remote sensing

MDPI

Article

Research on the Parallelization of the DBSCAN Clustering Algorithm for Spatial Data Mining Based on the Spark Platform

Fang Huang [1,2,*], Qiang Zhu [1], Ji Zhou [1], Jian Tao [3], Xiaocheng Zhou [4], Du Jin [1], Xicheng Tan [5] and Lizhe Wang [6,7,*]

1 School of Resources & Environment, University of Electronic Science and Technology of China, 2006 Xiyuan Ave., West Hi-Tech Zone, Chengdu 611731, China; jangzhu@163.com (Q.Z.); jzhou233@uestc.edu.cn (J.Z.); ikingdonblue@gmail.com (D.J.)
2 Institute of Remote Sensing Big Data, Big Data Research Center, University of Electronic Science and Technology of China, 2006 Xiyuan Road, West Hi-Tech Zone, Chengdu 611731, China
3 Texas A&M Engineering Experiment Station and High Performance Research Computing, Texas A&M University, College Station, TX 77843, USA; jtao@tamu.edu
4 Key Laboratory of Spatial Data Mining & Information Sharing of Ministry of Education, Fuzhou University, No. 2 Xueyuan Road, Fuzhou University New District, Fuzhou 350116, China; zhouxc@fzu.edu.cn
5 International School of Software, Wuhan University, 129 Luoyu Road, Wuhan 430079, China; xctan@whu.edu.cn
6 School of Computer Science, China University of Geosciences, Wuhan 430074, China
7 Institute of Remote Sensing and Digital Earth, Chinese Academy of Sciences, Beijing 10094, China
* Correspondence: fang.percy.huang@gmail.com (F.H.); lizhewang@icloud.com (L.W.)

Received: 27 October 2017; Accepted: 8 December 2017; Published: 12 December 2017

Abstract: Density-based spatial clustering of applications with noise (DBSCAN) is a density-based clustering algorithm that has the characteristics of being able to discover clusters of any shape, effectively distinguishing noise points and naturally supporting spatial databases. DBSCAN has been widely used in the field of spatial data mining. This paper studies the parallelization design and realization of the DBSCAN algorithm based on the Spark platform, and solves the following problems that arise when computing macro data: the requirement of a great deal of calculation using the single-node algorithm; the low level of resource-utilization with the multi-node algorithm; the large time consumption; and the lack of instantaneity. The experimental results indicate that the proposed parallel algorithm design is able to achieve more stable speedup at an increased involved spatial data scale.

Keywords: spatial data mining; DBSCAN algorithm; parallel computing; spark platform; traffic congestion area discovery

1. Introduction

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is an algorithm proposed by Ester et al. for clustering analyses based on the density method in 1996 [1]. It has features like the abilities to find clusters of any shape, identify noisy points effectively, and support spatial databases. It has been widely used in the field of data mining. However, in some large-scale spatial data mining studies [2], as the scale of data computation increases, the processing time of DBSCAN rises exponentially. Thus, the performance of this serial algorithm is unable to meet the needs of real-time application development.

To speed up the performance of serial DBSCAN processing, on one hand, some researchers began to improve and optimize the serial algorithm; on the other hand, the DBSCAN clustering algorithm has also been parallelized on computer clusters, graphics-processing units (GPUs), and the Hadoop platform to improve its efficiency. However, these approaches have the following disadvantages:

(1) The traditional parallel-processing platform is expensive and has inferior scalability and fault-tolerance properties, which result in bottlenecks in data transfer.
(2) When dealing with a multi-iterative clustering algorithm such as DBSCAN, Hadoop needs to read and write data frequently from/to the distributed file system. The efficiency of the system decreases as the amount of data increases.

To resolve the problem of long processing times associated with large-scale data processed using the serial DBSCAN algorithm, in this paper, the big data processing platform Spark was introduced to design and implement a parallel DBSCAN clustering algorithm. Spark, as a new generation of fast general-purpose engine for large-scale data processing, provides resilient distributed dataset (RDD) abstraction for data storage that eliminates the need for intermediate results to be sent to the distributed file systems, and hence it improves real-time data processing. Spark has features such as a large degree of scalability and high fault tolerance, which overcome the problems of the aforementioned parallel platforms. Based on the Spark platform, which was designed and optimized for memory usage and iterative processes, this paper studies parallelization strategies for the DBSCAN algorithm. Historically, the DBSCAN algorithm was designed and optimized on the single-node Spark platform and the Spark cluster platform based on different resource managers. Therefore, the research presented in this paper helps to improve the efficiency of the DBSCAN algorithm and generate new ideas for the parallelization of other types of spatial clustering algorithms, and to a certain extent, promotes the organic integration of the Spark platform with traditional spatial data mining technologies.

In addition, we utilized the parallel DBSCAN algorithm based on the Spark platform in a traffic congestion area discovery application. Compared with traditional processing methods, this approach can dramatically improve the efficiency of city congestion discovery. These experiments verify the high efficiency and practicability of the parallel algorithm proposed in this paper.

The paper is organized as follows. Section 2 presents related works, and conducts a systematic analysis. Section 3 gives a brief introduction to the DBSCAN algorithm, including its mathematical principles, and the implementation of serial DBSCAN algorithm. Section 4 concentrates on the implementation of the parallel DBSCAN algorithm and its optimization methods. The subsequent section focuses on the test results of the parallel algorithm on both the single node platform and the Spark cluster with different resource managers. The experimental results are systematically analyzed thereafter. Section 6 discusses the application of the parallel algorithm for congested area discovery in urban traffic. Finally, Section 7 draws some conclusions and discusses future research directions.

2. Related Works

To shorten the elapsed time taken for the serial DBSCAN processing, researchers began to improve and optimize some selected serial algorithms. Based on DBSCAN, Ankerst et al. proposed the ordering points to identify the clustering structure (OPTICS) algorithm to solve the problem of parameter selection that affects the algorithm. However, the OPTICS algorithm is complicated, and requires additional I/O operations. As a result, its performance is lower than that of DBSCAN [3]. Chen et al. designed the spatial access algorithm to improve DBSCAN, where the complexity of the algorithm was reduced from $O(n^2)$ to $O(n\lg n)$, which to a certain extent improved the efficiency of the algorithm [4]. Kryszkiewicz et al. proposed a method that uses the triangle inequality to reduce the neighboring search space. The search time was thus reduced, and the efficiency of the algorithm was improved [5].

Although optimization of the serial DBSCAN algorithm itself can improve its efficiency to some extent, with the geometric growth of the amount of data processed, this serial algorithm cannot meet real-world requirements. In recent years, due to its powerful computing capabilities, GPU-based computing, and the technology of cloud computing, etc. have enabled major breakthroughs in parallel computing performance [6,7]. GPU-based computing has advantages in processing intensive matrix operations, and it has been widely used in some compute-intensive applications [8–16]. Cloud computing has performance advantages in data processing, and the MapReduce model has been used widely on cloud computing platforms in many data-intensive applications [17–20]. In order to obtain more efficient processing capabilities, researchers also began to parallelize the clustering algorithms to improve the performance. The parallelization of clustering algorithms at home and abroad are mainly based on high-performance computing cluster platforms, GPU platforms, and Hadoop platforms. Relevant details include:

(1) Parallelization on high-performance cluster platforms. Xu et al. proposed a network-based fast parallel-DBSCAN (PDBSCAN) algorithm for use on cluster systems with master-slave architectures. First, the entire data area is divided into N disjointed data blocks, where each data block is processed by each computing node. After a given data block has been processed, the results are merged on the master node. This parallel algorithm requires extra communication time, where the inefficiencies related to combine the data with the main node data reduce the overall processing efficiency of the algorithm [21]. Based on the data-partition DBSCAN algorithm, Erdem & Gündem used high-performance clusters to improve the efficiency of the algorithm [22]. In these studies, parallel platforms based on high-performance clusters can help to improve the algorithm, but the conventional cluster platform has issues such as poor scalability, bad fault tolerance, etc. The shared architecture is also likely to cause bottlenecks in data transfer.

(2) Parallelization on GPU platforms. Böhm et al. proposed a GPU-based CUDA-DClust parallel algorithm that calculates the distance from the central point to surrounding points using the multi-thread programming model to efficiently query the neighboring data. The results show that the parallel algorithm enhances the overall efficiency with the help of the GPU [23]. However, CUDA-DClust needs to calculate the distance between many unnecessary objects and store them on the device memory. Andrade et al. used the compute unified device architecture (CUDA) on the GPU to accelerate the parallel algorithm [24]. Compared with the serial algorithm, the performance was greatly improved.

(3) Parallelization on Cloud-computing platforms. Dean et al. proposed to take advantage of MapReduce, a large-scale data processing programming model, to enhance the processing efficiency [25]. Böse et al. designed data-mining algorithms based on the MapReduce programming framework, and they demonstrated the feasibility of adopting the MapReduce programming model in data mining [26]. He et al. implemented the MR-DBSCAN algorithm with MapReduce, and achieved good scalability and speedup [27]. Dai et al. divided the data by reducing the number of boundary points, and they used KD-Tree spatial indices to parallelize DBSCAN on Hadoop to improve the performance of the algorithm [28]. Fu et al. studied the parallel DBSCAN clustering algorithm based on Hadoop. The experimental results demonstrated that the proposed algorithm could efficiently process large datasets on commodity hardware and have good scalability [29]. In addition, some researchers studied the parallel implementation of K-Means algorithm on Hadoop to improve the efficiency [30,31]. In these studies, the efficiency of the parallel algorithms were improved on Hadoop, but the results were less than ideal when processing massive spatial datasets [30–32]. The reasons for this were: (1) the startup time in MapReduce increased the overall run time of the jobs; (2) the frequent I/O carried out by the Hadoop distributed file system (HDFS) for the intermediate results, due to its fault-tolerance mechanism; and (3) with an increase in the data size, the processing efficiency was significantly decreased [33].

As the Hadoop platform cannot meet the needs of some specific real-time massive data processing, the AMPlab team at the University of California developed the Spark platform, which is a distributed computing framework based on the Hadoop MapReduce framework [34]. One of the major features of Spark is that it inherits the merits of Hadoop MapReduce while using a global cache mechanism so that intermediate results are stored in memory, requiring less frequent data I/O and effectively improving the processing speed [35]. Lawson used Spark to develop financial data processing applications and implemented the parallel alternating direction method of the multipliers algorithm in Spark. The experimental results show that the parallel algorithm was able to process millions of rows of data in only 520.1 s, which was a significant improvement in efficiency [36]. Lipka implemented a strategic iteration algorithm on Hadoop and Spark. The results showed that the processing speed with Spark was 71% higher than that with Hadoop. The more computing nodes Spark used, the less execution time the algorithm took [37]. Wang et al. studied the parallelization of the K-means algorithm on the Spark platform. By comparing the performance of the parallel and serial algorithms on the Spark platform, they presented their parallel algorithm run on the Spark platform for massive data processing and demonstrated improved performance [38]. Jiang et al. studied the parallel FP-Like algorithm in Spark. The parallel algorithm achieved better scalability and acceleration results, and helped to demonstrate the advantages of Spark vs. Hadoop compared with the parallel Apriori algorithm [39]. Jin et al. used Spark in the field of land-use analysis. Compared to the traditional superposition method, their method significantly improved the efficiency of the analysis [40]. Xie et al. studied different data index methods based on Spark. Their implementation effectively improved the index query, while supporting spatial data management [41]. To date, there have not been many studies on the parallelization of the DBSCAN spatial clustering algorithm in data mining with Spark.

In summary, the advent of the Big Data era [42] has resulted in a large amount of noisy spatial data. Traditional standalone systems, distributed cluster platforms, and Hadoop-distributed platforms cannot meet the development requirements of spatial data mining. Therefore, based on Spark, this paper studies parallelization strategies for the DBSCAN algorithm. Furthermore, the DBSCAN algorithm can be applied in many fields, including urban planning [43], hotspot clustering [44], anomaly detection [45], etc. In particular, there are many applications in the transportation area that use the DBSCAN algorithm. For instance, Silva et al. used the DBSCAN algorithm to process the real traffic data of Fortaleza to find the congested areas of the city [46]. Through a large amount of taxi trajectory data, one can quickly understand the dynamical distribution of vehicles and find congested areas in urban traffic. For example, Adiba et al. proposed an approach using the DBSCAN clustering method to identify the traffic congestion regions and their spatio-temporal distribution with the taxi trajectory data [47]. Wang et al. used their improved C-DBSCAN algorithm to deal with GPS trajectory data to discover the candidate locations for bus stops near the Capital International Airport in Beijing in 2012 [48]. Liu et al. divided the taxi trajectory data into time slices, finding the traffic-jam areas in some parts of Wuhan after using the DBSCAN algorithm to process the corresponding data [49].

3. DBSCAN Algorithm

In order to parallelize an algorithm, it is crucial to understand the working principles and implementation of the original serial algorithm, and carry out detailed analyses and tests first. After that, one can take different approaches to parallelize it and then adopt the parallel algorithm in applications.

3.1. Mathematical Principles of the DBSCAN Algorithm

Let the dataset to be processed be denoted as D, the algorithm's clustering radius, Eps, and the minimum number of objects in the neighborhood, $MinPts$. Then, the following are some basic concepts of the algorithm [1]:

(1) Eps neighboring area: let p be the center of a sphere in the dataset D. For data within the radius Eps of the object's area, a collection of points contained in the sphere is $N_{Eps}(p) = \{q \in D \mid dist(p, q) \leq Eps\}$. A definition diagram is shown in Figure 1.

(2) Density: at the position of data point p in the dataset D, the number of points, Num, contained in the neighborhood Eps is its density.

(3) Core point: at the position of data point p in the dataset D, if the density (Num) in the neighborhood Eps satisfies $Num \geq MinPts$, it is called a core point.
(4) Border point: at the position of data point p in the dataset D, if the density in the neighborhood Eps satisfies $Num \leq MinPts$ but it is inside the sphere, it is called a border point.
(5) Noise point: all the objects other than the core and border points in D.
(6) Direct density-reachable: given objects p, $q \in$ D, where there is a core point and this is inside the Eps neighborhood of q, it is said that from p to q is direct density-reachable, i.e., $q \in N_{Eps}(p)$, $|N_{Eps}(p)| \geq MinPts$, as shown in Figure 2.
(7) Density-reachable: given objects $p_1, p_2, p_3, p_4, \ldots . p_n \in D$, where $p_1 = q$, $p_n = q$, if p_{i+1} is directdensity-reachable from p_i, then p is density-reachable from q. The definition is shown in Figure 3.
(8) Density-connected: given objects p, $q \in$ D, if there is a point $o \in$ D that is density-reachable from p and q, then p and q are density-connected. The definition is shown in Figure 4.

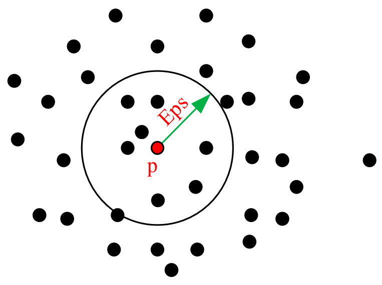


Figure 1. Neighborhood definition.

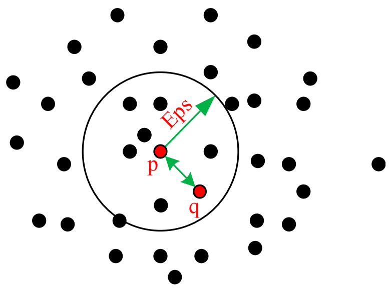


Figure 2. Definition of direct density-reachable.

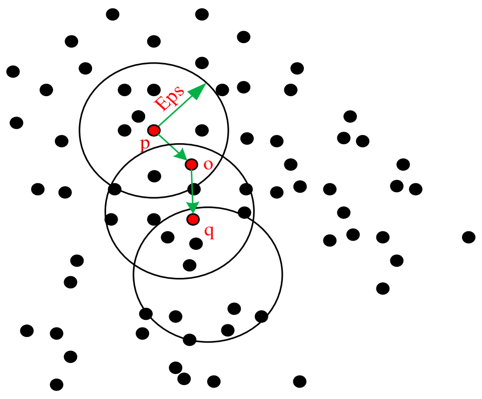


Figure 3. Definition of density-reachable.

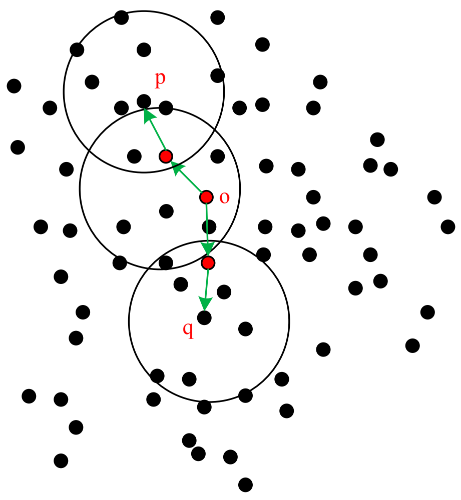


Figure 4. Definition of density-connected.

Based on the aforementioned definitions, the idea of the DBSCAN algorithm is as follows. The search can start from the neighboring Eps area of any interested data point. Given enough data points in the neighborhood ($\geq MinPts$) the cluster will expand. Otherwise, a data point is temporarily marked as noise. This point can later be found in other Eps neighborhoods and marked as part of a cluster. If a data object in a cluster is marked as a core, its Eps neighborhood is also part of the cluster. Thus, all the points found in the neighborhood, as well as the core neighborhood, are added to the cluster. This process is repeated until density-connected clusters are completely found. Finally, new and untreated points are retrieved and processed to find deeper clusters or noise. After all the objects in dataset D are checked, the algorithm ends.

3.2. Processing Procedure of the DBSCAN Algorithm

Based on the previous description of the basic concepts and ideas of the DBSCAN algorithm, its processing flow can be summarized in the following steps. Here, supposing the spatial dataset D, given clustering radius Eps, the minimum number of neighboring objects $MinPts$, and the current collection of objects as N_1.

(1) All the data objects in dataset D are marked as unchecked. Starting from any unchecked data point p, mark it as 'checked', then check its Eps neighborhood and calculate the number of objects in the neighborhood m. If m satisfies $m \geq MinPts$, then create a new cluster C_1, and add p to C_1, meanwhile add all the points in the neighborhood to the collection of objects N_1.
(2) For the collection of objects, N_1, if object q therein has not been checked, then mark q as 'checked', and then check its Eps neighborhood and calculate the number of objects in the neighborhood n. If n satisfies $n \geq MinPts$, then these objects are added to the object collection. If q does not belong to any cluster, then add q to C_1.
(3) Repeat step (2), and continue to check object set N_1 until it is empty.
(4) Repeat steps (1) to (3). When all the data objects are marked as 'checked', the algorithm ends.

The detailed implementation steps can be summarized in Algorithm 1.

Algorithm 1. DBSCAN algorithm implementation steps.

```
Input Data: Data Set D to be processed
Output Data: Cluster that satisfies the Clustering requirements
Parameters: Clustering Radius Eps, Minimal number of neighboring points MinPts
    Main function of the algorithm:
DensityCluster(D, Eps, MinPts)
{
    ClusterNum = 0                                      // Initialize cluster
    for each unchecked point M in D // Traverse all the unchecked points
        set M as checked                                // Mark it checked
        NeighbourResult = NeighbourSearching(M, Eps) // Search the neighborhood of M
        if sizeof(NeighbourResult) ≥ MinPts             // Mark it as a core and create a cluster
ClusterNum = UpdateClusterNum                          // Expand the cluster
            expandNeighbourPart(M, NeighbourResult, ClusterNum, Eps, MinPts)
        else                                            // If it is not a core
            set M as NOISE                              // Mark it as noise
}

Sub function:
expandNeighbourPart(M, NeighbourResult, ClusterNum, Eps, MinPts)
                                                       //cluster expansion function
NeighbourSearching(M, Eps)                             // Neighbor searching
... ...
```

4. Design and Implementation of the DBSCAN Algorithm on the Spark Platform

4.1. Analyzing the Sequential DBSCAN Algorithm

Before designing the parallel algorithm, it is necessary to use some professional performance-analysis tools to determine any hotspots of the algorithm. In this paper, the Intel® VTune™ profiling tool was used to perform hotspot analysis on the serial algorithm. In the test, it was found that the scale of the dataset was closely related to the performance of the DBSCAN algorithm. For this reason, the dataset scale was changed to carry out the test. The test results are shown in Table 1.

Table 1. Hotspots analysis test on the serial DBSCAN algorithm.

Dada Scale	Neighborhood Query Time (T1/s)	Data Reading and Writing Time (T2/s)	Total Running Time (Ts/s)	T1/Ts
10 K	32.97	3.66	36.63	90.01%
20 K	128.66	13.44	142.10	90.54%
40 K	518.15	52.25	570.40	90.84%

From Table 1, it can be seen that the serial algorithm has a hotspot that occupied the most time-consuming part of the algorithm, regardless of the changing scale of the datasets. The hotspot mainly arose due to the neighborhood query function (*FindArrivalPoints* ( )), which occupied more than 90% of the whole time consumption.

4.2. Parallel Design of the DBSCAN Algorithm

In this paper, the mesh and the secondary extended partition strategy was adopted [27] to parallelize the DBSCAN algorithm. After data partitioning using this strategy, the data was divided into local and border areas, and the data points were marked. All points in the boundary area were included in multiple local areas, thus providing the conditions for the later cluster-merging procedure.

Additionally, this strategy solved the problem whereby the same point is divided into different clustering classes during the clustering procedure.

The principle of the parallel algorithm adopted this strategy is described as follows (Figure 5): (1) The formatted dataset from the HDFS is read and cached into the memory. (2) Based on the computation of the RDD, Spark's memory capabilities, and the properties of directed acyclical graphs (DAG), the dataset is divided into computing nodes in order to perform local partition-clustering operations according to the above-mentioned strategy. (3) Each partition performs local clustering operations in parallel. (4) After the local data-clustering is complete, the clustering results are merged. (5) Finally, the merged results are re-marked, and then the global clusters are generated. A flowchart of these procedures is shown in Figure 5, where it is found that the parallel algorithm has four important stages: (1) data partitioning; (2) local clustering; (3) data merging; and (4) global clustering generation.

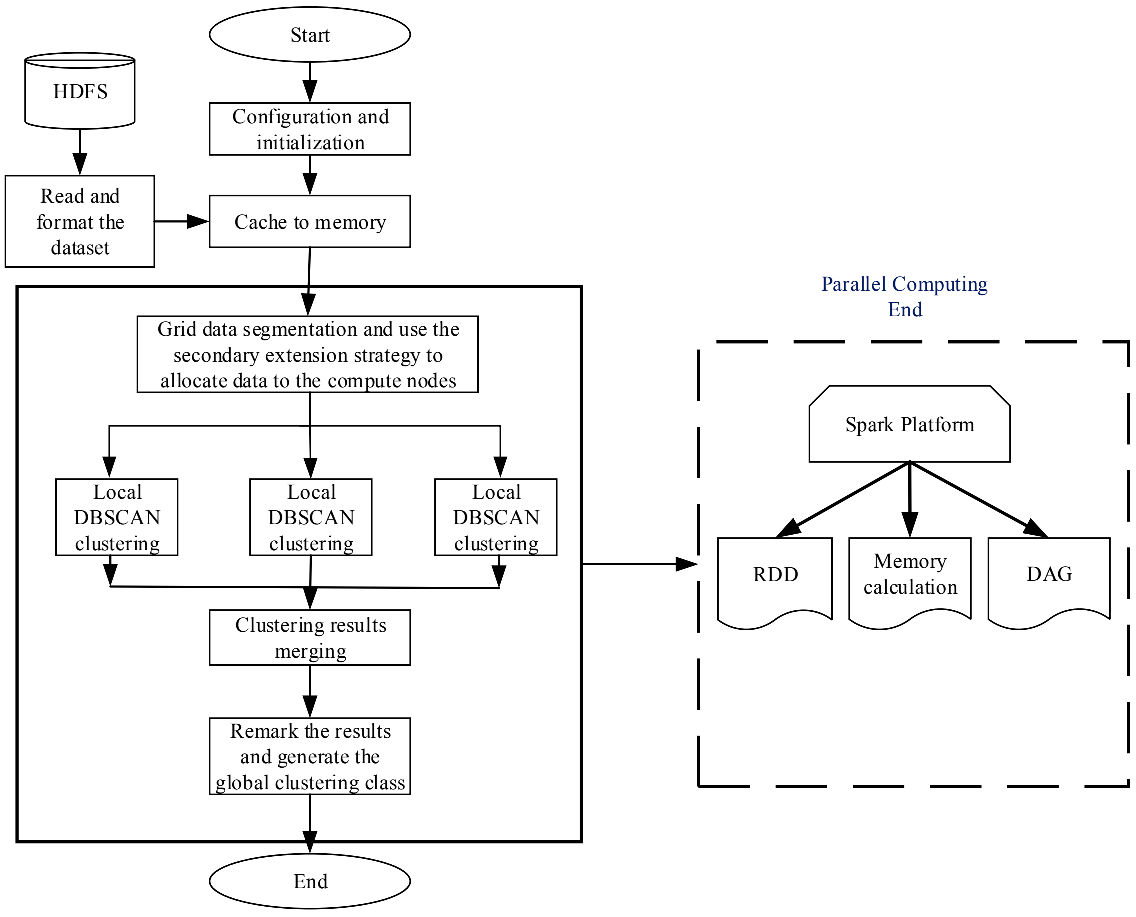

Start
HDFS
Configuration and initialization
Read and format the dataset
Cache to memory
Parallel Computing End
Grid data segmentation and use the secondary extension strategy to allocate data to the compute nodes
Spark Platform
Local DBSCAN clustering
Local DBSCAN clustering
Local DBSCAN clustering
RDD
Memory calculation
DAG
Clustering results merging
Remark the results and generate the global clustering class
End

Figure 5. Flow chart of the parallel DBSCAN algorithm.

4.2.1. Data Partitioning Stage

In this stage, the method of data partition is very important for the implementation of the parallel algorithm. Figure 6 demonstrates the procedure of performing the data partitioning. The data are divided into several data slices with a given length and width according to the data range. Normally, *3Eps* is a suitable number to be used in the division, and has been demonstrated to give the best performance, based on experience. In Figure 6, all the points *(x, y)* that fall into a given rectangle (3*Eps*, 3*Eps*) belong to one data slice (e.g., *S1*), and points within this slice are assigned to the same node processed by the *executor*.

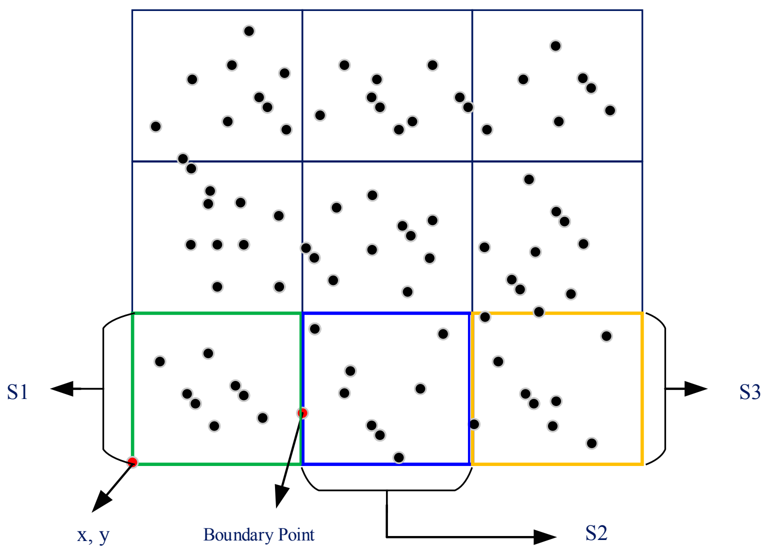


Figure 6. The first phase of data partitioning.

In Figure 6, i.e., the data partitioning procedure, allocating points that belong to the same cluster to different nodes is unavoidable; that is, the attribution problem of the boundary points resulting from the data division appears. However, these boundary points are essential to the following cluster-merging process. Therefore, the secondary slice expansion technique was adapted with the first data-partitioning operation.

In Figure 7, data slices 1 and 2 are taken as an example to explain the mechanism of the secondary slice extension method. Slices 1 and 2 have a common boundary point. In order to include the boundary point in its cluster, it is necessary to expand the boundary of both slices. Here, the border of each of the new slices is extended outwards by $0.1Eps$. Then, the new slices both contain the boundary point, which is beneficial for the subsequent clustering process.

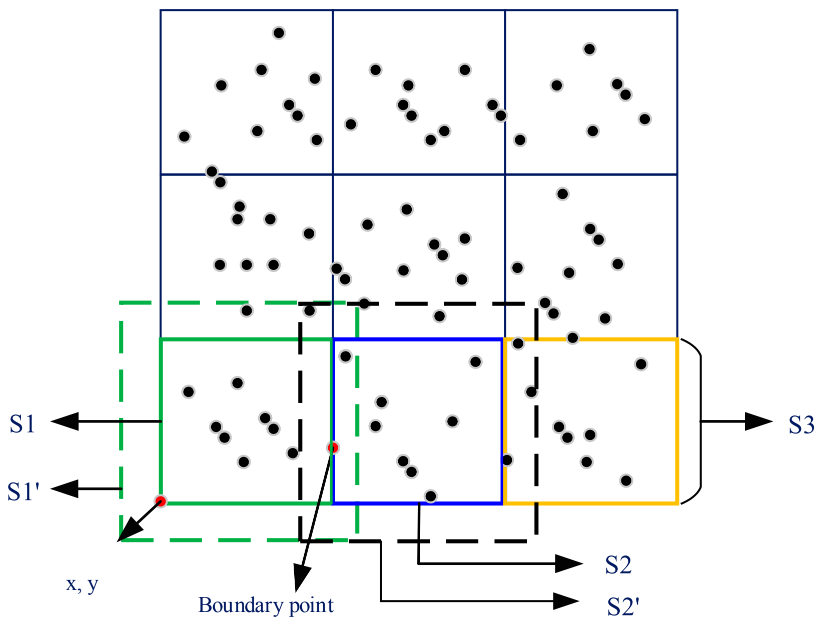


Figure 7. The second phase of data partitioning.

An RDD conversion flowchart of the data partitioning stage is shown in Figure 8.

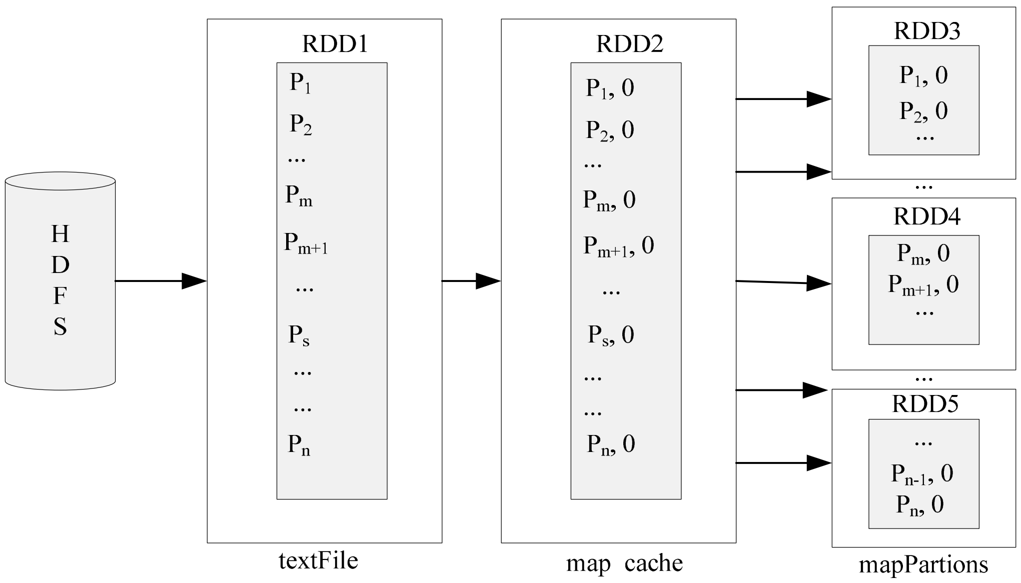


Figure 8. RDD conversion flowchart of the data partitioning stage.

From Figure 8, the text file to be processed is taken out from HDFS and loaded into a RDD. The *P* in Figure 8 represents a point (i.e., the point's location information). Subsequently, the data to be processed is transformed into a key/value pair by the *map* operator (where, 0 represents the initial cluster class ID), then is stored by the cache operator to the memory. Finally, the text file is partitioned by the *mapPartition* operator, where each RDD is treated as a partition.

4.2.2. Local Clustering Stage

At this stage, any data slices that have been segmented in the final step of the first stage are then distributed to the *executor* via the task scheduler for local DBSCAN clustering calculations. The process is demonstrated in Figure 9.

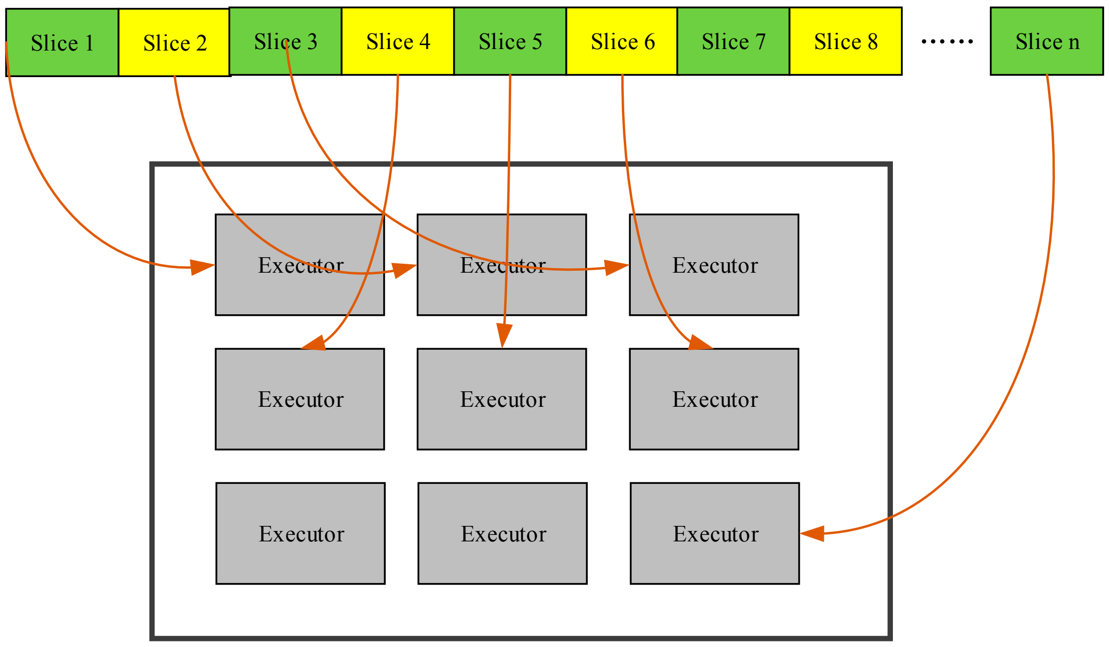


Figure 9. Local clustering calculation stage.

After clustering, the format of the obtained intermediate result is a Key/Value. The values that the Key/Value can take are *longitude*, *latitude*, *ClusterId*, *IsLocalRegion*, and *IsCorePoint*. *longitude* and *latitude* represent the coordinate position of the current data point, while *ClusterId* gives the cluster number of the current data point (the default value is 0). *IsLocalRegion* specifies whether the current data point is located in the area (*True*) or in the boundary area (*False*), and *IsCorePoint* indicates whether the current data is a core object (*True*) or a border object (*False*). An RDD conversion flow chart of the local DBSCAN clustering is shown in Figure 10.

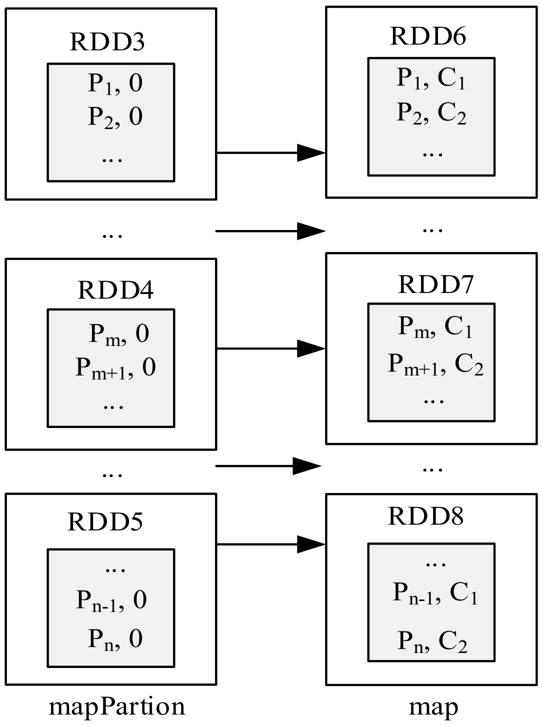


Figure 10. RDD conversion flowchart in the local clustering stage.

From Figure 10, via the *mapPartion* operator, the data are segmented into different data slices. Then, these data slices are assigned to compute nodes. The local DBSCAN clustering operation is performed on each slice by the executor called by the *map* operator. At this stage, the points on each slice are classified into their respective cluster classes.

4.2.3. Data Merger Stage

After each computing node finishes its data slice(s) calculation, the results are passed to the master node. The master node then processes each object according to the received key/value pairs. If *IsLocalRegion* is *True*, the object can output its *ClusterId* directly. For a data object with a cluster number of 0, it is necessary to determine the value of its *IsLocalRegion* parameter. If it is not in the boundary area, it is treated directly as a noise point. If it is in the boundary region, it is necessary to determine whether it is included in the field of other core points. For any data points with multiple cluster labels and which are also core points, it is necessary to indicate whether these clusters can be merged and renamed in the next stage.

As shown in Figure 11, after the clustering operation, cluster 1 and cluster 2 have been generated. *P* is a core point, i.e., it is a boundary point belonging to both clusters. Therefore, this means that these clusters are actually the same cluster, and can thus be merged into a single cluster.

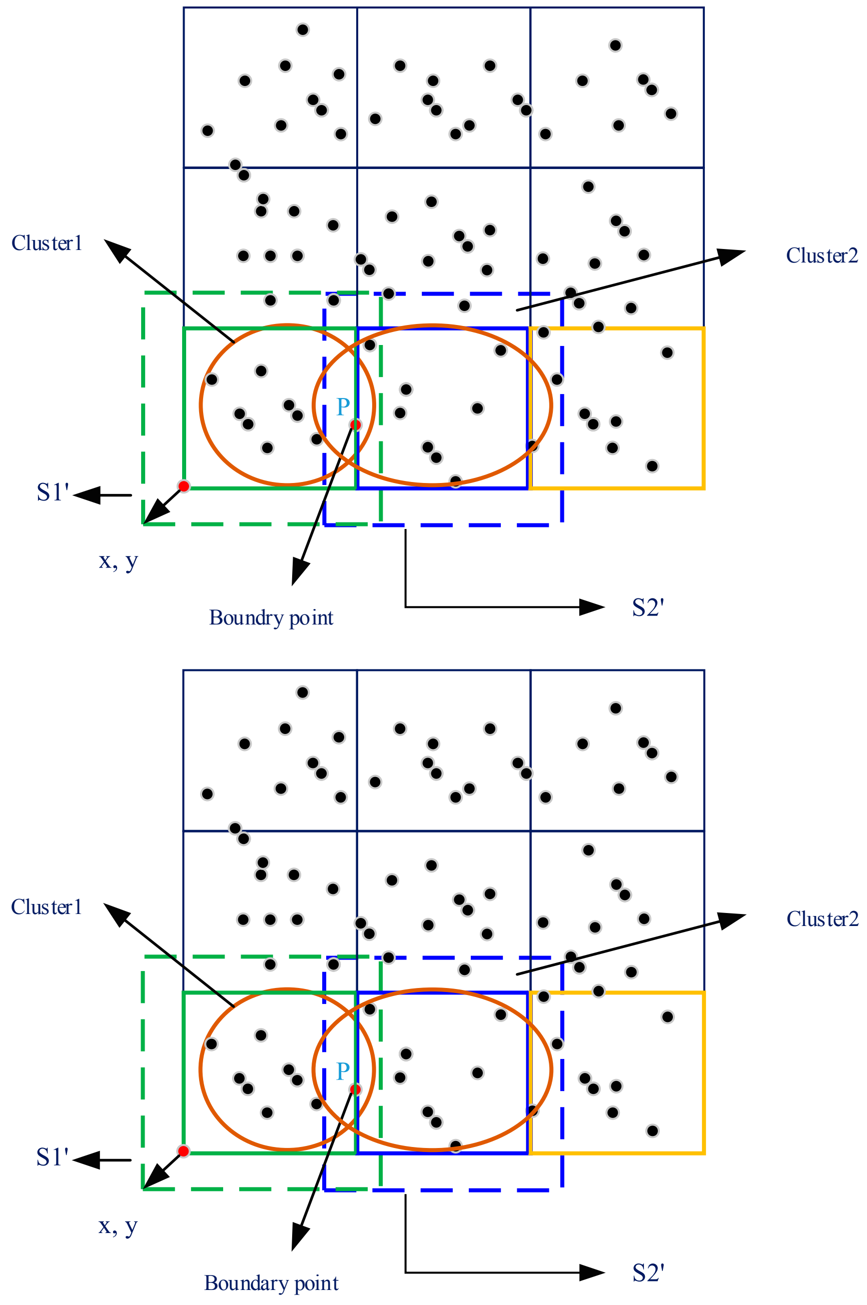


Figure 11. Data merging stage.

4.2.4. Global Cluster Generation Stage

In this stage, the resulting clusters are renamed, i.e., each cluster will have only one global cluster number. Thus, the final clustering results are generated, as shown in Figure 12, and a flowchart of the RDD conversion operations for generating the global cluster is shown in Figure 13.

As can be seen in Figure 13, the clustering results distributed to each RDD cluster are merged into one RDD and are re-marked and re-named. As a result, each cluster has only one global cluster class number, and generates the final clustering result. Finally, the result is dumped to a text file and stored in HDFS.

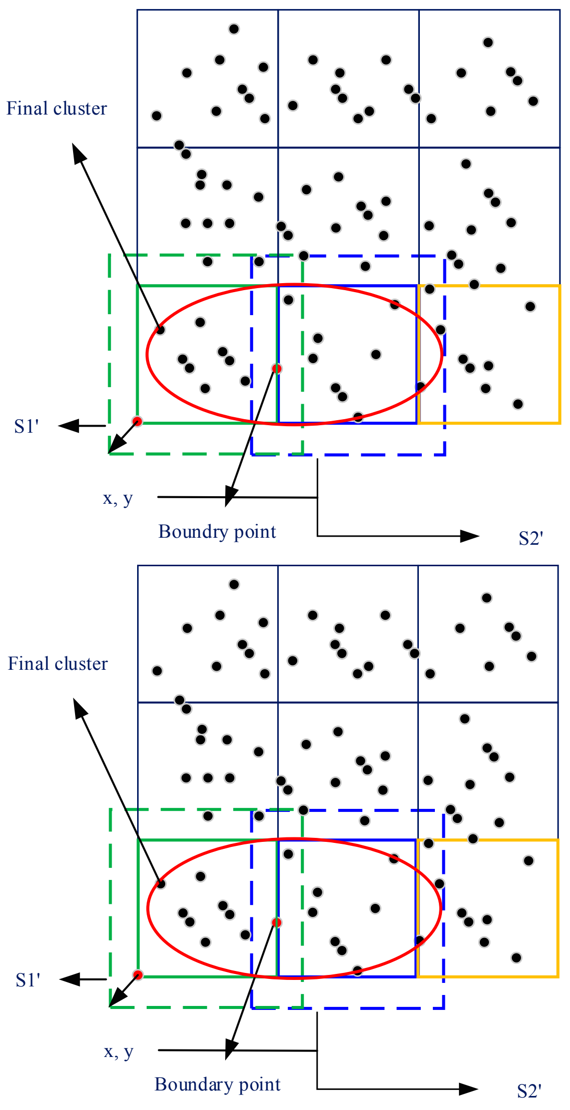


Figure 12. A global cluster generation stage.

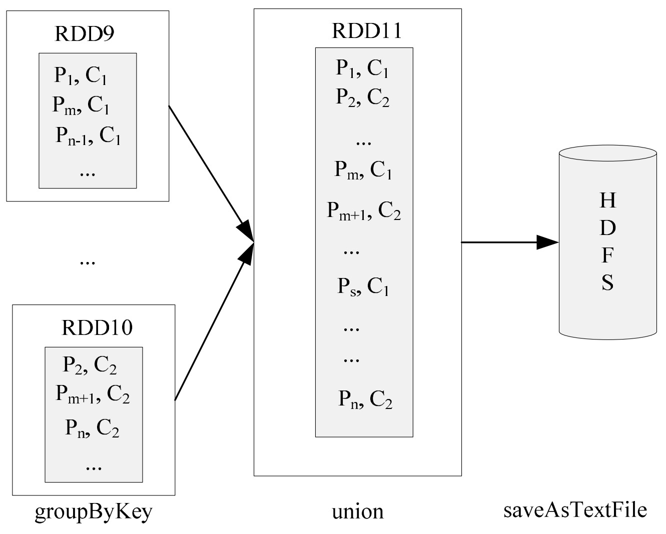


Figure 13. Flowchart of the RDD conversion of the global cluster generation.

4.3. Optimizing the Parallel DBSCAN Algorithm on the Single-Node Spark Platform

To implement the parallel algorithm on the Spark platform, it is necessary to select suitable optimization methods according to the algorithm's characteristics, data size, and cluster resources. Reasonable tuning methods can give full play to the computing capability of the Spark platform. The initial parallel DBSCAN algorithm was designed and implemented based only on the algorithm's characteristics, and it did not consider other optimization aspects. In the existing single-node parallel DBSCAN algorithm based on the Spark platform, there is a large amount of data transmissions, RDD object storage, and inter-node communication, which increases the running time of the algorithm to a certain extent. Considering these aspects, this section will optimize the parallel algorithm in terms of the aspects of data transmission, serialization, and resource parameter tuning based on the single-node parallel algorithm, whereby (1) the data transmission optimization method can improve the data transmission speed, and hence reduce the data transmission time; (2) the serialization optimization method can reduce the storage cost of the data and also reduce the overhead of the metadata; and (3) the resource parameter tuning can make full use of the platform computing resources to further improve the efficiency of the parallel algorithm.

4.3.1. Optimization Method 1: Optimization of Data Transmission

Spark uses the broadcast variable (*Broadcast*) to transmit a large amount of pre-processed data. Optimizing data transmission is needed to improve its overall speed. The introduction of *Broadcast* variables, i.e., read-only variables, are shared by multiple tasks in the Spark calculation process. Only one copy of the *Broadcast* variable needs to be saved on each node, and it does not need to carry out variable transferring for every task. This thereby reduces the transmission time, and improves the efficiency of the calculation. The data transmission of *Broadcast* in each iteration will be stored in the *worker* memory of the slave nodes. Generally, the memory will not be released in advance, except when the calculation node memory is insufficient. Therefore, when the memory size of each calculation task is greater than 20 kB, it is recommended that *Broadcast* be used to optimize algorithm performance on the Spark platform. The method for generating a *Broadcast* variable by calling the *SparkContext* method is as follows. Let us take the bordering area object variable (*borderPoint*) as an example, where *borderPoint* was reused several times in the program. Table 2 shows which variable used the pseudo codes in the program before the data transmission optimization, while the pseudo codes after the optimization are shown in Table 3.

Meanwhile, in the process of implementing the Spark-based parallel algorithm, when the collected data used by the *collect* operator is too large, the data is stored in the distributed file system. This optimization method can effectively reduce the data I/O overheads and relieve the storage pressure on the *Driver*. If the amount of collected data is small, the adjustment parameter *spark.akka.frameSize* alleviates the problem of AKKA buffer overflow caused by large-task distribution, because Spark finishes task distribution by passing messages between the *Actor* models in the AKKA library.

Table 2. Optimization Method One—the pseudo codes before data transmission optimization.

```
val data = sc.textFile(src)
val parsedData = data.map(s => Vectors
    .dense(s.split(' ')
    .map(_.toDouble)))
    .cache()                    //RDD conversion is performed by various operators
    ...
//The reused boundary area object variables in the clustering process
val borderPoint = partitions.map(lambda (key, value): ((key, 0), value))
...
```

Table 3. Optimization Method One—the pseudo codes after the data transmission optimization.

```
val data = sc.textFile(src)
val parsedData = data.map(s => Vectors
.dense(s.split(' ')
.map(_.toDouble)))
.cache()                              // RDD conversion is performed by various operators
. . .
//The reused boundary area object variables in the clustering process
val borderPoint = partitions.map(lambda (key, value): ((key, 0), value))
//Introduce the broadcast variables
val borderShare = vectors.context.broadcast(borderPoint)
. . .
```

Based on the Scala functional programming language, Spark uses the closure method to transmit data by referencing the function variables. Therefore, using the task to distribute the data will reduce the overall execution efficiency. Setting the parameters of *Spark.akka.frameSize* can fix the maximum capacity of the *Actor* communication message (such as the output of the task) in AKKA used by the Spark platform, whose value defaults to 10 MB. When dealing with large-scale data, the output of the task may be greater than this value. Generally, it needs to be set to a higher value according to the size of the data.

4.3.2. Optimization Method 2: Serialization Optimizing

Serialization plays important roles in the development of distributed computing platforms. In the serializing approach, chained object data turns into data objects stored in a continuous space via a byte array; thus, the serialized object will be stored in a distributed file system with a continuous space form, and transmitted in the form of a data stream. This approach reduces the overheads related to metadata information both of the object itself and the basic data types; therefore, it can effectively reduce the data storage space, and reduce the overhead and stress of the garbage collection (GC).

Those data formats require a lot of time to serialize, and any objects occupying the object's space will slow down the entire application's efficiency. The serialization method is one of the most important performance-tuning methods in Spark, because the RDD object storage and data transfer between nodes is necessary for carrying out the serialization process.

The Spark platform includes both Kyro and Java serialization libraries. Compared with the Java serialization library, the Kyro serialization library allows for serialization to be completed quickly and compactly. Also, users can define customized serialization methods that have good scalability. To implement the parallel DBSCAN algorithm on the Spark platform, which has higher network transmission requirements, better performance can be achieved with the Kyro serialization library. This serialization method is also recommended for network transmission-intensive computing in the official Spark documentation. Therefore, the Kyro serialization library was used to optimize the parallel algorithm.

Spark automatically introduces Kyro serialization support for many of the commonly used Scala core classes, all of which are supported by Spark using the Twitter chill library. Based on the foregoing analysis, the optimized program codes adopting Kyro serialization are shown in Table 4.

Table 4. The pseudo codes for Optimization Method 2.

```
// Import the Serialization classes
import com.esotericsoftware.kryo.Kryo
import org.apache.spark.serializer.KryoRegistrator
// Customized serialization classes
class MyRegistrator extends KryoRegistrator{
override def registerClasses(kyro: Kyro){
    kryo.register(classOf[MyClass1])
    kryo.register(classOf[MyClass2])
    }
}
// Codes of the main function
object DBSCANApp{
val log = LoggerFactory.getLogger(DBSCANApp.getClass)
def main(args: Array[String]) {
// Initialize the spark configuration information
val conf = new SparkConf().setAppName(s"DBSCANApp(eps=$eps, minPts=$minPts)
  -> $Output")
// Set the serialization class
conf.set("spark.serializer"," org.apache.spark.serializer.KryoSerializer")
// Introduce the customized serialization classes
conf.set("spark.kryo.registrator", "DBSCAN.MyRegistrator ")
val sc = new SparkContext(conf)
...
}
}
```

4.3.3. Optimization Method 3: Optimization of Resource Parameters

In a distributed computing environment like the Spark platform, reasonable configuration of resource parameters can make full use of the platform's computing resources and promote its parallel computing capabilities, thereby enhancing its performance. Table 5 gives the main resource parameters and their specific action descriptions in Spark. One thing to note is that each of the parameters corresponds to a part of the operating principle of the job execution.

Table 5. Resource parameter tuning.

Tuning Parameters	Function
num-executors	Set the specified number of Executor processes to execute the Spark job
executor-memory	Set the memory for each Executor process
executor-cores	Set the number of CPU cores for each Executor process
driver-memory	Set the memory of driver process
spark.default.parallelism	Set the number of default tasks
spark.storage.memoryFraction	Configure the amount of space used for caching in RDD, which defaults to 0.67
spark.shuffle.memoryFraction	Configure Executor memory scale in shuffle process, which defaults to 0.2

When the parameter of "num-executors" is set to 50–100, one should set "executor-memory" to 4 G–8 G, "executor-cores" to 2–4, the "driver-memory" generally to 1 G, and 500–1000 is recommended for "Spark.default.parallelism", so that the job requirements can be met. When Spark operations have more RDD persistence operations, the parameter "spark.storage.memoryFraction" can be increased. When shuffle class operations become relatively large, the parameter "spark.storage.memoryFraction" should be appropriately reduced. It is also recommended that the parameter "spark.shuffle.memoryFraction" be reduced when the Spark job RDD persistence operation is less and the Shuffle operation is more.

4.4. Implementation of the Parallel DBSCAN Algorithm on a Virtual Spark Cluster

When the user does not have enough physical machines to establish a Spark cluster at the beginning of the experiment, virtualization technology can offer an inexpensive way to construct a cluster to deploy enough virtual nodes for testing and rapid program deployment. Thus, it can manage the applications and facilitate interactive user applications quickly. Virtualization technology is a management technology that simulates real-world computing environments, including virtual computer hardware platforms, virtual storage devices, and virtual network resources. As a lightweight container of virtualization technology in recent years, *Docker* has the advantages of traditional virtualization, meanwhile, it also has characteristics such as file system isolation, resource isolation, network isolation, copy-on-write, and logging recodes.

In order to test the efficiency of the parallel DBSCAN algorithm in the Spark cluster environment, this study used Docker virtualization technology to build a virtualized Spark cluster platform, and tested the parallel DBSCAN algorithm based on this. The principle of the parallel DBSCAN algorithm based on a virtual Spark cluster is as follows. (1) Based on a high-performance physical Linux server, when using the Docker virtualization technology and the existing Spark platform mirrored on Docker Hub, one can dynamically extend the number of the constructed virtual Spark cluster nodes; (2) Then, the parallel DBSCAN algorithm can be tested with different nodes on the virtual Spark cluster; (3) Finally, the impact of the number of nodes with the parallel algorithm's acceleration ratio can be determined. A schematic diagram of the entire procedure is shown in Figure 14.

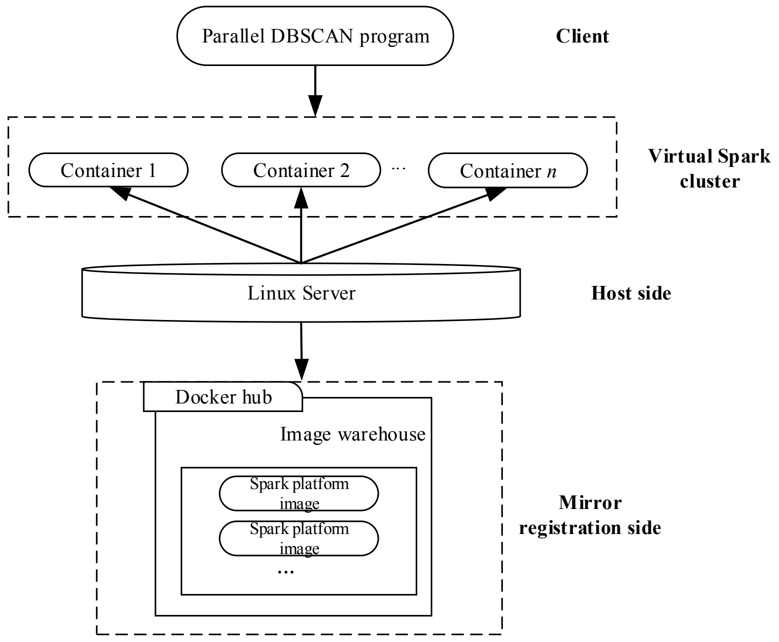


Figure 14. Parallel DBSCAN algorithm based on a Docker-based virtual Spark cluster.

4.5. Implementation of the Parallel DBSCAN Algorithm on the Spark Cluster with Yarn

This part migrates the parallel DBSACN algorithm from the virtual cluster to a physical Spark cluster, and introduces the Yarn resource manager to deploy the algorithm. When deploying the Spark platform with Yarn, which only supports coarse-grained mode, the container starts on the Yarn resource manager, the system resources cannot be dynamically expanded, and the resources used are fixed.

To implement the parallel DBSCAN algorithm with the Yarn resource manager, Yarn will replace the cluster manager in the Spark cluster. The execution process of the parallel algorithm with Yarn is described as follows:

(1) The Yarn client uploads the DBSCAN application packages to HDFS, and requests a job commit to the *Resource Manager* to run this task.
(2) *Resource Manager* responds to the job and requests the *Node Manager* to create a *Spark Application Master* and then starts it.
(3) *Spark Application Master* finds the resource files (the *Jar* packages, etc.) from the HDFS, and starts *DAGscheduler* and *YARN cluster scheduler* to process the initialization operations. Then, the Spark application master will apply the resources from the *Resource Manager*, and start the respective container through the *Node Manager*.
(4) Each container contains multiple *executors* to perform the corresponding tasks simultaneously, and report their state to the *Spark Application Master*.

The executing process of the parallel algorithm implemented with this mode is shown in Figure 15.

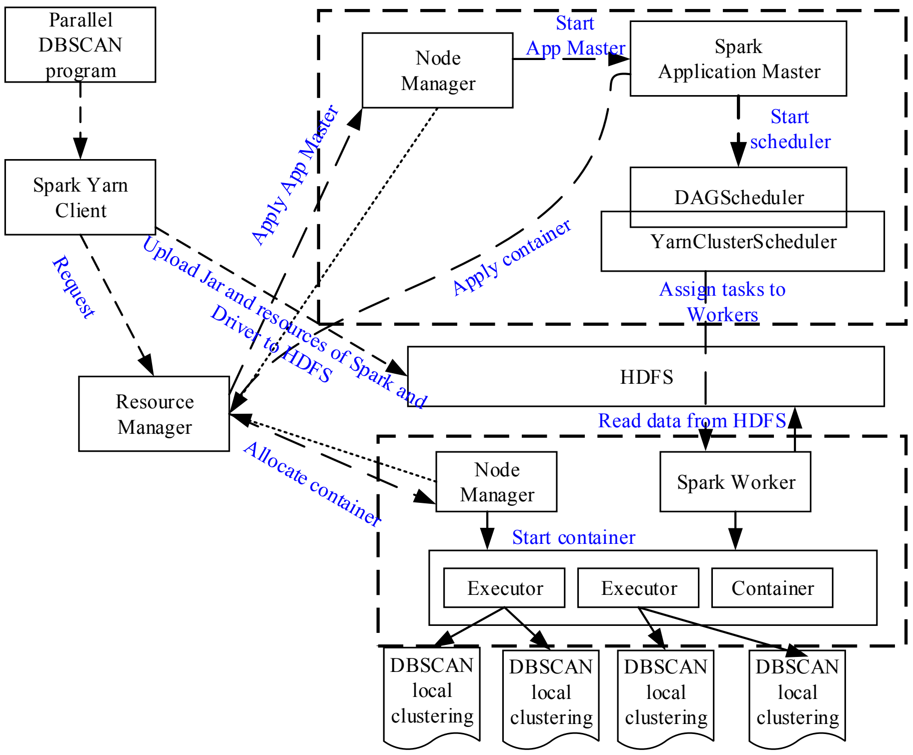


Figure 15. Parallel DBSCAN algorithm based on Yarn resource manager.

In this mode, the specified *executor* can be assigned to the parallel DBSCAN application via the parameter "-num-executors", and then each *executor* can be controlled by the parameters "-executor-memory" and "-executor-cores" to set the size of the assigned memory and the number of occupied CPU cores. This mechanism is conducive for controlling the system resources that the user-submitted applications take up; thus, it can fully share the cluster resources, and improve the throughput of the Yarn manager.

4.6. Implementation of the Parallel DBSCAN Algorithm on a Spark Cluster with Mesos

This part introduces the Mesos resource manager to deploy the Spark platform. Unlike the Yarn resource manager, which adopts a process with an isolation mechanism, the Mesos resource manager uses a Linux container to isolate resources such as memory and CPU. Mesos implements resource sharing in a fine-grained manner, which is aimed at improving cluster utilization, and is highly suitable for the unified management and scheduling of the resources at the data center scale. However, Yarn is more convenient and effective for the management and scheduling of big data jobs. Based on

Mesos deployment, a Spark cluster needs to implement two classes to perform a calculation through the Mesos resource manager. One class is *SparkScheduler*, which is derived from the system class *MesosScheduler*. The other class is *SparkExecutor*, which comes from the base system class of *executor*.

To implement the parallel DBSCAN algorithm based on the Mesos resource manager, the Mesos master node will replace the cluster manager in the Spark cluster. The process of executing the parallel algorithm with the Mesos resource manager is described as follows:

(1) After submitting a DBSCAN processing request through the client side, Spark will generate RDD and Map/Reduce functions, and then transform and generate the corresponding job information. Each job contains multiple tasks, and the job is submitted to the *SparkScheduler*.
(2) The job sets are sent to the Mesos master node by the *SparkScheduler*. The Mesos master node performs the corresponding task scheduling to the Mesos slave nodes as the manager, these slave nodes execute the corresponding task in parallel with multiple executors and then return the results. The process of executing the parallel algorithm is shown in Figure 16.

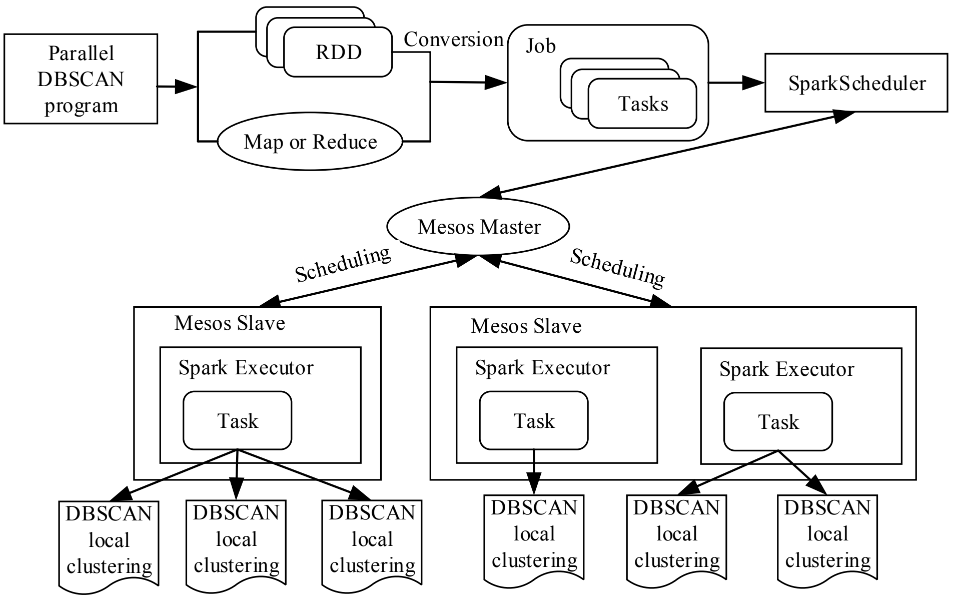


Figure 16. Parallel DBSCAN algorithm based on Mesos resource manager.

There are two supported modes, i.e., the coarse-grained and the fine-grained mode, which are used to deploy the parallel algorithm on Spark with Mesos resource management. With Mesos, *spark.mesos.coarse* is set to *True* to configure the coarse-grained scheduling mode statically. The feature of this mode is that the DBSCAN application has a separate and fixed memory allocation; thus, these free CPU resources can be used to prompt the resource utilization when the application occupying the machine cannot run a task.

5. Test and Analysis of the Proposed Parallel DBSCAN Algorithm

In this section, the serial and the corresponding parallel DBSCAN algorithms are used to process different GPS trajectory datasets on different computing platforms to evaluate the parallel algorithms' performance.

5.1. Configurations of the Experimental Platforms

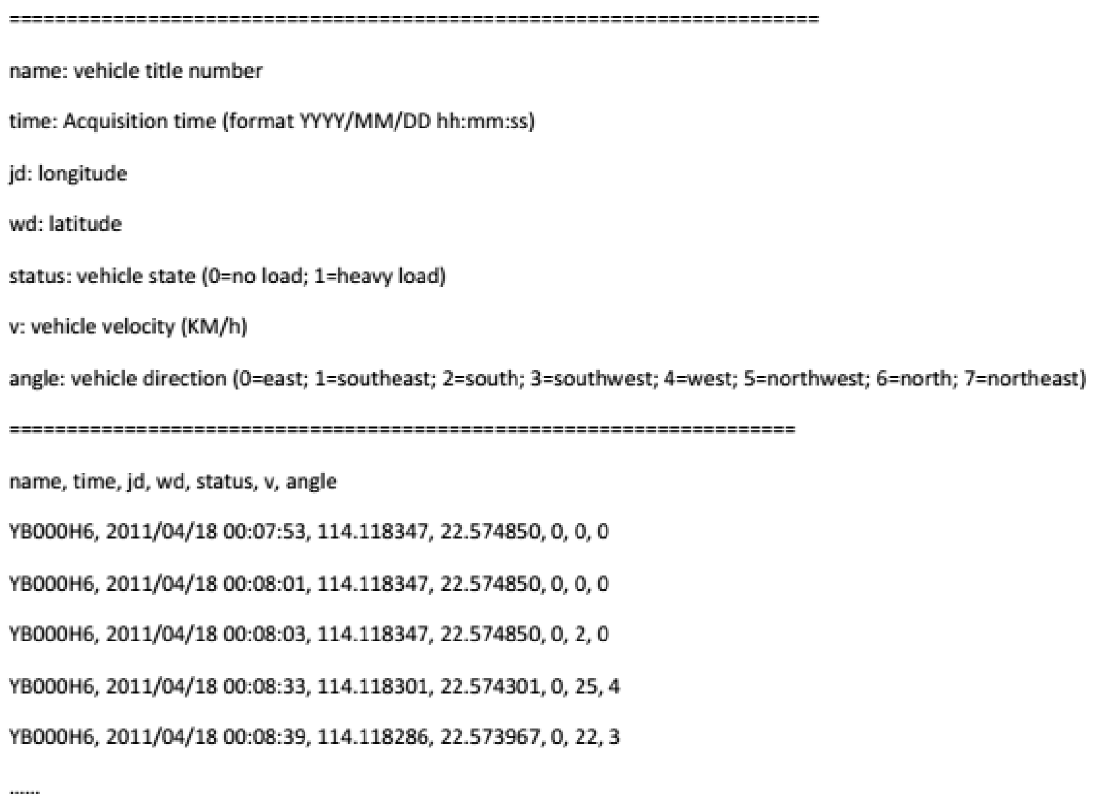
In this paper, there are three types of experimental platform, i.e., a single-node platform, a virtual Spark cluster, and a physical Spark cluster. The hardware configuration and details of the equipped software are shown in Table 6.

Table 6. Experimental platforms' configuration information.

Platform Name	Hardware Configuration	Installed Software	Computing Nodes
Single-node platform	Processors: Intel(R) Xeon(R) E5-2650 v2 @2.60GHz Cores: 8 Memory size: 64G Cache size: 20480KB	CentOS 7.0 Spark 1.6.0 Java 1.8.0 Scala 2.10.4	1
Virtual Spark cluster	Processors: Intel(R) Xeon(R) E5-2650 v2 @ 2.60GHz Cores: 8 Memory size: 64G Cache size: 20480KB	CentOS 7.0 Spark 1.6.0 Docker 1.8.0 Scala 2.10.4 Java 1.8.0	4 virtual node from 1 physical node with Docker software
Physical Spark Cluster	Processors: Intel(R) Xeon(R) E5-2650 v2 @2.60GHz Cores: 8 Memory size: 64G Cache size: 20480KB	CentOS 7.0 Spark 1.6.0 Hadoop 2.6.0 Java 1.8.0 Scala 2.10.4	4

5.2. Experimental Data

In this paper, all experimental data were obtained from the GPS trajectory datasets of Shenzhen in April 2011. The data sets are entirely in ASCII format, and contain vehicle license plate number, latitude and longitude information, vehicle status, speed, and direction of traffic. Before processing, we need to extract the latitude and longitude information from the existing data and form a new spatial dataset with different scales (in this study, different numbers of vehicle records) of 10 K, 20 K, 40 K, 80 K and 100 K. The hotspot detection processing step is rather time-consuming when the data scale reaches or exceeds 40 K. For real-world applications, the data scale needs to be further increased. The experimental data format and a description are shown in Figure 17.



```
====================================================================
name: vehicle title number
time: Acquisition time (format YYYY/MM/DD hh:mm:ss)
jd: longitude
wd: latitude
status: vehicle state (0=no load; 1=heavy load)
v: vehicle velocity (KM/h)
angle: vehicle direction (0=east; 1=southeast; 2=south; 3=southwest; 4=west; 5=northwest; 6=north; 7=northeast)
==================================================================
name, time, jd, wd, status, v, angle
YB000H6, 2011/04/18 00:07:53, 114.118347, 22.574850, 0, 0, 0
YB000H6, 2011/04/18 00:08:01, 114.118347, 22.574850, 0, 0, 0
YB000H6, 2011/04/18 00:08:03, 114.118347, 22.574850, 0, 2, 0
YB000H6, 2011/04/18 00:08:33, 114.118301, 22.574301, 0, 25, 4
YB000H6, 2011/04/18 00:08:39, 114.118286, 22.573967, 0, 22, 3
......
```

Figure 17. Description and data format of the GPS trajectory data.

5.3. Parallel Algorithm Evaluation Index

In this paper, two factors—speedup scalability is used to evaluate the parallel algorithms' performance. Among them, speedup is an important evaluation index, which can directly reflect the level of parallel efficiency [50]. The definition of speedup is as shown in Equation (1):

$$S_p = \frac{T_l}{T_p} \tag{1}$$

where S_p means speedup, T_l indicates the processing time of the serial algorithm, and T_p represents the processing time of the parallel algorithm. When S_p is larger, the performance of the parallel algorithm is higher.

5.4. Experiment Design

Based on five selected datasets with different data scales, the performance of the parallel DBSCAN algorithms was tested and compared with a serial algorithm with the fixed clustering parameters (Eps, $MinPts$) set to (15, 90). To obtain the accurate running time of the serial and parallel algorithm, functions such as *time.h*, *clock ( )* and *currentTimeMillis ( )* etc. were used for the time statistics. In order to reduce experimental error, all running time values are expressed as the statistical average time over several experiments. The entire experiment consisted of the following five tests on the parallel DBSCAN algorithm:

(1) Test the performance of the parallel DBSCAN algorithm based on a single-node Spark platform.
(2) Test the performance of the optimized parallel algorithm based on a single-node Spark platform.
(3) Test the performance of the parallel algorithm based on a virtual Spark cluster.
(4) Obtain the performance of the parallel algorithm based on a physical Spark cluster. The tests were conducted employing both the Mesos and Yarn resources managing modes in turn.

5.5. Experimental Results and Analysis

5.5.1. Testing the Parallel Algorithm on a Single-Node Spark Platform

For a single-node Spark platform, the running time of the serial and parallel DBSCAN algorithms for the five different data scales are shown in Table 7. For the five datasets, the processing performance of the parallel algorithm based on the single-node Spark platform is significantly higher than that of the serial algorithm. The speedup values of the parallel algorithm are shown in Figure 18.

Table 7. Running time of serial and parallel DBSCAN algorithm based on a single-node Spark platform.

Data Scales	Clustering Parameters	Sequential Elapsed Time (s)	Parallel Elapsed Time on Single Node Spark Platform (s)
10 K	(15, 90)	36.63	12.98
20 K	(15, 90)	144.10	29.89
40 K	(15, 90)	570.40	78.68
80 K	(15, 90)	2281.87	273.93
100 K	(15, 90)	3815.95	453.74

From Figure 18, generally, it can be seen that the speedup of the parallel algorithm based on the single-node Spark platform accelerates gradually as the data scale increases. When the data scale is small, the execution time is shorter, and the acceleration ratio is not ideal. This phenomenon is also influenced by factors related to the Spark platform's internal data transmission and communication processes. When the data scale reaches 40 K, the acceleration ratio tends to be stable. Due to the limitation of CPU core number in the single-node Spark platform, the parallel algorithm can only start a small number of *executors*. When the spatial data scale reaches the 100 K scale, the obtained speedup reaches 8.41.

The aforementioned test results show that it is feasible to parallelize the DBSCAN algorithm on the Spark platform, which can improve its processing efficiency to some extent.

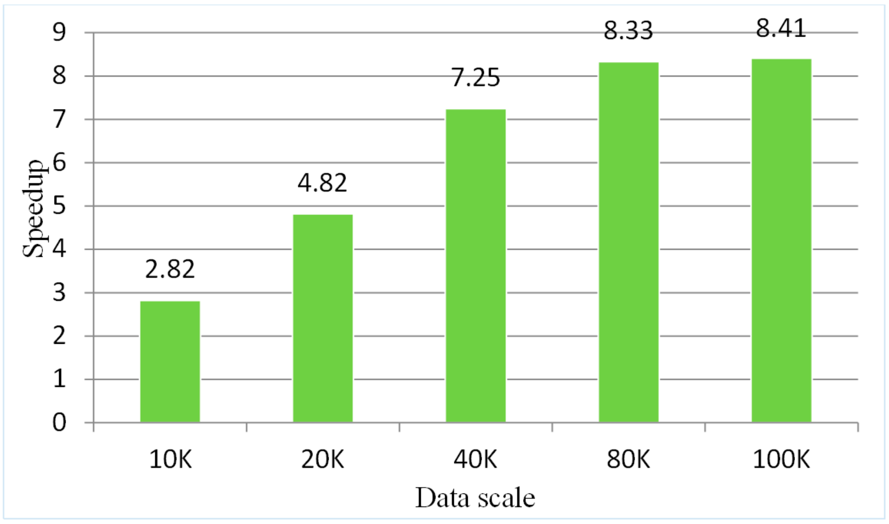


Figure 18. Speedup of the parallel DBSCAN algorithm based on a single-node Spark platform.

5.5.2. Testing the Optimized Parallel Algorithm on a Single-Node Spark Platform

The optimized parallel algorithm adopts three measures to improve its performance. The comparison of the parallel algorithm based on the same single-node Spark platform before and after the optimization are demonstrated in Table 8.

Table 8. Comparison of the elapsed time of the parallel algorithm and its optimized algorithm.

Data Scales	Elapsed Time of the Serial Processing (s)	Elapsed Time of the Parallel Algorithm before Optimization (s)	Elapsed Time of the Parallel Algorithm after Optimization (s)
10 K	36.63	12.98	7.55
20 K	144.10	29.89	15.92
40 K	570.40	78.68	43.12
80 K	2281.87	273.93	166.56
100 K	3815.95	453.74	274.53

From Table 8, it can be seen that the optimized parallel algorithm is more efficient than the original parallel algorithm. The acceleration ratios of five different data scales are shown in Figure 19, from which it can be seen that the obtained speedup values increase from 2.82, 4.82, 7.25, 8.33, and 8.41 to 4.85, 9.05, 13.23, 13.70 and 13.90, respectively, for the different data scales. The performance improvement ratio of the optimized parallel algorithm is improved by 71.99%, 87.76%, 82.48%, 64.47%, 65.28% based on the single-node Spark platform. The test results show that the performance of the optimized algorithm is further accelerated, which verifies the effectiveness of the three optimization methods described.

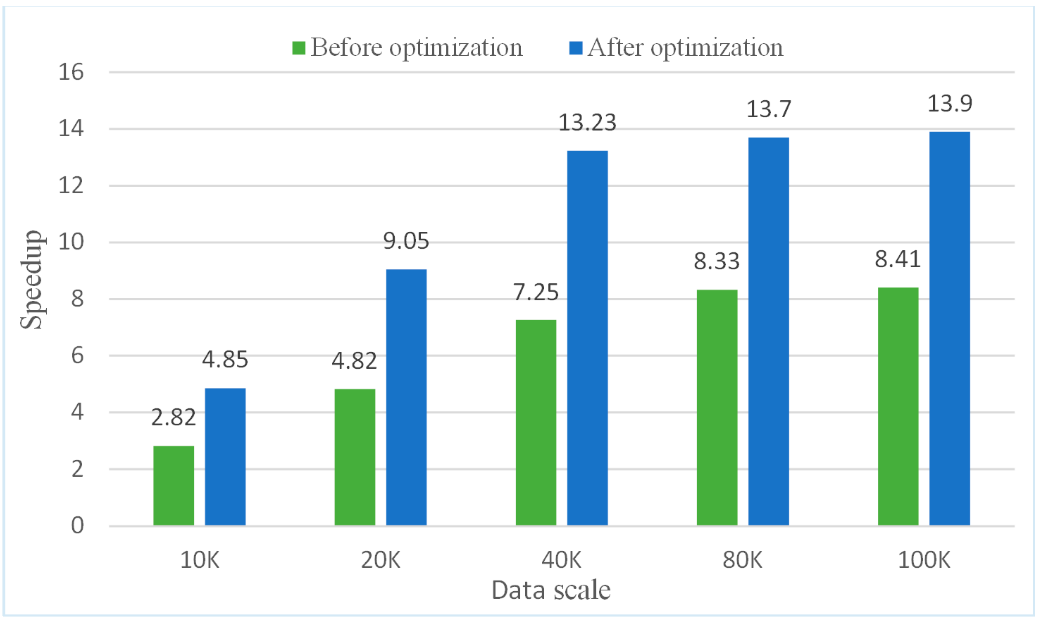


Figure 19. Comparison of achieved Speedup before and after the optimization.

5.5.3. Testing the Parallel Algorithm on a Virtual Spark Cluster Platform

In this section, Docker container virtualization technology is used to build a specified number of Spark virtual clusters dynamically based on a Linux server, and to test the efficiency of the parallel algorithm based on a virtualized Spark cluster. The elapsed time of the algorithm is shown in Table 9, from which it can be seen that the processing time of the parallel algorithm is not reduced with the number of virtual cluster nodes on the single-node Spark platform. The scalability of the parallel algorithm on a virtual cluster is unsatisfactory, because the acceleration ratio of the parallel algorithm does not linearly increase with the number of cluster nodes.

Combined with the Spark platform and Docker container's technical characteristics, the above results arise for the following reasons:

(1) The parallel algorithm has intensive I/O operations, so it requires a higher network-transmission state; however, the virtual cluster built with a single-node server encounters transmission pressure.
(2) The performance of the virtual Spark cluster cannot achieve the same desired effects as a physical Spark cluster.

Although the performance of the parallel algorithm on the Docker-based virtual Spark cluster is poor, it presents a simulation of the Spark cluster environment that is able to test the implementation of parallel programs in cluster mode for the user. In the next section, a physical Spark cluster is used to test the parallel algorithm, replacing the virtual Docker Spark cluster.

Table 9. Running time of serial and parallel DBSCAN algorithms based on Docker Spark platform (s).

Virtual Nodes / Data Scales	1	2	3	4
10 K	7.55	15.38	16.22	17.87
20 K	15.92	24.73	24.85	25.25
40 K	43.12	61.46	59.27	58.00
80 K	166.56	190.28	188.52	186.32
100 K	274.53	360.24	358.24	366.48

5.5.4. Testing the Parallel Algorithm on a Spark Cluster with the Yarn Resources Manager

In this section, a physical Spark cluster and the Yarn resource manager are used to deploy and test the performance of the parallel DBSCAN algorithm. The speedup changes in relation to different data scales in this mode are recoded and compared, as shown in Figure 20.

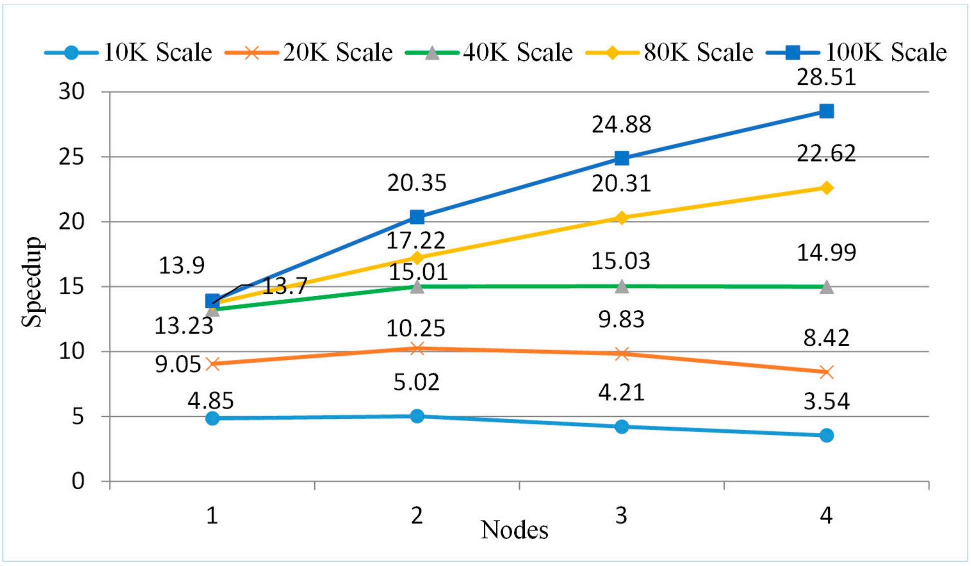


Figure 20. Achieved speedup of the parallel algorithm with different data scales with the Yarn resources manger.

As can be seen from Figure 20, the acceleration curves for the obtained speedup for data scales of 10 K and 40 K in this mode tend to be gentle as the number of computing nodes increases. This is mainly because the data scale is relatively small, and hence their processing times are relatively short. With an increase in computing nodes, the expenses of data transmission, resource scheduling, heartbeat detection, etc. reduce the execution efficiency of the algorithm. It can be seen that the acceleration ratio achieved is basically consistent with a linear increase with processing data scales of 80 K and 100 K. However, a single-node cluster cannot process such large amounts data, which indicates that the parallel DBSCAN algorithm based on the Spark in Yarn mode is effective when the data amount is sufficiently large. This method can solve the performance bottleneck that arises when using the serial algorithm, and thus dramatically improves processing efficiency.

5.5.5. Testing the Parallel Algorithm on a Spark Cluster with the Mesos Resources Manager

In this part, the same physical Spark cluster using the Mesos resource manager to deploy and test the parallel DBSCAN algorithm was tested. Meanwhile, the performance results achieved under these two modes were tested. The speedup changes for different data scales in this mode are recoded and compared in Figure 21.

In Figure 21, we find the same phenomenon as with the Yarn mode. For example, the acceleration curve tends to be flat with the data scales of between 10 K and 40 K, while better acceleration ratios are achieved as the data scale becomes larger.

To determine the performance difference of these two resources managers, experiments were performed using a data scale of 100 K for both of these modes. A comparison chart of their acceleration performance is shown in Figure 22.

From Figure 22, it can be seen that Yarn shows better performance, and is more suitable than Mesos for applications with more iterations. The reason for this is that the Yarn resource manager is able to deal with the dynamic distribution of resources according to the application requirements when dealing with algorithms with multiple iterations. As a result, Yarn can make full use of the computing capability of the Spark cluster platform.

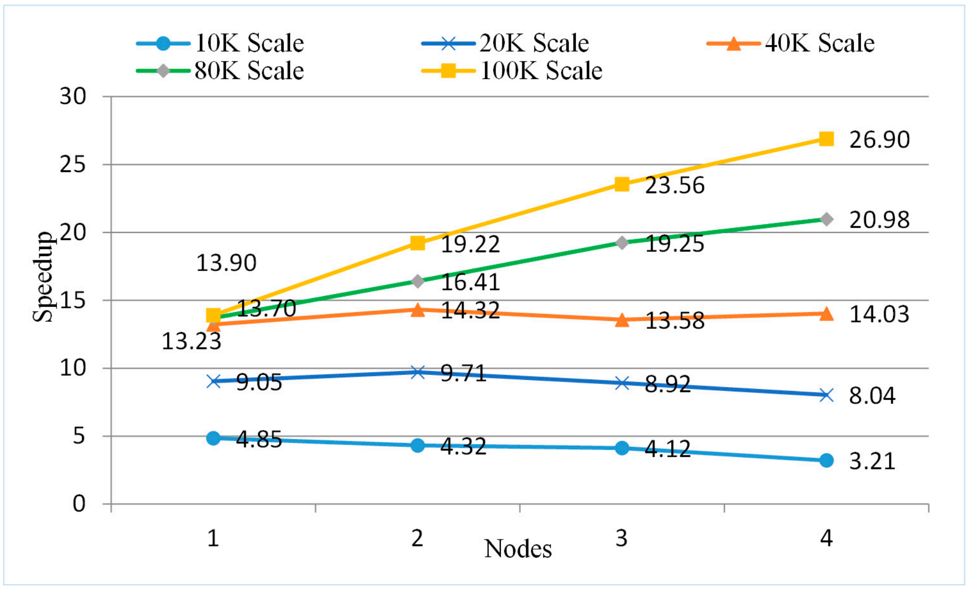


Figure 21. Achieved Speedup of the parallel algorithm at different data scales with the Mesos resource manager.

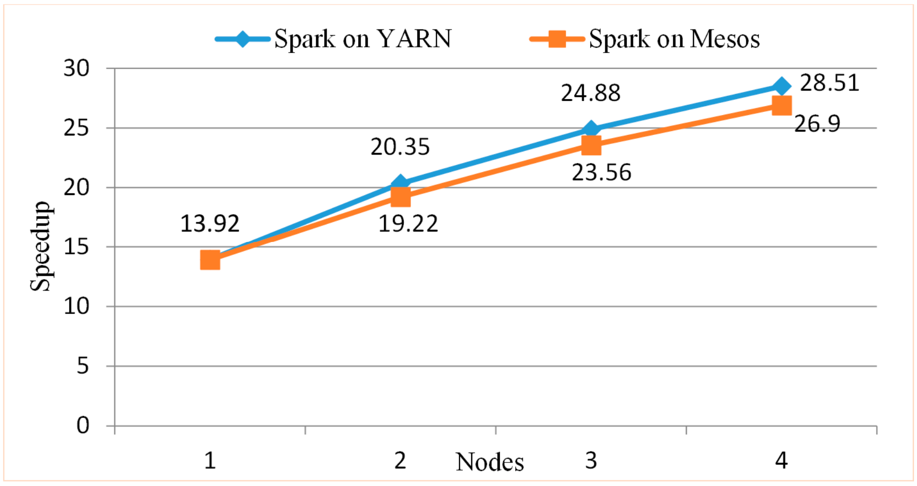


Figure 22. Performance comparison of Yarn and Mesos resources manager.

6. An Urban Congestion Area Discovery Application

6.1. Summary of City Traffic Congestion Area Discovery

Urban transportation is one of the main driving forces of urban development, and is responsible for connecting cities, regulating the passenger flow, and providing logistics for city development. The traffic situation has a decisive impact on a city's system development, and the lives of traveling people [51]. However, with the development of the urban economy, the numbers of private cars, taxis, and bus holdings have gradually increased, and the traffic infrastructure in some areas has been unable to meet the growing demand for vehicle operation and transport capacity. Thus, locating areas of traffic congestion, and subsequently targeting the planning of new urban roads to alleviate these problems are important hurdles that urgently need to be overcome in urban design and construction [52].

Spatial data contains rich spatial features, thematic features, temporal characteristics, and other semantic information. Therefore, by studying temporal and spatial trajectory spatial data, we can understand the rules of its movement and its corresponding changes. Taxi trajectory data can reflect the traffic situation in a city to a certain extent. Through a large volume of taxi trajectory data, one can quickly understand the dynamical distribution of vehicles and find congested areas in urban traffic. Locating these congested areas can help urban planners understand the problems in urban transportation, and hence adopt some measures to adjust, optimize and upgrade urban roads to meet the growing demand for traffic and transportation. The DBSCAN algorithm can be used in these applications.

6.2. Traffic Congestion Area Discovery Based on the Spark Platform

Although the serial DBSCAN algorithm has been successfully applied in the field of traffic-jam discovery, with the gradual increase in data size, it cannot meet the needs of some real-time applications. In order to speed up the process, this study applied the parallel DBSCAN algorithm to the problem of city traffic congestion discovery based on the Spark cluster platform.

First, the GPS trajectory data was cleaned to obtain a spatial data format suitable for data-clustering operations. Then, the spatial data was clustered on each computing node in the Spark cluster. After each computing node had finished a calculation, the clustering results were merged. Then, through certain data processing means, areas of traffic congestion were identified. Finally, based on comparison with the existing research results, the feasibility and practicality of the parallel DBSCAN algorithm were verified. The entire process is demonstrated in Figure 23.

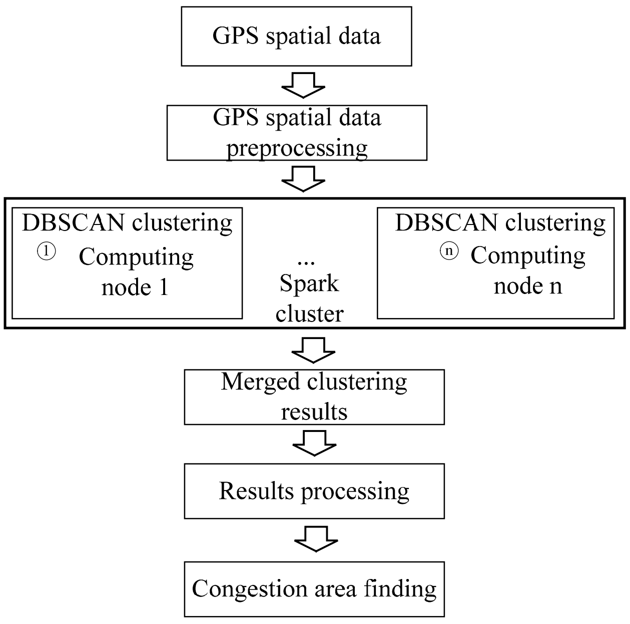


Figure 23. Entire process of the congested area discovery with GPS trajectory data.

6.3. Experimental Data and Platform Configuration

The experimental data used are taxi GPS data of Shenzhen City in the year of 2011. The data contains a total of 13,799 vehicles and a data size of 1331 MB. The acquired data are from April 18 to April 26, with an acquisition frequency of 1–5 s. The data format as shown in Table 10. Experiments were performed on this data set using the serial DBSCAN algorithm and the corresponding parallel algorithm based on the single-node platform and the Spark cluster platform. The hardware and software configuration of the test platforms were as described in Table 6.

Table 10. Experimental GPS data format (using one vehicle as an example).

Car No.	Time	ID	wd	Status	v	Angle
YB000H6	2011/04/18 00:07:53	114.118347	22.574850	0	0	0
YB000H6	2011/04/18 00:08:01	114.118347	22.574850	0	0	0
YB000H6	2011/04/18 00:08:03	114.118347	22.574850	0	2	0
YB000H6	2011/04/18 00:08:33	114.118301	22.574301	0	25	4
YB000H6	2011/04/18 00:08:39	114.118286	22.573967	0	22	3
……	……	……	……	……	……	……

6.4. Experiments and Analysis

After selecting the above GPS trajectory data, a pre-processing process was used to remove wild values, e.g., (1) GPS geographical location data that exceeded the geographic location of Shenzhen, (2) acquisition data obtained for time periods between 00:00 and 06:00, (3) repeated data points, etc. The GPS data format after pre-processing is demonstrated in Table 11.

Table 11. GPS data format after the pre-processing procedure.

Car No.	Time	ID	wd	Status	Velocity	Angle
YB000H6	2011/04/18 07:59:52	114.150284	22.591333	0	0	4
YB000H6	2011/04/18 08:04:52	114.118301	22.589149	0	16	5
YB000H6	2011/04/18 08:05:22	114.149269	22.588949	0	10	5
YB000H6	2011/04/18 08:06:22	114.149284	22.588949	0	0	5
……	……	……	……	……	……	……

After the pre-processing process, the data was processed a second time. In this process, we simplified the data, extracting only the latitude and longitude information of the taxi from the pre-processed data. Then, the spatial data was included in the cluster operation using the DBSCAN algorithm following extraction of the latitude and longitude information of the taxi from the pre-processed data. The format of the processed GPS tracking data (here using the car YB000H6 as an example) is shown in Table 12, while a visualization of the taxi trajectory distribution before the clustering is shown in Figure 24.

Based on the above optimized latitude and longitude data, the parallel DBSCAN algorithm based on the Spark platform was used to cluster and obtain the areas most highly frequented by taxis within a fixed time period, i.e., the trajectory interest area. The processed results are shown in Table 13, and a visualization of the taxi trajectory distribution obtained from the parallel DBSCAN clustering algorithm is shown in Figure 25. In this figure, there are two generated clusters (color with black and gray). The two clustered areas show the areas more densely frequented by taxis. The light gray-colored areas represent noise; that is, it shows points that do not satisfy the DBSCAN algorithm cluster conditions.

Table 12. GPS data format after optimization processing.

Car No.	Longitude	Latitude
YB000H6	114.150284	22.591333
YB000H6	114.118301	22.589149
YB000H6	114.149269	22.588949
YB000H6	114.149284	22.588949
YB000H6	114.135834	22.578899
YB000H6	114.115616	22.602633
……	……	……

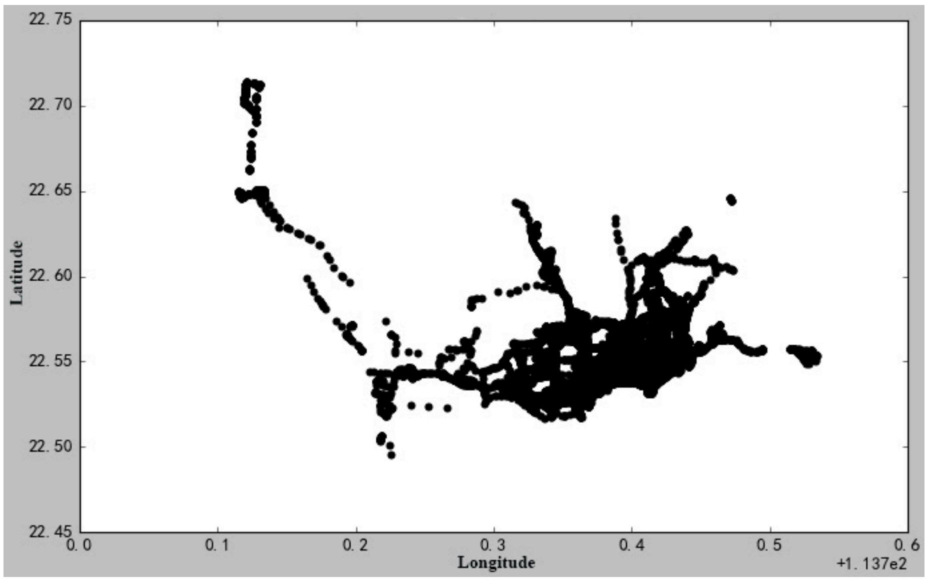


Figure 24. The trajectory distribution of the original data before clustering.

Table 13. Results format after clustering

Longitude	Latitude	Subordinated Clustering No.
114.150284	22.591333	1
114.118301	22.589149	1
114.149269	22.588949	1
114.149284	22.588949	1
114.135834	22.578899	2
114.115616	22.602633	0 (noise point)
......	......	......

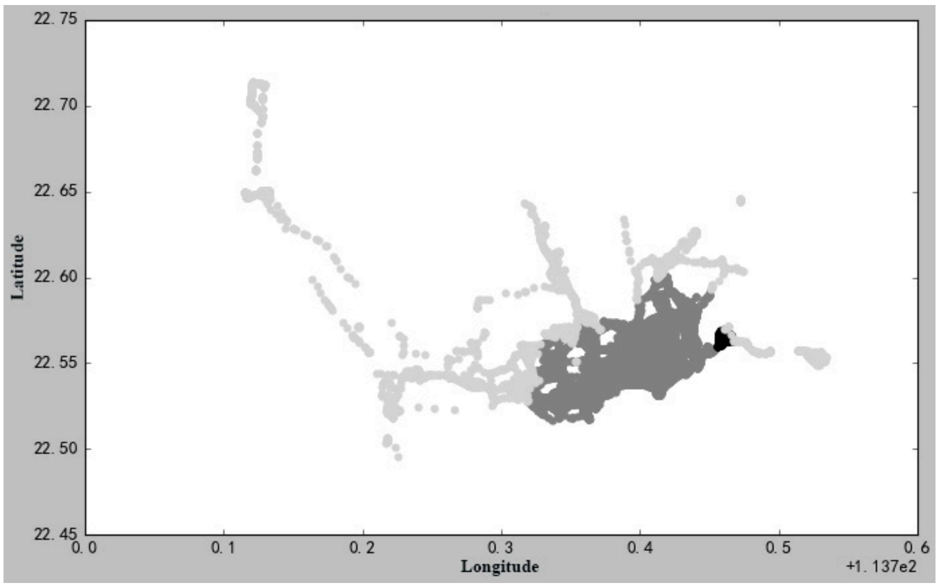


Figure 25. Visualization of the trajectory data distribution after the clustering operation.

All taxi trajectory GPS data were processed in the same way with the parallel DBSCAN algorithm on the Spark platform. After each node in the Spark cluster completed the data processing, the clustering results were merged to produce the final clustering result. The final clustering result shows dense areas of taxi traffic that appear at specific time slots in the city.

The final results of the entire clustering process show that 14 clusters were generated; that is, the city has 14 densely congested traffic areas in the designated time period. By obtaining the GPS coordinates of the central points of these clusters, we can locate the congested regional centers and find the corresponding congested sections. The coordinates of the central points of these clusters and the corresponding distribution of the congested urban sections are shown in Table 14. The results in the table are consistent with existing research findings for the Shenzhen Traffic Jamming Area [53], which shows that the parallel algorithm has achieved correct results and verifies the practicality of the parallel DBSCAN algorithm based on the Spark platform.

Table 14. Central points of congested areas.

Area No.	Longitude	Latitude	Congested Urban Sections
1	114.115838	22.585331	Bujiguan
2	114.163884	22.605763	Shawangguan
3	113.910531	22.552058	Nantouguan
4	114.092502	22.55025	Huaqiangbei Road
5	114.132781	22.560929	Dongmenzhong Road
6	114.094852	22.618864	Qingshuiqiao
7	113.990657	22.540561	Shennan Rd., Huaqiaocheng
8	114.243162	22.599692	Shenyan Rd., Yantianganqu
9	114.344448	22.601422	YanbaRd., Expressway Entrance
10	114.125419	22.605532	Buji Road
11	114.289037	22.748603	Shenhui Road
12	114.066885	22.611871	Bantian Wuhe Ave.
13	114.202285	22.558944	Luosha Rd.
14	113.910531	22.552058	#107 National Highway

At the same time, performance experiments based on two data sets were performed. One had 4000 K trajectory records and a data volume of 1054 MB, while the other had 2000 K trajectory records and a data size of 500 MB. The processing time required for these data sets using the traditional serial algorithm and the parallel algorithm on Spark platform was tested, and the results are shown in Table 15.

Table 15. Processing time comparison of the serial algorithm and the parallel algorithm with different data sets

Data Scale	Processing Platform	Selected Algorithm	Elapsed Time (h)
4000 K (1054 MB)	Single node	Serial DBSCAN Algorithm	Out of memory
	Spark cluster	Parallel DBSCAN Algorithm based on Spark	2.2
2000 K (500 MB)	Single node	Serial DBSCAN Algorithm	13
	Spark cluster	Parallel DBSCAN Algorithm based on Spark	0.4

From Table 15, it can be seen that when the data scale was 2000 K, the elapsed time of the serial algorithm was 13 h, which cannot satisfy real-time requirements. Compared with the serial algorithm, the parallel algorithm processing time was 0.4 h. The obtained speedup ratio is 32, which indicates that the parallel algorithm can effectively improve processing efficiency. When the spatial data scale reached 4000 K, the single-node method failed; however, the parallel algorithm's processing time was 2.2 h.

In summary, compared with traditional processing methods, the parallel DBSCAN algorithm based on the Spark platform can dramatically improve the efficiency of city congestion discovery. These experiments verify the high efficiency and practicability of the parallel algorithm proposed in this paper.

7. Conclusions and Future Directions

To resolve the problem of long processing times associated with large-scale data processed with the serial DBSCAN algorithm, in this paper, the big data processing platform Spark was used to design and implement a parallel DBSCAN clustering algorithm. The experimental results show that the DBSCAN algorithm achieves a stable acceleration effect on the single-node Spark platform compared with the serial program. In addition, on the Spark cluster platform, the Spark–Yarn deployment method is more suitable for applications that have many iterations compared to the Spark–Mesos method. However, there is still a place for optimizing the currently implemented parallel algorithm. For example, we could explore the data partitioning and merge the optimization strategies, or introduce the Spark map calculation method, etc., in order to further speed up the processing speed of the algorithm in the future.

Acknowledgments: This study was supported mainly by the Key Laboratory of Spatial Data Mining & Information Sharing of the Ministry of Education, Fuzhou University (Grant Nos. 2017LSDMIS03 & 2016LSDMIS06), Hubei Provincial Key Laboratory of Intelligent Geo-information Processing (China University of Geosciences) (Grant No. KLIGIP2016A03), the Engineering Research Center of Geospatial Information and Digital Technology (NASG) (Grant No. SIDT20170601), the Fundamental Research Funds for the Central Universities (Grant No. ZYGX2015J111), the National Key Research and Development program of China (Grant No. 2017YFB0504202), and also the National Science Foundation of the United States (Award No. 1251095, 1723292).

Author Contributions: Fang Huang, Lizhe Wang conceived and designed the experiments, and Fang Huang wrote the paper; Qiang Zhu, Du Jin performed the experiments; Jian Tao, Ji Zhou, Xiaocheng Zhou and Xicheng Tan analyzed the data and made key modifications to the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ester, M.; Kriegel, H.-P.; Sander, J.; Xu, X.W. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, Portland, OR, USA, 2–4 August 1996; pp. 226–231.
2. He, Y.; Tan, H.; Luo, W.; Mao, H.; Ma, D.; Feng, S.; Fan, J. MR-DBSCAN: A scalable MapReduce-based DBSCAN algorithm for heavily skewed data. *Front. Comput. Sci.* **2014**, *8*, 83–99. [CrossRef]
3. Ankerst, M.; Breunig, M.M.; Kriegel, H.-P.; Sander, J. OPTICS: Ordering points to identify the clustering structure. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, Philadelphia, PA, USA, 1–3 June 1999; pp. 49–60.
4. Chen, M.; Gao, X.; Li, H. Parallel DBSCAN with Priority R-tree. In Proceedings of the 2010 2nd IEEE International Conference on Information Management and Engineering, Chengdu, China, 16–18 April 2010; pp. 508–511.
5. Kryszkiewicz, M.; Lasek, P. TI-DBSCAN: Clustering with DBSCAN by Means of the Triangle Inequality. In *Rough Sets and Current Trends in Computing, Proceedings of the 7th International Conference, RSCTC 2010, Warsaw, Poland, 28–30 June 2010*; Szczuka, M., Kryszkiewicz, M., Ramanna, S., Jensen, R., Hu, Q., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 60–69.
6. Owens, J.D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A.E.; Purcell, T.J. A Survey of General-Purpose Computation on Graphics Hardware. *Comput. Graph. Forum* **2007**, *26*, 80–113. [CrossRef]
7. Wang, L.; Tao, J.; Ranjan, R.; Marten, H.; Streit, A.; Chen, J.; Chen, D. G-Hadoop: MapReduce across distributed data centers for data-intensive computing. *Future Gener. Comput. Syst.* **2013**, *29*, 739–750. [CrossRef]
8. Gan, L.; Fu, H.; Luk, W.; Yang, C.; Xue, W.; Huang, X.; Zhang, Y.; Yang, G. Solving the global atmospheric equations through heterogeneous reconfigurable platforms. *ACM Trans. Reconfig. Technol. Syst.* **2015**, *8*, 11. [CrossRef]
9. Li, L.; Xue, W.; Ranjan, R.; Jin, Z. A scalable Helmholtz solver in GRAPES over large-scale multicore cluster. *Concurr. Comput. Pract. Exp.* **2013**, *25*, 1722–1737. [CrossRef]
10. Liu, P.; Yuan, T.; Ma, Y.; Wang, L.; Liu, D.; Yue, S.; Kolodziej, J. Parallel processing of massive remote sensing images in a GPU architecture. *Comput. Inf.* **2014**, *33*, 197–217.

11. Chen, D.; Li, D.; Xiong, M.; Bao, H.; Li, X. GPGPU-Aided Ensemble Empirical-Mode Decomposition for EEG Analysis during Anesthesia. *IEEE Trans. Inf. Technol. Biomed.* **2010**, *14*, 1417–1427. [CrossRef] [PubMed]
12. Bernabé, S.; Lopez, S.; Plaza, A.; Sarmiento, R. GPU Implementation of an Automatic Target Detection and Classification Algorithm for Hyperspectral Image Analysis. *IEEE Geosci. Remote Sens. Lett.* **2013**, *10*, 221–225. [CrossRef]
13. Agathos, A.; Li, J.; Petcu, D.; Plaza, A. Multi-GPU Implementation of the Minimum Volume Simplex Analysis Algorithm for Hyperspectral Unmixing. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2014**, *7*, 2281–2296. [CrossRef]
14. Deng, Z.; Hu, Y.; Zhu, M.; Huang, X.; Du, B. A scalable and fast OPTICS for clustering trajectory big data. *Clust. Comput.* **2015**, *18*, 549–562. [CrossRef]
15. Chen, D.; Li, X.; Wang, L.; Khan, S.U.; Wang, J.; Zeng, K.; Cai, C. Fast and Scalable Multi-Way Analysis of Massive Neural Data. *IEEE Trans. Comput.* **2015**, *64*, 707–719. [CrossRef]
16. Huang, F.; Tao, J.; Xiang, Y.; Liu, P.; Dong, L.; Wang, L. Parallel compressive sampling matching pursuit algorithm for compressed sensing signal reconstruction with OpenCL. *J. Syst. Archit.* **2017**, *72*, 51–60. [CrossRef]
17. Yu, T.; Dou, M.; Zhu, M. A data parallel approach to modelling and simulation of large crowd. *Clust. Comput.* **2015**, *18*, 1307–1316. [CrossRef]
18. Wang, L.; Chen, D.; Liu, W.; Ma, Y.; Wu, Y.; Deng, Z. DDDAS-based parallel simulation of threat management for urban water distribution systems. *Comput. Sci. Eng.* **2014**, *16*, 8–17. [CrossRef]
19. Hu, C.; Zhao, J.; Yan, X.; Zeng, D.; Guo, S. A MapReduce based Parallel Niche Genetic Algorithm for contaminant source identification in water distribution network. *Ad Hoc Netw.* **2015**, *35*, 116–126. [CrossRef]
20. Kim, Y.; Shim, K.; Kim, M.-S.; Sup Lee, J. DBCURE-MR: An efficient density-based clustering algorithm for large data using MapReduce. *Inf. Syst.* **2014**, *42*, 15–35. [CrossRef]
21. Xu, X.; Jäger, J.; Kriegel, H.-P. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Min. Knowl. Discov.* **1999**, *3*, 263–290. [CrossRef]
22. Erdem, A.; Gündem, T.I. M-FDBSCAN: A multicore density-based uncertain data clustering algorithm. *Turkish J. Electri. Eng. Comput. Sci.* **2014**, *22*, 143–154. [CrossRef]
23. Böhm, C.; Noll, R.; Plant, C.; Wackersreuther, B. Density-based clustering using graphics processors. In Proceedings of the 18th ACM Conference on Information and Knowledge Management, Hong Kong, China, 2–6 November 2009; pp. 661–670.
24. Andrade, G.; Ramos, G.; Madeira, D.; Sachetto, R.; Ferreira, R.; Rocha, L. G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering. *Procedia Comput. Sci.* **2013**, *18*, 369–378. [CrossRef]
25. Dean, J.; Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
26. Böse, J.-H.; Andrzejak, A.; Högqvist, M. Beyond online aggregation: Parallel and incremental data mining with online Map-Reduce. In Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud, Raleigh, NC, USA, 26 April 2010; pp. 1–6.
27. He, Y.; Tan, H.; Luo, W.; Mao, H.; Ma, D.; Feng, S.; Fan, J. MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce. In Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems, Tainan, Taiwan, 7–9 December 2011; pp. 473–480.
28. Dai, B.R.; Lin, I.C. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. In Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, 24–29 June 2012; pp. 59–66.
29. Fu, Y.X.; Zhao, W.Z.; Ma, H.F. Research on parallel DBSCAN algorithm design based on mapreduce. *Adv. Mater. Res.* **2011**, *301–303*, 1133–1138. [CrossRef]
30. Kumar, A.; Kiran, M.; Prathap, B.R. Verification and validation of MapReduce program model for parallel K-means algorithm on Hadoop cluster. In Proceedings of the 2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT), Tiruchengode, India, 4–6 July 2013; pp. 1–8.
31. Anchalia, P.P.; Koundinya, A.K.; Srinath, N.K. MapReduce Design of K-Means Clustering Algorithm. In Proceedings of the 2013 International Conference on Information Science and Applications (ICISA), Suwon, Korea, 24–26 June 2013; pp. 1–5.

32. Xu, Z.Q.; Zhao, D.W. Research on Clustering Algorithm for Massive Data Based on Hadoop Platform. In Proceedings of the 2012 International Conference on Computer Science and Service System, Nanjing, China, 11–13 August 2012; pp. 43–45.
33. Nagpal, A.; Jatain, A.; Gaur, D. Review based on data clustering algorithms. In Proceedings of the 2013 IEEE Conference on Information & Communication Technologies, Thuckalay, India, 11–12 April 2013; pp. 298–303.
34. Lin, X.; Wang, P.; Wu, B. Log analysis in cloud computing environment with Hadoop and Spark. In Proceedings of the 2013 5th IEEE International Conference on Broadband Network & Multimedia Technology, Guilin, China, 17–19 November 2013; pp. 273–276.
35. Shukla, S.; Lease, M.; Tewari, A. Parallelizing ListNet training using spark. In Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval, Portland, OR, USA, 12–16 August 2012; pp. 1127–1128.
36. Lawson, D. *Alternating Direction Method of Multipliers Implementation Using Apache Spark*; Stanford University: Stanford, CA, USA, 2014.
37. Biglearn. Available online: http://biglearn.org/2013/files/papers/biglearning2013_submission_7.pdf (accessed on 15 December 2016).
38. Wang, B.; Yin, J.; Hua, Q.; Wu, Z.; Cao, J. Parallelizing K-Means-Based Clustering on Spark. In Proceedings of the 2016 International Conference on Advanced Cloud and Big Data (CBD), Chengdu, China, 13–16 August 2016; pp. 31–36.
39. Jiang, H.; Liu, Z. Parallel FP-Like Algorithm with Spark. In Proceedings of the 2015 IEEE 12th International Conference on e-Business Engineering, Beijing, China, 23–25 October 2015; pp. 145–148.
40. Jin, F.; Zhang, F.; Du, Z.H.; Liu, R.; Li, R.Y. Spatial overlay analysis of land use vector data based on Spark. *J. Zhejiang Univ.* **2016**, *43*, 40–44.
41. Xie, X.L.; Xiong, Z.; Hu, X.; Zhou, G.Q.; Ni, J.S. On Massive Spatial Data Retrieval Based on Spark. In *Web-Age Information Management, Proceedings of the WAIM 2014 International Conference on Web-Age Information Management, Macau, China, 16–18 June 2014*; Chen, Y.G., Balke, W.T., Xu, J.L., Xu, W., Jin, P.Q., Lin, X., Tang, T., Hwang, E.J., Eds.; Springer: Cham, Switzerland, 2014; pp. 200–208.
42. Suchanek, F.; Weikum, G. Knowledge harvesting in the big-data era. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 22–27 June 2013; pp. 933–938.
43. Li, X.Y.; Li, D.R. DBSCAN spatial clustering algorithm and its application in urban planning. *Sci. Surv. Mapp.* **2005**, *30*, 51–53.
44. Nisa, K.K.; Andrianto, H.A.; Mardhiyyah, R. Hotspot clustering using DBSCAN algorithm and shiny web framework. In Proceedings of the 2014 International Conference on Advanced Computer Science and Information System, Jakarta, Indonesia, 18–19 October 2014; pp. 129–132.
45. Çelik, M.; Dadaşer-Çelik, F.; Dokuz, A.S. Anomaly detection in temperature data using DBSCAN algorithm. In Proceedings of the 2011 International Symposium on Innovations in Intelligent Systems and Applications, Istanbul, Turkey, 15–18 June 2011; pp. 91–95.
46. Silva, T.L.C.D.; Neto, A.C.A.; Magalhaes, R.P.; Farias, V.A.E.D.; Macêdo, J.A.F.D.; Machado, J.C. Efficient and distributed DBScan algorithm using mapreduce to detect density areas on traffic data. In Proceedings of the 16th International Conference on Enterprise Information Systems, Lisbon, Portugal, 27–30 April 2014; pp. 52–59.
47. Adiba, M.E.; Lindsay, B.G. Database Snapshots. In Proceedings of the Sixth International Conference on Very Large Data Bases, Montreal, QC, Canada, 1–3 October 1980; pp. 86–91.
48. Wang, W.; Tao, L.; Gao, C.; Wang, B.F.; Yang, H.; Zhang, Z.A. C-DBSCAN Algorithm for Determining Bus-Stop Locations Based on Taxi GPS Data. In Proceedings of the 10th International Conference on Advanced Data Mining and Applications, Guilin, China, 19–21 December 2014; pp. 293–304.
49. Liu, C.K.; Qin, K.; Kang, C.G. Exploring time-dependent traffic congestion patterns from taxi trajectory data. In Proceedings of the 2015 2nd IEEE International Conference on Spatial Data Mining and Geographical Knowledge Services (ICSDM), Fuzhou, China, 8–10 July 2015; pp. 39–44.
50. Chen, X.W.; Lu, Z.H.; Jantsch, A.; Chen, S. Speedup analysis of data-parallel applications on Multi-core NoCs. In Proceedings of the IEEE 8th International Conference on ASIC, Changsha, China, 20–23 October 2009; pp. 105–108.

51. Ieda, H. Development and management of transport systems. In *Sustainable Urban Transport in an Asian Context*; Springer: Tokyo, Japan, 2010; pp. 277–335.
52. Yin, L. The Analysis of Our Urban Transportation Problem and the Research of Road Construction &map Planning Management. In Proceedings of the 2010 International Conference on E-Product E-Service and E-Entertainment, Henan, China, 7–9 November 2010; pp. 1–4.
53. Shao, Y.; Song, J.H. Traffic Congestion Management Strategies and Methods in Large Metropolitan Area: A Case Study in Shenzhen. *Urban Transp. China* **2010**, *8*. [CrossRef]