parallel computation of the local density of the blocks

The local density of blocks has to be computed in order to evaluate the criterion used for Task II of the intermediate refinement. After the master has the list of the blocks resulting from the divisive phase, it also has their ranges and their global density. In order to check if the local density around the centroid of each block is significantly higher than the global density, it computes the boundaries of a small range around the centroid (i.e., a range of size 1/10 of the whole block range), and then coordinates the parallel computation of the count of the points falling in each of these 'restricted' ranges. Even in this case, the only mathematical operation involved in the computation is sum, thus the operation is perfectly parallelizable. In fact, for each portion of data of a block, it is possible to independently compute the count of points laying in the restricted range, and then the overall count is simply given by the sum of the partial counts. This count allows the master to compute the local density, and thus apply the criterion described above.

4.3. Parallel assignment of the points to the clusters

The assignment of the points to the clusters, or their classification as outliers, is an operation performed in both the refinement steps. Each worker can perform this operation independently on the others. The master, which has the summary information about the clusters, needs to send to every worker the information needed to compute the distance between points and clusters. After the computation, the same information must be updated by the master, since the cluster assignment of some points is likely to change. To this end, each node can compute the summary information for its data, and then, again, the master can simply obtain the global values of the new centroids, number of points per cluster, and cluster radius, after summing up all the values obtained from the workers.

We presented the general ideas on which our 'abstract' algorithm CLUBS-P is based for the parallelization of CLUBS. In the following, we propose two concrete implementations of this algorithm, one using an ad-hoc approach, based on message passing, and one using the Apache Spark. The two implementations are available as open source software.[7]

5. An ad-hoc implementation of CLUBS-P based on message passing: CLUBS-MP

We implemented an ad-hoc version of CLUBS-P based on message passing. The communication between the nodes of the networks (master and workers) is based on the exchange of messages that are sent using sockets (specifically, our implementation is based on Java 8, and uses standard Java sockets).

The algorithm is implemented as follows. The clustering starts when the master sends a `LoadDataSetRequest` to the workers, specifying an identifier for the dataset to load. Each worker load data into main memory, and replies to the master with a `LoadDataSetResponse`, specifying the range containing the data loaded (i.e., while loading data, each node computes the minimum and maximum value for each coordinate). When the master has received a response from all the workers, it can compute the range of the global dataset (local datasets could span different subsets of the global range). Then the master sends an `InitRootRequest` to every worker, specifying the global domain. Each worker computes the marginal distributions of its local data and replies to the master with an `InitRootResponse`, which specifies C and LS for its local data. The master can now compute C and LS for the global dataset, and thus the $WCSS$, and it can initialize the queue with the blocks to be split, by enqueuing the block corresponding to the whole dataset. The local marginal distributions of blocks enqueued by the masters have been already computed by workers. This is true for the root, as described above, and it will be true for the subsequently enqueued blocks, as it will be clear in the following. While the queue is not empty, a block is dequeued, and the computation of the global marginal distributions is started. The global marginals are computed by summing all the local marginals in pairs. Initially, we can imagine that the global marginals are 'spread' among the t available workers. The master chooses half of the t workers, and each of chosen workers sends its marginals to another worker, that sums them to its marginals. Now, the global marginals are spread among $\lceil t/2 \rceil$ workers. The master, again, chooses half of the $\lceil t/2 \rceil$ workers, and each chosen worker sends its marginals to another worker. The operation is repeated until the whole marginal are stored at one node only. Fig. 2 shows an example of distributed computation of a marginal distribution, given 4 workers w_1, w_2, w_3, and w_4. Each worker w_i computes its local marginal distribution m_i (a). Then w_2 sends m_2 to w_1 and w_4 sends m_4 to w_3 (b). w_1 and w_3 sum the received marginals to their own marginals, thus the overall marginals are now distributed over w_1 and w_3. Now, w_3 sends $m_3 + m_4$ to w_1 (c), which again sums the received marginals to its own current marginal, thus obtaining the overall marginal distribution (d).

It is easy to see that this operation requires $t-1$ total marginal sending, and the worker with the heaviest workload will be required to sum the received marginals to its marginal $\lceil \log_2 t \rceil$ times. On the other hand, the workload of the nodes that send their marginals at the first step is very low.

Therefore, in order to balance the workload, the computation is performed separately for the marginals of each dimension, with the objective of assigning a bigger workload for some dimension to the nodes that were assigned a smaller workload for other dimensions.

We formalize this technique as follows. Let $(w_1, w_2, \ldots, w_t)$ be the list of the available workers. For the ith dimension we left-rotate the initial of workers by $i-1$ positions. Thus, for the ith dimension, the list of workers is $l_i = (w_i, w_{i+1}, \ldots, w_t, w_1, \ldots, w_{i-1})$.

Let l_i^e and l_i^o be the sub-lists of elements of l_i located at even and, respectively, odd positions. The master sends a `SendMarginalsRequest` to every worker w_s whose identifier is in l_i^e, specifying for each message the worker w_r that is located in l_i^o at the same position of w_s in l_i^o, and the index of the dimension. When the node w_s receives this messages, sends the marginals of the specified dimension to the specified node w_r, through a `ReceiveMarginalsRequest`. When w_r receives the marginals, sums them to its marginals, and sends a `ReceiveMarginalsResponse` to the master. When the master receives the `ReceiveMarginalsResponse` from all the nodes that where supposed to receive marginals from another node, it substitutes l_i with l_i^o, and repeats the process. The process ends when l_i consists of one only worker, namely, w_i.

Then, a `ComputeBestSplitRequest` is sent by master to w_i, which can independently compute the best split on the ith dimension by scanning the marginals computed on this dimension. The worker w_i replies with a `ComputeBestSplitResponse`, containing the best split position and reduction of $WCSS$ on the ith dimension.

When a `ComputeBestSplitResponse` is received by the master from a number of workers equal to the number of dimensions, it can choose the overall best split. Now, three situations are possible: (1) If the split is effective, i.e., it increases the CH-$index$, a `SplitRequest` is sent to every worker, containing the

[7] https://github.com/gmmazzeo/clubsmp and https://github.com/gmmazzeo/clubspark.

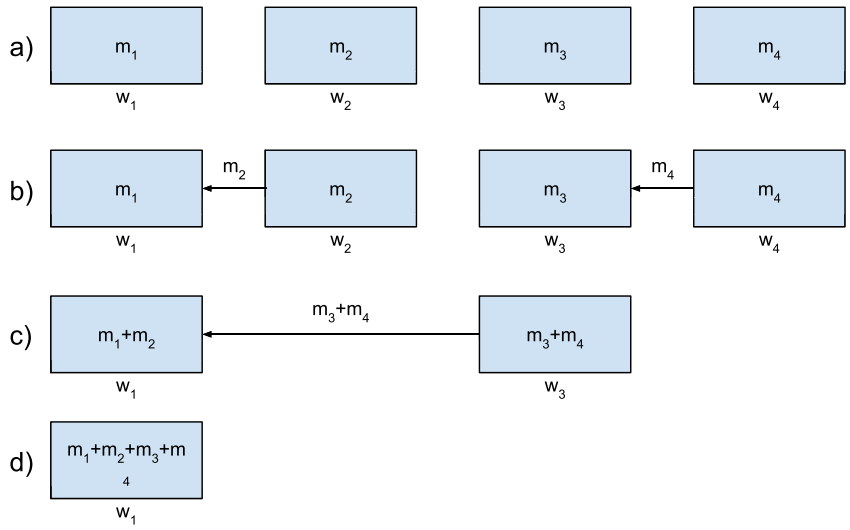

Fig. 2. Distributed computation of marginal distributions.

split dimension and position; (2) If the split would drop the *CH-index* under the 70% of the best *CH-index* currently found, the block is added to a list of final blocks (leaves of the binary partition tree that will not be split). Then, if the queue is empty, the intermediate refinement is started (details are provided in the following), otherwise a node is dequeued and the global marginal computation is started on this node, as explained before; (3) Finally, if the *CH-index* is not improved by the best split, but neither dropped significantly, the valley criterion is evaluated. Specifically, the master sends a `ComputeValleyCriterionRequest` to the worker w_i, specifying the dimension i. The worker computes the valley criterion and replies to the master with a `ComputeValleyCriterionResponse`, which specifies if the criterion is satisfied. When the criterion is satisfied on any dimension, the master sends a `SplitRequest` to every worker. If the criterion is not satisfied on any dimension, the master dequeues a block and starts the computation of its marginals, if the queue is not empty, otherwise starts the intermediate refinement step.

When a worker receives a `SplitRequest`, it creates two new blocks, and partitions local data among the two new blocks. Then, it computes the marginals and sends to the master a `SplitResponse`, which contains C and LS of the two new blocks. When the master receives the `SplitResponse`, it can sum the information received from workers, compute the *WCSS* of the two blocks and enqueue them.[8] Then a block is dequeued and the computation of the global marginal distribution is started.

As already stated, when the queue is empty, the intermediate refinement is started. Observe that the master now has a list with the blocks of the partition obtained through the divisive step, and for every block summary information is available. In particular, for each block the range and the number of contained points is known, therefore, the density can be computed. This enables to detect the outlier-block candidates, as described in previous section. For these blocks, the restricted density must be computed. To this end, a `RestrictedCountRequest`, specifying the identifiers of the outlier-block candidates and the corresponding restricted range, is sent to every worker. The workers scan their local data in the specified blocks and count the number of points contained in each restricted range. The result is sent to the master through a `RestrictedCountResponse`.

When the `RestrictedCountResponse` is received by the master from every worker, it is possible to decide which blocks are outlier-blocks. Then the master sends to the workers a `IntermediateRefinementRequest`, which specifies the centroids and the radii of the cluster-blocks. Then, each worker scans all data and assigns each of its points to the nearest non-noise block or labels it as an outlier. In the meanwhile C and LS are updated for each cluster. Finally, this information is sent to the master through an `IntermediateRefinementResponse`.

When the master receives from all the workers the `IntermediateRefinementResponse`, it can compute the global C and LS. This information enables to perform the agglomerative phase. The result of the agglomerative phase is a new set of clusters, with their C and LS, and thus centroids and radii, that are sent to every worker with the message `FinalRefinementRequest`, for the final refinement. Each worker performs the final refinement, like for the intermediate refinement case, and sends the updated C and LS for each cluster through a `FinalRefinementResponse`. When the master receives this message from all the workers, the clustering is completed.

6. CLUBS-P meets Spark: CLUBS★

The implementation of a distributed algorithm requires a lot of effort for solving specific issues related to the problem, but also general issues for distributed computing, such as load balancing and fault tolerance. Nowadays, a very popular framework for implementing parallel algorithms tailored for Big Data is Apache Spark.[9] As a matter of fact, it offers very useful functions which relieve the programmers from the burden of handling the general issues of distributed computing.

For sake of brevity, we will mention here only the Spark's features that are relevant to our scope. The main data structure used by Spark is *Resilient Distributed Dataset (RDD)*. An RDD is a read-only, partitioned collection of records. Spark's API allows the definition of custom functions in order to manipulate RDD when implementing our algorithm.

Two of the core Spark's functions, working on an RDD according to the map-reduce paradigm, are the following: (a) *mapPartitions*: RDD returned by mapPartitions is obtained by transforming each partition of the source RDD as specified by a user-defined function; (b) *reduce*: RDDs are aggregated in pairs by using a user-defined function that must be commutative and associative in order to perform the reduction step in parallel.

[8] As stated before, the master enqueues blocks when the local marginals have been computed by every worker.

[9] http://spark.apache.org/.

As described in Section 4, the operations that CLUBS-P performs on data are commutative and associative, therefore we can safely adopt Spark for implementing our algorithm, that we call CLUBS*.

Since the computation of marginal distributions is the most expensive operation and the most interesting, from a technical point of view, in the following we provide in Algorithm 1 the details of its implementation using Spark.

Algorithm 1: *Marginal Distribution Computation*

Input: **D**: RDD⟨Point⟩ representing points in a block *B*.
Output: Marginal distribution for *B*.
1: RDD⟨Map⟨Double, Integer⟩[]⟩ **mc** ← **D**.mapPartitions(new CMapper());
2: Map⟨Double, Integer⟩[] **C** ← **ms**. reduce(new CReducer());
3: RDD⟨Map⟨Double, Double[]⟩[]⟩ **mls** ← **data**.mapPartitions(new LSMapper());
4: Map⟨Double, Double[]⟩[] **LS** ← **mls**. reduce(new LSReducer());
5: **return** **C, LS**;

Lines 1–2 compute the *C* function over the input data, for every dimension. More specifically, line 1, computes this function for the data of each worker, through our object `CMapper`, which simply scans the data of the worker, updating a counter for each possible coordinate of each dimension. Then, line 2 aggregates the partial counts, using the object `CReducer`. The computation of *LS* functions is performed by lines 3–4. The object `LSMapper` is similar to `CMapper`, but instead of updating counters, it updates arrays representing the sum of points. The same holds for the object `LSReducer`.

In the following, we report a simple example that clarifies how map-reduce paradigm is applied by the algorithm.

Example 1. Consider a toy dataset consisting of 9 points partitioned over 2 nodes as follows::

$part_1 = \{(4, 4), (5, 7), (5, 8), (5, 4), (6, 4)\}$

and

$part_2 = \{(2, 7), (2, 5), (3, 6), (5, 7)\}$

The results of the mapping operations (lines 1 and 3) for each node and each dimension are reported in Table 1.

For instance, the points in partition 1 that have coordinate of dimension 2 equal to 4 are the following: (4, 4), (5, 4), and (6, 4). Therefore, for the dimension 2 and coordinate 4 **mc** is 3, and **mls** is $(4, 4) + (5, 4) + (6, 4) = (15, 12)$.

The results of the reducing operations (lines 2 and 4) for each dimension are reported in Table 2.

For instance, consider the coordinate 5 of dimension 1. **C** is obtained as $3 + 1 = 4$, and **LS** as $(15, 19) + (5, 7) = (20, 26)$.

Table 1
Results of the map operations.

$part_1$	dim 1	coord	4	5	6
		mc	1	3	1
		mls	(4, 4)	(15, 19)	(6, 4)
	dim 2	coord	4	7	8
		mc	3	1	1
		mls	(15, 12)	(5, 7)	(5, 8)
$part_2$	dim 1	coord	2	3	5
		mc	2	1	1
		mls	(4, 12)	(3, 6)	(5, 7)
	dim 2	coord	5	6	7
		mc	1	1	2
		mls	(2, 5)	(3, 6)	(7, 14)

Table 2
Results of the reduce operations.

dim 1	coord	2	3	4	5	6
	C	2	1	1	4	1
	LS	(4, 12)	(3, 6)	(4, 4)	(20, 26)	(6, 4)
dim 2	coord	4	5	6	7	8
	C	3	1	1	3	1
	LS	(15, 12)	(5, 2)	(3, 6)	(12, 21)	(5, 8)

7. Experimental results

As mentioned above, CLUBS-P shares the ideas of CLUBS$^+$. Since the clustering obtained by CLUBS-P for a given dataset is the same that obtained by CLUBS, we can say that the accuracy of CLUBS-P is the same as that of CLUBS. The latter has been proven to be better than that of existing algorithms [27], and, thus, we do not report accuracy results here, for space limitation and because our goal is rather to prove the effectiveness of the parallelization of CLUBS$^+$ in order to apply it in the Big Data scenario. We tested our parallel implementation on several synthetic datasets obtained by means of an open source synthetic data generator,[10] which generates data distributions where each cluster follows a possibly different gaussian distribution, while noise points are randomly added. We generated datasets with 10^7, 10^8 and 10^9 points. For each dataset size, we varied the dimensionality in the range [2..16] and the number of clusters in the range [2..32]. Finally, in order to test the robustness of our algorithm w.r.t. noise, we added a number of randomly distributed points ranging between 0 and 0.1 times the total number of points. Experiments were executed on a cluster having 16 computing machines (100 GB RAM available each). As previously said, our implementations, according to *Reproducible Research (RR)* policy are publicly available.[11]

We studied the running times obtained using 1, 2, 4, 8, or 16 workers,[12] varying one data parameter at a time, while maintaining constant the other ones. For the sake of presentation, the non-varying parameters, in each experiment were set as 10^8 points, 16 dimensions, 32 clusters, and 0.1 noise ratio, as this assignment has been found to be the most severe for testing the algorithm features.

In the following, we separately report the results obtained using *CLUBS-MP* and *CLUBS**, and finally we compare them.

7.1. Evaluating CLUBS-MP performances

Fig. 3 depicts the execution times.[13] obtained by using *CLUBS-MP*

As expected, the execution time significantly decreases as the number of workers increases. However, it is worth noticing that the speedup increases almost linearly with the number of workers, as shown by the about linear curve representing the running times.

In particular, Fig. 3(a) shows that for a large number of points, the speedup is kept constant as the number of worker increases.

10 https://github.com/gmmazzeo/clugen.

11 https://github.com/gmmazzeo/clubsmp and https://github.com/gmmazzeo/clubspark.

12 We ran one worker per different machine of the cluster, thus, the number of workers and machines here are the same.

13 These and the following graphics use logarithmic scale, for both x- and y-axis.

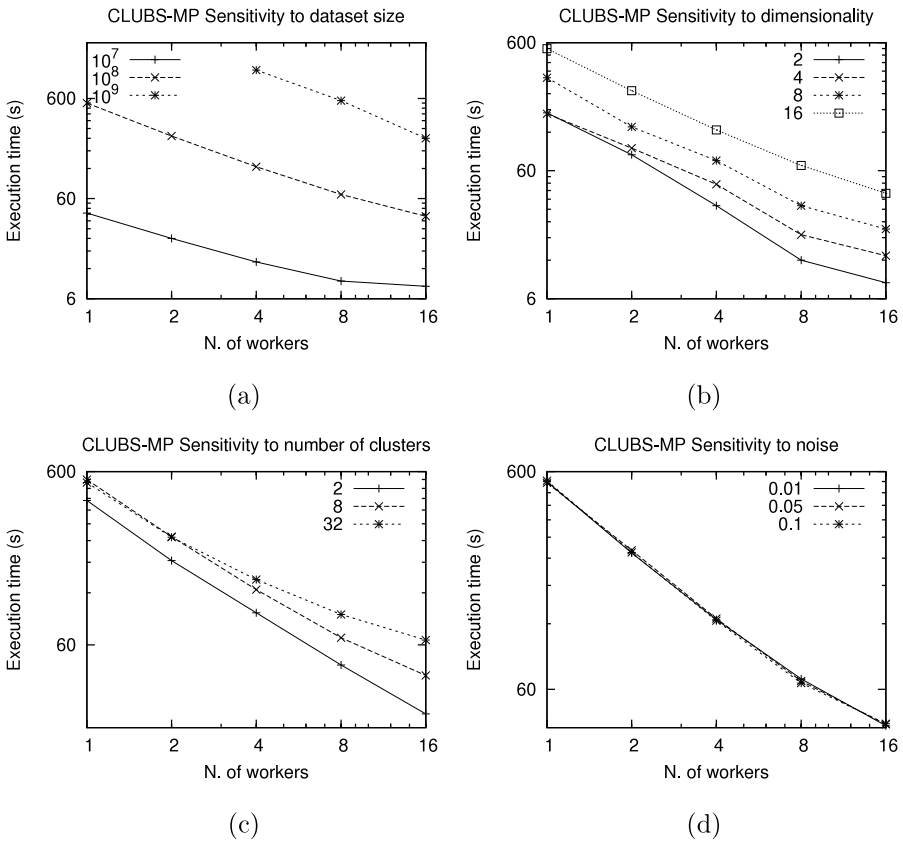

Fig. 3. CLUBS-MP Execution Times vs. number of workers varying dataset size (a), data dimensionality (b), number of clusters (c), and noise ratio (d).

We actually could not run experiments on 10^9 points using 1 or 2 workers, but the curves showing the results obtained on 10^8 points, clearly shows that the algorithm scales well for a relatively high number of points. Instead, if the number of points is not too high (e.g., 10^7), the overhead due to the network communication becomes more relevant w.r.t. the local computation. Thus, going from 8 to 16 workers, the speedup (i.e., the slope of the curve) decreases.

Fig. 3(b) shows that the speedup is not affected by data dimensionality. In fact, the slope of the curves for different dimensionality is about the same. Instead, the running time increases as the dimensionality increases. This is also true in the case of the non-parallel version of CLUBS, as in many other clustering algorithms (the computational cost usually linearly increases with the number of dimensions to be handled).

Fig. 3(c) shows that the speedup is slightly affected by number of clusters. Specifically, the speedup is slightly better for a small number of clusters. This depends on the fact that the divisive phase is longer for a larger number of clusters. This phase, furthermore, is the phase which requires the most of the overall communication between pairs of workers. Thus, while the local workload for workers does not change, as the number of clusters increases (we recall that the dataset size is kept constant), the time needed to exchange the marginal distribution increases, since a larger number of splits has to be done. Furthermore, the running times are larger as the number of clusters increases, but this is a feature inherited by the non-parallel version of CLUBS$^+$ (in particular, the running time of the divisive phase increases logarithmically with the number of clusters).

Finally, as regards the sensitivity to noise, the results shown in Fig. 3(d) are quite remarkable. In fact, both execution time and speedup are insensitive to the percentage of noise added to data. Again, this can be explained by the fact that the speedup is mainly affected by the communication between pairs of workers and, since data are not exchanged, but only summary information (the marginals), which does not depend on what data are processed (either noise or clusters).

7.2. Evaluating CLUBS*

In this section, we report the performances obtained using CLUBS* in the same experimental setting, which are depicted in Fig. 4. We found that the performances obtained by using the Spark-based implementation are less stable than the ad-hoc implementation. This is probably due to the fact that Spark uses some random shuffling of data which we cannot control. Furthermore, while in CLUBS-MP we assign a given portion of the dataset to each node, with Spark this is not achievable as the actual physical distribution of data among workers cannot be controlled. In general, we found a good speedup for 2 and 4 workers, while the speedup is reduced when the number of workers is further increased. This is mainly due to the overhead introduced by the Spark framework. In terms of sensitivity to the data parameters, from the experimental results we can observe that the speedup of CLUBS* is not affected by the data dimensionality and noise ratio (the absolute running time, instead, is affected by dimensionality, as expected). For the sensitivity to the number of clusters, like in the case of CLUBS-MP the speedup is reduced when the number of clusters increases, even in this case, because the divisive phase is the phase requiring the most network communication, and its duration increases with the number of clusters.

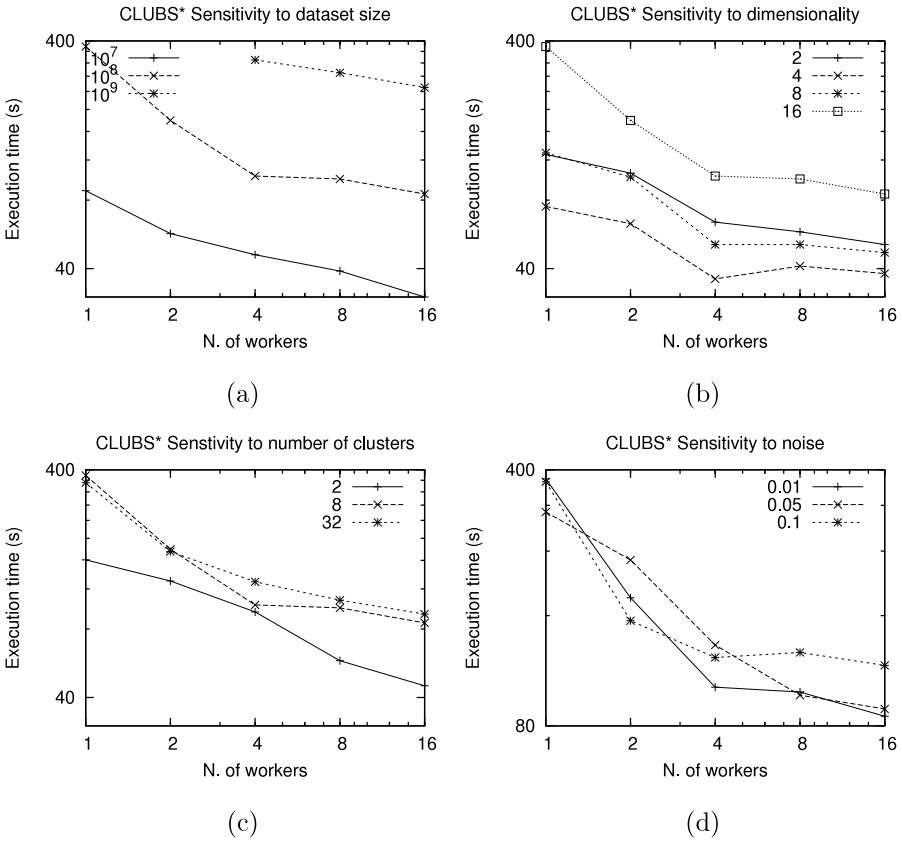

Fig. 4. CLUBS* Execution Times vs. number of workers varying dataset size (a), data dimensionality (b), number of clusters (c), and noise ratio (d).

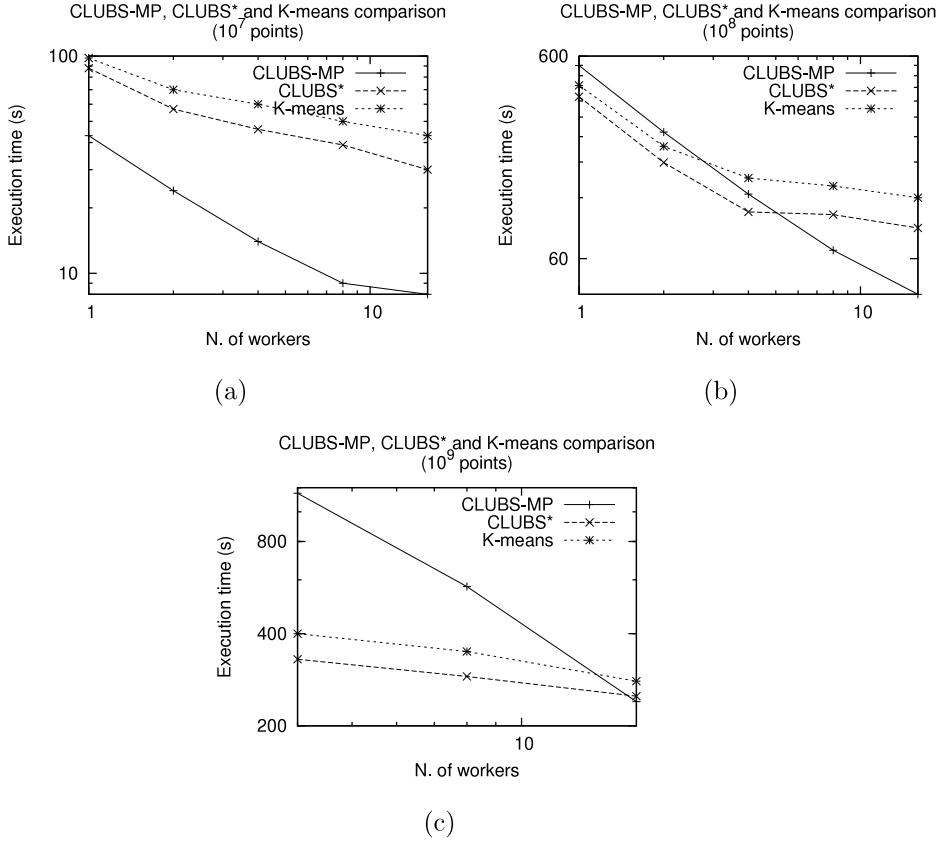

Fig. 5. CLUBS* and CLUBS-MP vs. k-means with 10^7 (a), 10^8 (b), and 10^9 (c) points.

7.3. Wrap-up

We now compare the performance of CLUBS-MP and CLUBS* w.r.t. the implementation of a recent and popular improvement of k-means, namely k-means‖ [23] that is available for Spark, then, we used this algorithm to our comparative analysis. Fig. 5 reports the execution times of the three algorithms for different dataset sizes w.r.t. k-means execution times. All the three datasets are 16-dimensional, have 32 clusters, and have 0.01 noise ratio (that is the most stressing test bench as mentioned above).

We can observe that CLUBS-MP scales better, i.e., its running times decrease faster w.r.t. the number of workers while the performances of CLUBS* are always better than k-means‖. This depends mainly on the fact that CLUBS-MP is an ad-hoc algorithm, which leverages the available workers in an optimal way, by using a protocol tailored for the algorithm. This is not clearly possible in the Spark-based implementation, since Spark is a general purpose framework. However, for large datasets and few workers, CLUBS* is faster than CLUBS-MP. We attribute this result to the fact that Spark leverages the multi-core architecture of each worker, while CLUBS-MP does not take into account the availability of multiple cores. These results are quite relevant for our goal. In particular, they show that Spark framework offers a good opportunity for implementing effective parallel strategies when dealing with really huge amounts of data. However, the overhead due to the abstractions provided by Spark reduce the possibility to have an algorithm that scales optimally w.r.t. the available workers.

8. Conclusion and future work

In this paper we discussed CLUBS-P, an algorithm for around-centroid clustering for Big Data. We evaluated two different implementations of CLUBS-P, in order to perform a kind of feasibility analysis w.r.t. the big data setting. The results obtained show a very good scalability of the algorithm, and its improvement over the current state-of-the-art parallel clustering algorithm, i.e., k-means‖. Furthermore, if the application scenario being analyzed does not require all the features offered by Spark, the implementation of CLUBS based on message-passing can offer an optimal usage of the resources of the available nodes. As a future work, we plan to test other partitioning and recombination strategies in order to further improve our (satisfactory indeed) performances.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] D.A. et al., Challenges and opportunities with big data, in: A Community White Paper Developed By Leading Researchers Across the United States, 2012, pp. 1–12.

[2] P. Wu, Z. Lu, Q. Zhou, Z. Lei, X. Li, M. Qiu, P.C.K. Hung, Bigdata logs analysis based on seq2seq networks for cognitive internet of things, Future Gener. Comput. Syst. 90 (2019) 477–488, http://dx.doi.org/10.1016/j.future.2018.08.021.

[3] G. Manogaran, D. Lopez, N. Chilamkurti, In-mapper combiner based mapreduce algorithm for processing of big climate data, Future Gener. Comput. Syst. 86 (2018) 433–445, http://dx.doi.org/10.1016/j.future.2018.02.048.

[4] G.M. Mazzeo, E. Masciari, C. Zaniolo, A fast and accurate algorithm for unsupervisedclustering around centroids, Inform. Sci. 400.

[5] E. Masciari, G.M. Mazzeo, C. Zaniolo, A new, fast and accurate algorithm for hierarchical clustering on euclidean distances, in: Advances in Knowledge Discovery and Data Mining, 17th Pacific-Asia Conference, PAKDD 2013, Gold Coast, Australia, April 14-17, 2013, Proceedings, Part II, 2013, pp. 111–122.

[6] E. Masciari, G. Mazzeo, C. Zaniolo, Analysing microarray expression data through effective clustering, Inform. Sci. 262 (0) (2014) 32–45.

[7] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, in: HotCloud'10, USENIX Association, Berkeley, CA, USA, 2010, p. 10.

[8] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, E.M. Nguifo, An experimental survey on big data frameworks, Future Gener. Comput. Syst. 86 (2018) 546–564, http://dx.doi.org/10.1016/j.future.2018.04.032.

[9] A. Shirkhorshidi, S. Aghabozorgi, T. Wah, T. Herawan, Big data clustering: A review, in: B. Murgante, S. Misra, A. Rocha, C. Torre, J. Rocha, M. Falcão, D. Taniar, B. Apduhan, O. Gervasi (Eds.), Computational Science and Its Applications – ICCSA 2014, in: Lecture Notes in Computer Science, vol. 8583, Springer International Publishing, 2014, pp. 707–720.

[10] L. Kaufman, P.J. Rousseeuw, Finding groups in data: An introduction to cluster analysis, John Wiley and Sons, 1990, URL http://www.amazon.com/Finding-Groups-Data-Introduction-Analysis/dp/0471878766.

[11] R. Ng, J. Han, Clarans: a method for clustering objects for spatial data mining, IEEE Trans. Knowl. Data Eng. 14 (5) (2002) 1003–1016.

[12] L. Kaufman, P. Rousseeuw, Clustering by Means of Medoids, in: Reports of the Faculty of Mathematics and Informatics, Faculty of Mathematics and Informatics, 1987.

[13] X.Z. Fern, C.E. Brodley, Random projection for high dimensional data clustering: A cluster ensemble approach, in: Proceedings of the Twentieth International Conference on Machine Learning, 2003, pp. 186–193.

[14] H. Tong, S. Papadimitriou, J. Sun, P.S. Yu, C. Faloutsos, Colibri: Fast mining of large static and dynamic graphs, in: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, in: KDD '08, ACM, New York, NY, USA, 2008, pp. 686–694.

[15] A. Lulli, M. Dell'Amico, P. Michiardi, L. Ricci, Ng-dbscan: scalable density-based clustering for arbitrary data, Proc. VLDB Endowment 10 (3) (2016) 157–168.

[16] X. Hu, J. Huang, M. Qiu, A communication efficient parallel DBSCAN algorithm based on parameter server, in: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, ACM, 2017, pp. 2107–2110.

[17] E. Januzaj, H.-P. Kriegel, M. Pfeifle, Dbdc: Density based distributed clustering, in: E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Böhm, E. Ferrari (Eds.), Advances in Database Technology - EDBT 2004, in: Lecture Notes in Computer Science, vol. 2992, Springer Berlin Heidelberg, 2004, pp. 88–105.

[18] G. Karypis, V. Kumar, Parallel multilevel k-way partitioning scheme for irregular graphs, in: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, in: Supercomputing '96, IEEE Computer Society, Washington, DC, USA, 1996.

[19] I.K. Savvas, D.C. Tselios, Parallelizing dbscan algorithm using MPI, in: 25th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2016, Paris, France, June 13-15, 2016, 2016, pp. 77–82.

[20] W. Zhao, H. Ma, Q. He, Parallel k-means clustering based on mapreduce, in: Proceedings of the 1st International Conference on Cloud Computing, in: CloudCom '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 674–679.

[21] Y. He, H. Tan, W. Luo, S. Feng, J. Fan, Mr-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data, Front. Comput. Sci. 8 (1) (2014) 83–99.

[22] R.L. Ferreira Cordeiro, C. Traina, A.J. Machado Traina, J. López, U. Kang, C. Faloutsos, Clustering very large multi-dimensional datasets with mapreduce, in: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, in: KDD '11, ACM, New York, NY, USA, 2011, pp. 690–698.

[23] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, S. Vassilvitskii, Scalable k-means++, PVLDB 5 (7) (2012) 622–633.

[24] S. Muthukrishnan, V. Poosala, T. Suel, On rectangular partitionings in two dimensions: Algorithms, complexity, and applications, in: ICDT, 1999, pp. 236–256.

[25] O. Arbelaitz, I. Gurrutxaga, J. Muguerza, J.M. PéRez, I.n. Perona, An extensive comparative study of cluster validity indices, Pattern Recognit. 46 (1) (2013) 243–256.

[26] T. Calinski, J. Harabasz, A dendrite method for cluster analysis, Comm. Statist. Theory Methods 3 (1) (1974) 1–27, http://dx.doi.org/10.1080/03610927408827101.

[27] CLUBS+ website, http://yellowstone.cs.ucla.edu/clubs/, accessed: 2016-03-25.

Michele Ianni received the PhD degree in Information and Communication Technologies from the University of Calabria, Italy, in 2018. He was a visiting researcher in SecLab, University of California, Santa Barbara. Currently, he is a postdoc in the DIMES Department, University of Calabria. His main research interests include binary exploitation, malware and trusted execution environments.

Elio Masciari is currently Associate professor at Federico II University of Naples. He was for 17 years a senior researcher at the Institute for High Performance Computing and Networks (ICAR-CNR) of the National Research Council of Italy. His research interests include Database Management, Semistructured Data and Big Data. He has served as a member of the program committee of several international conferences. He served as a reviewer for several scientific journals of international relevance. He is author of more than 120 publications on journals and both national and international conferences. He also holds "Abilitazione Scientifica Nazionale" for Full Professor role.

Giuseppe M. Mazzeo received the PhD degree in computer science and system engineering from the University of Calabria, Italy, in 2007. He was a Postdoctoral Scholar at Computer Science department of University of California, Los Angeles. His research interests include Data and Text Mining, Natural Language Processing, Information Retrival, Data Streams, and compression techniques for multidimensional data. Currently, he works for Facebook.

Mario Mezzanzanica is Associate Professor of Information Systems at the University of Milan Bicocca. He is the Scientific Director of the CRISP centre - Inter University Research Centre on public services. His research interests include Information Systems, Databases, Artificial Intelligence, Business Intelligence and Knowledge Discovery. He has a strong experience in applying his research to real-life scenarios, as he partnered with the role of advice and coordination in several projects for innovation of public services at both national and regional level. In 2012 he founded the Tabulaex spin-off company that realises novel solutions for analysing labour market data through AI techniques (aka Labour Market Intellingence). In 2018 Tabulaex has been acquired by Burning Glass. He is member of the editorial board of international journals in the field of Artificial Intelligence and Information Systems. He authored more than 100 scientific papers. Since 2010 he is director of the Master Business Intelligence and Big Data Analytics provided by University of Milan Bicocca. Since October 2018 he is the Head of the Statistical and Quantitative Methods Department.

Carlo Zaniolo is a professor Computer Science at the University of California at Los Angeles, where he occupies the N.E. Friedmann chair in Knowledge Science and he is acting director of the Scalable Analytics Institute in the UCLA School of Engineering and Applied Science. Carlo's recent research interests focus on advanced Big Data systems and on extending of database technology to support scalable data mining and machine learning applications.