Mr. Scan: Extreme Scale Density-Based Clustering using a Tree-Based Network of GPGPU Nodes

Benjamin Welton, Evan Samanas, and Barton P. Miller
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
{welton,samanas,bart}@cs.wisc.edu

ABSTRACT

Density-based clustering algorithms are a widely-used class of data mining techniques that can find irregularly shaped clusters and cluster data without prior knowledge of the number of clusters it contains. DBSCAN is the most well-known density-based clustering algorithm. We introduce our version of DBSCAN, called Mr. Scan, which uses a hybrid parallel implementation that combines the MRNet tree-based distribution network with GPGPU-equipped nodes. Mr. Scan avoids the problems of existing implementations by effectively partitioning the point space and by optimizing DBSCAN's computation over dense data regions. We tested Mr. Scan on both a geolocated Twitter dataset and image data obtained from the Sloan Digital Sky Survey. At its largest scale, Mr. Scan clustered 6.5 billion points from the Twitter dataset on 8,192 GPU nodes on Cray Titan in 17.3 minutes. All other parallel DBSCAN implementations have only demonstrated the ability to cluster up to 100 million points.

1. INTRODUCTION

We investigate techniques for density-based clustering of multi-billion point datasets such as geospatial data. Specifically we have developed a clustering technique that uses a hybrid computing model combining large-scale multicast/reduction overlay networks operating with nodes equipped with high-end GPGPUs. This hybrid computation allows for clustering of extremely large datasets in an efficient manner. In this paper, we introduce a new clustering algorithm, Mr. Scan, and an end-to-end implementation of this algorithm that we show can efficiently scale to billions of points on a leadership class supercomputer.

Clustering is the act of classifying data points, where data points that are considered similar are contained in the same cluster and dissimilar points are in different clusters.

SC13 November 17-22, 2013, Denver, Colorado, USA

ACM 978-1-4503-2378-9/13/11 ...$15.00
http://dx.doi.org/10.1145/2503210.2503262.

Clustering helps researchers and data analysts gain insight into their data, e.g., identifying and tracking objects such as gamma-ray bursts in sky observation data [10], monitoring the growth and decline of forests in the United States [23] and identifying performance bottlenecks in large-scale parallel applications [12]. We focus on a type of clustering algorithm called density-based clustering, which classifies points into clusters based on the density of the region surrounding the point. Density-based clustering detects the number of clusters in a dataset without prior knowledge and is able to find clusters with non-convex shapes.

Datasets such as the Sloan Digital Sky Survey [2] and geolocated tweets from Twitter [3] are useful to cluster but are too large (i.e., billions of data points) to be practically computed on a small or medium-sized parallel computer (100's to 1000's of nodes) by any non-trivial clustering algorithm. These large data sizes require the largest-scale parallel systems that are in use today. However, there are few distributed density-based clustering algorithms designed to run on these large-scale systems. Existing distributed density-based algorithms typically reduce the quality of the output when compared to the single-node version, or they do not scale to the sizes needed for these datasets.

Mr. Scan is our implementation of the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) clustering algorithm [11]. DBSCAN is the most widely cited density-based clustering algorithm and has been shown to be well-suited for data analysis in many fields, e.g., the analysis of laser ablated material [26] and tracking population movement by use of geotagged photographs [17]. The benefits DBSCAN has over other clustering algorithms are that it has the ability to find irregularly shaped clusters, it distinguishes data points that are considered noise (i.e., points in low density regions) from clusters, and it is able to cluster data where the number of clusters in the dataset is not known in advance. These features come at a cost, since the computational complexity of DBSCAN is $O(n^2)$ where n is the number of points in the input dataset. This complexity is a result of the calculation of a n x n matrix containing the distances between all points. This matrix can be replaced with a spatial tree index which reduces the cost of distance calculations leading to an average case complexity of $O(n \log n)$.

Mr. Scan is the first implementation of DBSCAN that can scale up to 6.5 billion data points and the first distributed DBSCAN algorithm that incorporates the use of GPGPUs. It uses a programming paradigm that organizes processes

into a multi-level tree with an arbitrary topology. In this multi-level tree paradigm, DBSCAN calculations are done on the GPGPU leaf nodes and these results are combined on non-leaf nodes. Mr. Scan is also the first clustering algorithm to use this programming paradigm to our knowledge. Using this multi-level tree design we demonstrate the capability to cluster 6.5 billion points using 8,192 GPGPU nodes in 17.3 minutes.

The ability to cluster billions of data points with DBSCAN can only be realized if the key obstacles to scaling DBSCAN are overcome: load balancing, cluster merging, and distributing data advantageously. The running time of DBSCAN increases as a function of spatial density of the input data points, which causes a load imbalance when compute nodes contain regions of varying density. We modify DBSCAN to find the most dense regions and infer their membership in a cluster without evaluating the points inside these dense regions. Results from DBSCAN compute nodes must be merged accurately without requiring the entirety of each cluster. We resolve this by requiring a small, bounded number of representative points per cluster to perform a merge. Finally, data must be distributed in a manner that balances DBSCAN's clustering operation and the overhead of merging clusters. We achieve this with a heuristic that spatially decomposes the data into partitions to balance the merge overhead. Each partition contains roughly equal point counts to aid in balancing DBSCAN clustering time.

In Section 2 we describe the DBSCAN algorithm and discuss other methods that attempt to parallelize DBSCAN and other related work. Section 3 introduces the Mr. Scan algorithm and how it overcomes DBSCAN's scaling obstacles. Section 4 describes the experiments used to benchmark Mr. Scan using data from the microblogging service Twitter and the Sloan Digital Sky Survey. Section 5 presents and discusses the scaling results of both datasets. Finally, Section 6 presents our concluding thoughts.

2. BACKGROUND AND RELATED WORK

Due to DBSCAN's popularity among density-based clustering algorithms, optimization and parallelization of the algorithm has been widely studied [5]. We first explain the DBSCAN algorithm in detail, then present previous parallelization efforts that are most significant to the parallelization style of Mr. Scan along with the most scalable algorithms.

2.1 The DBSCAN Clustering Algorithm

DBSCAN clusters data points by density. Its notion of density comes from its two parameters known as *Eps* and *MinPts*. DBSCAN operates by finding the *Eps-neighborhood* of each point. The *Eps*-neighborhood of a point p is the set of points that are located within *Eps* distance of p. The point p is considered a core point if there are at least *MinPts* points in its *Eps*-neighborhood. All other points are classified as *non-core* points. Non-core points can have two distinctions: a *border point* or a *noise point*. A border point is a non-core point that contains at least one core point in its *Eps*-neighborhood, whereas a noise point does not.

A cluster is formed by the set of core and border points reachable from a particular core point. Once an unvisited core point is found, it is considered a new cluster along with its *Eps*-neighborhood. This cluster is expanded by finding the *Eps*-neighborhood of each point classified in the cluster until all points that are reachable from the first core point are found. For this reason, DBSCAN's clustering results can vary slightly if the order in which *Eps*-neighborhoods are discovered is changed.

The performance of the DBSCAN algorithm varies greatly based on the presence (or lack thereof) of a spatial index. DBSCAN without a spatial index is $O(n^2)$ in time complexity. This is due to not limiting the amount of points compared by the distance function. Without a spatial index all points in the dataset must be compared with each other to determine which points are core. A spatial index however reduces the number of points which must be compared by limiting the search to a smaller subset of points that are in the region of the point being queried. The average case complexity improves to $O(n \log n)$ by use of a spatial index (e.g., R*-tree or KD-tree).

2.2 Past Optimizations of DBSCAN

DBSCAN has been parallelized by multiple past projects. One of the first was PDBSCAN [29]. This algorithm used a distributed R*-tree to partition the dataset among many compute nodes. Distributed R*-trees partition data but they replicate the entire index on each node. If a neighborhood query included an area of the dataset that resides on different node, the node that started the query must send a message to obtain the data. This algorithm showed linear speedup up to 8 nodes, but the amount of messages sent grew super-linearly in most cases, which hampered its scalability. Another algorithm, DBDC [15], assumes that the dataset to cluster is already distributed among the compute nodes. DBDC pioneered the idea of using many slave nodes to cluster a portion of the dataset and merging the final result at a master node, and also the idea of sending a smaller number of points to represent the locally found clusters to increase scalability. This technique scaled linearly up to 30 nodes, but the manner in which representative points were picked decreased the quality of the clustering output when compared to traditional DBSCAN, and the assumption of already distributed data further degraded quality.

Recently, there have been some Map/Reduce implementations of DBSCAN, MR-DBSCAN [14] and DBSCAN-MR [9]. MR-DBSCAN was able to cluster 1.9 billion points of 2D taxi-cab traces in approximately 5,800 seconds. However, the authors preprocessed the data prior to running DBSCAN to reduce the negative effects of high-density regions and did not account for this preprocessing time in their results. Also, the parameters for MR-DBSCAN's runs were chosen solely for speed and not for quality of the data analysis [13]. Aside from these issues, neither of the Map/Reduce implementations showed near-linear speedup nor the ability to scale weakly and only demonstrated their algorithms on up to 12 multi-core nodes. The highest scale for DBSCAN that we have seen prior to our current paper is from PDS-DBSCAN [24], which was able to achieve a 5,765x speedup on 8,192 distributed cores on a 72 million point astronomy dataset with a random distribution. This algorithm moved away from the master-slave approach and used a distributed disjoint-set data structure. Speedup decreased beyond 8,192 cores because of a large increase in messages sent between cores to access and update the data structure.

Several algorithms attempted to improve the single-core performance of DBSCAN. TI-DBSCAN [18] uses the tri-

angle inequality. The input dataset is sorted to determine a point's *Eps*-Neighborhood, which is similar to the way our GPU implementation of the algorithm uses its KD-tree. Another version of DBSCAN [19] attempts to remove core points early from the DBSCAN calculation. This idea is similar to Mr. Scan's dense box optimization, but their method appears that it would change the result of DBSCAN significantly, even though the authors do not comment on this effect in the paper. In comparison, Mr. Scan's dense region calculation has an extremely small impact on quality when compared to traditional DBSCAN.

3. THE MR. SCAN ALGORITHM

Mr. Scan is a parallel implementation of the DBSCAN algorithm with four phases: partition, cluster, merge, and sweep. Mr. Scan starts with a single input file on a parallel file system and writes a file of the points included in a cluster and their cluster IDs as output. The input points are contained in a single binary or text file. Each input point has a unique ID number, coordinates, and an optional weight that can be used for analysis of the clustered output.

Figure 1 gives an overview of the Mr. Scan algorithm. In the partition phase, the input file is read by a partitioner that creates one partition per clustering process. The input file can contain billions of points and reach sizes up to 300 GB, so the partitioner is distributed using MRNet [25] to parallelize this step. Each worker process of the partitioner writes the partitions to the file system in parallel to prepare for the next phase. The cluster phase is started by launching a second MRNet tree of processes. This tree has a user-specified number of levels of intermediate processes and one leaf process for each partition. Each Mr. Scan leaf process clusters its assigned partition using our GPGPU version of DBSCAN and picks a small, constant set of points to represent each cluster. The representative points are sent to the intermediate processes to start the merge phase where the clusters are progressively merged by each level of intermediate processes until they reach the root. The root performs the final merge and assigns a global ID to each cluster. Mr. Scan then starts the sweep phase, and sends the global cluster IDs down the tree, where each point is identified with its correct global cluster ID and written to the output file in parallel by the leaf processes.

In this section, we describe the design of each of Mr. Scan's phases, and how they solve the three challenges in scaling DBSCAN: load imbalance, distributed merge, and data distribution.

3.1 Partitioner

In addition to the basic goal of dividing an input dataset into n partitions given n leaf processes, we have three main goals for the design of Mr. Scan's partitioner. First and most important, the partitioner must produce partitions capable of yielding a correct DBSCAN result when clustered and merged. Second, the output partitions must have roughly equal computational costs when being clustered. The partitioner does not need to produce perfectly balanced partitions, since the dense box optimization described in Section 3.2.3 plays a large role in controlling load balance. However, the partitioner does hold some responsibility for controlling the load balance during the cluster phase. Third, the partitioner must perform well enough to avoid becoming a significant portion of Mr. Scan's overall time, especially as the size of the input dataset grows. This means as few operations as possible should be I/O bound, and leads to the design decision of distributing the partitioner among many nodes. We will discuss how we meet these three goals below.

3.1.1 Correctness

We define a correct partitioning as a set of partitions that merge to form a global clustering that is equivalent to executing non-parallel DBSCAN on the entire input dataset. It is impossible for this definition to be satisfied when the partitions each contain a disjoint subset of the input dataset, because any point whose *Eps*-neighborhood includes points in a different partition than its own would return an incomplete *Eps*-neighborhood [29] [14] [9]. To address this, we add a shadow region to each partition. The *shadow region* is the set of points not already included in the partition that lie *Eps* distance from the partition's boundary. A *shadow point* is a point that lies in a shadow region with respect to a partition, and a *partition point* is a point already included in the partition. When the shadow region is added to a partition, each partition point's *Eps*-neighborhood contains only partition points or shadow points, and thus is complete within the partition.

3.1.2 Partitioning Algorithm

The second goal of the partitioner is to control load balance in the cluster phase by creating computationally equivalent partitions. Therefore, we must have a way to estimate a partition's computational cost to DBSCAN. Mr. Scan uses the partition's point count for this estimation. We have established in Section 1 that DBSCAN's performance is largely dependent on the spatial density of points and not pure point count, so point count is not an ideal measure for an unmodified DBSCAN implementation. We use point count instead of density because our modified DBSCAN's performance is positively impacted by density, so point count is a more accurate measure in this case.

A DBSCAN partitioning algorithm must output DBSCAN partitions that are not only correct, but profitable. We denote a partition as profitable if it meets two constraints. The first is that the longest distance across the partition must be greater than *Eps*. If the distance is less than *Eps*, the *Eps*-neighborhood of each point is guaranteed to include each point in the partition, and there is no need to invoke DBSCAN. The second constraint is that each partition must contain at least *MinPts* points. Otherwise, we would already know that every point is a noise point, and DBSCAN is not needed.

We fulfill the first constraint in our algorithm by constructing the input dataset as a grid where each cell of the grid is the same size, *Eps* x *Eps*. Each partition is made up of at least one grid cell, which ensures that each partition's longest distance across is greater than *Eps*. Thus, the shadow region for each partition simply becomes the set of grid neighbors not already in the partition.

The partitioning algorithm starts by setting the *target size* of the partitions, which is an equal share of the input points. Since our partitioning algorithm forms partitions from regular grid cells that contain varying amounts of points, it is generally not possible to form partitions that are even roughly similar in their point counts when partitioning non-uniform data. Large grid cells do not pose a problem for load balancing in Mr. Scan because of our dense box opti-

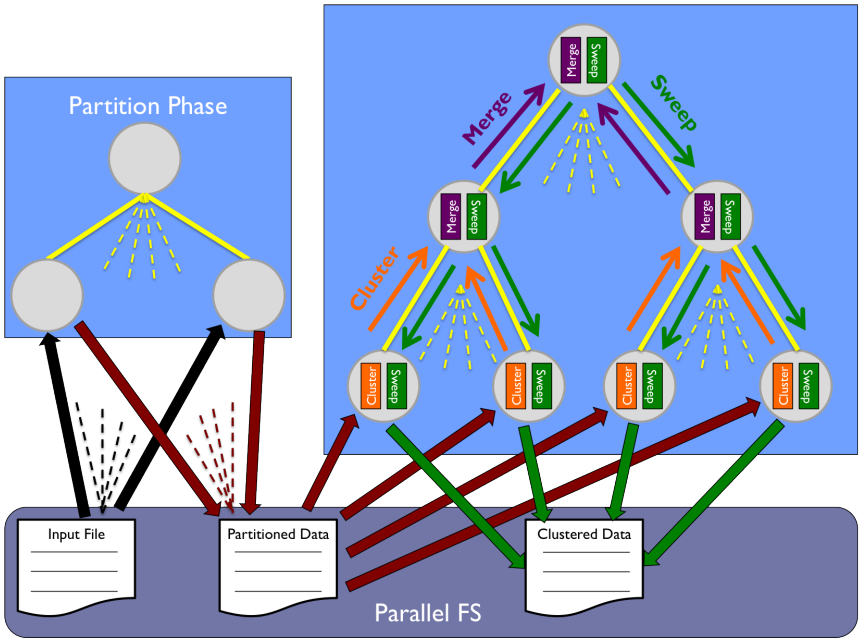


Figure 1: The Mr. Scan algorithm

mization described in Section 3.2.3.

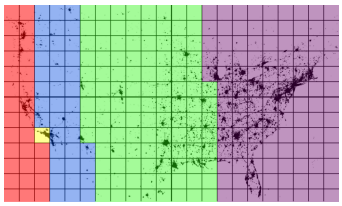
For simplicity, we describe the algorithm for forming partitions assuming that the input dataset's grid is 2D, however it can be extended to an arbitrary dimension. We iterate over all cells in the input grid first along the y axis, and then along the x axis. Partitions are formed sequentially through this iteration. Grid cells are added to a partition until the addition of the cell would cause the partition to exceed the initial target size. The only time that a grid cell will be added to a partition to make it exceed the target size is if the partition is the final partition formed or the partition does not yet contain any grid cells. Given the existence of grid cells that are larger than the target size, we must ensure that DBSCAN's second partitioning constraint is met: that every partition is greater than *MinPts* points. To meet this constraint, we keep track of the running difference of each partition's size from the target size. If this difference is positive, we form partitions proportionately smaller until the difference is neutral or negative again, keeping the minimum partition size set to *MinPts*. Once we finish partitioning grid cells, we add the correct shadow region to the partition, as shown in Figure 2b.

It is common in practice for the last partition to be much larger than most of the other partitions, because as the original partitions are formed, they are kept below the target size. The collective point difference of all partitions from the target size is then left for the final partition, resulting in the need for a rebalancing phase of the partitioning algorithm. Figure 2a demonstrates this, as the populous Eastern United States is included entirely in the last partition formed. Furthermore, the addition of the shadow regions increases the total number of points in the partitioned dataset, and also is likely to negatively affect whatever equality was established by the first iteration through the grid cells. Because of the increase in total points, we update the target size to the *final target size*, which is the mean of the point counts of all the partitions including shadow regions. Then, starting at the last partition formed we remove a grid cell, update the shadow region, and repeat until a specified threshold size is reached. The threshold is set to $1.075 \times finaltargetsize$ because it worked well in practice on our datasets. The removed grid cells are then added to the second-last partition formed, as in Figure 2c. This process is repeated for each partition, working sequentially backward through the partitions until we reach the first.

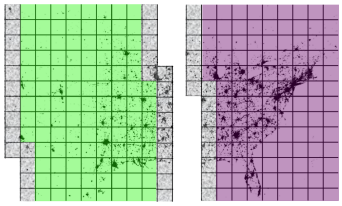
3.1.3 Distributed Partitioner

The distributed partitioner is implemented using MRNet, but it uses a separate network of processes than the final three phases of Mr. Scan. The separate MRNet tree is used because the partitioner is designed to output partitions in parallel to the Lustre file system. We use the file system instead of sending the points directly to the clustering nodes for simplicity of implementation. The next version of Mr. Scan will bypass use of the file system because Lustre has been shown to limit the bandwidth of parallel writes beyond 2000 processes [7], which is significantly smaller than the number of processes needed for the other three phases (up to 20,000). In practice the partitioner did not come close to this limit, as we did not require additional memory when distributing our largest input dataset over 129 processes. The partitioner uses a flat topology as is appropriate for the size of its task.

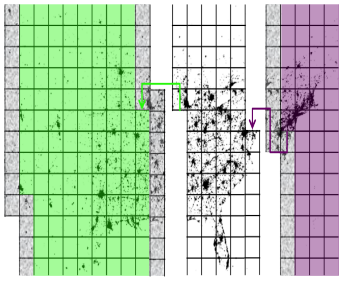
The strategy for distributing the partitioner is based on the fact that the algorithm for forming partitions does not use information about each individual point. The only information needed is a grid of *Eps* x *Eps* cells and the point count for each cell. Therefore, the partitioner is able to distribute the entire input dataset across the memory of the leaf processes and only send a point count of each non-empty *Eps* x *Eps* cell to the root. The root then serially executes the algorithm described in Section 3.1.2 to determine the boundaries of each partition and broadcasts the boundaries to the leaves. The leaves then write the complete point information to the correct position in a single output file in parallel, where the output file contains the points of each



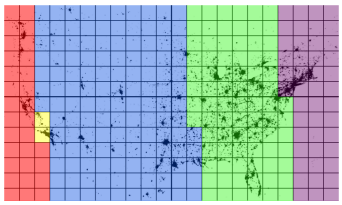
(a) Partition boundaries before rebalancing.



(b) Last two partitions after adding shadow regions



(c) The first step of rebalancing. We remove grid cells from the last partition formed until a threshold is reached and update its shadow region. The removed cells are added to the second-last partition, which gets rebalanced the same way.



(d) Partitions after rebalancing finishes (shadow regions not shown).

Figure 2: Mr. Scan's partition algorithm

partition in sequential order. Additionally, the root generates a metadata file to specify the offset from which each partition starts in the output file. When the output data is extremely dense, we have an optional optimization for the partitioner that chooses representative points for each shadow grid cell in the same manner as described in Section 3.3.1. The partitioner writes out these representative points instead of the contents of the shadow grid cell. This optimization drastically reduces the amount of data written to Lustre and local DBSCAN quality is preserved, but it also may cause the merge algorithm to occasionally miss the opportunity to combine clusters.

3.2 Clustering Phase

The clustering phase runs in parallel on each leaf node, executing a highly multi-threaded implementation of DBSCAN that executes on a GPGPU. The GPGPU algorithm developed for Mr. Scan is an extension of the CUDA-DClust algorithm [6], adding two key modifications to increase scalability both at the clustering and merge steps. The main contribution of these extensions is a reduction of run-time variability caused by differing point density. We start with an overview of the CUDA-DClust algorithm in Section 3.2.1. Our two extensions to CUDA-DClust, improving the host-GPGPU interaction and Dense Box point elimination, are described in Sections 3.2.2 and 3.2.3.

3.2.1 The CUDA-DClust algorithm

The design of the CUDA-DClust algorithm is conceptually similar to the DBSCAN implementation described in Section 2.1. Clustering in CUDA-DClust differs from DBSCAN in that multiple DBSCAN operations to take place on the dataset simultaneously. The number of DBSCAN operations running concurrently is determined by the number of GPGPU blocks. A GPGPU block is the CUDA term for the logical grouping of threads running on a multiprocessor. Figure 3 shows CUDA-DClust at the GPGPU block-level.

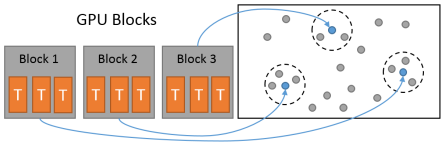


Figure 3: CUDA-DClust GPGPU blocks expanding seed points [6].

Each GPGPU block is assigned a single seed point, which is a point that has not yet been expanded. The DBSCAN algorithm is then run on this point to expand the cluster and find neighboring points. CUDA-DClust uses a modified KD-tree to help DBSCAN determine possible neighbor points. The difference between a standard KD-tree and the CUDA-DClust modified KD-tree is that a leaf represents a region of points instead of a single point. The use of a KD-tree reduces the cost of point expansion by limiting the number of neighbors that need to be checked to the points in the same region of the point being expanded. Other indexing structures, such as the R*-tree typically used in a CPU implementation of DBSCAN, cannot be used on the GPGPU due to the overhead of traversing a tree of arbitrary depth.

After expansion, if the point is determined to be a core point, it is marked as being a member of a cluster and all of its neighbors are added to the block's queue for expansion on subsequent DBSCAN iterations. Otherwise the point is

marked as noise.

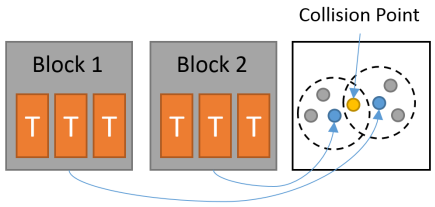


Figure 4: Collision between two concurrently running blocks [6].

After all blocks have completed the expansion of their respective seed points, control is transferred back to the CPU. The CPU then copies the current state for each block from the GPGPU, checks for collisions between blocks, and reseeds the blocks that have an empty point queue with a new unprocessed point. A collision occurs between two blocks when one block attempts to expand a point that has already been expanded by another block or if the point is already in the queue to be expanded by another block. Figure 4 shows an example of this collision where two blocks share a point after an expansion. These collisions indicate that the clusters being expanded in two different GPGPU blocks are actually the same cluster. Collisions between blocks must be tracked and corrected by merging the clusters expanded by all blocks that collide.

3.2.2 Limiting Host to GPGPU interaction

Synchronous memory transfers between CPU and GPGPU are costly operations which should be kept to a minimum. CUDA-DClust has the negative property of performing at least two memory operations between the host and GPGPU after every DBSCAN iteration. This results in a total of $2 \times (Points/BlockCount)$ copy operations between the host and GPGPU. Since leaf nodes in Mr. Scan may have widely varying point counts due to differing shadow area densities, nodes that have high point counts can take much longer to cluster. We modified CUDA-DClust so that there is only a single round trip memory operation regardless of point and block count (copying raw input to the GPGPU and retrieving the clustered result from the GPGPU).

In our new clustering algorithm, there are now two passes over the point data. Pass one classifies all the core points in the dataset and uses a method similar to CUDA-DClust to expand points. One difference is that points are not placed into a block's queue if the number of neighbor points is greater than *MinPts*, and expansion during this phase stops as soon as *MinPts* is reached. Instead of requiring a synchronous memory copy after every input seed point is expanded, the next input seed point for DBSCAN is determined by the parameters of the CUDA kernel call. This allows for all kernel invocations needed to cluster the dataset to be issued in bulk without any intervening memory copies.

The second pass expands the core points found in the first pass to generate the clusters. The clusters are found by running the same operation as above on only the core points in the dataset. When a point is expanded, all of that points neighbors are marked as being members of the cluster. When a collision occurs between two expanded clusters, the collision between these clusters is marked and rectified on the CPU after all points in the dataset has been classified. When all points have been classified, the CPU merges clusters that have collided and the final clusters are revealed.

3.2.3 Dense Box Algorithm

The dense box algorithm allows for points in dense data regions to be marked as members of a cluster without incurring the cost of expanding each point individually. Dense regions of data are detected by using the sub-divided point space generated by the modified KD-tree described in Section 3.2.1. All points in a sub-division with *dimension size* less than or equal to $\frac{Eps}{2\sqrt{2}}$ *by* $\frac{Eps}{2\sqrt{2}}$ and $pointcount \geq MinPts$ will be marked as members of a cluster. The points that are marked as being members of a cluster are not expanded when they are encountered by DBSCAN.

Since we are using an existing sub-division of the point space, there is little added complexity for detecting and eliminating dense boxes. The worst case complexity of this algorithm is $O(l)$ where l is the number of sub-divisions. In return we see a reduction in the complexity of DBSCAN. Worst case DBSCAN complexity drops from $O(n^2)$ to $O((n-p)^2)$ where p is the number of points eliminated by dense box. Average case run time for DBSCAN drops from $O(nlog(n))$ to $O((n-p)log(n))$. We see that as the complexity of DBSCAN rises, the number of points p removed by dense box increases.

3.3 Merge Algorithm

Merging clusters generated by multiple nodes is needed to generate the final output clustering. The merging process is not trivial because a single cluster may span multiple nodes and these clusters only merge if they have a core point in common. Mr. Scan's merge algorithm detects and merges clusters with overlapping core points quickly without requiring the presence of the entire clustered output. Using the entire clustered output would exhaust computational and memory limits as the output grows in size so we select a fixed number of points per grid cell (eight points) to represent the cluster's core points. The points selected to represent the core points are called representative points and are described in Section 3.3.1. The representative points and the set of non-core points of the cluster are used for merging in a method described in Section 3.3.2.

3.3.1 Selection of Representative Points

The set of representative points is the minimum set of core points from a single cluster that can correctly detect a merge inside a single grid cell. Clusters that have overlapping core points need to have at least one core point of overlap within the collective *Eps*-neighborhood that is formed by the set of representative points of the grid cell. We have determined that eight points can represent the core points of a grid cell of arbitrary density. The eight selected representative points are the points closest to the center of the sides of the grid cell and the corners of the grid cell. Figure 5 shows that when two clusters have an overlapping core point in a grid cell that at least one will be within the *Eps*-neighborhood of a representative point.

3.3.2 Merging

The merge algorithm is run on all clusters with overlapping grid cells. At this point in the algorithm, all clusters are composed of grid cells with each grid cell containing a set of representative points and the set of non-core points. The

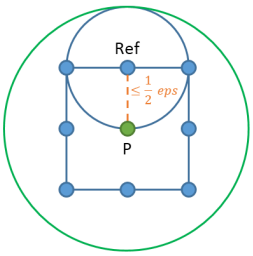


Figure 5: Any overlapping core point P must be within $\frac{1}{2} \times Eps$ of at least one corner or side of the grid cell. We label this point *Ref*. This means that the representative point for *Ref* must fall within a $\frac{1}{2} \times Eps$-neighborhood of *Ref*. Since this entire region (shown as the blue circle) is contained in the *Eps*-neighborhood of P this means that P is always within *Eps* of a representative point.

merge operation is done on every pair of overlapping grid cells between two clusters. There are three types of grid cell overlaps that the merge operation must be able to handle.

The first type, a core point overlap, is when both clusters marked the same point as a core point (shown in Figure 6). Whenever a core point overlap occurs we know the two clusters merge. This case is detected if any representative point from one cluster falls within *Eps* of a representative point from the other cluster.

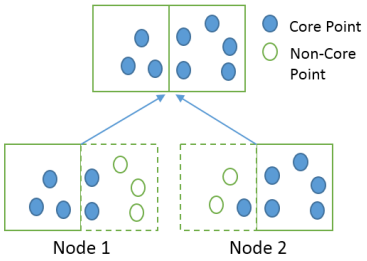


Figure 6: Example of a core point overlap between clusters detected on different nodes.

The second type, a non-core/core overlap, is when one cluster detects a point as being core and another cluster in the grid cell detects the same point as being non-core. This case (shown in Figure 7) arises in Mr. Scan because shadow cells by definition do not have a complete set of neighbor cells on a node. We have shown in Section 3.1.1 the partitioner guarantees that all points within *Eps* of any point in the non-shadow region will be included in the partition, however this is not true of shadow regions. Points in shadow regions could have neighbors that are not visible to the node causing a misclassification of the point. We take advantage of the fact that a non-shadow region will always have the correct classification for points in its cell to detect and merge clusters that have a non-core/core overlap. If we obtain the difference between the set of non-core points found by the shadow region and the set of non-core points found by the non-shadow region we get a set of points that is unique to the shadow region. If any point in the resulting set is within *Eps* of any representative point in the non-shadow region, the clusters merge.

The third type, a non-core/non-core overlap, is when two clusters do not merge but have overlapping non-core points.

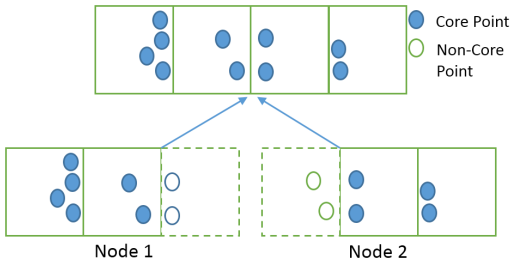


Figure 7: Example of the second type of merge where a core point detected by one node is being detected as a non-core point in a neighboring node. This case is caused by a shadow region not having all neighbor points available (furthest left and furthest right grid cells shown).

We detect this case so that we can remove duplicate non-core points from the output data. Since the non-core points are members of both clusters we can remove the duplicate non-core points from either of the clusters. We resolve this case by removing all duplicate non-core points from the shadow region.

3.4 Sweep Step

The purpose of the sweep step is to write the finalized clusters out to the file system. The sweep step starts with the results of the completed merge operation. It first calculates file offsets to be used by the leaf nodes to write out the points for each cluster. Next a globally unique identifier is assigned to each cluster. This information is then sent down the tree with each level of the tree reversing the merge operation. When a leaf node receives the unique identifier labeling information, it writes all points present in those clusters out to the file system.

4. EXPERIMENT SETUP

We have two goals in evaluating Mr. Scan. The first goal is to test our ability to run DBSCAN on datasets that are several billion points in size in a reasonable amount of time. These datasets must represent real-world problems, and our experiments must use DBSCAN parameters that are useful to the problem. Datasets this size have not been successfully clustered with any density-based clustering algorithm. The second goal is to evaluate whether Mr. Scan exhibits good scaling properties. This is a difficult proposition because memory limits make comparison to a single node implementation impossible. We first evaluate weak scaling, where each leaf process is responsible for roughly 800,000 points. We then evaluate strong scaling, comparing performance at scale to the smallest Mr. Scan instantiation that memory limits allowed on our largest dataset.

We tested Mr. Scan with a synthetically generated dataset derived from geo-located tweets from Twitter and an image dataset obtained from the Sloan Digital Sky Survey. All experiments were run on the Titan supercomputer located at Oak Ridge National Laboratory. Titan is a Cray XK7 system with 18,688 compute nodes. Each compute node has sixteen 2.2GHz AMD Opteron processors with 32 GB of memory, and an NVIDIA Tesla K20 accelerator with 6 GB of memory. Titan is connected to a large Lustre file system. When we ran our experiments, only 8,972 compute nodes were available.

4.1 Twitter Experiment

Research that analyzes user activity on social media sites like Twitter and Facebook is rapidly increasing in popularity. Twitter has been used to detect and predict flu outbreaks [21], alcohol consumption [8], rainfall [21], overall mood of a nation [22], political biases [20], and popular topics [16]. The importance of this research is evidenced by the United States Library of Congress archiving all Twitter tweets from 2006 to the present [27]. Facebook is also used for social science research [28], and Flickr has been used to analyze popular places in a city using photo location data [17]. Many of these research efforts lacked the ability to add location-based information to large scale analyses, so Mr. Scan should make large scale analysis using location information from social media more feasible.

To test Mr. Scan's ability to cluster these datasets, we collected a set of 8,519,781 geo-located tweets from Twitter's public API between August 11-21, 2012. We then used the distribution of these tweets to generate random datasets of arbitrary size for ease of experimentation. In our experiments, we used latitude and longitude as 2D Cartesian Coordinates and fixed our *Eps* value at 0.1 degree to represent a fine-grained analysis. We tested four different *MinPts* values: 4, 40, 400, and 4000 to represent a wide range of output densities.

4.2 Sloan Digital Sky Survey Experiment

The Sloan Digital Sky Survey (SDSS) is an imaging survey that is used to map and catalog astronomical objects using images obtained from terrestrial based telescopes [2]. The images generated by the telescopes in the survey may contain hundreds of new objects which need to be classified and cataloged. Since there are tens of thousands of such images, an automated process is needed to detect these objects. The DBSCAN algorithm has been used to automate the process of detecting, tracking, and classifying objects obtained from terrestrial based telescopes [10] [24]. As data sizes in grow, automated cataloging (and re-cataloging) of these datasets will be needed. Testing was done on the Baryon Oscillation Spectroscopic Survey [1] γ frame photo object data released by SDSS in Data Release 9.

5. EVALUATION

We present results from the evaluation of Mr. Scan for both the Twitter and Sky Survey datasets. In Section 5.1 both the weak and strong scaling results are presented for the Twitter dataset. A breakdown of the running time for each portion of the Mr. Scan algorithm is also provided and discussed. Section 5.2 contains the weak scaling results for the SDSS dataset.

5.1 Twitter Experiment

We evaluated both the weak and strong scaling properties of Mr. Scan using our Twitter datasets. We tested weak scaling by clustering a fixed number of points per leaf node and increasing the data size proportionally to the number of leaf nodes. Table 1 describes the configurations of nodes and data sizes tested for weak scaling. We tested strong scaling by clustering our largest dataset of 6.5 billion points, starting at the number of leaf nodes that had sufficient memory to support their partition size. We then increase the node count to the highest amount the machine allowed. Finally, we evaluated the quality of Mr. Scan's output compared to a single CPU implementation of DBSCAN.

# of points	# of MRNet internal processes	# of leaves	# of partition nodes
1,600,000	0	2	2
6,400,000	0	8	4
25,600,000	0	32	8
102,400,000	0	128	16
409,600,000	2	512	32
1,638,400,000	8	2048	64
3,276,800,000	16	4096	96
6,553,600,000	32	8192	128

Table 1: Configurations used in weak scaling experiment

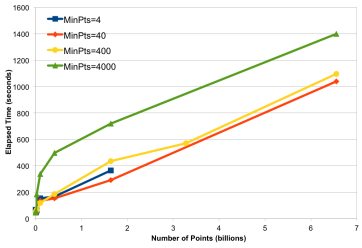


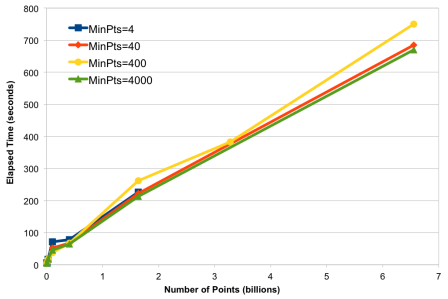
Figure 8: Elapsed time of Mr. Scan for the configurations listed in Table 1. *Eps*=0.1 and *MinPts* varies as indicated.

For all experiments, we use a fixed *Eps* value of 0.1 degree. MRNet uses trees with one compute node per process because there is one GPGPU per compute node on Titan. We use tree topologies that aim to decrease the amount of non-leaf processes in the allocation. Each topology has at most three levels, and each intermediate process has a 256-way fanout of child processes whenever possible. The number of nodes used for the partitioner for each run was determined by selecting the best performing configuration from a prior experiment.

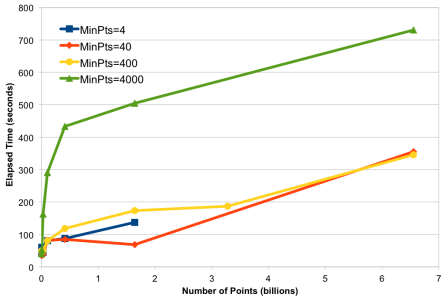
5.1.1 *Weak scaling*

We first present the total time of all Mr. Scan phases in Figure 8, and then we break down this result into the phases that showed different scaling behaviors in Figures 9a, 9b, and 9c. We used four different *MinPts* values for this weak scaling benchmark. The total time of all Mr. Scan phases includes startup and I/O costs, which has not been reported by previous projects that have parallelized DBSCAN. We were able to successfully cluster 6.5 billion points, while largest prior results we found only clustered up to 100 million points. Figure 8 shows that this was accomplished between 1,040 and 1,401 seconds depending on parameters, or between 17.3 and 23.4 minutes. We see that Mr. Scan weak scales linearly with a relatively gentle slope: as the data increases by 4096x (from 1.6 million to 6.5 billion), the total elapsed time has a growth between 18.48x to 31.68x. While the total elapsed times achieved by this scaling property are reasonable and enabling, Mr. Scan does not demonstrate ideal weak scaling.

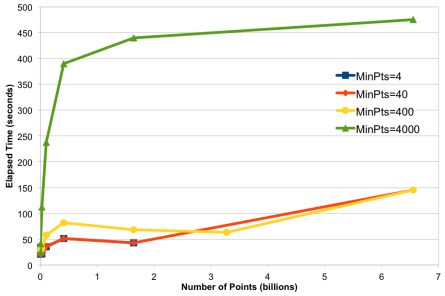
We will see the reason for this growth in total time when we look at the scaling behavior of the partition phase. Figure 9a shows that the partition phase scales linearly with the amount of data, and comparing with Figure 8, we see that this phase takes up roughly 68% of Mr. Scan's overall time. This performance is the largest reason for Mr. Scan's less than favorable weak scaling and is due to the step of writing



(a) Partition Time



(b) Cluster-Merge-Sweep Time



(c) GPGPU DBSCAN Time

Figure 9: Breakdown of Mr. Scan showing the scaling patterns of the various steps for the Twitter dataset. Figures 9a and 9b show a breakdown of Mr. Scan's total excecution time. Figure 9c shows only the time spent clustering in the GPGPU, which represents a portion of the time in Figure 9b.

partitions to Lustre for consumption by the cluster phase. We observed that with *MinPts* set to 400, this write operation took 65.2% of the partition phase, while the initial read operation took 29.92% of the time. The write operation performs poorly because it is dominated by small random writes. This behavior exists because each partitioner leaf process can hold a random portion of data, and may need to contribute some point data to nearly every partition. These contributions are generally small, and each must be written at a specific offset for its respective partition. A better design for this step would be to send partitioned data as messages over the network directly to Mr. Scan's clustering processes. Writing to Lustre was the quickest path to determine the scalability of the other phases, and we plan to implement directly sending data to the leaf nodes as our next step.

Figure 9b shows the scaling results for Mr. Scan's final three phases and Figure 9c breaks out the time spent inside the GPU running DBSCAN and CPU-GPU interaction. We see that for *MinPts* 4, 40, and 400, DBSCAN time actually decreases at one point for each because of our dense box optimization. The 6.5 billion point dataset, however, suggests a further linear trend upward after the decrease. At this high scale, the slowest clustering process is clustering a dense partition that consists of one *Eps* x *Eps* grid cell, and the time of the cluster phase is dictated by the slowest node. This result suggests that the clustering time cannot be decreased by a different grid-based partitioning algorithm. We believe that this upward trend may show the limit of possible optimization using our dense box strategy. With *MinPts* set to 4000, DBSCAN time does scale logarithmically, even though it takes longer to complete. Since our dense box optimization is based on finding *MinPts* points in a small area, it is not as effective when *MinPts* is higher, and we see the optimization producing good scaling at higher point counts than the other three runs. The final three Mr. Scan phases in Figure 9b exhibit similar characteristics to Figure 9c, except that when *MinPts* is 4000, there is still a slight linear growth caused by MRNet startup time. This problem is either due to linear behavior in Cray ALPS on Titan, or to the 256-way fanouts we use in our MRNet tree.

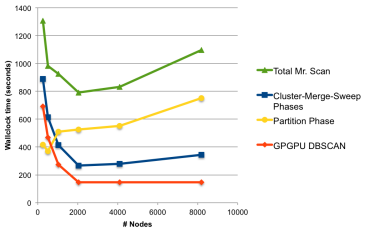


Figure 10: Elapsed time of Mr. Scan when clustering 6.5 billion points and varying the number of cluster processes.

5.1.2 *Strong scaling*

Figure 10 shows results from the strong scaling experiment, where the smallest tree is a tree with no intermediate processes and 256 leaves and each configuration clusters 6.5 billion points. The GPU DBSCAN time improves well at first, providing a 4.7x speedup at 2048 leaves from the smallest tree. Additional leaves do not provide any speedup after 2048 because, again, the slowest cluster process is executing a partition made up of a single dense grid cell. Since this partition cannot be subdivided further, we have again found

a limit to the dense box optimization or we need to subdivide grid cells when they have extremely high density. This performance also suggests that for this dataset, the ideal number of points per leaf process is closer to 3.2 million than 800,000. The total time reflects DBSCAN's inability to improve at higher scale, and also some linear performance that occurs in the partition phase. The linear performance likely comes from the smaller writes to Lustre needed to output more partitions for the same amount of data.

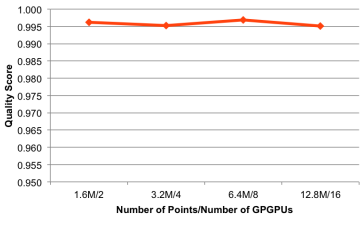


Figure 11: Quality score of Mr. Scan's output compared to the output of DBSCAN on a single CPU.

5.1.3 *Quality*

We evaluated the quality of Mr. Scan's output in comparison to DBSCAN on a single CPU. Quality is measured with a metric defined by the authors of DBDC [15]. The metric assigns a quality score between 0 and 1 to each point as $\frac{|A \cap B|}{|A \cup B|}$, where A is the cluster the point belongs to in DBSCAN's output, and B is the equivalent cluster from Mr. Scan's output. If a point is misidentified as a noise or non-noise point, it gets a quality score of 0. The final quality score is an average of the points' quality scores. Therefore, this metric is maximized when all clusters found contain the exact same points in the output, and when all noise points are identical as well. We were limited to 12.8 million points for this experiment by the memory of a single compute node. We used ELKI 0.4.1 [4] as our reference DBSCAN implementation, which took over 35 hours to cluster the 12.8 million points. Figure 11 shows near-perfect quality at the scales we were able to test, as Mr. Scan did not get lower than a .995 quality score.

5.2 SDSS Experiment

The SDSS experiment consisted of a weak scaling experiment with maximum point count of 1.6 billion points processed on 2048 nodes. This experiment was run with a fixed *Eps* value of 0.00015 and a fixed *MinPts* of 5. Figure 12 shows the weak scaling results for this experiment.

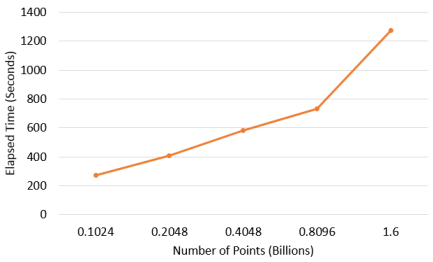


Figure 12: Elapsed time of Mr. Scan for the SDSS dataset at 0.00015 *Eps* and 5 *MinPts*

The weak scaling behavior of the SDSS dataset resembles that of the Twitter dataset. We see similar upward trends in time as the number of nodes increase. As with the Twitter dataset most of the increase in time is contributed by the partitioner. The partitioning time for the SDSS dataset is shown in Figure 13. The reason for the lack of scaling of this portion of Mr. Scan for the SDSS dataset is identical to performance issues discussed for the Twitter dataset (file I/O performance issues).

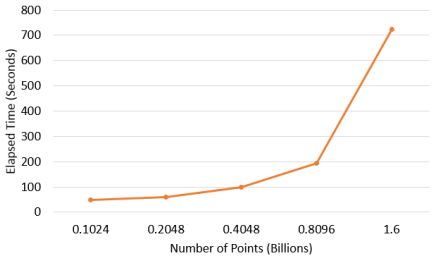


Figure 13: Elapsed partitioning time for Mr. Scan for the SDSS dataset

6. CONCLUSION

We have shown that Mr. Scan is capable of clustering 6.5 billion points in 17.3 minutes, which is the largest known run of DBSCAN by point and node count. Mr. Scan shows that a tree-based distribution network of GPGPU-equipped nodes is useful for developing large-scale data analysis applications. The scaling efficiency of Mr. Scan was impacted by the I/O time needed to transfer partitions to clustering nodes. As future work, we plan to correct this I/O problem by either sending partitions over the network or using Lustre more efficiently. On the current datasets, we achieve maximum efficiency at around 2,000 nodes. This leads us to believe that as we scale up to more nodes, we should be able to efficiently run even larger datasets.

7. REFERENCES

[1] SDSS - Baryon Oscillation Spectroscopic Survey, April 2013. http://www.sdss3.org/surveys/boss.php.

[2] Sloan Digital Sky Survey, April 2013. www.sdss.org.

[3] Twitter, April 2013. https://twitter.com.

[4] E. Achtert, A. Hettab, H.-P. Kriegel, E. Schubert, and A. Zimek. Spatial Outlier Detection: Data, Algorithms, Visualizations. In *Advances in Spatial and Temporal Databases*, volume 6849 of *Lecture Notes in Computer Science*, pages 512–516. Springer Berlin Heidelberg, 2011.

[5] T. Ali, S. Asghar, and N. Sajid. Critical Analysis of DBSCAN Variations. In *International Conference on Information and Emerging Technologies 2010 (ICIET 2010)*, Karachi, Pakistan, June 2010.

[6] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther. Density-Based Clustering using Graphics Processors. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM '09)*, pages 661–670, Hong Kong, China, November 2009. ACM.

[7] L. Crosby. Performance Characteristics of the Lustre File System on the Cray XT5 with Respect to Application I/O Patterns. In *Cray User Group 2009 Proceedings (CUG 2009)*, Atlanta, GA, USA, 2009.

[8] A. Culotta. Lightweight Methods to Estimate Influenza Rates and Alcohol Sales Volume from Twitter Messages. *Language Resources and Evaluation*, 47(1):217–238, 2013.

[9] B.-R. Dai and I.-C. Lin. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. In *IEEE 5th International Conference of Cloud Computing (IEEE CLOUD 2012)*, Honolulu, HI, USA, June 2012.

[10] S. Davidoff and P. Wozniak. RAPTOR-scan: Identifying and Tracking Objects Through Thousands of Sky Images. In *Gamma-Ray Bursts: 30 Years of Discovery: Gamma-Ray Symposium*, Santa Fe, NM, USA, September 2003.

[11] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *The Second International Conference on Knowledge Discovery and Data Mining (KDD '96)*, Portland, OR, USA, August 1996.

[12] T. Gamblin, B. R. de Supinski, M. Schulz, R. J. Fowler, and D. A. Reed. Clustering Performance Data Efficiently at Massive Scales. In *ACM/SIGARCH International Conference on Supercomputing (ICS 2010)*, Epochal Tsukuba, Tsukuba, Japan, June 2010.

[13] Y. He. Personal communication, March 2013.

[14] 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 *The 17th IEEE International Conference on Parallel and Distributed Systems (ICPADS '11)*, Tainan, Taiwan, December 2011.

[15] E. Januzaj, H.-P. Kriegel, and M. Pfeifle. DBDC: Density Based Distributed Clustering. In *Int. Conf. on Extending Database Technology (EDBT '04)*, pages 88–105, Heraklion, Crete, Greece, March 2004.

[16] A. Karandikar. Clustering Short Status Messages: A topic model based approach. Master's thesis, University of Maryland, Baltimore County, 2010.

[17] S. Kisilevich, F. Mansmann, and D. A. Keim. P-DBSCAN: A Density Based Clustering Algorithm for Exploration and Analysis of Attractive Areas Using Collections of Geo-Tagged Photos. In *1st International Conference on Computing for Geospatial Research & Application (COM.Geo '10)*, Washington, DC, USA, June 2010.

[18] M. Kryszkiewicz and P. Lasek. TI-DBSCAN: Clustering with DBSCAN by Means of the Triangle Inequality. In *The Seventh International Conference of Rough Sets and Current Trends in Computing (RSCTC 2010)*, Warsaw, Poland, June 2010.

[19] M. Kryszkiewicz and L. Skonieczny. Faster Clustering with DBSCAN. In *International Conference on Intelligent Information Systems 2005: New Trends in Intelligent Information Processing and Web Mining (IIPWM 2005)*, pages 605–614, Gdansk, Poland, June 2005.

[20] V. Lampos. On Voting Intentions Inference from Twitter Content: A Case Study on UK 2010 General Election. *ACM Computing Research Repository (CoRR)*, abs/1204.0423, 2012.

[21] V. Lampos and N. Cristianini. Nowcasting Events from the Social Web with Statistical Learning. *ACM Transactions on Intelligent Systems and Technology (ACM TIST)*, 3(4):72, 2012.

[22] T. Lansdall-Welfare, V. Lampos, and N. Cristianini. Effects of the Recession on Public Mood in the UK. In *22nd International World Wide Web Conference (WWW '12)*, pages 1221–1226, Lyon, France, April 2012.

[23] R. Mills, F. M. Hoffman, J. Kumar, and W. W. Hargrove. Cluster Analysis-Based Approaches for Geospatiotemporal Data Mining of Massive Data Sets for Identification of Forest Threats. *Procedia CS*, 4:1612–1621, 2011.

[24] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. keng Liao, F. Manne, and A. N. Choudhary. A New Scalable Parallel DBSCAN Algorithm using the Disjoint-Set Data Structure. In *ACM/IEEE Supercomputing Conference 2012 (SC 2012)*, Salt Lake City, UT, USA, November 2012.

[25] P. Roth, D. Arnold, and B. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *ACM/IEEE Supercomputing Conference 2003 (SC 2003)*, Phoenix, Arizona, November 2003.

[26] S. Sonntag, C. T. Paredes, J. Roth, and H.-R. Trebin. Molecular Dynamics Simulations of Cluster Distribution from Femtosecond Laser Ablation in Aluminum. *Applied Physics A*, 104(2):559–565, 2011.

[27] Telegraph. Library of Congress Is Archiving All Of America's Tweets, January 2013. http://www.businessinsider.com/library-of-congress-is-archiving-all-of-americas-tweets-2013-1.

[28] R. E. Wilson, S. D. Gosling, and L. T. Graham. A Review of Facebook Research in the Social Sciences. *Perspectives on Psychological Science*, 7(3):203–220, 2012.

[29] X. Xu, J. Jäger, and H.-P. Kriegel. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Mining and Knowledge Discovery*, 3(3):263–290, 1999.