μDBSCAN: An Exact Scalable DBSCAN Algorithm for Big Data Exploiting Spatial Locality

Aditya Sarma, Poonam Goyal, Sonal Kumari, Anand Wani, Jagat Sesh Challa, Saiyedul Islam, Navneet Goyal
ADAPT Lab, Dept. of Computer Science, Pilani Campus
Birla Institute of Technology & Science, Pilani, Pilani, India
{f2013079, poonam, sonal.kumari, h20180143, jagatsesh, sislam, goel}@pilani.bits-pilani.ac.in

***Abstract*—DBSCAN is one of the most popular and effective clustering algorithms that is capable of identifying arbitrary-shaped clusters and noise efficiently. However, its super-linear complexity makes it infeasible for applications involving clustering of Big Data. A major portion of the computation time of DBSCAN is taken up by the neighborhood queries, which becomes a bottleneck to its performance. We address this issue in our proposed micro-cluster based DBSCAN algorithm, μDBSCAN, which identifies core-points even without performing neighbourhood queries and becomes instrumental in reducing the run-time of the algorithm. It also significantly reduces the computation time per neighbourhood query while producing exact DBSCAN clusters. Moreover, the micro-cluster based solution makes it scalable for high dimensional data. We also propose a highly scalable distributed implementation of μDBSCAN, μDBSCAN-D, to exploit a commodity cluster infrastructure. Experimental results demonstrate tremendous improvements in performance of our proposed algorithms as compared to their respective state-of-the-art solutions for various standard datasets. μDBSCAN-D is an exact parallel solution for DBSCAN which is capable of processing massive amounts of data efficiently (1 billion data points in 41 minutes on a 32 node cluster), while producing a clustering that is same as that of traditional DBSCAN.**

***Index Terms*—Density-based Clustering, Exact Clustering Algorithm, Spatial Locality, Optimized Neighborhood Query, Big Data, Distributed Computing**

I. Introduction

Today, data size is increasing at an unprecedented rate. Clustering on ever-increasing data has become pervasive and requires scalable and efficient solutions. DBSCAN [1] is one of the most popular density-based clustering algorithm. In density-based clustering, regions of high density are interspersed by regions of low density. DBSCAN is vastly used in various domains and applications that include - astronomy, cloud computing, bio-informatics, video surveillance, anomaly detection, stock market prediction, etc [2]. The key parameters of DBSCAN are: ϵ and $MinPts$. The clustering is determined on basis of the given values to the parameters and can greatly vary with them. DBSCAN performs two major processing steps. The first step is execution of an ϵ-neighborhood (region) query for each data point in the dataset. This results in $O(n^2)$ time complexity in worst case, where n is number of data points. Use of spatial indexing structures reduces its time complexity to $O(n \log n)$. The second step is cluster formation process where data points are accessed in sequential order, limiting its scalability for parallel solutions. With increase in volume and dimensionality, the cost of neighborhood queries and sequential processing of points, become a bottle-neck and thus hindering the utility of DBSCAN for big data. Motivated by this, many algorithms [3]–[7] have been proposed to address these issues and enhance its performance.

A scalable parallel algorithm for DBSCAN [5], proposed in 2012, breaks the sequential data access pattern of traditional DBSCAN using disjoint-set data structure. However, the algorithm does not optimize the performance of DBSCAN. Couple of sampling based parallel algorithms [8], [9] have also been proposed for DBSCAN that are based on approximate neighborhood query computations. They claim to get good performance in comparison to [5] by compromising the clustering quality. Recently, a couple of parallel algorithms [4], [10] have been reported that accelerate the neighborhood querying while preserving the clustering quality. These algorithms (HPDBSCAN [10] and GridDBSCAN [4]) use gridding to divide the data-space in smaller cells and then optimize the neighborhood query by limiting the search to the neighboring cells. However, the number of cells chosen in them is exponential with respect to dimensionality, and thus give a degraded performance for high dimensional data. GridDBSCAN also reduces the number of neighborhood queries (up to 15%) by exploiting gridding based on ϵ. G-DBSCAN [3], a sequential algorithm, optimizes neighborhood queries using groups (initial clusters) and reduces the number of distance calculations by eliminating noise points early, while constructing initial clusters.

We propose a novel, simple yet effective *micro-cluster* based DBSCAN algorithm, μDBSCAN, which addresses both the issues - a total of n number of queries (n is order of billions) and sequential data access pattern, in a proactive way. 1) It identifies core points even without performing ϵ-neighborhood queries and thus saves a substantial fraction of queries, while providing exact DBSCAN clusters. Note that by exact clustering we mean that the number of cluster, the number of core points in each cluster are exactly same as obtained by traditional DBSCAN 2) Its distributed approach breaks the sequential data access pattern and forms tiny clusters (known as micro-clusters) in the entire space, while restricting their number, using spatial locality. μDBSCAN also optimizes neighborhood query processing time by reducing the search space in two ways: (i) by exploiting reachability within micro-clusters and (ii) by using a two level *R-tree*, referred to as *μR-tree*. We also propose a parallel μDBSCAN called μDBSCAN-D, to exploit distributed memory architectures (typically cluster of computing nodes). We use data-parallel

strategy where data is divided equally amongst the computing nodes using spatial data distribution, for independent execution of μDBSCAN at each computing node. Local clusterings are then merged to get the global clustering. The main contributions of the paper are as follows:

- We propose a novel, simple and efficient micro-cluster based DBSCAN, μDBSCAN, which reduces the number of neighborhood queries significantly. Our novelty lies in smart identification of core points with formal justification. We also reduce the average time complexity of DBSCAN to $O(n \log m + n \log r)$, where n, m and r denote number of data points, *micro-clusters* and average number of points in one *micro-cluster* respectively (see Section IV-C). The experimental analysis shows that μDBSCAN outperforms the existing baselines approaches. It also shows that μDBSCAN is able to save upto 96% of the queries in the datasets used for experimentation (see Section VI).
- We theoretically prove the correctness of μDBSCAN from the perspective of DBSCAN clustering. Our analysis guarantees that the clustering obtained by μDBSCAN is exactly same as that of traditional DBSCAN (see Section IV-D).
- A scalable distributed implementation of μDBSCAN, μDBSCAN-D, is also proposed for exploiting distributed memory architecture that can cluster data points of billion scale. μDBSCAN-D achieves lowest run-time and gives scalable performance with increasing processing elements and data size. This is mainly due to efficient parallelization strategy and minimal parallelization overhead. Experimental analysis shows that μDBSCAN-D outperforms the best existing distributed DBSCAN implementations and clusters 1B data points in 41 minutes over 32 computing nodes.

II. The DBSCAN Algorithm

DBSCAN [1] is a density-based clustering algorithm, which finds arbitrary shaped clusters using two density parameters: ϵ & $MinPts$. DBSCAN performs ϵ-neighborhood query for each point of the dataset X and thus requires distance computations between every pair of points, resulting in $O(n^2)$ complexity. n is the total number of points in X. DBSCAN labels each point $\in X$ as *core*, *border*, or *noise* using the ϵ-neighborhood queries. A *core* point x initiates a cluster, and the cluster is expanded by repeated neighborhood queries on each point in the ϵ-neighborhood of x and the neighborhoods of points retrieved, and so on until no core point is found in any of the neighborhoods. This completes the expansion of a cluster. The next random point from the remaining unprocessed points is visited to extract another cluster and this process continues until all the points are processed. The pseudo-code of DBSCAN that uses *Union-Find* structure [5] is given in Algorithm 1. Note: UNION(x,q) assigns q to the cluster to which x belongs to. Some definitions of concepts used in DBSCAN are given below using points p, $q \in X$:

ϵ-neighborhood ($N_\epsilon(p)$): $\forall q \in$X, if DIST$(p, q) < \epsilon$, then $q \in N_\epsilon(p)$

Core Point: p is core if, $|N_\epsilon(p)| \geq MinPts$.

Directly-density-reachable (ddr): p is directly-density-reachable to q, if $p \in N_\epsilon(q)$ and q is core. Denoted as $p\ ddr\ q$.

Algorithm 1: UNION-FIND DBSCAN

```
1  procedure DBSCAN ()
       Input  : Data List X, ε, MinPts
       Output: A Set of Clusters in Union-Find Structure
2      foreach Point x ∈ X do
3          parent(x) ← x;
4      foreach Point x in X do
5          x.Nbhrs ← GET-EPS-NEIGHBORHOOD (x, ε);
6          if |x.Nbhrs| ≥ MinPts then
7              mark x as a core point;
8              for q ∈ x.Nbhrs do
9                  q.Nbhrs ← GET-EPS-NEIGHBORHOOD (q, ε);
10                 if |q.Nbhrs| > MinPts then UNION (x, q) ;
11                 else if q is not yet assigned to any other cluster then
12                     UNION (x, q);
```

Density-Reachable (dr): p is density-reachable from q, if there is a chain of points $p_1, p_2, ..., p_n \in X$, $p_1 = q$ and $p_n = p$ such that $\forall i, p_{i+1}\ ddr\ p_i$. Denoted as $p\ dr\ q$.

Density-Connected (dc): p is density-connected to q, if there is a point $o \in X$, s.t., $p\ dr\ o$ and $q\ dr\ o$. Denoted as $p\ dc\ q$.

Border Point: p is a border point, if it is not a core point, but $p\ ddr\ q$, where q is a core point.

Noise Point: p is a noise point, if it is not a core point and is $not\ ddr$ from any core point.

Cluster: A DBSCAN cluster, C, is a maximal set of density-connected points [1], i.e., C should satisfiy three conditions:

- Condition 1 - *Maximality*: For each pair p and q, if $p \in C$ and $q\ dc\ p$ with respect to ϵ and $MinPts$, then $q \in C$.
- Condition 2 - *Connectivity*: For each pair p and q, if p, $q \in C$, then $p\ dc\ q$ holds true with respect to ϵ and $MinPts$.
- Condition 3 - *Noise*: same as definition of a noise point.

III. Related Work

Literature reveals many variants of DBSCAN proposed using various optimizations that include - gridding [4], [11], [12], reduced neighborhood queries [6], [7], [13], exploiting spatial information [3], [14], etc. The existing approaches can be broadly divided into two categories based on the clustering results obtained: *exact clustering* and *approximate clustering*. For a given dataset, the algorithms that produce - 1) "same set of core points"; 2) "same core point to cluster membership"; and 3) "same number of clusters"; as that of traditional DBSCAN are said to produce exact clustering. The algorithms that don't meet the above criteria are said to produce approximate clustering. Note that change in ordering of points in the dataset doesn't change the above values.

In the last decade, a few DBSCAN variants have been proposed to improve the run-time performance. However, most of these algorithms [6], [7], [13], [15] produce approximate clustering. QIDBSCAN [6], DBSCALE [7], ODBSCAN [15] etc. perform neighborhood queries by considering some representative points in the axis direction that lie at the ϵ-extended spherical boundary of a core-point. These algorithms do not satisfy the condition of maximality of DBSCAN (see Section II) and thus do not produce exact clustering.

A graph-based DBSCAN, (G-DBSCAN) [3], is presented that optimizes the neighborhood query computations by making groups of data points (initial clusters) and pruning noise points for a given density parameters. DBSCAN is applied using the group information. It performs neighborhood queries

for all the points in the dataset (except noise points) in an optimized way and obtains exact DBSCAN clustering. The authors claim its time complexity to be $O(nd)$, where d is the average number of points in 5*ϵ region of a point. However, d can be as large as n. AnyDBC [16] is an anytime density-based clustering algorithm which gives clustering results at anytime with some approximation. The algorithm executes in an iterative fashion. The greater the number of iterations, the greater the accuracy/closeness of the clustering to the traditional DBSCAN. Exact DBSCAN clustering is obtained after a large number of iterations.

Since sequential DBSCAN is unable to cluster large datasets efficiently, many researchers have focused on developing parallel solutions [4], [5], [10], [17], [18]. Early parallel implementations of DBSCAN used master-slave model, where master performs computations sequentially [17], [18]. This hampers their scalability.

An efficient parallel DBSCAN algorithm, PDSDBSCAN-D, was then proposed [5], which broke the sequential data access pattern of classical DBSCAN by using a disjoint-set data structure known as union-find [19]. They presented parallel versions for both distributed memory and shared memory architectures and has experimentally shown their scalability with increase in number of processors. Based on PDSDBSCAN-D, the same authors have presented two parallel algorithms known as Pardicle [8] and BD-CATS [9] for clustering massive datasets. However, they produce approximate clustering.

Recently, a grid-based DBSCAN (GridDBSCAN [4]) along with different parallelization strategies have been proposed. GridDBSCAN optimizes run-time performance of DBSCAN by reducing: 1) total number of neighborhood queries; and 2) search space for each query; while producing exact DBSCAN clustering. The authors claim to save up to 15% of the neighborhood queries. They also show that the parallel approaches proposed for distributed memory, shared memory and hybrid architectures scale well with increase in number of processing elements. Another grid-based parallel DBSCAN (HPDBSCAN) [10] has been proposed that first distributes the data in arbitrary order for gridding & indexing, and then redistributes it to the computing nodes for clustering on the basis of load balancing cost heuristic. It makes use of cells to reduce the search space for efficient querying but doesn't reduce the number of neighborhood queries.

Apart from MPI based parallel algorithms, a few parallel versions have also been proposed for MapReduce, Spark and GPGPU based frameworks. MR-DBSCAN [20] gives a 4-stage MapReduce implementation of DBSCAN that uses a quick partitioning strategy for large scale non-indexed data. Some other MapReduce implementations of DBSCAN [21], [22] are also reported in literature. Mr. Scan algorithm [23] is a hybrid (CPU+GPGPU) algorithm which was first introduced for GPGPU systems, that produces approximate clustering. Authors have modified the DBSCAN algorithm to find more dense regions and infer their membership in a cluster without processing the points belonging to these dense regions. Merging is done on the basis of some representative points from each cluster and thus resulting in an approximate clustering. RP-DBSCAN [24] is another approximate parallel implementation of DBSCAN that is based on Spark. It uses random data distribution and thus saves the overhead of spatial partitioning. However, this leads to increase in overall execution time.

The existing sequential and parallel DBSCAN algorithms are either approximate or their optimizations are limited, and hence there is a scope to further improve the algorithm. One can optimize the query search, omit unnecessary neighborhood queries, and apply good parallelization strategy that makes the algorithm suitable for big data. The aim of this work is to give a sequential DBSCAN that reduces the neighborhood queries substantially while giving exact clustering, and enable its efficient scalable parallelization.

IV. The proposed Algorithm: μDBSCAN

A. Overview of μDBSCAN

The μDBSCAN algorithm first scans the data points, forms micro-clusters, builds a hierarchical structure known as μR-tree, and then processes the micro-clusters to get preliminary clusters. These micro-clusters and the points in the dataset are further processed to get final clustering. A micro-cluster is a hyper-sphere of radius ϵ and center as a point p, along with points in ϵ neighborhood of p (see Fig. 2). It is denoted by either MC or $MC(p)$, if reference of its center p is required. Another point q would belong to $MC(p)$, if DIST(q, p) $< \epsilon$. Note that $MC(p)$.*center* is p itself. Also, note that any given point r in the dataset can belong to only one micro-cluster only. μDBSCAN exploits these MCs to reduce the number of neighborhood queries and then performs computations on the remaining unprocessed individual data points. According to the type of the MC to which they belong to, some points in the dataset are declared as core without even performing neighborhood queries, thus saving on the number of queries. Please note that the notion of micro-cluster used in this paper is different from that used for stream clustering [25].

μR-tree is a two-level R-tree that stores MCs in order to exploit spatial locality. The structure of μR-tree is illustrated in Fig. 1. The first level R-tree stores MCs and the second level consists of multiple R-trees known as Auxiliary R-trees (*AuxR-trees*), one each for each MC. An AuxR-tree stores points belonging to its respective MC. Each leaf node of the first level R-tree acts as the root to the corresponding AuxR-tree. For more details on R-trees, please refer to [26]. The design of μR-tree enables reduction in search space for executing an ϵ-neighborhood query, thus leading to reduction in its cost.

A *union-find* data structure is used to efficiently merge points into clusters, and also for storing clustering information [19]. The usage of the union-find data structure plays out well in the distributed scenario where there is a need to aggregate local clustering results into a global aggregate.

B. Algorithm Description

The entire μDBSCAN algorithm (Algorithm 2) can be broadly divided into four steps: 1) μR-tree construction and discovery of preliminary clusters; 2) Finding reachable MCs and their filtration; 3) Clustering and dynamic identification of core points without performing neighborhood queries; 4) Establishing final connections. We explain them as follows:

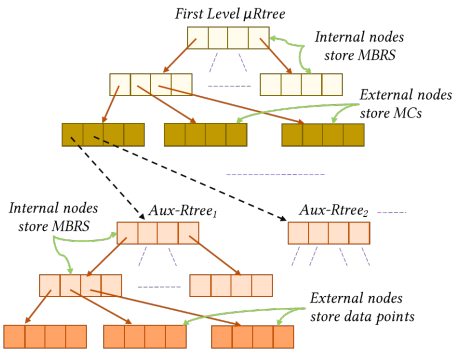


Fig. 1: μR-tree Structure

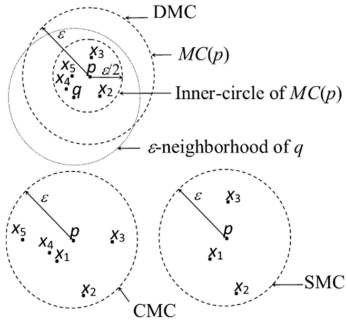


Fig. 2: CMC, DMC and SMC for MinPts=5 and radius=ϵ

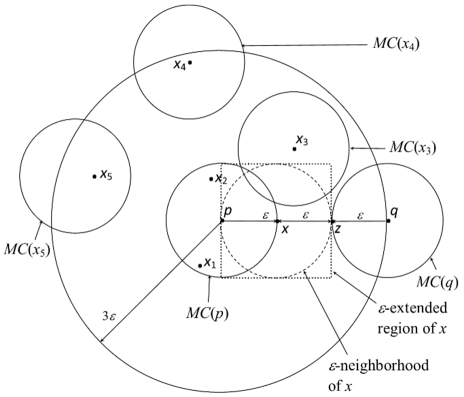


Fig. 3: Reachable micro-clusters of $MC(p)$

1) μR-tree construction and discovery of preliminary clusters: We build the first level μR-tree by inserting MCs into it (Algorithm 3). A point p is inserted into an existing MC Z, if DIST (p, $Z.center$) $< \epsilon$ (note that every MC maintains a list of its points). If there is no such MC present in the tree, a new MC is constructed with p as its center and is inserted into first level μR-tree. The insertion and search are similar to those of traditional R-tree. To optimize the number of MCs, we do not create a new MC if there exists any MC which is less than 2ϵ apart from the current point p. In this case, we insert p into an $unassignedList$ for its post assignment. After completing the first round of MC construction, we insert points of the $unassignedList$ one by one into an existing MC if possible or a new MC is created and inserted. This completes the construction of first level of μR-tree. An *AuxR-tree* is constructed under each MC for its points.

The two level R-tree reduces the search cost as it breaks the downward propagation of MBR (minimum bounding rectangle) overlaps in the tree nodes. The overlap would propagate till leaf node in case of single R-tree of size n. Therefore, the search space and distance calculations during point neighborhood queries are reduced. The proposed algorithm initially works on MCs rather than points. To reduce the cost of processing of MCs, we ensure that the number of MCs remains limited. To limit the no of MCs, we use 2ϵ condition to create a new MC as explained in the above paragraph. Few relevant definitions:

(i) *Inner-Circle (IC)*: Inner-circle of a $MC(p)$, $IC_{MC(p)}$, is defined as the set of points S such that $\forall s \in S, s \neq p$, DIST($s$, p) $\leq \epsilon/2$.
(ii) *Dense Micro-Cluster (DMC)*: An MC is a dense micro-cluster if $|IC_{MC}| \geq MinPts$.
(iii) *Core Micro-Cluster (CMC)*: An MC is a core micro-cluster if $|MC| \geq MinPts$.
(iv) *Sparse Micro-Cluster (SMC)*: An MC is a sparse micro-cluster if $|MC| < MinPts$.
(v) *Reachable micro-clusters*: An $MC(q)$ is called a reachable micro-cluster to $MC(p)$ if DIST(p, q)$\leq$ 3ϵ. This definition is symmetric. Fig. 3 shows four reachable MCs for $MC(p)$.

In the process of discovering preliminary clusters, each MC is processed and classified into one of the three types as a DMC, CMC or SMC, as per the definitions given above (Algorithm 4). According to the type of the MC, we do the following:

I. MC is DMC: All the points $\in IC_{MC}$ are marked as *wndq-core* (*w*ithout *n*eighborhood *q*uery) and are added to $wndqCorelist$ for later use. A preliminary cluster is formed by doing a union operation of all the points in MC with its center. All the points $\in MC$ but outside IC_{MC}, are considered as border for time being.
II. MC is CMC: $MC.center$ is marked as a *wndq-core* and a union is performed on all the points of MC with its center. We add center of MC to the $wndqCorelist$. All the points in MC except the center are considered as border for time being.
III. MC is SMC: Do nothing.

The following Lemma justifies the method of core point identification without performing neighborhood query:

Lemma 1. *If $|IC_{MC}| > MinPts$ then $\forall q \in IC_{MC}$, q is a core point.*

Proof. If $|IC_{MC}| > MinPts$, then for each $q \in IC$, $N_\epsilon(q)$ contains whole of IC_{MC} as maximum distance between any two points $\in IC$ is $< \epsilon$. Thus, $|N_\epsilon(q)| \geq |IC|$ (see Fig. 2). □

Lemma 2. *The center of a CMC is a core point.*

Proof. By definitions of micro-cluster and CMC. □

2) Finding reachable micro-clusters and their filtration: In order to perform correct DBSCAN clustering, we need to perform ϵ-neighborhood queries for all those points that were not tagged as *wndq-core* earlier. To reduce the cost of these queries, we compute a set of reachable MCs for every MC in

Algorithm 2: μDBSCAN

```
1  procedure μDBSCAN ()
       Input  : Data List X, ε, MinPts
       Output: A set of clusters in a union-find data structure UF
2      μR-tree μR;  Micro-Clusters List MCList;
3      Points List wndqCorelist, noiseList;
4      BUILD-MICRO-CLUSTERS (X, μR, MCList);
5      foreach Z in MCList do
6          PROCESS-MICRO-CLUSTERS (Z);
7          FIND-REACHABLE-MC (Z, μR.root);
8      PROCESS-REM-POINTS (X);
9      POST-PROCESSING-CORE (wndqCorelist);
10     POST-PROCESSING-NOISE (noiseList);
```

Algorithm 3: BUILD-MICRO-CLUSTERS

```
1  procedure BUILD-MICRO-CLUSTERS ()
       Input  : Data List X, μR-tree μR, MCs List MCList
       Output: Populates μR and MCList with MCs
2      Points List unassignedList;
3      foreach Point p ∈ X do
4          if μR is empty then
5              Z = CREATEMC (p);  μR.INSERT (Z);
                 MCList.INSERT (Z);
6          else
7              pFlag = PROCESS-POINT (p, μR.root,
                 unassignedList);
8              if pFlag == 0 then
9                  Z = CREATE-MC (p); μR.INSERT(Z);
                     MCList.INSERT(Z);
10     foreach Point p ∈ unassignedList do
11         uFlag = PROCESS-UNASSIGNED-POINT (p, μR.root);
12         if uFlag == 0 then
13             Z = CREATE-MC (p);  μR.INSERT (Z);
                 MCList.INSERT (Z);
14     Create Aux-Rtree for each MC of MCList;
15 procedure PROCESS-POINT ()
       Input  : Point p, Root of μR-tree node, Points List
                unassignedList
       Output: Point p processed appropriately
16     pFlag = 0;
17     if node is internal then
18         foreach Entry e ∈ node do
19             if p lies in e.MBR then
20                 pFlag = PROCESS-POINT (p, node.childNode,
                     unassignedList);
21             if pFlag == 0 && reg_2ϵ(p) overlaps with e.MBR then
22                 pFlag = PROCESS-POINT (p, node.childNode,
                     unassignedList);
23             if pFlag == 1 then return 1;
24     else if node is external then
25         foreach Entry e ∈ node do
26             if DIST (p, e.MC.center) ¡ ϵ then
27                 INSERT-POINT-INTO-MC (p, Y);  return 1;
28         foreach Entry e ∈ node do
29             if DIST (p, e.MC.center) ¡ 2ϵ then
30                 unassignedList.ADD (p);  return 1;
31     return pFlag;
32 procedure PROCESS-UNASSIGNED-POINT ()
       Input  : point p, root of μR-tree node
       Output: unassigned point p processed appropriately
33     uFlag = 0;
34     if node is internal then
35         foreach entry e ∈ node do
36             if reg_ϵ(p) overlaps with e.MBR then
37                 uFlag = PROCESS-UNASSIGNED-POINT (p,
                     node.childNode);
38                 if uFlag == 1 then return 1;
39     else if node is external then
40         foreach entry e ∈ node do
41             if DIST (p, e.MC.center) ¡ ϵ then
42                 INSERT-POINT-INTO-MC (p, Y);  return 1;
43     return uFlag;
```

Algorithm 4: PROCESS-MICRO-CLUSTERS

```
1  procedure PROCESS-MICRO-CLUSTERS ()
       Input  : MC Z
       Output: Points in Z tagged appropriately
2      if |Z.IC| > MinPts (// If Z is DMC) then
3          foreach Point x in Z.IC do
4              x.core = TRUE;  wndqCorelist.ADD (x);  x.tag =
                 wndq-core;
5          foreach Point p in Z do
6              UNION (Z.center, p);
7      else if |Z| > MinPts (// If Z is CMC) then
8          Z.center.core = TRUE;  wndqCorelist.ADD (Z.center);
9          Z.center.tag = wndq-core;
10         foreach Point p in Z do
11             UNION (Z.center, p);
```

Algorithm 5: FIND-REACHABLE-MC

```
1  procedure FIND-REACHABLE-MC ()
       Input  : MC Z, Root of μR-tree node
       Output: List of reachable MCs populated for Z
2      if node is internal then
3          foreach Entry e of node do
4              if e.childNode.MBR overlaps with reg_ϵ(Z.center) then
5                  FIND-REACHABLE-MC (Z, e.childNode);
6      else if node is external then
7          foreach Entry e of node do
8              if DIST (e.MC.center, Z.center) ≤ 3ϵ then
                 Z.reachList.ADD (Z) ;
```

$MCList$. So, this will restrict the search space for each query, as it will be sufficient to search only in reachable MCs. This is substantiated by the following lemma.

Lemma 3. *For the ϵ-neighborhood query of any point $x \in MC(p)$, it is sufficient to search only in reachable micro-clusters of $MC(p)$.*

Proof. If there exists a $MC(q)$ such that ϵ-neighborhood query of x needs to be searched in $MC(q)$ then DIST(x, z)$\leq \epsilon$ for some $z \in MC(q)$. This implies that DIST(x, q) is at most 2ϵ. Maximum distance occurs when x is on the boundary of $MC(p)$ and $MC(q)$ touches $N_\epsilon(x)$ (see Fig. 3). Therefore, DIST(p, q)$\leq 3\epsilon$. □

For an MC, Z, a set of reachable MCs is identified by searching in the first level μR-tree (Algorithm 5). The number of reachable MCs for any MC would be less because, we have optimized the number of MCs generated in the previous step by making sure that the intersection between MCs is minimal. Note that the set of reachable MCs is stored along with the respective MC. For computing neighborhood query of a point $x \in Z$, we have to search only a subset of reachable MCs of Z. This subset contains only those reachable MCs that overlap with the ϵ-extended minimum bounding rectangle (MBR) of x. This further reduces the number of MCs searched. For example, in Fig. 3, the MBR of x is overlapping with only two of four reachable MCs of the MC containing x. The auxiliary R-trees of the above subset of reachable MCs are searched for computing exact neighborhood query of the point x. Since AuxR-trees are smaller in size, the query time is highly reduced, in contrast to searching in a single R-tree with entire dataset indexed.

3) Clustering and dynamic identification of wndq-core points: In this step, we process each $non\text{-}wndq\text{-}core$ point p (perform ϵ-nbh query of p) and do the following (Algo 6):

(i) If $|N_\epsilon(p)| < MinPts$: p is marked as noise, if no core point exists in $N_\epsilon(p)$, and is inserted into the $noiseList$. Otherwise p is assigned to the cluster of the core point q and union is performed on p and q.

(ii) If $|N_\epsilon(p)| \geq MinPts$: p is marked as core and a union is performed for every point $q \in N_\epsilon(p)$, with p. If q is a border point and is already assigned to another cluster then this union operation is not performed.

(iii) If $|N_{\epsilon/2}(p)| \geq MinPts$: We mark all non-core points in $N_{\epsilon/2}(p)$ as *wndq-core*. This step saves neighborhood queries while processing *non-wndq-core* points. A union is performed for every point $q \in N_\epsilon(p)$, with p. If q is outside the IC and is a border point already assigned to another cluster then this merging is not performed.

Algorithm 6: PROCESS-REM-POINTS

```
1  procedure PROCESS-REM-POINTS ()
       Input  : Data List X
       Output: Processes points in X that were not tagged as wndq-core
2      foreach Point p ∈ X that is not tagged as wndq-core do
3          FIND-NBHD (p, ϵ) // p.Nbhrs is updated
4          if |p.Nbhrs| < MinPts then
5              foreach Point x ∈ p.Nbhrs do
6                  if x.core == TRUE then UNION (x, p); break ;
7              if p is not yet assigned to any cluster then
                   noiseList.ADD (p) ;
8          else
9              p.Core = TRUE;
10             foreach x ∈ Nbhrs do
11                 if x.core == TRUE then UNION (x, p) ;
12                 else if x is not yet assigned to any cluster then
13                     UNION(x, p);
14 procedure FIND-NBHD ()
       Input  : Data Point p, ϵ
       Output: ϵ-neighborhood of p computed
15     foreach MC Z in p.MC.ReachList do
16         if Z.AuxR.root.MBR overlaps with reg_ϵ(p) then
17             NBHD-QUERY_ϵ (Z.AuxR.root, p);
18     if p.IC > MinPts then
19         foreach neighbor q in p.Nbhrs do
20             if DIST (q, p) < ϵ/2 then
21                 q.core = TRUE; wndqCorelist.ADD (q); UNION
                       (p, q);
```

Algorithm 7: POST-PROCESSING-CORE

```
1  procedure POST-PROCESSING-CORE ()
2      foreach Point p ∈ wndqCoreList do
3          foreach Reachable MC RμC of p.MC do
4              if reg_ϵ(p) overlaps with RμC then
5                  foreach core point q ∈ RμC do
6                      if p & q are not in the same cluster then
7                          if Dist (p, q) ≤ ϵ then UNION (q, p) ;
```

4) Establishing Final Connections: In the process of identifying core points without performing ϵ-neighborhood queries, merging of some of the core points (using union operation) might not have been performed. To address this issue, we process points in *wndqCoreList* (Algorithm 7). For each point p in *wndqCoreList*, the filtered set of reachable MCs (as explained in Section IV-B2) are considered to identify points for distance computations with p. Distance between p and a core point belonging any of the above reachable subset of MCs is calculated when the two core points are not in the same cluster. If distance is found to be less than ϵ then a union operation is performed. Although it is an additional cost, it is very less in comparison to an ϵ-neighbourhood query.

Also, during the above steps, some of the *border* points could have been marked as *noise*. This is because the marked *noise* point p can be in the ϵ-neighborhood of a *wndq-core* point q, which was declared as *wndq-core* after p was marked as *noise*. To rectify this, we process each *noise* point $p \in noiseList$ and if a core point q is present $N_\epsilon(p)$, we do a union of p & q and label it as *border* (Algorithm 8). This completes all the connections that are possible in a DBSCAN clustering. Note that this step doesn't require any additional neighborhood queries, as they were pre-computed and stored in the previous step.

C. Complexity Analysis

Let $n = |X|$, $m = |MCList|$, and r be the average number of points in any MC. Points $\in X$ are added to μR-tree in two steps. First, all the points are added to the first level μR-tree which would hold a total of m objects (MCs), where $m << n$ as $m = n/r$ (see Table II). Second, R-tree of average size r is constructed for each MC. The average time complexity of the overall algorithm is $O(n \log m + n \log r)$ which is much lesser than $O(n \log n)$ complexity of classical DBSCAN. In case of smaller value of ϵ, the number of MCs (m) is high, and r will be low. And, the converse also holds true. In worst case the above complexity can tend to $O(n \log n)$. This happens when the m is very large. The space complexity of the algorithm is $O(n \log n)$. The step-wise analysis is given in Table I.

Algorithm 8: POST-PROCESSING-NOISE

```
1  procedure POST-PROCESSING-NOISE ()
2      foreach Point p ∈ noiseList do
3          if p.nbhrs contains a core point q then
4              UNION (q, p); noiseList.REMOVE (p);
```

D. Correctness Analysis

Thoerem 1. *μDBSCAN satisfies all three conditions - **Connectivity**, **Maximality**, and **Noise**, of DBSCAN*

First, we prove the following lemmas and then we give proofs for each of the above condition.

Lemma 4. *μDBSCAN identifies all core points. And, all the core points are identified before step 4 of the algorithm.*

Proof. Lemma 1 proves that all *wndq-core* points are indeed core points. *wndq-core* points are identified in step 1 of μDBSCAN (Algo 4) Then ϵ-neighborhood queries are performed on all the points other than *wndq-core* points in step 3 (Algo 6), which identifies left out core points if any. Thus, μDBSCAN identifies all the core points, and it does so before step 4 of the algorithm. □

Lemma 5. *Let q and y_{k-1} be two points, then q ddr $y_{k-1} \implies q$ and y_{k-1} are in the same cluster.*

TABLE I: Time and Space Complexity of μDBSCAN

Time Complexity	
Construction of μR-tree	*max*$(O(n \log m), O(mr \log r))$ where $n \approx mr = O(n \log m + n \log r)$
Discovery of preliminary clusters	$O(mr) = O(n)$
Finding Reachable MCs of an MC	$O(m \log m)$
Find neighborhood query for *non-wndq-core* points	$O(n_1 l \log r)$, where n_1 is total number of *non-wndq-core* points s.t. $n_1 < n$, l = max. size of filtered reachable MCs
Total cost of the algorithm	$O(n \log m + n \log r)$
Space Complexity	
μR-tree	$O(n \log n)$
Noise-List	Number of noise points $\times$ $(MinPts - 1) \approx O(n_2 \cdot MinPts)$ where n_2 is the number of noise points s.t. $n_2 << n$
Wndq-core-list	$O(n)$
Total	$O(n \log n + n + n_2 \cdot MinPts) = O(n \log n)$

Proof. **Case 1**: y_{k-1} is a processed core point, i.e., ϵ-neighborhood query was performed on y_{k-1}, and if q ddr y_{k-1}, both q and y_{k-1} are merged with a union operation (case occurs in Algo 6).
Case 2: y_{k-1} is *wndq-core* point, then ϵ-neighborhood query on y_{k-1} is not performed (case occurs in Algo 4). Therefore, its neighborhood information is not available. There are two cases: 1) if q is *non-wndq-core*, its neighborhood query is performed (in Algo 6) and both q and y_{k-1} will be in the same cluster. 2) if q is $wndq\text{-}core$ then its ϵ-neighborhood query is not performed. If $y_{k-1} \in IC_q$ or vice versa is true with either y_{k-1} or q as one of the micro-cluster centers, then q and y_{k-1} will be merged into one cluster (in Algo 4). Otherwise, we insert all $wndq\text{-}core$ points in a *wndq-core-list* which is processed in Algo 7 and both q and y_{k-1} will be merged into one cluster. □

Proof. (of Theorem 1) **Noise**: The algorithm marks a point p as noise only after performing its neighborhood query and satisfies two conditions: $|N_\epsilon(p)| < MinPts$ and $N_\epsilon(p)$ does not have any core point in it (lines 4-7 of Algo 6). This implies that p is not ddr to any core point $q \in X$. However, it is possible that p is processed before q is marked as *wnqd-core* (otherwise ϵ-neighborhood query would have been performed on p). To address this case, we put all the noise points in the $noiseList$ along with the neighborhood information and this list is processed after all the core points are identified in Algo 8. This makes sure that all the noise points are indeed marked as noise. □

Proof. (of Theorem 1) **Maximality**: Given a point $p \in$ cluster C, and point q dc p. $p \in C \implies p$ is a core point or a border point and q dc $p \implies$ there is a common core point x, s.t., p dr x and q dr x. That is, there is a series of points $x = x_0$, x_1, ..., $x_k = p$, such that x_{i+1} ddr x_i and similarly $x = y_0$, y_1, ..., $y_k = q$ such that y_{i+1} ddr y_i. We have already shown in Lemma 5 that if q ddr y_{k-1} then $q(= y_k)$ and y_{k-1} will be in the same cluster. By induction we can show that $\forall i$, y_i and y_{i-1} are in the same cluster, and therefore q and x are in the same cluster. Similarly we can show that x and p are in the same cluster, and thus p and q are in the same cluster. □

Proof. (of Theorem 1) **Connectivity**: Two points p, q can belong to a cluster only through line 6, 8, 9 & 10 of Algo 2. In each case we argue that p dc q.
Case1: (line 6, i.e., Algo 4) p, $q \in C \implies p$, $q \in N_\epsilon(x)$, where x is *wndq-core*. This implies that p ddr x and q ddr x. This implies that p dc q (by definitions in Section II).
Case2: (line 8, i.e., Algo 6) In this case, two points p and q are merged only when one of them is a core point (lines 6 & 11), i.e., p dr q or q dr p. This implies that p dc q. A point p is also getting merged in line 21 with another point q, where q is *wndq-core*, which is covered in case1.
Case3: (line 9, i.e., Algo 7) In this case a *wndq-core* point p gets merged with another core point q, if DIST $(p, q) < \epsilon$. So, p ddr q and this implies p dc q.
Case4: (line 10, i.e., Algo 8) If $p \in$ *noiseList* and q is a core point $\in N_\epsilon(p)$, p becomes a border point and is merged into cluster of q. So, p ddr q and this implies p dc q. □

Algorithm 9: μDBSCAN-D

```
1 procedure μDBSCAN-D ()
      Input : DataList X, Number of processes p, ϵ, MinPts
      Output: A set of clusters in distributed Union-Find data structure
2     SAMPLING-BASED-KD-PARTITIONING ();
3     foreach Process P_i (1 ≤ i ≤ p) in parallel do
4         FETCH-EPS-EXTENDED-RGN (ϵ);
5         LOCAL-μDBSCAN (X_i,ϵ,MinPts); //X_i is local data at P_i
6     PARALLEL-MERGING ();
```

V. μDBSCAN-D

In this section, we present, μDBSCAN-D, an efficient parallel implementation of μDBSCAN for distributed memory systems, implemented using MPI (see Algorithm 9). μDBSCAN-D has three major phases: 1) Spatial Data Partitioning, 2) Local μDBSCAN, and 3) Merging of Local Clusterings. They are explained below:

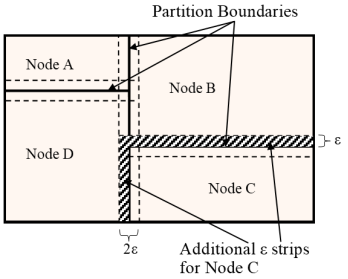


Fig. 4: Spatial Partitioning through kd-tree

A. Spatial Data Partitioning

The neighborhood queries and merging of data points in DBSCAN are dependent on the spatial position of the points. Thus, it is essential that the spatial locality of the data is maintained in the data partitioning phase to minimize the communication costs during execution of remaining phases. Another consideration for data partitioning is that the data should be divided equally among the computing nodes to achieve load balancing. The kd-tree data partitioning approach (previously used in clustering [4], [5], [8], [9] recursively divides data amongst the computing nodes based on axis aligned split (see Fig. 4). The axis is chosen with the largest spread and split is performed on the basis of median (for perfect load balancing). However, for very large data, computing median is expensive. In our work, a sampling based median approach [9] is applied. In this, a sample of points from each node is chosen and aggregated. The median of the sample is calculated and the data is divided at each node. The first $p/2$ nodes contain data points less than the median on the chosen axis and the last $p/2$ nodes contain data greater than median. This process is repeated for each set of $p/2$ nodes recursively to get partitions of $p/4$ data and so on. In total, this process is executed $\log(p)$ times, where p is the number of processors or computing nodes.

B. Local μDBSCAN

After the data partitioning phase, each computing node executes μDBSCAN on its local data. Before it starts clustering, every computing node is given data points belonging to other nodes that lie in an ϵ-extended strip (also known as *halo-region*) of the partition boundary. Fig. 4 shows ϵ-extended strip

for partition boundary of node C. The points in this halo-region are required to compute the correct ϵ-neighborhood queries for the points lying in the ϵ-boundaries of the current node, without any extra communication overhead. So, after receiving halo regions, each computing node constructs micro-clusters on the local data and indexes them in a local μR-tree. After that micro-clusters are processed as described in Section IV to get local clustering. Note that we use distributed Union-Find data structure [19] for performing Union and Find operations required for merging.

C. Merging of Local Clustering Results

In this phase, local clusterings obtained at each computing node are merged to get global clustering, i.e., to identify clusters that span across multiple partitions. The merging strategy is very efficient because it doesn't require even a single neighborhood query. In the course of local clustering phase, each computing node has accumulated a list of point pairs (x, y), where $x \in$ local node, $y \in$ any remote node and $N_\epsilon(x)$ contains y. Essentially y is a remote point (point in halo region). The cluster to which each x belongs to, can get merged with the cluster to which y belongs to (this cluster resides in a remote node) and form a cluster spanning multiple nodes. For this, each computing node communicates all the above point pairs to all other nodes to perform UNION operation. On receiving such a query, each node will do a UNION operation depending upon the case whether point y is a 1) core point; or 2) not a core point but assigned to some other cluster; or 3) not yet assigned to any cluster. We follow a similar technique as presented in [5]. For further details on this merging step, please refer to the above paper.

Parallel μDBSCAN satisfies the three conditions of *maximality*, *connectivity*, and *noise* of the classical DBSCAN. We are omitting the proofs because of space constraint.

D. Space and Time Complexity

In our analysis, we are not factoring in the data distribution time as it is done offline. It is a common practice in literature. The time complexity of parallel μDBSCAN (worst case) = local μDBSCAN time + merging time = $O((n/p)\log(m/p) + (n/p)\log(r/p)) + O(\delta \cdot n/p)$ where, p is the number of processing elements and δ is the fraction of n/p points lying in ϵ boundaries. The space occupied by μ-Rtree, noise list, wndqcore list and reachable micro-cluster lists will be reduced by a factor of p in μ-DBSCAN-D. Thus the total space complexity is $O(n/p \log n/p)$.

VI. EXPERIMENTAL RESULTS AND ANALYSIS

All experiments were conducted on a cluster of 32 computing nodes, where each node is an IBM x3250 m4 Server with Intel Xeon CPU E3-1230 v2 @ 3.30GHz (64-bit) and 32 GB (DDR3 - 1600 MHz) RAM. The experiments on sequential μDBSCAN are conducted over a single node of the cluster. For executing parallel algorithms, we use only one CPU core from each computing node unless explicitly stated. All algorithms were implemented in C/C++ with MPI[1]

[1]Complete source code of μDBSCAN and μDBSCAN-D is available at https://github.com/ADAPTLab

We have bench-marked our proposed sequential and parallel algorithms with their respective existing solutions for various real datasets commonly used in literature for evaluating density-based clustering algorithms [4], [5], [8], [24], [27]. The details of the datasets used (including parameters: ϵ and $MinPts$) are given in Tables II, V & VI. MPAGalaxiesDelucia2006a (MPAGD*) [28], DGalaxiesBower2006a (DGB*) [29], MPAGalaxiesBertone2007a (MPAGB*) [30] and friends-of-friends (FOF*) [31] datasets are obtained from the Galaxy and Halo databases on the Millennium Run [31]. In addition, we have also used a few other real datasets: 3D Road Network (3DSRN) [32] contains vechicular GPS data; Household Power (HHP*) and KDDBIO145K (KDDB*) datasets are borrowed from UCI Repository [33]. KDDB* datasets have been sampled for various dimensions (14, 24 and 74 (full)).

For comparative experimental study, we use the implementations of both sequential and parallel DBSCAN algorithms that were made publicly available by their respective authors. G-DBSCAN is the only algorithm implemented by us.

A. Performance Evaluation of μDBSCAN

First, we compare the run-time performance of sequential μDBSCAN with that of classical DBSCAN that uses R-tree as indexing structure (*R-DBSCAN*), *G-DBSCAN* [3], as well as the latest proposed sequential *GridDBSCAN* [4]. The results presented in Table II show that μDBSCAN has performed consistently better than the remaining three algorithms for all the datasets. Note that GridDBSCAN has thrown memory error for two datasets listed in the table.

As explained in Section IV-B, μDBSCAN labels core points in the dataset without performing neighborhood queries and thus saves a lot of neighborhood queries. To substantiate this, we as well report percentage of queries saved by μDBSCAN for each dataset in the above table. It can be observed that 43%- 96% of queries are saved for various datasets. We also report the number of MCs formed for each dataset. We can clearly observe that the number of MCs is significantly less than the data size, which leads to reduction in computational cost as explained in Section IV-B. The complexity of μDBSCAN is dominated by the first term - $n \log m$, where m is significantly small.

We also report the split-up of execution times of various steps of μ-DBSCAN in Table III. The results show that the tree construction takes significant portion of the execution time. This is because we form MCs while constructing the tree. The proportion of post processing is high in case when the number of queries saved is high as indicated by 3DSRN and KDDB145K14D datasets. This is because of increase in number of *wndq-core* points. Similar observations have been made for other datasets as well.

We also report peak memory consumption of μDBSCAN and other sequential algorithms for various datasets (see Table IV). The results show that the peak memory consumption for μDBSCAN is much lesser than GridDBSCAN, especially for high dimensional datasets. The peak memory consumption of R-DBSCAN and G-DBSCAN is lesser than μDBSCAN. This is because R-DBSCAN uses a simple R-tree that occupies

TABLE II: Run time comparision (in sec.) of μDBSCAN with other sequential algorithms

Dataset	n	d	ϵ	$MinPts$	R-DBSCAN (run time)	G-DBSCAN (run time)	GridDBSCAN (run time)	μDBSCAN (run time)	No. of MCs (m)	% query saves in μDBSCAN
3DSRN	0.43M	3	0.01	5	49.51	245.45	41.97	22.87	22353	80.99%
DGB0.5M3D	0.5M	3	1	5	37.06	3103.57	53.87	23.39	99031	43.60%
HHP0.5M5D	0.5M	5	0.6	6	5040.36	1079.37	1406.51	795.03	8625	93.49%
MPAGB6M3D	6M	3	1	5	15922.28	$>$ 12 hrs	2704.71	572.28	734881	69.47%
FOF56M3D	56M	3	3	6	59154.04	$>$ 12 hrs	17036.34	6960.05	782969	95.68%
MPAGD100M3D	100M	3	1	5	18574.45	$>$ 12 hrs	$>$ 5 hrs (Mem Err)	11329.92	3268853	86.92%
KDDB145K14D	145K	14	200	5	3604.48	584.23	5192.62	360.9	906	96.34%
KDDB145K24D	143K	24	600	5	8270.85	2612.07	$>$ 2 hrs (Mem Err)	2578.58	655	96.60%

TABLE III: %age Split-up of execution time of various steps of μDBSCAN

Dataset	Tree Construction	Finding Reachable Groups	Clustering	Post Core & Noise Processing
3DSRN	31.49%	0.08%	10.06%	63.09%
DGB0.5M3D	20.46%	27.73%	15.27%	36.53%
MPAGB6M3D	15.11%	13.92%	13.55%	57.42%
KDDB145K14D	0.75%	0.01%	2.56%	96.68%

TABLE IV: Peak memory consumption of μDBSCAN and others algorithms

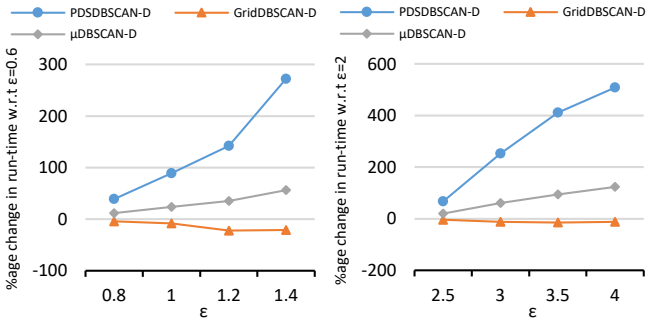
Dataset	R-DBSCAN	G-DBSCAN	Grid DBSCAN	μDBSCAN
3DSRN	125 MB	50 MB	458 MB	158 MB
DGB0.5M3D	143 MB	74 MB	617 MB	261 MB
MPAGB6M3D	2178 MB	–	9844 MB	2530 MB
KDDB145K14D	61 MB	32 MB	20.17 GB	67 MB

lesser memory than μR-tree, and G-DBSCAN doesn't use any indexing structure. However the run-time performance of μDBSCAN has been far superior than both of these and a little higher memory requirement is worth it and justified. Note that peak memory of G-DBSCAN for MPAGB6M3D is left blank in the table as we killed its execution after running for a long time (indicated in Table II as well). Also, since GridDBSCAN takes a large amount of memory, especially for high dimensional datasets, it justifies the memory error encountered for two datasets indicated in Table II. However μDBSCAN was able to run without any such error for the same available memory (32 GB). One can also see from Table V that parallel μDBSCAN was able to process datasets of billions scale on a cluster of 32 nodes with 32 GB RAM in each. However, other algorithms couldn't execute due to memory error.

B. Performance Evaluation of μDBSCAN-D

We have used kd-tree for geometrically partitioning the data among the processing elements [9], which read the data using parallel I/O. However, we do not include data partitioning and file read/write operations while computing the speedup and comparing the parallel algorithms.

We compare μDBSCAN-D with PDSDBSCAN-D [5], GridDBSCAN-D [4], HPDBSCAN [10] and RP-DBSCAN [24]. The run-time performance of each of the above algorithms executed using 32 nodes of the cluster, is presented in Table V. The results clearly show that μDBDSCAN-D gives lowest run time, with exception of HPDBSCAN. However, the source code provided by authors of HPDBSCAN doesn't produce clusters as that of classical DBSCAN. For example, for FOF56M3D, number of clusters differ by approximately 27%. Also, we have observed that the number of clusters produced by HPDBSCAN is not consistent with change in number of processors. Also, HPDBSCAN has run only for



(a) (b)

Fig. 5: Effect of varying ϵ on PDSDBSCAN-D, GridDBSCAN-D and μ-DBSCAN-D for (a) MPAGD100M3D and (b) FOF56M3D datasets

limited number of datasets (as shown in Table V) and has thrown run-time error for the others. The execution time of RP-DBSCAN is much higher ($>$ 5 times in most cases) than μDBSCAN-D. However, RP-DBSCAN is a spark implementation and the comparison cannot be considered fair. Moreover RP-DBSCAN is a ρ-approximate algorithm (we used ρ = 0.99 as suggested by the authors). Hence, we don't consider both HPDBSCAN and RP-DBSCAN for further experimentation. Cells marked with '-' in the table indicate that execution for that dataset couldn't run. This is because, those algorithms are not capable of handling a large number of floating points on the available resources. However, μDBSCAN-D was able to execute without any such errors.

It can be observed from Table V that μDBSCAN-D is able to cluster 1 billion data points (3 billion floating points) in 41 minutes over 32 processors, which is remarkable. The other algorithms were unable to process such large data with the same available resources.

We present, the effect of variation in ϵ on run-time performance of μDBSCAN-D, PDSDBSCAN-D and GridDBSCAN-D. The results presented in Fig. 5 shows that %age increase in run-time with increase in ϵ, is much smaller for μDBSCAN-D than PDSDBSCAN-D. This is because, with increase in ϵ the time for micro-clusters formation and reachable group identification decreases, and time for post processing of *wndq-core* points increases, with the later dominating. Increase in post processing is because of increase in number of *wndq-core* points. The time for GridDBSCAN-D decreases with increase in ϵ. However, the overall execution time of μDBSCAN-D has been consistently lesser at all ϵ values for all datasets.

We present the effect of increase in dimensionality on run-time performance of μDBSCAN-D for various dimensionality samples of KDDB143K74D datset. The results presented in Fig. 6 show that the run-time increases from 8.15 sec. to 460.83 sec. when dimensions are increased from 14 to 74.

TABLE V: Run-time performance (in seconds) of μDBSCAN-D when compared with the baseline approaches on 32 computing nodes

Dataset	n	d	ϵ	$MinPts$	PDSDBSCAN-D	GridDBSCAN-D	HPDBSCAN	RPDBSCAN	μDBSCAN-D
MPAGD8M3D	8M	8	1	5	37.7	169.379	10.85	1832.99	23.97
MPAGD100M3D	100M	3	1	5	468.72	1369.41	140.85	58883.56	345.95
FOF56M3D	56M	3	3	6	185.78	423.24	10	2030.35	123.31
FOF28M14D	28M	14	7	5	-	-	-	6516.56	1631.58
KDDB145K14D	145K	14	200	5	126.82	483.87	-	115.8	8.15
KDDB145K74D	145K	74	1500	5	-	-	-	-	460
MPAGD1B3D	1B	3	0.4	5	-	-	-	-	2474.23
FOF500M3D	500M	3	3.5	5	-	-	-	-	4229.81

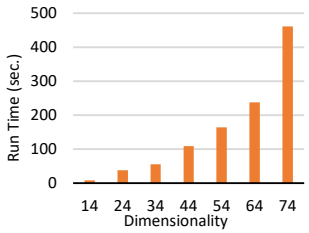


Fig. 6: Performance of μDBSCAN-D with variation in dimensionality of KDDBIO143K74D dataset

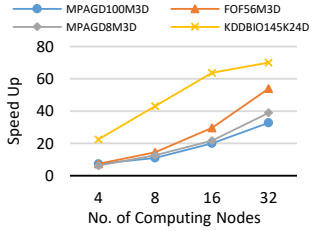


Fig. 7: Scalability of μDBSCAN-D with increase in number of computing nodes for various datasets

This is expected because of increase in computational cost of the queries. For, this experiment, we kept the number of clusters almost same for each dataset sample. We don't present for PDSDBSCAN-D and GridDBSCAN-D as they were not capable of handling high dimensional data/ large number of floating points, as indicated by Table V.

We analyze the scalability μDBSCAN-D for different real datasets having different dimensions and sizes by varying the number of computing nodes from 4 to 32. The results presented in Fig. 7 show the speedup with respect to sequential algorithm. The maximum speedup achieved is 70 for 32 nodes. The super-linear speed-up is because of reduction in time for construction and querying of R-trees, as smaller R-trees (in case of higher number of processors) exhibit better performance than a single larger R-tree [4], [27]. We also present the execution time for very large datasets while increasing the number of processing cores from 32 to 128, in Table VI. In this experiment, we use multiple cores of each computing node of the same 32-node cluster. Each core executed a separate MPI Process. These results are consistent with the above results and substantiate the scalability of μDBSCAN-D.

We also report the execution time split-up of various steps of μDBSCAN-D in Table VII for various datasets. The first four steps are part of local clustering, and the merging time indicates the time for merging the local clusterings. It can be observed from the table that merging time is low for all datasets, thus making our approach scalable. This shows that the overhead of parallelization is minimal in μDBSCAN-D.

Next, we report the execution time and speed-up attained for individual steps of μDBSCAN-D (executed over 32 nodes) when compared to sequential μDBSCAN for MPAGD8M3D dataset, in Table VIII. The results clearly show that speedup is attained for individual steps as well. This shows that our parallelization strategy is efficient.

TABLE VI: Run-time of μDBSCAN-D with increasing number of processing cores for large datasets

Dataset	ϵ	$MinPts$	32	32*2=64	32*4=128
FOF500M3D	3.5	5	4229.81	2641.03	1800.62
MPAGD800M3D	0.5	5	1881.2	977.85	624.44

TABLE VII: Percentage split-up of execution time of various steps of μDBSCAN-D, when executed over 32 nodes

Dataset	FOF28M14D	MPAGD100M3D	FOF56M3D
Tree Construction	4.19%	8.09%	26.39%
Finding Reach. Groups	1.04%	3.95%	1.6%
Clustering	80.94%	25.32%	10.74%
Post Processing	8.52%	40.99%	39.4%
Merging Time	3.88%	1.83%	2.27%

TABLE VIII: Execution time and Speed-up attained for various steps of μDBSCAN-D (executed on 32 nodes) when compared to μDSBCAN for MPAGD8M3D dataset

	MuDSBSCAN	MuDBSCAN-D	Speed-Up
Tree Construction	157.46 sec.	1.89 sec.	83.12
Finding Reachable Groups	170.76 sec.	0.96 sec.	176.45
Clustering	124.21 sec.	4.72 sec.	26.31
Post Processing	388.74 sec.	11.12 sec.	34.95
Merging Time	—	2.34 sec.	—
Total Time	841.21 sec.	23.97 sec.	35.08

VII. Conclusion & Future Work

We proposed an efficient sequential micro-cluster based DBSCAN (μDBSCAN) algorithm which produces exact DBSCAN clustering and is also amenable to parallelization. The proposed algorithm exploits spatial locality and labels points as core points without performing neighbourhood queries. We have also presented a parallel version of μDBSCAN, μDBSCAN-D, which exploits distributed memory architecture. Our theoretical analysis for time and space complexity, correctness and efficiency proves the soundness of the proposed algorithms. The proposed algorithms also exhibit lesser average case time complexity than that of the classical DBSCAN.

The experimental results presented show that μDBSCAN and μDBDSCAN-D outperform their respective baseline algorithms. The parallel version is also able to cluster 1 billion data points in only 41 minutes over 32 processors. Whereas the baselines approaches don't have the capability of processing such large data with the same quantum of available computing resources. The proposed algorithms are able to save neighborhood queries upto 96% in the datasets used for experimentation. They are also scalable and less susceptible to change in ϵ and dimensionality of the dataset. Overall, our approach is robust, efficient, scalable, and produces exact DBSCAN clustering.

In future, we intend to extend this approach to leverage multiple cores available in each computing node to make the algorithm work even faster. Similar designs can be made for the GPGPU architectures as well. This approach can also be adopted to fast clustering of data streams.

REFERENCES

[1] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "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*. AAAI Press, 1996, pp. 226–231.

[2] C. C. Aggarwal and C. K. Reddy, *Data Clustering: Algorithms and Applications*, 1st ed. Chapman and Hall/CRC, 2013.

[3] K. Mahesh Kumar and A. Rama Mohan Reddy, "A fast DBSCAN clustering algorithm by accelerating neighbor searching using Groups method," *Pattern Recognition*, vol. 58, no. C, pp. 39–48, 10 2016.

[4] S. Kumari, P. Goyal, A. Sood, D. Kumar, S. Balasubramaniam, and N. Goyal, "Exact, Fast and Scalable Parallel DBSCAN for Commodity Platforms," in *Proceedings of the 18th International Conference on Distributed Computing and Networking - ICDCN '17*. ACM New York, 2017, pp. 1–10.

[5] M. M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary, "A new scalable parallel DBSCAN algorithm using the disjoint-set data structure," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 11 2012, pp. 1–11.

[6] C.-F. Tsai and T.-W. Huang, "QIDBSCAN: A Quick Density-Based Clustering Technique," in *2012 International Symposium on Computer, Consumer and Control*. IEEE, 6 2012, pp. 638–641.

[7] C.-F. Tsai and Chun-Yi Sung, "DBSCALE: An efficient density-based clustering algorithm for data mining in large databases," in *2010 Second Pacific-Asia Conference on Circuits, Communications and System*. IEEE, 8 2010, pp. 98–101.

[8] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey, "Pardicle: Parallel Approximate Density-Based Clustering," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 11 2014, pp. 560–571.

[9] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey, "BD-CATS: big data clustering at trillion particle scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*. New York, New York, USA: ACM Press, 2015, pp. 1–12.

[10] M. Götz, C. Bodenstein, and M. Riedel, "HPDBSCAN: Highly Parallel DBSCAN," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments - MLHPC '15*. ACM New York, USA, 2015, pp. 1–10.

[11] C. Xiaoyun, M. Yufang, Z. Yan, and W. Ping, "GMDBSCAN: Multi-Density DBSCAN Cluster Based on Grid," in *2008 IEEE International Conference on e-Business Engineering*. IEEE, 2008, pp. 780–783.

[12] C.-F. Tsai and C.-T. Wu, "GF-DBSCAN: a new efficient and effective data clustering technique for large databases," in *Proceedings of the 9th WSEAS international conference on Multimedia systems & signal processing*. WSEAS, 2009, pp. 231–236.

[13] C.-F. Tsai and C.-W. Liu, "KIDBSCAN: A New Efficient Data Clustering Algorithm," in *International Conference on Artificial Intelligence and Soft Computing (ICAISC 2006)*. Springer, Berlin, Heidelberg, 2006, pp. 702–711.

[14] J. Li, W. Yu, and B.-P. Yan, "Memory effect in DBSCAN algorithm," in *2009 4th International Conference on Computer Science & Education*. IEEE, 7 2009, pp. 31–36.

[15] J. H. Peter and A. Antonysamy, "An Optimised Density Based Clustering Algorithm," *International Journal of Computer Applications*, vol. 6, no. 9, pp. 20–25, 9 2010.

[16] S. T. Mai, I. Assent, and M. Storgaard, "AnyDBC: An Efficient Anytime Density-based Clustering Algorithm for Very Large Complex Datasets," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*. ACM New York, USA, 2016, pp. 1025–1034.

[17] S. Brecheisen, H.-P. Kriegel, and M. Pfeifle, "Parallel Density-Based Clustering of Complex Objects," in *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, Berlin, Heidelberg, 2006, pp. 179–188.

[18] M. Coppola and M. Vanneschi, "High-performance data mining with skeleton-based structured parallel programming," *Parallel Computing*, vol. 28, no. 5, pp. 793–813, 5 2002.

[19] M. M. A. Patwary, J. Blair, and F. Manne, "Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure," in *Proceedings of the 9th International Conference on Experimental Algorithms*. Springer, Berlin, Heidelberg, 2010, pp. 411–423.

[20] Y. He, H. Tan, W. Luo, S. Feng, and J. Fan, "MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data," *Frontiers of Computer Science*, vol. 8, no. 1, pp. 83–99, 2 2014.

[21] C.-C. Chen and M.-S. Chen, "HiClus: Highly Scalable Density-based Clustering with Heterogeneous Cloud," *Procedia Computer Science*, vol. 53, pp. 149–157, 1 2015.

[22] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan, "MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 12 2011, pp. 473–480.

[23] B. Welton, E. Samanas, and B. P. Miller, "Mr. Scan: Extreme scale density-based clustering using a tree-based network of GPGPU nodes," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*. New York, New York, USA: ACM Press, 2013, pp. 1–11.

[24] H. Song and J.-G. Lee, "RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning," in *Proceedings of the 2018 International Conference on Management of Data - SIGMOD '18*. ACM New York, USA, 2018, pp. 1173–1187.

[25] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A Framework for Clustering Evolving Data Streams," in *Proceedings of 29th International Conference on Very Large Databases*. ACM New York, 2003, pp. 81–92.

[26] A. Guttman, "R-trees," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84*. ACM New York, 1984, pp. 47–57.

[27] P. Goyal, S. Kumari, D. Kumar, S. Balasubramaniam, N. Goyal, S. Islam, and J. S. Challa, "Parallelizing OPTICS for Commodity Clusters," in *Proceedings of the 2015 International Conference on Distributed Computing and Networking - ICDCN '15*. New York, New York, USA: ACM Press, 2015, pp. 1–10.

[28] G. De Lucia and J. Blaizot, "The hierarchical formation of the brightest cluster galaxies," *Monthly Notices of the Royal Astronomical Society, Volume 375, Issue 1, pp. 2-14.*, vol. 375, pp. 2–14, 2007.

[29] R. G. Bower, A. J. Benson, R. Malbon, J. C. Helly, C. S. Frenk, C. M. Baugh, S. Cole, and C. G. Lacey, "Breaking the hierarchy of galaxy formation," *Monthly Notices of the Royal Astronomical Society*, vol. 370, no. 2, pp. 645–655, 8 2006.

[30] S. Bertone, G. De Lucia, and P. A. Thomas, "The recycling of gas and metals in galaxy formation: predictions of a dynamical feedback model," *Monthly Notices of the Royal Astronomical Society*, vol. 379, no. 3, pp. 1143–1154, 8 2007.

[31] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce, "Simulations of the formation, evolution and clustering of galaxies and quasars," *Nature*, vol. 435, no. 7042, pp. 629–636, 6 2005.

[32] M. Kaul, B. Yang, and C. S. Jensen, "Building Accurate 3D Spatial Networks to Enable Next Generation Intelligent Transportation Systems," in *2013 IEEE 14th International Conference on Mobile Data Management*. IEEE, 6 2013, pp. 137–146.

[33] "KDD Cup 2004 Bio Dataset," 2004. [Online]. Available: http://cs.joensuu.fi/sipu/datasets/