ABSTRACT

We present PS-DBSCAN, a communication efficient parallel DBSCAN algorithm that combines the disjoint-set data structure and Parameter Server framework in Platform of AI (PAI). Since data points within the same cluster may be distributed over different workers which result in several disjoint-sets, merging them incurs large communication costs. In our algorithm, we employ a fast global union approach to union the disjoint-sets to alleviate the communication burden. Experiments over the datasets of different scales demonstrate that PS-DBSCAN outperforms the PDSDBSCAN with 2-10 times speedup on communication efficiency.

We have released our PS-DBSCAN in an algorithm platform called Platform of AI (PAI) in Alibaba Cloud. We have also demonstrated how to use the method in PAI.

KEYWORDS

Density-Based clustering, Parallel DBSCAN, Parameter Server

1 INTRODUCTION

Clustering is an unsupervised data mining technology that divides a set of objects into subgroups by maximizing inter-group distances and minimizing intra-group distances. Usually, the clustering algorithm can be divided into four classes: partition-based, hierarchy-based, grid-based and density-based. Among all the clustering algorithms, DBSCAN [6], a density-based algorithm, is one of the most popular. The key idea of DBSCAN is that, for one point $p$ of the data set in $d$-dimensional space $\mathbb{R}^d$, if its neighborhood within the $d$-dimensional ball with radius $\varepsilon$, i.e., $\varepsilon$-neighborhood, contains at least $\text{minPoints}$ points, all the points inside this ball including $p$ formed a cluster. And $p$ is defined as a core point. Whenever a new core point is added to the cluster of $p$, all the points within the new core point’s $\varepsilon$-neighborhood are added to the cluster. This process goes on recursively in this way until all the clusters extended to their maximum size.

![Figure 1: The communication mode of MPI based DBSCAN.](image-url)
neighbor points 4 and 5, node 3 needs to route from node 8 and 6 to reach its parent node 11. Since MPI uses a peer-to-peer communication pattern, this process will generate a lot of merging requests. In general, the MPI based setting will have communication overhead when points from the same cluster are scattered over more partitions. And this scenario will be worse with the increase of worker number. More details of the communication process are in [14].

To overcome the communication bottleneck, we employ a parameter server framework [11] to implement parallel DBSCAN algorithm using the disjoint-set data structure mentioned in the paper [14]. The details about our Parameter Server framework can be found in [2, 17]. In our proposed algorithm, a global vector that records the class label of all data points is stored in the server processors. In worker processors, we employ a fast global union approach to union the disjoint-sets locally and push the resulted label vector to servers to update the global vector. This method alleviates the communication burden. Experiments over the datasets of different scales demonstrate that PS-DBSCAN outperforms PDSDBSCAN-D with 2-10 times speedup on communication efficiency.

The remainder of this paper is organized as follows. Section 2 describes the details of our parallel implementation of DBSCAN based on Parameter Server framework, referred to as PS-DBSCAN. In section 3 we compare the speedup of communication between our algorithm and the MPI based method PDSDBSCAN-D. Section 4 demonstrates the usage of our PS-DBSCAN in our PAI. In section 5, we survey the related work. Section 6 gives a brief conclusion and an overview of future work.

2 METHODOLOGY

Our PS-DBSCAN is built based on Alibaba parameter server system called KunPeng [17]. The KunPeng architecture is shown in Fig. 2. We use SDK of KunPeng to implement the distributed algorithm.

To illustrate our algorithm, we use Fig. 3 as a running example. Our algorithm starts by randomly dividing the input data points $P_{ts}$ into $p$ partitions and distributing them to $p$ workers, e.g., in Fig. 3(a), nodes 1 ~ 3 are in worker $w_1$ and nodes 4 ~ 8 are in worker $w_2$. In our setting, we have servers to maintain globalLabel, and local workers to maintain their own localLabel. Initially, all the workers perform clustering operations in parallel, where each worker uses QueryRadius to find each local data point’s $e$-nearest neighbors and MarkCorePoints accordingly. All the localCoreRecord will be synchronized with the servers to get globalCoreRecords. A LocalMerge operation is performed by each worker to create localCluster based on the $e$-nearest neighborhood information and globalCoreRecord. With the localCluster, all the workers start to label its local data points and communicate with servers to remove labeling conflicts. The steps PropagateMaxLabel, MaxReduceToServer, PullFromServer, GlobalUnion, and GetMaxLabel are performed iteratively until no labeling conflicts found. The key steps are discussed as follows.

- **MarkCorePoint**: A point $p$ is marked as a core point if its $e$-neighborhood size is at least minPoints.
- **PropagateMaxLabel**: This is a local clustering processing where all the nodes in the same cluster are labeled as the maximum local node id. As in Fig. 3(b), node 4 ~ 9 are labeled with id 9.
- **MaxReduceToServer**: A Synchronous Max Reduce operator is used to merge local clustering results with server results, where each node will be labeled as the maximum node id from all local workers. As in Fig. 3(c), node 6 takes 11 from $w_3$, i.e. max9$[w_2, 11[\{w_3\}]$.
- **PullFromServer**: This is a typical PS operator to pull results from the server. Interested readers can refer to [11] for details.
- **GlobalUnion**: This step starts from the maximum node id $N - 1$ to 0, for each node, if its root node id does not equal to the corresponding global label, we modify it to the global label. This is an effective way to compress the path of disjoint-set and redirect each local node to its root parent. For example, in Fig. 3(c), node 3 will directly link to its root node 11. Unlike Fig. 1, where node 3 needs to route from nodes 8 and 6 to link to 11. This is the key step to reduce communication burden.
- **GetMaxLabel**: This step is performed on the local cluster to label each data point with the maximum node id within a cluster. The detailed algorithm is described as in Fig. 4. After this step, all the local nodes are labeled as the maximum node within the cluster. As shown in Fig. 3(d), with this step, all the local nodes in $w_1$ are labeled as node 11.

We present our PS-DBSCAN method in Algorithm 1.

In a nutshell, comparing with the MPI-based PDSDBSCAN-D, our method has two advantages. First, each worker maintains a local cluster and we only generate merging requests when it has modified labels. This can help to reduce communication overhead. Second, with GlobalUnion, each data point is able to find its root parent directly without generating many merge requests. This makes our algorithm 2-10 times faster than the PDSDBSCAN-D.

3 EXPERIMENTS

We quantitatively evaluated our PS-DBSCAN here. We first designed experiments to examine the communication efficiency and speedup gain of our method comparing to the MPI-based PDSDBSCAN. Our method has better scalability than PDSDBSCAN where it shows good performance with up to 1600 CPU cores.
Input: A set of points $P$, distance threshold $\epsilon$ and density threshold $\text{minPoints}$.

Output: clusters of data points

1. **Input:** A set of points $P$, distance threshold $\epsilon$ and density threshold $\text{minPoints}$.
2. **Output:** clusters of data points
3. Randomly divide $P$ into partitions and distribute to workers.
4. **InitOnServer** ($\text{globalCoreRecord}$, $\text{globalLabel}$)
5. **InitOnWorker** ($\text{localCoreRecord}$, $\text{localLabel}$, $\text{localCluster}$)
6. for each worker $i$, parallel do
   7. for each point $p$ in $P_i$ do
      8. $p_n = \text{QueryRadius}(p, \epsilon)$
      9. $\text{localCoreRecord} = \text{MarkCorePoint}(p_n, p, \text{minPoints})$
   10. end for
11. **ReduceToServer** ($\text{localCoreRecord}$)
12. **PullFromServer** ($\text{globalCoreRecord}$)
13. $\text{localCluster} = \text{LocalMerge}(\text{localCoreRecord})$
14. **isFinish**, $\text{maxLabel} = \text{GetMaxLabel}(\text{localClusters}, \text{localLabel})$
15. while not **isFinish** do
16. **PropagateMaxLabel** ($\text{localClusters}$, $\text{localLabel}$, $\text{maxLabel}$)
17. **MaxReduceToServer** ($\text{localLabel}$)
18. **PullFromServer** ($\text{globalLabel}$)
19. **GlobalUnion** ($\text{localLabel}$)
20. **isFinish**, $\text{maxLabel} = \text{GetMaxLabel}(\text{localClusters}, \text{localLabel})$
21. end while
22. end for
23. **Return**: $\text{globalLabel}$

Figure 3: Sample workflow of PS-DBSCAN.

Algorithm 1 PS-DBSCAN

Setup. We evaluated our methods on a cluster where each computer node has 24 cores, 4 Intel Xeon E5-2430 hex-core processors, and 96GB memory. We implemented the PDSDBSCAN-D with open source code 2 on the cluster. As only single-threaded implementation of PDSDBSCAN-D is available, we limited to use one core in each computer node in our experiments. Note that, the cluster is used as a production cluster shared by many applications, to avoid the impact of other tasks, we repeated the experiments 6 times and take the mean results by ignoring the best and worst results.

Datasets. To investigate the performance of our PS-DBSCAN, we first generated two synthetic datasets: $D10m$ and $D100m$. $D10m$ has 10 million data points and each data point has an average of 25 directly density-reachable core points (or $\epsilon$-neighborhood), while $D100m$ has 100 million points and each has 15 $\epsilon$-neighborhood. We pre-computed pair-wise distance information for both of them.

Furthermore, we used two large real-world datasets from [9], one is Geo-tagged tweets, and the other BremenSmall that contains 3D-point cloud of an old town. The Tweets was obtained using the free twitter streaming API and contains location of all geo-tagged tweets, it consists of 16,602,137 2D-points. And BremenSmall is a set of 3D-point cloud of the old town of Bremen, which contains 2,543,712 points.

3.1 Examination of Communication Efficiency

Table 1 shows the communication time of MPI-based PDSDBSCAN-D and our PS-DBSCAN on synthetic and real-word datasets using

![Algorithm CheckAndGetMaxLabel]

**Algorithm CheckAndGetMaxLabel**

Input: cluster flag and label of each local point. 
Output: flag and the max label for each cluster.

1: CheckAndGetMaxLabel($\text{localClusters}$, $\text{localLabel}$, $\text{maxLabel}$)
2: $\text{isFinish} = \text{true}$
3: for each point $p$ in $\text{localClusters}$
4: $\text{pc} = p$.cluster_label
5: if $\text{localLabel}[\text{pc}] > \text{maxLabel}[\text{pc}]$
6: $\text{maxLabel}[\text{pc}] = \text{localLabel}[\text{pc}]$
7: $\text{isFinished} = \text{false}$
8: return $\text{isFinished}$

Figure 4: Pseudocode of CheckAndGetMaxLabel.
100, 200, 400, 800, and 1600 cores. Some important observations are discussed in order.

First, on all the datasets, the PDSDBSCAN-D tends to be slower than our PS-DBSCAN with the increase of CPU nodes. The reason is that PDSDBSCAN’s peer-to-peer communication pattern has communication overhead with a large number of CPU nodes.

Second, our PS-DBSCAN has a very limited number of communication iterations regardless of the growing number of CPU nodes. This is because our global union methods help to reduce the number of merging requests.

Third, MPI-based PDSDBSCAN-D is not stable with a large number of CPU nodes. For example, with 1600 CPU nodes, PDSDBSCAN fails to generate results, while our PS-DBSCAN still works. Furthermore, the PDSDBSCAN is severely affected by a large amount of the neighbors. For Tweets datasets with 169 $\epsilon$-nearest neighbors when $\epsilon = 0.01$ and 3600 neighbors when $\epsilon = 0.01$, PDSDBSCAN fails. Both of these problems make PDSDBSCAN not ideal for a very large data set.

Last but not least, on the largest dataset $D_{100m}$, the communication time of PS-DBSCAN decreases first and then increases as the nodes increases. Close examination shows, when the amount of the data points is too large, the total merge time will benefit from the increase in the number of nodes to some extent.

### Table 1: Communication time on all datasets.

| Cores | 100 | 200 | 400 | 800 | 1600 |
|-------|-----|-----|-----|-----|------|
| $D_{10m}$ (125million edges) | | | | | |
| PDSDBSCAN-D | 37.52 | 51.34 | 102.78 | 120.23 | NA |
| PS-DBSCAN | 9.23 | 10.18 | 11.12 | 11.4 | 24.78 |
| Speedup | 4.07x | 5.04x | 9.24x | 10.55x | |
| $D_{100m}$ (750million edges) | | | | | |
| PDSDBSCAN-D | 243.44 | 202.23 | 204.64 | 263.34 | NA |
| PS-DBSCAN | 71.81 | 56.18 | 39.24 | 46.54 | 52.83 |
| Speedup | 3.39x | 3.60x | 5.22x | 5.66x | |
| BremenSmall ($\epsilon=0.000001$, points=2,543,712) | | | | | |
| PDSDBSCAN-D | 17.15 | 48.17 | 61.08 | 70.11 | NA |
| PS-DBSCAN | 8.5 | 14.26 | 14.86 | 15.02 | 20.64 |
| Speedup | 2.01x | 3.38x | 4.11x | 4.67x | |
| Tweets ($\epsilon=0.0001$, points=16,602,137) | | | | | |
| PDSDBSCAN-D | NA | NA | NA | NA | NA |
| PS-DBSCAN | 13.04 | 13.31 | 17.74 | 18.95 | 21.16 |
| Speedup | (e=0.01,points=16,602,137) | | | | |
| PDSDBSCAN-D | NA | NA | NA | NA | NA |
| PS-DBSCAN | 23.75 | 26.52 | 26.7 | 29.79 | 36.16 |

### 3.2 Examination of Speedup Gains

We further examined the speedup gains of our PS-DBSCAN over PDSDBSCAN-D.

As in Fig. 5, with more CPU cores, our method has a larger speedup gain. In general, PS-DBSCAN outperforms the PDSDBSCAN with 2-10 times speedup on communication efficiency.

We have released our PS-DBSCAN in an algorithm platform called Platform of AI (PAI) in Alibaba Cloud. Below we demonstrate the usage of PS-DBSCAN in our cloud-based platform - PAI.

### 4 DEMONSTRATION

In this section, we demonstrate the usage of PS-DBSCAN in PAI. PAI provides an interface to interact with PS-DBSCAN component. The whole workflow is shown in Fig 7(a), where an input table named as “hxdb_sample_6” is linked to the PS-DBSCAN component “DBSCAN-1”. The output of the component is linked to an output table “hxdb_tmp_output-1”. With this workflow, the method automatically pulls the data from the input table and run the PS-DBSCAN algorithm, and the final results are stored in the output table.

We also provide an interface for users to tune the parameters, as in Fig 7(b). Specifically, we can tune the following parameters based on the interface.

- Input type: vector or linkage
- Dimension: input data dimension
Figure 7: PS-DBSCAN component in PAI.

- Epsilon: the distance threshold of DBSCAN
- minPts: the density threshold of DBSCAN
- input format: the number of input columns
- server number: the number of server nodes
- worker number: the number of worker nodes
- server cores: CPU cores for each server
- worker cores: CPU cores for each worker
- server memory: server memory
- worker memory: worker memory

We present the input and output tables of our PS-DBSCAN algorithm in Fig 8. The table is stored in MaxCompute platform. Interested readers can find the details here: https://www.aliyun.com/product/odps/.

We support two types of data as input:

- Vector: each node has an index and is represented by a vector, as shown in Fig 8(a).
- Linkage: each record in the table is a link between two nodes.

After running this algorithm, we can get the clustering result of our input data, as shown in Fig 8(b).

To test the PS-DBSCAN method, users can register PAI online via this link https://pai.base.shuju.aliyun.com/ and search for PS-DBSCAN in the search bar.

5 RELATED WORK

There are generally two lines of work for paralleling DBSCAN, one is on MapReduce-based big data platforms such as Apache Spark.
We have released our PS-DBSCAN in an algorithm platform called Platform of AI (PAI) in Alibaba Cloud and also demonstrated how to use it in PAL.

6 CONCLUSIONS

We presented a communication efficient parallel DBSCAN based on Parameter Server, named PS-DBSCAN. This algorithm uses a disjoint-set data structure from [14] and employed a fast global union approach based on parameter server framework to union the disjoint-sets to alleviate the communication burden. Our method does not require specific data pre-processing and is communication efficient compared to the competing MPI based DBSCAN methods.

ACKNOWLEDGMENTS

The authors would like to thank Jun Zhou, Xu Chen and Ang Wang for providing valuable comments and helpful suggestions. The authors would also like to thank anonymous reviews for their valuable comments.

REFERENCES

[1] Stefan Brecheisen, Hans-Peter Kriegel, and Martin Pfeifle. 2006. Parallel Density-Based Clustering of Complex Objects. In PAKDD.
[2] Cen Chen, Peilin Zhao, Longli Li, Jun Zhou, Xiaolong Li, and Minghui Qiu. 2017. Locally Connected Deep Learning Framework for Industrial-scale Recommender Systems. In WWW'17.
[3] Min Chen, Xuedong Gao, and Huifei Li. 2010. Parallel DBSCAN with priority r-tree. In The 2nd IEEE International Conference on Information Management and Engineering (ICIME) 508–511.
[4] Massimo Coppola and Marco Vannucci. 2002. High-performance data mining with skeleton-based structured parallel programming. Parallel Comput. 28, 5 (2002), 793–813.
[5] Irving Cordova and Teng-Sheng Moh. 2015. DBSCAN on Resilient Distributed Datasets. (2015). 531–540.
[6] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In SIGKDD. 226–231.
[7] Yanxiang Fu, Weizhong Zhao, and Huifang Ma. 2011. Research on parallel DBSCAN algorithm design based on mapreduce. In Advanced Materials Research. 1133–1138.
[8] Junhao Gao and Yufei Tao. 2015. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In SIGMOD 519–530.
[9] Markus Götz, Christian Bodenstein, and Morris Riedel. 2015. HPDBSCAN: Highly Parallel DBSCAN. In Proc. of the Workshop on Machine Learning in High-Performance Computing Environments (MLHPC ’15). Article 2. 10 pages.
[10] Yoabin He, Haoyu Tan, Wuman Luo, Huaqian Mao, Di Ma, Shengzhong Feng, and Jianping Fan. 2011. MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce. In KDD ’11. 473–480.
[11] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amir Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiung Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In OSDI. 583–598.
[12] Aliaksei Litouka. 2014. Spark DBSCAN source code. (2014). https://github.com/alitouka/spark_dbscan
[13] Helmut Neukirchen. 2016. Survey and Performance Evaluation of DBSCAN Spatial Clustering Implementations for Big Data and High-Performance Computing Paradigms. (2016).
[14] Mostofa Ali Patwary, Diaana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. 2012. A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-set Data Structure. In Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis.
[15] Mansour Raad. 2016. DS-DBSCAN On Spark source code. (2016). https://github.com/mraad/ps-dbscan
[16] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. 2002. A fast parallel clustering algorithm for large spatial databases. In High Performance Data Mining. 263–290.
[17] Jun Zhou, Xiaolong Li, Peilin Zhao, Chaoshao Chen, Longli Li, Xinming Yang, Qing Cui, Jin Yu, Xu Chen, Yi Ding, and Yuan Alan Qi. 2017. KunFang: Parameter Server Based Distributed Learning Systems and Its Applications in Alibaba and Ant Financial. In KDD. 1693–1702.

9Mansour Raad. 2016. https://github.com/mraad/