2016 IEEE International Parallel and Distributed Processing Symposium Workshops

A novel scalable DBSCAN algorithm with Spark

Dianwei Han, Ankit Agrawal, Wei-keng Liao, Alok Choudhary
EECS Department
Northwestern University, Evanston, IL 60208, USA
corresponding authors: {dianweih, ankitag, wkliao, choudhar}@eecs.northwestern.edu

***Abstract*—DBSCAN is a well-known clustering algorithm which is based on density and is able to identify arbitrary shaped clusters and eliminate noise data. However, parallelization of DBSCAN is a challenging work because based on MPI or OpenMP environments, there exist the issues of lack of fault-tolerance and there is no guarantee that workload is balanced. Moreover, programming with MPI requires data scientists to have an advanced experience to handle communication between nodes which is a big challenge.**

We present a new parallel DBSCAN algorithm using the new big data framework Spark. In order to reduce search time, we apply kd-tree in our algorithm. More specifically, we propose a novel approach to avoid communication between executors so that we can locally obtain partial clusters more efficiently. Based on Java API, we select appropriate data structures carefully: Using Queue to contain neighbors of the data point, and using Hashtable when checking the status of and processing the data points. In addition, we use other advanced features from Spark to make our implementation more effective. We implement the algorithm in Java and evaluate its scalability by using different number of processing cores. Our experiments demonstrate that the algorithm we propose scales up very well. Using data sets containing up to 1 million high-dimensional points, we show that our proposed algorithm achieves speedups up to 6 using 8 cores (10k), 10 using 32 cores (100k), and 137 using 512 cores (1m). Another experiment using 10k data points is conducted and the result shows that the algorithm with MapReduce achieves speedups to 1.3 using 2 cores, 2.0 using 4 cores, and 3.2 using 8 cores.

***Keywords*-DBSCAN; clustering; big data; Spark framework;**

I. Introduction

Clustering is a data mining approach that divides data into groups that are meaningful, useful, or both [20]. Cluster analysis has been successfully applied to many fields: bioinformatics, machine learning, information retrieval, and statistics [20]. Well-known algorithms include K-means [13], BIRCH [24], WaveCluster [19], and DBSCAN [6]. Current clustering algorithms haven been categorized into four types: partitioning based, hierarchy-based, grid-based, and density-based [6]. Density Based Spatial Clustering of Applications with Noise (DBSCAN) is a density based clustering algorithm [6].

Parallel DBSCAN has been implemented with MPI and OpenMP [15], [7], [25], [4]. Generally, an MPI implementation can obtain better performance but the programmers will run into other issues: (1) for running-long jobs, it would be very frustrating if one failed process causes the whole job to be failed when using many cores to handle an extra large data set.

(2) programmers need to take care of implementation in detail, such as how to partition the data, how to deal with communication, synchronization, file location, and workload balancing. Besides parallelization with MPI, MapReduce-based approach is presented as well [14], [7], [9].

We propose a new distributed parallel algorithm with Spark that implements DBSCAN. The idea of our approach is as follows. The algorithm first reads data from the Hadoop Distributed File System (HDFS) and forms Resilient Distributed Datasets (RDDs), transforming them into data points. Certainly, this process is done in Spark driver. It then pushes all the data into multiple executors. Within each executor, partial clusters are built and sent to driver at the end of *foreach* statement. Each executor just performs its computation without communicating with others. This way we avoid shuffle operations that are very expensive. So we place some additional points (SEEDs: the new term we introduce in our paper) in each partial cluster. After all the partial clusters are collected through shared variable accumulator, the algorithm identifies the clusters that are supposed to be merged by SEEDs. Merging is done in driver code too. In our new design and implementation, we use the power of shared variables of Spark framework: broadcast and accumulator. Also, in order to shorten the search time for points' neighbors, we implement Java-based $kd-tree$ [3] to reduce complexity from $O(n^2)$ to $O(nlogn)$. The experiments performed on a distributed-memory machine show that the proposed algorithm can obtain scalable performance.

The organization of the paper is as follows: In Section II, we briefly give an overview of two frameworks based on big data: Map Reduce and Spark, and the basic idea of DBSCAN algorithm. Our proposed DBSCAN algorithm is introduced in Section III. In Section IV, we present the parallel implementation with Spark. The experiments and the results are presented in Section V, followed by some concluding remarks in Section VI.

II. Preliminaries

In this section, we first briefly review the basic idea of DBSCAN algorithm. And then we introduce two distributed computation frameworks that are very powerful and widely used in big data applications.

A. DBSCAN algorithm

DBSCAN is a clustering algorithm proposed by Ester [6]. And it has become one of the most common clustering algorithms because it is capable of discovering arbitrary shaped clusters and eliminating noise data [6]. The basic idea of this algorithm is finding all the *core points* and forming the clusters by clustering core points with all points (core or non–core) that are *reachable* from it. Essentially, DBSCAN algorithm is based on three basic definitions: core points, directly density–reachable, and density–reachable [25]. Given a data set D, of points.

eps-neighborhood of a point p is the neighborhood of $p \in D$ within a radius *eps*.

Definition 1: A point p is a *core point* if it has neighbors within a given radius (*eps*), and the number of neighbors is at least *minpts* (which is a threshold). In this case, the number of neighbors is called *density*.

Definition 2: A point y is *directly density-reachable* from x if y is within *eps-neighborhood* of x and x is a *core point*.

Definition 3: A point y is *density-reachable* from x if there is a chain of points $p_1, p_2, ..., p_n$, with $p_1 = x$, $p_n = y$ and p_{i+1} is directly density–reachable from p_i for all $1 <= i < n$, $p_i \in D$.

The pseudocode of the DBSCAN algorithm is given in Algorithm 1 [8]. The algorithm starts with an arbitrary point $p \in D$ and checks its *eps*–neighborhood (Line 4). If the *eps*–neighborhood size is bigger than pre–defined number minpts, the code generates a new cluster C. The algorithm then retrieves all density reachable points from p in D, and add them to the cluster C (Line 8–20). Otherwise, if the *eps*–neighborhood contains less than minpts points, then p is marked as noise (Line 6). The computational complexity of Algorithm is $O(n^2)$ where n is the number of data points. If we use spatial indexing, the complexity reduces to $O(nlogn)$ [3].

B. Two frameworks based on big data: MapReduce and Spark

Figure 1 shows that in Hadoop version 1, only MapReduce framework is available for distributed computation. But in Hadoop version 2, based on Yarn (resource manager), MapReduce, Spark, and other data processing frameworks are available. MapReduce and Spark may share the same HDFS, but it should be pointed out that Spark jobs can be run with or without Yarn (Standalone mode).

1) Map Reduce: In big data domain, MapReduce is a simple but powerful framework which makes programmer easily implement parallel processing. It is based on Hadoop Distributed File System (HDFS), which allows programmers to focus mainly on the problem at hand instead of worrying about the low level implementation details. Figure 2 tells us about how this programming model works. MAP workers read data from HDFS and process the data based on the business logic and then write intermediate data to local disk for sorting and shuffling process. It is also in the form of key–value pair. The locations of these buffered pairs on the local disk are passed to the master, which forwards it to the reduce workers. After a reduce worker is notified by master, it uses remote procedure call to read data from local disk of MAP workers, and then sorts data so that all occurrences of the same key are grouped together. The output of reduce function will be appended to final output files (generally HDFS).

Compared with the other distributed computation framework, MapReduce has the following advantages:

Algorithm 1 The DBSCAN algorithm

```
Input $(eps, minpts, D)$
Output (a set of clusters)
1.  initialize all points as unvisited
2.  for each unvisted point $p \in D$ do
3.    mark $p$ as visited
4.    Let $N$ be eps–neighborhood of $p$
5.    if the size of $N <$ minpts points then
6.      mark $p$ as noise
7.    else
8.      create a new cluster C, and add $p$ to C
9.      for each point $p' \in N$
10.       if $p'$ in unvisited then
11.         mark $p'$ as visited
12.         let $N'$ be the eps–neighborhood of $p'$
13.         if the size of $N'$ is $>=$ minpts then
14.           add those points to $N$
15.         endif
16.       endif
17.       if $p'$ is not yet a member of any cluster
18.         add $p'$ to C
19.       endif
20.     endfor
21.   endif
22. endfor
```

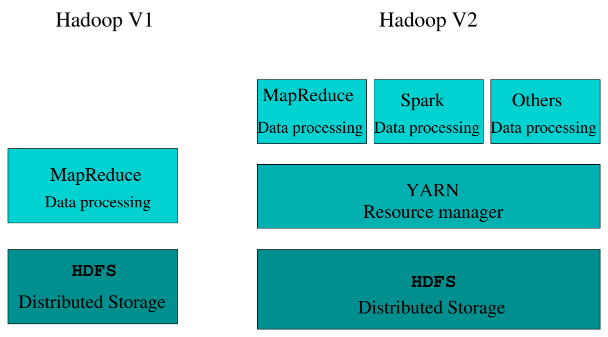


Figure 1: **An overview of Hadoop Architecture.**

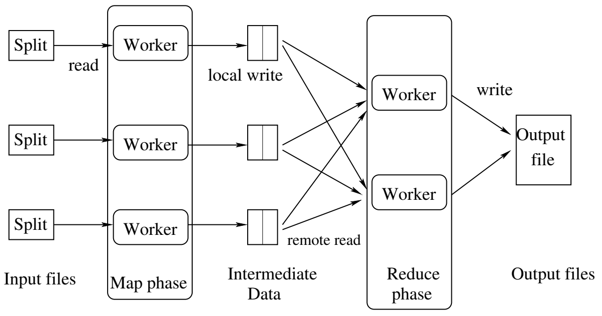


Figure 2: **An overview of data flow in MapReduce.**

- Extremely Scalable. It does not require the support from centralized RAID–based SAN or NAS storage systems. Every node has its own local hard–drives. The nodes are loosely coupled and connected with standard network devices. So adding and removing nodes to a cluster becomes very easy and convenient, and has no impact to running MapReduce jobs [17].
- Highly Parallel and Abstracted. Based on the framework's principle, programmers do not have to take care of low level implementation details such as message transferring between master and workers, file location, and workload balancing. They only need focus on the problem itself. One of the major contributions of MapReduce is that it supports parallelization automatically. The programmers only need to implement map() method of Mapper class and reduce() method of Reducer class and the framework will do the rest.
- Highly Reliable and Fault–tolerant. A single process failure in MPI will cause the whole job to fail. In MapReduce framework, another task will be automatically launched if one task fails and the job will continue running. This feature is especially useful and important for long–running jobs.

2) Spark: At a high level, a running Spark application has one driver process talking to many executor processes, sending them work to do and collecting the results of that work. The first thing a Spark program must do is to create a SparkContext object in driver code, which tells Spark how to access a cluster. Then it reads one file or multiple files in HDFS and processes them as Distributed Datasets (RDD), which is a collection of elements partitioned across the nodes and can be operated on in parallel. RDD is the main abstraction Spark provides, and RDDs can be created from a file in the Hadoop file system or by transforming other RDDs. An RDD can depend on zero or more other RDDs. These dependency relationships can be thought of as a graph. Stages are a unit of execution, and they are generated by the DAGScheduler from the graph of RDD dependencies. DAGScheduler computes a Directed Acyclic Graph (DAG) of stages for each job, keeping track of which RDDs and stage outputs are materialized. It then submits stages as TaskSets to TaskScheduler. TaskScheduler launches tasks to executors via Resource manager, which in this case, is YARN. After executors complete their tasks, they will send the results back to the driver (see Figure 3) (if it is the final RDD of an action such as count()) [12], or write output to external storage. Spark framework captures all the important features that MapReduce have. In addition, it has the following new features.

- In–memory computations. In Spark, Resilient Distributed Datasets (RDDs) are the first abstraction that allows programmers to perform in–memory computations on large clusters. RDDs are motivated by two types of applications that MapReduce handle inefficently: iterative algorithms and interactive data mining [22]. Figure 2 depicts that MapReduce frameworks does not fit iterative algorithms. In order to use MapReduce model to tackle iterative algorithms, many rounds of map–reduce executions will be performed which is not very efficient because map's intermediate results should be writen to local disks and then they are remotely read to reduce workers, and disk I/O operations are very expensive in this case. Another use case is interactive data mining, where a user runs the same queries on the same subset of data multiple times. In MapReduce framework, the only way to do that is to write intermediate data to an external storage file [22] which will introduce a lot of overhead due to disk I/O and data replication. In Spark framework, RDDs can be cached for re–use. In Spark, the benefit of keeping everything in memory is the ability to perform iterative computations at blazing fast speeds.
- Supporting Streaming data, complex analytics, and real time analysis. MapReduce offers a very simple but powerful programming model that are efficient for data–intensive algorithms [11]. But we can not use MapReduce to perform real time analysis and implementing complex graph based algorithms in an efficient manner.
- Fast fault recovery. In MapReduce old version, if the JobTracker does not receive any heartbeat from a TaskTracker for a specified period of time, the JobTracker understands that the worker associated to that TaskTracker has failed. When this situation happens, the JobTracker needs to reschedule all pending and in–progress tasks to another TaskTracker, because the intermediate data belonging to the failed TaskTracker may not be available anymore [21]. After hadoop–0.21, checkpointing was added where JobTracker records its progress in a file. When a JobTracker starts, it can restart work from where it left off. MapReduce uses replication strategy to handle fault recovery. On the other hand, Spark reconstructs RDDs via lineage to handle this issue. Compared to the replication method,

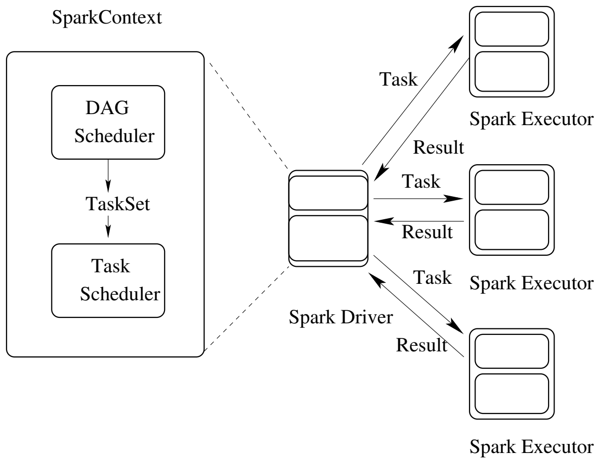


Figure 3: **An overview of data flow in spark.**

which consumes more memory, recontruction of RDDs takes shorter time [23].

Even though spark is very efficient, offers parallelization automatically, we still need to avoid shuffle operation. So in our implementation of DBSCAN we avoid all–to–all communication.

III. New DBSCAN with Spark

As far as we know, there are many DBSCAN implementations with Hadoop's MapReduce [9], [14], [7]. But we have not seen any implementation with Spark yet because of following reasons. First of all, compared with MapReduce, Spark appeared recently. So not many people are familiar with this framework. Secondly, even though Spark has many advatages compared to other distributed computation frameworks, the programmers still need to spend a lot of time on designing new algorithm to avoid shuffle operations to make their parallization more efficient. For example, after we update one data point's state in one executor we need to spread this updation across the cluster. So this will introduce shuffle operations which are very expensive in Spark. Let us take a look at the pseudocode of our new DBSCAN's algorithm.

A. Pseudocode of DBSCAN algorithm with Spark

The pseudocode of the DBSCAN algorithm with Spark implementation is given in Algorithm 2. The algorithm starts with the code in Spark driver, which reads data, generates RDDs and transforms them into appropriate RDDs (Line 1, Line 2, and Line 3). The code in Spark executor is in Lines 4 through 29. After comparing with Algorithm 1, we can see that two places are new: Line 12 and Lines 26 through 28. We assume each executor only computes the points that belong to it. Otherwise, there would be a lot of overlap of computation between different executors. Placing SEEDs is in Line 12. The detailed description regarding it

Algorithm 2 DBSCAN algorithm with Spark

```
Input($eps, minpts, D$)
Output (a set of clusters)
1. read an input file from HDFS and generate RDDs
from the read data
2. transform the existing RDDs into appropriate RDDs
with Point type
3. distribute those RDDs into executors
4. foreach (closure'start)
5.   if point $p$ is not in hashtable then
6.     get the neighbors of $p$ using eps and kdtree
7.     push all appropriate neighbors into Queue $N$
8.     if the size of $N <$ minpts points then
9.       mark $p$ as noise
7.     else
8.       create a new cluster C, and add $p$ to C
9.       while $N$ is not empty
10.        Let $p'$ be the removed point from $N$
11.        put the index of $p'$ into the hashtable
12.        place SEEDs processing

13.        if $p'$ in unvisited then
14.          mark $p'$ as visited
15.          let N' be the eps–neighborhood of $p'$
16.          if the size of N' is $>=$ minpts then
17.            add appropriate points to $N$
18.          endif
19.        endif
20.        if $p'$ is not yet a member of any cluster
21.          add $p'$ to C
22.        endif
23.      endwhile
24.    endif
25.  endif
26.  if curent point is the last one in closure then
27.    send partial clusters to driver through accumulator
28.  endif
29. endforeach
30. analyze partial clusters based on the placed SEEDs
31. search for all partial clusters and merge them if necessary
```

will be given in next Section. The partial clusters are sent back to driver right before the executor finishes its task by accumulator, which also will be explained in detail in the next Section. This implementation is meant for ensuring that merging process will not be started until all the executors finish their tasks. Lines 30 through 31 perform merging partial clusters and produce the final global clusters (see Algorithm 4). So the code [1–3] is run in Driver mode, code [4–29] is run in Executor mode, and code [30–31] is run in Driver mode.

B. Data Structures that having impact on performance

Using Java as the programming language in our imple–mentation, we need to consider using the appropriate data structures for efficiency. Here, two data structures Hashtable and Queue are discussed.

If we take a look at Line 11, this operation should be *put(key, value)*, which is usually $O(1 + n/K)$ where K is the hash table size. If K is large enough, the the result is effectively $O(1)$. Method *containsKey(key)* is performed in Line 5, Line 7, and Line 17. Again, under normal circumstances, it is $O(1)$. The add operations on Queue are performed in Line 7 and Line 17, and remove operation on Queue is performed in Line 10. The number of add operations should be the same as the number of remove operations according to the condition in Line 9 (while loop will not terminate until it is empty). Among LinkedList, ArrayList, and Vector, the best performance on both add and remove operations is obtained using LinkedList. In our code, we thus use LinkedList to implement Queue.

We have thus far presented the new DBSCAN algorithm in a nutshell in this section. Next, we present the details of the novelties of our new implementation in the next section.

IV. Critical Techniques in Parallel DBSCAN with Spark

In this Section, we will focus on the implementation details of our parallel DBSCAN algorithm with Spark, which make the parallel processing more efficient. The first part gives the pseudocode of algorithms. Then we briefly present two critical techniques in our implementation that make good use of the power from Spark framework. Lastly, we analyze the time complexity of the whole algorithm.

A. Novel clustering algorithm without communication between executors

According to the traditional method, we need to update data points' state by map function and then propagate this update to other executors. However, that implementation is considered not efficient because it will introduce a shuffle operation in order to make this update visible by other ex–ecutors. Therefore, we propose a novel clustering algorithm to get around the shuffle operation. After data points have been partitioned to each executor, we just let each executor compute the partial clusters locally for data points that are assigned to this executor. The merging process is deferred until all the partial clusters have been sent back to the driver. This new design, however, introduces new challenges: how to build the partial clusters in executors so that they can be merged in the driver? And how to identify those partial clusters which are supposed to be merged into one cluster? The pseudocode of algorithms and an example are given as follows.

Algorithm 3 gives the basic idea of our design. In order to avoid overlap of computation of partial clusters, we would let individual executors only deal with the points that belong to this partition so that the executors do not have to communicate to spread points' updated states across the clusters. However, we could not merge the partial clusters into the global clusters after all the partial clusters are collected in driver because there are no points that are shared between different partial clusters. Therefore, we introduce the term: SEEDs, which are points that do not belong to the current partition. And these SEEDs serve as something like markers so that we can easily identify outer master partial clusters by using them and merge them into a bigger cluster.The SEEDs are not related to the locations. If the current point's index is beyond the range of current partition it is taken as a SEED. So the main goal on executor side is to place SEEDs, and on driver side, we dig out SEEDs and identify master partial clusters and merge them.

Algorithm 3 Placing SEEDs in Executors

```
1. identify the current partition of this executor as par_A
2. initialize the place_flg for all the partitions
3.    while $N$ is not empty
4.       let $p'$ be the removed point from $N$
5.       put the index of $p'$ into the hashtable

6.       for $j = 0..$ all the possible partitions
7.          if $p'$ is in par_A
8.             let continue_flg be true
9.             break
10.         else
11.            if place one seed already
12.               let continue_flg to be false
13.               break
14.            else
15.               $place\_flg = 1$
16.               let continue_flg to be true
17.               break
18.            endif
19.         endif
20.      endfor
21.      if $continue\_flg = false$
22.         continue
23.      endif
24.      if place_flg is 1
25.         place a seed
26.      endif
27.   endwhile
```

Before moving on to the algorithm of digging out SEEDs from partial clusters in Spark driver, we would like to use an example to display how to identify SEEDs and search for master partial clusters. Figure 4a shows that there are 2 partial clusters from 2 partitions. SEEDs are those points whose indexes are beyond the partition's range. For example, for C[0], its range is from 0 to 2499. So the point whose

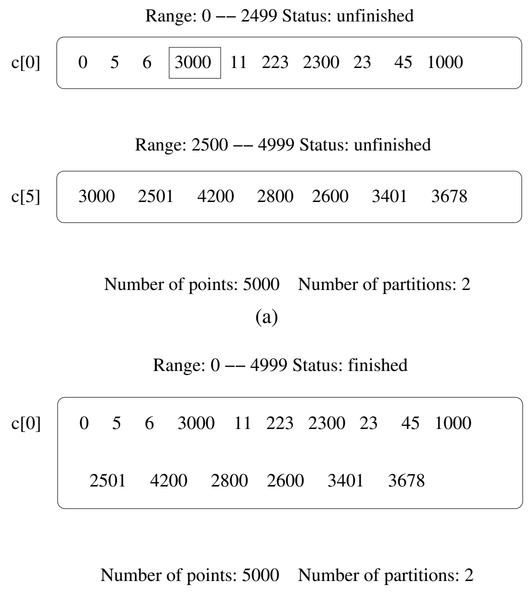

Range: 0 –– 2499 Status: unfinished
c[0]
0 5 6 3000 11 223 2300 23 45 1000
Range: 2500 –– 4999 Status: unfinished
c[5]
3000 2501 4200 2800 2600 3401 3678
Number of points: 5000 Number of partitions: 2
(a)
Range: 0 –– 4999 Status: finished
c[0]
0 5 6 3000 11 223 2300 23 45 1000
2501 4200 2800 2600 3401 3678
Number of points: 5000 Number of partitions: 2
(b)

Figure 4: **An example showing the proposed merging cluster algorithm at different stages.** (a) There are two partitions and two partial clusters. Integers in squares are SEEDs. (b) After C[0] merges C[5], C[0] status is updated as "finished" from "unfinished".

indexe is greater than 2499 is 3000. Then the algorithm will identify the master partial clusters. Obviously, for 3000, the master partial cluster is C[5] because it contains 3000, and 3000 is a regular element in this cluster. When we merge two partial clusters we need to remove duplicate elements. Figure 4b show the resulting cluster C[0].

In Spark driver, Algorithm 4 shows how to use SEEDs to merge partial clusters into global clusters. First of all, it identifies the SEEDs by comparing elements with its range. In general, the number of SEEDs should be equal to or greater than the number of partitions. So we obtain an array of seeds (see Line 3). Lines from 4 through 8 form a for loop, which finds the master cluster that contains the seed as a regular element, then merges the two clusters, and finally, updates the status of master cluster. When the for loop terminates the status of current cluster is updated from 'unfinished' to 'finished'.

B. Shared variables: broadcast and accumulator

Normally, when each executor performs computations remotely, it works on separate copies of all the variables sent by the driver. These variables are copies to each node, and no update to the variables are spread back to the driver program. So, read–write shared variables across tasks would be not so efficient [18]. Spark does provide two limited types of shared variables for two common usage patterns: broadcast variables and accumulators [18]. Broadcast variables are read–only and cached on each executor instead of being shipped to each task. It is very efficient to use broadcast variables to give every executor a copy of a large input dataset in an efficient manner. Spark distributes broadcast variables using efficient broadcast algorithm to reduce communication cost [18].

Algorithm 4 Using SEEDs and merge partial clusters in Driver

```
1. for $i = 0..$ all partial clusters
2.   if the status of current partial cluster is 'unfinished'
3.     seed = identify seeds from current partial cluster
4.     for $j = 0..seed.size()$
5.       rtn_index = find master partial cluster index
6.       merge current with master cluster
7.       update the status of master cluster to 'finished'
8.     endfor
9.     update the status of current cluster to 'finished'
10.  endif
11. endfor
```

As we know in our DBSCAN algorithm, it is necessary for executors to know some parameters and variables, such as eps, minimum number of points, partition information, and especially, the kdtree. With this kind of information, executor can successfully compute the partial clusters without exchanging message with other executors. When we are broadcasting large numbers of bytes, optimizing broadcasts is essential, such as choosing an appropriate data serialization format that is both fast and compact, and compression techniques [12].

Another important shared variable is accumulator which meets our other needs. Accumulators are variables that are only "added" to through an associative operations and be efficiently supported in parallel. Originally, it has been used widely to implement counters (as in MapReduce) or sums. Because it can be used as "Writable" variables in executors, we use it to implement bringing back the partial clusters. That is another important feature of our design.

C. Time complexity analysis

We first analyze the time complexities of both sequential and parallel algorithms. The speed–up of the parallel method is then derived. We define some related notations as follows:
n : the number of data points;
p : the number of partitions;
m : the number of partial clusters;
K : the maximum size of partial clusters;
$t^{ave}_{straggling}$: the average wait time for framework to allow all stragglers to finish.
T^{ave}_{s} : the average time complexity of the sequential algorithm;
T^{ave}_{p} : the average time complexity of the parallel algorithm;

S_{ave} : the average speed–up.

Basically, there are three parts in our algorithm.

In the first part, the driver transforms data from HDFS into appropriate form that can be used in executors and constructs the $kd-tree$. The time for this phase includes reading file, transforming RDDs, and building $kd-tree$. We assume we use Δ for the first two items. For $kd-tree$ construction, we use $O(nlogn)$ [10]. So summing them up, we use $\Delta + O(n*logn)$.

In the second part, all executors compute the local partial clusters based on the partitions they receive. Basically, searching a point from a balanced $kd-tree$ takes $O(logn)$ time, which is best case. In the worst case, the time could be n. Some researches have reported that (near neighbor) range search's upper bound is $O(n^{1-1/d}+k)$ [10]. So we use V to represent the search time, which is between $logn$ and $n^{1-1/d}+k$; If we use parallel processing, we need to add the time for SEEDs placement part. Let us assume an additional $O(m*V)$ time is added. So in parallel processing, we would spend $O((n/p*V)+(m*V))+t^{ave}_{straggling}$ time in our case.

In the last part, after all executors send back all the partial clusters to the driver, the driver merges them and produces the global clusters. Based on our Algorithm 4, the search operations takes $O(n)$ time at most if we check each element in the partial clusters. For merging phase, it takes Km times which is less than n. So we use $O(n+Km)$ time.

To sum up:
$T^{ave}_s = O(\Delta + n*logn + n*V + n + Km)$.
$T^{ave}_p = O(\Delta + n*logn + n/p*V + m*V + t^{ave}_{straggling} + n + Km)$.
$S_{ave} = T^{ave}_s / T^{ave}_p$.

V. EXPERIMENTS AND RESULTS

We perform a series of experimental tests to verify the effectiveness and efficiency of our DBSCAN algorithm with Spark and MapReduce's implementations. We need to note that all parallel executions generate the same result as the serial execuation. The dimension of data is relevent to the computational cost of querying the $kd-tree$. We do not perform tests based on varying number of attributes because we focus on Spark implementation instead of $kd-tree$ implementation in our work. The tests are done on different sizes of data points with multiple dimensions. Our experi–mental results have been reported in terms of the CPU times. After comparing with the results from Patwary *et al.* [15], we find that our results match them so we do not list the accuracy in our paper.

A. Experimental setup

To perform the experiment for our DBSCAN's parallel implementation with Spark, we use Edison (operated by Lawrence Berkeley National Laboratory and the Department of Energy Office of Science), a Cray XC30 distributed memory parallel computer. It has 5,576 compute nodes, 133,824 cores in total. Each node has two 12–core Intel "Ivy Bridge" processors at 2.4 GHz and 64 GB DDR3 1866 MHz memory. Each core has its own L1 and L2 caches, with 64 KB (32 KB instruction cache, 32 KB data) and 256 KB, respectively; A 30–MB L3 cache shared between 12 cores on the "Ivy Bridge" processor [5]. The algorithms have been implemented in Java (1.7) using the Spark(1.5) and Hadoop (2.4).

Table I: Properties of test data

Name	Points	d	eps	minpts
c10k	10,000	10	25	5
c100k	102,400	10	25	5
r10k	10,000	10	25	5
r100k	102,400	10	25	5
r1m	1,024,000	10	25	5

Our testbed consists of 5 datasets, which are divided two groups: ($c10k, c100k$), and ($r10k, r100k, r1m$). Both groups of datasets (*synthetic-cluster*) have been generated synthetically using the IBM synthetic data generator [1], [16]. Table I lists the properties of our test data.

B. Construction of $kd-tree$

As discussed in Section IV, we use $kd-tree$ ([2], [3]) to reduce the running time of the DBSCAN algorithm from $O(n^2)$ (naive linear search) to V which is from $O(nlogn)$ to $O(n^{1-1/d}+k)$ (where d is the dimension size of record, and k is number of reported neighbors) [10]. In our im–plementations, we used $kd-tree$ and therefore obtained reduced time complexities. Even though there is an overhead in contructing the $kd-tree$ before running the DBSCAN with Spark, we would see that we benefit a lot. We use the similar approach as the one used by Patwary [15]. Figure 5 gives a comparison of the time taken by building the $kd-tree$ over the whole DBSCAN algorithm in percentage. It is seen that, the contruction of $kd-tree$ only takes a very tiny fraction of the time (0.05% to 0.5%) w.r.t. the whole DBSCAN algorithm. We noticed that the percentages of construction of tree is higher for r10k and c10k. This is because these two data sets consist of small number of points, and so, the whole algorithm takes shorter time.

C. Comparison of the time spent in driver and in executors

In this part, we discuss the time distribution in our program. Figure 6a –Figure 6d shows the time distribution between executors and driver according to our experiments. Based on the Algorithm 2, we expect to see more time will be spent in driver with the number of partial clusters increasing. Let us take a look at Figure 6a first. When we use more cores (1 to 8) to run our program, we see the number of partial clusters becomes bigger (10 to 392), but the time spent in driver does not change very much.

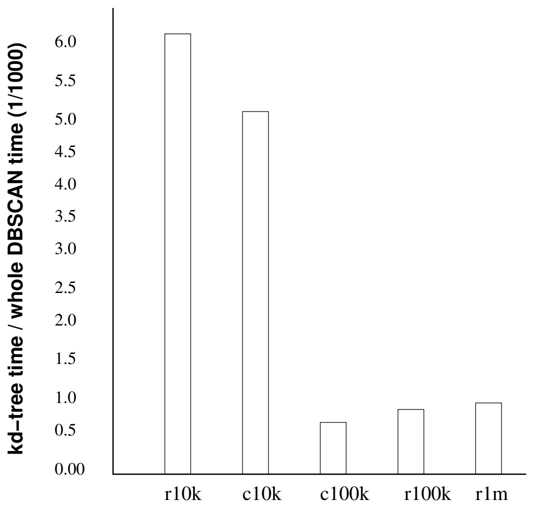


Figure 5: **Time taken by the construction of $kd-tree$ and the whole DBSCAN algorithm.** Notes: the whole times we use is when we use 8 partitions

That is because the data set is too small. Take a look at Figure 6c and Figure 6d, their patterns are exactly the same. When using more cores (4 to 32), more partial clusters are produced (from 720 to 9279), and the time spent in driver gradually becomes more. This is consistent with our analysis on the time complexity that we conduct in Section IV, where when the number of partial clusters m increases, the time $n + Km$ becomes large as well. Figure 6b follows the complexity analysis as well.

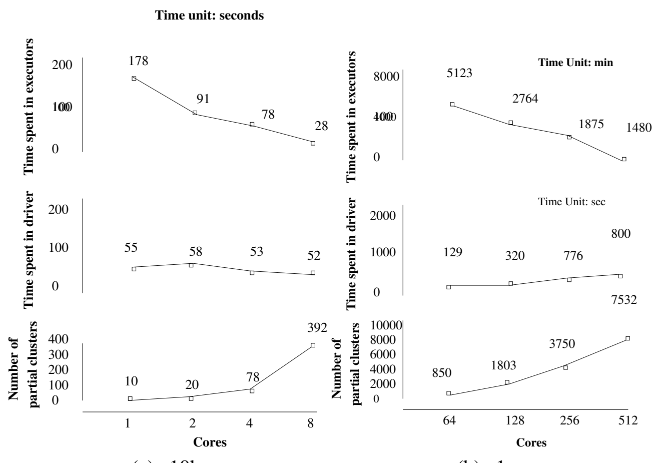
D. Comparison of the time taken by MapReduce and Spark

As we are not able to get source code from the other research teams [9], [7], [14], we have implemented our own DBSCAN with MapReduce approach. From Figure 7, it is seen that 9–16 times faster performance is obtained from Spark than MapReduce. Due to the length of time taken by MapReduce, we have not conducted further tests on medium scale and large scale data sets.

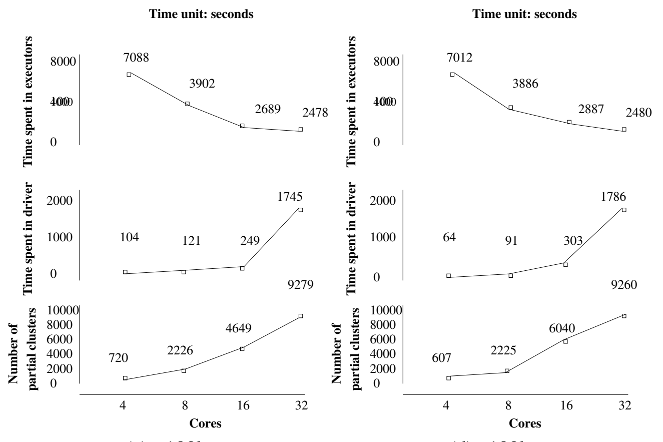
E. Scalability of Parallel DBSCAN with Spark

Before we discuss the scalability of our algorithm, we need to mention that for large data sets ($>= 1$ million data points), we use $kd - tree$ with pruning branches to shorten search time.

The speedup obtained by our DBSCAN algorithm with Spark is given in Figure 8. The left column in Figure 8 shows the speedup considering only the computation in executors while the right column shows the results considering the computation in executors and driver. It is obvious that the local computation in executors scales better than the whole computation since their computations are independent. For 10k data sets, we obtain speedup upto 1.9, 3.6, and 6.2 respectively using 2, 4, and 8 cores. For 100k data sets,



(a) r10k. (b) r1m.



(c) c100k. (d) r100k.

Figure 6: The time distribution between driver and executors

speedup upto 3.3, 6.0, 8.8, and 10.2 respectively using 4, 8, 16, and 32 cores. For 1m data set, speedup upto 58, 83, 110, and 137 respectively using 64, 128, 256, and 512 cores.

Take a look at right column, Figure 8b, Figure 8d, and Figure 8f show the speedup when total time is considered. The curves seem more flat compared with the ones in left column. For 10k data sets, because the total time is less, the merging time is not significant. For 100k data sets, more partial clusters are collected in driver. When using 4, 8, and 16 cores, the local computation time still dominates the total time, so speedup does not change very much. When using 32 cores, 9279 partial clusters are generated in executors and collected in driver. So the speedup drops to 5.6 from 10.2.

For r1m, we use pruning branches technique, and thus the neighbor size of each point is decreased. Also we filter out those partial clusters whose size is too small, and their removal does not impact the accuracy significantly. Therefore, the speedup of total time does not change a lot compared with local computation.

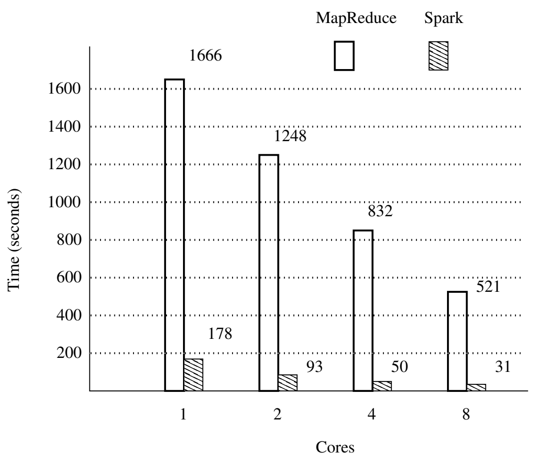


Figure 7: **Time used by MapReduce and Spark.** Number of poinst: 10000, dimension: 10, eps, 25.0, minPnts: 5

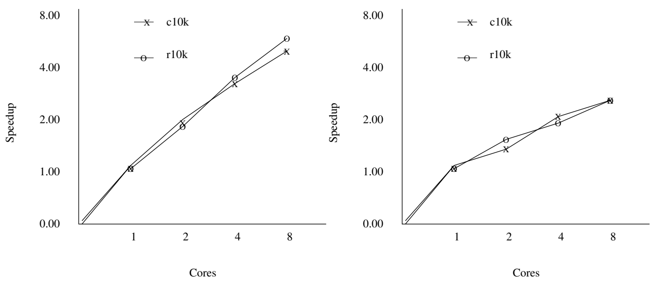
VI. Conclusions

DBSCAN algorithm has been very popular since it is able to identify arbitrary shaped clusters as well as handle noisy data. However, parallelization of DBSCAN based on MPI and OpenMP suffers from lack of fault-tolerance. Moreover, in order to implement parallelization with MPI or OpenMP, data scientists need to take care of implementation in detail, such as handling communication, dealing with synchronization, and so forth, which can pose a challenge for many users. In this paper, we presented a new Parallel DBSCAN algorithm with Spark. It avoids the communication between executors and thus leads to a better scalable performance. The results of these experiments demonstrate that our new DBSCAN algorithm with Spark is scalable and outperforms the implementation based on MapReduce by a factor of more than 10 in terms of efficiency.

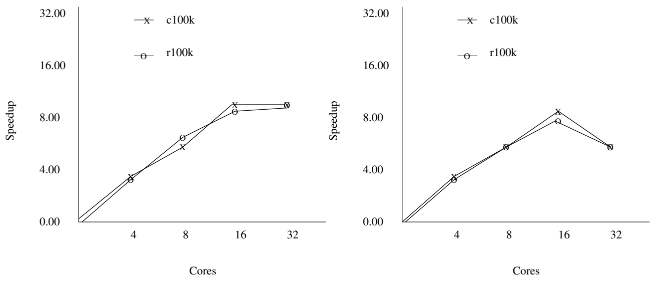
Future research will be conducted to improve search efficiency of $kd - tree$ which has an important impact on the performance of our algorithm. Also, we plan to perform extended experiments on larger data sets and high dimensional feature spaces will be investigated as well. We did not partition data points based on the neighborhood relationship in our work and that might cause workload to be unbalanced. So, in the future, we will consider partitioning the input data points before they are assigned to executors.

Acknowledgment

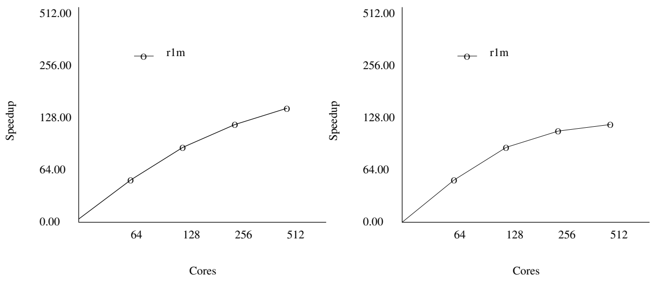
This work is supported in part by the following grants: NSF awards CCF-1409601, IIS-1343639, and CCF-1029166 ; DOE awards DESC0007456 and DE-SC0014330 ; AFOSR award FA9550-12-1-0458; NIST award 70NANB14H012. This research used Edison Cray XC30 computer of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.



(a) Comp. in executor. (10k points) (b) Comp. in executor and driver. (10k points)



(c) Comp. in executor. (100k points) (d) Comp. in executor and driver. (100k points)



(e) Comp. in executor. (1m points) (f) Comp. in executor and driver. (1m points)

Figure 8: Speedup of DBSCAN algorithm with Spark. Left side: Time spent in executor. Right side: Time spent in driver and executor.

References

[1] R. Agrawal and R. Srikant, "Quest synthetic data generator," *IBM Almaden Research Center,* 1994.

[2] N. Beckmann *et al.*, "The r*-tree: an efficient and robust access method for points and rectangles," in*Proceedings of the 1990 ACM SIGMOD international conference on Management of data,* vo. 19, no. 2, pp. 323-331, 1990.

[3] J. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM,* vol. 18, no. 9, pp. 509-517, 1975.

[4] S. Brecheisen *et al.*, "Parallel Density-Based Clustering of Complex Objects," *Advances in Knowledge Discovery and Data Mining,* pp. 179-188, 2006.

[5] DOE Office of Science (2015, September 17). *Edison Configuration* [Online]. Available: https://www.nersc.gov/users/computational–systems/edison/configuration/

[6] M. Ester *et al.*, "A Density–Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining,* vol. 1996, AAAI Press, 1996, pp. 226–231.

[7] Y. Fu *et al.*, "Research on Parallel DBSCAN Algorithm Design Based on MapReduce," *Advanced Materials Research,* vol. 301, pp. 1133–1138, 2011.

[8] J. Han *et al.*, *Data mining: concepts and techniques.* Morgan Kaufmann, 2011

[9] Y. He *et al.*, "MR–DBSCAN: a scalable MapReduce–based DBSCAN algorithm for heavily skewed data," *Frontiers of Computer Science*, vol. 8, no. 1, pp. 83–99, 2014.

[10] H. M. Kakde (2005, August 25). *Range Searching using Kd Tree* [Online]. Available: http://www.cs.utah.edu/ lifeifei/cs6931/kdtree.pdf

[11] S. J. Kang *et al.*, "Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems," *Advances in Multimedia.* vol. 2015, 2015.

[12] H. Karau *et al.*, *Learning Spark: Lightning-fast Data Analysis.* O'Reilly Media, 2015.

[13] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability,* vol. 1. USA, 1967, pp. 281–297.

[14] M. Noticewala and D. Vaghela, "MR–IDBSCAN: Efficient Parallel Incremental DBSCAN Algorithm using MapReduce," *International Journal of Computer Applications,* vol. 93, no. 4, pp. 13–17, 2014.

[15] M. M. A. Patwary *et al.*, "A new scalable parallel DBSCAN algorithm using the disjoint–set data structure," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012*, IEEE Computer Society Press, pp. 62:1–62:11, 2012.

[16] J. Pisharath *et al.*, *NU-MineBench 3.0.* Technical Report CUCIS–2005–08–01, Northwestern University, Tech. Rep., 2010.

[17] S. Sakr and M. M. Gaber, *Large Scale and Big Data: Processing and Management.* CRC Press, 2014

[18] Apache Spark (2015). *Spark Programming Guide* [Online]. Available: http://spark.apache.org/docs/latest/programming–guide.html

[19] G. Sheikholeslami *et al.*, "WaveCluster: A Wavelet Based Clustering Approach for Spatial Data in Very Large Databases," The VLDB Journal, vol. 8, no. 3, pp. 289–304, 2000.

[20] P. Tan *et al.*, *Introduction to Data Mining.* Pearson, 2005

[21] T. White, *Hadoop: The Definitive Guide.* O'Reilly Media, 2011

[22] M. Zaharia *et al.*, "Resilient Distributed Datasets: A Fault–Tolerant Abstraction for In–Memory Cluster Computing," in*Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012 pp. 2–2.

[23] M. Zaharia, *An Architecture for Fast and General Data Processing on Large Clusters.* Technical Report UCB/EECS–2014–12, University of California, Berkeley, Tech. Rep., 2014.

[24] T. Zhang *et al.*, "BIRCH: an efficient data clustering method for very large databases," in *ACM SIGMOD Record,* vol. 25(2). ACM, 1996, pp. 103–114.

[25] A. Zhou *et al.*, "Approaches for Scaling DBSCAN Algorithm to Large Spatial Databases," in *Journal of computer science and technology,* vol. 15, no. 6, 2000, pp. 509–526.