Exploiting Variant-Based Parallelism for Data Mining of Space Weather Phenomena

Michael Gowanlock, David M. Blair, Victor Pankratius
Massachusetts Institute of Technology, Haystack Observatory
[*mgowanlock, dblair, vpankratius*]*@haystack.mit.edu*

***Abstract*—This paper studies a form of parallelism termed variant-based parallelism, which exploits commonalities and reuse among variant computations in order to improve multithreading scalability. The problem is motivated by space weather studies that aim to identify changes in the Earth's ionosphere caused by auroral activity, tsunamis, and earthquakes. Today it is common to execute cluster algorithm variants with different parameters in order to determine which ones best explain phenomena in empirical data. We propose a novel approach and a set of optimizations to maximize throughput in such clustering algorithms. This is achieved by executing multiple clustering algorithm variants in parallel and developing efficient approaches to concurrently cluster data and maximize the reuse of results from completed variants. We present evaluations on real-world space weather datasets with up to 5 million ionospheric total electron content data points as well as synthetic datasets with up to a million data points. Results show a 1101% performance improvement due to indexing tailored for variant-based clustering, and a 2209% performance improvement when applying all of our proposed optimizations. Our optimizations enable new approaches in computer-aided discovery and could enable the short run times required for early warning systems for natural hazards.**

***Keywords*-Computer-Aided Discovery, Data Mining, DBSCAN, Parallel Clustering.**

I. Introduction

Scientific data volumes in the geosciences are drastically increasing due to a multitude of sensor networks deployed across the planet. To cope with such volumes, domain scientists would benefit from computer-aided discovery and data mining tools to help identify new phenomena. However, many data mining approaches are domain-independent, leading to automatically identified features which lack a true physical meaning. Scientists therefore resort to creating parameterized models based on various hypotheses, which are then executed with different parameter values to determine which model best describes reality. Even though this problem can be approached via embarrassingly parallel executions, doing so potentially misses many opportunities for speedups via data reuse (e.g. intermediate results).

This paper studies the aforementioned form of parallelism, here termed variant-based parallelism due to the use of commonalities and data reuse across similar computations. We use clustering of space weather data [1] as a vehicle to study this problem in a real-world context. In particular, our work is motivated by studies of the Earth's ionosphere, which consists of charged particles from about 90–1000 km altitude [2]. The ionosphere can delay signals received from satellites (e.g. GPS satellites [3]) at different frequencies depending on the Total Electron Content (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 [4], and space-based phenomena such as solar coronal mass ejections [1]. The application of data mining techniques such as clustering helps identify regions of high TEC values that propagate in a wave-like fashion in the form of Traveling Ionospheric Disturbances (TIDs), which can disrupt terrestrial and space-based communication or cause breakdowns of the power grid [1], [5]. Efficient data mining using model variants therefore has high societal relevance for natural hazards monitoring, and is also relevant to other application areas such as the detection of γ-rays [6].

Variant-based parallelism provides a systematic approach to enhance parallel scalability and clustering throughput. Our work leverages Density-Based Spatial Clustering of Applications with Noise (DBSCAN [7]) to find patterns of arbitrary shape as well as outliers in ionospheric spatiotemporal data. Due to the nature of the application, we prioritize clustering throughput rather than minimizing the response time of a single clustering execution.

In addition, we make the following novel contributions:
• We introduce VARIANTDBSCAN, a parallelization technique 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 trade increased computation for decreased memory accesses. This is due to VARIANTDBSCAN being memory-bound in two dimensions. We show that indexing is essential for efficient variant-based parallel clustering.
• We advance reuse strategies for cluster results between variants that are predicated on selecting relevant data from completed variants. We propose two data reuse heuristics.
• We propose two scheduling methods to improve clustering throughput. Our online scheduling problem arises as a function of the order in which variant clustering computations are started and completed, which determines how and when the data can be reused.
• We present evaluations on four real-world space weather

datasets and twelve synthetic datasets.

The paper is organized as follows. Section II describes the problem statement and algorithmic background for our application context. Section III outlines related work. Section IV introduces efficient indexing and data reuse approaches, as well as novel scheduling heuristics for variant execution. The experimental evaluation is detailed in Section V, and Section VI presents the conclusion.

II. Problem Statement and Algorithmic Background

Space weather data for the ionosphere commonly 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 1).

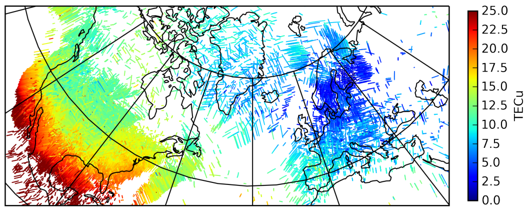


Figure 1: Example of a Total Electron Content (TEC) map of the Earth's ionosphere, obtained through GPS satellite signal processing [2]. The data shown is that of dataset *SW1*, as described in Section V-A.

A. Problem Statement

We present a new method for exploiting variant-based parallelism and make 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.

B. Background of the Clustering Algorithm

We briefly outline the principles of the DBSCAN clustering algorithm, which is described in detail in [7]. This algorithm is chosen as the core of this study because of its suitability for ionospheric feature detection.

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 [8]). The algorithm returns a series of clusters (an individual set of points belonging to a cluster is denoted as C), and the set of points labeled as noise.

Due to space limitations we comment only on the key points of the algorithm. After initializing the visited set, cluster set, and noise set, the algorithm loops over all points in D that have not yet been visited. The set of neighbors within ϵ of an unvisited point p, are obtained by searching the points in D using the R-tree [8]. Note that a brute-force approach at this step would require examining all of the points in D, thus the algorithm would have a time complexity of $O(|D|^2)$; however, it is claimed [7] that using

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

```
1: DBSCAN($D$, $\epsilon$, minpts, Rtree $T$)
2: visitedSet← $\emptyset$; clusterSet← $\emptyset$; noiseSet← $\emptyset$
3: for all $p \in D | p \notin$visitedSet do
4:     $C \leftarrow \emptyset$ ; visitedSet ← visitedSet $\cup \{p\}$
5:     $N \leftarrow$ NeighborSearch($p, \epsilon, T$)
6:     if $|N| <$ minpts then noiseSet ← noiseSet $\cup \{p\}$
7:     else
8:         $C \leftarrow C \cup \{p\}$; clusterSet ← clusterSet $\cup \{p\}$
9:         for all $i \in N$ do
10:            $N \leftarrow N \backslash i$
11:            if $i \notin$ visitedSet then
12:                visitedSet ← visitedSet $\cup \{i\}$
13:                $\hat{N} \leftarrow$ NeighborSearch($i, \epsilon, T$)
14:                if $|\hat{N}| \geq$ minpts then $N \leftarrow N \cup \hat{N}$
15:            if $i \notin$ clusterSet then
16:                $C \leftarrow C \cup \{i\}$; clusterSet ← clusterSet $\cup \{i\}$
```

a spatial or spatiotemporal index such as the R-tree can reduce the complexity to $O(|D|\log|D|)$, although this is disproved in [9]. Next, the algorithm determines if a point is a core point and adds it to a current cluster, or marks it as noise. Each of the directly density-reachable neighbors of a point are defined by the set N. Neighbors of points $i \in N$ are candidates to expand the cluster.

III. Related Work

Efficiency improvements of DBSCAN for single cluster execution, i.e., without variant parallelism, have been proposed in [10], [11], [12], [13], [14], [15], [16]. In [10], 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 [11] 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 [17], and 128 compute nodes with an emphasis on load balancing to address skewed datasets [13]. A distributed-memory implementation of DBSCAN using the disjoint-set data structure is presented in [14], and an approximate version in [15]. The work with the greatest scalability uses GPUs [18], and other GPU implementations are discussed in [16], [19]. Spatiotemporal applications are discussed in [20], and [21] attempts to reduce the search space by exploiting the triangle inequality property.

A related algorithm that finds good parameter values for DBSCAN is OPTICS [22]. 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.

IV. 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.

A. Indexing Approach for Shared Memory Algorithms

We employ an R-tree spatial index [8] 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 [23].

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($p$, $\epsilon$, Rtree $T$)
2:     overlappingMBBSet ← $\emptyset$; candidateSet ← $\emptyset$
3:     MBB ← generateMBB($p$, $\epsilon$)
4:     overlappingMBBSet ← $T$.search(MBB, $\epsilon$)
5:     candidateSet ← dataLookup(overlappingMBBSet)
6:     $N \leftarrow$ filterCandidateSet($p$, $\epsilon$, 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. 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.

B. 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 II-B.

Algorithm 3 The VariantDbscan Algorithm.

```
1: VariantDbscan($D$, $V$, Rtree $T_{high}$, Rtree $T_{low}$)
2: parallel for $v_i \in E$ do
3:     $C_v \leftarrow$ schedule($v_i$)
4:     if $C_v \neq \emptyset$ then
5:         destroyedSet $\leftarrow \emptyset$; visitedSet $\leftarrow \emptyset$
6:         clusterSeedSet $\leftarrow$ getSeedList($C_v$)
7:         for all $j \in$ clusterSeedSet do
8:             if $j \in$ destroyedSet then continue
9:             $C \leftarrow \emptyset$; $C \leftarrow C_v[j]$; visitedSet $\leftarrow C$
10:            MBB$\leftarrow$generateClusterMBB($C$)
11:            candidateSet $\leftarrow T_{high}$.search(MBB)
12:            outsidePointsSet $\leftarrow$candidateSet$\setminus C$
13:            for all $p \in$ outsidePointsSet do
14:                neighborSet $\leftarrow$NeighborSearch($p$, $v_i^{\epsilon}$, $T_{low}$)
15:                expandSet $\leftarrow$expandSet $\cup$ neighborSet $\cap$ C
16:                visitedSet $\leftarrow$visitedSet$\setminus$expandSet
17:            ExpandCluster($D$, $v_i^{\epsilon}$, $v_i^{minpts}$, $T_{low}$,
                   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 IV-A.

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 IV-D). 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 IV-A.

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 IV-C).

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 on 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 2 (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.

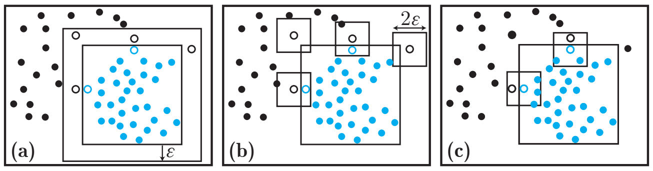


Figure 2: Illustration of lines 10-17 in Algorithm 3. Across (a)-(c), the blue points belong to a cluster. The blue unfilled points are the ones that are found by the algorithm that are used to expand the cluster. The unfilled black circles are candidates possibly within ϵ of points inside the preexisting 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 2 (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 IV-A). 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 2 (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 2 (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 2 (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 2 (c). EXPANDCLUSTER is similar to lines 9–16 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 ($D$, $\epsilon$, *minpts*, Rtree $T$, growPntsSet,
                  destroyedSet, $C$, $C_v$)
2: $N \leftarrow$ growPntsSet
3: for all $i \in N$ do
4:     $N \leftarrow N \backslash i$
5:     if $i \notin$ visitedSet then
6:         visitedSet $\leftarrow$ visitedSet $\cup \{i\}$; $\hat{N} \leftarrow$ NeighborSearch($i, \epsilon, T$)
7:         if $|\hat{N}| \geq$ *minpts* then $N \leftarrow N \cup \hat{N}$
8:     if $i \notin$ clusterSet then
9:         $C \leftarrow C \cup \{i\}$; clusterSet $\leftarrow$ clusterSet $\cup \{i\}$
10:    if $i \in C_v$ then
11:        id$\leftarrow$getOldClusterID($i$); destroyedSet$\leftarrow$destroyedSet$\cup$\{id\}
```

C. 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.

D. 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 IV-B). 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 3 (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 3 (b) and (c) illustrate two possible schedules resulting from this tree. From S_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 3 (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 3 (b)). If no variant can

$$\begin{tikzcd}
 & (0.2,28) \arrow[r] & (0.2,24) \arrow[r] & (0.2,20) & & \\
(0.2,32) \arrow[ur] \arrow[dr] & & (0.4,28) \arrow[r] & (0.4,24) \arrow[r] & (0.4,20) & \\
 & (0.4,32) \arrow[ur] \arrow[r] & (0.6,32) \arrow[r] & (0.6,28) \arrow[r] & (0.6,24) \arrow[r] & (0.6,20)
\end{tikzcd}$$

(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

Figure 3: (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 $T = 1$.

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 3). 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 3 (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 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).

V. EXPERIMENTAL EVALUATION

A. Datasets

We utilize 3 different classes of datasets to enable an evaluation under different scenarios (Table I). Class *SW-* consists of real-world space weather TEC datasets using the GPS processing methods described in [2]. Classes *cF-* and *cV-* are synthetic datasets created in order to conduct experiments 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. All datasets are available at [24].

Table I: Characteristics of Datasets

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

B. Experimental Methodology

We develop the multithreaded implementations using OpenMP in C++ using the O3 compiler optimization flag. The executions occur 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.

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 emerges. A heuristic [7] for selecting *minpts* finds 4 to be a good value; we therefore use 4 as the smallest value. Due to space constraints, we cannot list each variant in V in the form $v_i = (v_i^{\epsilon}, v_i^{minpts})$, so we 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 $V = \{(0.1, 1), (0.1, 2), (0.2, 1), (0.2, 2)\}$. We compare the performance of the implementations to a reference implementation that executes DBSCAN (Algorithms 1 and 2) with $T = 1$ and $r = 1$ (sequential without index optimization).

C. Efficient Indexing for Variant-Parallel Clustering

As described in Section IV-A, one challenge of executing multiple variants in parallel is the memory-bound nature of clustering from scratch with DBSCAN in 2-D. Therefore, we

Table II: 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

elect to index multiple points per MBB (r) in the R-tree. 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 II ($S1$). 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).

Figure 4 shows the relative speedup as the ratio of the response time of the reference implementation to the multithreaded implementation ($T = 16$ with various values of r) vs. each scenario in $S1$. We show $r = 1$ with $T = 16$ (blue bars), corresponding to no index optimization, but with each thread concurrently clustering a single variant. 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 number of MBBs which is calculated as $\lceil |D|/r \rceil$, depth of the R-tree, and the value of ϵ. We do not explore these factors individually, instead we determine good values of r empirically. We find that the range of good values of r is relatively consistent ($70 \leq r \leq 110$) across $S1$. We highlight those values in Figure 4. If $r = 1$ and $T = 16$, then the maximum speedup is $2.37\times$ (*cF_1M_5N*); however, by using efficient indexing, we achieve relative speedup values from $7.91\times$ on *cF_10k_5N* to $31.96\times$ on *cV_10k_30N*. The real-world *SW1* dataset with $r = 100$ is 1101% faster than the reference implementation. 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 are inhibited by memory-bound ϵ-neighborhood calculations.

D. *Efficient Data Reuse for Variant-Parallel Clustering*

We outline the efficiency of reusing cluster results between variants as described in Section IV-C, 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 III ($S2$).

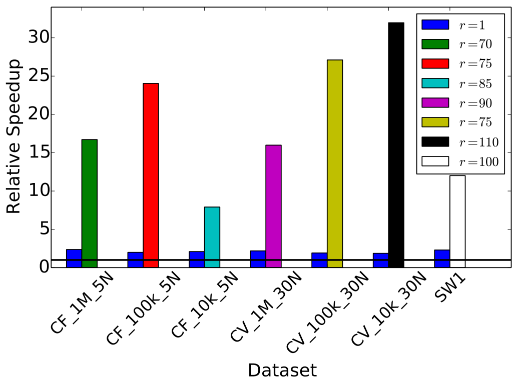


Figure 4: Relative speedup of VARIANTDBSCAN using $S1$ compared to the reference implementation. VARIANTDBSCAN is configured with $T = 16$ (each thread executing a single variant) and using good values for r (see text). Values over $y = 1$ show a performance gain.

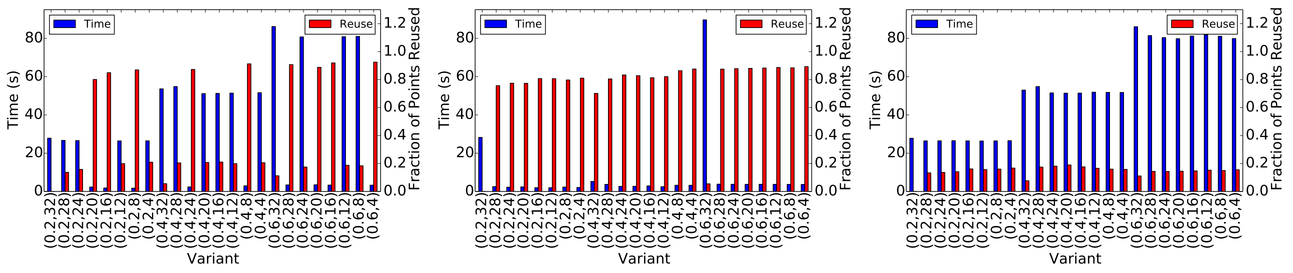
Table III: Scenario 2 ($S2$)

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 $A = \{0.2, 0.4, 0.6\}$, $B = \{4, 8, \dots, 32\}, |V| = 24$

In Figure 5, 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 5 (a)), we find that using the CLUSDENSITY (Figure 5 (b)) significantly reduces the response time across the individual variants. CLUSPTSSQUARED leads to poor performance (Figure 5 (c)).

To summarize results of all variants in Figure 5, Figure 6 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 7 (a); we plot the relative speedup as the ratio of the response time of the reference implementation to the response



(a) ClusDefault (b) ClusDensity (c) ClusPtsSquared

Figure 5: Response time (left y-axis) and fraction of points reused (right y-axis) vs. individual variant for *SW1* in $S2$ using the three cluster reuse schemes (a) ClusDefault, (b) ClusDensity, and (c) ClusPtsSquared. All variants are clustered with $T = 1$, and $r = 70$, with variant parameters (ϵ, *minpts*) shown below each bar.

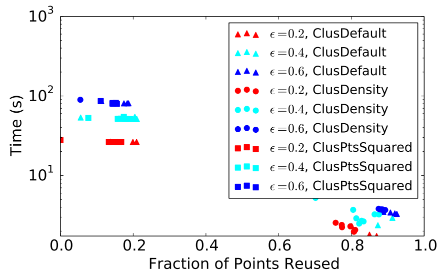


Figure 6: Response time vs. fraction of points reused, where values are grouped by ϵ family (color), and cluster reuse scheme (marker shape); response time and reuse values from Figure 5 (a)–(c) are shown (from *SW1* in $S2$). Response times are lower if sufficient data reuse occurs.

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 7 (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 [25], 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 and F is the cluster assigned to the point in VariantDbscan. The average quality score of a variant is the average of all of the quality scores of the individual points. Figure 7 (c) plots a selection of quality scores for the results in Figure 7 (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.

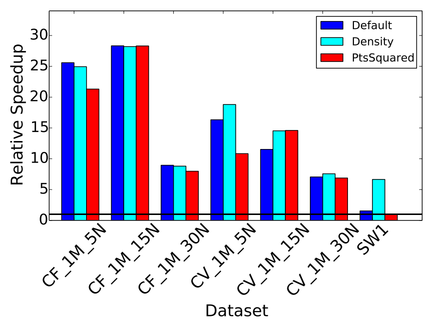
E. Combining Indexing, Data Reuse, and Scheduling

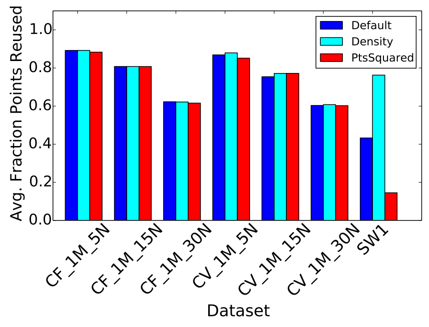
Above, we have shown that significant performance gains can be achieved using efficient indexing techniques and reusing data between variants. We combine indexing and result reuse in a multithreaded implementation that relies on assigning variants to threads based on the input parameters. To demonstrate 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 size of *SW4*, in that one instance, a different set of variants is employed. We utilize the datasets and parameters in Table IV ($S3$). 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. The maximum fraction of variants that can be reused for V_1, V_2, and V_3 is $f = \frac{|V|-T}{|V|} = \frac{57-16}{32} = 0.719$.

Table IV: Scenario 3 ($S3$)

Dataset (V)	V	A	B
SW1 (V_1, V_3)	V_1	$\{0.2, 0.3, 0.4\}$	$\{10, 15, \dots, 100\}$
SW2 (V_1, V_3)	V_2	$\{0.15, 0.25, 0.35\}$	$\{10, 15, \dots, 100\}$
SW3 (V_1, V_3)	V_3	$\{0.04, 0.06, \dots, 0.4\}$	$\{4, 8, 16\}$
SW4 (V_2, V_3)	$V = A \times B; \lvert V_1\rvert = \lvert V_2\rvert = \lvert V_3\rvert = 57$		

Figure 8 shows the relative speedup using the cluster reuse and scheduling algorithms on the four real-world datasets. Across all scenarios, ClusDensity (green and purple bars) outperforms ClusPtsSquared. In experimental scenarios


(a)


(b)

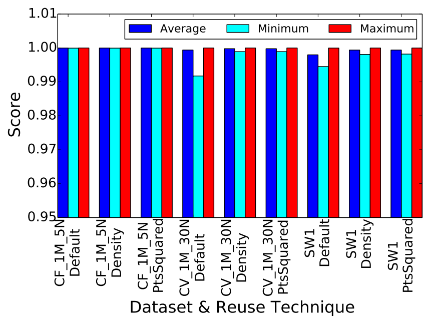

(c)

Figure 7: (a) Relative speedup of VARIANTDBSCAN with $S2$ 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). (c) Quality of results; see text for details.

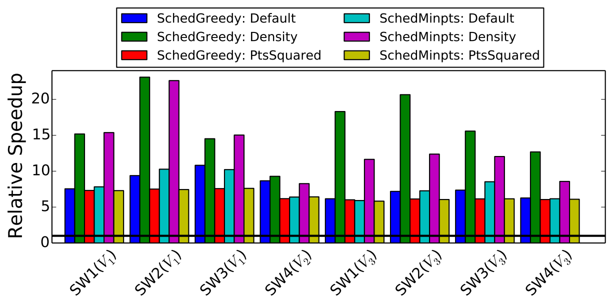
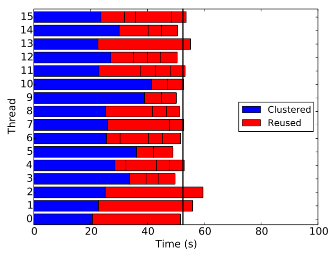


Figure 8: Relative speedup of VARIANTDBSCAN (using the scheduling and cluster reuse algorithms and $T = 16$) compared to the reference implementation, vs. each *SW*-dataset in $S3$. The left-most four scenarios have a smaller number of values for ϵ than *minpts* (V_1, and V_2), and the converse is shown in the four right-most scenarios (V_3).


(a) SCHEDGREEDY

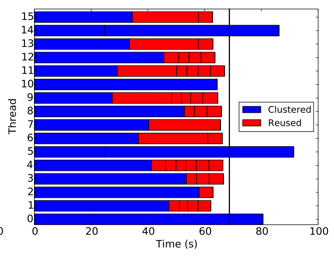

(b) SCHEDMINPTS

Figure 9: 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).

with CLUSDENSITY, SCHEDGREEDY outperforms SCHEDMINPTS in 6 of 8 instances. An interesting question is to what extent this is a function of prioritizing variants, or to what extent it is due to resource underutilization. We select a scenario where the performance is significantly different between scheduling heuristics.

Figure 9 shows a comparison of the makespans that contrast the two scheduling heuristics when processing V_3 with CLUSDENSITY on *SW1* in $S3$. Overall, reusing data with VARIANTDBSCAN (red bars) requires less time than when clustering from scratch (blue bars). Comparing Figure 9 (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 8), and is less sensitive to the parameter distribution (unlike SCHEDMINPTS, Figure 9), 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.

VI. CONCLUSION

Increasing volumes of scientific data are becoming more challenging to analyze, and scientists have to overcome the key problem of how to efficiently evaluate a variety of scientific models for a given dataset. Variant-based parallelism, as described in this paper, shows great promise for helping to overcome these types of problems. In space weather applications it could contribute towards accelerated natural hazard warning systems. Our approach allows for concurrent clustering of a dataset using multiple parameters. We maximize clustering throughput by mitigating the memory bottleneck through efficient indexing, reducing computation and memory pressure by reusing results of previously executed variants, and by optimizing the order of variant execution through scheduling heuristics. We also find that in the case where low data reuse occurs between variants, the overhead of reusing data is not prohibitive in comparison to clustering the variant from scratch. Our combination of techniques significantly outperforms a sequential implementation across a wide range of application scenarios and input parameters on real-world datasets.

Acknowledgments

We acknowledge support from NSF ACI-1442997. We thank Bill Rideout for providing us with the data used in our performance evaluation.

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] 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*.

[3] 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.

[4] 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.

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

[6] 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.

[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] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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 *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th Intl. Conf. on*, Dec 2011, pp. 473–480.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.