A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-Set Data Structure

Md. Mostofa Ali Patwary[1,†], Diana Palsetia[1], Ankit Agrawal[1], Wei-keng Liao[1], Fredrik Manne[2], Alok Choudhary[1]
[1]Northwestern University, Evanston, IL 60208, USA [2]University of Bergen, Norway
[†]Corresponding authors: mpatwary@eecs.northwestern.edu

***Abstract*—DBSCAN is a well-known density based clustering algorithm capable of discovering arbitrary shaped clusters and eliminating noise data. However, parallelization of DBSCAN is challenging as it exhibits an inherent sequential data access order. Moreover, existing parallel implementations adopt a master-slave strategy which can easily cause an unbalanced workload and hence result in low parallel efficiency.**

We present a new parallel DBSCAN algorithm (PDSDBSCAN) using graph algorithmic concepts. More specifically, we employ the disjoint-set data structure to break the access sequentiality of DBSCAN. In addition, we use a tree-based bottom-up approach to construct the clusters. This yields a better-balanced workload distribution. We implement the algorithm both for shared and for distributed memory.

Using data sets containing up to several hundred million high-dimensional points, we show that PDSDBSCAN significantly outperforms the master-slave approach, achieving speedups up to 25.97 using 40 cores on shared memory architecture, and speedups up to 5,765 using 8,192 cores on distributed memory architecture.

***Index Terms*—Density based clustering, Union-Find algorithm, Disjoint-set data structure.**

I. Introduction

Clustering is a data mining technique that groups data into meaningful subclasses, known as *clusters*, such that it minimizes the intra-differences and maximizes inter-differences of these subclasses [1]. Well-known algorithms include K-means [2], K-medoids [3], BIRCH [4], DBSCAN [5], STING [6], and WaveCluster [7]. These algorithms have been used in various scientific areas such as satellite image segmentation [8], noise filtering and outlier detection [9], unsupervised document clustering [10], and clustering of bioinformatics data [11]. Existing data clustering algorithms have been roughly categorized into four classes: partitioning-based, hierarchy-based, grid-based, and density-based [12], [13]. DBSCAN (Density Based Spatial Clustering of Applications with Noise) is a density based clustering algorithm [5]. The key idea of the DBSCAN algorithm is that for each data point in a cluster, the neighborhood within a given radius (*eps*) has to contain at least a minimum number of points (*minpts*), i.e. the density of the neighborhood has to exceed some threshold.

The DBSCAN algorithm is challenging to parallelize as its data access pattern is inherently sequential. Many existing parallelizations adopt the master-slave model. For example, in [14], the data is equally partitioned and distributed among the slaves, each of which computes the clusters locally and sends back the results to the master in which the partially clustered results are merged sequentially to obtain the final result. This strategy incurs a high communication overhead between the master and slaves, and a low parallel efficiency during the merging process. Other parallelizations using a similar master-slave model include [15], [16], [17], [18], [19], [20]. Among these master-slave approaches, various programming mechanisms have been used, for example, a special parallel programming environment, named skeleton based programming in [17] and parallel virtual machine in [19]. A Hadoop-based approach is presented in [18].

To overcome the performance bottleneck due to the serialized computation at the master process, we present a fully distributed parallel algorithm that employs the disjoint-set data structure [21], [22], [23], a mechanism to enable higher concurrency for data access while maintaining a comparable time complexity to the classical DBSCAN algorithm. The idea of our approach is as follows. The algorithm initially creates a singleton tree (single node tree) for each point of the dataset. It then keeps merging those trees belonging to the same cluster by using the disjoint-set data structure until it has discovered all the clusters. The algorithm thus generates a single tree for each cluster containing all points of the cluster. Note that the merging can be performed in an arbitrary order. This breaks the inherent data access order and achieves high data parallelization resulting in the first truly scalable implementation of the DBSCAN algorithm. The parallel DBSCAN algorithm is implemented in C++ both using OpenMP and MPI to run on shared-memory machines and distributed-memory machines, respectively. To perform the experiments, we used a rich testbed consisting of instances from real and synthetic datasets containing hundreds of millions of high dimensional data points. The experiments conducted on a shared-memory machine show scalable performance, achieving a speedup up to a factor of 25.97 on 40 cores. Similar scalability results have been obtained on a distributed-memory machine with a speedup up to 5,765 using 8,192 cores. Our experiments also show that PDSDBSCAN significantly outperforms previous approaches to parallelize DBSCAN. Moreover, we observe that the disjoint-set data structure based sequential DBSCAN algorithm performs equally well compared to the existing classical DBSCAN algorithm both when considering the time complexity and also when comparing the actual performance without sacrificing the quality of the solution.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

The remainder of this paper is organized as follows. In Section II, we describe the classical DBSCAN algorithm. In Section III, we propose the new disjoint-set based DBSCAN algorithm, and it's parallel version along with correctness and time complexities in Section IV. We present our experimental methodology and results in Section V. We conclude our work and propose future work in Section VI.

II. The DBSCAN Algorithm

DBSCAN is a clustering algorithm that relies on a density-based notion of clusters [5]. The basic concept of the algorithm is that for each data point in a cluster, the neighborhood within a given radius (eps) has to contain at least a minimum number of points ($minpts$), i.e. the *density* of the neighborhood has to exceed some threshold. A short and brief description based on [5], [9], [20] is given below.

Let X be the set of data points to be clustered using DBSCAN. The neighborhood of a point $x \in X$ within a given radius eps is called the *eps-neighborhood* of x, denoted by $N_{eps}(x)$. More formally, $N_{eps}(x) = \{y \in X | dist(x,y) \leq eps\}$, where $dist(x,y)$ is the distance function. A point $x \in X$ is referred to as a *core point* if its eps-neighborhood contains at least a minimum number of points ($minpts$), i.e., $|N_{eps}(x)| \geq minpts$. A point $y \in X$ is *directly density-reachable* from $x \in X$ if y is within the eps-neighborhood of x and x is a core point. A point $y \in X$ is *density-reachable* from $x \in X$ if there is a chain of points $x_1, x_2, \ldots, x_n$, with $x_1 = x$, $x_n = y$ such that x_{i+1} is directly density-reachable from x_i for all $1 \leq i < n$, $x_i \in X$. A point $y \in X$ is *density-connected* to $x \in X$ if there is a point $z \in X$ such that both x and y are density-reachable from z. A point $x \in X$ is a *border point* if it is not a core point, but is density reachable from at least one other core point. A *cluster* C discovered by DBSCAN is a non-empty subset of X satisfying the following two conditions (conditions 2.1 and 2.2).

Condition 2.1 (Maximality): For all pairs $(x,y) \in X$, if $x \in C$ and y is a core point that is density-reachable from x, then $y \in C$. If y is a border point then y is in exactly one C' such that $x \in C'$ and y is density-reachable from x.

Condition 2.2 (Connectivity): For all pairs $(x,y) \in C$, x is density-connected to y in X.

Condition 2.3 (Noise): A point $x \in X$ is a *noise* point if x is not directly density-reachable from any core point.

Note that we have extended the original definition of maximality since a border point can be density-reachable from more than one cluster.

The pseudocode of the DBSCAN algorithm is given in Algorithm 1. The algorithm starts with an arbitrary point $x \in X$ and retrieves its eps-neighborhood (Line 4). If the eps-neighborhood contains at least $minpts$ points, the procedure yields a new cluster, C. The algorithm then retrieves all points in X, which are density reachable from x and adds them to the cluster C (Line 8-17). If the eps-neighborhood of x has less than $minpts$, then x is marked as noise (Line 6). However, x could still be added to a cluster if it is identified as a border point while exploring other core points (Line 16-17).

Algorithm 1 The DBSCAN algorithm. Input: A set of points X, distance threshold eps, and the minimum number of points required to form a cluster, $minpts$. Output: A set of clusters.

```
1: procedure DBSCAN($X, eps, minpts$)
2:     for each unvisited point $x \in X$ do
3:         mark $x$ as visited
4:         $N \leftarrow$ GETNEIGHBORS($x$, $eps$)
5:         if $|N| < minpts$ then
6:             mark $x$ as noise
7:         else
8:             $C \leftarrow \{x\}$
9:             for each point $x' \in N$ do
10:                $N \leftarrow N \setminus x'$
11:                if $x'$ is not visited then
12:                    mark $x'$ as visited
13:                    $N' \leftarrow$ GETNEIGHBORS($x'$, $eps$)
14:                    if $|N'| \geq minpts$ then
15:                        $N \leftarrow N \cup N'$
16:                if $x'$ is not yet member of any cluster then
17:                    $C \leftarrow C \cup \{x'\}$
```

The retrieval of the eps-neighborhood of a point (Line 4 and Line 13, the GETNEIGHBORS function) is known as a *region-query* and the retrieval of all the density-reachable points from a core point in Lines 8 through 17 is known as *region-growing*. Note that a cluster can be identified uniquely by starting with any core point of the cluster [20]. The computational complexity of Algorithm 1 is $O(n^2)$, where n is the number of points in X. But, if spatial indexing (for example, using a *kd-tree* [24] or an *R*-tree* [25]) is used for serving the region-queries (GETNEIGHBORS functions), the complexity reduces to $O(n \log n)$ [26].

However, DBSCAN has a few limitations. First, although clustering can start with any core point, the process of region growing for a core point is inherently sequential. Given a core point x, the density reachable points from x are retrieved in a breadth-first search (BFS) manner. The neighbors at depth one (eps-neighbors of core point x) are explored and added to C. In the next step, the neighbors at depth two (neighbors of neighbors) are added and the process continues until the whole cluster is explored. Note that any points at higher depth cannot be explored until the lower depth points are exhausted. This limitation can be an obstacle for parallelizing the DBSCAN algorithm. One can also view the region growing in a depth-first-search (DFS) manner, but it still suffers from the same limitation. Secondly, during region growing when the eps-neighborhood N' of a new core point x' is retrieved (Line 13), N' is merged with the existing neighbor set N (Line 15), which takes linear time with respect to $|N'|$ as each point in N' is moved to N.

III. A New DBSCAN Algorithm

Our new DBSCAN algorithm exploits the similarities between the region growing and computing connected components in a graph. The algorithm initially creates a singleton tree for each point of the dataset. It then keeps merging those trees which belong to the same cluster until it has discovered all the clusters. The algorithm thus generates a single tree for each cluster containing all points of the cluster. To break the inherent data access order and to perform the merging efficiently, we use the disjoint-set data structure.

A. The Disjoint-Set Data Structure

The disjoint-set data structure defines a mechanism to maintain a dynamic collection of non-overlapping sets [21], [27]. It comes with two main operations: FIND and UNION. The FIND operation determines to which set a given element belongs, while the UNION operation joins two existing sets [22], [28].

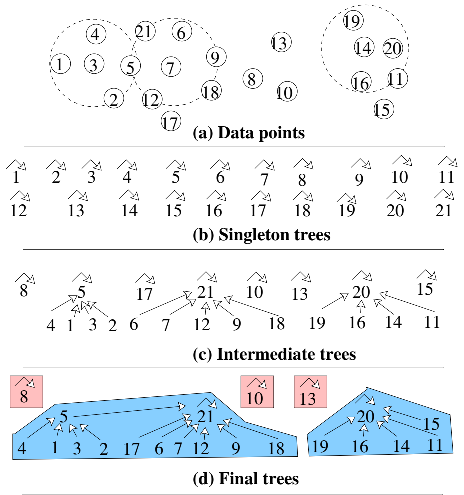


Figure 1. An example showing the proposed DSDBSCAN algorithm at different stages. (a) Sample data points with the circles denoting eps and $minpts = 4$. (b) The singleton trees when the algorithm starts. (c) Intermediate trees after exploring the eps-neighborhood of three randomly selected points 3, 7, and 14. (d) The resulting trees when the algorithm terminates where the singleton trees (8, 10, and 13) are noise points and the two blue colored large trees (rooted at 20 and 21) are clusters.

Each set is identified by a representative x, which is usually some member of the set. The underlying data structure of each set is typically a rooted tree represented by a parent pointer $p(x)$ for each element $x \in X$; the tree root satisfies $p(x) = x$ and is the representative of the set. Then, creating a new set for each element x is achieved by setting $p(x)$ to x.

The output of the FIND(x) operation is the root of the tree obtained by following the path of parent pointers (also known as the *find-path*) from x up to the root of x's tree. UNION(x, y) merges two trees containing x and y by changing the parent pointer of one root to the other one. To do this, the UNION(x, y) operation first calls two FIND operations, FIND(x) and FIND(y). If they return the same root (x and y belong to same set), no merging is required. But if the returned roots are different, say r_x and r_y, then the UNION operation sets $p(r_x) = r_y$ or $p(r_y) = r_x$. Note that this definition of the UNION operation is slightly different from its standard definition which requires that x and y belong to two different sets before calling UNION. We do this for ease of presentation.

There exists many different techniques to perform the UNION and FIND operations [21]. For example, it is possible to reduce the height of a tree during a FIND operation so that subsequent FIND operations run faster. In this paper, we have used the empirically best known UNION technique, known as Rem's algorithm (a lower indexed root points to a higher indexed root) with the splicing compression technique. Details on these can be found in [22].

B. The Disjoint-Set based DBSCAN Algorithm

The pseudocode of the disjoint-set data structure based DBSCAN algorithm (DSDBSCAN) is given in Algorithm 2. We will refer to the example in Figure 1 while presenting the algorithm. Initially DSDBSCAN creates a new set for each point $x \in X$ by setting its parent pointer to point to itself (Line 2-3 in Algorithm 2; Figure 1(b)). Then, for each point $x \in X$, the algorithm does the following: Similar to DBSCAN, it first computes x's eps-neighborhood (Line 5). If the number of neighbors is at least $minpts$, then x is marked as a core point (Line 7). In this case, for each eps-neighbor x' of x, we merge the trees containing x and x' depending on the following two conditions. (i) If x' is a core point, then x and x' are density-connected and therefore they should belong to the same tree (cluster). The algorithm performs the merging of the two trees containing x and x' using a UNION operation as discussed above. (ii) If x' is not a core point, then it is a border point as x' is directly density reachable from x. Therefore, if x' has not already been added to another cluster as a border point (one border point can not belong to multiple clusters), x' must be added to the cluster to which x belongs. This is done using a UNION operation on the two trees containing x and x' (the tree containing x' must be a singleton tree in this case). If x' has already been added to another cluster (implying that it is a border point of another core point, say z), we continue to the next step of the algorithm. The algorithm terminates when the eps-neighborhood of all the points have been explored. Figure 1(c) shows the intermediate trees after exploring the eps-neighborhood of points 3, 7, and 14. The final result in Figure 1(d) contains three noise points (singleton trees 8, 10, and 13) and two clusters (two blue colored large trees rooted at 20 and 21, each representing a cluster).

Algorithm 2 The disjoint-set data structure based DBSCAN Algorithm (DSDBSCAN). Input: A set of points X, distance eps, and the minimum number of points required to form a cluster, $minpts$. Output: A set of clusters.

```
1:  procedure DSDBSCAN($X, eps, minpts$)
2:      for each point $x \in X$ do
3:          $p(x) \leftarrow x$
4:      for each point $x \in X$ do
5:          $N \leftarrow$ GETNEIGHBORS($x, eps$)
6:          if $|N| \geq minpts$ then
7:              mark $x$ as core point
8:              for each point $x' \in N$ do
9:                  if $x'$ is a core point then
10:                     UNION($x, x'$)
11:                 else if $x'$ is not yet member of any cluster then
12:                     mark $x'$ as member of a cluster
13:                     UNION($x, x'$)
```

It is worthwhile to mention that DSDBSCAN adds (merges) points to its clusters (trees) without any specific ordering, which allows for a highly parallel implementation as discussed in the next section. The time complexity of DSDBSCAN (Algorithm 2) is $O(n \log n)$, which is exactly the same as DBSCAN

(Algorithm 1). We now show that DSDBSCAN satisfies the same conditions as DBSCAN, thereby proving the correctness of DSDBSCAN.

Theorem 3.1: The solution given by DSDBSCAN satisfies the following three conditions: Maximality, Connectivity, and Noise.

Proof: **(i)** We start with maximality, which is defined in Condition 2.1: For any two points $(x, y) \in X$, if $x \in C$ and if y is a core point density-reachable from x, then $y \in C$. We prove this by contradiction. Let us assume that $\exists x \in C$ and y is a core point density-reachable from x, but $y \notin C$. Then, x and y must be in different trees (as the same tree means they are in the same cluster) from the DSDBSCAN perspective. DSDBSCAN ensures that neighboring core points belong to the same tree (Line 9-10). Thus, there must exist a series of neighboring core points $x = x_0, x_1, \ldots, x_k = y$. Since DBSCAN will execute UNION(x_i, x_{i+1}) for each $0 \leq i < k$, it follows that if x and y are not in the same tree, the series of core points from y to x doesn't exist, which contradicts the assumption. Therefore, x and y are in the same tree. If y is a border point then let S be the set containing each core point x such that y is directly density reachable from x . Then it follows that y will be put in the same cluster as the first core point in S that is explored by the algorithm.
(ii) We now prove the connectivity condition. As defined by Condition 2.2, for any pair of points $(x, y) \in C$, x must be density-connected to y. With respect to DSDBSCAN, this means for all pairs (x, y) in the same tree, x is density-connected to y in X. We prove this by induction on the number of points in C at any given time during the execution of the algorithm. Let UNION(x,y) be the last operation performed on C that resulted in an increase in the size of C, and let $x \in C_1$ and $y \in C_2$ be the two trees that x and y belonged to prior to this operation. Then it follows that x is a core point. If $|C| = 2$ then x and y are density-connected since both x and y are density-reachable from x. Assume that $|C| = k > 2$ and that the proposition is true for any set of point smaller than k. Then it is true for both C_1 and C_2. It follows that for every $v \in C_1$ there is a point $z \in V_1$ such that both v and x are density-reachable from z. Since x is a core point y is density-reachable from z in C. Thus if $|C_2| = 1$ the result follows immediately. If $|C_2| > 1$ then y must also be a core point. For any point $v' \in C_2$ there is a point $z' \in V_2$ such that both y and v' are density reachable from z'. Thus there exists a path from z' to y consisting only of core points in C_2. It follows that z' is also density-reachable from y and thus any point in C_2 is density-reachable from y and also from z in C.
(iii) The noise condition (Condition 2.3) says that if a point $x \in X$ is a noise point, it should not belong to any cluster. In terms of DSDBSCAN this means x should belong to a singleton tree when the algorithm terminates. We prove this by contradiction as well. Let x be a noise point that belongs to a tree having more points than x. If this is the case, then x is density-reachable from at least one point of the tree (otherwise the algorithm would not have merged the trees). This implies that x is not a noise point, which contradicts with the assumption. Therefore, x belongs to a singleton tree if and only if it is a noise point. Thus, DSDBSCAN satisfies all three conditions similar to the DBSCAN algorithm. ■

IV. THE PARALLEL DBSCAN ALGORITHM

As discussed above, the use of the disjoint-set data structure in density based clustering works as a primary tool in breaking the access order of points while computing the clusters. In this section we present our disjoint-set based parallel DBSCAN (PDSDBSCAN) algorithm. The key idea of the algorithm is that each process core first runs a sequential DSDBSCAN algorithm on its local data points to compute local clusters (trees) in parallel without requiring any communication. After this we merge the local clusters (trees) to obtain the final clusters. This is also performed in parallel as opposed to the previous master-slave approaches where the master performs the merging sequentially. Moreover, the merging of two trees only requires changing the parent pointer of one root to the other one. This should be contrasted to the existing algorithms where the master traverses the entire cluster when it relabels it. Since the entire computation is performed in parallel, substantial scalability and speedup have been obtained. Similar ideas used in the setting of graph coloring and computing connected components have been presented in [23], [29], [30].

Algorithm 3 The parallel DBSCAN algorithm on a shared memory computer (PDSDBSCAN-S) using p threads. Input: A set of points X, distance eps, and the minimum number of points required to form a cluster, $minpts$. Let X be divided into p equal disjoint partitions $\{X_1, X_2, \ldots, X_p\}$, each assigned to one of the p running threads. For each thread t, Y_t denotes a set of pairs of points (x, x') such that $x \in X_t$ and $x' \notin X_t$. Output: A set of clusters.

```
1:  procedure PDSDBSCAN-S($X, eps, minpts$)
2:      for $t = 1$ to $p$ in parallel do        ▷ Stage: Local comp. (Line 2-18)
3:          for each point $x \in X_t$ do
4:              $p(x) \leftarrow x$
5:          $Y_t \leftarrow \emptyset$
6:          for each point $x \in X_t$ do
7:              $N \leftarrow$ GETNEIGHBORS($x$, $eps$)
8:              if $|N| \geq minpts$ then
9:                  mark $x$ as a core point
10:                 for each point $x' \in N$ do
11:                     if $x' \in X_t$ then
12:                         if $x'$ is a core point then
13:                             UNION($x$, $x'$)
14:                         else if $x' \notin$ any cluster then
15:                             mark $x'$ as member of a cluster
16:                             UNION($x$, $x'$)
17:                     else
18:                         $Y_t \leftarrow Y_t \cup \{(x, x')\}$
19:     for $t = 1$ to $p$ in parallel do        ▷ Stage: Merging (Line 19-25)
20:         for each $(x, x') \in Y_t$ do
21:             if $x'$ is a core point then
22:                 UNIONUSINGLOCK($x$, $x'$)
23:             else if $x' \notin$ any cluster then        ▷ Line 23-24 are atomic
24:                 mark $x'$ as member of a cluster
25:                 UNIONUSINGLOCK($x$, $x'$)
```

A. Parallel DBSCAN on Shared Memory Computers

The details of PDSDBSCAN on shared memory parallel computers (denoted by PDSDBSCAN-S) are given in Algorithm 3. The data points X are divided into p partitions $\{X_1, X_2, \ldots, X_p\}$ (one for each of the p threads running in parallel) and each thread t owns partition X_t. We divide the algorithm into two segments, *local computation* (Line 2-18) and *merging* (Line 19-25). Both steps run in parallel. Local computation is similar to sequential DSDBSCAN except each thread t identifies clusters using only its own data points X_t instead of X. During the computation using a point $x \in X_t$ by thread t, if x is identified as a core point and x' falls within the eps-neighborhood of x, we need to merge the trees containing x and x'. If t owns x', that is, $x' \in X_t$, then we merge the trees immediately (Line 11-16) similar to the DSDBSCAN algorithm. But if t does not own x', $x' \notin X_t$, (note that the GETNEIGHBORS function returns both local and non-local points as all points are in the commonly accessible shared memory), the merging is postponed and resolved in the ensuing merging step. To do this, the pair x and x' is added to the set Y_t (Line 18), which is initially set to empty (Line 5). The only non-local data access by any thread is the reading to obtain the neighbors using the GETNEIGHBORS function. Therefore, no explicit communication between threads or locking of points is required during the local computation. The merging step (Line 19-25) also runs in parallel. For each pair $(x, x') \in Y_t$, if x' is a core point or has not been added to any cluster yet, the trees containing x and x' are merged by a UNION operation. This implicitly sets x and x' to belong to the same cluster. Since this could cause a thread to change the parent pointer of a point owned by another thread, we use locks to protect the parent pointers, similar to what was done in [23].

Algorithm 4 Merging the trees containing x and x' with UNION using lock [23].

```
1:  procedure UNIONUSINGLOCK($x, x'$)
2:      while $p(x) \neq p(x')$ do
3:          if $p(x) < p(y)$ then
4:              if $x = p(x)$ then
5:                  LOCK($p(x)$)
6:                  if $x = p(x)$ then
7:                      $p(x) \leftarrow p(x')$
8:                  UNLOCK($p(x)$)
9:              $x = p(x)$
10:         else
11:             if $x' = p(x')$ then
12:                 LOCK($p(x')$)
13:                 if $x' = p(x')$ then
14:                     $p(x') \leftarrow p(x)$
15:                 UNLOCK($p(x')$)
16:             $x' = p(x')$
```

The idea behind the parallel UNION operation on a shared memory computer is that the algorithm uses a separate lock for each point. A thread wishing to set the parent pointer of a root r_1 to r_2 during a UNION operation would then have to acquire r_1's lock before doing so. Therefore, to perform a UNION operation, a thread will first attempt to acquire the necessary lock. Once this is achieved, the thread will test if r_1 is still a root. If this is the case, the thread will set the parent pointer of r_1 to r_2 and release the lock. On the other hand if some other thread has altered the parent pointer of r_1 so that the point is no longer a root, the thread will release the lock and continue executing the algorithm from its current position. The pseudocode of UNIONUSINGLOCK is given in Algorithm 4. More details on parallel UNION using locks can be found in [23].

Although locks are used during the merging (Line 22 and 25 in Algorithm 3), one thread has to wait for another thread only when both of them require the lock for the same point at the same time. In our experiments, this did not happen very often. We also noticed that only a few pairs of points require parallel merging of trees (and thus eventually the use of locks) during the UNION operation. In most cases, the points in the pairs belong to the same trees after only a few UNION operations and therefore no lock is needed because no merging of trees is performed in UNIONUSINGLOCK (see Algorithm 4). Moreover, since multiple threads can lock different non-shared points at the same time, multiple UNION operations using locks can be performed in parallel.

Due to space considerations we only outline the proof that PDSDBSCAN-S satisfies the same conditions as DBSCAN and DSDBSCAN. First, we note that it is sufficient to show that the parallel algorithm will perform exactly the same set of operations as DSDBSCAN irrespective of the order in which these are performed. The local computation stage in PDSDBSCAN-S will perform exactly the same operations on local data that DSDBSCAN would also have performed. During this stage PDSDBSCAN-S will also discover any operation that DSDBSCAN would have performed on two data points from different partitions. These operations are then performed in the subsequent merge stage. Finally, the correctness of the parallel merge stage follows from the correctness of the UNION-FIND algorithm presented in [23].

B. Parallel DBSCAN on Distributed Memory Computers

The details of parallel DBSCAN on distributed memory parallel computers (denoted by PDSDBSCAN-D) are given in Algorithm 5. Similar to PDSDBSCAN-S and traditional parallel algorithms, we assume that the data points X has been equally partitioned into p partitions $\{X_1, X_2, \ldots, X_p\}$ (one for each processor) and each processor t owns X_t only. From the perspective of processor t, each $x \in X_t$ is a *local point*, other points not in X_t are referred to as *remote points*. Since the memory is distributed, any other partition $X_i \neq X_t$, $1 \leq i \leq p$ is not visible to processor t (in contrast to PDSDBSCAN-S which uses a global shared memory). We therefore need the GETLOCALNEIGHBORS (Line 5) and GETREMOTENEIGHBORS (Line 6) functions to get the local and remote points, respectively. Note that retrieving the remote points requires communication with other processors. Instead of calling GETREMOTENEIGHBORS for each local point during the computation, we take advantage of the eps parameter and gather all possible remote neighbors in one step before we start the algorithm. In the DBSCAN algorithm, for any given point x, we are only interested in the neighbors that falls within the eps distance of x. Therefore, we extend the bounding box of X_t by a distance of eps in every direction in each dimension and query other processors with the extended bounding box to return their local points that falls within it. Thus, each processor t has a copy of the remote points X'_t that it requires for its computation. We consider this step as a preprocessing step (named *gather-neighbors*). Our experiments show that gather-neighbors takes only a fraction of the total time compared to PDSDBSCAN-D. Thus, the GETREMOTENEIGHBORS function returns the remote points from the local copy, X'_t without communication.

Algorithm 5 The parallel DBSCAN algorithm on a distributed memory computer (PDSDBSCAN-D) using p processors. Input: A set of points X, distance eps, and the minimum number of points required to form a cluster, $minpts$. Let X be divided into p equal disjoint partitions $\{X_1, X_2, \ldots, X_p\}$ for the p running processors. Each processor t also has a set of remote points, X'_t stored locally to avoid communication. Each processor t runs PDSDBSCAN-D to compute the clusters. Output: A set of clusters.

```
1: procedure PDSDBSCAN-D($X, eps, minpts$)
2:     for each point $x \in X_t$ do                ▷ Stage: Local comp. (Line 2-16)
3:         $p(x) \leftarrow x$
4:     for each point $x \in X_t$ do
5:         $N \leftarrow$ GETLOCALNEIGHBORS($x$, $eps$, $X_t$)
6:         $N' \leftarrow$ GETREMOTENEIGHBORS($x$, $eps$, $X'_t$)
7:         if $|N| + |N'| \geq minpts$ then
8:             mark $x$ as a core point
9:             for each point $y \in N$ do
10:                if $y$ is a core point then
11:                    UNION($x$, $y$)
12:                else if $y \notin$ any cluster then
13:                    mark $y$ as member of a cluster
14:                    UNION($x$, $y$)
15:            for each point $y' \in N'$ do
16:                $Y_t \leftarrow Y_t \cup$ UNIONQUERY$(x, P_x, y')$   ▷ $P_x = P_t$
17:    Send UNIONQUERY$(x, P_x, y') \in Y_t$ to $P_{y'}$   ▷ Stage: Merging (L 17-28)
18:    Receive UNIONQUERY from other processors
19:    for each received UNIONQUERY ($x$, $P_x$, $y'$) do
20:        if $y'$ is a core point then
21:            PARALLELUNION($x$, $P_x$, $y'$)
22:        else if $y' \notin$ any cluster then
23:            mark $y'$ as member of a cluster
24:            PARALLELUNION($x$, $P_x$, $y'$)
25:    while (Any processor has any UNIONQUERY) do
26:        Receive UNIONQUERY from other processors
27:        for each received UNIONQUERY ($x$, $P_x$, $y'$) do
28:            PARALLELUNION($x$, $P_x$, $y'$)
```

We divide PDSDBSCAN-D into two segments, *local computation* (Line 2-16) and merging (17-28), similar to PDSDBSCAN-S except each processor now needs to take special measure getting the neighbors that belong to other processors (as discussed above in the gather-neighbors step). The parallel UNION operation is now performed using message passing between processors. During the local computation, we compute the local neighbors, N (Line 5) and remote neighbors, N' (Line 6) for each point x. If x is a core point (when $|N| + |N'| \geq minpts$), we perform a UNION operation on x and each local point $y \in N$ similar to the PDSDBSCAN-S algorithm and for each remote point $y' \in N'$, we send a UNIONQUERY to processor $P_{y'}$ (the owner of y') asking to perform a UNION operation of the tree containing x and y', if possible.

In the merging stage (17-28), we have two sub-steps, *merging-decision* (Line 17-24) and *propagate* (Line 25-28). During merging-decision, for each received UNIONQUERY(x, P_x, y'), we check whether y' is a core point or if it has been added to any other cluster yet (similar to the merging stage in PDSDBSCAN-S). If we do not need to UNION the trees containing x and y', we continue to the next UNIONQUERY. Otherwise, we call PARALLELUNION(x, P_x, y') to perform a UNION of the trees containing x and y' in the distributed-memory architecture. For this we use similar techniques as presented in [22] and [30].

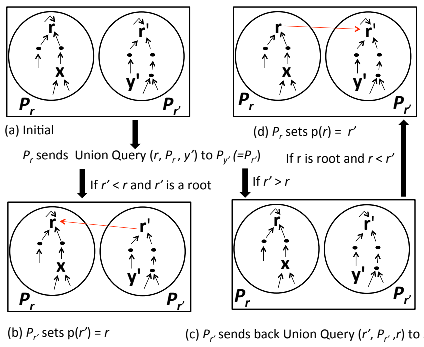
Algorithm 6 Merging trees containing x and y' on $P_{y'}$ [30]

```
1: procedure PARALLELUNION($x$, $P_x$, $y'$)
2:     $r$ = FIND($y'$)
3:     if $p(r) = r$ then        ▷ $r$ is a global root at $P_t = P_r = P_{y'}$
4:         if $r < x$ then
5:             $p(r) = x$
6:         else
7:             Send UNIONQUERY$(r, P_r, x)$ to $P_x$
8:     else    ▷ $r$ is the last point on $P_t$ on the find path towards the global root
9:         if $p(r) < x$ then
10:            Send UNIONQUERY$(x, P_x, p(r))$ to $P_{p(r)}$
11:        else
12:            Send UNIONQUERY$(p(r), P_{p(r)}, x)$ to $P_x$
```

The details of PARALLELUNION are given in Algorithm 6. During PARALLELUNION(x, P_x, y'), $P_{y'}$ (which is always the owner of y') calls the FIND operation to get the root of the tree containing y'. As the trees might span among the processors and the memory is distributed, $P_{y'}$ may not able to access the root (we use the term *global root*) if this belongs to other processors. In this case the FIND operation returns a *local root* (the boundary point on $P_{y'}$ on the find-path of y'). If $P_{y'}$ owns the global root, r and it satisfies the required criteria of the UNION operation (in our case the lower indexed root should point to a higher indexed point), we merge the trees by setting $p(r)$ to x (Line 5). Otherwise we need to send a UNIONQUERY based on one of the following two conditions: (i) r is a global root but $r > x$ (Line 7): Although P_r owns the global root r in this case, we can not set $p = x$ as it would violate the required criteria that a lowered indexed root point to a higher indexed one. We therefore send a UNIONQUERY to P_x to perform the UNION operation. (ii) r is a local root: We send a UNIONQUERY either to P_x in Line 12 (if $p(r) > x$) or $P_{p(r)}$ in Line 10 (otherwise) to reduce the search space. More details can be found in [30].

An example of the PARALLELUNION operation to merge the trees containing x and y' is shown in Figure 2. Initially P_r (the owner of x) sends a UNIONQUERY to $P_{r'}$ (the owner of y') to UNION the trees (Figure 2(a)). If $P_{r'}$ satisfies the required criteria (as discussed above), then the trees are merged by setting $p(r') = r$ on $P_{r'}$ as shown in Figure 2(b). But, if the criteria is not satisfied, then $P_{r'}$ sends back a UNIONQUERY to P_r (Figure 2(c)). Then P_r merges the trees by setting $p(r) = r'$ (Figure 2(d)). In the general case the UNIONQUERY might travel among several processors, but for simplicity we considered only two processors in this example.

PDSDBSCAN-D satisfies the same conditions: maximality, connectivity, and noise properties (see Theorem 3.1), similar



(a) Initial
P_r sends Union Query (r, P_r, y') to P_y' (=P_r')
If r' < r and r' is a root
If r' > r
If r is root and r < r'
(d) P_r sets p(r) = r'
(b) P_r' sets p(r') = r
(c) P_r' sends back Union Query (r', P_r', r) to P_r

Figure 2. An example of the processing of a PARALLELUNION(x, P_x, y') operation. (a) Initial condition of the trees on P_r and $P_{r'}$ containing x and y', respectively. (b) After the merging when $p(r')$ is set to r. (c) $P_{r'}$ sends a UNIONQUERY(r', $P_{r'}$, P_r) to P_r. (d) After the merging when $p(r)$ is set to r'. The tree spans multiple processors after the PARALLELUNION operation.

to DBSCAN, DSDBSCAN, and PDSDBSCAN-S. We omit the proof due to space considerations.

We performed several communication optimizations including the bundling of all the UNIONQUERY messages in each round of communication and the compression and decompression (Line 17 and 18 in Algorithm 5, respectively) of all the UNIONQUERY messages during the first round of communication. We again omit the details due to space considerations.

V. EXPERIMENTAL RESULTS

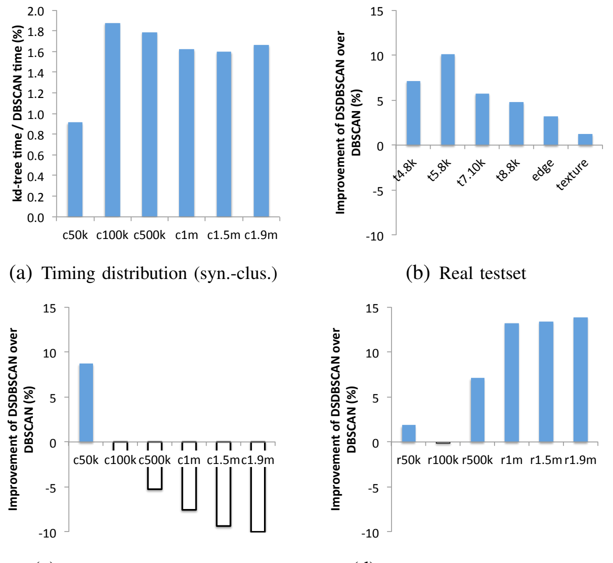
We first present the experimental setup used for both the sequential and the shared memory DBSCAN algorithms. The setup for the distributed memory algorithm is presented later.

For the experiments we used a Dell computer running GNU/Linux and equipped with four 2.00 GHz Intel Xeon E7-4850 processors with a total of 128 GB memory. Each processor has ten cores. Each of the 40 cores has 48 KB of L1 and 256 KB of L2 cache. Each processor (10 cores) shares a 24 MB L3 cache. All algorithms were implemented in C++ using OpenMP and compiled with gcc (version 4.6.3) using the -O3 flag.

Our testbed consists of 18 datasets, which are divided into three categories, each with six datasets. The first category, called *real*, have been collected from Chameleon (*t4.8k*, *t5.8k*, *t7.10k*, and *t8.8k*) [31] and CUCIS (*edge* and *texture*) [32]. The other two categories, *synthetic-random* and *synthetic-cluster*, have been generated synthetically using the IBM synthetic data generator [33], [34]. In the synthetic-random datasets (*r50k*, *r100k*, *r500k*, *r1m*, *r1.5m*, and *r1.9m*), points in each dataset have been generated uniformly at random. In synthetic-cluster datasets (*c50k*, *c100k*, *c500k*, *c1m*, *c1.5m*, and *c1.9m*), first a specific number of random points are taken as different clusters, points are then added randomly to these clusters. The testbed contains up to 1.9 million data points and each data point is a vector of up to 20 dimensions. Table I shows structural properties of the dataset. In the experiments, the two input parameters (eps and $minpts$) shown in the table have been chosen carefully to obtain a fair number of clusters and noise points in a reasonable time. Higher value of eps increases the time taken for the experiments while the number of clusters and noise points are reduced. Higher value of $minpts$ increases the noise counts.

Table I
STRUCTURAL PROPERTIES OF THE TESTBED (REAL, SYNTHETIC-CLUSTER, AND SYNTHETIC-RANDOM) AND THE RESULTING NUMBER OF CLUSTERS, NOISE, AND TIME TAKEN BY CLASSICAL DBSCAN AND PDSDBSCAN-S ALGORITHMS USING ONE PROCESS CORE FOR CAREFULLY SELECTED INPUT PARAMETERS eps AND $minpts$. d DENOTES THE DIMENSION OF EACH POINT.

					Time (sec.)			
Name	Points	d	eps	$minpts$	DBSCAN	PDSDBSCAN-S(t_1)	Clusters	Noise
$t4.8k$	8,000	2	10	20	0.04	0.04	15	278
$t5.8k$	8,000	2	8	21	0.05	0.04	15	886
$t7.10k$	10,000	2	10	12	0.04	0.04	9	692
$t8.8k$	8,000	2	10	10	0.03	0.03	23	459
$edge$	17,695	18	3	2	18.22	17.99	9	97
$texture$	17,695	20	3	2	18.80	18.93	47	1,443
$c50k$	50,000	10	25	5	3.19	2.97	51	3,086
$c100k$	100,000	10	25	5	5.94	6.62	103	6,077
$c500k$	500,000	10	25	5	31.51	34.30	512	36,095
$c1m$	1,000,000	10	25	5	66.14	73.93	1,025	64,525
$c1.5m$	1,500,000	10	25	5	100.25	114.73	1,545	102,394
$c1.9m$	1,900,000	10	25	5	126.68	144.87	1,959	135,451
$r50k$	50,000	10	100	4	4.60	4.55	1,748	34,352
$r100k$	100,000	10	100	4	11.90	12.26	740	23,003
$r500k$	500,000	10	90	5	165.55	158.73	161	25,143
$r1m$	1,000,000	10	75	5	451.99	405.00	312	45,614
$r1.5m$	1,500,000	10	65	10	768.54	694.57	968	257,204
$r1.9m$	1,900,000	10	65	10	1,077.84	958.68	987	278,788



(a) Timing distribution (syn.-clus.)
(b) Real testset
(c) Synthetic-cluster testset
(d) Synthetic-random testset

Figure 3. (a): Time taken by the construction of *kd-tree* and DBSCAN algorithm for synthetic-cluster datasets. (b)-(d): Performance comparison between DBSCAN and DSDBSCAN algorithms.

A. DBSCAN *vs.* DSDBSCAN

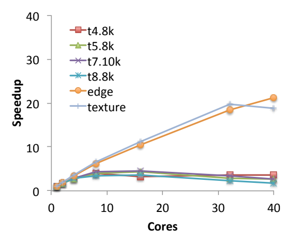
As discussed in Section II, to reduce the running time of the DBSCAN algorithms from $O(n^2)$ to $O(n \log n)$, spatial indexing (*kd-tree* [24] or *R*-tree* [25]) is commonly used [26]. In all of our implementations, we used *kd-trees* [24] and therefore obtain the reduced time complexities. Moreover, *kd-tree* gives a geometric partitioning of the data points, which we use to divide the data points equally among the cores in the parallel DBSCAN algorithm. However, there is an overhead in constructing the *kd-tree* before running the DBSCAN algorithms. Figure 3(a) shows a comparison of the time taken by the construction of the *kd-tree* over the DBSCAN algorithm in percent for the synthetic-cluster datasets. As can be seen, constructing the *kd-tree* takes only a fraction of the time (0.91% to 1.88%) taken by the DBSCAN algorithm. We found similar results for the synthetic-cluster datasets (0.23% to 1.23%). However, these ranges are higher (0.06% to 18.28%) for real dataset. This is because each real dataset consists of a small number of points, and therefore, the DBSCAN algorithms takes much less time compared to the other two categories. It should be noted that we have not parallelized the construction of the *kd-tree* in this paper, we therefore do not consider the timing of the construction of the *kd-tree* in the following discussion.

Figure 3(b), 3(c), and 3(d) compare the performance between the classical DBSCAN (Algorithm 1) and the sequential disjoint-set data structure based DBSCAN (DSDBSCAN, Algorithm 2). In each figure, the performance improvement of DSDBSCAN over the classical DBSCAN has been shown in percent. This is calculated by the equation [(time taken by DBSCAN - time taken by DSDBSCAN) * 100 / time taken by DBSCAN]. Therefore, any bar with a height greater than zero (light blue color) or less than zero (black and white color) means that DSDBSCAN is performing better or worse, respectively, compared to the classical DBSCAN algorithm. As can be seen, DSDBSCAN performs well on real (1.24% to 10.08%) and synthetic-random (-0.14% to 13.84%) datasets, whereas for synthetic-cluster datasets, the performance varies from -10% to 8.70%. We observe that the number of Union operations is significantly higher for synthetic-cluster datasets compared to the other two categories and this number increases with the size of the datasets (Figure 3(c)). The raw runtime numbers of the classical DBSCAN algorithm are listed in Table I.

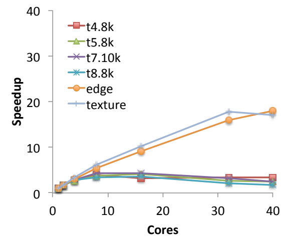
B. *Parallel* DBSCAN *on a Shared Memory Computer*

Figure 4 shows the speedup obtained by PDSDBSCAN-S (Algorithm 3) for various numbers of process cores (threads). The raw run-times taken by one process core in case of PDSDBSCAN-S (denoted by PDSDBSCAN-S(t_1)) and classical DBSCAN are provided in Table I. The one giving the smallest running time of these two has been used to compute the speedup. The left column in Figure 4 shows the speedup results considering only the local computation stage whereas the right column shows results using total time (local computation plus merging) for the three datasets. Clearly, the local computation stage scales well across all the datasets as there is no interaction between the threads. Since local computation takes substantially more time than the merging, the speedup behavior of just the local computation is nearly identical to that of the overall execution. Note that the speedup for some real datasets in Figure 4(a) and 4(b) saturate at eight process cores as they are relatively small compared to the other datasets.

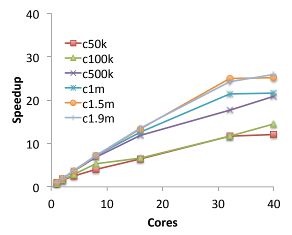
Figure 5(a) shows a comparison of the time taken by the merging stage over the local computation stage in percent for the PDSDBSCAN-S algorithm using dataset $c1.9m$ for various number of process cores. We observe that although the merging time increases with the number of process cores, this is still only a small fraction (less than 0.70%) of the local computation time (eventually the total time). We observe



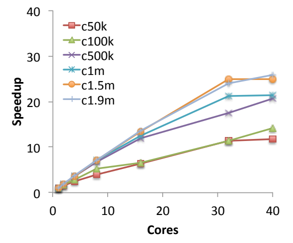
(a) Local comp. (real)



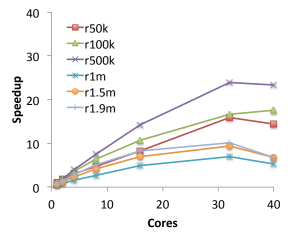
(b) Local comp. and merging (real)



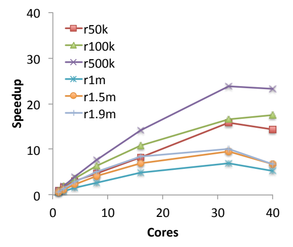
(c) Local comp. (syn.-clus.)



(d) Local comp. and merging (syn.-clus.)



(e) Local comp. (syn.-rand.)



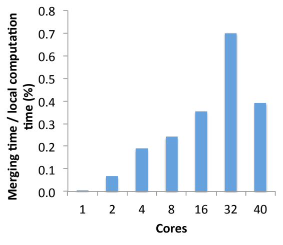
(f) Local comp. and merging (syn.-rand.)

Figure 4. Speedup of parallel DBSCAN (PDSDBSCAN-S). Left column: Local computation in PDSDBSCAN-S. Right column: Total time (local computation + merging) of PDSDBSCAN-S.

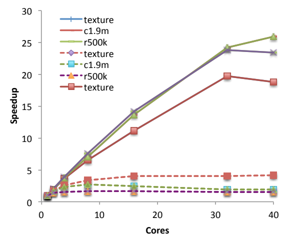
similar performance on the synthetic-cluster (less than 2.82%, on average 0.57%) and the synthetic-random datasets (less than 0.31%, on average 0.06%). However, this fraction is higher for real datasets (less than 8.16% with the exception on instance *edge*, where the value is less than 17.37% and on average 3.91%). As mentioned above, this happens since the real datasets are fairly small and therefore multiple threads are competing to lock only a limited number of points, which in turn increases the merging time.

As discussed in the previous section, although locks are used in the Union operation during the parallel merging stage, only a few pairs of points actually result in successful Union operations. In most cases, the points in the pairs already belong to the same tree and therefore no lock is needed. Our experimental results show that on average only 0.27% and 0.09% of the pairs for the real and the synthetic-cluster datasets, respectively, turn out to actually result in a successful Union operation. This is also true for the synthetic-random datasets (4.25%) except for two cases, $r50k$ and $r100k$, where the average value is 39.38%.

We have selected one dataset from each of the three categories ($texture$, $c1.9m$, and $r500k$ from real, synthetic-cluster, and synthetic-random, respectively) to present the maximum speedup obtained in our experiments. The speedups are plotted in Figure 5(b). As can be seen, we have been able to obtain a maximum speedup of 21.20, 25.97, and 23.86 for



(a) Timing distribution (c1p9m)



(b) Speedup of PDSDBSCAN-S

Figure 5. (a) Timing distribution for various number of threads ($c1.9m$). (b) The maximum speedup of PDSDBSCAN-S algorithm (solid lines) from each of the three test sets (real, synthetic-cluster, and synthetic-random) and corresponding speedup obtained using the master-slave based classical parallel DBSCAN algorithm (dashed lines).

$texture$, $c1.9m$, and $r500k$, respectively (as shown by the solid lines). However, the ranges of the maximum speedup are 3.63 to 21.20 (average 9.53) for real, 12.02 to 25.97 (average 20.01) for synthetic-cluster, and 7.01 to 23.86 (average 13.96) for synthetic-random datasets.

Finally, we have compared our parallel DBSCAN algorithm with the previous master-slave approaches [14], [15], [16], [17], [18], [19], [20]. As their source codes are not available, we have implemented their ideas, where the master process perform the cluster assignment while the slave processes answer the neighborhood queries [15], [17]. As can be seen in Figure 5(b) (the dashed lines), the master-slave approach performs reasonably well up to 8 cores (maximum speedup 3.35) and then, with the increment of the number of cores, speedup remains constant. The maximum speedup we obtained using the master-slave based parallel DBSCAN algorithm is 4.12 using 40 cores, which is significantly lower than the maximum speedup (25.97) we obtained using our disjoint-set based parallel DBSCAN algorithm. In Figure 5(b), we presented only three testsets (one from each category) as we noticed similar results for other testsets.

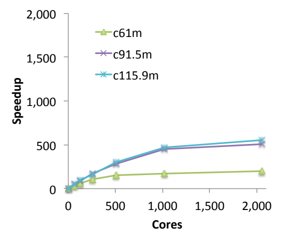
C. *Parallel* DBSCAN *on a Distributed Memory Computer*

To perform the experiment for PDSDBSCAN-D, we use Hopper, a Cray XE6 distributed memory parallel computer where each node has two twelve-core AMD 'MagnyCours' 2.1-GHz processors and shares 32 GB of memory. Each core has its own 64 KB L1 and 512 KB L2 caches. Each six cores on the MagnyCours processor share one 6 MB of L3 cache. The algorithms have been implemented in C/C++ using the MPI message-passing library and has been compiled with gcc (4.6.2) and -O3 optimization level.

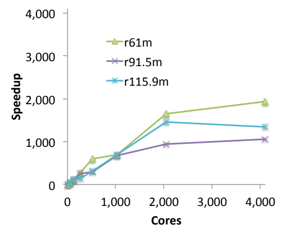
The datasets used in the previous experiments are relatively small for massively parallel computing. We therefore consider a different testbed of 10 data sets, which are again divided into three categories, each with three, three, and four datasets, respectively. The first two categories, called *synthetic-cluster-extended* (*c61m*, *c91.5m*, and *c115.9m*) and *synthetic-random-extended* (*r61m*, *r91.5m*, and *r115.9m*), have been generated synthetically using the IBM synthetic data generator [33], [34]. As the generator is limited to generate at most 2 million high dimensional points, we replicate the same data towards the left and right three times (separating each dataset with a reasonable distance) in each dimension to get datasets with hundreds of million of points. The third category, called *millennium-run-simulation* consists of four datasets from the database [35], [36] on Millennium Run, the largest simulation of the formation of structure with the ΛCDM cosmogony with a factor of 10^{10} particles. The four datasets, MPAGalaxiesBertone2007 (mb) [37], MPAGalaxiesDeLucia2006a (md) [38], DGalaxiesBower2006a (db) [39], and MPAHaloTreesMhalo (mm) [37] are taken from the Galaxy and Halo databases (as the name specified). To be consistent with the size of the other two categories we have randomly selected 10% of the points from these datasets. However, since the dimension of each dataset is high, we are eventually considering almost billions of floating point numbers. Table II shows the structural properties of the datasets and related input parameters. To perform the experiments, we use a parallel kd-tree representation as presented in [40] to geometrically partition the data among the processors. However, we do not include the partitioning time while computing the speedup by the PDSDBSCAN-D.

Table II
STRUCTURAL PROPERTIES OF THE TESTBED (SYNTHETIC-CLUSTER-EXTENDED, SYNTHETIC-RANDOM-EXTENDED, AND MILLENNIUM-RUN-SIMULATION) AND THE TIME TAKEN BY PDSDBSCAN-D USING ONE PROCESS CORE.

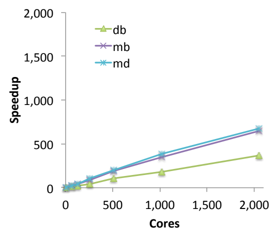
Name	Points	d	eps	$minpts$	Time(sec.)
$c61m$	61,000,000	10	25	5	4,121.75
$c91.5m$	91,500,000	10	25	5	5,738.29
$c115.9m$	115,900,000	10	25	5	8,511.87
$r61m$	61,000,000	10	25	2	1,112.94
$r91.5m$	91,500,000	10	25	2	2,991.38
$r115.9m$	115,900,000	10	25	2	4,121.75
DGalaxiesBower2006a (db)	96,446,861	8	5	3	219.33
MPAHaloTreesMhalo (mm)	72,322,888	9	5	3	806.56
MPAGalaxiesBertone2007 (mb)	100,446,132	8	5	3	240.17
MPAGalaxiesDeLucia2006a (md)	100,446,132	8	5	3	248.53



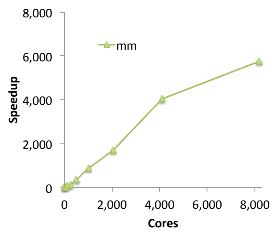
(a) Synthetic-cluster-extended dataset



(b) Synthetic-random-extended dataset



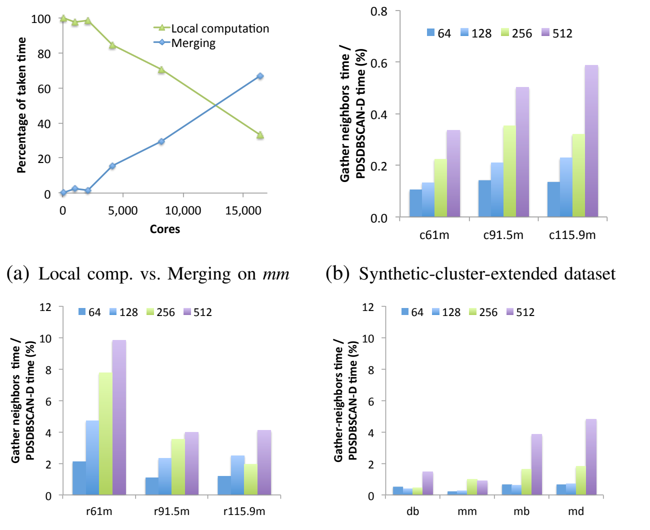
(c) Millennium-run-simulation dataset



(d) Millennium-run-simulation dataset

Figure 6. Speedup of PDSDBSCAN-D on Hopper at NERSC, a CRAY XE6 distributed memory computer, on three different categories of datasets.

Figure 6(a), 6(b), and 6(c) show the speedup obtained by PDSDBSCAN-D algorithm using synthetic-cluster-extended, synthetic-random-extended, and millennium-simulation-run datasets (except *mm*), respectively. We use a maximum of 4,096 process cores for these datasets as the speedup de-



(a) Local comp. vs. Merging on *mm* (b) Synthetic-cluster-extended dataset

(c) Synthetic-random-extended dataset (d) Millennium-run-simulation dataset

Figure 7. (a) Trade-off between local computation and merging w.r.t the number of processors on *mm*, a millennium-run-simulation dataset. (b)-(d) Time taken by the preprocessing step, *gather-neighbors*, compared to the total time taken by PDSDBSCAN-D using 64, 128, 256, and 512 process cores.

creases on larger number of process cores. As can be seen, the speedups on synthetic-cluster-extended and millennium-simulation-run (*db, mb, md*) datasets (Figure 6(a) and 6(c), respectively) are significantly lower than synthetic-random-extended datasets (Figure 6(b)). We observed that the number of UNION operations and the number of local neighbors on synthetic-cluster-extended and millennium-simulation-run (*db, mb, md*) datasets are significantly higher than the synthetic-random-extended dataset. However, on the dataset *mm* in millennium-simulation-run (Figure 6(d)), we get a speedup of 5,765 using 8,192 process cores.

Figure 7(a) shows the trade-off between the local computation and the merging stage by comparing them with the total time (local computation time + merging time) in percent. We use *mm*, the millennium-run-simulation dataset for this purpose and continue up to 16,384 process cores to understand the behavior clearly. As can be seen, the communication time increases while the computation time decreases with the number of processors. When using more than 10,000 process cores, communication time starts dominating the computation time and therefore, the speedup starts decreasing. For example, we achieved a speedup of 5,765 using 8,192 process cores whereas the speedup is 5,124 using 16,384 process cores. We observe similar behaviors for other datasets.

Figure 7(b), 7(c), and 7(d) show a comparison of the time taken by the gather-neighbors preprocessing step over the total time taken by PDSDBSCAN-D in percent on all datasets using 64, 128, 256, and 512 process cores. As can be seen, the gather-neighbors step adds an overhead of maximum 0.59% (minimum 0.10% and average 0.27%) of the total time on synthetic-cluster-extended datasets. Similar results are found on millennium-simulation-run datasets (maximum 4.82%, minimum 0.21%, and average 1.25%). However, these numbers are relatively higher (maximum 9.82%, minimum 1.01%, and average 3.76%) for synthetic-random-extended datasets as the points are uniformly distributed in the space and therefore the number of points gathered in each processor is higher compared to the other two test sets. It is also to be noted that these values increase with the number of processors and also with the *eps* parameter as the overlapping region among the processors is proportional to the number of processors. We observe that on 64 process cores the memory space taken by the remote points in each processor is on average 0.68 times, 1.57 times, and 1.02 times on synthetic-cluster-extended, synthetic-random-extended, and millennium-simulation-run datasets, respectively, compared to the memory space taken by the local points. These values changes to 1.27 times, 2.94 times, and 3.18 times, respectively on 512 process cores. However, with this scheme the local-computation stage in PDSDBSCAN-D can perform the clustering without any communication overhead similar to PDSDBSCAN-S. The alternative would be to perform communication for each point to obtain its remote neighbors.

VI. CONCLUSION AND FUTURE WORK

In this study we have revisited the well-known density based clustering algorithm, DBSCAN. This algorithm is known to be challenging to parallelize as the computation involves an inherent data access order. We present a new parallel DBSCAN (PDSDBSCAN) algorithm based on the disjoint-set data structure. The use of this data structure works as a mechanism for increasing concurrency, which again leads to scalable performance. The algorithm uses a bottom-up approach to construct the clusters as a collection of hierarchical trees. This approach achieves a better-balanced work-load distribution. PDSDBSCAN is implemented using both OpenMP and MPI. Our experimental results conducted on a shared memory computer show scalable performance, achieving speedups up to a factor of 25.97 when using 40 cores on data sets containing several hundred million high-dimensional points. Similar scalability results have been obtained on a distributed-memory machine with a speedup of 5,765 using 8,192 process cores. Our experiments also show that PDSDBSCAN significantly outperforms existing parallel DBSCAN algorithms. We intend to conduct further studies to provide more extensive results on much larger number of cores with datasets from different scientific domains. Finally, we note that our algorithm also seems to be suitable for other parallel architectures, such as GPU and heterogenous architectures.

ACKNOWLEDGMENT

This work is supported in part by NSF award numbers CCF-0621443, OCI-0724599, CCF-0833131, CNS-0830927, IIS-0905205, OCI-0956311, CCF-0938000, CCF-1043085, CCF-1029166, and OCI-1144061, and in part by DOE grants DE-FG02-08ER25848, DE-SC0001283, DE-SC0005309, DE-SC0005340, and DE-SC0007456. This research used Hopper Cray XE6 computer of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, “From data mining to knowledge discovery in databases,” *AI magazine*, vol. 17, no. 3, p. 37, 1996.

[2] J. MacQueen *et al.*, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1. USA, 1967, pp. 281–297.

[3] H. Park and C. Jun, “A simple and fast algorithm for K-medoids clustering,” *Expert Systems with Applications*, vol. 36, no. 2, pp. 3336–3341, 2009.

[4] T. Zhang, R. Ramakrishnan, and M. Livny, “BIRCH: an efficient data clustering method for very large databases,” in *ACM SIGMOD Record*, vol. 25(2). ACM, 1996, pp. 103–114.

[5] M. Ester, H. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in *Proc. of the 2nd KDD*, vol. 1996, 1996, pp. 226–231.

[6] W. Wang, J. Yang, and R. Muntz, “STING: A statistical information grid approach to spatial data mining,” in *Proc. of the International Conference on Very Large Data Bases*. IEEE, 1997, pp. 186–195.

[7] G. Sheikholeslami, S. Chatterjee, and A. Zhang, “WaveCluster: a wavelet-based clustering approach for spatial data in very large databases,” *The VLDB Journal*, vol. 8, no. 3, pp. 289–304, 2000.

[8] A. Mukhopadhyay and U. Maulik, “Unsupervised satellite image segmentation by combining SA based fuzzy clustering with support vector machine,” in *Proc. of 7th ICAPR’09*. IEEE, 2009, pp. 381–384.

[9] D. Birant and A. Kut, “ST-DBSCAN: An algorithm for clustering spatial-temporal data,” *Data & Knowledge Engineering*, vol. 60, no. 1, pp. 208–221, 2007.

[10] M. Surdeanu, J. Turmo, and A. Ageno, “A hybrid unsupervised approach for document clustering,” in *Proc. of the 11th ACM SIGKDD*. ACM, 2005, pp. 685–690.

[11] S. Madeira and A. Oliveira, “Biclustering algorithms for biological data analysis: a survey,” *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, vol. 1, no. 1, pp. 24–45, 2004.

[12] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques*. Morgan Kaufmann, 2011.

[13] H. Kargupta and J. Han, *Next generation of data mining*. Chapman & Hall/CRC, 2009, vol. 7.

[14] S. Brecheisen, H. Kriegel, and M. Pfeifle, “Parallel density-based clustering of complex objects,” *Advances in Knowledge Discovery and Data Mining*, pp. 179–188, 2006.

[15] D. Arlia and M. Coppola, “Experiments in parallel clustering with DBSCAN,” *Euro-Par 2001 Parallel Processing*, pp. 326–331, 2001.

[16] M. Chen, X. Gao, and H. Li, “Parallel DBSCAN with priority r-tree,” in *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*. IEEE, 2010, pp. 508–511.

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

[18] Y. Fu, W. Zhao, and H. Ma, “Research on parallel DBSCAN algorithm design based on mapreduce,” *Advanced Materials Research*, vol. 301, pp. 1133–1138, 2011.

[19] X. Xu, J. Jäger, and H. Kriegel, “A fast parallel clustering algorithm for large spatial databases,” *High Performance Data Mining*, pp. 263–290, 2002.

[20] A. Zhou, S. Zhou, J. Cao, Y. Fan, and Y. Hu, “Approaches for scaling DBSCAN algorithm to large spatial databases,” *Journal of computer science and technology*, vol. 15, no. 6, pp. 509–526, 2000.

[21] T. Cormen, *Introduction to algorithms*. The MIT press, 2001.

[22] M. Patwary, J. Blair, and F. Manne, “Experiments on union-find algorithms for the disjoint-set data structure,” in *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA 2010)*. Springer, LNCS 6049, 2010, pp. 411–423.

[23] M. M. A. Patwary, P. Refsnes, and F. Manne, “Multi-core spanning forest algorithms using the disjoint-set data structure,” in *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2012)*. IEEE, 2012, accepted and presented, to appear.

[24] M. B. Kennel, “KDTREE 2: Fortran 95 and C++ software to efficiently search for near neighbors in a multi-dimensional Euclidean space,” 2004, institute for Nonlinear Science, University of California.

[25] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, “The r*-tree: an efficient and robust access method for points and rectangles,” *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, vol. 19, no. 2, pp. 322–331, 1990.

[26] J. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[27] R. Tarjan, “A class of algorithms which require nonlinear time to maintain disjoint sets,” *Journal of computer and system sciences*, vol. 18, no. 2, pp. 110–127, 1979.

[28] B. Galler and M. Fisher, “An improved equivalence algorithm,” *Comm. of ACM*, vol. 7, pp. 301–303, 1964.

[29] M. Patwary, A. Gebremedhin, and A. Pothen, “New multithreaded ordering and coloring algorithms for multicore architectures,” in *Proceedings of 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011)*. Springer, LNCS 6853, 2011, pp. 250–262.

[30] F. Manne and M. Patwary, “A scalable parallel union-find algorithm for distributed memory computers,” *Parallel Processing and Applied Mathematics*, pp. 186–195, 2010.

[31] “CLUTO - clustering high-dimensional datasets,” 2006, http://glaros.dtc.umn.edu/gkhome/cluto/cluto/.

[32] “Parallel K-means data clustering,” 2005, http://users.eecs.northwestern.edu/ wkliao/Kmeans/.

[33] R. Agrawal and R. Srikant, “Quest synthetic data generator,” *IBM Almaden Research Center*, 1994.

[34] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi, “NU-MineBench 3.0,” Technical Report CUCIS-2005-08-01, Northwestern University, Tech. Rep., 2010.

[35] G. Lemson and the Virgo Consortium, “Halo and galaxy formation histories from the millennium simulation: Public release of a VO-oriented and SQL-queryable database for studying the evolution of galaxies in the LambdaCDM cosmogony,” *Arxiv preprint astro-ph/0608019*, 2006.

[36] V. Springel, S. White, A. Jenkins, C. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly *et al.*, “Simulations of the formation, evolution and clustering of galaxies and quasars,” *Nature*, vol. 435, no. 7042, pp. 629–636, 2005.

[37] S. Bertone, G. De Lucia, and P. 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, 2007.

[38] G. De Lucia and J. Blaizot, “The hierarchical formation of the brightest cluster galaxies,” *Monthly Notices of the Royal Astronomical Society*, vol. 375, no. 1, pp. 2–14, 2007.

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

[40] Y. Liu, W.-k. Liao, and A. Choudhary, “Design and evaluation of a parallel HOP clustering algorithm for cosmological simulation,” in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS ’03. Washington, DC, USA: IEEE Computer Society, 2003, p. 82.1.