Iterative big data clustering algorithms: a review

Amin Mohebi1,*,†, Saeed Aghabozorgi1, Teh Ying Wah1, Tutut Herawan1 and Ramin Yahyapour2

1Faculty of Computer Science and Information Technology, University of Malaya, Kuala Lumpur, Malaysia
2Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG), Göttingen, Germany

SUMMARY
Enterprises today are dealing with the massive size of data, which have been explosively increasing. The key requirements to address this challenge are to extract, analyze, and process data in a timely manner. Clustering is an essential data mining tool that plays an important role for analyzing big data. However, large-scale data clustering has become a challenging task because of the large amount of information that emerges from technological progress in many areas, including finance and business informatics. Accordingly, researchers have dealt with parallel clustering algorithms using parallel programming models to address this issue. MapReduce is one of the most famous frameworks, and it has attracted great attention because of its flexibility, ease of programming, and fault tolerance. However, the framework has evident performance limitations, especially for iterative programs. This study will first review the proposed iterative frameworks that extended MapReduce to support iterative algorithms. We summarize these techniques, discuss their uniqueness and limitations, and explain how they address the challenging issues of iterative programs. We also perform an in-depth review to understand the problems and the solving techniques for parallel clustering algorithms. Hence, we believe that no well-rounded review provides a significant comparison among parallel clustering algorithms using MapReduce. This work aims to serve as a stepping stone for researchers who are studying big data clustering algorithms. Copyright © 2015 John Wiley & Sons, Ltd.

1. INTRODUCTION
In today’s era of big data, we are dealing with a massive amount of data that are increasing significantly in different ways. For example, Yahoo! Web graph is reported to reach 1 billion nodes and 7 billion edges in 2002 [1]; social networking websites such as Twitter and Facebook data span several terabyte; Wikipedia and YouTube data are of similar size, which produces hundreds of gigabytes per minute [2]. To deal with this huge amount of data, clustering has become an essential tool that empowers data scientists for analyzing and discovering distribution patterns in a broad range of data. Accordingly, numerous data mining algorithms have been studied to deal with large-scale data sets. However, traditional data mining techniques are no longer able to capture the large amount of data. Hence, many researchers have shifted their focus to parallel clustering algorithms that would improve the bottleneck of traditional clustering methods on a single machine [3]. In this way, parallel processing models have recently attracted considerable attention, which are used to process large-scale data sets. MapReduce has become the most popular parallel processing model that is used in many companies, including Facebook, mainly because of its simplicity [4]. Despite its advantages, MapReduce has to deal with numerous critical issues and challenges as
explained in [5] such as wasting bandwidth, I/O, and CPU cycles for iterative programs, including PageRank [6], social network analysis, neural-network analysis, and clustering.

This survey aims to provide a valuable guidebook of problems and solving techniques in the context of iterative applications. Furthermore, we aim to provide a point of reference for future works that aim to deal with big data sets using the parallel clustering algorithms. To summarize, the main contributions of this work are as follows:

1. The most significant past literature related to MapReduce processing model was reviewed with an emphasis on iterative processing improvement techniques.
2. A significant number of research papers related to parallel clustering algorithms based on MapReduce were discussed.
3. The advantages and disadvantages of current works are discussed, which reveal the differentiating factors among the existing parallel algorithms using MapReduce.
4. Observations were made based on the problem and solutions to provide opportunities for future work.

The most relevant work to the first contribution of this paper is the recent survey of large-scale analytical query processing in MapReduce [7]. However, our work narrows down this paper to focus more on iterative processing techniques. In addition, our study provides an in-depth analysis of various clustering algorithms that can be executed in parallel using MapReduce. To the best of our knowledge, no relevant survey with an emphasis on parallel clustering algorithms based on MapReduce has been conducted.

The rest of this paper is organized as follows: Section 2 provides an overview of big data with focus on MapReduce/Hadoop. Section 3 presents the existing approaches to improve the performance of MapReduce based on iterative processing techniques and reviews current parallel clustering algorithms based on MapReduce. Section 4 analyzes the current parallel clustering algorithms. Section 5 concludes this work.

2. BIG DATA MANAGEMENT

Big data is a term utilized to refer to the large amount of data that can be useful for people and corporations. However, data are increasingly sourced from the various fields that are disorganized, messy, and difficult to be clustered such as large amounts of public and private data. Hence, most companies cannot store, process, and analyze these data. Big data is still in its fancy stage. But, several definitions of big data currently exist. For example, the most recent definition of big data is characterized by five Vs in [8, 9], namely, (1) volume, (2) variety, (3) velocity, (4) value, and (5) veracity (Figure 1).

![Figure 1. Five Vs of big data.](image-url)
• **Volume.** This dimension is related to the ability of the clustering algorithm to handle a massive amount of data. In this way, parallelism helps to accommodate the rapid growth of big data by dividing the input data into the memory of several machines.

• **Variety** refers to the ability of the clustering algorithm to deal with different types of data.

• **Velocity** refers to the speed of clustering algorithms in dealing with big data. Speed up with respect to parallelism can be considered for this dimension of big data. Most iterative clustering algorithms use several iterations to obtain a result, and the methods must be applied several times to improve the quality. These methods will probably be impractical in processing large amounts of data, which take too long to be processed. Parallelism can solve this issue by distributing the data into separate machines, thereby increasing the speed.

• **Value** refers to the process of discovering hidden values of big data sets, which then documents and socializes as realized values.

• **Veracity** is one of the important aspects of big data, which refers to the reliability, provenance, and accurateness of data.

2.1. **MapReduce overview**

The growth rate in volume of data has led many programmers to distribute the data among several machines to accelerate the computation and process massive amounts of data. In this way, data processing models such as Samza [10], Tez [11], and MapReduce [12] are used to divide the data between machines, and each processor can work separately with the same algorithm on several parts of the data. Overall, parallel processing applications are divided into two groups, namely, traditional parallel applications [13, 14] and data-intensive applications [15–17]. Traditional parallel applications are assuming that data can be fit into the memory of distributed machines. However, the data collection size is rapidly increasing, so traditional parallel computation mechanisms cannot solve the big data problem.

Data-intensive applications need more computational and data resources, which run within the conditional loop until termination. However, selection of moving the computer to the data or data moving to the computer is based on the requirements and loads. But, in data-intensive applications, data locality has become more important than before, because it significantly reduces network cost.

There are various parallel processing models for computing data-intensive applications such as OpenMP, MPI [18], and MapReduce. However, MapReduce is one of the most widely used frameworks due to its simplicity (two functions, a map, and a reduce), and the programmers can easily monitor the data distribution, replications, load balancing, and so on [19].

In MapReduce, data points are processed over a distributed environment of commodity machines through two phases. First, the Mapping phase is defined by a Map function. The input data are split into Map functions, and computations are performed to produce the intermediate results that are in the form of key/value pairs. The next phase is the Reduce function, which takes a single key and processes the specified function on its associated values at a time, collected, and considered as output of the job. After intermediate results are generated, the data are shuffled into the corresponding Reducer to perform the Reduce tasks. A major portion of the job execution time is utilized to move the intermediate results over the network. Overall, data are processed through the following steps as shown in Figure 2.

**Map function.** A Map task reads data points as its input. It then produces the intermediate key/value pairs that consist of two parts, namely, key that is a group number of values and a value that is associated with the key. Map tasks call a process to produce the new key/value, which is locally sorted by their keys.

**Combiner function.** An optional function is applied to intermediate results that have significant repetition in their own Map task. The function can perform a partial reduction before passing the intermediate key/value pairs to the reducer.

**Partition function.** This function aims to perform partitioning. The intermediate keys are hashed to determine which reduce partition should process the keys. By default, hash partitioner is used in Hadoop, which generally provides good balancing.

**Reduce function.** The output of Map tasks are collectively considered inputs of the reduce phase. The Reduce task is applied once for each key, and its values are then processed with respect to the
user-defined Reduce function, which is invoked by each reducer once for each key. The input to each reducer is processed in increasing key order.

2.2. Hadoop overview

Using traditional mechanism to analyze and query massive amounts of data is difficult. Hadoop [5] is an open-source software that enables data scientists to distribute large-scale data sets into blocks and takes advantage of the distributed environment in commodity hardware. Hadoop is designed to scale out by adding more machines in a cluster with commodity hardware, rather than scale up, which implies the addition of more resources such as RAM and CPU, which are expensive. The Hadoop distributed file system (HDFS) is designed as workflow rather than primary storage. The data are stored into the HDFS once and can be accessed several times, bringing the code to data, which are split into blocks (typically 64 MB) and are accessible between machines of a cluster for parallel processing. Hadoop can cope with failures through data replication, which can be stored and replicated over machines, with one of the replicas stored in a different rack for further data reliability and availability. Hadoop cluster is a set of commodity machines that are networked in one location. Hadoop consists of two parts, namely, HDFS and MapReduce engine, for parallel processing purposes. The cluster consists of different numbers of daemon/servers.

DataNode stores the data in HDFS, provides the location of data, and connects to NameNode, which does not keep the data, but merely stores the metadata that can be mapped from file to block and location of block. A secondary NameNode assists in monitoring the state of the cluster in case the latter fails. Integration occurs between HDFS and MapReduce, which is performed through master and slave nodes. Typically, a master node consists of TaskTracker and JobTracker from the MapReduce layer, whereas slave nodes comprise only the TaskTracker from this layer (Figure 3). JobTracker is responsible for scheduling MapReduce jobs and assigns the tasks to the TaskTrackers that are used for execution in each slave that is used for execution in each slave node.
JobTracker may assign several tasks to Tasktracker depending on available resources. When the TaskTracker completes its tasks, the JobTracker allocates new tasks. In case of machine failure, a heartbeat message is sent from TaskTracker to JobTracker periodically. If the master node did not receive this heartbeat message from one machine for some time, it will consider that machine as a failed machine. Instead of fixing the failed machine, Hadoop reschedules the failed task to another machine as backup. The failure of the JobTracker itself cannot be solved automatically; however, this failure should not be a serious problem because the probability of failure of one particular machine is low.

2.3. Hadoop/MapReduce limitations for iterative applications

MapReduce has been researched \cite{19, 20} in terms of its performance limitations. Figure 4 shows the generic weaknesses and limitations of MapReduce as explained in \cite{7}. Our focus is on iterative programs. Thus, we identified limitations that are related to MapReduce for these programs, which are discussed as follows:

* **Lack of loop-aware task scheduling.** MapReduce does not support multi-staging of tasks in a single run. Hence, the same operations (e.g., task initialization and cleanup) start new MapReduce jobs, which have to repeatedly reload and dump data during iterations. This means scheduling overhead when dealing with large-scale data sets.

* **Lack of handling the static data.** Two kinds of data are involved in MapReduce process, namely, static and state data. Static data is the key problem of MapReduce, which must be unchanged during iterations.

    By design, in some of iterative algorithms (e.g., $k$-means), after generating state data, the Reducer needs the static data for performing some operations (e.g., averaging). Hence, the static data must be reloaded and shuffled during iterations, which is impractical for huge data sets, and it results in communication overhead.

* **Lack of Asynchronous execution.** Starting new map tasks before completing the reduce tasks in the previous iteration is not allowed in standard MapReduce. Hence, each iteration must wait until the previous iteration is complete, which results in synchronization overhead.

* **Lack of built-in termination condition.** The system has no built-in feature for termination condition, which requires an additional MapReduce job at each iteration, thereby incurring overhead because of scheduling extra tasks.

![Figure 4. MapReduce limitations.](image-url)
3. REVIEW OF THE MAIN AVAILABLE WORKS

In this section, a review first was performed on the frameworks that improved MapReduce based on iterative processing techniques. Then, in the main part of this paper, a closer look at parallel clustering algorithms based on MapReduce is conducted. Despite the existence of this work [21], which reviewed only a few parallel clustering algorithms, we have not found any well-rounded and comprehensive review that focused on parallel clustering algorithms using MapReduce.

3.1. Evaluation of iterative MapReduce frameworks

In this section, it is attempted to provide a summary of the current iterative frameworks based on the significant parameters presented in Figure 5. Table I summarizes the reviewed frameworks based on the taxonomy presented in Figure 5.

3.1.1. Execution model. MapReduce is slow in dealing with iterative programs, which require a chain of map/reduce jobs. The MapReduce model splits jobs into multiple jobs that are not allowed to run in a single run and thus retards the process. Hence, numerous frameworks are proposed to support the iteration programs. HaLoop [5] is similar to MapReduce, and tasks are scheduled to slave nodes through the parallel master node with a TaskTracker daemon, which is intended to communicate with the master node. Each task either can Reduce or Map tasks. By design, there are main modifications in HaLoop. First, a new feature in master node is known as loop control, which starts new map–reduce tasks repeatedly until the stopping condition is met. In contrast to MapReduce, this feature eliminates the extra MapReduce (MR) jobs for checking the stop condition, which reduces the communication overhead. Second, a new task scheduler for iterative applications supports data locality to co-locate the tasks for the same operations, which enables the data reuse within the iterations. Third, the intermediate results are cached to the disk to improve performance. However, this framework like MapReduce uses synchronous execution, and the reducer has to wait until finishing the map tasks. Figure 6 shows the modified architecture of HaLoop.

Another framework that uses MapReduce execution model is Twister [22], which relies on a publish/subscribe messaging mechanism for its communication. The architecture of Twister differs from MapReduce by several modifications. First, Twister stores the input data in local disk as native...
<table>
<thead>
<tr>
<th>Framework</th>
<th>Execution model</th>
<th>In-memory system</th>
<th>Modification in programming interface</th>
<th>Termination condition support</th>
<th>Iterative processing technique</th>
<th>File system</th>
<th>Load balancing support</th>
<th>Fault tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaLoop [5]</td>
<td>Synchronous</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Inter-iteration</td>
<td>HDFS</td>
<td>N/A</td>
<td>Strong</td>
</tr>
<tr>
<td>Twister [22]</td>
<td>Synchronous</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Long running</td>
<td>N/A</td>
<td>N/A</td>
<td>Weak</td>
</tr>
<tr>
<td>PriIter [31]</td>
<td>Asynchronous</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Socket connection</td>
<td>HDFS</td>
<td>Yes</td>
<td>Strong</td>
</tr>
<tr>
<td>MapReduce online [25]</td>
<td>Synchronous</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
<td>Pipelining</td>
<td>HDFS</td>
<td>N/A</td>
<td>Strong</td>
</tr>
<tr>
<td>iMapReduce [32]</td>
<td>Asynchronous</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Socket connection</td>
<td>HDFS</td>
<td>Yes</td>
<td>Strong</td>
</tr>
<tr>
<td>Spark [26]</td>
<td>Asynchronous</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>General abstraction</td>
<td>HDFS</td>
<td>N/A</td>
<td>Weak</td>
</tr>
<tr>
<td>CBP [30]</td>
<td>Asynchronous</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>General abstraction</td>
<td>HDFS</td>
<td>N/A</td>
<td>Weak</td>
</tr>
<tr>
<td>Pregel [23]</td>
<td>Synchronous</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>A natural graph API</td>
<td>GFS</td>
<td>N/A</td>
<td>Weak</td>
</tr>
</tbody>
</table>

HDFS, Hadoop distributed file system; N/A, not applicable; CBP, continuous bulk processing.
files, rather than distributed file system. However, this can be considered as a drawback of Twister, which means the static data must fit into the collective memory of worker nodes. This assumption would be impractical to handle huge data sets, because it would require many machines (e.g., 256 machines with 4 GB RAM for processing 1 TB data). Second, Twister also maintains the intermediate results in the memory, which improves the performance. Third, Twister uses a driver, which handles the entire computation. In this way, a broker network is established to send input data from the main driver program to the Mapper and Reducer, which are then collected and returned to the main program (Figure 7).

Pregel [23] is also an in-memory system that uses bulk synchronous parallel [24] as its computation model. It also suffers from inefficiencies due to synchronous scheduling. MapReduce online [25] also uses synchronous execution. This framework supports pipelining of intermediate results between tasks. The modification is applied in transferring data between Reduce and Map tasks, which creates a connection between Map and Reduce tasks. Hence, the output of Reduce tasks can directly pass to the Map tasks. However, this framework does not support overlapping of tasks. Therefore, the Map tasks have to wait until finishing the Reduce tasks.
Recently, the researchers proposed several frameworks that support asynchronous execution, which is not allowed in MapReduce. For example, Spark [26] is an in-memory framework with a different system for its computation model. Spark is similar to DryadLINQ [27] and FlumeJava [28] because it presents each data set as objects that can be converted into Resilient Distributed Data set (RDDs) by using transformations (e.g., map and filter). Several options are known as actions, which act as an aggregator to collect the result and return it to the driver (e.g., collect and count). Spark takes advantage of an advanced directed acyclic graph [29], which can reduce extra stages of MapReduce jobs. Therefore, the jobs can be executed faster, which allows Spark to run more complex tasks in a single run of many stages instead of splitting them into multiple jobs. Continuous bulk processing (CBP) [30] also maintains the iteration state in memory. Unlike MapReduce, CBP supports cyclic data flow that can be helpful to speed up the iterative applications. In this way, CBP provides a data abstraction, which is useful to control the iterative data flows.

PrIter [31] is the only framework that uses prioritized execution, which assigns a priority value to each data point. Instead of updating all the data, a subset of data with high priority is selected for execution at each iteration. In iMapReduce [32], a persistent socket connection is proposed which can connect the Mapper to its corresponding Reducer. This method has the ability to perform Map tasks asynchronously such that a joint operation is proposed for static and state data when the state results arrive.

3.1.2. Iterative processing technique. HaLoop outperforms standard MapReduce by supporting iterative processing method, which has the intent of co-locating tasks for the same data point over iterations. In this way, ‘inter-iteration locality’ is used to allow the placement of Map and Reduce tasks on the same machine in different iterations, but with access to the same data. This approach allows the user to cache the data and re-use them between iterations, thereby reducing I/O cost.

Twister uses ‘long-running map/reduce tasks’ technique. In this way, the state data can be unchanged. Therefore, reloading the large data at each iteration is not required. This approach improves the performance overhead for iterative applications.

iMapReduce supports the iteration by using persistent mechanism, which keeps the MapReduce tasks alive in the entire iteration process. In this way, a persistent direct connection is proposed between Map and Reduce tasks, which supports the passing of state data directly to the mapper without writing them on DFS. Therefore, this framework can eliminate the creation of new MapReduce jobs repeatedly for the same operations. In addition, the proposed socket connection can avoid the need of a new MapReduce job for joining the state and static data.

PrIter performs similar to iMapReduce by providing a persistent connection between Map and Reduce tasks. The framework supports the iterative processing by dividing the data into initial static and state, which are preloaded to workers before starting the map tasks. Hence, the framework can keep the static data unchanged, and the state data can be updated and joined with static data during iterations. In this framework, a termination check is proposed to terminate the iterative computation, if the stopping condition is met.

In MapReduce online, the iterative processing is handled by providing a pipelined architecture, which provides a connection between Map and Reduce tasks. In MapReduce online, the map tasks have to wait until the completion of Reduce tasks in the previous iteration. However, the proposed method so-called online aggregation [33] can help to represent snapshots of computation between MapReduce jobs.

Spark takes advantage of a data abstraction that has the ability to cache both static and state data in memory for general reuse. Spark represents each data set as an object. Therefore, it allows users to choose which data set should be cached for further iterations. CBP also offers a data abstraction that supports iterative processing. This abstraction empowers the framework for maintenance of persistent state to reuse the previous work for incremental processing and groupwise processing, which helps to
reduce the data movement in the system. Pregel also supports the iterative processing by offering a natural graph API that persists the values associated to each vertex and its edges across each iteration.

3.1.3. Load balancing and fault tolerance. HaLoop provides a similar fault tolerance as Hadoop. HaLoop uses an intra-iteration fault tolerance and reconstructs the cache to the worker assigned with the failed worker partition. Twister uses a similar fault tolerance mechanism in Google and Hadoop implementations to save the application state during iterations. Therefore, the entire computation can roll back iterations in case of failure. The fault tolerance of iMapReduce is the same as that of the standard MapReduce. However, state data are stored in a distributed file system to retain the results from previous iterations. Therefore, the last iteration will be recomputed along with its state data in case of failure. All tasks in iMapReduce are persistent and conflict with the standard MapReduce task scheduling mechanism. Thus, iMapReduce cannot benefit from MapReduce mechanism for load balancing. This framework performs task migration periodically, which means that the master node distinguishes between leaders and stragglers by comparing the completion time of the received notification of completion of tasks after each iteration.

MapReduce online added a bookkeeping mechanism to the current Hadoop implementation. In addition, the output of map tasks can be considered tentative until JobTracker notifies the reducer about the successful completion of tasks. In case of failure in reduce tasks, a new copy of task is started. Given that the output of map tasks is stored in a local disk in case of failure in map tasks, the output will be reproduced if any reduce tasks fail.

PrIter uses ‘MRPair migration’ mechanism for load balancing. In this approach, the master node compares the average processing time received from map/reduce pairs with the threshold. If the processing time is less than a threshold, then the assigned worker is considered a slow worker and the pairs migrate to the faster worker. This new MRPair loads both StateTable and graph partition from DFS. PrIter is resilient against task failures. The master node considers a task or worker failure if the node did not receive any response from probing during a period of time. The failure task is migrated to a new healthy worker and all the other MRPaired roll back to the same fixpoint to re-compute the last failed task.

Spark takes advantage of RDDs for fault tolerance. In case of failure, rolling back to the entire program at the last checkpoint is not required. Spark performs fault tolerance differently in a more efficient way that reloads only the lost partitions or RDDs, which helps to accelerate fault recovery.

Pregel offers fault tolerance by using check pointing. This approach is achieved through ping message. If the master did not receive a ping message from the workers, then those workers are considered failures. In case of failure, the master node assigns the graph partition to the healthy available workers, and they reproduce their partition state from the latest checkpoint at the beginning of the last iteration. However, to recover a failure, all the vertices in the iteration must be re-executed.

3.1.4. File system. It is another class in Figure 5 in which Haloop, PrIter, MapReduce online, Spark, and CBP support HDFS; however, Pregel supports Google File System (GFS). By contrast, this feature is not available in Twister, and the user needs to separate large data sets into multiple splits.

3.2. Parallel clustering algorithms based on MapReduce

This section summarizes the majority of available parallel clustering algorithms. We first categorize the current parallel algorithms into six classes, namely, centroid-based clustering, density-based clustering, connectivity-based clustering, high-dimensional clustering, similarity-based clustering, and co-clustering. Next, we briefly discuss the articles based on the taxonomy presented in Figure 9.

3.2.1. Centroid-based clustering. K-means clustering has become one of the most popular centroid-based clustering algorithms and it has been identified as one of the top 10 algorithms in [34]. This algorithm has been widely used to analyze and process large amounts of data. Traditional serial execution of $k$-means takes a number of $(Nkd)$ computations at each iteration, where $N$ is the number of points, $k$ is the number of clusters, and $d$ is the dimension. With a single processor, this computation can be expensive when the algorithm deals with large-scale data sets, because an unspecified number of iterations is required. The distance computation of one object and its corresponding center is
irrelevant to the distance computation of the other objects and their corresponding centers. Hence, the distance computation can be executed in parallel by using MapReduce.

Zhao et al. [35] and Ping et al. [36] proposed a parallel k-means clustering algorithm using MapReduce. They presented this approach by using three functions of MapReduce, namely, Map, Combine, and Reduce. First, the Map tasks calculate the closest distance for data points from every initial centroid of clusters. Next, the combiner calculates a partial sum of values, which can be used as input for Reduce tasks. Finally, the Reduce tasks compute the centroids by dividing the partial sum of samples into the number of samples assigned to a similar cluster. This process is repeated until the stopping condition is met. These steps are summarized in Table II.

However, both studies take the advantage using MapReduce to speed up the clustering process. But, data-intensive algorithms (e.g., k-means) need several iterations before completion, so that they would require several scans over data sets. In this way, the algorithm performs several k-means jobs (Figure 10) during iterations. The Mapper processes each data object, and it is called several times, which can be problematic in handling large data sets.

In another approach, Nguyen et al. [37] present a new approach using Incremental k-means Algorithm [38] (IKM). The authors implemented their approach based on two stages of MapReduce paradigm. First, the Mapper loads data segment, and it executes the IKM on the loaded data segment. Then, the Reducer receives the intermediate results and executes the IKM again to obtain the clustering results. The authors claimed that this approach only needs one scan over data set, which can reduce the communication cost between Mapper and Reducer. However, they do not provide a detailed algorithm.

Table II. MapReduce tasks of Parallel k-means.

<table>
<thead>
<tr>
<th>Steps</th>
<th>Tasks</th>
</tr>
</thead>
</table>
| Step 1  | (i) The data points are inputted in the form of point set and centroids.  
         | (ii) Each map task calculates the distance of each object from cluster centers.  
         | The objects are then assigned to the similar cluster center based on the obtained distance. |
| Step 2  | A combiner is performed to reduce the size before sending the key/value pairs to the reducer. |
| Step 3  | The reducer generates the final results by dividing the partial sum of data points into the number samples. |
| Step 4  | If the convergence condition is met, the program will stop. If not, the program iterates Steps 2 to 4. |
In addition, this approach is an approximate solution, so that it cannot produce the exact clustering results.

Unlike other studies that focused on post-initialization, Bahmani et al. [39] focused on initial phase to optimize clustering accuracy.

This approach would benefit from $K$-means++ [40], which obtains an initial set of cluster centers that is near the optimum solution. However, $K$-means++ has the drawback of inherent sequential nature, which makes parallel execution difficult, so that it cannot be used to deal with large-scale data sets. Therefore, scalable ++ is used to run $K$-means++ in parallel by reducing the number of passes that need to be obtained. This approach is available in MLlib, a Spark implementation of machine-learning algorithms. Spark extends MapReduce and it seems to be suitable for iterative algorithms, with providing an in-memory system in the form of RDDs to speed up reuse. For example, in [41], $k$-means is programmed by Spark framework within 10 iterations using 25–100 machines. The results illustrate that RDDs greatly outperform standard Hadoop/MapReduce implementation in the future iterations.

Li et al. [42] focused on concurrently running $k$-means processes based on MapReduce with multiple initial center groups. Its main goal includes avoidance of serial execution of $k$-means and more focus on initial centroids. In this approach, the intermediate results are evaluated at each iteration to prune the hopeless $k$-means processes. By this way, the hopeless $k$-means process attempts are abandoned, which speeds up the future iterations. However, because of using MapReduce, it still lacks the ability to cache data between iterations for improving performance.

Aljarah and Ludwig [43] used global best particle swarm optimization [44], which performs a globalized search to determine the best solution. Therefore, this method can solve the sensitivity problem of $k$-means on initial cluster centers and also avoid local optima convergence. This approach consists of three modules that are executed by two MapReduce jobs (Figure 11). First, the new particle centroids are generated, which are then evaluated by fitness function (second module) in another MapReduce phase. In the third module, the outputs of the first and second modules are merged, and the new output is sent for the next iterations. However, this approach could achieve high clustering quality and good speedup. But, it still cannot be used for huge data sets because it would require reloading the whole point sets, which is necessary to generate new centroids and calculate the distance between all data points and particle centroids.

Cui et al. [45] presented a novel approach to eliminate the $k$-means dependency, which can be considered to be the biggest bottleneck in parallelism. This approach uses three MapReduce jobs. The first MapReduce job samples the original data sets using $k$ as the number of clusters and a sampling factor that has probability, which can be expressed as

$$P_x = \frac{1}{e^2 N}$$
where \( N \) is the number of points, and \( \varepsilon \in (0,1) \) is used to control the sampling size. At the end of the sample section, \( 2^{k} \) samples remain, which are clustered by Mappers in the second MapReduce job. The Mappers generate \( 2^{k} \)\(^2 \) centers, which are sent to one Reducer and merged into \( k \) final centers. Then, the final results are produced in the third MapReduce job. The final results show that this algorithm performs better than traditional \( k \)-means, parallel \( k \)-means [35], and \( k \)-means \( \parallel \) [39]. However, the sampling approach probably cannot produce the clustering quality similar to that of sequential \( k \)-means because of clustering on divided data.

The work in [46] uses Cop-\( k \)means, a modified version of \( k \)-means with two pairwise of constraints, namely, Must-link (\( ML \)) and Cannot-link (\( CL \)). The main aim is to avoid computational dependencies between Mappers. \( ML \) and \( CL \) make the algorithm too sensitive to the assignment order of instances, so that this may lead to consequent constraint violation. This means dependence on assigning the instances in other Mappers. To address this issue, the authors provide a Map function by generating \( ML \) and \( CL \) constraints from the partial data set assigned to every Mapper. The output of each Mapper is all the instances with the assigned cluster, which are sent to single the Reducer. The Reducer then generates the new centers, which are sent back to the Mappers. The experimental study indicates that with the increase in number of constraints, the standard Cop-\( k \)means shows greater computation time than its parallel version. However, the clustering accuracy reduces due to sequencing \( CL \) distance.

Sun et al. [47] combined information bottleneck (IB) theory [48] and centroid-based method using iterative MapReduce framework called Twister. IB clustering is expensive because of its high computational cost. Thus, the parallel IB based on MapReduce is proposed to determine the initial centroids \( C^0 \). This is a one-pass part without the need of reloading the whole points set. The main focus of this work is on a parallel clustering method based on Twister. In this way, the distance of each sample and center of centroids \( C^0 \) is calculated to generate the new centroids. This approach takes advantage of long running of Twister, which provides caching strategies between iterations. For example, the input of each Mapper is cached to speed up reuse of static data. In addition, the input of Reducers can be cached for performing the termination check. Twister is compared against Hadoop and shown to outperform it in various setups (e.g., different number of nodes and iterations).

The work in [49] presents a parallel iterative clustering algorithm based on Partitioning Around Medoids [50] (PAM). First, the algorithm is run based on MapReduce, and each Map function calculates the closets medoid with the assigned object to its cluster. Second, the Reducer calculates the
most central object as new medoid to the cluster. The authors then focused on running PAM using Twister. The results indicate that Twister can solve the iteration problems 50–100 times faster than Hadoop. The reason is that Twister is an in-memory system and stores the input data in memory. Hence, PAM algorithm can store all the clustered objects in memory without rereading them again from file system. This results in having a better running time.

The work [49] also studied clustering large applications [50] (CLARA), which is an iterative \( k \)-medoid algorithm. In contrast to PAM, the whole iteration can be reduced into two MapReduce jobs because it only clusters the small subsets of data set. By this way, the first job calculates random subsets, which are sent to the next job for measuring the quality. The results indicate that this approach could achieve to minimal job latency, because the input data is only loaded twice.

Punam et al. [51] introduced the implementation of canopy clustering algorithm based on Twister and Hadoop. This is achieved by applying four major MapReduce modules. In the first module, a list of canopy centers are generated, which are assigned to the points in the next module. However, authors argue in favor of Twister, which offers long running of tasks. But, this can be observed that these two modules have less expense because of having few MR jobs. The main part of this work is the last two modules, where \( k \)-means algorithm is programmed to generate the final centers.

Hence, the authors used iterative Twister framework. The experimental study illustrates that Twister can outperform Hadoop in terms of clustering time, which was 24 times faster, and it also offers better speedup, which was three times faster.

The work in [5] provides a comparison of HaLoop and Hadoop performance for \( k \)-means algorithm. It has two main phases: the first job calculates the cluster means at each iteration, and then the second MapReduce job is applied to calculate and determine every cluster membership after completion of iterations. The results show that HaLoop outperforms Hadoop in various setups. This makes sense due to the significant difference of HaLoop, a cache mechanism to store the Mapper inputs, which are invariant during \( k \)-means iterations. Hence, whole of iteration can be reduced into two MapReduce jobs. Another reason is the proposed loop-control in HaLoop, which is responsible for checking the termination condition, so that it terminates extra MapReduce jobs required for this checking.

### 3.2.2. Density-based clustering

Considerable parallel density-based research usually focused on improving communication cost because of the high computational complexity of serial density-based algorithms. One of the most important density-based clustering algorithms is density-based spatial clustering of applications with noise (DBSCAN) [52]. Li and Xi [53] are the first to focus on running DBSCAN in MapReduce. This approach splits data points into clusters within each partition, which are then merged using MapReduce. The authors stated that their approach could work efficiently in massive data sets. However, they do not provide detailed algorithms to partition the data and merge the clusters.

He et al. [54] proposed a parallel density-based clustering algorithm using MapReduce. The main contribution of this work is to reduce spatial complexity. By this way, this approach is programmed in four-stage MapReduce paradigm. First, the authors proposed a partitioning strategy using grid file, which are executed by only one MR round. In the second stage, DBSCAN is executed locally in each grid cell with spatial indices such as \( R^k \)-tree. The third stage involves finding the same point indexes from merge candidates set and obtaining the clustering ID, which can be achieved by one MapReduce phase. Final stage includes two MR jobs: (1) build global mapping merges the results from the previous stage to discover a global structure of clusters mapping, although this can be executed with a single thread due to its small size; and (2) re-label for replacing the local clustering results with a new global clustering ID. However, this approach reduced the MapReduce iterations. But, the problem of this approach is that collecting all the clusters in each grid into one single machine results in high computational overhead. Moreover, the performance of this algorithm highly depends on distribution of raw spatial data. In addition, \( e \)-neighborhoods can be problematic in case of clusters with varying densities [55].

OPTICS [56] is proposed to deal with the \( e \)-neighborhoods problem in DBSCAN. This approach checks the expansion of a cluster by checking its \( e \). If \( e \) rapidly grows, \( e \) terminates the current cluster and starts another cluster. However, this algorithm cannot be easily implemented based on MapReduce, because the points have to be merged serially to the recent cluster, which has to be
performed to determine where the clusters should be separated. Thus, Kim et al. [55] proposed a new density-based algorithm called DBCURE to discover clusters with varying densities and used $R^*$-tree to find the ellipsoidal $r$-neighborhood of a given data point. This algorithm can be implemented based on MapReduce, which empowers the original DBCURE to find several clusters together in parallel. However, $R^*$-tree is an iterative algorithm that is not suitable for MapReduce style.

Zhao et al. [57] used structural clustering algorithm (SCAN), which finds the clusters by using breath-first-search in an iterative way. The main contribution of this work is a 3-stage MapReduce paradigm. First, the structural similarity of edges is calculated. Next, the algorithm uses a cutting-off strategy for structural similarity of edges in which each edge that is less than $\epsilon$ will be stopped. Finally, label propagation [58] is used to find the connected components. By this way, the Mappers output the structural information of each vertex to the reducer to update the label of each vertex as activated or inactivated, which will be iterated until all nodes are inactivated. However, the authors reduced the MR jobs by removing iterative mechanism such as breath-first-search. But, there is still computational overhead due to extra MR jobs that needed in last stage for reloading the vertices from file system.

3.2.3. Connectivity-based clustering. Connectivity-based clustering, which is also known as hierarchical clustering, has been widely used in various applications that take advantage of its informative representation of clustering results. However, hierarchical clustering is inefficient in handling large data sets due to large a number of iteration, which results in high computational cost. Therefore, the need for parallelism of hierarchical clustering algorithms becomes even more vital to deal with a massive amount of data.

Sun et al. [59] proposed the first parallel hierarchical clustering algorithm based on MapReduce by using two techniques, as indicated in Figure 12. First, co-occurrence-based feature selection method is used to reduce the dimension of feature vectors, which is necessary to improve the efficiency of hierarchical clustering algorithms. Second, batch updating is used to reduce the number of iterations, which can improve the computational cost. The main advantage of this technique is to store as many state data and communications operations as possible in one iteration. This means a significant reduction in number of MapReduce jobs. In this way, several iterations are combined into on matrix file, which is updated and stored in the memory. The results show that the proposed method could reduce the execution time, and also, the number of updating iterations is reduced to 6.01%.

Jin et al. [60] proposed a novel approach to run hierarchical clustering algorithms in parallel. The main idea is to split the main problem into a set of overlapped sub-problems and then solve each sub-problem that can be merged into an overall solution. This approach includes two types of MapReduce jobs. First, a Prim-MR job is proposed, which is executed once and then followed by a series of Kruskal-MRs, where mappers act as identity Mapper and pass the key/value pairs to Kruskal Reducer. The aim of first stage is to prune the incorrect edges in a pre-stage step, which seems to come with the advantage of data reduction after. However, the number of iterations is reduced, and the results show achieving good speedup. But, the overhead still incurs due to the setting up of several MapReduce jobs in second stage to run the Kruskal algorithm [61].

In another work, Gao et al. [62] presented a new approach by using agglomerative hierarchical clustering (AHC) based on MapReduce. This work aims to present a text clustering algorithm, which can process large amounts of data in parallel. Its main goal includes a neuron set initialization method, which can avoid inappropriate features and improve the clustering accuracy. The experimental study indicates that the clustering time and accuracy is improved when compared with traditional $k$-means and AHC algorithm.
3.2.4. High-dimensional clustering. High-dimensional clustering is a challenging task because of a growing number of objects. In such data sets, the problems can be irrelevant features or noise dimensions, which lead to unwanted scatter of data points and hide the real cluster pattern of data [63]. Two algorithms are proposed to solve the aforementioned problems, namely, projected and subspace clustering algorithms. Both algorithms intend to identify the clusters and their subspaces. However, projected and subspace clustering algorithms cannot detect clusters in large-scale data sets because of its high computational property and I/O cost. Moreover, they need a large number of iterations over all the data. To solve such problems, parallelism can be considered.

Ferreira Cordeiro et al. [64] proposed the first parallel subspace-clustering algorithm based on MapReduce. The authors studied algorithms that aim to minimize communication cost. First, parallel clustering (ParC) is proposed. In this way, each Mapper reads data points once and followed by Reducer to run the subspace clustering algorithm.

The aim of this stage is to find local clusters. Finally, all the clusters are merged into a single machine to generate the final global clusters. Another algorithm that is proposed in this work is sample and ignore. This algorithm can be executed by using two different MapReduce jobs. In the first phase, the main idea is to sample the data in Mappers, which are sent to Reducer to perform the subspace clustering. The initial clusters are generated in the first MapReduce job. Then, the local clustering results are generated in phase 2, which are merged with initial clusters in a single machine. This approach uses a novel method called BoW, which chooses the best of two algorithms. However, this method is still an approximate solution because of its sampling mechanism, which cannot produce accurate results. But, the approach can produce the clusters with few MapReduce jobs, because the whole algorithm can be executed by using a few MR rounds.

High-dimensional clustering based on MapReduce had not received much attention until Fries et al. [65] presented an exact solution for projected clustering. They used P3C [66] algorithm because of its parallelization-friendly structure and sound statistical model that allows an efficient MapReduce-based solution. However, the original P3C is still incapable of detecting clusters in big data sets because of issues related to Sturges’ rule and the Poisson test, as explained in [67]. Therefore, the authors modified the original P3C to solve the aforementioned issues. One of the strengths of this approach is that it provides a shortcut that calculates each object at the same time, which is helpful to achieve good load balancing. The results show that this approach could reduce the I/O and running time cost while increasing the accuracy of clustering results.

3.2.5. Similarity-based clustering. Similarity-based methods consist of the algorithms that work on similarity matrices instead of representing the instances as a vector in a feature space (i.e., kernel k-means and spectral clustering). Kernel k-means needs to calculate and store the entire kernel matrix that must be accessed over iterations. However, this approach makes kernel k-means incapable of dealing with massive data sets.

The work in [68] focuses on kernel k-means that needs all entries of kernel matrix to be loaded at each iteration. This means that sequential kernel k-means cannot solve big data problems. Hence, the main aim of this approach is on scalability issue by eliminating the bottlenecks of the kernel k-means. In this way, a unified parallelization strategy is proposed to compute a generic family of embeddings of the data instances in a given data set. To reduce the communication cost, the authors proposed an in-memory matrix, which consists of a summary of embeddings. Moreover, this approach used two instances of embeddings that are based on Nystöm method [69] and \( p \)-stable distribution for approximating vector norms. Finally, combining one of the methods and the proposed parallel strategy generates the final clustering results. The experimental study shows that clustering quality is comparable with that of serial kernel k-means; however, it still is an approximate solution.

Spectral clustering is the next similarity-based clustering algorithm that can be used in a wide range of applications. The running time is the most important issue in spectral clustering, which can be cubic when dealing with large-scale data sets. Tsironis et al. [70] investigated on spectral clustering algorithms to be efficiently parallelized. The authors proposed a method by focusing on two directories, namely, k-means selection and eigensolver selection. They took advantage of k-means \([39]\) for the initialization phase, which enables their method to carefully select the initial centroids. The purpose of HEIGEN is to improve the efficiency of eigenvalues or eigenvectors by
adopting the use of block-based operations, which decreases the running time in spectral clustering. However, HEIGEN can be executed in parallel by MapReduce framework to enable handling of large matrices. But, parallel $k$-means and eigensolver are iterative algorithms, which require several MapReduce rounds until completion.

The work in [71] presented a parallel spectral clustering using $t$-nearest-neighbor sparse similarity matrices. By design, the sparse similarity matrix is not iterative, and the whole of an algorithm can be reduced into three MapReduce phases: (1) compute distances and find nearest neighbors; (2) modify the distance matrix to be symmetric; and (3) compute similarities. After generating the sparse matrix, the authors employed PARPAK [72] to obtain $k$-first eigenvectors. PARPAK is an eigensolver, which can be considered as an iterative algorithm. The authors used MPI and the results indicate that MPI is suitable for iterative algorithms. The reason is that MPI sends and receives data from each node’s memory, and it does not use disk I/O. However, there is a limitation that data must fit in the memory of each node.

3.2.6. Co-clustering. Co-clustering is a data mining technique that can be applied in various applications such as cancer subtype identification [73], document mining [74], and bio-informatics [75]. The problem of co-clustering is that it is impractical to apply on massive matrices. Thus, a distributed co-clustering can be considered as a solution, which splits raw input to the end clusters.

Papadimitriou and Sun [76] proposed the distributed co-clustering framework, which presents practical approaches for distributed co-clustering. The basic idea is to have two kinds of MapReduce jobs, namely, iteration over raw and the column group, and initialization of group matrix $G$ and label vector $r$ and $c$. For an $m \times n$ input matrix, $r$ is the row group assignments denoted by $\{1,2,...,k^m\}$, $c$ is the column group assignments denoted by $\{1,2,...,l^n\}$, and $G$ is the $(K \times l)$ group matrix. Table III shows the summary of MapReduce job steps for the co-clustering algorithm. To reduce communication cost, this approach does not transfer the matrix data between Mapper and Reducer, because it only transfers the label of each vector and $G$ matrix. The results demonstrate that distributed co-clustering is an exact solution and can obtain quality results similar to serial co-clustering. However, a large overhead will be incurred due to a global synchronization that is needed within iterations.

The same work is implemented by using iMapReduce in [77]. The results indicate that this approach can achieve fast convergence as compared with standard MapReduce. This makes sense due to the elimination of the repeated jobs by leveraging a persistent job for the whole process of co-clustering algorithm without initializing extra jobs for each row (column).

4. ANALYSIS

In the following, an analysis of parallel clustering algorithms is provided based on the different important parameters. It is noted that comparing such algorithms is not easy due to the different mechanisms used. Therefore, the main objective is that what is used, which helps to identify the drawback of existing parallel algorithms and may be useful for future researches. Table IV categorizes the reviewed works into several classes, mainly, algorithm class, data shuffling, and problem, mechanism, and the required modifications.

<table>
<thead>
<tr>
<th>Steps</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map function</td>
<td>(i) A sequence file of key/value pairs is stored on Hadoop distributed file system (HDFS)</td>
</tr>
<tr>
<td></td>
<td>(ii) Group matrix $G$ and column labels are globally broadcasted to all mappers.</td>
</tr>
<tr>
<td></td>
<td>(iii) The mapper computes the optima row label $r(i)$ for each row $i$.</td>
</tr>
<tr>
<td></td>
<td>(iv) The mapper computes the associated per-column statistics for that row.</td>
</tr>
<tr>
<td>Reduce function</td>
<td>For each cluster label, the row group statistics and group members are merged</td>
</tr>
<tr>
<td>Global sync</td>
<td>Collecting the new results for $G$ matrix and $r$ row-label vector</td>
</tr>
</tbody>
</table>

Table III. Parallel co-clustering steps.
<table>
<thead>
<tr>
<th>Reference</th>
<th>Clustering algorithm</th>
<th>Framework</th>
<th>Algorithm class</th>
<th>MR jobs</th>
<th>Data shuffling</th>
<th>Mechanism</th>
<th>Main problem</th>
<th>Mapper input cache</th>
<th>Reducer output cache</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>[35]</td>
<td>K-means</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Weak</td>
<td>Simple MR jobs</td>
<td>RID</td>
<td>Yes</td>
<td>Yes</td>
<td>Joining</td>
</tr>
<tr>
<td>[36]</td>
<td>K-means</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Weak</td>
<td>Simple MR jobs</td>
<td>RID</td>
<td>Yes</td>
<td>Yes</td>
<td>Joining</td>
</tr>
<tr>
<td>[49]</td>
<td>CLARA</td>
<td>MapReduce</td>
<td>Two</td>
<td>Low</td>
<td>Medium</td>
<td>Simple MR jobs</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>[49]</td>
<td>PAM</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Weak</td>
<td>Simple MR jobs</td>
<td>RID</td>
<td>Yes</td>
<td>Yes</td>
<td>Joining</td>
</tr>
<tr>
<td>[74]</td>
<td>Co-clustering</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Weak</td>
<td>Simple MR jobs</td>
<td>Breath-first-search</td>
<td>Yes</td>
<td>Yes</td>
<td>Batch Updating</td>
</tr>
<tr>
<td>[49]</td>
<td>Cop-k-means</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Medium</td>
<td>Simple MR jobs</td>
<td>RID</td>
<td>Yes</td>
<td>Yes</td>
<td>Joining</td>
</tr>
<tr>
<td>[37]</td>
<td>K-means</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Weak</td>
<td>2PKM</td>
<td>RID</td>
<td>Yes</td>
<td>Yes</td>
<td>Joining</td>
</tr>
<tr>
<td>[37]</td>
<td>K-means</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Weak</td>
<td>Simple MR jobs</td>
<td>RID</td>
<td>Yes</td>
<td>Yes</td>
<td>Joining</td>
</tr>
<tr>
<td>[43]</td>
<td>PSO</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Weak</td>
<td>Three-MR module</td>
<td>RID</td>
<td>Yes</td>
<td>Yes</td>
<td>Joining</td>
</tr>
<tr>
<td>[49]</td>
<td>CLARA</td>
<td>MapReduce</td>
<td>Two</td>
<td>Low</td>
<td>Medium</td>
<td>Simple MR jobs</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>[54]</td>
<td>DBSCAN</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Weak</td>
<td>DBCURE</td>
<td>CT</td>
<td>Yes</td>
<td>Yes</td>
<td>Joining</td>
</tr>
<tr>
<td>[54]</td>
<td>DBSCAN</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Medium</td>
<td>Grid</td>
<td>R-tree &amp; KD-tree</td>
<td>Yes</td>
<td>Yes</td>
<td>Partitioning strategy.</td>
</tr>
<tr>
<td>[57]</td>
<td>SCAN</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Medium</td>
<td>Structural similarity &amp; propagation</td>
<td>Breath-first-search</td>
<td>Yes</td>
<td>Yes</td>
<td>BSP</td>
</tr>
<tr>
<td>[60]</td>
<td>SHC</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Medium</td>
<td>Prim MR job</td>
<td>Rereading MSTs</td>
<td>Yes</td>
<td>Yes</td>
<td>Lazy operations</td>
</tr>
<tr>
<td>[59]</td>
<td>AHC</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Strong</td>
<td>Batch updating</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>[62]</td>
<td>AHC</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Weak</td>
<td>Simple MR jobs</td>
<td>A series of MR jobs</td>
<td>Yes</td>
<td>Yes</td>
<td>Lazy operations</td>
</tr>
<tr>
<td>[68]</td>
<td>Kernel k-means</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Weak</td>
<td>Embedding</td>
<td>Rereading Massive kernel matrix</td>
<td>Yes</td>
<td>Yes</td>
<td>Joining</td>
</tr>
<tr>
<td>[71]</td>
<td>Spectral</td>
<td>MapReduce/MPI</td>
<td>Third</td>
<td>Low</td>
<td>Medium</td>
<td>SSM/PARPAK</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>[70]</td>
<td>Spectral</td>
<td>MapReduce</td>
<td>Third</td>
<td>Low</td>
<td>Weak</td>
<td>Simple MR jobs</td>
<td>RID</td>
<td>Yes</td>
<td>Yes</td>
<td>Joining</td>
</tr>
<tr>
<td>[65]</td>
<td>P3C</td>
<td>MapReduce</td>
<td>Third</td>
<td>High</td>
<td>Weak</td>
<td>P3C⁺</td>
<td>CT</td>
<td>Yes</td>
<td>Yes</td>
<td>Joining</td>
</tr>
<tr>
<td>[64]</td>
<td>Subspace clustering</td>
<td>MapReduce</td>
<td>One &amp; two</td>
<td>Low</td>
<td>Strong</td>
<td>ParC &amp; SnI</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>[5]</td>
<td>K-means</td>
<td>HaLoop</td>
<td>Third</td>
<td>Low</td>
<td>Strong</td>
<td>Caching</td>
<td>SE</td>
<td>N/A</td>
<td>N/A</td>
<td>FT &amp; AE support</td>
</tr>
<tr>
<td>[47]</td>
<td>IB</td>
<td>Twister</td>
<td>Third</td>
<td>Low</td>
<td>Strong</td>
<td>Long running</td>
<td>FT &amp; SE</td>
<td>N/A</td>
<td>N/A</td>
<td>FT &amp; AE support</td>
</tr>
<tr>
<td>[49]</td>
<td>PAM</td>
<td>Twister</td>
<td>Third</td>
<td>Low</td>
<td>Strong</td>
<td>Long running</td>
<td>FT &amp; SE</td>
<td>N/A</td>
<td>N/A</td>
<td>FT &amp; AE support</td>
</tr>
<tr>
<td>[51]</td>
<td>Canopy</td>
<td>Twister</td>
<td>Two</td>
<td>Low</td>
<td>Strong</td>
<td>Long running</td>
<td>FT &amp; SE</td>
<td>N/A</td>
<td>N/A</td>
<td>FT &amp; AE support</td>
</tr>
<tr>
<td>[41]</td>
<td>K-means</td>
<td>Spark</td>
<td>Third</td>
<td>Low</td>
<td>Strong</td>
<td>Abstraction</td>
<td>Concurrent reduce tasks</td>
<td>N/A</td>
<td>N/A</td>
<td>Group reduction support</td>
</tr>
<tr>
<td>[77]</td>
<td>Co-clustering</td>
<td>iMapReduce</td>
<td>Third</td>
<td>Low</td>
<td>Strong</td>
<td>Socket connection</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

RID, rereading input data; FT, fault tolerance; SE, synchronous execution; AE, asynchronous execution; SSM, sparse similarity; CT, computation time.
4.1. Algorithm class

First, the current clustering algorithms are classified based on the number of MR jobs that are needed to perform MapReduce model as explained in [49]. This classification is a necessary part, because it helps to identify the suitable framework for executing the algorithms. For example, ParC is the only algorithm among the compared algorithms that can be classified into the first class. This class can be considered as embarrassing algorithms, because the whole of an algorithm can be performed in a single MR round. SnI and CLARA are classified into second class, which can perform the MapReduce model in a constant number of MR rounds. The first-class and second-class algorithms may be implemented with no cost as they require few MR jobs. The rest of the compared algorithms are classified into the third class, which entails extra costs because they require synchronization steps between iterations (e.g., check the termination condition). For example, the work in [35] belongs to the third class, and it requires several iterations until the stop condition is met. Hence, this is not suitable for MapReduce style. By contrast, the work [49] uses Twister to support ‘PAM-like’. This approach can reduce the whole of an algorithm in a single MR round as it provides a ‘long-running’ strategy to keep the tasks alive until meeting the termination condition.

4.2. Data shuffling

Improved performance can be achieved by reducing the shuffling of data over iterations. This includes techniques such as passing directly the data from Reducer to Mapper and joining state data with static data, which can result in better performance with removing extra MR jobs. Therefore, there is no need to write the output of reducers to the DFS.

The work in [49] uses CLARA, which requires two MR jobs to be performed, so the data-shuffling cost is medium. The work in [42] uses k-means based on MapReduce. In this way, after generating and sending the intermediate results to the Reducer, the static data are needed for averaging the operations. Hence, the static data must be shuffled between Mappers and Reducers at each iteration, which increases network communication cost. By contrast, [77] uses iMapReduce that lets the reduce tasks broadcast the updated state data to all Map tasks, which is shown to minimize the communication cost.

In case of Single-linkage Hierarchical Clustering (SHC) algorithm, in [60], the shuffling is dramatically decreased due to the Prim-MR job, which has the intent of cutting the edges in an early stage. However, the MSTs are needed to store on DFS at the Map side and then shuffle to the Reducers. Moreover, Kruskal Reducer cannot be processed and has to wait for input shuffling. The work in [59] reduced the shuffling cost by using batch updating, which is shown to minimize the number of iterations.

In the case of DBSCAN, [54] used grid file, which can be performed in a single MR round. Therefore, in the partitioning stage, the shuffling cost is minimized. The reason is that the data within space Si and its halo replication from bordering spaces are easily shuffled to the target Reducer. The authors used spatial indices such as breath-first-search in the next stage when running local DBSCAN, which is an iterative algorithm, so that it increases the shuffling cost for the computations of the neighbors of almost each node. This is in fact quite expensive when we speak about big data. In [57], the main contributions are to reduce the shuffling cost included by using structural similarity for each edge of the graph and cutting off the edges within a threshold.

The algorithms proposed for subspace clustering are SnI and ParC, which aim at reducing the data shuffling; they are most likely to exert their advantages on big data sets. In this way, SnI provides better performance due to its sampling method in the first phase and the filtering in the second phase. In contrast, the work in [65] did not use sampling approach, thus it is classified as an exact algorithm, although it may increase the shuffling cost due to the extra MR jobs.

To reduce shuffling cost, the approach in [74] synchronizes the only updated cluster assignments and cluster information, and it avoids shuffling of the row and column between iterations. In [77], co-clustering is implemented by iMapReduce, which supports the iterative processes of co-clustering algorithms by joining the static and state data.
4.3. Mechanism, problem, and the required modification

The works in [35–37, 49, 42] and [43] apply simple MR jobs to achieve a parallel clustering algorithm. However, these approaches could produce good results in terms of clustering time. But, the results show that they can be captured in large data sets due to job lag and rereading the input data at each iteration. In particular, these approaches work better by these modifications: (1) caching the Mapper input, which can help to keep the static data in memory, so that it can be used in each iteration without reloading; (2) caching the Reducer input, which is useful for checking the termination condition, and it can terminate extra MapReduce jobs for this step; and (3) a joining mechanism that can build a connection from Reduce task to Map task, which is suitable for algorithms such as k-means because Reducer needs static for the averaging operation.

Some of the studies [46, 57] use iterative graph algorithms (e.g., breath-first-search), which are not suitable for MaReduce style. This is because, at each iteration, the whole graph structure (i.e., adjacency lists) must be reloaded and shuffled from Mapper to the Reducer, while the graph structure is static, so that it should be unchanged during iterations. Bulk synchronous parallel model [30] is an appropriate alternative for iterative graph algorithms, which allows the graph algorithms to run on large vertices and edges.

The work [60] uses Kruskal’s algorithm to reduce the single-linkage problem of hierarchical clustering algorithms. The main issue of this approach is rereading the MSTs, which could be quite expensive in large data sets. Hence, one of the alternatives for this issue could be lazy operations that do not need to materialize the MSTs on DFS. In addition, proposing a location-aware schedule can help the KruskalReducer to be processed without waiting for the input shuffling.

The approach described in [71] proposes a hybrid mechanism by using MapReduce and MPI. In this way, MPI is used for the iterative part of the spectral algorithm, which seems to work better than MapReduce in terms of iterative applications. The work in [47, 49] implemented iterative algorithms (e.g., k-means, PAM, and IB) based on Twister. As expected, the proposed long-running mechanism helps to reduce the overhead within the iterations.

The work in [5] is also implemented by using iterative MapReduce (HaLoop), which takes advantage of caching mechanisms. Interestingly, the objective of the proposed approach is to reduce the MR rounds by caching the Mapper input and Reducer output. However, [5] claimed that HaLoop works better in terms of fault tolerance when compared with Twister due to long-running MaReduce tasks. But, these works lack the ability of asynchronous execution, thus, the Map tasks cannot start their execution, so that they have to wait until finishing other Map tasks. In contrast, the approaches in [41, 77] used iMapReduce and Spark, which supports asynchronous execution, which are shown to minimize the clustering time for k-means and co-clustering algorithms.

5. CONCLUSIONS AND OUTLOOK

Efficient parallel clustering algorithms and frameworks ensure that scalability and good performance are achieved when dealing with large-scale data sets. MapReduce introduces a new parallel data-processing approach in the era of big data. MapReduce has attracted considerable attention because of its flexibility, ease of programming, and fault tolerance. However, MapReduce has evident performance limitations, especially when dealing with iterative applications.

In this study, we studied various parallel clustering algorithms, which have attracted considerable attention in case of big data. These algorithms can outperform traditional sequential algorithms in terms of scale up and speedup metrics. Based on the detailed discussion of different parallel algorithms, the field of parallel big data clustering is still young and open for new research. The amount of data grows at an exponential rate, but the improvement in parallel processing models is slow. We observed that MapReduce has limitations in case of iterative applications. In addition, employing state-of-the-art technologies and techniques in several important big data parallel processing models (i.e., MapReduce, Spark, Twister, and HaLoop) still cannot solve the problem of iterative algorithms.

To summarize, the main findings of this study could be the following: (1) this is necessary to design a parallel processing model from scratch, which combines the merits of existing parallel processing models to make them more fault tolerant, while supporting iterative algorithms. (2)
However, MapReduce/Hadoop implementation does not support iterative techniques, but this should not be the end of MapReduce. This is because MapReduce is more mature when compared with iterative algorithms, and many vendors support it. Hence, a hybrid system could be a good alternative to handle the iterative big data algorithms. (3) Parallel data processing can help to improve the clustering time of large data sets, but it may degrade the quality and performance. Therefore, the main concern is to achieve a reasonable trade-off between quality and speed in the content of big data. (4) A comprehensive study with the aim of finding clustering algorithms that can be processed in parallel is unavailable. Notably, the main challenge is to find methods that can eliminate the inherent dependence of algorithms. (5) Most studies used the standard MapReduce, thus, a possible research direction is to apply more parallel clustering algorithms with other iterative frameworks (e.g., Pregel, PrIter, and CBP).

ACKNOWLEDGEMENTS

This work is supported by the University of Malaya High Impact Research grant no. vote UM.C/625/HIR/MOHE/SC/13/2 and the University of Malaya Research grant no. vote RP0061-13ICT.

REFERENCES

5. Bu Y, Howe B, Balazinska M, Ernst MD. HaLoop: efficient iterative data processing on large clusters, in 36th International Conference on Very Large Data Bases, 2010; vol. 3.
65. Fries ST, Wels S. Projected clustering for huge data sets in MapReduce | Chair of Computer Science 9, in International Conference on Extending Database Technology (EDBT 2014), Athens, Greece, 2014; 49–60.