Optimizing Parallel Clustering Throughput in Shared Memory

Michael Gowanlock* David M. Blair* Victor Pankratius*
* Massachusetts Institute of Technology, Haystack Observatory
Westford, MA, U.S.A.
[gowanloc, dblair, pankrat]@mit.edu

Abstract—This article studies the optimization of parallel clustering throughput in the context of variant-based parallelism, which exploits commonalities and reuse among variant computations for multithreading scalability. This direction is motivated by challenging scientific applications where scientists have to execute multiple runs of clustering algorithms with different parameters to determine which ones best explain phenomena observed in empirical data. To make this process more efficient, we propose a novel set of optimizations to maximize the throughput of Density-Based Spatial Clustering of Applications with Noise (DBSCAN), a frequently used algorithm for scientific data mining in astronomy, geoscience, and many other fields. Our approach executes multiple algorithm variants in parallel, computes clusters concurrently, and leverages heuristics to maximize the reuse of results from completed variants. As scientific datasets continue to grow, maximizing clustering throughput with our techniques may accelerate the search and identification of natural phenomena of interest with computational support, i.e., Computer-Aided Discovery. We present evaluations on a whole spectrum data sets, such as geoscience data on space weather phenomena, astronomical data from the Sloan Digital Sky Survey on intermediate-redshift galaxies, as well as synthetic datasets to characterize performance properties. Selected results show a 1115% performance improvement due to indexing tailored for variant-based clustering, and a 2209% performance improvement when applying all of our proposed optimizations.

Index Terms—Parallel Clustering, Clustering Throughput, DBSCAN, Data Mining, Computer-Aided Discovery.

1 Introduction

Clustering algorithms play an important role in the analysis of scientific data sets. As scientific data volumes continue to increase, throughput scalability is starting to get more attention in addition to traditional techniques that aim to improve the response time of single analysis runs. Throughput optimization is key in exploratory studies where clustering is executed multiple times on the same data set, but with different parameters. The motivation for this approach is related to the fact that only certain clustering parameter values can lead to a physically meaningful classification that makes scientific sense. However, parameters are typically not known a priori and also depend on the input data properties, so domain scientists need to inspect the results of different clustering variant instances. This motivates the study of a new form of parallelism that executes algorithmic instances with different configurations, which we term variant-based parallelism. In contrast to embarrassing parallelism, variant-based parallelism can exhibit commonalities in the computation and generation of results, and intermediate results may be reusable across instances operating with different parameters. Such reuse patterns can be exploited for more sophisticated throughput optimizations.

Variant-based parallelism has significant relevance for science applications that process large data sets. To put this into perspective, we study concrete examples from geoscience and astronomy as a vehicle to quantify the gains of our proposed parallel optimizations on real applications. One case study highlights clustering in the context of space weather [1], which characterizes the Total Electron Content (TEC) in the Earth's ionosphere at a given time. There, domain scientists are interested in finding clusters of similar TEC values that propagate in space and time; one particular class, called Traveling Ionospheric Disturbances (TIDs), can disrupt terrestrial and space-based communication or even cause breakdowns of the power grid [1], [2]. In addition, ionospheric TEC has been shown to be sensitive to tsunamis and earthquakes [3], as well as space-based phenomena such as solar coronal mass ejections [1]. Therefore, any analysis throughput improvement is of high societal relevance and has the potential to enhance natural hazards monitoring systems. Another case study focuses on astronomical data from the Sloan Digital Sky Survey [4]. This survey contains the positions of intermediate redshift galaxies. One application of clustering these galaxies is to understand star formation in galaxy clusters. Other science studies, not discussed here, have applied clustering to detect γ-rays [5] or other large structures in the universe, such as groups, filaments, and clusters [6]. Finally, we also evaluate our approach in experimental setups with synthetic data and controlled levels of noise.

Our work leverages Density-Based Spatial Clustering of Applications with Noise (DBSCAN [7]) to find regions of arbitrary shape and size as well as outliers in our data. The astronomy and geoscience motivation for this work is to examine datasets at differing densities and scales, which can be used in a "Computer-Adided Discovery" system assisting researchers in finding new phenomena [8]. This article extends [9] and makes the following novel contributions:

- We introduce VARIANTDBSCAN, a parallelization tech-

nique consisting of executing algorithmic clustering variants for different parameters in parallel and maximizing reuse of data between variants to improve efficiency in a shared-memory environment.
• We introduce indexing techniques that trades off increased computation for decreased memory accesses. This is important because VARIANTDBSCAN is memory-bound in two dimensions. We demonstrate that indexing is essential for efficient variant-based parallel clustering.
• We develop two different reuse strategies for cluster results between variants that are predicated on selecting relevant data from completed variants.
• We present two scheduling heuristics that improve clustering throughput. The underlying problem is an online scheduling problem: how and when clustering results can be reused depends on the order in which variant clustering computations are started and completed.
• We present successful evaluations on four real-world space weather datasets, two galaxy datasets, and twelve synthetic datasets.

The paper is organized as follows. Section 2 describes the problem statement and algorithmic background for our application context. Section 3 outlines related work. Section 4 introduces efficient indexing and data reuse approaches, as well as novel scheduling heuristics for variant execution. The experimental evaluation is detailed in Section 5, and Section 6 presents the conclusion and future work directions.

2 PROBLEM STATEMENT AND ALGORITHMIC BACKGROUND

2.1 Problem Statement

We present a new method for maximizing clustering throughput in a shared memory environment under the following assumptions:

- Let D be a database of 2-dimensional (2-D) points to be clustered. Each point, p_i, $i = 1, \ldots, |D|$, is defined by coordinates (x_i, y_i).
- The clustering algorithm has to find a number of clusters of arbitrary shape, while also detecting outliers. This is satisfied by the DBSCAN algorithm [7], which requires two parameters: (i) the distance, ϵ, that is searched within the neighborhood of a point object; and (ii) the number of neighboring points, *minpts*, within ϵ that are required for each point that is a member of a cluster. In contrast to other clustering techniques, clusters are defined by density: points that are found near each other are assigned to the same cluster, while points with insufficient density in their neighborhood are outliers.
- Let V be a set of parameterized DBSCAN variants denoted as v_i, $i = 1, \ldots, |V|$, where each v_i is defined by $(v_i^{\epsilon}, v_i^{minpts})$, the input parameters of DBSCAN. For a given V, there will be $|V|$ different cluster results.
- We assume that we can store all relevant data in memory.

The aim is to optimize the throughput of all the clustering algorithm executions with varying input parameters, as defined by V. While the choice of ϵ and *minpts* is specific to the application, note that increasing the value of ϵ increases the neighborhood search time, and increasing *minpts* increases the number of noise points. For a given D, these values are chosen so as to obtain meaningful cluster results (e.g., if ϵ is large and *minpts* is small, then one massive cluster can emerge, which is not valuable). As point density increases, smaller ϵ values are utilized. Ideal values of ϵ and *minpts* will therefore achieve a balance between too much noise and too few clusters.

2.2 Background of the Clustering Algorithm

We outline the principles of the DBSCAN clustering algorithm, which is described in detail in [7]. We use DBSCAN because it can detect clusters of arbitrary shape and density, which is useful for understanding natural phenomena, such as space weather, or studying galaxy clusters. To that end, we utilize these types of real-world data in the experimental evaluation.

The threshold distance defined by ϵ determines the ϵ-neighborhood of p, $N_{\epsilon}(p) = \{q \in D | dist(p,q) \leq \epsilon\}$, with the distance function $dist(p,q)$ being an arbitrary distance measure (e.g. Euclidean). For a given p, if $|N_{\epsilon}(p)| \geq$ *minpts*, then p is referred to as a *core point*. There are a number of reachability criteria set by the DBSCAN algorithm, described as follows. First, given a core point p and a point $q \in D$, we say that $q \in N_{\epsilon}(p)$ is *directly density reachable* from p, as it lies within the ϵ-neighborhood of p. However, points $(p,q) \in D$ may be *density reachable* if they are part of a chain of connected points belonging to a cluster, i.e., $p_1, \ldots, p_n$, where p_{i+1} is *directly density reachable* from p_i, for each of $1 < i < n$ and $p = p_1$, and $q = p_n$. Points $(p,q) \in D$ are *density connected* if there exists a point $r \in D$, that is *density reachable* to both p and q. Finally, a point $p \in D$ is considered a *border point* if it is not a core point, but is density reachable from another core point. Border points are assigned to clusters, while points unreachable by core points are outliers.

The pseudocode is outlined in Algorithm 1, which has the inputs: (i) the database D of points to be clustered; (ii) the threshold distance (ϵ); and (iii) the minimum number of points, *minpts*, within the ϵ-neighborhood to be considered a core point, and (iv) an index T (here, an R-tree [10]). The algorithm computes a set of clusters, $\mathcal{C}$, where a set of points belonging to a single cluster is denoted as C, and the set of points labeled as noise.

We outline the algorithm as follows. We initialize four sets (lines 2-5) as follows: (a) *visitedSet* stores the points that have already been visited by the algorithm, (b) *clusterSet* contains all of the points assigned to a cluster, (c) *noiseSet* contains those points that are noise/outliers, and (d) $\mathcal{C}$ which is the set of individual clusters output by the algorithm. After initializing these sets, the algorithm loops over all points in D that have not yet been visited on line 6. Then, a new cluster is initialized (line 7). The point is marked as visited on line 8. The set of neighbors within ϵ of the point p, are obtained by searching the points in D using the R-tree [10] (line 9). Next, the algorithm determines if a point is a core point and adds it to a current cluster, or marks it as noise (line 10). If the point is a core point, then it

Algorithm 1 The Density-Based Spatial Clustering of Applications with Noise Algorithm (DBSCAN).

```
1: DBSCAN(D, ϵ, minpts, Rtree T)
2: visitedSet← ∅
3: clusterSet← ∅
4: noiseSet← ∅
5: C ← ∅
6: for all p ∈ D|p ∉visitedSet do
7:     C ← ∅   //C contains multiple clusters (C).
8:     visitedSet ← visitedSet ∪ {p}
9:     N ← NeighborSearch(p, ϵ, T)
10:    if |N| < minpts then noiseSet ← noiseSet ∪ {p}
11:    else
12:        C ← C ∪ {p};
13:        clusterSet ← clusterSet ∪ {p}
14:        for all i ∈ N do
15:            N ← N\i
16:            if i ∉ visitedSet then
17:                visitedSet ← visitedSet ∪ {i}
18:                N̂ ← NeighborSearch(i, ϵ, T)
19:                if |N̂| ≥ minpts then N ← N ∪ N̂
20:            if i ∉ clusterSet then
21:                C ← C ∪ {i};
22:                clusterSet ← clusterSet ∪ {i}
23:    C ← C ∪ C
24: return
```

is added to the cluster, C, (line 12) and the set of points assigned to clusters (line 13). Then, all of the points within the ϵ-neighborhood are searched (lines 14), and the cluster is expanded by searching all of these subsequent neighbors that have not been visited (lines 16-19), and the points are added to the cluster (lines 20-21). After all of the points have been visited, the algorithm outputs the set of individual clusters, $\mathcal{C}$.

Note that on lines 9 and 18 a brute-force approach at these steps would require examining all of the points in D, thus the algorithm would have a time complexity of $O(|D|^2)$; however, using a spatial index [7] such as the R-tree can reduce the complexity to $O(|D|\log|D|)$. The time complexity of $O(|D|\log|D|)$ has been debated in the literature, and we refer the reader to [11] for further information.

3 Related Work

Efficiency improvements of DBSCAN for single cluster execution, i.e., without variant parallelism, have been proposed in [12], [13], [14], [15], [16], [17], [18]. In [12], two phases of DBSCAN are parallelized, by denoting a master that performs the clustering, and workers assigned to range queries. The ϵ-neighborhood is retrieved for a point, and if it happens to be a core point, then the resulting neighbors are searched in parallel. The work in [13] also employs a master-slave approach, decomposing the data based on the distance between points. Other works have used the MapReduce framework to execute DBSCAN on up to 13 compute nodes [19], and 128 compute nodes with an emphasis on load balancing to address skewed datasets [15]. A distributed-memory implementation of DBSCAN using the disjoint-set data structure is presented in [16], and an approximate version in [17]. A distributed GPU implementation is advanced in [20], and other GPU implementations are discussed in [18], [21]. Spatiotemporal applications are discussed in [22], and [23] attempts to reduce the search space by exploiting the triangle inequality property. The best DBSCAN scalability result to date was achieved by the BD-CATS implementation [24], which clustered 1 (1.4) trillion points end-to-end in 20 (30) minutes using datasets from cosmology and plasma physics.

A related algorithm that finds good parameter values for DBSCAN is OPTICS [25]. It takes as input a maximum ϵ value, that we denote as δ, and a fixed value for *minpts*. It generates an ordering of the data points in the database, D, and corresponding cluster structures for all $\epsilon \leq \delta$. The algorithm can yield clustering results similar to DBSCAN for a large range of ϵ values. Unfortunately OPTICS is unsuitable if a range of *minpts* values are required in addition to multiple values of ϵ.

4 Variant-Based Parallelism Optimizations

This section introduces a set of optimizations that are required to successfully scale parallel clustering which includes the proposed algorithm, indexing approach, data reuse strategies, and scheduling the execution of variants.

4.1 Indexing Approach for Shared Memory Algorithms

We employ an R-tree spatial index [10] to reduce response time of DBSCAN and efficiently calculate the ϵ-neighborhood of points in D. Because DBSCAN is memory-bound in 2-D due primarily to ϵ-neighborhood searches, scalability in a shared-memory environment is limited. To reduce memory accesses, we exploit the trade-off between R-tree search accuracy and the number of candidate points which may be within ϵ of a searched point. As the number of memory accesses decreases at the expense of more computation, it allows ϵ-neighborhood searches to occur in parallel across variants. A similar trade-off has been explored to study moving object trajectories [26].

The R-tree creates an index for points inside a set of minimum bounding boxes (MBBs). Let r denote the number of points stored per MBB. Indexing each point in its own MBB (i.e., $r = 1$) yields the most accurate search and the tree contains $|D|$ leaf nodes. To search the R-tree, a query MBB is constructed around a point, $p_i \in D$, that is augmented by ϵ, where $MBB_i^{min} = (x_i - \epsilon, y_i - \epsilon)$, and $MBB_i^{max} = (x_i + \epsilon, y_i + \epsilon)$. As r increases, the areas of the MBBs increase and so does the probability that a query MBB overlaps an indexed MBB, requiring more candidate points to be filtered. Before indexing, we sort the points $p_i \in D$ into bins in the x and y dimensions of unit width.

Algorithm 2 The NeighborSearch Algorithm.

```
1: NeighborSearch(p, ϵ, Rtree T)
2:     overlappingMBBSet ← ∅; candidateSet ← ∅
3:     MBB ← generateMBB(p, ϵ)
4:     overlappingMBBSet ← T.search(MBB, ϵ)
5:     candidateSet ← dataLookup(overlappingMBBSet)
6:     N ← filterCandidateSet(p, ϵ, candidateSet)
7: return N
```

The calculation of the ϵ-neighborhood set for a point p is given in Algorithm 2 and called by the clustering algorithm (Algorithm 1). The algorithm determines the set of MBBs that overlap a query MBB and the set of points that are candidates that may be within ϵ, and derives a query MBB by enlarging p by ϵ. The index tree is searched, and results

in a set of overlapping MBBs. We index multiple points per MBB to decrease tree depth and to reduce index search time. The resulting MBBs contain multiple points, and a lookup array maps the indexed MBBs to the individual data points in D, yielding the candidate set. These points are then filtered to find those that are within ϵ of p, which results in the final set of points.

4.2 Exploiting Data Reuse Across Variants with VARIANTDBSCAN

The main idea of variant-based parallelism is to use outputs of previously clustered variants (the cluster results) as inputs to other variants. We take previously defined clusters, find the points along the cluster edges, and incrementally add appropriate points to the cluster. Reusing a previously defined cluster obviates ϵ-neighborhood searches on all of the points, thereby reducing the total response time for a given variant and improving clustering throughput across V.

Reusing results from another variant requires defining reusability criteria as follows: a variant, v_i, can only reuse the clustering results from another variant, v_j, if it satisfies the inclusion criteria $v_i^{\epsilon} \geq v_j^{\epsilon}$ and if $v_i^{minpts} \leq v_j^{minpts}$. Therefore, when increasing ϵ and/or decreasing *minpts*, an original cluster that is being reused can only increase in size, as all of the original points are guaranteed to still be part of the same cluster. These are consistent with the reachability criteria outlined in Section 2.2.

Algorithm 3 The VARIANTDBSCAN Algorithm.

```
1:  VariantDbscan(D, V, Rtree T_high, Rtree T_low)
2:  parallel for v_i ∈ E do
3:      C_v ← schedule(v_i)
4:      if C_v ≠ ∅ then
5:          destroyedSet ← ∅; visitedSet ← ∅
6:          clusterSeedSet ← getSeedList(C_v)
7:          for all j ∈ clusterSeedSet do
8:              if j ∈ destroyedSet then continue
9:              C ← ∅; C ← C_v[j]; visitedSet ← C
10:             MBB←generateClusterMBB(C)
11:             candidateSet ← T_high.search(MBB)
12:             outsidePointsSet ←candidateSet\C
13:             for all p ∈ outsidePointsSet do
14:                 neighborSet ←NeighborSearch(p, v_i^ε, T_low)
15:                 expandSet ←expandSet ∪ neighborSet ∩ C
16:                 visitedSet ←visitedSet\expandSet
17:             ExpandCluster(D, v_i^ε, v_i^minpts, T_low,
                  expandSet, destroyedSet, C, C_v)
18:         Cluster remainder of points with Dbscan.
19:     else Cluster with Dbscan.
20: end parallel for
```

Algorithm 3 outlines VARIANTDBSCAN. It reuses clustering results from previous variants (clustered with either DBSCAN or VARIANTDBSCAN). Inputs of the algorithm are: (i) the point database D, (ii) the list of variants V, and two R-tree indexes, (iii) T_{high}, and (iv) T_{low}, corresponding to a high resolution R-tree that indexes a single point per MBB, and a lower resolution R-tree used to improve the efficiency of the ϵ-neighborhood searches, as described in Section 4.1.

The algorithm iterates over each variant v_i in its main loop (line 2), which is parallelized to cluster multiple variants of DBSCAN simultaneously. For a given variant, v_i, we check a schedule on line 3 that determines if there is a completed variant that can be reused (the schedule will be described in Section 4.4). If no such variant is available, DBSCAN is executed (line 19), clustering the points without reusing any previous results as described using Algorithm 1 with our indexing scheme from Section 4.1.

If a suitable variant has finished, sets containing a list of destroyed clusters and of points that have been visited are initialized in line 5, and appropriate clusters that should be reused from within C_v are chosen by calling the *getSeedList* method (line 6). This filters the list of total clusters, C_v, yielding a list of cluster IDs to expand, which is stored in *clusterSeedSet* (the criteria used to determine if a cluster should be expanded will be described in Section 4.3).

A loop then iterates over the candidate clusters that may be expanded (line 7). Previous clusters can be destroyed through the process of reusing cluster results. For example, if cluster c_a is expanded and new points are added to the cluster that previously existed in another cluster, c_b, we say that c_b has been destroyed and is not a candidate cluster that can be expanded. We check to see if a given cluster has already been destroyed in line 8. If the cluster has not been destroyed, a new cluster is initialized (line 9). Then, we copy all of the points that were in the previous cluster, $C_v[j]$, to a new cluster, C, (line 9), obviating ϵ-neighborhood searches and filtering on all of these points (recall that C_v is the list of all clusters and corresponding points from a previous clustering result variant, whereas $C_v[j]$ refers to the points belonging to a single cluster).

Next, VARIANTDBSCAN marks all of the points that were in the previous cluster as visited (line 9). Lines 10–16 find the points inside the preexisting cluster, C, that will expand the cluster (Figure 1 (a), unfilled blue points). The key idea is to find these points with few ϵ-neighborhood searches. This is accomplished by generating an MBB around the cluster that is augmented by v_i^{ϵ} on line 10, leading to an MBB that ensures that any points within the cluster that are within v_i^{ϵ} of any other point will be found.

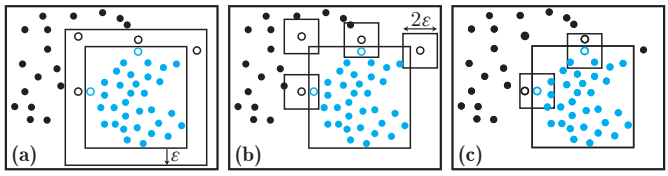


Fig. 1: Illustration of lines 10-17 in Algorithm 3. Across (a)-(c), the blue points belong to a cluster. The blue unfilled points are the ones that are found by the algorithm that are used to expand the cluster. The unfilled black circles are candidates possibly within ϵ of points inside the preexisting cluster, and solid black points are unclustered points.

On line 11, a high-resolution R-tree index is searched to find all of the points within an augmented MBB around the cluster (Figure 1 (a)). Using this high resolution index comprising each point in its own MBB reduces point filtering overhead because the MBB enclosing the cluster (line 10) is much larger than the MBB constructed for an ϵ-neighborhood search around a point (e.g. the lower resolution R-tree in Section 4.1). The search stores all of the points found within the MBB in *candidateSet*, which includes all of the points inside of the cluster C. The points lying outside

of a cluster, *outsidePointsSet* (Figure 1 (a), unfilled black circles), are obtained by taking the relative complement of the *candidateSet* from the set of points in C (line 12). Neighborhood searches are then performed on each point in *outsidePointsSet* (lines 13–14; Figure 1 (b)), with the results of each search stored in *neighborSet*. *neighborSet* therefore contains both points inside of C that will expand the cluster, and points unassigned to C. We then isolate the points inside of C by taking the intersection of *neighborSet* and C, and add these points to a set *expandSet* that will be used to grow the preexisting cluster (line 15, blue unfilled points in Figure 1 (c)). Removal of the points in *expandSet* from the set of visited points occurs on line 16. Finally, the preexisting cluster is expanded on line 17 using the points in *expandSet* following EXPANDCLUSTER (Algorithm 4). Cluster points and corresponding MBBs are exemplified in Figure 1 (c). EXPANDCLUSTER is similar to lines 14–22 in Algorithm 1, but it includes the set of destroyed clusters that are maintained while expanding the cluster. After all of the selected clusters have been reused or destroyed, all remaining points are clustered with DBSCAN on line 18.

Algorithm 4 The Expand Cluster Method.

```
1: ExpandCluster (D, ϵ, minpts, Rtree T, growPntsSet,
                  destroyedSet, C, C_v)
2: N ← growPntsSet
3: for all i ∈ N do
4:     N ← N\i
5:     if i ∉ visitedSet then
6:         visitedSet ← visitedSet ∪ {i}; N̂ ← NeighborSearch(i, ϵ, T)
7:         if |N̂| ≥ minpts then N ← N ∪ N̂
8:     if i ∉ clusterSet then
9:         C ← C ∪ {i}; clusterSet ← clusterSet ∪ {i}
10:    if i ∈ C_v then
11:        id←getOldClusterID(i); destroyedSet←destroyedSet∪{id}
```

4.3 Selection of Cluster Candidates for Variant Reuse

An interesting question is what clusters should be selected as reuse candidates between variants. The VARIANTDBSCAN algorithm (Algorithm 3, line 6) creates a list of seed clusters that are candidates for reuse. As mentioned in the last section, when reusing the points assigned to previous clusters, candidate clusters for reuse may become invalid. As an example, consider the cluster set $C = \{c_a, c_b\}$ and points $p \in c_a$ and $q \in c_b$ assigned to cluster c_a and c_b, respectively. If cluster c_a is reused and new points are added to the cluster and q becomes directly density reachable from p, then q is added to cluster c_a. Cluster c_b is then invalid for reuse, as one of its points has been reassigned.

We propose 3 cluster reuse prioritization techniques:

1) **CLUSDEFAULT–** Select clusters in the order in which they were originally generated.
2) **CLUSDENSITY–** Select clusters for reuse in order of highest to lowest density. We use a simple density measure: $|C|/a$, where $|C|$ is the number of points in the cluster C, and a is the area of an MBB that circumscribes the cluster.
3) **CLUSPTSSQUARED–** The same as CLUSDENSITY, except using $|C|^2/a$. If there are clusters with a large fraction of the points in D, then this metric may be preferable to the density metric above, as a very dense cluster may not contain a large number of points.

Optimizing for cluster reuse can potentially yield a significant reduction in the number of ϵ-neighborhood searches to improve clustering throughput. For example, one approach is to prioritize the clusters with the greatest number of points to maximize reuse, since if the cluster with fewer points is selected, then ϵ-neighborhood searches will need to be performed on all of the points in the larger cluster. More advanced choices of which cluster to reuse are possible, although these may have computational costs which outweigh their benefits.

4.4 Variant Scheduling

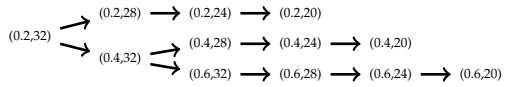
The parameterized cluster algorithm variants ($v_i \in V$) are executed by different threads. VARIANTDBSCAN will cluster from scratch if no appropriate variants can be reused (Section 4.2). Variants in V are sorted first by non-decreasing ϵ and then by non-increasing *minpts* (i.e., $v_i^{\epsilon} \leq v_{i+1}^{\epsilon}$, then sort by $v_i^{minpts} \geq v_{i+1}^{minpts} \iff v_i^{\epsilon} = v_{i+1}^{\epsilon}$). Each thread starts by clustering from scratch when no reuse is possible.

As an example for the need of scheduling, consider Figure 2 (a) showing a dependency tree that minimizes the component-wise differences between parameters (the output of one variant is shown as input into another). Assuming global knowledge and disregarding the ordering of variants, the instance (0.6,20) could reuse (0.2,32) because $\epsilon = 0.6 > 0.2$ and *minpts* is smaller. However, it is more beneficial to reuse (0.6,24) to minimize the component-wise difference in parameters, thus potentially improving the fraction of points that are reused.

Figures 2 (b) and (c) illustrate two possible sequential schedules resulting from this tree. From schedule 1, after $v_i = (0.2, 32)$ is clustered from scratch (within VARIANTDBSCAN, line 19), the rest of the variants can reuse previous clustering results. The ordering of processing variants in Figure 2 (b) and (c) assumes that a single thread performs all of the clustering sequentially ($T = 1$). We propose two thread scheduling heuristics:

1) **SCHEDGREEDY–** A thread is assigned a variant and will cluster with VARIANTDBSCAN using the results of a completed variant with the smallest difference in parameters (e.g., Figure 2 (b)). If no variant can be reused, then the variant with the smallest ϵ and largest *minpts* value is clustered from scratch (line 19 in VARIANTDBSCAN).
2) **SCHEDMINPTS–** Create an ordered list with all unique v_i^{ϵ} in elements of the set V that have maximum *minpts* values (e.g., select (0.2,32), (0.4,32), (0.6,32) in Figure 2). This list will be prioritized first to be clustered from scratch. This allows subsequent threads to cluster with VARIANTDBSCAN using a completed variant that may more likely have similar parameters. All remaining variants will be clustered with VARIANTDBSCAN using the criteria outlined in SCHEDGREEDY. This is shown in Figure 2 (c).

Threads are assigned to variants using a thread pool. The heuristics above exploit two aspects of data reuse: (i) minimizing the time to solution of individual variants such that



(a)

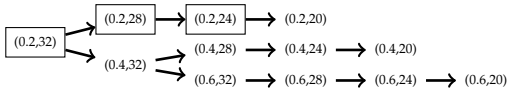
((0.2,32),(0.2,28),(0.2,24),(0.2,20),(0.4,32),(0.4,28),
(0.4,24),(0.4,20),(0.6,32),(0.6,28),(0.6,24),(0.6,20))

(b) Schedule 1

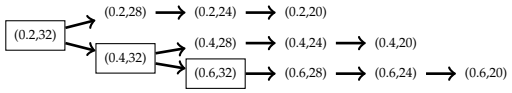
((0.2,32),(0.4,32),(0.6,32),(0.2,28),(0.2,24),(0.2,20),
(0.4,28),(0.4,24),(0.4,20),(0.6,28),(0.6,24),(0.6,20))

(c) Schedule 2

Fig. 2: (a) Dependency tree with neighboring nodes $(v_i^{\epsilon}, v_i^{minpts})$ that have minimal difference between input parameters across variants partially ordered by $v_i^{\epsilon} \geq v_j^{\epsilon}$ and $v_i^{minpts} \leq v_j^{minpts}$, where v_i reuses results from v_j. (b) Example schedule derived from a depth-first ordering of the dependency tree with $T = 1$ threads. (c) Example schedule that prioritizes clustering *minpts* values first with $T = 1$.



(a) SchedGreedy



(b) SchedMinpts

Fig. 3: Example of prioritized variants with $T = 3$, where the variants outlined in the boxes are first clustered from scratch using (a) SchedGreedy, and (b) SchedMinpts.

they can be reused for future variants (SchedGreedy); and (ii) maximizing the diversity of clustered variants so that more point reuse may be achieved (SchedMinpts). These two data reuse considerations are conflated with T. Given $|V|$ and T, initially all threads will cluster with Dbscan as no variants can be reused (line 19 in Algorithm 3); therefore, the maximum fraction f of variants that are candidates to reuse data are $f = \frac{|V|-T}{|V|}$ (so at minimum, $\geq 1 - f$ are clustered from scratch). As an example, consider the case where there are 3 threads ($T = 3$) clustering the variants in Figure 3. The variants clustered from scratch are highlighted by the boxes, where SchedGreedy (Figure 3 (a)) initially clusters those variants with the smallest difference in parameters, while SchedMinpts (Figure 3 (b)) initially clusters variants with a wider range of input parameters. With $T > 1$ (unlike Figure 2), we do not show a resultant schedule, as variants are not statically assigned to threads.

5 Evaluation

5.1 Datasets

We utilize 4 different classes of datasets to enable a case study evaluation under different scenarios (Table 1). For the first case study, class *SW-* consists of real-world space weather TEC datasets that have been collected for studies of the Earth's ionosphere, which consists of charged particles from about 90–1000 km altitude [27]. The ionosphere can delay signals received from satellites (e.g. GPS satellites [28]) at different frequencies depending on TEC along the line of sight to each satellite. This proportional delay can be measured by GPS receivers on the ground and used to create maps that help track fluctuations in TEC caused by ground-based phenomena such as tsunamis and earthquakes [3], and space-based phenomena such as solar coronal mass ejections [1]. Space weather data typically consists of a map of TEC values. A typical task is to select a range of TEC values and determine clusters for the resulting thresholded set of 2-D data points (e.g., red features in Figure 4). These points are an example of the *SW-* class of datasets.

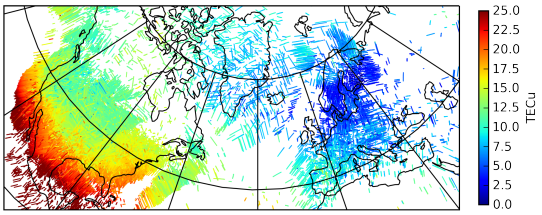


Fig. 4: Example of a Total Electron Content (TEC) map of the Earth's ionosphere, obtained through GPS satellite signal processing [27]. The data shown is that of dataset *SW1*, as described in Section 5.1.

For the second case study, the *SDSS-* class contains a sample of galaxies having an intermediate redshift (z) of $0.30 \leq z \leq 0.35$ from the Sloan Digital Sky Survey (SDSS) [4]. In this context, Dbscan is being used for structure detection of galaxy clusters [6].

The third case study includes classes *cF-* and *cV-* which are synthetic datasets. These were created in order to conduct controlled robustness evaluations on various point distributions. In the synthetic datasets, a fraction of the points are: (i) assigned to synthetic clusters, the center of which is selected at random in a 2-D region; and (ii) uniformly distributed noise points. The class *cF-* contains a fixed number of synthetic clusters for a given number of total points, where the number of clusters is calculated as $|D| \times 10^{-4}$. There are a uniform number of points per cluster, determined by the fraction of points that are not noise; these noise points may have the effect of producing additional or larger clusters when clustering. Class *cV-* varies the number of points per cluster, where the total number of points assigned to clusters is calculated as in *cF-*. The number of points assigned to a cluster was generated in the range of 0%–500% of those in the *cF-* class. The *cF-*, *cV-*, and *SW-* classes of datasets are available at [29]. Refer to [4] for the class of *SDSS-* datasets.

5.2 Experimental Methodology

We develop multithreaded implementations of the previously described variant-based parallel clustering approach using OpenMP in C++ and the O3 compiler optimization flag. The execution is carried out on up to 16 cores of dedicated 2.40 GHz Intel Xeon E5-2630 v3 processors with 20 MB L3 cache. Response times are averaged over 3 trials. In the following experimental scenarios we omit detailing

TABLE 1: Characteristics of Datasets

Dataset	$\|D\|$	Noise	Dataset	$\|D\|$	Noise
cF_1M_5N	10^6	5%	*cV_1M_30N*	10^6	30%
cF_100k_5N	10^5	5%	*cV_100k_30N*	10^5	30%
cF_10k_5N	10^4	5%	*cV_10k_30N*	10^4	30%
cF_1M_15N	10^6	15%	*SW1*	1864620	N/A
cF_1M_30N	10^6	30%	*SW2*	3162522	N/A
cF_100k_30N	10^5	30%	*SW3*	4179436	N/A
cF_10k_30N	10^4	30%	*SW4*	5159737	N/A
cV_1M_5N	10^6	5%	*SDSS1*	2×10^6	N/A
cV_1M_15N	10^6	15%	*SDSS2*	5×10^6	N/A

performance results for all individual datasets in cases where the salient performance characteristics are similar.

The selection of ϵ and *minpts* values for the variants in V is not trivial, and $|V|$ must be sufficient to demonstrate successful data reuse. We select values which give a reasonable number of clusters with regards to the size of the dataset and avoid selecting ϵ and *minpts* such that either one or zero clusters emerge. A heuristic [7] for selecting *minpts* finds 4 to be a good value; we therefore use 4 as the smallest value. We do not list each variant in V in the form $v_i = (v_i^{\epsilon}, v_i^{minpts})$, and elect to use the following notation where applicable: $A = \{v_i^{\epsilon}, v_{i+1}^{\epsilon}, \dots\}, B = \{v_j^{minpts}, v_{j+1}^{minpts}, \dots\}$, where $V = A \times B$ (V is the Cartesian product of the two sets). For example, if $A = \{0.1, 0.2\}, B = \{1, 2\}$ then $V = \{(0.1, 1), (0.1, 2), (0.2, 1), (0.2, 2)\}$.

Reference Implementation: We compare the performance of all implementations to a reference implementation that executes DBSCAN (Algorithms 1 and 2) with $T = 1$ and $r = 1$ (sequential without index optimization).

5.3 Efficient Indexing for Variant-Parallel Clustering

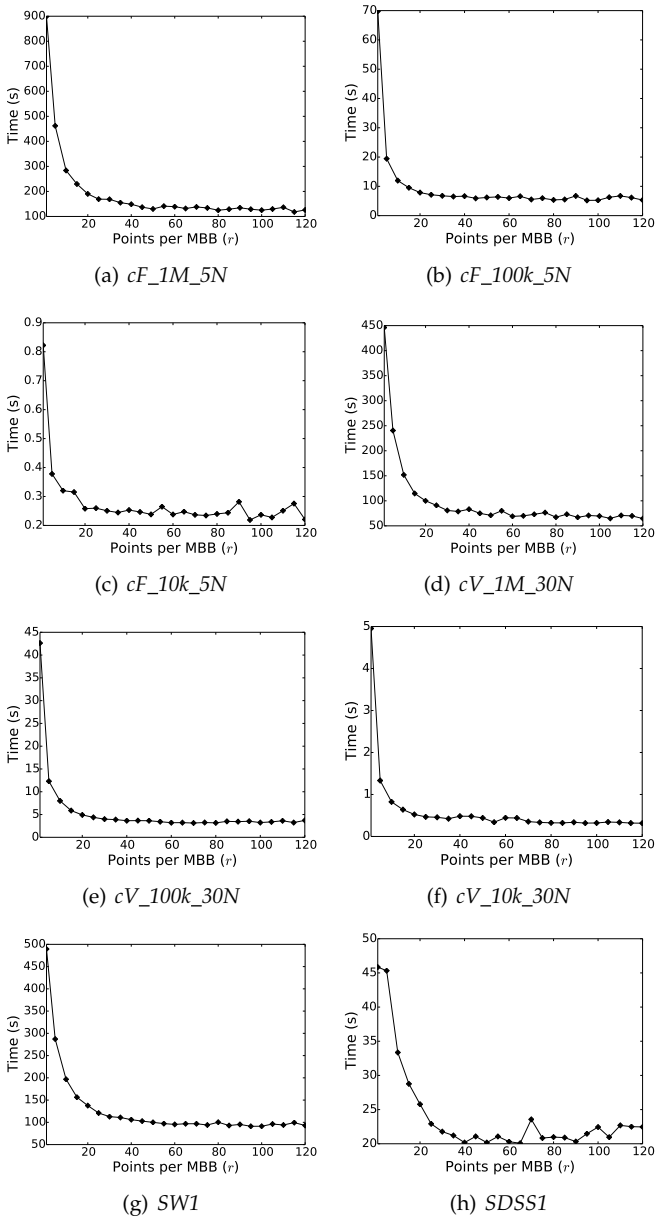
As described in Section 4.1, one challenge of executing multiple variants in parallel is the memory-bound nature of clustering from scratch with DBSCAN in 2-D. To reduce tree depth, leading to a shorter tree traversal, multiple points can be stored per MBB in the R-tree. Therefore, when searching for all $p_i \in D$, fewer internal nodes of the R-tree will be accessed, thereby alleviating memory pressure. To demonstrate the effectiveness of the indexing scheme, a number of threads concurrently perform DBSCAN on 16 identical variants; we elect to cluster the same variant multiple times, so that our results can be interpreted without being confounded by other effects, such as uneven workloads assigned to different threads. When the number of threads $T = 1$, a single thread sequentially clusters 16 variants, whereas when $T = 16$ each thread (core) concurrently clusters a single variant. We utilize datasets and parameters as shown in Table 2 ($S1$). The synthetic datasets have been chosen to illuminate the performance with realistic levels of noise (5–30%) and number of data points (10^4–10^6).

Index optimization is achieved by selecting a value of r that leads to a good response time; this in turn is affected by the spatial data distribution, the total number of MBBs (which is calculated as $\lceil |D|/r \rceil$), depth of the R-tree, and the value of ϵ. We determine good values for r experimentally, as described next.

Figure 5 shows the response time vs. the number of points per MBB (r) for $S1$. Across all datasets, we find that a value of r that leads to low response time occurs in a large

TABLE 2: Scenario 1 ($S1$)

Dataset	v_i^{ϵ}	v_i^{minpts}	i	Clusters
cF_1M_5N	0.5	4	$1, \dots, 16$	672
cF_100k_5N	4	"	"	200
cF_10k_5N	10	"	"	15
cV_1M_30N	0.5	"	"	74
cV_100k_30N	2	"	"	14802
cV_10k_30N	10	"	"	1
SW1	0.5	"	"	2333
SDSS1	0.4	"	"	119



(a) *cF_1M_5N* (b) *cF_100k_5N*

(c) *cF_10k_5N* (d) *cV_1M_30N*

(e) *cV_100k_30N* (f) *cV_10k_30N*

(g) *SW1* (h) *SDSS1*

Fig. 5: Response time vs. the number of points per MBB (r) for all of the datasets in scenario $S1$.The figure quantifies the trade off between the highest resolution index ($r = 1$) having the highest response time and decreasing response times with lower resolution indexes ($r > 1$). For each dataset, values for $r \geq 50$ improve the response times.

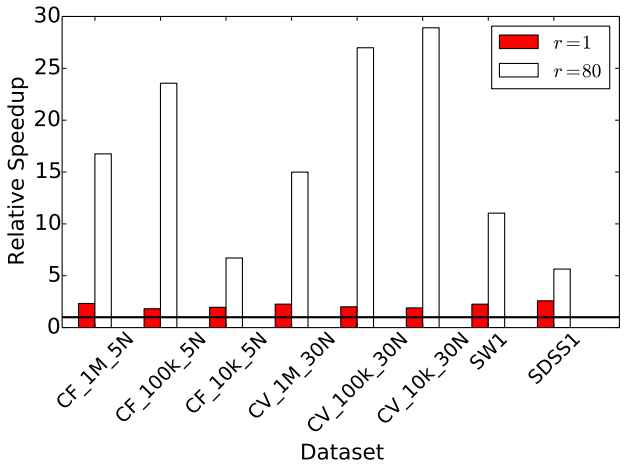


Fig. 6: Relative speedup of VARIANTDBSCAN in scenario $S1$ compared to the reference implementation. The plot shows significant performance gains when the index is optimized with r. VARIANTDBSCAN is configured with $T = 16$ (each thread executing a single variant), where we compare using $r = 1$ and $r = 80$ across all datasets. Values over $y = 1$ show a performance gain over the reference implementation.

range ($50 \leq r \leq 120$). The range is large is because the total number of MBBs decreases with r as $\lceil |D|/r \rceil$. With increasing r, the search is less precise, returning more candidate points that are not within ϵ of the point being searched. Therefore, while we only show results up to $r = 120$, eventually, filtering overheads will cause the response time to increase beyond $r = 120$.

Figure 6 shows the relative speedup as the ratio of the response time of the reference implementation to the multithreaded implementation having $T = 16$ and $r = 80$ on $S1$. We configure the multithreaded implementation with $r = 80$ because it leads to reasonably low response times across all of the datasets (Figure 5). We show $r = 1$ with $T = 16$ (red bars), corresponding to no index optimization, but with each thread concurrently clustering a single variant. If $r = 1$ and $T = 16$, then the maximum speedup is 2.33× (*cF_1M_5N*); however, by using efficient indexing (with $r = 80$), we achieve relative speedup values from 6.71× on *cF_10k_5N* to 28.9× on *cV_10k_30N*. These speedup values were achieved using a nominal value for r that performs well across all scenarios in $S1$. However, we could select the best values for r for each dataset. For instance, the real-world *SW1* dataset with $r = 95$ is 1115% faster than the reference implementation, in comparison to 1004% faster when $r = 80$. This evaluation and those to follow are not exhaustive of the different parameter values of DBSCAN; Scenario $S1$ demonstrates that without efficient indexing, multithreaded implementations that concurrently execute DBSCAN are inhibited by memory-bound ϵ-neighborhood calculations.

5.4 Efficient Data Reuse for Variant-Parallel Clustering

We outline the efficiency of reusing cluster results between variants as described in Section 4.3, and Algorithm 3. We use $T = 1$ (a single thread clusters all of the variants) to demonstrate data reuse independently of clustering the variants in parallel. We utilize $|V| = 24$ that are applied to each dataset, outlined in Table 3 ($S2$).

TABLE 3: Scenario 2 ($S2$)

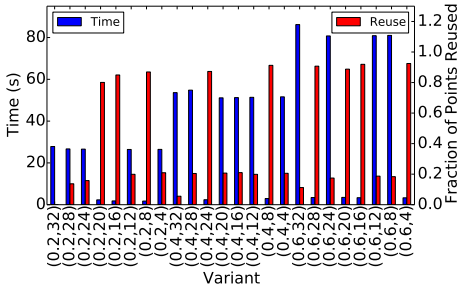
Dataset	Parameters
cF_1M_5N, *cV_1M_5N*, *cF_1M_15N*, *cV_1M_15N*, *cF_1M_30N*, *cV_1M_30N*, *SW1*	For all datasets: $V = A \times B$, where $A = \{0.2, 0.4, 0.6\}$, $B = \{4, 8, \ldots, 32\}, \|V\| = 24$

In Figure 7, we show the response time and the fraction of points reused for each individual variant in the *SW1* dataset. Variant $v_i = (0.2, 32)$ is clustered from scratch (line 19 in VARIANTDBSCAN), and the rest are clustered using the SCHEDGREEDY ordering. We observe that high data reuse leads to lower response time. In comparison to CLUSDEFAULT (Figure 7 (a)), we find that using the CLUSDENSITY (Figure 7 (b)) significantly reduces the response time across the individual variants. CLUSPTSSQUARED leads to poor performance (Figure 7 (c)).

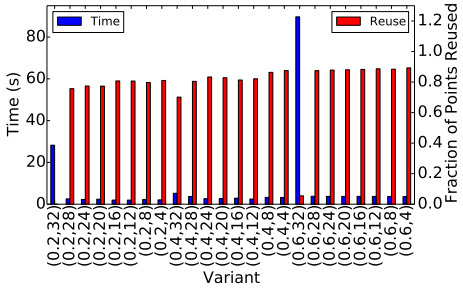
To summarize results of all variants in Figure 7, Figure 8 shows that across data reuse methods, high reuse leads to low response time. In the low reuse regime, where more ϵ-neighborhood searches occur, the difference in response time across ϵ values is more pronounced than it is in the high data reuse regime. To cluster V, the reference implementation yields a response time of 1235.0 s, and when using VARIANTDBSCAN the response time is: 801.5 s (CLUSDEFAULT), 185.8 s (CLUSDENSITY), and 1282.6 s (CLUSPTSSQUARED). VARIANTDBSCAN with CLUSPTSSQUARED is slower than the reference implementation (sequential DBSCAN), while with CLUSDENSITY it is 565% faster. The slowdown using CLUSPTSSQUARED illustrates that reusing results can degrade performance if high data reuse cannot be achieved.

We summarize similar results for other datasets in Figure 9 (a); we plot the relative speedup as the ratio of the response time of the reference implementation to the response time using VARIANTDBSCAN. The minimum and maximum relative speedup values for the synthetic datasets are 6.88×, and 28.3×, respectively. The performance improvements for the datasets with the most noise (*cF_1M_30N*, and *cV_1M_30N*) are the lowest. Thus, datasets with a low degree of noise will benefit the most from VARIANTDBSCAN. In Figure 9 (b), we show the average number of points reused across all of V for each dataset. For the datasets that have the most noise, $\sim 60\%$ of points on average are reused between variants. We conclude that reusing data between variants can significantly improve performance on both synthetic and real-world datasets (recall that $T = 1$ using both VARIANTDBSCAN and the reference implementation).

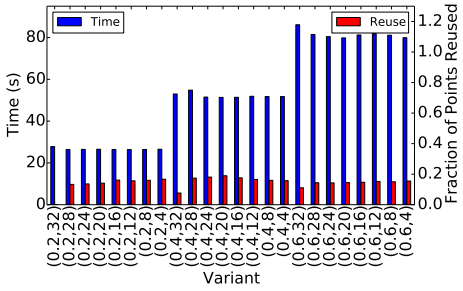
The order in which points are processed can slightly change clustering results, as points can be misidentified when comparing the output of DBSCAN to VARIANTDBSCAN. We evaluate the quality of the results using the metric in [30], by comparing the cluster or noise assignment of each point in VARIANTDBSCAN vs. DBSCAN. If a point is misidentified as a noise (or non-noise) point, then the point receives a score of 0. If a point is in a cluster in both executions, then it receives a score between 0 and 1 as $\frac{|E \cap F|}{|E \cup F|}$, where E is the cluster assigned to the point using DBSCAN



(a) CLUSDEFAULT



(b) CLUSDENSITY



(c) CLUSPTSSQUARED

Fig. 7: Response time (left y-axis) and fraction of points reused (right y-axis) vs. individual variant for *SW1* in scenario $S2$ using the three cluster reuse schemes (a) CLUSDEFAULT, (b) CLUSDENSITY, and (c) CLUSPTSSQUARED. Across all plots high data reuse successfully leads to low response times. All variants are clustered sequentially ($T = 1$), and $r = 70$, with variant parameters (ϵ, *minpts*) shown below each bar.

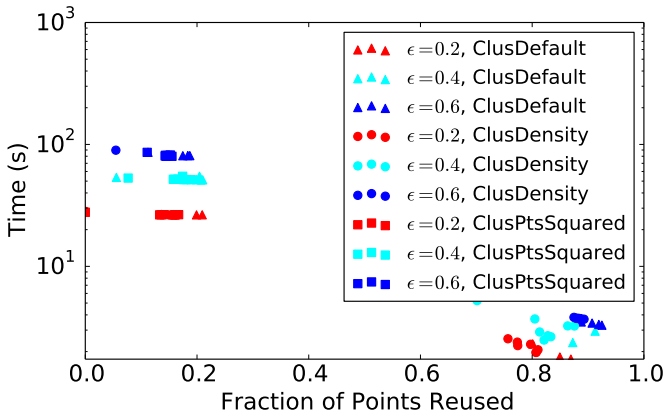
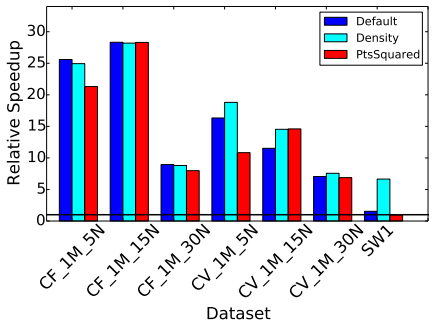
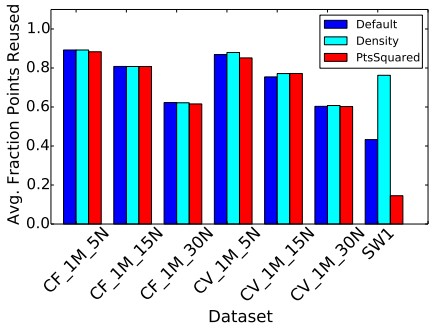


Fig. 8: Response time vs. fraction of points reused, where values are grouped by ϵ family (color), and cluster reuse scheme (marker shape); response time and reuse values from Figure 7 (a)–(c) are shown (from *SW1* in scenario $S2$). Response times are lower if sufficient data reuse occurs.



(a)



(b)

Fig. 9: Sensitivity analysis quantifying speedup and data reuse based on datasets with different characteristics. This exemplifies that the choice of reuse technique depends on the input dataset. (a) Relative speedup of VARIANTDBSCAN with scenario $S2$ compared to the reference implementation (values over $y = 1$ indicate a performance gain). VARIANTDBSCAN is configured with SCHEDGREEDY, and $r = 70$. The 3 cluster reuse schemes are shown: CLUSDEFAULT (blue), CLUSDENSITY (cyan), and CLUSPTSSQUARED (red). (b) Average fraction of points reused across each variant in V in each experimental scenario shown in (a).

and F is the cluster assigned to the point in VARIANTDBSCAN. Therefore, if both algorithms have identical output, then they will receive a score of 1. The average quality score of a variant is the average of all of the quality scores of the individual points. Figure 10 plots a selection of quality scores for the results in Figure 9 (a), where we show the average of the quality scores across all $|V| = 24$ variants. The lowest average quality score is 0.998, showing that VARIANTDBSCAN yields nearly identical output to DBSCAN. Other quality metrics like the Omega-Index [31] used in [17] could be used. However, the metric in [30] is nearly the same as the Omega-Index, as both metrics use a score as determined by whether each pair of points between the two clustering outputs are found within the same cluster.

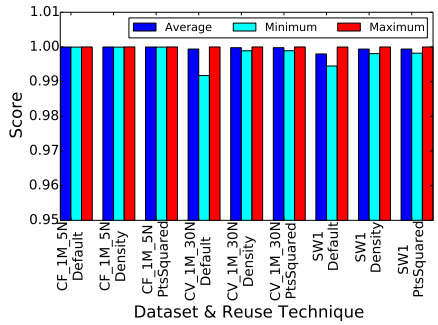


Fig. 10: Quality of the output of VARIANTDBSCAN in comparison to DBSCAN. The score value indicates that VARIANTDBSCAN generates highly similar clustering results to DBSCAN.

5.5 Combining Indexing, Data Reuse, and Scheduling

In the previous sections, we have shown that significant performance gains can be achieved using efficient indexing techniques and reusing data between variants. We combine indexing and result reuse in a multithreaded implementation that relies on assigning variants to threads based on the input parameters. To demonstrate the utility of the scheduling algorithms when multiple threads are concurrently clustering with VARIANTDBSCAN, we use two sets of variants, one with a smaller number of values for ϵ than *minpts*, and one with the converse. Due to the differences in data density, and dataset sizes in both classes of real-world datasets (*SW-* and *SDSS-*), we select parameters that do not give too few or too many clusters, i.e., they give a range of values that may be meaningful for the application areas. Thus, we do not use a static set of parameter values across datasets. We utilize the datasets and parameters in Table 4 ($S3$). As was shown in Section 5.3, a good value for r to optimize index searches can be found in a large range, and we select $r = 70$. Recall that if no completed variants can be reused, then the variant is clustered from scratch (line 19 in Algorithm 3). When $T = 16$, 16 threads initially cluster with DBSCAN, as no results exist that can be reused. Therefore, the maximum fraction of variants that can be reused across $S3$ for V_1, V_2, V_3, V_4, and V_5 is $f = \frac{|V|-T}{|V|} = \frac{57-16}{57} = 0.719$.

TABLE 4: Scenario 3 ($S3$)

Dataset (V)	V	A	B
SW1 (V_1, V_3)	V_1	$\{0.2, 0.3, 0.4\}$	$\{10, 15, \ldots, 100\}$
SW2 (V_1, V_3)	V_2	$\{0.15, 0.25, 0.35\}$	$\{10, 15, \ldots, 100\}$
SW3 (V_1, V_3)	V_3	$\{0.04, 0.06, \ldots, 0.4\}$	$\{4, 8, 16\}$
SW4 (V_2, V_3)	V_4	$\{0.3, 0.4, 0.5\}$	$\{10, 15, \ldots, 100\}$
SDSS1 (V_4, V_5)	V_5	$\{0.06, 0.08, \ldots, 0.42\}$	$\{5, 10, 15\}$
SDSS2 (V_1, V_5)	$V = A \times B; \|V_1\| = \|V_2\| = \|V_3\| = \|V_4\| = \|V_5\| = 57$		

Figure 11 shows the relative speedup using the cluster reuse and scheduling algorithms on the four real-world *SW-* datasets. Across all scenarios, CLUSDENSITY (green and purple bars) outperforms CLUSPTSSQUARED. In experimental scenarios with CLUSDENSITY, SCHEDGREEDY outperforms SCHEDMINPTS in 6 of 8 instances. To understand the source of the speedup being either a function of (a) index optimization; or (b) data reuse/scheduling, we show the speedup if only the index optimizations are applied (red

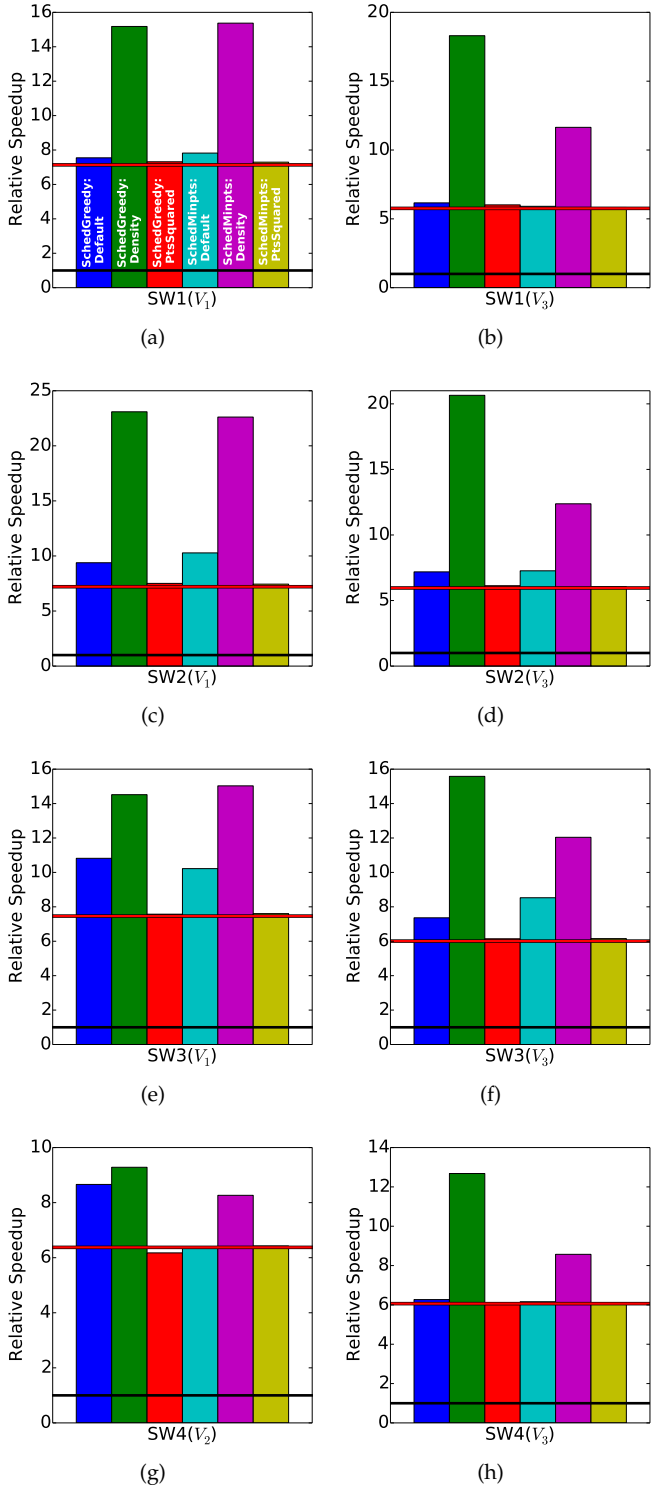


Fig. 11: Relative speedup of VARIANTDBSCAN applying all of the optimizations, i.e., scheduling algorithms, cluster reuse heuristics, index optimization ($r = 70$), and multithreading ($T = 16$) for *SW-* in scenario $S3$ compared to the reference implementation. The left column (a, c, e, g) has a smaller number of values for ϵ than *minpts* (V_1, and V_2), and the converse is shown in the right column (V_3) (b, d, f, h). Values over the black horizontal line indicate a speedup relative to the reference implementation, whereas the red horizontal line shows the speedup when only using the index optimizations ($r = 70$, $T = 16$). The labels on the bars in (a) correspond to the same configurations in (b)-(h).

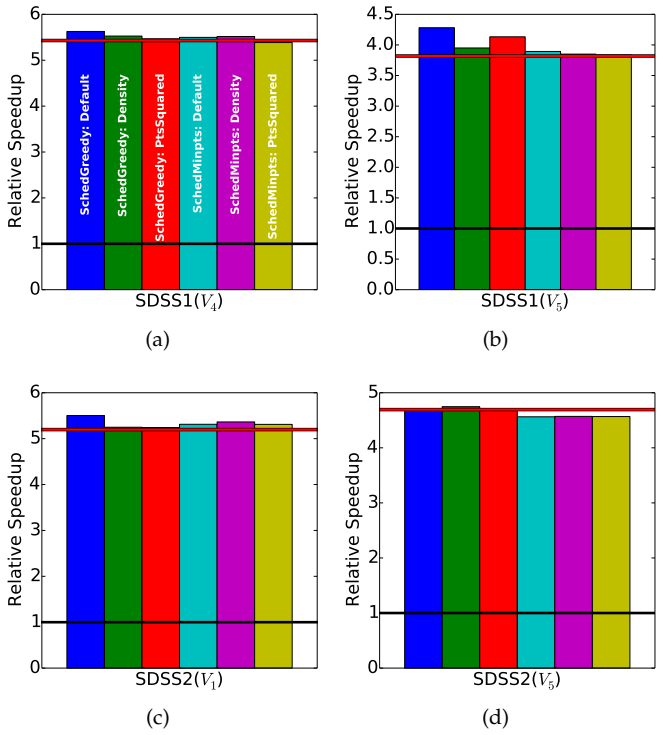


Fig. 12: Relative speedup of VARIANTDBSCAN applying all of the optimizations, i.e., scheduling algorithms, cluster reuse heuristics, index optimization ($r = 70$), and multi-threading ($T = 16$) for *SDSS-* in scenario $S3$ compared to the reference implementation. The left column (a, c) has a smaller number of values for ϵ than *minpts* (V_1, and V_4), and the converse is shown in the right column (V_5) (b, d). Values over the black horizontal line indicate a speedup relative to the reference implementation, whereas the red horizontal line shows the speedup when only using the index optimizations ($r = 70$, $T = 16$). The labels on the bars in (a) correspond to the same configurations in (b)-(d).

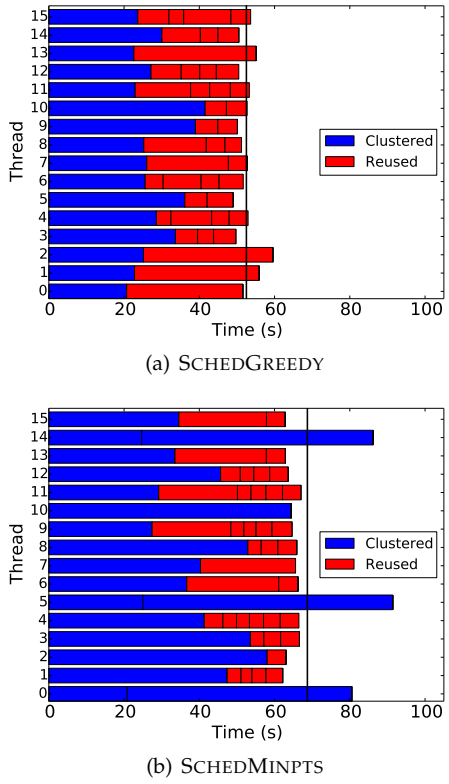


Fig. 13: Makespan of processing V_3 with CLUSDENSITY on *SW1* using (a) SCHEDGREEDY, and (b) SCHEDMINPTS (bars show variants clustered with and without data reuse). The black line shows the lower-bound time if all threads finish simultaneously (i.e., no cores are idle).

horizontal line). In Figure 11 we see that performance is not significantly degraded across scenarios that do not perform well (predominantly those that do not use CLUSDENSITY); we would observe a slowdown if any of the bars were significantly below the respective red horizontal lines. We show the relative speedup over the reference implementation on the *SDSS-* datasets in Figure 12. As the dataset is more uniform than the *SW-* datasets, the number of points are more evenly distributed amongst clusters in the *SDSS-* datasets; therefore, there are fewer good clusters that can be reused. This also results in more cluster candidates for expansion that are destroyed. We find that in contrast to the *SW-* datasets, the *SDSS-* datasets cannot efficiently exploit data reuse. Therefore, exploiting data reuse is largely a function of data density and distribution, favoring spatially skewed datasets.

Since 57 variants are being clustered in $S3$, towards the end of the computation, some threads will remain idle, as there will be no more variants to be clustered. Additionally, given that when using CLUSDENSITY, SCHEDGREEDY outperforms SCHEDMINPTS in 6 of 8 instances on the *SW-* datasets (Figure 11), an interesting question is to what extent this is a function of prioritizing variants, or due to resource underutilization. To understand the source of the differences in performance between the scheduling heuristics, we examine a scenario where the performance is significantly different.

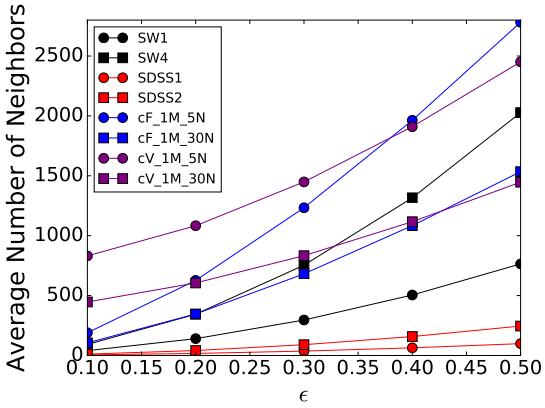
Figure 13 shows a comparison of the makespans that contrast the two scheduling heuristics when processing V_3 with CLUSDENSITY on *SW1* in $S3$ (Figure 11 (b)). Overall, as expected, reusing data with VARIANTDBSCAN (red bars) requires less time than when clustering from scratch (blue bars). Comparing Figure 13 (a) and (b), three more variants are clustered from scratch (shown in blue) when using SCHEDMINPTS, as it attempts to cluster a wider range of ϵ values to improve data reuse, and this is a function of the parameters in V_3 of $S3$. If the initial number of variants clustered from scratch were $\leq T = 16$, then SCHEDGREEDY and SCHEDMINPTS would achieve similar levels of performance. Comparing each makespan to the lower-bound time, assuming no cores are idle, the slowdown for SCHEDGREEDY and SCHEDMINPTS is 13.5% and 33.0%, respectively. Since SCHEDGREEDY performs well across all experimental scenarios (Figure 11), and is less sensitive to the parameter distribution (unlike SCHEDMINPTS, Figure 13), we conclude that SCHEDGREEDY is a more robust scheduling heuristic. In summary, using CLUSDENSITY, VARIANTDBSCAN is between 727% (*SW4*, V_2) and 2209% (*SW2*, V_1) faster than the reference implementation on *SW-*

datasets.

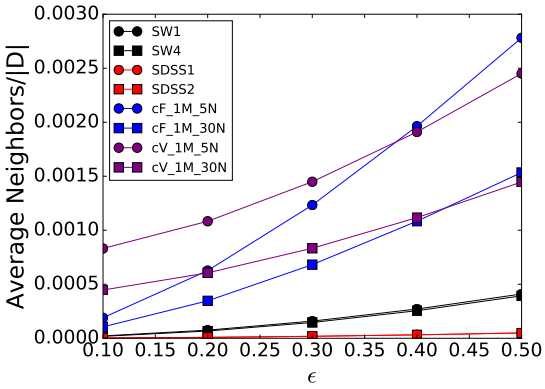
5.6 Discussion

In Section 5.5 we showed that the *SW-* class of datasets significantly benefit from data reuse (Figure 11) whereas the *SDSS-* class does not, but still achieves a performance improvement through efficient indexing (Figure 12). Therefore, it may be useful to know under which conditions a given dataset will benefit from data reuse in VARIANTDBSCAN. The performance of DBSCAN (and VARIANTDBSCAN) is partially a function of the average number of neighbors within ϵ of the points in the dataset. This is because the index returns a candidate set that is refined with distance calculations to produce those neighbors within ϵ. The larger this intermediate set is on average, the greater the computational and memory load of the respective algorithms. Datasets that are more uniformly distributed, have a significant amount of noise, or have fewer well-defined clusters will have fewer neighbors within ϵ on average. Whereas a denser dataset will have more neighbors within ϵ on average. Thus, for denser datasets, or datasets with well-defined clusters, there is a larger opportunity to exploit data reuse (by avoiding the overheads) than in more uniformly distributed datasets. Figure 14 (a) shows the average number of neighbors within ϵ for 8 datasets in the four dataset classes (*SW-*, *SDSS-*, *cF-*, *cV-*). The synthetic datasets all have 10^6 points, and represent cases where the clusters are well-defined. In the *cF_1M_5N* and *cV_1M_5N* datasets (both 5% noise), the average number of points within ϵ is mucher higher than the real-world datasets, even though the datasets are smaller. *cF_1M_30N* and *cV_1M_30N* have 30% noise, and have on average fewer neighbors within ϵ in comparison to *cF_1M_5N* and *cV_1M_5N*. Comparing the real-world datasets that have a similar number of points, from Figure 14 (a) we see that the *SW1* and *SW4* have significantly many more neighbors than the *SDSS1* and *SDSS2* datasets, respectively. This explains why in Figure 11 there is a performance improvement above the index-only (red line) speedup, but data reuse is limited in Figure 12. Furthermore, this shows why the performance owing to data reuse in Figure 9 is much higher for the synthetic datasets than *SW1*. The absolute average number of neighbors is not particularly useful when determining whether data reuse is likely, so we normalize this by dividing by the size of the dataset in Figure 14 (b). Since *SW1* and *SW4* both take advantage of data reuse, we can see that for this range of ϵ values ($0.1 - 0.5$), a given dataset can achieve performance improvements when the normalized average number of neighbors is $\gtrsim 0.0002$. Values above this should significantly improve performance through data reuse, as demonstrated in the synthetic datasets.

We have focused on 2-D datasets because they are used in the two domain science areas of geoscience and astronomy that we are interested in here. However, VARIANTDBSCAN may be useful in higher dimensions as well. The performance of many indexing schemes degrade with increasing dimensionality [32]. Tree indexes visit a larger fraction of nodes with increasing dimension, which may eventually degrade into a brute-force search. Assuming that a given dataset is not too sparse, having a similar normalized average number of neighbors as *SW1* or *SW4* in Figure 14 (b), then the performance gain of reusing data between variants would be similar to the level shown in Figure 11. However, since an increase in dimensionality would require a more exhaustive index search, the gain from data reuse is expected to be greater than for 2-D datasets as all of the reused data points would avoid these comparatively more expensive index searches.



(a)



(b)

Fig. 14: (a) The average number of neighbors of all of the points in a given dataset vs. ϵ. The larger the value of ϵ, the greater number of neighbors are found within the ϵ radius. (b) The average number of neighbors vs. ϵ in (a) divided by the size of each dataset.

Other architectures are promising for VARIANTDBSCAN. The GPU could be used in two scenarios. One scenario could perform the entire algorithm on the GPU and reuse results between variants by leaving the results of a clustering run in global memory, and begin the following clustering execution by using a similar algorithm to the one presented in this work. Another scenario would be to use a hybrid approach that splits the computation between the CPU and GPU, such as performing all of the ϵ-neighborhood searches on the GPU in parallel and sending these results to the CPU for subsequent clustering. ϵ-neighborhood searches around a point can be executed independently to take advantage of the many-core architecture. Both approaches would rely on efficient GPU-friendly indexing schemes, and a batching scheme to accommodate larger datasets that overlaps computation with communication to obviate

the host-GPU memory transfer bottleneck. Furthermore, for larger datasets that do not fit within one node, a number of the existing optimizations from the distributed memory DBSCAN algorithms can be employed. However, given that some nodes will attain higher data reuse than others between variants, new optimizations to reduce load imbalance will be needed.

6 Conclusion

Variant-based parallelism extends the available portfolio of parallel optimization techniques. Given the ever increasing volume of scientific datasets, this article shows how this direction can be exploited to significantly enhance clustering throughput and subsequent scientific analyses.

Our approach aims to scale concurrent clustering of a dataset where each instance executes with different parameters. To achieve an increase of clustering throughput, we develop a range of optimizations as follows. Optimized indexing mitigates the memory bottleneck and reduces computation and memory pressure. Reuse of results from previously executed variants reduces computation and data movements. Intelligently scheduling the order of variant execution further increases the reuse potential of intermediate results. Combining all of our techniques significantly outperforms a sequential reference implementation across both synthetic and real-world datasets. We also find that in cases where low data reuse occurs between variants, the overhead of reusing data is not prohibitive in comparison to clustering a variant from scratch.

The performance improvements achieved in this work also have a direct impact on concrete science applications; for example, future natural hazards monitoring systems could be equipped with a more efficient detection of space weather events that can lead to large-scale blackouts of our power grid.

Interesting directions for future work can leverage GPU parallelism in hybrid implementations for the ϵ-neighborhood searches. In addition, further work is needed to explore potential optimizations that are less sensitive to the data density and distribution of the input datasets.

Acknowledgments

We thank the anonymous reviewers for their useful comments and suggestions. We acknowledge support from NSF ACI-1442997.

References

[1] D. Shreiner and The Khronos OpenGL ARB Working Group, *Severe Space Weather Events–Understanding Societal and Economic Impacts: A Workshop Report*. National Academies Press, 2009.

[2] A. Coster and T. Tsugawa, "The Impact of Traveling Ionospheric Disturbances on Global Navigation Satellite System Services," in *Proc. of URSI General Assembly*, 2008.

[3] G. Occhipinti, L. Rolland, P. Lognonné, and S. Watada, "From Sumatra 2004 to Tohoku-Oki 2011: The systematic GPS detection of the ionospheric signature induced by tsunamigenic earthquakes," *Journal of Geophysical Research: Space Physics*, vol. 118, no. 6, pp. 3626–3636, 2013.

[4] S. Alam *et al.*, "The Eleventh and Twelfth Data Releases of the Sloan Digital Sky Survey: Final Data from SDSS-III," *The Astrophysical Journal Supplement Series*, vol. 219, Art. ID 12.

[5] A. Tramacere and C. Vecchio, "γ-ray DBSCAN: a clustering algorithm applied to Fermi-LAT γ-ray data. I. Detection performances with real and simulated data," *Astronomy & Astrophysics*, vol. 549, p. A138, Jan. 2013.

[6] S. Dehghan and M. Johnston-Hollitt, "Clusters, Groups, and Filaments in the Chandra Deep Field-South up to Redshift 1," *Astronomical Journal*, vol. 147, p. 52, Mar. 2014.

[7] 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*, 1996, pp. 226–231.

[8] V. Pankratius, J. Li, M. Gowanlock, D. M. Blair, C. Rude, T. Herring, F. Lind, P. J. Erickson, and C. Lonsdale, "Computer-Aided Discovery: Towards Scientific Insight Generation with Machine Support," *IEEE Intelligent Systems*, vol. 31, pp. 3–10, 2016.

[9] M. Gowanlock, D. M. Blair, and V. Pankratius, "Exploiting Variant-Based Parallelism for Data Mining of Space Weather Phenomena," in *Proc. of the 30th IEEE Intl. Parallel & Distributed Processing Symposium*, 2016, pp. 760–769.

[10] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1984, pp. 47–57.

[11] J. Gan and Y. Tao, "DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation," in *Proc. of the 2015 ACM SIGMOD Intl. Conf. on Management of Data*, 2015, pp. 519–530.

[12] D. Arlia and M. Coppola, "Experiments in Parallel Clustering with DBSCAN," in *Proc. of Euro-Par 2001*, 2001, pp. 326–331.

[13] S. Brecheisen, H.-P. Kriegel, and M. Pfeifle, "Parallel Density-Based Clustering of Complex Objects," in *Advances in Knowledge Discovery and Data Mining*, 2006, vol. 3918, pp. 179–188.

[14] M. Chen, X. Gao, and H. Li, "Parallel DBSCAN with Priority R-tree," in *The 2nd IEEE Intl. Conf. on Information Management and Engineering*, April 2010, pp. 508–511.

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

[16] M. A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, and A. Choudhary, "A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-set Data Structure," in *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 62:1–62:11.

[17] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey, "Pardicle: Parallel approximate density-based clustering," in *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 560–571.

[18] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha, "G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering," *Procedia Computer Science*, vol. 18, no. 0, pp. 369 – 378, 2013, 2013 Intl. Conf. on Computational Science.

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

[20] B. Welton, E. Samanas, and B. P. Miller, "Mr. Scan: Extreme Scale Density-based Clustering Using a Tree-based Network of GPGPU Nodes," in *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 84:1–84:11.

[21] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther, "Density-based clustering using graphics processors," in *Proc. of the 18th ACM Conf. on Information and Knowledge Management*, 2009, pp. 661–670.

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

[23] M. Kryszkiewicz and P. Lasek, "TI-DBSCAN: Clustering with DBSCAN by Means of the Triangle Inequality," in *Proc. of the 7th Intl. Conf. on Rough Sets and Current Trends in Computing*, 2010, pp. 60–69.

[24] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey, "Bd-cats: Big data clustering at trillion particle scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 6:1–6:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807616

[25] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," in *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, 1999, pp. 49–60.

[26] M. Gowanlock and H. Casanova, "In-Memory Distance Threshold Queries on Moving Object Trajectories," in *Proc. of the Sixth Intl. Conf. on Advances in Databases, Knowledge, and Data Applications*, 2014, pp. 41–50.

[27] V. Pankratius, A. Coster, J. Vierinen, P. Erickson, and B. Rideout, "GPS data processing for scientific studies of the earth's atmosphere and near-space environment," *To appear in Encyclopedia of Geographical Information Systems, Shashi Shekhar, Hui Xiong (Eds.), Springer Verlag*.

[28] Y. Otsuka, K. Suzuki, S. Nakagawa, M. Nishioka, K. Shiokawa, and T. Tsugawa, "GPS observations of medium-scale traveling ionospheric disturbances over Europe," in *Annales Geophysicae*, vol. 31, no. 2, 2013, pp. 163–172.

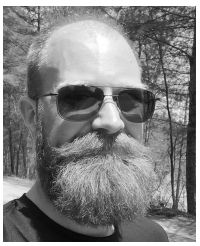
[29] ftp://gemini.haystack.mit.edu/pub/informatics/dbscandat.zip, accessed 21-January-2016.

[30] E. Januzaj, H.-P. Kriegel, and M. Pfeifle, "DBDC: Density Based Distributed Clustering," in *Proc. of the 9th Intl. Conf. on Advances in Database Technology*, vol. 2992, 2004, pp. 88–105.

[31] L. M. Collins and C. W. Dent, "Omega: A general formulation of the rand index of cluster recovery suitable for non-disjoint solutions," *Multivariate Behavioral Research*, vol. 23, no. 2, pp. 231–242, 1988.

[32] J. Kim, W. K. Jeong, and B. Nam, "Exploiting massive parallelism for indexing multi-dimensional datasets on the gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2258–2271, Aug 2015.

Michael Gowanlock received the B.Sc., and M.Sc. degrees from Trent University in Peterborough, Canada, in 2008, and 2010 respectively, and the Ph.D. degree from the University of Hawai'i at Mānoa in Honolulu, U.S.A., in 2015. He is currently a Postdoctoral Associate at MIT Haystack Observatory. His research interests are in the areas of parallel computing and astrobiology.



David M. Blair received a B.S. in Geological Sciences from Case Western Reserve University in 2008, and a Ph.D. in Earth, Atmospheric, and Planetary Sciences from Purdue University in 2015. He is currently a scientific computing coordinator at Brown University. His research interests include planetary geology and geophysics, space exploration, and geoscience informatics.

Victor Pankratius received a doctorate in 2007 and a Habilitation in computer science from Karlsruhe Institute of Technology, Germany, in 2012. His research interests include parallel computing and multicore software engineering, as well as their application in new areas of Astroinformatics and Geoinformatics. Currently is a research scientist at Massachusetts Institute of Technology, Haystack Observatory, where he leads the Astro-&Geo-Informatics group.