Artificial Intelligence Review
https://doi.org/10.1007/s10462-020-09918-2

A survey on parallel clustering algorithms for Big Data

Zineb Dafir[1] · **Yasmine Lamari**[1] · **Said Chah Slaoui**[1]

Abstract
Data clustering is one of the most studied data mining tasks. It aims, through various methods, to discover previously unknown groups within the data sets. In the past years, considerable progress has been made in this field leading to the development of innovative and promising clustering algorithms. These traditional clustering algorithms present some serious issues in connection with the speed-up, the throughput, and the scalability. Thus, they can no longer be directly used in the context of Big Data, where data are mainly characterized by their volume, velocity, and variety. In order to overcome their limitations, the research today is heading to the parallel computing concept by giving rise to the so-called parallel clustering algorithms. This paper presents an overview of the latest parallel clustering algorithms categorized according to the computing platforms used to handle the Big Data, namely, the horizontal and vertical scaling platforms. The former category includes peer-to-peer networks, MapReduce, and Spark platforms, while the latter category includes Multi-core processors, Graphics Processing Unit, and Field Programmable Gate Arrays platforms. In addition, it includes a comparison of the performance of the reviewed algorithms based on some common criteria of clustering validation in the Big Data context. Therefore, it provides the reader with an overall vision of the current parallel clustering techniques.

Keywords Algorithms · Big Data · Clustering · Data mining · DBSCAN · FPGA · GPU · *k*-means · MapReduce · MPI · Multi-cores CPU · Spark

1 Introduction

With the advent of the Big Data phenomenon, the data analysis techniques are currently being modernized in order to address the emerging challenges. Data clustering is no exception to this trend. This long-established data mining technique is used to partition a set

✉ Zineb Dafir
zineb.dafir@um5s.net.ma

Yasmine Lamari
yasmine.lamari@um5s.net.ma

Said Chah Slaoui
said.slaoui@menara.ma

[1] Faculty of Science of Rabat, Mohammed V University, Rabat, Morocco

Published online: 06 October 2020

of data instances into homogeneous subsets, such that each subset is formed by similar instances, and at the same time dissimilar to instances belonging to other subsets (Han et al. 2012). The primary objective is to discover previously unknown groups, which is a sought-after result in several problems in everyday life. This can be achieved through different categories of clustering methods such as hierarchical methods, partitioning methods, density-based methods, grid-based methods, or other clustering techniques (Fahad et al. 2014).

1.1 Challenges

Most traditional clustering algorithms are specialized and operate under specific conditions to solve a particular type of problem. Besides, they are outdated and impractical in the context of the Big Data due to their computational costs and their inability to handle heterogeneous data. They also suffer from their dependence on prior knowledge, data order, and complex input parameters. In order to overcome these limitations, the research today is heading to the parallel computing concept by giving rise to the so-called parallel clustering algorithms. As the designation implies, this kind of algorithms can partition data sets into several chunks, and then for each chunk, execute activities simultaneously on one or on many processing devices. The intermediate clustering results are usually aggregated at the end to produce the final result. The concept of parallelism aims to improve the speed-up, the throughput, and the scalability of the clustering process so that it becomes effective to meet the challenges of Big Data.

These challenges arise primarily from the characteristics that define the Big Data, namely the *volume*, the *velocity*, and the *variety* of data, which stand for the 3V's definition proposed by the Gartner company in Laney (2001). The first dimension refers to the quantity of generated and stored data. Recently, the volume of data ranges from *petabytes* toward *zettabytes* in a continued increase. Then, the second dimension represents the frequency at which the data is coming and updated because the data is constantly in motion. The third dimension to be also considered is the variety of data, since data is collected from multiple sources and in different forms. These dimensions describe the Big Data phenomenon, and they serve also as criteria for evaluating algorithms that attempt to solve the Big Data issues such as clustering algorithms.

1.2 Scope of the article

This paper presents an overview of the latest parallel clustering algorithms categorized according to the computing platforms used to handle the Big Data. Indeed, at some point, we need to know what has been accomplished and what remains to be done regarding the clustering of Big Data. Accordingly, this paper aims to provide the reader with an overall vision of the parallel methods of clustering Big Data that have been developed recently as well as the current trend of the research in such relevant field. In this respect, we opted for a recent classification of the different platforms for the Big Data analytic which is proposed in Singh and Reddy (2014). According to this classification, two categories of platforms of Big Data can be distinguished: the horizontal scaling platforms and the vertical scaling platforms. The first category gathers systems that distribute the workload across many servers or commodity machines. And so it includes peer-to-peer networks, MapReduce, and Spark platforms. While the second category brings together systems that work on a single server and allows adding additional resources, such as processors, memory, and fast

hardware. This category includes High Performance Computing Clusters (HPC), Multicore processors, Graphics Processing Unit (GPU), and Field Programmable Gate Arrays (FPGA) platforms. Figure 1 illustrates the classification of different Big Data platforms.

1.3 Contributions

This section explores the fundamental contributions of this paper as follows:

1. Presents a technical overview of different Big Data platforms.
2. Surveys the new parallel clustering algorithms and their categorization according to the platforms adopted to ensure their parallelization.
3. Introduces a comparison of the algorithms studied in terms of clustering validation criteria and Big Data characteristics.

1.4 Organization

The remainder of this survey is organized as follows: Sect. 2 reviews the most popular platforms of Big Data. Section 3 exposes the most recent parallel clustering methods classified based on the used platform. Section 4 compares the studied parallel clustering algorithms.

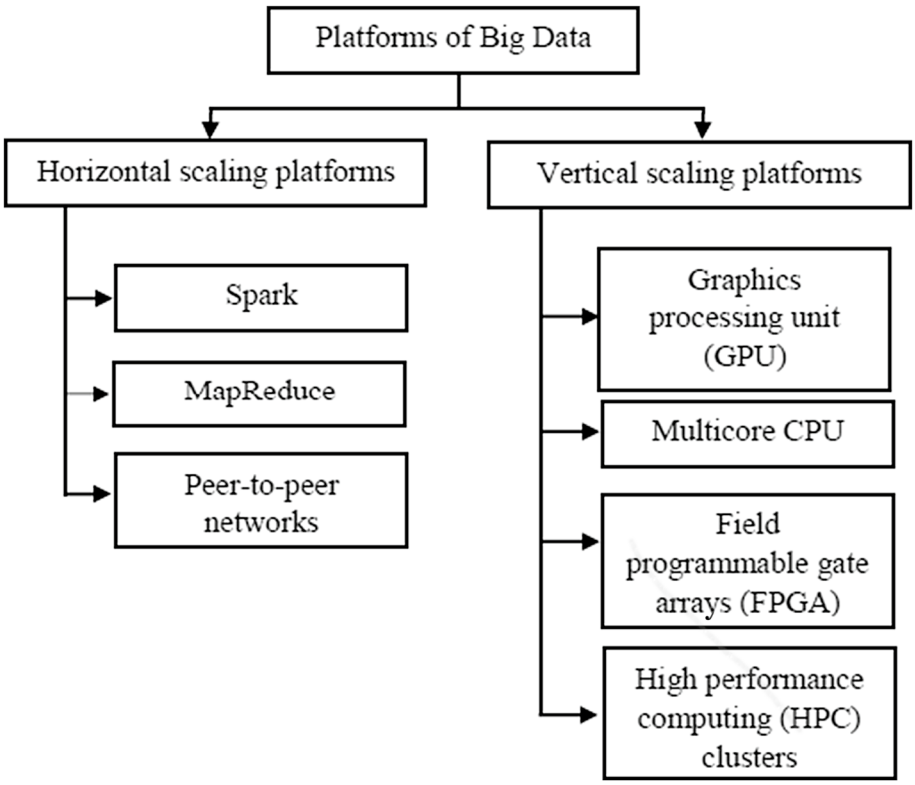

Fig. 1 Classification of Big data platforms

Finally, Sect. 5 summarizes the main observations of this survey and the corresponding recommendations.

2 Overview of the different platforms of Big Data

This section presents a brief overview of the most popular platforms of Big Data. First, the horizontal scaling platforms are presented, including MapReduce, Spark, and Peer-to-peer networks. Then, the vertical scaling platforms are presented, which comprise Graphics Processing Unit, Multi-core CPU, and Field Programmable Gate Arrays. These platforms have been the basis for several designs and clustering algorithms of which some relevant works are examined in this paper.

2.1 MapReduce

MapReduce is a popular parallel programming model, which was first introduced by Google in Dean and Ghemawat (2004). It was designed to read, process, and write a huge amount of data. This programming model consists of two main functions: the Map and the Reduce functions. The Map function takes as input a logical record (also called a chunk of data) and produces a set of intermediate key/value pairs. Once the Map phase is completed, the Reduce phase starts processing the intermediate key/value pairs produced in the previous phase. Indeed, the Reduce function takes as input the set of intermediate key/value pairs that share the same key, and then, merges together all values assigned to the input key in order to produce the set of values associated with the corresponding key. There are two other optional functions used to refine the execution of the programs: the Partitioner and the Combiner functions. The Partitioner function splits the intermediate keys according to the number of Reduce tasks or the number of output files which is specified by the developer. The Combiner function aims to summarize intermediate results produced by each Map task in order to avoid potential repetitions, thus optimizing the transfer of data to the Reduce task over the network. All these functions are programmable by the developer. Figure 2 describes the operational flowchart of the MapReduce programming model (Dean and Ghemawat 2004).

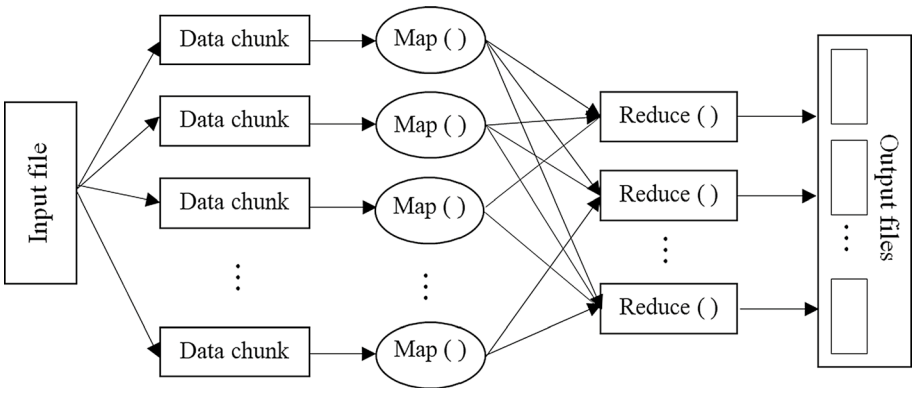

Fig. 2 Flowchart of MapReduce model

2.2 Spark

Apache Spark is a big data processing framework designed for data-intensive applications and executed on commodity clusters (Zaharia et al. 2010). Unlike MapReduce, Spark framework supports iterative jobs and allows running queries on big datasets by loading only the useful dataset into memory. In this way, the execution time is reduced considerably.

Spark introduces three fundamental aspects: resilient distributed datasets (RDDs), parallel operations, and shared variables. RDD is a collection of objects shared by a set of machines that can be recovered in case of loss; it can be also stored in memory to reuse it in multiple parallel MapReduce jobs. The second aspect is the parallel operations which can be performed on RDDs. It includes the Reduce, *Collect* and *Foreach* operations. The last aspect consists of broadcast variables and accumulators.

Spark is flexible, easy to use and does not need any abstraction to program. It processes the data in real-time using the Spark Streaming module and caches partial results in memory using distributed workers. Moreover, Spark is efficient and exceeds the Hadoop MapReduce framework by 10× in interactive machine learning workloads while preserving the fault tolerance and scalability of MapReduce. However, Spark shows some limitations, such as the fact that it requires large resources, and it is also expensive in terms of memory. Figure 3 depicts the operational flowchart of the Spark model (Zaharia et al. 2010).

2.3 Peer-to-peer networks

Peer-to-peer (P2P) networking represents a distributed architecture that divides tasks among peers. The definition of P2P networking includes any type of network architecture which does not need a server to control the transfer of information between contributors, and which makes a part of their resources accessible by other contributors in the same network (Milojicic et al. 2002). Hence, P2P systems allow valuable externalities, lower cost of ownership and sharing, and finally anonymity. The most used scheme in this platform is the Message Passing Interface (MPI). The fundamental idea behind the standard MPI is to provide the necessary abstractions in order to ensure the communication between peers. It is also characterized by its ability to keep processes alive during

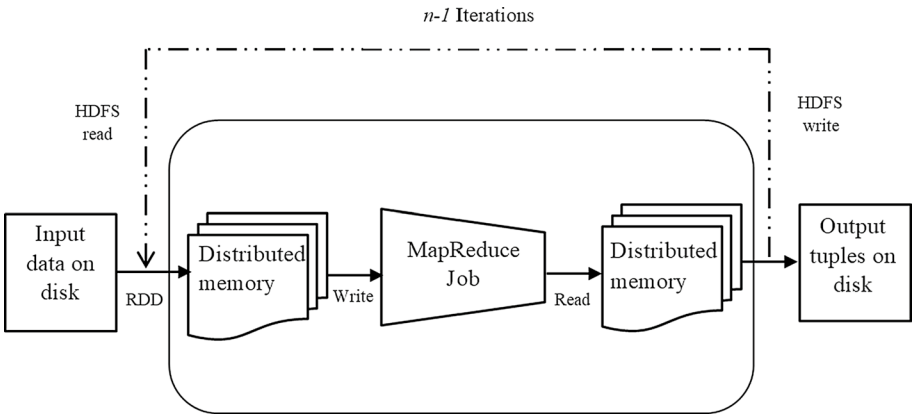

Fig. 3 Flowchart of Apache Spark model

the running of the system, which means that there is no need to read the data several times from the disk. This quality makes the MPI suitable for iterative jobs.

The Architecture of P2P network has the ability to add nodes dynamically and thus to scale up efficiently to a larger size that may be needed to process applications with a huge amount of data, as shown in Fig. 4. This kind of platforms is known to be resistant to failures, which means that when a node experiences a local failure, it will have no impact on the remaining nodes. Another advantage lies in the fact that the capacity of a P2P system increases with the addition of new nodes. However, adding new nodes to the system can slow down the transfer of data to the connected users. P2P systems suffer also from security issues and require high bandwidth usage.

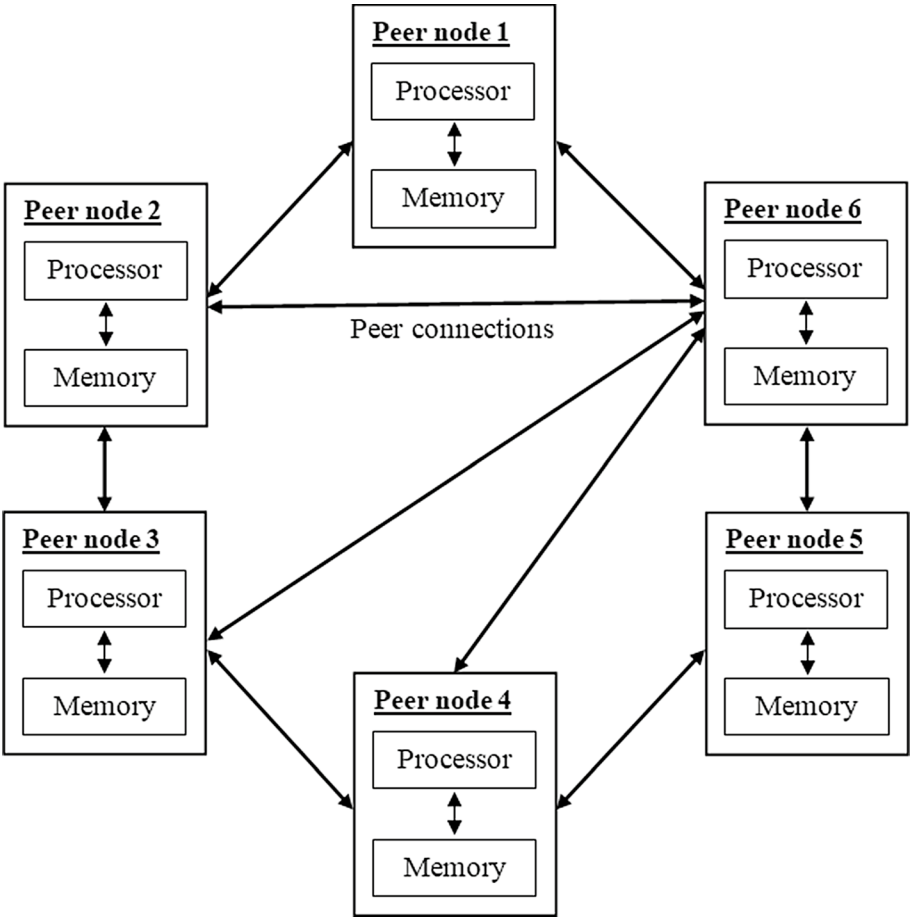

Fig. 4 A typical architecture of the peer-to-peer network

2.4 Graphics processing unit

A Graphics Processing Unit (GPU) is a single-chip processor which was initially dedicated for processing 2D and 3D calculations. As an input, it receives geometry primitives from the CPU in a 3D form. Then it transforms them from individual vertices into pixels, which are shaded and mapped onto the screen. In order to produce the final image, the processed pixels are combined to form an output destined to be visualized in a display device. These stages form the so-called graphics pipeline as shown in Fig. 5 (Owens et al. 2008). The GPUs follow the single program multiple data (SPMD) as a programming model and the single instruction, multiple data (SIMD) as parallel architecture.

Today, modern GPUs are gaining increasing attention due to their massively parallel processing architecture that accelerates the performance of applications requiring high floating point calculations. Indeed, GPUs are not beneficial only for graphics applications; they are also used to perform non-specialized calculations giving rise to the general purpose graphics processing unit (GPGPU) computing. In order to take full advantage of the parallelism offered by GPUs, new parallel programming languages, such as CUDA (Nickolls et al. 2008) and OpenCL (Stone et al. 2010), have emerged. These languages simplify and improve the performance of the linear algebra on the GPUs (Owens et al. 2008).

GPUs have demonstrated energy and cost efficiency for arithmetical intense and streaming-memory problems. However, their major drawback lies in the limited memory capacity, which often necessitates complex memory management.

2.5 Multi-core CPU

A Multi-core platform is a processor that integrates multiple cores in a single chip. Generally, there are three commonly recognized architectures of multi-core CPU (Akhter and Roberts 2006). The first architecture shares the on-chip cache between execution units, while the second architecture provides a dedicated cache for each execution core. The third architecture adopts a hybrid approach that subdivided the cache into two types of layers,

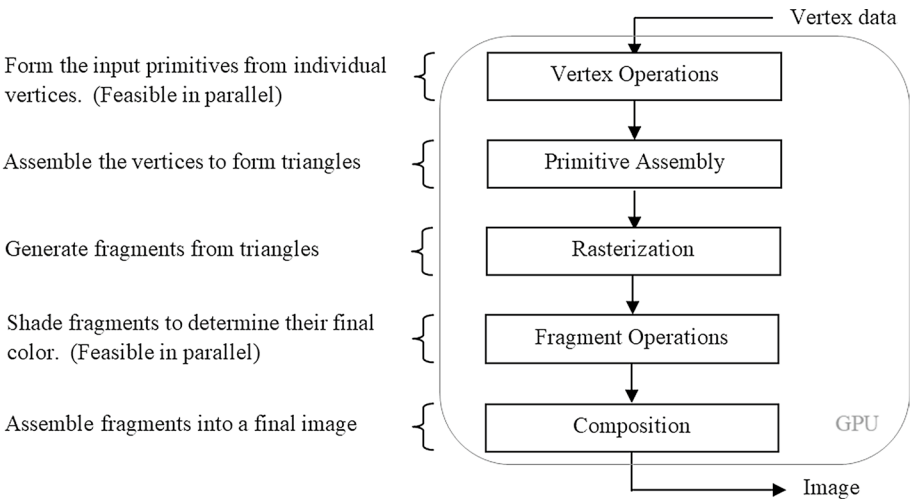

Fig. 5 A typical structure of graphics pipeline

layers dedicated to a particular execution core and others shared by all execution cores. Figure 6 describes a typical architecture of multi-core platform.

In order to take advantage of the parallelism offered by the multi-core platforms, one has to consider the allocation of the work among all available processors (Gepner and Kowalik 2006). The multi-threading model is a common way that allows the parallel execution on a multi-core platform. This is achieved by dividing the work into separate execution units which can run on the different processors at the same time.

The multi-core platforms offer a high performance and low heat generation. They are useful for applications with high capacity of parallelism that can profit from all the available cores, otherwise, the multi-core platforms won't be a practical choice. This entails a substantial effort to parallelize an application as far as possible, which is not always feasible.

2.6 Field programmable gate arrays

A Field Programmable Gate Arrays (FPGA) is an integrated circuit device. It is composed of an array of programmable logic blocks, in addition to a hierarchy of reconfigurable interconnects and I/O blocks (Brown et al. 1992). Different architectures of logic blocks can be conceived in order to form a complex circuit. Such circuits contain several other subcircuits and have more than one output. These blocks are interconnected via reconfigurable interconnects, which consists of wire segments and programmable switches. Like the logic blocks, the structure of programmable switches can be conceived in different ways. The configurations of all components of the FPGA are described using a hardware description language (HDL).

The main advantages of FPGAs lie in the reduced costs of prototypes they offer, in addition to their expandability and flexibility. In fact, the flexibility is regarded as both an advantage and a drawback since it makes the FPGAs larger, slower, and more power consuming (Farooq et al. 2012).

The FPGAs are omnipresent in various applications and can figure out any computational problem, especially the applications that require exploiting the parallelism available on FPGAs.

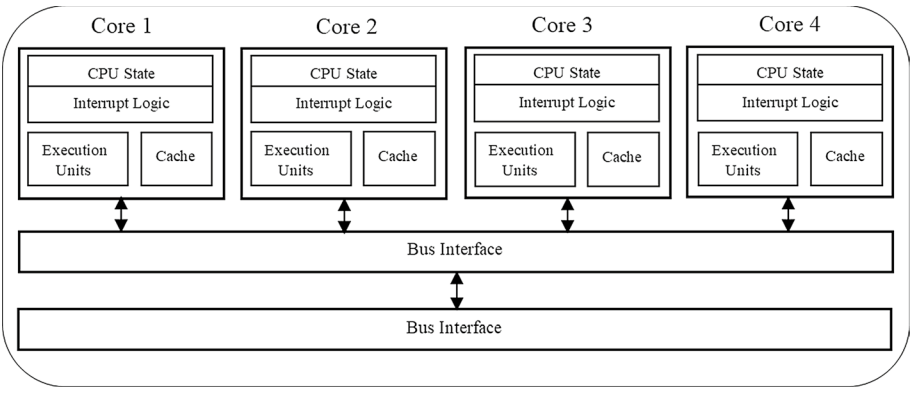

Fig. 6 A typical architecture of multi-core platform

3 Parallel clustering algorithms

This section exposes the most recent and relevant parallel algorithms for clustering Big Data. The aim is to explore a variety of types of clustering, such as partitioning-based clustering, density-based clustering, bio-inspired methods and many other techniques of clustering. First, we describe the parallel clustering algorithms based on horizontal scaling platforms. Then, we describe the parallel clustering algorithms based on vertical scaling platforms. Figure 7 presents the main works reviewed in this paper.

3.1 Horizontal scaling platforms-based clustering algorithms

As mentioned above, the horizontal scaling platforms considered in this survey are MapReduce, Spark, and Peer-to-Peer networks. This section covers a selection of clustering algorithms that are implemented using this kind of platforms.

3.1.1 Clustering algorithms using MapReduce

The work proposed in Cui et al. (2014) is a processing model in MapReduce which eliminates the iteration dependence of the *k*-means algorithm through a sampling technique. The

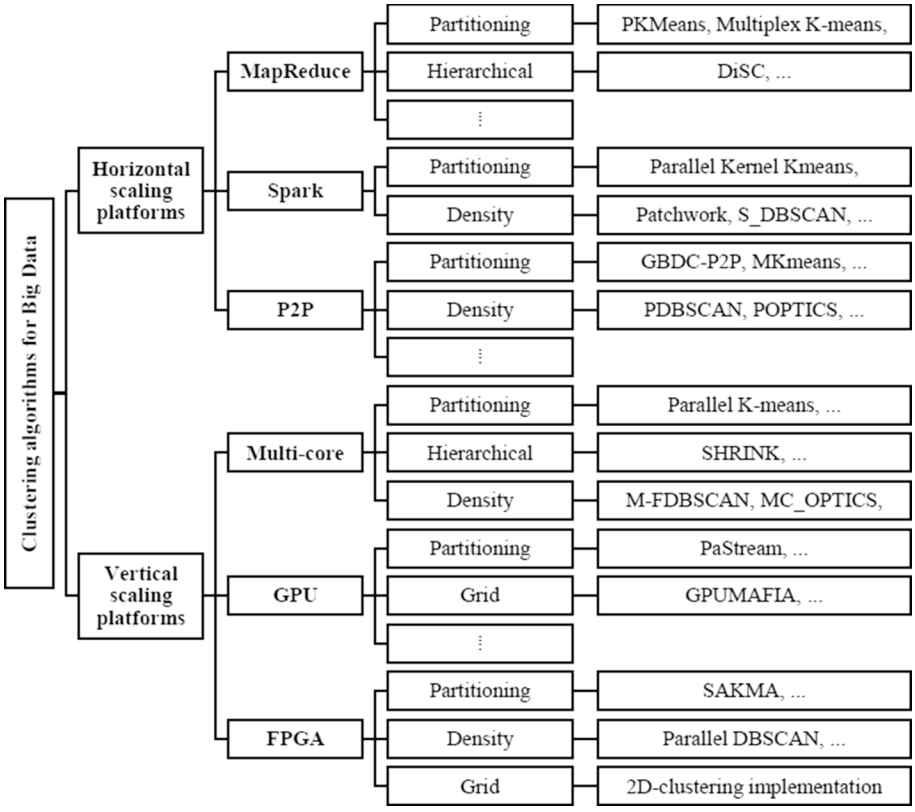

Fig. 7 Parallel clustering algorithms for Big Data

main idea of this optimized *k*-means is to estimate the iterations using a sampling technique in order to obtain only some subsets from big datasets. Then, by processing these subsets, the sets of centers are constructed and used to cluster the original datasets. The proposed algorithm consists of three MapReduce jobs. The first job is responsible for sampling the original dataset. The second job performs the samples clustering step in mappers, and then the merging step in one reducer in order to produce *k* final centers from the intermediates centers. For this purpose, the authors introduced two novel methods for merging: *weight-based merge clustering* (WMC) and *distribution-based merge clustering* (DMC). At last, the third job generates the Voronoi diagram using *k* points from the previous job, partition the original dataset, and then, obtain the final clustering result. The experiments on synthetic and real datasets show that the proposed method performs better compared with other parallel versions of the *k*-means algorithm.

In the same context, Li et al. (2014) suggested Multiplex *k*-means, a parallel execution of multiple *k*-means processes using MapReduce. In the proposed method, several processes are launched serially, then only the best result is considered. The execution of these processes is costly in terms of time and resources, unlike the proposed Mux *k*-means algorithm, which simultaneously runs multiple *k*-means using different centroid groups, and keeps the best solution at the end. The proposed algorithm involves four steps. It first runs the *k*-means processes using Map and Reduce operations. The Map operation calculates the distance between the current point and all existing centroids, while the Reduce operation is responsible for updating the centroids. After each iteration, the quality of clustering result is evaluated based on the Total Within-Cluster Variation value (TWCV). The TWCV metric is the sum of the distances between each point and the centroids of each group. More the value of TWCV is lower more the quality of clustering is higher. Then, in the second step, the groups with a lower value of TWCV are pruned. The third step consists of swapping similar centroids. Finally, the last step consists of generating new centroids using two methods: the *Random Search with a definite Scope* (RSDS) and the *Average of Dissimilar Group Pairs* (ADGP). This process is repeated until the centers become steady. The algorithm was implemented using the Hadoop MapReduce framework and tested with real-life datasets. The experiments show that the Mux-*k*-means can achieve better results than when using other serial versions of the *k*-means algorithm.

MR-DBSCAN (He et al. 2014) is a MapReduce-based implementation of the well-known DBSCAN algorithm. This work introduces a new cost-based data partitioning method in order to take into account the density of points. The proposed method is a 3-stage end-to-end solution. The first stage, which is called data partitioning, produces small partitions from the whole dataset according to the spatial proximity. The second stage, which is called *local clustering*, groups partitions independently. Finally, the third stage, which is called *global merging*, aggregates the produced partitions in the previous stage in order to form the final result. This latter is achieved in two steps: *build merge mapping* and *relabel data*. The build merge mapping step first determines all pairs of intersecting partitions, then computes the global clusters and builds a mapping from local to global clusters. And the relabel data step adjusts the intermediate results of local clustering by replacing local cluster ID's with global ones and determines the type of all points. The MR-DBSCAN was evaluated using two large real-life datasets. A set of experiments were carried out in order to study the performance changes when varying some sensitive parameters related to the DBSCAN algorithm, such as *MinPts*, ε and partitioning methods. The results of experiments confirm the efficiency and scalability of the proposed method.

MR-ABC (Banharnsakun 2017) is a MapReduce-based implementation of the artificial bee colony (ABC) for large-scale data clustering. The proposed method optimizes

the assignment of large data to clusters through the minimization of the sum of squared Euclidean distance. The main operations involved in this method are to update centroids of clusters and to evaluate the fitness. First, the proposed method generates the initial solutions. Then, it updates the new centroid values for the employed bees. After that, the fitness is calculated and evaluated based on the sum of the squared Euclidean distance. Since this task is time-consuming, the author uses the MapReduce model in order to calculate the fitness value. Then, each onlooker bee selects the centroid values which produce a higher fitness from employed bees and updates them. This process reiterates until the number of iterations reaches a threshold value. In order to evaluate the MR-ABC algorithm, experiments were conducted on synthetic and real-life datasets. The results obtained show that the proposed method outperforms the PKMeans (Zhao et al. 2009) and the parallel K-PSO (Wang et al. 2012) algorithms in terms of quality. Besides, the running time and speed-up results of MR-ABC with 10 Hadoop cluster nodes demonstrated that the MR-ABC can process large amounts of data in reasonable time.

In Yang and Li (2013), the authors studied the Ant Colony optimization algorithm in order to propose a new approach for big data semantic clustering (MBSC) based on MapReduce. The parallel implementation of Ant Colony Clustering method based on semantic content involves a single MapReduce job. The Map task splits the data records into data chunks. Then, it defines the key-value pairs as the length of traversal path without dropping records and the set of traversed nodes. In each step, the Map function reads the pheromone value and calculates the swarm similarity, which is transformed to probability value, in order to decide either to drop or to select a record. In this way, the data records with the same similarity are gathered in the same cluster. The reduce task collect solutions from all data chunks given by ants and then update the pheromone value for a next iteration. The process is repeated until the final result is reached. The comparison shows that the MBSC algorithm is more efficient than *k*-means in terms of time when considering different MSBC parameters.

In Jin et al. (2013), the authors proposed a parallel method of spectral clustering using MapReduce. The spectral clustering is based on the evaluation of the sparse matrix eigenvalue. The calculation of the similarity matrix and other parameters used in this algorithm are expensive, hence the need to use MapReduce in order to reduce the computation time. The proposed parallelization of the spectral clustering consists of three steps. The first step calculates the similarity matrices, which are simplified by the use of the Map and Reduce operations. The second step calculates the k minimum feature vectors using the Lanczos algorithm (Lanczos 1950). Finally, the third step parallelizes the *k*-means algorithm. The parallelization of *k*-means is divided into two fundamental steps. The Map function calculates the nearest centroid for each point. Then, the combiner function partially merges samples with the same centroid and sends the result to the Reduce function in the form of pairs, where the key is the centroid and the value is the list of points having the same centroid. The Reduce function collects the points with the same centroid and updates the centroids values by calculating the average of each set of points assigned to the centroids. The process reiterates until stable centroids are obtained. The experiments were performed using the classic dataset using several properties (correctness validation, a test of speed-up ratio, analysis of scalability). The results show that the parallel spectral algorithm is efficient for processing large datasets.

In Sun et al. (2014), the authors developed a parallel method combining the information bottleneck (IB) theory with centroid-based clustering. Their main contributions include the use of the IB theory-based hierarchy clustering to determine the centroid of each Map computational node, in addition to the use of an objective method in order to determine the

number of clusters. The parallel IB theory is based on MapReduce. The MapReduce job is designed with multiple Map tasks and a single Reduce task. In this method, the data is divided into partitions, and then each partition is treated independently and in parallel by a Map computational node. In each Map computational node, the IB theory-based clustering method is applied to each partition in order to obtain the sub-centroid. All sub-centroids are gathered in the Reduce task in order to create a new dataset. Then, the IB theory-based clustering method is once again applied to the new dataset in order to generate the initial centroid of the global dataset. Once the initial center is calculated, the parallel centroid clustering based on an iterative MapReduce model, called Twister (Ekanayake et al. 2010), is applied. It is designed with multiple Map tasks and a single Reduce task, which returns its output to the Map tasks iteratively. The parallel centroid clustering processes as follows. First, the initial sample dataset is partitioned and the initial centroids obtained previously are mapped to each computational node. In each Map computational node, the sub-centroids are recalculated with centroid-based clustering method. All sub-centroids are gathered in the Reduce computational node and the global centroid is updated. Then, the new centroids are sent to the main computational node to be used in the next iteration. The process stops when a certain difference, which is measured with Kull-back divergence, is less than the fixed threshold value. In order to visualize the clustering results, the multidimensional scaling (MDS) has been applied, as a dimension reduction method, on the DNA data used in the experiments. The results show that the developed method is better than a typical parallel *k*-means implementation.

The *Best of both Worlds* (BoW) approach, which is proposed in Cordeiro et al. (2011), combines two parallel clustering methods: the *Parallel Clustering* (ParC) and the *Sample-and-Ignore* (SnI), in order to reduce the I/O and network costs. The ParC method is executed in five phases using three different strategies for partitioning data. In the first phase, the data is distributed to the mappers from the distributed file system. Each mapper computes the key value of the received data element. In the next phase, each reducer processes the elements with the same key. And then, it normalizes them and runs the plugged-in clustering algorithm in order to obtain the β-clusters from each reducer. In the following phase, the algorithm finds a pair composed of the reducer description as key and the cluster description as value. The last phase is executed serially by putting together the β-clusters pairs which overlap in the space. The SnI method consists of executing the ParC algorithm after applying a pre-processing step. This step starts by sampling the datasets and find initial clusters in order to cluster only the unclassified elements using the ParC algorithm. The BoW method chooses the least expensive clustering strategy in terms of time and other parameters using a cost-based optimization. The proposed approach has been developed using Hadoop MapReduce framework, and tested with up to 1024 cores in parallel, on real-life and synthetic datasets. The results show that the BoW approach is scalable, and gives approximately the same results compared to the serial clustering algorithms.

3.1.2 Clustering algorithms using Spark

A recent algorithm, called Parallel Kernel Kmeans, has been introduced in Tsapanos et al. (2016). The proposed method, which expands a previous work called the Trimmed Kernel *k*-Means (Tsapanos et al. 2015), handles clustering of large datasets using Spark framework. The proposed method begins by sub-sampling the data to perform the clustering of large datasets. Then, it calculates the Kernel matrix iteratively using MapReduce and writes it to the disk for future use. Thereafter, it reads the Kernel matrix from the disk in

order to compute the Kernel matrix trimming. The calculation of this matrix consists of two main steps. The first step finds the winning cardinality of each node and trimmed rows accordingly using the Map function, while the second step sums up in the same vector the votes of the cardinality of each cluster using the Reduce function. The Map and Reduce functions are used again to remove the winning votes. This process reiterates until finding the cardinality of all nodes. The next step applies the Kernel *k*-means to adjacency lists given by the calculated Kernel matrix trimming. The process of Kernel *k*-means proceeds as follows: first, it associates each sample to a cluster randomly and then calculates the partial sum of the entries retrieved from the adjacency list by a mapping function. The reduce function calculates total sums from every cluster. Thereafter, the map function performs the distance computations in order to define the new cluster assignment for each node. The last distributed operation is the nearest neighbour assignment. The performance of the proposed method was evaluated using the Youtube Faces dataset, and compared with the approximate Kernel *k*-means and the Trimmed Kernel *k*-means algorithms. The clustering results show that the Parallel Kernel *k*-means algorithm is more efficient than the Trimmed Kernel *k*-means algorithm, and yields results close to those given by the approximate Kernel *k*-means algorithm.

The work suggested in Mallios et al. (2016) is a framework for clustering and classification large amount of data using Spark.The *k*-means and ID3 algorithms were studied and implemented using the proposed framework. The proposed framework is a parallel execution of multiple round-trips performed by the master node and several workers. Each round-trip consists of three main phases: Local Pre-processing, Global Distributed Processing, Local Post-processing. In the first phase, the master node requests to the workers to compute intermediate results from the stored data as part of the Map function. In the second phase, the workers swap the results obtained, and then collect the global information. In the last phase, each worker completes its own computations based on the intermediate results obtained from the previous phase. After having completed these phases, the master node runs a new round-trip. The data, which is swapped between the master node and the workers or between workers, operates in three modes: No-restriction mode, Partially Restricted Data Exchange Mode and Strict Restricted Data Exchange Mode. Additionally, the data processed by the workers can be executed differently according to the algorithm used in the proposed framework. The results show that the algorithms tested in the proposed framework were efficient in terms of time and scalability. Moreover, the *k*-means algorithm, which is implemented using the proposed framework, exceeds 31 % the quality of results provided by the *k*-means algorithm using Spark.

There are several other works related to the implementation of *k*-means using Spark, such as Wang et al. (2016), Zayani et al. (2016) and Sinha and Jana (2016).

In Wang et al. (2016), the authors conceived a parallel *k*-means algorithm using Spark. The proposed design can select the appropriate variant of the *k*-means algorithm and the distance function to use. The algorithm reads data from the HDFS and writes it in RDDs blocks using two different strategies. In the first strategy, each data instance is loaded directly into the RDD block in the form of a set of dense vectors, while in the second strategy it is represented by a set of sparse vectors. The next step computes the distance between each two sparse vectors in parallel. Finally, the last step performs two different methods for updating the centroids according to the clustering type used (crisp clustering or fuzzy clustering). For the crisp clustering, the assigning and the distance calculation steps are performed as part of the Map function, whereas the centroids updating step is performed as part of the Reduce function. This latter consists of

collecting the instances with the same centroid, and then compute their average value. For the fuzzy clustering, the centroids updating step is performed using a predefined equation.

Another work related to the *k*-means algorithm is proposed by Zayani et al. (2016), which is called the Parallel Overlapping *k*-means method (POKM). It was designed to perform non-disjoint partitioning of large-scale data. The proposed work is based on a geometrical method, called Overlapping *k*-means (OKM), which consists of two steps: assigning each instance to multiple clusters and updating centroids. These steps are repeated until reaching the convergence criterion. Following the same manner, the Spark-based version is based on two steps: the data assignment and prototypes calculation steps. Ones the input data is partitioned, the first step would be to apply a Map function for each block, which assigns the instances to multiple clusters, in order to parallelize the assigning procedure locally. Then, another function operates on the global clusters, which are produced previously, by sorting the prototypes of clusters according to their proximity to the processed instance. This process assigns the clusters from the nearest to the farthest as long as a predefined criterion decreases. The Prototypes calculation step is performed by Map and Reduce tasks. The Map task performs local computations for each cluster and sends as output a set of key-value pairs, where the key is the index of the prototype cluster and value is a couple of computed parameters. The Reduce task updates the prototype clusters and returns the final ones.

In the same context, Sinha and Jana (2016) presented a method based on classical *k*-means algorithm to process a big data streams using the Spark framework. The proposed algorithm consists of four main processes. The first process consists of generating initial clusters using a large value of k. The second process applies the *k*-means algorithm to the *k* clusters. The next process consists of merging the centroids that are spaced from each other by less than a certain threshold value. The last process returns the final clusters by merging the previously obtained clusters. This algorithm is characterized by its free over-resolution.

The reviewed methods demonstrated their efficiency in terms of scalability and quality of clustering large data.

The work introduced in Bharill et al. (2016) is a scalable random sampling and an iterative optimization fuzzy c-means algorithm. The proposed method, which is called SRSIO-FCM, divides randomly the dataset into many subsets. Then, it generates the centers of clusters randomly in order to cluster the first subset. The cluster centers and the membership information of the first subset are calculated, and then they are given as input to cluster the second subset. In turn, the cluster centers and membership information of the second subset are calculated. After the first two iterations that present a particular case, the membership information of all the processed subsets are combined in order to compute the new cluster centers, which will be given as input to cluster the next subset. The following subsets are clustered in the same manner. The authors proposed a parallel implementation of the LFCM algorithm (Havens et al. 2012) using Spark, which they called the SLFCM algorithm (Scalable Literal Fuzzy c-Means), in order to handle the parallel computation of membership information and centers of clusters. It performs Map and ReduceByKey operations, where the Map phase is responsible for the membership degree calculation of a data point with respect to each center, and the ReduceByKey phase is responsible for updating the values of the center of cluster based on the output of the Map phase. The performance of the SRSIO-FCM algorithm was compared with the Spark based-implementations of two well-known fuzzy clustering algorithms. The experiments on some big datasets demonstrated that the proposed method achieves almost the same quality of clustering results in less time.

The work proposed in Gouineau et al. (2016) is a distributed density clustering algorithm to analyze very large datasets, called Patchwork. It was designed using the MapReduce model to parallelize the calculations and implemented with Apache Spark. The Patchwork algorithm is a part of density clustering algorithms that are known for their linear computational complexity and near-linear horizontal scalability. It consists of two fundamental steps. The first step consists of dividing the multidimensional feature space into a grid in order to determine dense regions and then finding the ID of the cell (hypercube in D-dimensional feature space) for each point using the Map function. After that, the Reduce function collects the tuples with the same cell ID and constructs collections of cells with their density. The second step consists of creating clusters using the collections of cells and sorting them by decreasing the density until all cells are processed. An optional filter is used to help to find clusters with enough cells. The proposed algorithm was compared with Spark MLLib *k*-means and Spark DBSCAN using four Synthetic datasets. It is efficient and yields 40× better results than those produced by Spark MLLib *k*-means.

In Liu et al. (2017), the authors presented a parallel implementation of the density peaks clustering algorithm (Rodriguez and Laio 2014) using the Spark's API for graphs computation, called GraphX. The authors noticed that the original method needs to calculate the distances between all pairs of data points which results in high computational cost. In order to overcome this issue, they have proposed a parallel version of the density peaks clustering algorithm using the Spark RDD model. The process starts by initializing the thresholds values of the local density and the distance from points of higher density and then generating a graph. The graph construction requires first importing separately vertex and edge data stored in the HDFS to vertex RDD and edge RDD, then computing the distance and updating its value in each edge, and finally combining vertex RDD and edge RDD to form a Graph. The next step consists of computing the truncated distance, the local density, and then the distance from points of higher density. The last step performs the clustering of data points by selecting cluster centers, isolated points, and then performs the classification based on the thresholds fixed previously. The experimental results demonstrated that the proposed implementation can be up to 10× faster than the density peaks clustering algorithm when implemented using MapReduce.

Remaining within the context of the density-based clustering methods, several studies have been conducted on the DBSCAN algorithm in order to propose parallel implementations based on Spark, such as Luo et al. (2016), Han et al. (2016) and Cordova and Moh (2015).

The S_DBSCAN algorithm, introduced in Luo et al. (2016), starts by partitioning the raw data based on random samples. It produces partial clustering results by performing local DBSCAN algorithm in parallel. At this stage, the Map task generates partial clusters, while the ReduceByKey task saves each partial cluster as a new RDD to the HDFS, then computes the centroid for each partial cluster. Finally, it merges the independent clustering results obtained in the previous steps based on the centroid.

The proposed implementation in Han et al. (2016) starts by reading data from the HDFS and then generates RDDs in order to transform them into data points. In the next step, multiple executors build independently partial clusters and sent them back to the driver. When all the partial clusters are collected, the merging process starts identifying the clusters that need to be merged based on a new concept introduced by the authors, called SEEDs. SEEDs are points that do not belong to the current partition and they are placed by executors to serve as markers in order to identify master partial clusters and then merge them. The proposed method is distinguished by the use of a Java-based kd-tree implementation in order to optimize the complexity of the task of searching for the neighbours of points.

The RDD-DBSCAN algorithm, introduced in Cordova and Moh (2015), consists of five steps. This algorithm starts the same way as the Apache Spark-based implementations of DBSCAN described above. It divides data and then performs local DBSCAN algorithm on each partition. After that, it identifies clusters which span multiple partitions and generates global cluster identifiers. Finally, all points are relabelled using the newly-found, globally-unique identifiers.

The three methods were assessed based on their accuracy and speed up. They have proven their efficiency in processing large data. The algorithm, proposed in Han et al. (2016), can achieve a performance 16× faster than the MapReduce implementation of the DBSCAN algorithm. The experiments of the RDD-DBSCAN algorithm demonstrated that the communication costs do not significantly affect its performance.

3.1.3 Clustering algorithms using peer-to-peer networks

Recently, Azimi et al. (2017) introduced a new gossip-based distributed clustering algorithm for peer-to-peer unstructured networks (GBDC-P2P). The GBDC-P2P is based on K-medoids and *k*-means algorithms for extracting the representative data and discovering the final clustering results, in addition to the CYCLON algorithm (Voulgaris et al. 2005) to ensure the interactions between peers. It starts by selecting M representatives in each peer among its internal data. In the next step, which concerns the gossip-based interactions, each peer sends the representatives to the neighbour peers. This process of interactions between peers is based on the CYCLON algorithm. Then, the summarization step summarizes the external data of peers once their memory is full. This step is performed only when necessary. The property of the data summarization is provided by the K-medoids algorithm. The adaptation of the GBDC-P2P algorithm to dynamic network conditions is ensured using an age variable for each external data of peers. At each round of gossip-based operations, all peers increment the age variables of their external data. This mechanism allows replacing the old external data of peers. Finally, an improved version of the *k*-means algorithm, called Persistent *k*-means, is proposed and it is performed in each peer in order to calculate the final clustering results. The proposed algorithm was compared to other methods in the literature in order to demonstrate its efficiency. The clustering results obtained by the GBDC-P2P algorithm are very close to those obtained by the centralized *k*-means algorithm.

In Gehweiler and Meyerhenke (2010), a distributed heuristic, called DIDIC, is proposed for clustering a virtual P2P supercomputer. It consists of three phases, namely, the establishment of the initial situation, the elimination global knowledge with suitable diffusive processes, and the determination clustering. In the first phase, if no initial configuration is provided, then a random configuration is considered. In the second phase, the diffusive clustering process is carried out. Two diffusion systems are used per cluster to represent the same load colour. The primary diffusion system exploits the property of diffusion and random walks in order to identify dense graph regions, while, the secondary system sends a load of the system to nodes belonging to cluster in order to accelerate the forming of large cluster-connected components. The last phase aims to flood areas by new clusters when the diffusion process shows a really strong desire for this by integrating the isolated nodes. The authors adopted the bulk synchronous parallel (BSP) model (Valiant 1990), which provides an abstract view of technical structure and communication features of the hardware, in order to ensure the parallelism of the proposed method. The clustering results of the DIDIC heuristic were compared to

results obtained with the MCL algorithm (Enright et al. 2002), which tend to be close and sometimes slightly better in favour of the proposed method.

In Yıldırım and Özdoğan (2011), the authors proposed a MPI-based implementation of a grid-based clustering algorithm based on the wavelet transform, which is known as the WaveCluster algorithm (Sheikholeslami et al. 2000). The WaveCluster algorithm transforms the original feature space by adopting the wavelet transform, thus forming a new space in which the dense regions must be sought. As a result, it produces sets of clusters at different resolutions and scales. In order to ensure the parallelism the WaveCluster algorithm, the authors adopted the replicated approach (Skillicorn 1999), which consists in splitting the data set over processors that execute nearly identical code segments of the algorithm. Therefore, processors need to exchange their local results and to check the correctness. Furthermore, they adopted the master/slave model with the Single Process, Multiple Data (SPMD) technique on a distributed-memory multiprocessor system. The MPI was used to manage the communication requirements between master and slave nodes. Experiments were conducted on a PC cluster of 8 compute nodes with 32 processors at total and having fast Ethernets as underlying communication hardware. The clustering results demonstrated that the proposed method yields a high speed up and scales linearly.

p-PIC (Yan et al. 2013) is a MPI-based implementation of the power iteration clustering (PIC) algorithm (Lin and Cohen 2010). The original algorithm produces clustering results by embedding data points in a low-dimensional subspace derived from the similarity matrix. It consists of three main operations: the similarity matrix calculation and normalization, the iterative matrix-vector multiplication, and the clustering. The authors of the p-PIC algorithm noticed that the original method depends on the memory resources because it stores data and its associated similarity matrix in memory. Therefore, they suggested enhancing the two first operations involved in this method by exploring the parallelization strategies in order to reduce the computation and communication costs. Indeed, each processor stores only two cases in memory at a time, in addition to this, the similarity matrix is split according to the strategy of row-wise block stripped matrix-vector multiplication and stored at the processors executed in parallel. The experimental results on a local cluster and Amazon EC2 cloud show that the p-PIC algorithm reaches almost a linear speed-up on all tested datasets.

A MPI-based implementation of the DBSCAN algorithm is presented in Savvas and Tselios (2016). The authors designed an approach based on three phases, namely, the splitting, the DBSCAN execution, and the final clusters forming. Furthermore, they adopted the Single Instruction Multiple Data (SIMD) technique, which allows executing the same task by each computational node of the system on different data. The master node is responsible for splitting and assigning the data to the worker nodes, while the worker nodes perform independently the DBSCAN algorithm. Consequently, each worker node produces its own local clusters in the form of a collection of pairs, wherein the centroids and the radius of the clusters are saved. Then, each worker node sends its collection to its neighbor node in order to explore possible aggregations of clusters. This is achieved by computing the intersections of circles then examining these intersections. In the case where the circles do not intersect then there is no action since these form different clusters. This process is repeated until all information become on the final worker node, which will emit the clusters as a final result to the master node. The experiments were conducted on 33 computational nodes and they demonstrated that the proposed approach reduces the time complexity and yields identical results compared to the original sequential DBSCAN.

3.2 Vertical scaling platforms-based clustering algorithms

In this section, we present the recent clustering methods based on vertical scaling platforms, namely the GPU, Multi-core CPU, and FPGA platforms.

3.2.1 Clustering algorithms using multi-core CPU

Authors of Hadian and Shahrivari (2014) suggested a parallel variant of the *k*-means algorithm using multiple CPU cores of a single machine. The proposed algorithm divides the datasets into chunks and then distributes them to the processing units. The processing units perform the clustering on the chunks of data in parallel. The proposed algorithm consists of two thread sets: Master thread and chunk-clustering threads. The master thread is responsible for reading datasets and organizing them into chunks that have a predefined size. After that, the master thread sends the chunks using the concept of the queue. The next step consists of performing the clustering by multiple threads using *k*-means++ and then storing the centroids obtained from each chunk in the global list of centroids. Once the clustering of all chunks is completed, the global list of centroids is loaded by the master thread in order to run the next step, which consists of clustering the centroids using *k*-means++ to produce the final centroids. The proposed algorithm was evaluated using a 12-cores machine, and compared with the *k*-means, *k*-means++, and stream-based algorithms. The results show that this algorithm can achieve a near-linear scalability, and yields the same quality of results as *k*-means++. In addition, it runs much faster than stream-based algorithms.

An adaptation of an uncertain data clustering algorithm (Erdem and Gündem 2014), called fast density-based spatial clustering of applications with noise (FDBSCAN) (Kriegel and Pfeifle 2005), to multi-core platform. The idea behind the original FDBSCAN algorithm is to integrate the fuzzy distance functions directly into the traditional DBSCAN algorithm in order to measure the similarity between fuzzy objects. The adaptation of this clustering algorithm to the multi-core platform resulted in the so-called M-FDBSCAN, which has demonstrated a significant acceleration of processing. The newly introduced algorithm operates in three steps. First, it splits the 2-dimensional fuzzy data object dataset into subsets according to the number of available cores in the multi-core system. Then, it applies the FDBSCAN algorithm to each subset concurrently which leads to the determination of the final cluster regions partially. Finally, the last step in this algorithm consists of merging the subsets pairs to get the final cluster regions. The M-FDBSCAN algorithm is implemented using OpenMP in C to achieve its parallelism. Experiments conducted on synthetic datasets demonstrated that the proposed algorithm scales linearly when increasing the number of cores and it outperforms the FDBSCAN algorithm when processing huge amounts of data.

Earlier, the work published in Kraus and Kestler (2010) suggests a parallel implementations of the *k*-means and K-modes algorithms using a multi-core platform with transactional memory in order to process a large amount of data. The *k*-means and *k*-modes algorithms were implemented as follows: The first step consists of distributing the initial datasets to the cores. Then, each data point is assigned to the nearest centroid. The next step consists of updating the centroids by *k* data threads. The operations assigning data and updating centroids are performed in parallel using a software transactional memory (STM), named deferred-update STM. Each data point is processed by running simultaneously the

threads. The data must be verified before sending it to the shared memory. If any change has been made to the data, the transaction is sent. Otherwise, the transaction is rejected. The parallel multi-core *k*-means algorithm (McKmeans) was compared to the single-core *k*-means algorithm implemented in R, by simulating artificial data, using a dual quad-core computer. Then, it was compared to the network-based ParaKMeans tested on the web with the same datasets. These comparisons show the high performance of the proposed algorithm. Besides, the speed-up achieved by the proposed algorithm is 10× better compared to single-core implementations.

3.2.2 Clustering algorithms using graphics processing unit

Three parallel implementations of the *k*-means algorithm are suggested in Cuomo et al. (2017) for the purposes of reducing the execution time using GPUs and CUDA. These implementations follow a hybrid approach, in which the *k*-means algorithm is performed partly in host and partly in device. Data transfer between host and device is efficiently carried out using coalescing memory accesses and pitch for matrices memory allocation. The proposed parallel approach calls the device at each iteration. This requires fixing some parameters and checking the dimension of the matrix of data points. Some dimensions need the data points to be divided in chunks and then process them one by one. Then, the host calculates the new centroids using the information of the new indices produced by the device, and finally it checks the convergence. The authors observed that the *k*-means algorithm needs three data structures to store the set of data points, the centroids, and the indices that indicate the membership of each data point. In order to address this observation they proposed a lighter data structure that can reduce the data transferring time. In their first implementation, they proposed to use a matrix to store the data and to update the results for the labelling stage on the CPU, while in the second implementation; they used two different data structures to store them. The third implementation, which is considered the most refined solution, adopts a parallel technique to compute the squared Euclidean distance. This refined implementation achieves a speed-up reaching 88× faster than the CPU version.

Authors of Huang et al. (2015) proposed PaStream, a parallel algorithm for clustering data streams based on NVIDIA GPUs. This single-pass algorithm discovers clusters with arbitrary shapes and detects outliers. The PaStream algorithm is implemented following a framework for clustering fast evolving data streams, called CluStream (Aggarwal et al. 2003). The PaStream algorithm requires an initialization step, in which some micro-clusters are generated using the *k*-means algorithm. Then, it involves two phases: an on-line phase, in which data points are gathered into micro-clusters, and an off-line phase, in which micro-clusters are aggregated into macro-clusters. In the on-line phase, for each data point, three cases must be considered. In the first case, an existing micro-cluster absorbs the data point. In the second case, the data point forms a new micro-cluster, while in the third case, two closest micro-clusters merge to form one. This decision is made according to the absorption rate. Furthermore, the authors have defined a merging factor, which determines when the close micro-clusters should be merged in order to optimize memory and time. In the off-line phase, the Euclidean distance between all centers is calculated, then, the decision graph is plotted based on the local distance and the local density for each micro-cluster. It should be noted that this algorithm operates on data groups and not on data points at its arrival. This helps reduce the time required to transfer data from CPU

to GPU. The experiments on real and synthetic datasets show that the PaStream algorithm outperforms its version based on CPU in terms of time and quality of the produced result.

In Zhang et al. (2010) a parallel implementation of a nature-inspired algorithm for document clustering is introduced. The original method is called the flocking-based document clustering algorithm (Cui et al. 2006). In order to reduce the execution time and the complexity of this algorithm, the authors exploited the computational power of Beowulf-like clusters equipped with GPUs. In a typical flocking model, the behaviour of an individual is based only on its neighbour flock mates within a certain range. This behaviour is described by three rules: separation, alignment, and cohesion. In order to adapt this model to the document clustering problem, the document is considered as an individual that participates in the flocking formation. Then, the three rules are combined for similar neighbour documents, while only the separation rule is applied for non-similar neighbour documents. The authors developed a special GPU cluster programming model to implement the proposed method. It consists of a distributed object interface to unify CUDA memory management and explicit message passing routines, a mechanism to spawn a flexible number of host threads for parallelization that may exceed the number of GPUs in the system, in addition to an interface for advanced users to control thread scheduling in clusters. The experiments show that the GPU-based implementation scales up over one million documents processed simultaneously in a sixteen-node moderate GPU cluster. Moreover, the proposed method can reach up to 50× speed-up compared to its CPU-based implementation.

Async-EM (Altinigneli et al. 2013) is a parallel variant of the Expectation–Maximization (EM) algorithm based on the GPU platform. Their main contributions concern the synchronization of cores and the organization of memory access. Indeed, the authors noticed that several updates of the global cluster representatives cause an inefficient use memory bandwidth and synchronization overhead. Therefore, to avoid these issues, they proposed to update the global cluster representatives only when a certain number of membership changes have occurred. This is achieved through the idea of the asynchronous model updates combined with an efficient technique, called the model of consolidation. The model of consolidation is responsible for merging the different sets of cluster representatives when the local model updates are exchanged, and it exploits the special characteristics of the memory hierarchy of modern GPUs. The experiments on real and synthetic datasets show that the Async-EM. This method outperforms the incremental-EM and the Batch-EM algorithms in terms of the convergence, the modelling error, and the execution time performances. A comparison between the GPU and the CPU performances of the proposed approach has also shown its effectiveness.

The work proposed in Melo et al. (2016) is part of the density-based clustering. It focuses on the parallelization of the OPTICS algorithm using GPU and based on data indexing strategy. The OPTICS algorithm creates an augmented ordering of the dataset representing its density-based clustering structure (Ankerst et al. 1999). The proposed parallel implementation of the OPTICS algorithm consists of two phases: the graph construction and the OPTICS process. The proposed approach adopts a data representation in a graph form following the METIS data structure, which is presented in Karypis and Kumar (1998). And it uses three vectors to store the vertices, the nodes in the adjacency list, and the distance. The process consists of four main steps, respectively, the vertices degree calculation, the adjacency index calculation, the adjacency lists assembly and sorting. These steps, in addition to the construction of data structure and the storage at the end of the adjacency lists of nodes, are each processed in parallel using GPU. The experiments show that the proposed approach reduces significantly the complexity of the OPTICS algorithm and makes it faster than the serial CPU-based version.

In the same context, authors of Deng et al. (2015) have proposed a new trajectory clustering approach based on the POPTICS algorithm (Patwary et al. 2013). In order to adapt the POPTICS algorithm to the trajectory data, the authors applied a spatio-temporal distance to measure the similarity between trajectories and an indexing approach based on the STR-tree structure to their proposed algorithm. This latter, which is called Tra-POPTICS, is based on shared memory and involves three main steps. In the first step, each CPU thread processes a local disjointed subset of trajectory data. It finds out the neighbours of each trajectory. Then, it computes the core distance of each trajectory and then local Prim's minimum spanning tree (MST). The next step generates the global MST, and finally, the last step extracts the clusters from the global MST. As part of this study, a parallel version of the Tra-POPTOCS algorithm, called G-Tra-POPTICS has been designed using the Hyper-Q feature of Kelper GPU and massive GPU threads. The experiments on real trajectory dataset demonstrated that the Tra-POPTICS algorithm and its parallel version have reached the quality of clustering result produced by a variant of OPTICS algorithm called T-OPTICS (Nanni and Pedreschi 2006). In addition, the GPU-based version outperforms the Tra-POPTICS algorithm regarding the computational time.

Another work related to the density-based clustering is presented in Andrade et al. (2013). It is a GPU-based implementation of the DBSCAN algorithm, called G-DBSCAN, which uses a simple graph-based data indexing technique. The proposed approach consists of two main steps, namely, the graph construction and the identification of clusters using the breadth-first search algorithm (Harish and Narayanan 2007). The First step aims to construct a graph in order to represent the data. Indeed, each object in the dataset is represented as a node in the graph, and when the similarity measure between two objects is less than a threshold, which is given as an input parameter, an edge is added between them. Therefore, this first step performs the calculation of vertices degree, the calculation of the adjacency lists indices, and finally the assembly of adjacency lists. The second step aims to identify the clusters by traversing the graph created in the previous step using the breadth-first search algorithm. According to the experiments, the G-DBSCAN algorithm is 100× faster than the serial implementation of the DBSCAN algorithm.

CUDA-MCL (Bustamam et al. 2012) is a parallel variant of the Markov clustering algorithm (MCL) (Van Dongen 2008) applied to the protein-protein interaction networks. The MCL algorithm discovers cluster structure in graphs by adopting the concept of random walks. The proposed parallel implementation is based on GPU programming via CUDA. The MCL algorithm is based on two algebraic operations on the Markov matrix, namely, the expansion and the inflation. Therefore, the performance of this algorithm depends on the size of the Markov matrix. The main contributions of the authors include the enhancement of the performance of the original MCL by implementing parallel tasks for the expansion and inflation operations, in addition to the storage optimization using sparse matrix data structures. The proposed algorithm involves three parallel threads CUDA kernels. A kernel to compute the parallel MCL expansion processes, a second kernel to compute the parallel MCL inflation processes, and the last kernel to compute the parallel local and global chaos. Comparing to the original MCL algorithm, which is based on CPU, the proposed approach is faster on large datasets.

The Calculation-On-Demand CAST with GPU (COD-CAST-GPU) algorithm (Lin et al. 2014) is a parallel design of the Clustering Affinity Search Technique (CAST) (Ben-Dor et al. 1999). As its name suggests, it is based on the GPU platform and the individual memory of graphics card. The CAST algorithm requires as input the similarity matrix and the affinity threshold. Therefore, the size of such matrix is a critical point which causes the storage problems. The idea behind the proposed algorithm is to calculate the similarity

between data nodes only when it is needed in order to avoid the prior calculation of the similarity matrix. In addition, in order to accelerate the CAST algorithm and gain in terms of performance, the authors exploited the capabilities of the GPU. The proposed algorithm starts by selecting a random node as a new cluster. Then, it uses two operations: *ADD* and *REMOVE* to form this new cluster. After each movement, the affinity values are updated in parallel, and this is where the GPU comes in. The technique adopted by the authors showed remarkable improvements in terms of performance compared to the original algorithm.

3.2.3 Clustering algorithms using field programmable gate arrays

Authors of Scicluna and Bouganis (2015) proposed to accelerate the well-known DBSCAN algorithm using an implementation based on FPGA in order to achieve a high-performance clustering. The proposed FPGA-based implementation is partially parallelized because the authors considered that the step aiming to obtain the extended neighbourhood of points for a cluster consumes the longest portion of the execution time. And since this step has no data dependencies, it is suitable to be performed in parallel. The proposed method is a fully configurable IP core, thus it contains many parameters to be adjusted, such as the size and dimensions of the input data, internal precision, pipeline depths, and the level of parallelism. Performance evaluation based on 2D point clustering shows that the execution time of FPGA-based implementation is not impacted by the increase in the number of dimensions. The proposed method can reach mean speed-ups of 31× in real-life datasets and 202× in synthetic datasets when compared to the state-of-the-art methods.

The study introduced in Shi et al. (2014) is also related to the DBSCAN algorithm. It is a FPGA-based implementation of this algorithm while adopting task-level and data-level parallelism. Their main contributions include a design of a data reuse pipeline structure in order to overcome the problem of the extra memory access caused by the data dependencies in the parallel algorithm. In addition to a robust collision check mechanism in order to prevent wrong clustering results in some special conditions. The experiments on synthetic datasets show that proposed parallel architecture can reach up to 86× speed-up compared to a software implementation on the general-purpose processor and up to 2.9× compared to a software implementation on the graphic processor.

SAKMA (Jia et al. 2015) is a FPGA-based implementation of the *k*-means algorithm. This architecture accelerates the whole *k*-means algorithm through certain approaches such as the pipeline, the tile technique, the duplication parallelism, and the hardware adder tree structures. The SAKMA architecture contains a processing system part in software and a processing logic part in hardware. The FPGA-based implementation of the *k*-means algorithm is carried out by the IP Core. The frequent off-chip memory access is a common problem that the authors resolved using the tiled technique. This technique aims to divide the large volume of memory blocks into small tiles that can be buffered on-chip. The experimental results on real biological and synthetic datasets show that the proposed method can reach a speed-up at 20.5× with the affordable hardware cost when compared with other state-of-the-art methods.

In the same context, authors of Winterstein et al. (2013) proposed to accelerate the *k*-means algorithm using the FPGA and a binary tree data structure. Indeed, the authors adopted the filtering algorithm (Kanungo et al. 2002) which intends to reduce the computational load using a kd-tree as the main data structure. This algorithm consists of constructing a tree from the set of points, then it traverses the tree iteratively and updates the centers. The main challenge is how to pipeline and to parallelize the

kd-tree processing using multiple banks of distributed on-chip memory. Besides, the on-chip dynamic memory allocation is used in order to efficiently utilize the memory resources by allowing the allocation of the average amount of memory required during runtime. The proposed method needs 5× fewer computational FPGA resources compared to the conventional *k*-means algorithm implemented in parallel for the same throughput constraint.

The work proposed in An et al. (2012) is part of the clustering-based prototype learning algorithms. It is a *k*-means-based multi-prototype learning system strengthened by a FPGA-based implementation coprocessor for the nearest Euclidean distance searching. This technique allows overcoming the high computational cost of the nearest neighbour searching implementations for the *k*-means clustering algorithm and the one nearest neighbour (1-NN) classification algorithm. The aim of the proposed system is to construct prototypes using the *k*-means algorithm iteratively until its convergence to stable centroids, then, the prototypes are used in order to recognize the test samples by applying the 1-NN classifier to search for the nearest Euclidean distance among these prototypes. Finally, the *k* prototypes with the best training result are selected to represent the final recognition. The experiments on a handwritten digits dataset show that the proposed learning system can reach an accuracy rate of 97.91% with 930 prototypes.

Another contribution to the *k*-means algorithm is proposed in Hussain et al. (2011) which aims to design a FPGA-based implementation of the *k*-means algorithm for the Micro-array datasets, which are known for their large sizes. The proposed approach requires a careful analysis of samples of Micro-array data in order to fix the values of some critical parameters, such as data size, dynamic range, precision, and memory requirement. The *k*-means algorithm is implemented using three main blocks. The first block is responsible for the calculation of distances, the second block is responsible for the assigning data points, and the third block is responsible for recalculating the centers. Respecting this order, each block executes its operations in parallel and as soon as it finishes it activates the following block. This process reiterates until the convergence. The experiments on a sample of a real-life dataset show that the proposed method can reach a speed-up up to 51.7× compared to a software model and 206.8× more energy efficient than the CPU implementation.

Authors of Sotiropoulou et al. (2014) proposed 2D-clustering implementation based on multi-core FPGA for real-time image processing. The proposed clustering implementation operates on zero-suppressed data and it consists of a pipeline of three modules: the hit decoder module, the grid clustering module, and the centroid calculation module. The first module is a preprocessing step which transforms the incoming data to a recognized form. The second module identifies the clusters using an innovative moving window technique to reduce the FPGA resources required for this process. Finally, the third module is a post-processing step which performs the data reduction process. In other words, the cluster data are replaced with a single set of centroid coordinates. In order to parallelize this implementation, the authors proposed to instantiate multiple grid clustering modules which work independently on data from separate pixel modules. This is achieved through two logic modules, the parallel data distributor module, and the data merger module. The experiments concerned the 2D-clustering single flow implementation and its parallel version up to 16 clustering engines. A comparison with a previous version showed that the proposed parallel implementation uses 64× fewer logic resources.

4 Comparison of the parallel clustering algorithms

The research carried out in the context of this survey has highlighted several noteworthy observations. The first point concerns the fact that the majority of the parallel clustering algorithms, which are proposed recently, focus on some well-known algorithms, such as the *k*-means (Hartigan and Wong 1979), DBSCAN (Ester et al. 1996), and OPTICS (Ankerst et al. 1999) algorithms. The nature of these algorithms makes them suitable for parallelism in different designs and based on various platforms to handle Big Data.

4.1 Experimental setup

Table 1 provides a general idea of the performance of the *k*-means algorithm when implemented using different Big Data platforms. Each parallel *k*-means version is firstly classified according to the Big Data platform used to attain the parallelism of the original algorithm. Then, we present the initial parameters, the type of data processed and the criteria of clustering validation, namely, the quality of the obtained results, the running time consumed, and the speed-up achieved. The quality of the results depends on the data type processed and therefore the measures used differ in some cases. Finally, we present criteria related to the ability of the proposed algorithm to handle Big Data through the volume, the velocity, and the variety of the processed data.

4.2 Parallel *k*-means algorithms

Although *k*-means is known to be popular and easy to implement, it suffers from some weaknesses such as the determination of the suitable number of clusters *k*, in addition to the scalability issues when treating sparse values. *k*-means is an iterative algorithm which involves three main steps: initialization, clusters assignments, and centroids updates. There are several designs that may be put forward to perform the clusters assignments and centroids updates steps in parallel. From Table 1, it can be observed that the MapReduce-based implementation of the *k*-means algorithm can process up 4 billion data points in only a few minutes.

4.3 Parallel DBSCAN algorithms

DBSCAN is as important as *k*-means. It aims to discover the clusters and the noise in a spatial database requiring two parameters: Eps and MinPts. The Eps parameter denotes the maximum radius of the neighbourhood from a point, while the MinPts parameter denotes the minimum number of points required to form a dense region. Once these input parameters are set, DBSCAN performs clustering in a two-step approach. First, it starts by selecting an arbitrary point from the dataset that fulfils the core point condition as a seed. Then, it groups all points that are density-reachable from the selected seed including itself, thus obtaining a cluster.

DBSCAN offers several advantages including finding clusters of arbitrary shapes and to handling automatically outliers. And unlike *k*-means, it does not need to set the number of clusters. However, the estimation of its input parameters is a complex and critical task. This algorithm suffers also from the scalability issues like the rest of the traditional

Table 1 Comparison of the recent parallel clustering algorithms of *k*-means

Platforms	Algorithms	Clustering criteria					Big Data criteria		
		Input parameters	Datasets type	Quality	Running time	Speed-up	Volume	Velocity	Variety
Horizontal scaling									
Map-Reduce	Cui et al. (2014)	K: number of clusters	Real, synthetic	DBI: 0.0133 for WMC 0.0128 for DMC (N = 4b, K = 100, 16 reducers)	5.38 min for WMC 5.29 min for DMC (N = 4b, K = 100, 16 reducers)	–	Up to 4 billion points	No	Data points
Spark	Tsapanos et al. (2016)	ε: threshold	Real	NMI: 0.8412 (N = 621k, ε = 0.5)	9 h (N = 621k, ε = 0.5)	2.6× (N = 621k, ε = 0.5)	Up to 2.8 million points	No	Face videos
Peer-to-peer	Azimi et al. (2017)	K: number of clusters	Real, synthetic	AC: 80.14% (N = 35k, K = 7)	–	–	Up to 1 million points	No	Data points
Vertical scaling									
GPU	Cuomo et al. (2017)	K: number of clusters D: input size	Real, synthetic	–	–	88× (N = 500k, K = 128, D = 128)	Up to 500,000 points	No	Data points
Multi-core CPU	Hadian and Shahrivari (2014)	K: number of clusters	Real, synthetic	SSE: 7.42E+12	12.9 s	8.2×	Up to 11,620,300 points	No	Data points

Table 1 (continued)

Platforms	Algorithms	Clustering criteria					Big Data criteria		
		Input parameters	Datasets type	Quality	Running time	Speed-up	Volume	Velocity	Variety
		chunk_size: size of chunks		(N = 11m, K = 10)	(N = 11m, K = 10)	(N = 11m, K = 100, 12 cores, chunk_size= 10000)			
FPGA	Jia et al. (2015)	K: number of clusters	Real, synthetic	–	238.86 s (N = 10k, K = 300)	20.5× (N = 3000, K = 60)	Up to 20,000 points	No	Data points

AC, accuracy; DBI, Davies–Bouldin index; DMC, distribution-based merge clustering; N, the size of the tested dataset; NMI, normalized mutual information; SSE, sum of squares error measure; WMC, weight-based merge clustering

clustering algorithms. The two steps involved in this algorithm can be easily performed in parallel on chunks of the dataset. Table 2 presents the performance of the DBSCAN algorithm when implemented using different Big Data platforms. The experiments of the parallel implementations of DBSCAN demonstrated that they reached an interesting speed-up in comparison with the original version algorithm that attains 11× (the MapReduce-based implementation of DBSCAN for 1.2 billion records).

4.4 Observations and open issues

Also, it has to be noted that most of the parallel clustering algorithms proposed in the literature do not handle real-time data and focus on a single type of data, which limits their adequacy to process Big Data. Indeed, most of the data nowadays are unceasingly produced in real-time, which necessitate a continuous processing. Such processing should minimize the storage and computation costs in order to analyse large-scale real-time data. Despite the difficulties encountered with this kind of data, the parallel clustering methods, which are proposed in Huang et al. (2015) and Sotiropoulou et al. (2014), handle data stream and real-time data efficiently using the GPU and FPGA platforms respectively.

The variety of data is also an important dimension of the Big Data, which is unfortunately difficult to handle in the context of clustering. Indeed, most of the parallel clustering algorithms are designed for numerical data. A few others are specialized in other types of data, such as text data, multimedia data. However, the Big Data also takes into account heterogeneous structured, semi-structured, and even unstructured data as it represents the largest proportion. Indeed, clustering unstructured data is a challenging task due to the absence of a recognizable representation. To the best of our knowledge, there is a lack of parallel algorithms for clustering multi-view, heterogeneous, or multi-modal big data.

It should also be noted that most of the reviewed algorithms require fixing some initial parameters, which involves a complex beforehand study to decide the appropriate values. Such studies are often calling on information about the data distribution, in addition to considerable efforts and time, which is not always feasible. However, there are very few free-parameter algorithms, which are easy to use particularly in the context of Big Data, where excessive human interactions should be prevented or minimized when processing data. Therefore, there is a persistent need for parallel clustering algorithms that can meet all observations mentioned above.

5 Conclusions

This paper proposed an in-depth review of the latest parallel clustering algorithms sorted according to the Big Data platforms used. There are two fundamental categories of platforms, which can handle large-scale data processing. In the first category, we addressed the clustering algorithms based on MapReduce, Spark, and Peer-to-Peer networks. These platforms form part of the horizontal scaling platforms. While in the second category, which is known as the vertical scaling platforms, we focus on the clustering algorithms conceived with Multi-core processors, GPU, and FPGA. All the reviewed algorithms were analysed according to the strategies adopted to ensure the parallelism. This work also includes a detailed comparison of the discussed clustering algorithms based on some common criteria of validation clustering result in Big Data context.

Table 2 Comparison of the recent parallel clustering algorithms of DBSCAN

Platforms	Algorithms	Clustering criteria				Big Data criteria		
		Input parameters	Datasets type	Running time	Speed-up	Volume	Velocity	Variety
Horizontal scaling								
Map-Reduce	He et al. (2014)	ε: distance	Real	58 s	About 11.29×	Up to 1.2 billion records	No	Spatial data
		Minpts: minimum cluster size		(64 tasks, N = 1.2b, ε = 0.001, Minpts = 500)	(64 tasks, N = 1.2b, ε = 0.001, Minpts = 500)			
Spark	Han et al. (2016)	ε: distance	Synthetic	1493 min	About 137×	Up to 1 million points	No	Data points
		Minpts: minimum cluster size		(N = 1m, 512 cores, ε = 25, Minpts = 5)	(N = 1m, 512 cores, ε = 25, Minpts = 5)			
Peer-to-peer	Savvas and Tselios (2016)	ε: distance	Synthetic	67.55 s	About 17.5×	Up to 100,000 points	No	2D data points
		Minpts: minimum cluster size		(N = 100k, 33 nodes)	(N = 100k, 33 nodes)			
Vertical scaling								
GPU	Andrade et al. (2013)	R: proximity radius	Synthetic	82.9 s	111.6×	Up to 700,000 points	No	2D data objects
		MinPts: minimum cluster size		(N = 700k, MinPts = 4, R = 0.05)	(N = 700k, MinPts = 4, R = 0.05)			
Multi-core CPU	Erdem and Gündem (2014)	ε: minimum distance	Synthetic	9.4 s	–	Up to 50,000 points	No	2D fuzzy data objects
		μ: minimum cluster size		(N = 50k, ε = 120, μ = 7, c = 24)				
		c: number of cores						
FPGA	Scicluna and Bouganis (2015)	ε: distance	Real	211.88 ms	33.77×	Up to 25,000 points	No	Data points

Table 2 (continued)

Platforms	Algorithms	Clustering criteria				Big Data criteria		
		Input parameters	Datasets type	Running time	Speed-up	Volume	Velocity	Variety
		Minpts: minimum cluster size		(N = 19,504, parallel elements = 300, Minpts = 80, $\varepsilon = 25$)	(N = 19,504, parallel elements = 300, Minpts = 80, $\varepsilon = 25$)			

N, the size of the tested dataset

After this thorough study, it is observed that most of the reviewed approaches concern some well-known clustering algorithms, such as *k*-Means, DBSCAN, and OPTICS. This choice lies in the fact that these algorithms are widely studied and suitable for parallelism in different ways. Also, it was noted that some Big Data platforms are becoming less commonly used in clustering, such as the Peer-to-Peer networks. This is due to the rapid advancements in the field of parallel and distributed computing. These efforts gave rise to new programming models and more powerful hardware that exceed the limitations of the old platforms. It is important to point that most of the parallel clustering algorithms proposed in the literature do not handle real-time data and focus on a single type of data, which limits their capability to process Big Data. Consequently, the real-time and the heterogeneous data processing remains challenging issues in the context of clustering.

References

Aggarwal CC, Han J, Wang J, Yu PS (2003) A framework for clustering evolving data streams. In: Proceedings of the 29th international conference on very large data bases, VLDB '03, vol 29. VLDB Endowment, Berlin, pp 81–92

Akhter S, Roberts J (2006) Multi-core programming: increasing performance through software multi-threading, 1st edn. Books by engineers, for engineers. Intel Press, Hillsboro

Altinigneli MC, Plant C, Böhm C (2013) Massively parallel expectation maximization using graphics processing units. In: Proceedings of the 19th ACM SIGKDD international conference on knowledge discovery and data mining, KDD '13. ACM, Chicago, pp 838–846. https://doi.org/10.1145/2487575.2487628

An F, Koide T, Mattausch HJ (2012) A k-means-based multi-prototype high-speed learning system with FPGA-implemented coprocessor for 1-NN searching. IEICE Trans Inf Syst E95–D(9):2327–2338

Andrade G, Ramos G, Madeira D, Sachetto R, Ferreira R, Rocha L (2013) G-DBSCAN: a GPU accelerated algorithm for density-based clustering. Procedia Comput Sci 18(Supplement C):369–378. https://doi.org/10.1016/j.procs.2013.05.200

Ankerst M, Breunig MM, Kriegel HP, Sander J (1999) Optics: ordering points to identify the clustering structure. In: Proceedings of the 1999 ACM SIGMOD international conference on management of data, SIGMOD '99. ACM, Philadelphia, pp 49–60. https://doi.org/10.1145/304182.304187

Azimi R, Sajedi H, Ghayekhloo M (2017) A distributed data clustering algorithm in p2p networks. Appl Soft Comput 51(Supplement C):147–167. https://doi.org/10.1016/j.asoc.2016.11.045

Banharnsakun A (2017) A mapreduce-based artificial bee colony for large-scale data clustering. Pattern Recognit Lett 93(Supplement C):78–84. https://doi.org/10.1016/j.patrec.2016.07.027

Ben-Dor A, Shamir R, Yakhini Z (1999) Clustering gene expression patterns. J Comput Biol 6(3–4):281–297. https://doi.org/10.1089/106652799318274

Bharill N, Tiwari A, Malviya A (2016) Fuzzy based scalable clustering algorithms for handling big data using apache spark. IEEE Trans Big Data 2(4):339–352. https://doi.org/10.1109/TBDATA.2016.2622288

Brown SD, Francis RJ, Rose J, Vranesic ZG (1992) Field-programmable gate arrays. Kluwer international series in engineering and computer science. Springer, Boston. https://doi.org/10.1007/978-1-4615-3572-0

Bustamam A, Burrage K, Hamilton NA (2012) Fast parallel markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ELLPACK-R sparse format. IEEE/ACM Trans Comput Biol Bioinform 9(3):679–692. https://doi.org/10.1109/TCBB.2011.68

Cordova I, Moh TS (2015) DBSCAN on resilient distributed datasets. In: International conference on high performance computing simulation (HPCS). IEEE, Amsterdam, pp 531–540. https://doi.org/10.1109/HPCSim.2015.7237086

Cui X, Gao J, Potok TE (2006) A flocking based algorithm for document clustering analysis. J Syst Archit 52(8):505–515. https://doi.org/10.1016/j.sysarc.2006.02.003

Cui X, Zhu P, Yang X, Li K, Ji C (2014) Optimized big data k-means clustering using MapReduce. J Supercomput 70(3):1249–1259. https://doi.org/10.1007/s11227-014-1225-7

Cuomo S, De Angelis V, Farina G, Marcellino L, Toraldo G (2017) A GPU-accelerated parallel k-means algorithm. Comput Electr Eng. https://doi.org/10.1016/j.compeleceng.2017.12.002

Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th conference on symposium on opearting systems design and implementation, OSDI'04, vol 6. USENIX Association, Berkeley

Deng Z, Hu Y, Zhu M, Huang X, Du B (2015) A scalable and fast optics for clustering trajectory big data. Cluster Comput 18(2):549–562. https://doi.org/10.1007/s10586-014-0413-9

Ekanayake J, Li H, Zhang B, Gunarathne T, Bae SH, Qiu J, Fox G (2010) Twister: a runtime for iterative MapReduce. In: Proceedings of the 19th ACM international symposium on high performance distributed computing. ACM, pp 810–818

Enright AJ, Van Dongen S, Ouzounis CA (2002) An efficient algorithm for large-scale detection of protein families. Nucleic Acids Res 30(7):1575–1584

Erdem A, Gündem Tİ (2014) M-FDBSCAN: a multicore density-based uncertain data clustering algorithm. Turk J Electr Eng Comput Sci 22:143–154. https://doi.org/10.3906/elk-1202-83

Ester M, Kriegel HP, Sander J, Xu X (1996) A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the second international conference on knowledge discovery and data mining, KDD'96. AAAI Press, Portland, pp 226–231

Fahad A, Alshatri N, Tari Z, Alamri A, Khalil I, Zomaya AY, Foufou S, Bouras A (2014) A survey of clustering algorithms for Big Data: taxonomy and empirical analysis. IEEE Trans Emerg Top Comput 2(3):267–279. https://doi.org/10.1109/TETC.2014.2330519

Farooq U, Marrakchi Z, Mehrez H (2012) FPGA architectures: an overview. In: Tree-based heterogeneous FPGA architectures, chap. 2. Springer, New York, pp 7–48. https://doi.org/10.1007/978-1-4614-3594-5_2

Ferreira Cordeiro RL, Traina Junior C, Machado Traina AJ, López J, Kang U, Faloutsos C (2011) Clustering very large multi-dimensional datasets with mapreduce. In: Proceedings of the 17th ACM SIGKDD international conference on knowledge discovery and data mining, KDD '11. ACM, San Diego, pp 690–698. https://doi.org/10.1145/2020408.2020516

Gehweiler J, Meyerhenke H (2010) A distributed diffusive heuristic for clustering a virtual p2p supercomputer. In: IEEE international symposium on parallel distributed processing, workshops and Phd forum (IPDPSW). IEEE, Atlanta, pp 1–8. https://doi.org/10.1109/IPDPSW.2010.5470922

Gepner P, Kowalik MF (2006) Multi-core processors: New way to achieve high system performance. In: International symposium on parallel computing in electrical engineering (PARELEC'06). Bialystok, Poland, pp 9–13. https://doi.org/10.1109/PARELEC.2006.54

Gouineau F, Landry T, Triplet T (2016) Patchwork, a scalable density-grid clustering algorithm. In: Proceedings of the 31st annual ACM symposium on applied computing, SAC '16. ACM, Pisa, pp 824–831. https://doi.org/10.1145/2851613.2851643

Hadian A, Shahrivari S (2014) High performance parallel k-means clustering for disk-resident datasets on multi-core CPUs. J Supercomput 69(2):845–863. https://doi.org/10.1007/s11227-014-1185-y

Han D, Agrawal A, Liao WK, Choudhary A (2016) A novel scalable DBSCAN algorithm with Spark. In: IEEE international parallel and distributed processing symposium workshops (IPDPSW). IEEE, Chicago, pp 1393–1402. https://doi.org/10.1109/IPDPSW.2016.57

Han J, Kamber M, Pei J (2012) Cluster analysis: basic concepts and methods. In: Data mining, The Morgan Kaufmann series in data management systems, 3rd edn, chap. 10. Morgan Kaufmann, pp 443–495. https://doi.org/10.1016/B978-0-12-381479-1.00010-1

Harish P, Narayanan PJ (2007) Accelerating large graph algorithms on the GPU using CUDA. In: High performance computing—HiPC 2007. Lecture notes in computer science. Springer, Berlin, pp 197–208. https://doi.org/10.1007/978-3-540-77220-0_21

Hartigan JA, Wong MA (1979) Algorithm as 136: a k-means clustering algorithm. Appl Stat 28(1):100. https://doi.org/10.2307/2346830

Havens TC, Bezdek JC, Leckie C, Hall LO, Palaniswami M (2012) Fuzzy c-means algorithms for very large data. IEEE Trans Fuzzy Syst 20(6):1130–1146. https://doi.org/10.1109/TFUZZ.2012.2201485

He Y, Tan H, Luo W, Feng S, Fan J (2014) MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. Front Comput Sci 8(1):83–99. https://doi.org/10.1007/s11704-013-3158-3

Huang P, Li X, Yuan B (2015) A parallel gpu-based approach to clustering very fast data streams. In: Proceedings of the 24th ACM international on conference on information and knowledge management, CIKM '15. ACM, Melbourne, pp 23–32. https://doi.org/10.1145/2806416.2806545

Hussain HM, Benkrid K, Seker H, Erdogan AT (2011) FPGA implementation of k-means algorithm for bioinformatics application: an accelerated approach to clustering microarray data. In: NASA/ESA conference on adaptive hardware and systems (AHS). IEEE, San Diego, pp 248–255. https://doi.org/10.1109/AHS.2011.5963944

Jia F, Wang C, Li X, Zhou X (2015) SAKMA: specialized FPGA-based accelerator architecture for data-intensive k-means algorithms. In: Algorithms and architectures for parallel processing. Springer, Cham, pp 106–119. https://doi.org/10.1007/978-3-319-27122-4_8

Jin R, Kou C, Liu R, Li Y (2013) Efficient parallel spectral clustering algorithm design for large data sets under cloud computing environment. J Cloud Comput Adv Syst Appl 2(1):18. https://doi.org/10.1186/2192-113X-2-18

Kanungo T, Mount DM, Netanyahu NS, Piatko CD, Silverman R, Wu AY (2002) An efficient k-means clustering algorithm: analysis and implementation. IEEE Trans Pattern Anal Mach Intell 24(7):881–892. https://doi.org/10.1109/TPAMI.2002.1017616

Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392. https://doi.org/10.1137/S1064827595287997

Kraus JM, Kestler HA (2010) A highly efficient multi-core algorithm for clustering extremely large datasets. BMC Bioinform 11(1):169. https://doi.org/10.1186/1471-2105-11-169

Kriegel HP, Pfeifle M (2005) Density-based clustering of uncertain data. In: Proceedings of the eleventh ACM SIGKDD international conference on knowledge discovery in data mining. ACM, Chicago, pp 672–677. https://doi.org/10.1145/1081870.1081955

Lanczos C (1950) An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. United States Governm., Press Office Los Angeles

Laney D (2001) 3D data management: controlling data volume, velocity, and variety. Technical Report, 949, Gartner

Li C, Zhang Y, Jiao M, Yu G (2014) Mux-Kmeans: multiplex Kmeans for clustering large-scale data set. In: Proceedings of the 5th ACM workshop on scientific cloud computing, ScienceCloud '14. ACM, Vancouver, pp 25–32. https://doi.org/10.1145/2608029.2608033

Lin F, Cohen WW (2010) Power iteration clustering. In: Proceedings of the 27th international conference on machine learning (ICML-10). Omnipress, Haifa, pp 655–662

Lin KW, Lin CH, Hsiao CY (2014) A parallel and scalable cast-based clustering algorithm on GPU. Soft Comput 18(3):539–547. https://doi.org/10.1007/s00500-013-1074-y

Liu R, Li X, Du L, Zhi S, Wei M (2017) Parallel implementation of density peaks clustering algorithm based on spark. Procedia Comput Sci 107(Supplement C):442–447. https://doi.org/10.1016/j.procs.2017.03.138

Luo G, Luo X, Gooch TF, Tian L, Qin K (2016) A parallel DBSCAN algorithm based on spark. In: IEEE international conferences on big data and cloud computing, social computing and networking, sustainable computing and communications. IEEE, Atlanta, pp 548–553. https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.85

Mallios X, Vassalos V, Venetis T, Vlachou A (2016) A framework for clustering and classification of big data using spark. In: Debruyne C, Panetto H, Meersman R, Dillon T, Kühn E, O'Sullivan D, Ardagna CA (eds) On the move to meaningful internet systems: OTM 2016 conferences, vol 10033. Springer, Cham, pp 344–362. https://doi.org/10.1007/978-3-319-48472-3_20

Melo D, Toledo S, Mourao F, Sachetto R, Andrade G, Ferreira R, Parthasarathy S, Rocha L (2016) Hierarchical density-based clustering based on GPU accelerated data indexing strategy. Procedia Comput Sci 80:951–961. https://doi.org/10.1016/j.procs.2016.05.389

Milojicic DS, Kalogeraki V, Lukose R, Nagaraja K, Pruyne J, Richard B, Rollins S, Xu Z (2002) Peer-to-peer computing. Technical Report. HPL-2002-57, HP Labs

Nanni M, Pedreschi D (2006) Time-focused clustering of trajectories of moving objects. J Intell Inf Syst 27(3):267–289. https://doi.org/10.1007/s10844-006-9953-7

Nickolls J, Buck I, Garland M (2008) Scalable parallel programming. In: IEEE hot chips 20 symposium (HCS). IEEE, pp 40–53

Owens J, Houston M, Luebke D, Green S, Stone J, Phillips J (2008) GPU computing. Proc IEEE 96(5):879–899. https://doi.org/10.1109/JPROC.2008.917757

Patwary MA, Palsetia D, Agrawal A, Liao WK, Manne F, Choudhary A (2013) Scalable parallel optics data clustering using graph algorithmic techniques. In: Proceedings of the international conference on high performance computing, networking, storage and analysis, SC '13. ACM, Denver, pp 49:1–49:12. https://doi.org/10.1145/2503210.2503255

Rodriguez A, Laio A (2014) Clustering by fast search and find of density peaks. Science 344(6191):1492–1496. https://doi.org/10.1126/science.1242072

Savvas IK, Tselios D (2016) Parallelizing DBSCAN algorithm using MPI. In: IEEE 25th International conference on enabling technologies: infrastructure for collaborative enterprises (WETICE). IEEE, Paris, pp 77–82. https://doi.org/10.1109/WETICE.2016.26

Scicluna N, Bouganis CS (2015) ARC 2014: a multidimensional FPGA-based parallel DBSCAN architecture. ACM Trans Reconfig Technol Syst 9(1):2:1–2:15. https://doi.org/10.1145/2724722

Sheikholeslami G, Chatterjee S, Zhang A (2000) Wavecluster: a wavelet-based clustering approach for spatial data in very large databases. VLDB J Int J Very Large Data Bases 8(3–4):289–304. https://doi.org/10.1007/s007780050009

Shi S, Yue Q, Wang Q (2014) FPGA based accelerator for parallel DBSCAN algorithm. Comput Model New Technol 18(2):135–142

Singh D, Reddy CK (2014) A survey on platforms for big data analytics. J Big Data 2(1):8. https://doi.org/10.1186/s40537-014-0008-6

Sinha A, Jana PK (2016) A novel k-means based clustering algorithm for big data. In: International conference on advances in computing, communications and informatics (ICACCI). IEEE, pp 1875–1879. https://doi.org/10.1109/ICACCI.2016.7732323

Skillicorn D (1999) Strategies for parallel data mining. IEEE Concurr 7(4):26–35. https://doi.org/10.1109/4434.806976

Sotiropoulou CL, Gkaitatzis S, Annovi A, Beretta M, Giannetti P, Kordas K, Luciano P, Nikolaidis S, Petridou C, Volpi G (2014) A multi-core FPGA-based 2D-clustering implementation for real-time image processing. IEEE Trans Nuclear Sci 61(6):3599–3606. https://doi.org/10.1109/TNS.2014.2364183

Stone JE, Gohara D, Shi G (2010) OpenCL: a parallel programming standard for heterogeneous computing systems. Comput Sci Eng 12(3):66

Sun Z, Fox G, Gu W, Li Z (2014) A parallel clustering method combined information bottleneck theory and centroid-based clustering. J Supercomput 69(1):452–467. https://doi.org/10.1007/s11227-014-1174-1

Tsapanos N, Tefas A, Nikolaidis N, Pitas I (2015) A distributed framework for trimmed kernel k-means clustering. Pattern Recognit 48(8):2685–2698. https://doi.org/10.1016/j.patcog.2015.02.020

Tsapanos N, Tefas A, Nikolaidis N, Pitas I (2016) Efficient mapreduce kernel k-means for big data clustering. In: Proceedings of the 9th hellenic conference on artificial intelligence, SETN '16. ACM, Thessaloniki, pp 28:1–28:5. https://doi.org/10.1145/2903220.2903255

Valiant LG (1990) A bridging model for parallel computation. Commun ACM 33(8):103–111. https://doi.org/10.1145/79173.79181

Van Dongen S (2008) Graph clustering via a discrete uncoupling process. SIAM J Matrix Anal Appl 30(1):121–141. https://doi.org/10.1137/040608635

Voulgaris S, Gavidia D, van Steen M (2005) Cyclon: inexpensive membership management for unstructured p2p overlays. J Netw Syst Manag 13(2):197–217. https://doi.org/10.1007/s10922-005-4441-x

Wang J, Yuan D, Jiang M (2012) Parallel K-PSO based on MapReduce. In: IEEE 14th international conference on communication technology, pp 1203–1208. IEEE, Chengdu. https://doi.org/10.1109/ICCT.2012.6511380

Wang B, Yin J, Hua Q, Wu Z, Cao J (2016) Parallelizing k-means-based clustering on spark. In: International conference on advanced cloud and Big Data (CBD). IEEE, Chengdu, pp 31–36. https://doi.org/10.1109/CBD.2016.016

Winterstein F, Bayliss S, Constantinides GA (2013) FPGA-based k-means clustering using tree-based data structures. In: The 23rd international conference on field programmable logic and applications. IEEE, Porto, pp 1–6. https://doi.org/10.1109/FPL.2013.6645501

Yan W, Brahmakshatriya U, Xue Y, Gilder M, Wise B (2013) p-PIC: parallel power iteration clustering for big data. J Parallel Distrib Comput 73(3):352–359. https://doi.org/10.1016/j.jpdc.2012.06.009

Yang J, Li X (2013) MapReduce based method for big data semantic clustering. In: IEEE international conference on systems, man, and cybernetics. IEEE, pp 2814–2819. https://doi.org/10.1109/SMC.2013.480

Yıldırım AA, Özdoğan C (2011) Parallel wavecluster: a linear scaling parallel clustering algorithm implementation with application to very large datasets. J Parallel Distrib Comput 71(7):955–962. https://doi.org/10.1016/j.jpdc.2011.03.007

Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on hot topics in cloud computing, HotCloud'10. USENIX Association, Berkeley

Zayani A, Ben N'Cir CE, Essoussi N (2016) Parallel clustering method for non-disjoint partitioning of large-scale data based on spark framework. In: IEEE international conference on big data (Big Data). IEEE, Washington, DC, pp 1064–1069. https://doi.org/10.1109/BigData.2016.7840708

Zhang Y, Mueller F, Cui X, Potok T (2010) Large-scale multi-dimensional document clustering on GPU clusters. In: IEEE international symposium on parallel distributed processing (IPDPS). IEEE, pp 1–10. https://doi.org/10.1109/IPDPS.2010.5470429

Zhao W, Ma H, He Q (2009) Parallel k-means clustering based on MapReduce. In: Cloud computing. Lecture notes in computer science. Springer, Berlin, pp 674–679. https://doi.org/10.1007/978-3-642-10665-1_71