https://toc.ui.ac.ir

Transactions on Combinatorics
ISSN (print): 2251-8657, ISSN (on-line): 2251-8665
Vol. 14 No. 3 (2025), pp. 135-156.
© 2025 University of Isfahan

www.ui.ac.ir

DENSITY-BASED CLUSTERING IN MAPREDUCE WITH GUARANTEES ON PARALLEL TIME, SPACE, AND SOLUTION QUALITY

SEPIDEH AGHAMOLAEI* AND MOHAMMAD GHODSI

Abstract. A well-known clustering problem called Density-Based Spatial Clustering of Applications with Noise (DBSCAN) involves computing the solutions of at least one disk range query per input point, computing the connected components of a graph, and bichromatic fixed-radius nearest neighbor. MapReduce class is a model where a sublinear number of machines, each with sublinear memory, run for a polylogarithmic number of parallel rounds.

Most of these problems either require quadratic time in the sequential model or are hard to compute in a constant number of rounds in MapReduce. In the Euclidean plane, DBSCAN algorithms with near-linear time and a randomized parallel algorithm with a polylogarithmic number of rounds exist.

We solve DBSCAN in the Euclidean plane in a constant number of rounds in MapReduce, assuming the minimum number of points in range queries is constant and each connected component fits inside the memory of a single machine and has a constant diameter.

1. Introduction

Most clustering algorithms, including k-center and k-means, create convex clusters. A widely used clustering algorithm for finding non-convex clusters is Density-based spatial clustering of applications with noise (DBSCAN). Improving the running time of DBSCAN in the Euclidean plane has attracted some research in computational geometry [1] that use tools such as Delaunay triangulation, grids, and approximate range counting. A randomized parallel algorithm for Euclidean DBSCAN in the plane

MSC(2010): Primary: 68Q25; Secondary: 68W10, 68W25.

Keywords: Massively parallel algorithms, range searching, unit disk graph, near neighbors, clustering.

Article Type: Research Paper.

Communicated by Alireza Abdollahi.

*Corresponding author.

Received: 12 July 2023, Accepted: 06 March 2024, Published Online: 23 April 2024.

Cite this article: S. Aghamolaei and M. Ghodsi, Density-Based clustering in mapreduce with guarantees on parallel time, space, and solution quality, *Trans. Comb.*, **14** no. 3 (2025) 135-156.

http://dx.doi.org/10.22108/toc.2024.138377.2091 .

exists that with high probability has linear work (total time of parallel machines) and logarithmic time for constant radius (r) and the minimum number of points (f) [2]. Existing algorithms rely on computing connectivity or require superlinear running time, which makes them unfit for big data settings such as streaming and MapReduce.

Another clustering method with non-convex clusters is correlation clustering. Consider a weighted graph $G = (V, E, w)$ whose edges are marked positive (E^+) or negative (E^-) and have weights $w(e)$, for $e \in E$. Correlation clustering asks for a partitioning of the vertices into k subsets (by finding a k-cut) that minimizes the cost of this clustering which is the total of the sum weight of the positive edges between these subsets and the sum of the weight of the negative edges inside these subsets:

$$\min_{k\text{-cuts } \delta} \sum_{e \in E^+ \cap \delta} w(e) + \sum_{e \in E^- \setminus \delta} w(e).$$

Correlation clustering is NP-hard and it admits a PTAS [3] as well as constant-factor approximation algorithms based on different assumptions [4, 5].

1.1. Problem Definition.

Euclidean DBSCAN can be broken into $3+1$ subproblems (3 of them are needed to solve DBSCAN, where the last one is used to improve the results):

- **Threshold Range Counting Query (TRC):** Given a set of points P in the Euclidean plane, a radius r, and an integer f, find the subset of P where for each point p, there are at least f other points of P within distance r of p.
- **Connected Components of Unit Disk Graph (UDG) in The Euclidean Plane:** Given a set of points P in the Euclidean plane and a radius r, find the connected components of its disk graph. For the definition of disk graph, see section 2.1.1.
- **Bichromatic Nearest Neighbor (NN) Range Query with Frequency:** Given two sets of points P (red points) with at least f near neighbors and Q (blue points) with at most f near neighbors in the Euclidean plane and a radius r, for each point of Q, find a point in P within distance r.
- **Biconnected Components:** Inspired by the use of cut size in correlation clustering, we introduce the problem of *finding biconnected components among DBSCAN clusters* in addition to clusters and cluster cores.

The value of the radius in different subproblems must be set based on the parameters of DBSCAN. For example, the radius is set to half the distance between directly reachable points in DBSCAN (eps) when computing connectivity but it must be set to the distance itself when counting the number of points in disk range queries.

1.2. Literature Review.

Fixed-radius nearest neighbor is finding points within distance r of a given point from a point set P. DBSCAN algorithm [6] has time complexity $O(n^2)$ in the worst case, which can be improved to $O(n \log n)$ in some cases in the Euclidean plane using data structures for fixed-radius nearest neighbor with $O(\log n)$ expected time on uniformly distributed point sets [7, 8]. One of these cases is if the radius r is so small that $O(\log n)$ points fall inside it (and therefore the number of nodes in the part of the tree that is searched is small). Another one is when the density of the clusters is almost the same everywhere (no local outliers exist-Chapter 12 of [9]) and having dense regions would not violate the uniform distribution, resulting in expected $O(f \log n)$ time. Such a time complexity is the same as $O(\log n)$ if $f = O(1)$. Parallel algorithms for fixed-radius nearest neighbor using grids exist [10].

A $O(n \log n)$-time algorithm for DBSCAN using grids and Delaunay triangulation exists [11]. The algorithm uses sequential subroutines for computing the connected components and fixed-radius near neighbors, so it does not generalize to parallel models such as MapReduce. There is an algorithm with running time $O(n \log \log n)$ in the Euclidean plane and there is a lower bound $\Omega(n^{4/3})$ on the time complexity of the exact algorithms for this problem in three dimensions [1]. A relaxed version of DBSCAN called α-approximate DBSCAN introduced by [1] allows the radius of the disk graph to be multiplied by a factor α, for which they give an algorithm with $O(n)$ expected time. The graph and the value r used in the algorithm also affect the running time of the algorithm [12].

Algorithms with a better time complexity than linear in the maximum of the input size and the output size are usually achieved by parallelism, distributing data, or using randomization. MapReduce class (MRC) is a theoretical model for several parallel and distributed frameworks such as MapReduce and Spark. A more detailed explanation is given in Section 2.2. The complexity of some data structures for range searching in MapReduce has been analyzed for a single query [13], however, when a linear number of such queries are run in parallel, the total time complexity exceeds the available memory. In this case, the amortized complexity for a set of queries becomes more important. Some of the algorithmic tools in MapReduce such as grids, spanners [14] and range searching data structures using arrangements and point location [15] and segment tree [16] can be used to solve different parts of DBSCAN. Plane-sweeping has been used to solve several range query problems [15, 16]. For point location using trapezoidal decomposition and for line sweeping using sweep tree, there are $O(\log n)$ time randomized CREW PRAM algorithms for n processors [17]. Using the simulation from PRAM to MapReduce [18, 19] gives $O(1)$-round algorithms. However, in most cases, deterministic algorithms are achievable using the dependencies between the queries used to solve a specific problem, such as simultaneous near-neighbors on a grid used in [14].

1.2.1. Threshold Range Queries.

Given a set of points P and a shape (range) Q, find the subset of P that lies in Q (reporting queries). Aggregated forms of these queries also exist, where instead of finding the subset of the points of P in Q, the problem asks for some function of them such as counting, summation, and finding the minimum and maximum.

The problem mostly appears in frequent itemset mining and it has been referred to as iceberg queries [20]. It appears in database applications for aggregation queries with a threshold and in network flow analysis for finding network flows with at least a certain amount of traffic [20].

Windowing range queries in external memory ([21, Section 6 of Chapter 9]) discuss the computational complexity of answering such queries.

1.2.2. **Related Geometric Graph Problems in MapReduce.** A complete graph whose vertices are a set of points in a d-dimensional Euclidean space ($\mathbb{R}^d$) and the weight of the edge connecting two vertices is the Euclidean distance between the points corresponding to these vertices, the graph is called a geometric graph.

For a given graph, the subgraph that connects all the vertices and minimizes the sum of the edge weights is called the minimum spanning tree. The minimum spanning tree of a geometric graph is called an Euclidean minimum spanning tree (EMST).

In MapReduce, testing whether two vertices of a graph are connected is conjectured to require at least a logarithmic number of rounds (one-cycle vs. two-cycles conjecture) [22] and other equivalent conjectures [23]. An approximation algorithm for EMST in MRC using $O(1)$ rounds [24] and an exact algorithm for connectivity using the Hamming distance (ℓ_0) in logarithmic time exist [22].

Single-linkage clustering asks for the disk graph of the smallest radius with at most k connected components. There is a $O(\log n)$-round $(1+\epsilon)$-approximation algorithm for single-linkage clustering in MapReduce for Euclidean space and several other metric spaces [22].

A subgraph of a geometric graph that preserves the shortest path between any pair of vertices by a factor $1+\epsilon$, for an arbitrary $\epsilon > 0$ is called a geometric spanner (ϵ-spanner) [25]. Delaunay triangulation and Yao-graph are examples of geometric spanners. To compute the Delaunay triangulation in MapReduce, lift the points into three dimensions using the transform $(x, y) \to (x, y, x^2 + y^2)$, compute their 3D convex hull, then, project the output back onto the plane. For the proof of correctness, see pages 515-516 [26]. The lifting process is embarrassingly parallel (it is obvious to solve it in parallel) and there is a constant-round MapReduce algorithm for 3D convex hull [27]. Yao-graph is constructed by partitioning the angle around each point into k cones of degree $\frac{2\pi}{k}$ and the edges of the i-th cone around each point must be parallel to each other. To compute a Yao-graph with k cones, connect each point to its nearest point from the set of points inside each cone. To route from a point s to a point t, at each step, locate the cone that contains t and go to the nearest neighbor in that cone. Yao-graph is a spanner for $k \geq 4$ [28, 29]. A constant-round MapReduce algorithm for a spanner based on Yao-graph with 4 cones, using grids and dynamic programming also exists that guarantees an approximation factor $\frac{1}{1-\frac{(2\pi/k)^2}{8}}$ for the shortest path, assuming $k \geq 3$ [14] (the paper limits the number of cones to at least 7, but the proof of lem 5 in that paper works for $k \geq 3$)[1].

1.2.3. **Graph Biconnectivity.** A PRAM algorithm for biconnectivity with $O(\log n)$ time using $O(n+m)$ edges on a graph with n vertices and m edges exists [30]. Using the simulation of Frei and Koichi [18, 19], this algorithm can be converted into an algorithm in the MapReduce model with $O(1)$ rounds.

1.3. Hardness of Optimizing DBSCAN in MRC.

1.3. Hardness of Optimizing DBSCAN in MRC. Several optimizations related to DBSCAN and its preprocessing and postprocessing steps, such as parameter setting and output-size estimation are:

[1]This is not a contradiction with the existence of Yao-graph for a smaller range of k since the algorithm for k cones builds k Yao-graphs with 4 cones, there are $4k$ cones in total.

- The existence of a good index (data structure) for finding near neighbors (points within distance r of another point): The proofs based on orthogonal vector hypothesis (OVH) that the nearest neighbor data structure and bichromatic closest pair in the d-dimensional Euclidean space cannot be solved in $O(n^{2-\epsilon}\operatorname{poly}(d))$ time, for any $\epsilon > 0$ [31] also prove the lower bounds on finding a point within distance r and bichromatic pair with distance at most r. The latter appears when we want to add points with less than f points within distance r of themselves to core points (points with at least f points within distance r of themselves).

  If the goal is to minimize the number of core points in the clusters, the problem reduces to finding the minimum connected dominating set (CDS) in unit disk graphs which is NP-hard. There is a $(2\ln D + O(1))$-approximation algorithm for CDS [32].
- Testing whether two given vertices s and t in a graph are in the same connected component known as the st-connectivity problem in MRC is conjectured to require at least a logarithmic number of rounds [22]. Also, the time complexity of the sequential version of the parallel algorithm (known as the work of the algorithm) must be subquadratic in the input size based on the restrictions of the MRC model.

The main question is whether there exists a reduction in MRC with $O(1)$ rounds that converts DBSCAN in MRC to computing the connected components in MRC. We investigate the problem using range queries for counting the number of points in each range and computing the nearest neighbors (the points from a set P that have the minimum distance to a set of query points).

1.4. Contributions.

We discuss our contributions to each of the problems separately:

- **Threshold Range Counting Query (TRC):** We give an exact algorithm with $O(1)$ rounds in MRC for $f = O(1)$.

  The previous work on range queries in MapReduce [15] for equal disks as the queries builds a Voronoi diagram (using the 3D convex hull algorithm of [27] on lifted points as described in pages 515-516 of [26]), connects each point to its neighboring cells in the Voronoi diagram, prunes away the edges of length more than r, raises the resulting graph to power f, prunes away the edges of length more than r from the resulting graph, and finally, it computes the vertex degrees and reports the vertices with degree at least f.

  Since $f = O(1)$, we were able to replace the line sweeping algorithm in [15] with graph powering to remove the constraint that each vertical line can intersect at most a sublinear number of shapes. This algorithm takes at least f rounds, which is slower than our algorithm which takes less than 4 rounds.
- **Connected Components of Unit Disk Graph (UDG) in The Euclidean Plane:** We give a constant-round MRC algorithm for the case when the diameter of the connected components is $O(1)$ and the number of connected components is sublinear in n, i.e., $(o(n))$. Building the unit disk graph might require quadratic space, so, it is not possible to build the graph and use an algorithm for finding the connected components of the graph in polylogarithmic rounds in the maximum diameter of the connected components using PRAM simulation in MRC.

We use a geometric spanner that can be built in $O(1)$ rounds in MapReduce [14] to reduce the amount of space to linear and then prune the edges of the spanner based on the length constraint of the unit disk graph of radius r to get an approximation of the distances in the unit disk graph. We give an algorithm that given a traversal of (a spanning tree of) the graph, can find the connected components in $O(1)$ rounds. We use a hierarchy of grids on the approximation of the unit disk graph that we computed to build such an ordering and then use it to find the connected components. Then, we further refine it to get the connected components of the unit disk graph.

- **Bichromatic Fixed-Radius Near Neighbors (NN) with Frequencies:** We give an exact algorithm with $O(1)$ rounds in MRC if the number of points in one color (blue) is at most f and the number of points with the other color (red) is at least f and $f = O(1)$.

  A closely related problem is bichromatic closest pair which asks for the minimum of the bichromatic nearest neighbors of the points. Because of the quadratic lower bound (ignoring polylogarithmic factors) from OVH on the time complexity sequential algorithms for bichromatic closest pair [31], there is no hope of finding a MRC algorithm for it (as its sequential simulation would violate the lower bound). However, the constraints of DBSCAN on the number of points within distance r allow us to design a MRC algorithm that has a near-linear time complexity in the sequential model.
- **Biconnected Components:** We give an algorithm with $O(1)$ rounds in MRC for computing the biconnected components of an induced subgraph of a Yao-graph with 4 cones, after removing the edges of length more than r, assuming all the conditions of our algorithm for computing the connected components. Unlike the previous work [30] that reduces the problem of biconnectivity to connectivity, we count the number of disjoint paths.

2. Preliminaries

2.1. Some Definitions from Graph Theory.

2.1.1. **Disk Graph and Unit Disk Graph (UDG).** A disk graph of radius r on a set of points P in the plane is the intersection graph of disks of radius r centered at the points of P, which means there is one vertex per point of P and an edge between two points if their disks intersect. A unit disk graph is one with $r = 1$. It is easy to see that we can scale the points such that a disk graph of radius r becomes a unit disk graph.

2.1.2. **Connectivity and k-Connectivity.** If there is a path between each pair of vertices in a graph, the graph is called connected. Assuming the graph has at least k vertices, if, for each pair of vertices, there are k disjoint paths in the graph, the graph is called k-connected. Equivalently, if less than k vertices are removed from the graph, it remains connected. Biconnectivity is 2-connectivity.

2.1.3. **Edge Contraction.** Contracting an edge $\{u, v\}$ in the graph is to replace u and v with a vertex x and for each edge $\{u, a\}$ adjacent to u other than $\{u, v\}$ and each edge $\{v, b\}$ adjacent to v other than $\{u, v\}$, add the edges $\{x, a\}$ and $\{x, b\}$.

2.2. **Theoretical Models for MapReduce.** MapReduce class (MRC) [33] is a model where data is distributed among a set of machines and the memory of each machine (m) and the number of machines (L) is both sublinear in the input size (n) and the number of rounds (R) where the machines run in parallel and communicate at the end is limited to polylogarithmic in n. Formally, there exist constants $\eta, \psi \in (0, 1)$ such that $m = O(n^\eta), L = O(n^\psi)$ and the conditions $Lm = o(n^2)$ and $R = \text{polylog}(n)$ hold.

Massively parallel computation (MPC) [34] is a more restricted model that requires the total amount of computations to be at most near-linear (that is, linear ignoring polylogarithmic factors). This means the number of machines times the memory of each machine is near-linear, i.e., $Lm = O(n \,\text{polylog}(n))$. This is different from the massively parallel model designed for graphical processing units (GPUs). Based on the definitions of MRC and MPC, all MPC algorithms are in MRC.

All the problems in class **NC** are in MRC by a simulation from PRAM to MRC [27, 18, 19]. Using the current best simulation [18, 19], the MRC algorithm has a logarithmic factor better time complexity (round complexity) than the PRAM algorithm.

Some of the basic parallel computations that have been generalized to MRC are parallel semi-group and parallel prefix sum and run in $O(1)$ rounds in MRC [27]. For a set of items $x_1, x_2, \ldots, x_n$ and a binary associative operator $\oplus$, these computations are defined as follows:

- semi-group: $x_1 \oplus x_2 \oplus \cdots \oplus x_n$, and
- prefix-sum: $x_1 \oplus x_2 \oplus \cdots \oplus x_i$, for $i = 1, \ldots, n$. There is a similar computation called diminished prefix sum where the i-th output is $x_1 \oplus x_2 \oplus \cdots \oplus x_{i-1}$, for $i = 1, \ldots, n$.

Examples of semi-group computations are computing the maximum, minimum, and sum of a set of numbers, and examples of prefix-sum are sorting (element ranking) and taking the maximum of the prefixes of a set of numbers. Polynomial-time computations on data of size $O(m)$ can also be done in one machine locally, so, it can be performed in one round or during an existing round (as a pre-processing or post-processing step).

In the MapReduce framework, the data are represented as ¡key,value¿ pairs, so, we use this notation in some of the cases. Counting the number of times an item is repeated, computing the degree of the vertices from the edge list, sorting and balanced partitioning into partitions of size m (by sorting the input into the machines) are some of the algorithms that take $O(1)$ rounds and are given in the documentation of the framework or appear many times in the literature.

We use **in parallel do** to refer to a computation done on each element in the loop independently or locally inside one machine (such as hashing), as opposed to the steps done globally using communications between several machines (such as sending data to all machines or sorting). Sometimes we only say send to mean create the local message and communicate globally to route the message to the destination machine by sorting. Sometimes, for simplicity of explanation, we use the term block from external memory algorithms to refer to a set of m consecutive data items. When we say an algorithm is in MapReduce, we mean MRC.

2.3. **Grids and Unit Disk Graphs.** A grid is a partitioning of the plane using a set of vertical and horizontal lines (axis-parallel lines). There are two main types of grids: the grid with square cells of equal size, which is called a uniform grid, and a grid where between two consecutive vertical or horizontal lines there are at most m points, for a given positive integer m, which we call a balanced grid. Next, we review MPC algorithms for uniform grids Algorithm 1 and balanced grids Algorithm 2.

Algorithm 1 Uniform Grid in MPC

```
Input: a set of points $P$, a real value $r > 0$
Output: a grid on $P$ of side length $r$
1: Find the smallest enclosing bounding box $B$ of $P$ by computing $\min_{(x,y)\in P} x, \max_{(x,y)\in P} x, \min_{(x,y)\in P} y, \max_{(x,y)\in P} y$.
2: Build a uniform square grid on $B$ with cells of side length $r$.
3: Hash the points of $P$ into the cells of $B$ by sorting the grid lines and the points of $P$ together in each coordinate and let M be the set of triples $(i, j, p)$ for $p \in P$ and cells $(i, j)$.
```

Algorithm 2 creates a grid with cells that have $O(m)$ points. To get the number of partitions near n/m, use a load-balancing or scheduling algorithm.

Algorithm 2 Balanced Grid in MPC

```
Input: a set of points $P$, an integer $m \leq |P|$
Output: a $\lceil n/m \rceil \times \lceil n/m \rceil$ grid on $P$ with $O(m)$ points in each cell
1: Sort the points of $P$ on their $x$ coordinates and let $X$ be the set of $x$ coordinates at ranks $i \times m$, for $i = 1, \dots, \lfloor n/m \rfloor$.
2: Sort the points of $P$ on their $y$ coordinates and let $Y$ be the set of $y$ coordinates at ranks $i \times m$, for $i = 1, \dots, \lfloor n/m \rfloor$.
3: Sort the points of $P$ on their $x$ coordinates in union with $X$ to partition them into $\lfloor n/m \rfloor$ subsets of size at most $m$ denoted by $P_i$, $i = 1, \dots, \lfloor n/m \rfloor$.
4: for $i = 1, \dots, \lfloor n/m \rfloor$ in parallel do
5:     Sort the points of $P_i$ on their $y$ coordinate to find the cell of the grid $X \times Y$ that contains each point.
6: Return the grid $X \times Y$ with the points of $P$ inside its cells.
```

To build a balanced grid with square cells, we use a spatial indexing method known as z-order curve. Represent the rank of a point p in a point set P in the order of x coordinates with I and the same definition for the y coordinates with J. Then, the spatial index of p is computed by interleaving the bits of I and J. For example, if $I = 5 = (110)_2$ and $J = 1 = (001)_2$, then, $z = (101001)_2$ or $z = (010110)_2$, depending on whether we interleave x with y or vice versa (we choose one of these rules and apply it in the same way to all the points).

Algorithm 3 Square Balanced Grid in MPC

```
Input: a set of points $P$, an integer $m \leq |P|$
Output: a $\lceil n/m \rceil \times \lceil n/m \rceil$ grid on $P$ with $O(m)$ points in each cell
1: Sort the points of $P$ on their $x$ coordinates.
2: Sort the points of $P$ on their $y$ coordinates.
3: for all points of $P$ in parallel do
4:     Compute the index $\phi$ of each point using the $z$-order curve on the ranks of $x$ and $y$ coordinates of that point.
5: $\sigma$ = Sort the points of $P$ in the order of their $\phi$.
6: Partition $\sigma$ into blocks of size $\Theta(m)$ such that the longest prefix in each partition is different.
7: return The bounding squares of the partitions from the previous step.
```

The round complexity of building a grid is $O(1)$, since it only involves sorting and local operations like hashing. These algorithms are a part of the literature and appear in other papers, too.

2.3.1. **Building a Disk Graph Using Grids.** One way of building unit disk graphs is by rounding to the vertices of a grid [35]. Grids are also used for solving various problems in computational geometry, including finding an approximation solution for the smallest disk that contains k points ([36, section 1.3]).

3. An Algorithm for DBSCAN in MRC

Sections 3.1, 3.5 and 3.6 discuss threshold range queries, bichromatic nearest neighbors, and bi-connectivity. Sections 3.2 to 3.4 discuss algorithms for computing connected components of graphs with a given traversal, unit disk graphs with approximations on the radii of the disks whose connected components can be covered by a sublinear number of convex connected components, and unit disk graphs with connected components that can be covered by a sublinear number of convex connected components and the diameter of the connected components is also constant. Each subsection has a sketch of the algorithm, the algorithm, and its proof.

The main idea of our range query algorithms is to bound the number of times a point directly appears in the queries to a constant number by introducing some intermediate queries. We use different types of grid-based partitioning (Algorithms 1 to 3) to provide guarantees on the shape of the cells and the number of points in the cells.

For connected components, we have two sets of constraints: 1) Because of the hardness result on the number of rounds, we look for special cases that are solvable in $O(1)$ rounds, 2) Given the possibly large size of the disk graph, we still need to keep sublinear-size summaries of the connected components. For the first problem, we assume after decomposing the connected components into a set of convex connected components (rectangles in the case of grid graphs), there are a sublinear number of connected components left. For the second problem, in disk graphs, we use a set of intervals on the partition boundaries and take their intersections to tell if the connected components on each side are in fact one component (connected).

3.1. Threshold Range Counting (TRC) in MPC.

Assume $f = O(1)$. Build a uniform grid of cell length $\frac{\sqrt{2}}{2}r$. If the number of points inside a grid cell is more than f, report them as a part of the solution S. Other cells have less than f points, so, send their points to their neighboring cells. Finally, sum the number of points within distance r of the points not yet added to S from the cell containing them and their neighboring cells and if it reaches f, add them to S. Algorithm 4 implements this algorithm. In Theorem 3.1, we prove this algorithm is in MPC and it computes the solution correctly.

Algorithm 4 Threshold Range Counting in MPC

```
Input: a point set $P$ (distributed among $\frac{n}{m}$ machines), a real value $r > 0$, an integer $f$
Output: the subset of $P$ with frequency $f$
 1: Build a grid $B$ using Algorithm 1 with parameters $P$ and $\sqrt{2}/2r$.
 2: Count the number of points $\nu$ in each cell $(i, j)$ of $B$.
 3: for cells $(i, j)$ of $B$ with $\nu \geq f$ in parallel do
 4:     Add all the points in cell $(i, j)$ to $O$.
 5: for $p \in P \setminus O$ in parallel do
 6:     Cell $(i, j)$ = the cell of $B$ that contains $p$
 7:     $F(p, (i, j))$ = the number of points in the disk of radius $r$ around $p$ in cell $(i, j)$.
 8:     Send $p$ to the 8 neighboring cells of $(i, j)$.
 9: for $p \in P \setminus O$ in parallel do
10:     Compute $F(p, (i', j'))$ for each request.
11:     Send $F(p, (i', j'))$ to the cell $(i, j)$ containing $p$.
12: for $p \in P \setminus O$ in parallel do
13:     $f_p$ = the sum $F(p, (i', j'))$ for each $p$.
14:     if $f_p \geq f$ then
15:         Create a request to add $p$ to $O$.
16: Add the points from Line 15 to $O$ and output $O$.
```

Theorem 3.1. *For $f = O(1)$,* Algorithm 4 *finds the subset of points that have f points within distance r of themselves in MPC using $O(1)$ rounds.*

Proof. Building a uniform grid takes $O(1)$ rounds. In a grid of cell length $\frac{\sqrt{2}}{2}r$, equivalently, a square cell with diameter r, all the points have distance at most r. So, if there are f points in a cell $c = (i, j)$, all of them are dense (have f points within radius r of themselves). Otherwise, if a point in cell c has at least f near neighbors, some of these neighboring points are in the 8 neighboring cells since even if the point is on the boundary of c, its neighbors of distance at most r can only be in a neighboring cell.

Since c had less than f points and $f = O(1)$, then, all the points in c can be sent to neighboring cells without exceeding the memory constraints. At most $8f$ points are sent to each machine, so, it is possible to test all the cases which is at most $8fn = O(n)$ in one round. Sending these counts also takes the same amount of memory and requires another round. Summing up the values for each of the points in sparse cells (cells with less than f points) and comparing the count against f is done locally. The overall round complexity is $O(1)$ rounds. □

3.2. **Connected Components in MPC Given Graph Traversals.** Trees with a given traversal can be easily partitioned into blocks of adjacent vertices: If the order of the edges of the graph is based on a traversal, for example, breadth-first search (BFS) or depth-first search (DFS), then, we can compute the connected components in each machine since the neighbors of the vertices are in the same machine as those vertices.

In these cases, we show that the connectivity problem and the problem of computing the connected components also become easy. By scanning the edges in each machine and contracting them into one vertex in case of one connected component or more in case of more connected components, then, we update the edges to be between these new vertices and repeat this. So, we can find all the connected components in $O(1)$ rounds in MRC if there are a sublinear number of connected components.

Algorithm 5 Connected Components of Graphs with Traversals in MRC

```
Input: a graph $G = (V, E)$, a traversal order $\phi$ on $G$
Output: the connected components of $G$
 1: Compute the vertex degrees $\deg(v)$ of $V \in G$
 2: for $v \in G$ in parallel do
 3:     if $\deg(v) > m$ then
 4:         Add $\lceil \frac{\deg(v)}{m} \rceil$ new vertices to $G$ and break the adjacency list of $v$ among them.
 5: Partition $E$ in the order of $\phi$ into blocks of size $m$ denoted by $S_1, \dots, S_{n/m}$.
 6: for $i = 1, \dots, n/m$ in parallel do
 7:     Compute the connected components $CC(u), \forall u \in S_i$.
 8: while $E \neq \emptyset$ in parallel do
 9:     for $(u, v) \in E, u \in S_i, v \in S_j, i \neq j$ in parallel do
10:         Send $CC(u)$ to $S_j$ and $v, CC(v)$ to $S_i$ along with $(u, v)$.
11:     for $(u, v) \in E, u \in S_i, v \in S_j, i \neq j, CC(u) \neq CC(v)$ in parallel do
12:         $CC(u) = CC(v) = \min(CC(u), CC(v))$.
13:     for $(u, v) \in E, u, v \in S_i, CC(u) \neq CC(v)$ in parallel do
14:         $CC(u) = CC(v) = \min(CC(u), CC(v))$.
15:     Contract the edges $(u, v) \in E$ with $CC(u) = CC(v)$.
16: return the connected component of each vertex $CC(u), \forall u \in P$
```

Theorem 3.2. Algorithm 5 *finds the connected component of G in $O(1)$ rounds in MRC, if the number of the connected components of G is $O(m)$.*

Proof. Computing the degree of vertices is a summation over the pairs (u, v) and (v, u) for the edge between vertices u and v. After sorting these pairs on their first member (vertex), we get a series of summations. Since summation is a semi-group computation and the total number of pairs is $2|E| = O(n)$, these summations can be performed simultaneously by different machines that contain them in $O(1)$ rounds.

Breaking vertices of degree more than m into $\lceil \frac{\deg(v)}{m} \rceil$ new vertices guarantees the adjacent vertices to each vertex fit inside the memory of a single machine. Updating the edges requires at most one sorting, which takes $O(1)$ rounds in MRC.

Since we have a traversal of the graph (in the order ϕ), then, the vertices of a connected component appear consecutively. So, in each machine, there are $O(m)$ connected components. This case happens when some of the connected components are small and fit entirely inside the memory of a single machine.

Mapping the vertices to the index of the connected components that contain them and updating the edges takes two rounds using hashing on the vertices (at the end of the round where we compute the connected components) and then on the edges. The total space is proportional to the changed edges and vertices, which is $O(n)$.

At most $O(m)$ machines can have more than one cluster, so, at the first iteration of the while loop at least $\lceil (n - O(m))/m \rceil$ edges are contracted into a single vertex (both their endpoints belong to the same component). By induction on the number of iterations, at the t-th iteration there are $O(n/m^t)$ edges. This means the while loop is repeated $O(\log_m n)$ times. Substituting $m = O(n^\eta)$ from the definition of MRC gives $O(1/\eta) = O(1)$ rounds. □

3.3. **Connected Components of Approximate UDG in $\mathbb{R}^2$.** We relax this problem to finding the connected components if we are allowed to increase the radius of the disk graph by a factor at most $1 + \epsilon$, for a positive arbitrary constant ϵ. First, we compute a Yao-graph spanner with 4 cones in MapReduce. We choose one of the cones (the one that contains the edges in the direction of right and up), then, similar to the dynamic program for building the Yao-graph in MapReduce described in [14], we compute the connected components inside each block. Finally, we merge the connected components using the edges between different blocks.

In order for this to work, each connected component must consist of a set of convex polygons such that the total number of these polygons is sublinear in the input size. Otherwise, the number of components inside the blocks might be too many for the algorithm to merge given the memory bounds of MapReduce. We show that Algorithm 6 implements this and finds the connected components.

Algorithm 6 Connected Components of Approximate UDG in The Plane

```
Input: a set of points $P$, a real value $r > 0$
Output: the connected components of an approximate UDG of $P$ of radius $r$
1: Compute a Yao-graph spanner $H$ with 4 cones on $P$ using the algorithm of [14].
2: Remove the edges of length more than $3r$ from $H$.
3: Build a balanced grid $B$ using Algorithm 2.
4: Send the vertices that fall on cell boundaries of $B$ to their neighboring cells.
5: Compute the connected components in each cell of $B$.
6: for $t = 1, \dots, \log_m n$ do
7:     for the cells created by merging $m$ consecutive cells from the previous step in each dimension starting from multiples of $m$ in parallel do
8:         Compute the intervals that are the intersection of the connected components with cell boundaries.
9:         Merge the connected components of the $m$ consecutive cells by taking the intersections of their intervals.
```

Lemma 3.3. *Removing the edges of length more than $2(1+\epsilon)r$ from a spanner preserves the connected components of the subgraph with the edges of length at most $2r$, for any constant $\epsilon > 0$.*

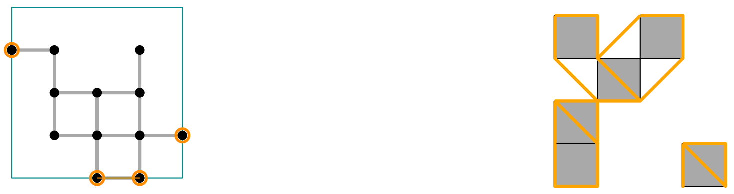
Proof. If two vertices are connected with an edge of length $2r$, then, based on the definition of spanners, they are connected with a path of length at most $2r(1+\epsilon)$ in an ϵ-spanner. If there is an edge of length more than $2r(1+\epsilon)$ in a path, the length of the path would exceed $2r(1+\epsilon)$ because of that edge, which is a contradiction. □

Lemma 3.4. *In* Algorithm 6*, the connected components of the Yao-graph are computed correctly.*

Proof. Computing the connected components in each cell preserves the connected components of the subgraph of H inside that cell because its construction is equivalent to contracting the edges inside each cell. A connected component of a grid graph can connect to a connected component in an adjacent cell if they share vertices. So, taking the intersection of the intervals on the boundaries guarantees the connectivity is preserved. □

In order to keep the number of connected components bounded throughout the algorithm, we add a condition on the number of convex polygons needed to cover the connected components. Figure 1a shows a connected component of a part of a grid graph computed by a Yao-graph with 4 cones. It can be decomposed into 5 convex connected components. The number of intervals on the boundaries is at most as many as monotone connected components, however, in this example, it is 3 (less than the maximum number of intervals 5).

Figure 1b shows the number of connected components in G in Algorithm 6 is independent from the number of connected components in H. So, a traversal of G might not correspond to any traversal of H and a sorting step similar to the one we used in the algorithm is necessary.



(A) The subset of points stored from a connected component inside a cell.

(B) Possible connected components of the grid (black cells are non-empty).

FIGURE 1. Two levels of grids in Algorithm 6.

Theorem 3.5. *If the total number of monotone pieces of connected components of H is $O(m)$, then, Algorithm 6 finds the connected components of the approximate UDG with disks of radius $3r$ in $O(1)$ rounds of MRC, where m is the memory of a single machine in MRC.*

Proof. Computing a spanner graph using [14] takes $O(1)$ rounds in MRC. The stretch factor of the spanner based on Yao-graph in [14] with 4 cones is:

$$\frac{1}{1-\frac{(2\pi/\kappa)^2}{8}} = \frac{1}{1-\frac{\pi^2}{32}} < 3/2.$$

Using Lemma 3.3, computing a 2-spanner preserves the connected components of the disk graph of radius $3/2r$.

Building a grid takes $O(1)$ rounds. The number of boundary vertices sent to adjacent machines (at most 4 machines- it happens for the point that is a vertex of the grid) is $O(m)$, since the construction of the grid guarantees there are at most m points between consecutive lines (both vertical and horizontal). Since this repeats the data at most 4 times, the memory of each machine remains $O(m)$ and the total memory used is also $O(n)$. Each cell of the grid has $O(m)$ points, so, computing the connected components in each cell can be done locally by a single machine.

Using Lemma 3.4, the connected components of the Yao-graph are preserved. The convex connected components are rectangles in grid graphs. We assumed there are $O(m)$ connected components. Cutting a convex connected component with a rectangle (such as a grid cell) creates at most 4 components. So, the number of connected components in each cell built by merging smaller cells is $O(m)$. So, the number of intervals on the cell boundaries for these connected components is $O(m)$ throughout the execution of the algorithm. Taking the intersection of the intervals and assigning the minimum of the numbers assigned to their connected components are two semi-group computations that can be performed simultaneously. This takes $O(\log_m n) = O(\frac{1}{\eta}) = O(1)$ rounds. Computing the connected components is done by semi-group, so, it takes $O(1)$ rounds, for each t. So, it takes $O(\log_m n)$ rounds in total. □

3.3.1. **Bounding The Number of Convex Connected Components in MRC.** A polygonal curve is a sequence of points where each point other than the last point, is connected to the next point. The ply of a one-dimensional polygonal curve T, denoted by $\mathrm{ply}(T)$, is the maximum number of edges that contain a point on the real line. We prove the number of convex connected components is at most $\text{ply}(T_x) \times \text{ply}(T_y)$, where T is a walk that visits all the vertices of a disk graph and T_x and T_y are the projections of T on the x and y axes of the Cartesian coordinate system.

First, we give a sequential algorithm and a proof based on induction. To compute a spanning walk T, first, traverse the graph to get a sequence of the vertices, that without loss of generality, we assume is $v_1, v_2, \ldots, v_n$. We use induction on the length of the prefix of the traversal to prove at each step, we have a walk. To convert the spanning tree of this traversal to a walk, visit the vertices in the order of the traversal as long as there is an edge between two consecutive vertices, otherwise, backtrack (go back in the sequence) to the first vertex that has such an edge. This is always possible since the graph is connected and there is a path between any two vertices in a spanning tree, including the one constructed so far by the algorithm (based on the construction of the traversal algorithm, each prefix of the traversal is a tree). At each step, the algorithm goes back at most $n-1$ vertices, so, the size of T is $O(n^2)$.

In MRC, we cannot build a walk of size $\Theta(n^2)$, as the memory is limited to $o(n^2)$. Another way to make sure the number of connected components does not exceed m is to run Algorithm 6 and terminate as soon as the number of connected components during the merging of neighboring cells exceeds m. This does not necessarily mean the number of convex connected components is $O(m)$, but it allows us to run the algorithm.

3.4. Bounded-Diameter Connected Components of UDG in $\mathbb{R}^2$.

When the diameter of a (connected component of a) graph G is $\Delta = O(1)$, it is possible to compute G^Δ (G to the power Δ) in Δ rounds by running a breadth-first search with depth Δ from each vertex.

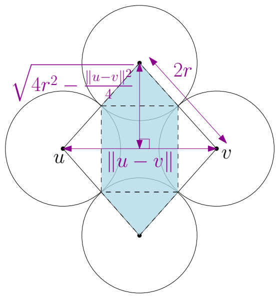
In unit disk graphs, if the diameter is Δ, the maximum radius that needs to be checked is at most

$$\frac{\Delta-1}{2}2r = (\Delta-1)r.$$

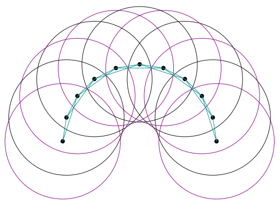
So, the queries for computing the connectivity of two points can be defined in one round by checking if points with distances at most $2r, 4r, \ldots, 2\lceil\frac{\Delta-1}{2}\rceil r$ exist in the regions around each point, in each of the four cones (right and up, right and down, left and up, left and down).

Figure 2a shows an example of dividing the region into 3 polygonal queries of constant complexity for $\Delta = 2$. There might be longer paths between the points than $\Delta - 1$, but we only need to find one of them. See Figure 2b for an example with the longest path of length 10 and the shortest path of length 5.

Algorithm 7 solves the connected components problem for UDG using a connectivity testing subroutine for UDG with $\Delta = O(1)$.



(A) The region that can connect the two disks of radius r on the right and the left if it contains a point with distance 2.



(B) There can be an arbitrary number of disks connecting two disks. Here, the diameter of the UDG is 5.

FIGURE 2. Connectivity queries for connectivity testing in unit disk graphs.

Algorithm 7 Bounded-Diameter Connected Components of UDG

```
Input: a set of points $P$, a real value $r > 0$
Output: the connected components of the UDG of $P$ of radius $r$
 1: $C_1, \dots, C_k$ = the output of Algorithm 6 for $P$ and $r$.
 2: for $i = 1, \dots, k$ in parallel do
 3:     $E_i$ = the edges of $C_i$ of length more than $2r$.
 4: for $i = 1, \dots, k$ in parallel do
 5:     for $(u, v) \in E_i$ in parallel do
 6:         $Q_i = Q_i \cup$ queries for computing the connectivity of the disk of radius $r$ centered at $u$ and
            $v$ in the UDG.
 7: $Q = \cup_{i=1}^{k} Q_i$
 8: Build a square balanced grid $B$ using Algorithm 3 on $P$ with radius $2r$.
 9: Break the ranges of $Q$ by taking their intersection with the cells of $B$.
10: for in each cell $c$ of $B$ in parallel do
11:     Locally compute the queries $Q$ in $c$ using the subset of the points of $P$ in $c$.
12: for $q \in Q$ in parallel do
13:     Answer $q$ based on the emptiness of its pre-computed subqueries.
14: Aggregate the results of the queries in $Q$ for each edge $e \in \cup_{i=1}^{k} E_i$.
15: Return the edges of $C_i$, for $i = 1, \dots, k$ of length at most $2r$ union the edges with non-empty
    queries from the previous step.
```

Lemma 3.6. *The number of queries required to compute the connectivity of a pair of points in the UDG of diameter Δ is $O(\text{polylog}(n))$ for $\Delta = O(\text{polylog}(n))$.*

Proof. Since the Euclidean plane is a doubling space with doubling dimension $\log_2 7$, a disk of radius $2r$ can be covered by 7 disks of radius r. We have $2\lceil\frac{\Delta-1}{2}\rceil \leq \Delta$. So, a disk of radius Δr can be covered by at most $7^{\log_2(\Delta)} = \Delta^{\log_2 7}$ disks of radius r. Since $\Delta = O(\text{polylog}(n))$, the number of disk queries with distance at most Δr is also $\Delta^{\log_2 7} = O(\text{polylog}(n))^{O(1)} = O(\text{polylog}(n))$.

The shape of the region for a pair of disks is a hexagon with opposing edges that are parallel and equal (See Figure 2a). Drawing axis-parallel lines that are $2r$ apart (grid lines) results in at most 4 polygons with at most 6 vertices. So, the number of queries is $O(\text{polylog}(n))$. □

Theorem 3.7. *Algorithm 7 finds the connected components of UDG in $O(1)$ rounds, assuming the number of convex pieces of connected components is sublinear in n, for $m = \Omega(\sqrt{n})$ and the diameter of each convex piece is at most $\Delta = O(\text{polylog}(n))$.*

Proof. Using Theorem 3.5, Algorithm 6 takes $O(1)$ rounds. To run an algorithm on the connected components, it is enough to sort and partition them. Pruning the edges takes one round. The queries for each pair of points can be computed locally. Building a square balanced grid takes $O(1)$ rounds. Based on Lemma 3.6, we have $|Q| = O(n\,\text{polylog}(n))$. So, creating the queries takes one round. Locally computing the queries takes one round. Sorting and sending a constant number of copies of the answered queries to be summed up takes another round. In Algorithm 7, the AND of the binary queries (empty/non-empty) for each path are computed and the OR of the paths gives the result of the aggregation that was to determine if a path exists. Aggregating the results of the queries takes $O(1)$ rounds for computing the paths of length $O(\text{polylog}\, n)$. The edges of length at most $2r$ are the edges of C_i, for $i = 1, \dots, k$, that are not in $\cup_{i=1}^{k} E_i$, which we computed in the first for loop of the algorithm. If the results are saved at that step, then it does not require another round. Otherwise, a round is needed to prune away the edges of length more than $2r$. The edges computed by the queries are already available as they were created after the aggregation. So, the total number of rounds is $O(1)$ and the algorithm correctly computes all the edges without violating the memory constraints of MRC. □

3.5. **Bichromatic Near Neighbor (NN) with Frequency Limit.**

Assume P is the set of points with frequency at least f and Q be the points with frequency less than f. We call a point dense if the number of points within radius r of it is at least f. Build a uniform grid with cell diameter r. If a cell contains points from both P and Q, then, we connect all the points of Q in that cell to one of the points of P in that cell. Otherwise, we send the subset of Q in that cell to the neighboring cells which does not violate the condition because the number of points of Q in each cell is less than f and $f = O(1)$.

Algorithm 8 Bichromatic Fixed-Radius Near Neighbor (NN) Range Query

Input: A set of points P, a set of points Q, a real value $r > 0$

Output: The index of the connected component of each point of Q in the disk graph

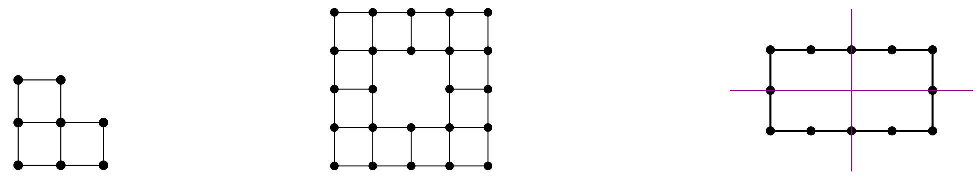
```
1: Build a grid $B$ using Algorithm 1 with parameters $P$ and $\sqrt{2}/2r$.
2: Check if each cell of $B$ has at least one point of $P$ by hashing the points to the cells, then, mark
   one of the points of $P$ in that cell.
3: for each point $q \in Q$, $q \in C$ in parallel do
4:     Cell $c =$ the cell in $B$ that contains $q$.
5:     if there is a point $p \in P$ in $c$ then
6:         Add $q$ to the component of $p$.
7:     else
8:         Send $q$ to all neighboring cells of $c$.
9: $Q' =$ the set of points sent to other cells in the previous step.
10: for each non-empty cell $c$ of $B$ in parallel do
11:     Build a balanced grid $B_c$ in $c$ using Algorithm 2.
12: for $q \in Q'$, $q$ was sent to cell $c$ in Line 8 in parallel do
13:     if there is a $p \in P$, $p \in c$, and $\text{dist}(p, q) \leq r$ then
14:         Add $q$ to the component of $p$
15: return a near-neighbor of each point of $Q$ from Lines 6 and 14.
```

Theorem 3.8. Algorithm 8 *solves the bichromatic fixed-radius near neighbors in $O(1)$ rounds in MRC if one of the point sets (P) contains only dense points and the other point set (Q) contains no dense points, for $f = O(1)$.*

Proof. Building a uniform grid takes $O(1)$ rounds. Checking if the cells contain points from both point sets can be done locally in each cell, so, it takes $O(1)$ rounds. Sending the points of Q to neighboring cells takes $8|Q| = O(n)$ communications and $O(1)$ rounds. Since the points of Q are non-dense, the number of points within distance r of them are at most $8f = O(1)$ points. So, a cell containing a point of Q has less than $f = O(1)$ points and sending them to be checked against other points involves $O(n)$ pairs in total. So, the last for takes $O(1)$ round. The total round complexity of the algorithm is $O(1)$. □

3.6. **Biconnected Components in DBSCAN (BCD).** Assume the number of convex connected components is sublinear and the diameter of the connected components is $\Delta = O(1)$. Figure 3 shows some examples of biconnected components in grid graphs.

One way to find the biconnected components is to modify Algorithm 7 to work on all the edges of the graph and count the number of paths between two points while considering whether they are disjoint or not. This is possible because the diameter is assumed to be $O(1)$.



(A) A non-convex biconnected component.

(B) A biconnected component with holes. Not a Yao-graph. Not an induced subgraph of a grid graph.

(C) A biconnected component with four non-biconnected components after being split between four cells.

FIGURE 3. Examples of biconnected components and their properties.

Algorithm 9 Bounded-Diameter Biconnected Components of Disk Graph

```
Input: a set of points $P$, a real value $r > 0$
Output: the biconnected components of the disk graph of $P$ of radius $r$
 1: Compute a Yao-graph $H = (P, E)$ with 4 cones on $P$ using the algorithm of [14].
 2: for $(u, v) \in E$ in parallel do
 3:     $Q = Q \cup$ queries for the connectivity of $u$ and $v$ in the disk graph of radius $r$.
 4: Build a square balanced grid $B$ using Algorithm 3 on $P$ with cell side $2r$.
 5: Break the ranges of $Q$ by taking their intersection with the cells of $B$.
 6: for in each cell $c$ of $B$ in parallel do
 7:     $Q_c =$ the part of the queries in $Q$ that fall inside the cell $c$.
 8:     $P_c =$ the subset of $P$ that falls inside the cell $c$.
 9:     Locally count the number of points of $P_c$ in queries $Q_c$.
10: for $q \in Q$ in parallel do
11:     $C(q) =$ the number of points in pre-computed subqueries $\forall c \in B : Q_c \subset q$.
12: for each edge $(u, v) \in E$ in parallel do
13:     $C((u, v)) =$ the number of disjoint paths between $u$ and $v$.
14: $E' =$ the edges with $C(e) \geq 2$, for all $e \in E$.
15: Compute the connected components of $G = (P, E')$ using Algorithm 7.
16: Return the connected components from the previous line.
```

Theorem 3.9. Algorithm 9 *finds the biconnected components of UDG of radius* $3/2r$ *in* $O(1)$ *rounds in MRC, assuming the number of convex pieces of connected components is sublinear in* n*, for* $m = \Omega(\sqrt{n})$ *and the diameter of each convex piece is at most* $\Delta = O(1)$.

Proof. The only changes made to Algorithm 7 are on the computation done on the points of the queries and the way the results are aggregated. In Algorithm 9, after computing the number of disjoint paths from the polygonal queries for each path in the set of paths connecting the endpoints of an edge e using $2t - 1$ edges, we find the maximum sum possible using a subset of disjoint paths. Since the number of queries for each edge is $O(1)$ using $\Delta = O(1)$ (see the proof of Lemma 3.6), the number of paths is $O(1)^{O(1)}) = O(1)$. So, this still takes $O(1)$ rounds. Using the proof of Theorem 3.7, the round complexity of the algorithm is $O(1)$. □

4. Conclusions and Open Problems

We gave a MapReduce algorithm for DBSCAN with a constant number of rounds and subquadratic communications by assuming some constraints on its various parameters. Prior to our work, the only algorithm is the database implementation based on self-join that can be made parallel using triangle counting problem in MapReduce [37]. Our methods for simplex (polygonal) range queries in MapReduce by breaking them into smaller pieces using a grid, computing a summary for connectivity in intersection graphs by taking the intersection of the shapes with the boundary of the range used in the partitioning, and creating the list of the edges of length at most r using a geometric spanner might be useful to similar problems as well. Instead of using methods similar to randomly shifted grid [24], we checked all the possible cases after adding an assumption on the complexity of the items that need to be checked.

References

[1] J. Gan and Y. Tao, On the hardness and approximation of Euclidean DBSCAN, *ACM Trans. Database Syst.*, **42** no. 3 (2017) 1–45.

[2] Y. Wang, Y. Gu and J. Shun, Theoretically-efficient and practical parallel DBSCAN, In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, (2020) 2555–2571.

[3] N. Bansal, A. Blum and S. Chawla, Correlation clustering, *Mach. Learn.*, **56** (2004) 89–113.

[4] N. Ailon, M. Charikar and A. Newman, Proofs of conjectures in "aggregating inconsistent information: Ranking and clustering", Technical report, Technical Report TR-719-05, Department of Computer Science, Princeton University, 2005.

[5] S. Chawla, K. Makarychev, T. Schramm and G. Yaroslavtsev, Near optimal LP rounding algorithm for correlation clustering on complete and complete k-partite graphs, In *Proceedings of the 2015 ACM Symposium on Theory of Computing*, ACM, New York, (2015) 219–228.

[6] M. Ester, H.-P. Kriegel, J. Sander and X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, (1996) 226–231.

[7] J. H. Friedman, J. L. Bentley and R. A. Finkel, An algorithm for finding best matches in logarithmic expected time, *ACM Transactions on Mathematical Software*, **3** no. 3 (1977) 209–226.

[8] J. L. Bentley, Survey of techniques for fixed radius near neighbor searching, *Technical report*, Stanford Linear Accelerator Center, Calif.(USA), 1975.

[9] J. Han, M. Kamber and J. Pei, *Data mining concepts and techniques third edition*, University of Illinois at Urbana-Champaign Micheline Kamber Jian Pei Simon Fraser University, 2012.

[10] R. C. Hoetzlein, Fast fixed-radius nearest neighbors: interactive million-particle fluids, In *GPU Technology Conference*, **18** (2014) pp. 2.

[11] M. de Berg, A. Gunawan and M. Roeloffzen, Faster DBSCAN and HDBSCAN in low-dimensional Euclidean spaces, *Internat. J. Comput. Geom. Appl.*, **29** no. 1 (2019) 21–47.

[12] E. Schubert, J. Sander, M. Ester, H. P. Kriegel and X. Xu, DBSCAN revisited, revisited: why and how you should (still) use DBSCAN, *ACM Trans. Database Syst.*, **42** no. 3 (2017) 21 pp.

[13] P. K. Agarwal, K. Fox, K. Munagala and A. Nath, Parallel algorithms for constructing range and nearest-neighbor searching data structures, In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, (2016) 429–440.

[14] S. Aghamolaei, F. Baharifard and M. Ghodsi, Geometric spanners in the MapReduce model, *Computing and combinatorics*, Lecture Notes in Comput. Sci., Springer, Cham, 2018 675–687.

[15] S. Aghamolaei, V. Keikha, M. Ghodsi and A. Mohades, Windowing queries using Minkowski sum and their extension to MapReduce, *J. Supercomput.*, **77** (2021) 936–972.

[16] S. Aghamolaei, V. Keikha, M. Ghodsi and A. Mohades, Sampling and sparsification for approximating the packedness of trajectories and detecting gatherings, *Int. J. Data Sci. Anal.*, **15** no. 2 (2023) 201–216.

[17] J. H. Reif and S. Sen, Optimal randomized parallel algorithms for computational geometry, *Algorithmica*, **7** no. 1 (1992) 91–117.

[18] F. Frei and K. Wada, Efficient circuit simulation in MapReduce, In *Proceedings of the 30th International Symposium on Algorithms and Computation*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019 pp. 21.

[19] F. Frei and K. Wada, Efficient deterministic MapReduce algorithms for parallelizable problems, *Journal of Parallel and Distributed Computing*, **177** (2023) 28–38.

[20] M. Garofalakis, J. Gehrke and R. Rastogi, Data stream management: A brave new world, In *Data Stream Management: Processing High-Speed Data Streams*, (2016) 1–9.

[21] L. Arge, *External memory data structures*, Handbook of massive data sets, Massive Comput., **4**, Kluwer Acad. Publ., Dordrecht, 2002 313–357.

[22] G. Yaroslavtsev and A. Vadapalli, Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p-distances, In *Proceedings of the 35th International Conference on Machine Learning*, (2018).

[23] D. Nanongkai and M. Scquizzato, Equivalence classes and conditional hardness in massively parallel computations, *Distrib. Comput.*, **35** no. 2 (2022) 165–183.

[24] A. Andoni, A. Nikolov, K. Onak and G. Yaroslavtsev, Parallel algorithms for geometric graph problems, In *STOC'14—Proceedings of the 2014 ACM Symposium on Theory of Computing*, ACM, New York, (2014) 574–583.

[25] G. Narasimhan and M. Smid, *Geometric spanner networks*, Cambridge University Press, 2007.

[26] C. D. Toth, J. O'Rourke and J. E. Goodman, *Handbook of discrete and computational geometry*, CRC press, second edition edition, 2004.

[27] M. T. Goodrich, N. Sitchinava and Q. Zhang, Sorting, searching, and simulation in the MapReduce framework, In *Proceedings of the 22nd International Symposium on Algorithms and Computation*, Springer Science & Business Media, **7074** (2011) 374–383.

[28] L. Barba, P. Bose, M. Damian, R. Fagerberg, W. L. Keng, J. O'Rourke, A. van Renssen, P. Taslakian, S. Verdonschot and G. Xia, New and improved spanning ratios for yao graphs, *J. Comput. Geom.*, **6** no. 2 (2015) 19–53.

[29] D. Bakhshesh and M. Farshi, A lower bound on the stretch factor of yao graph y4, *Scientia Iranica*, **29** no. 6 (2022) 3244–3248.

[30] R. E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, *SIAM J. Comput.*, **14** no. 4 (1985) 862–874 (1985).

[31] K. Bringmann, Fine-grained complexity theory: Conditional lower bounds for computational geometry, In *Proceedings of the 17th conference on computability in Europe*, Springer, (2021) 60–70.

[32] S. Guha and S. Khuller, Approximation algorithms for connected dominating sets, *Algorithmica*, **20** (1998) 374–387.

[33] H. Karloff, S. Suri and S. Vassilvitskii, A model of computation for MapReduce, In *Proceedings of the 21st annual ACM-SIAM symposium on Discrete Algorithms*, SIAM, Philadelphia, PA, (2010) 938–948.

[34] P. Beame, P. Koutris and D. Suciu, Communication steps for parallel query processing, *J. ACM*, **64** no. 6 (2017) 58 pp.

[35] J. L. Bentley, D. F. Stanat and E. H. Williams Jr, The complexity of finding fixed-radius near neighbors, *Information Processing Lett.*, **6** no. 6 (1977) 209–212.

[36] S. Har-Peled, *Geometric approximation algorithms*, Mathematical Surveys and Monographs, **173**, American Mathematical Society, Providence, RI, 2011.

[37] S. Suri and S. Vassilvitskii, Counting triangles and the curse of the last reducer, In *Proceedings of the 20th international conference on World wide web*, (2011) 607–614.

Sepideh Aghamolaei

Department of Computer Engineering, Sharif University of Technology, Tehran, Iran

Email: `sepideh.aghamolaei14@sharif.edu`

Mohammad Ghodsi

Department of Computer Engineering, Sharif University of Technology, Tehran, Iran and

School of Computer Science, Institute for Research in Fundamental Sciences (IPM) Tehran, Iran

Email: `ghodsi@sharif.edu`