Hybrid CPU/GPU Clustering in Shared Memory on the Billion Point Scale

Michael Gowanlock
Northern Arizona University
School of Informatics, Computing, and Cyber Systems
Flagstaff, AZ, U.S.A.
michael.gowanlock@nau.edu

ABSTRACT

Many applications require clustering data using an unsupervised approach. One such clustering algorithm is DBSCAN, which is inherently sequential, thus limiting parallelization opportunities. Consequently, several recent works have proposed novel shared- and distributed-memory approaches for scaling DBSCAN. We propose BPS-HDBSCAN, a shared-memory CPU/GPU approach that clusters on the billion-point scale. The major pillars of BPS-HDBSCAN are as follows: (*i*) distance calculation avoidance in dense data regions; (*ii*) efficient merging of subclusters; (*iii*) obviating limited GPU memory capacity by both batching the result set and partitioning the input dataset; and, (*iv*) computing data partitions in parallel, which effectively exploits both CPU and GPU resources. BPS-HDBSCAN is highly efficient, and to our knowledge, is the first shared-memory DBSCAN algorithm to cluster on the billion point scale.

KEYWORDS

DBSCAN, GPU, Heterogeneous Computing, Parallel Clustering

ACM Reference Format:
Michael Gowanlock. 2019. Hybrid CPU/GPU Clustering in Shared Memory on the Billion Point Scale. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3330345.3330349

1 INTRODUCTION

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [9] algorithm uses two input parameters: ϵ and *minpts*, where the former is a search radius, and the latter is the minimum number of points required to form a cluster. The parameters specify a density threshold that informs cluster creation and expansion.

DBSCAN has been intensely studied due to its utility in many applications. For example, in astrophysics, large-scale surveys require clustering millions or billions of objects. A major science goal of the *Gaia* mission [5] is to map the Milky Way galaxy and understand its structure, such as the distribution and morphology of star clusters, which can be detected through DBSCAN [2, 6]. DBSCAN has also been used in other astronomical contexts, for instance, to understand galaxy morphology [20], or environmental effects on radio galaxies [19]. However, modern astronomical surveys are large. The *Gaia* catalog contains 1.69 billion objects, which makes data analysis a significant challenge. Therefore, advances in parallel clustering algorithms are needed to address recent and near-future surveys and missions. We use the *Gaia* catalog to demonstrate the utility of our approach.

We consider the problem of clustering a dataset, D, of $p_i \in D$ points, where $i = 1, \ldots, |D|$, in shared-memory using multi-core CPUs and many-core GPUs. We focus on spatial data (e.g., in the astronomical context), and use the Euclidean distance. DBSCAN has a worst-case quadratic complexity, but a spatial index can decrease the complexity to $O(|D|\log|D|)$ [9], where $|D|$ is the dataset size[1].

The parallel computing community has focused on breaking the sequential nature of DBSCAN to exploit parallelization opportunities [13, 23, 30]. Both the Mr. Scan [30] and the PDSDBSCAN [23] algorithms cluster in parallel and then merge clusters to produce the final cluster assignments. Hybrid-DBSCAN [13, 14] formulates the problem as a self-join with a distance similarity predicate (on the GPU), followed by cluster assignment (on the CPU).

We propose BPS-HDBSCAN that can cluster on the order of a billion data points on a single computer, using multi-core CPUs and many-core GPUs. We make the following key contributions:

- We advance an approach that avoids expensive distance calculations as a function of DBSCAN parameters and data properties.
- We address two memory limitations: (*i*) batching to incrementally compute the ϵ-neighborhood of points on the GPU; and, (*ii*) partitioning the input dataset such that each data partition and all algorithm components fit within the GPU's global memory.
- We exploit both CPU and GPU architectures by clustering partitions and performing intra-partition tasks in parallel. This ensures that CPU and GPU resources remain saturated with work.
- A strength of BPS-HDBSCAN is a combination of several algorithmic design decisions inspired by previous work [14, 23, 30].
- To our knowledge, BPS-HDBSCAN is the first shared-memory DBSCAN algorithm to cluster on the billion-point scale. Previous work on this scale uses distributed-memory; therefore, our algorithm requires fewer computational resources.

The paper is organized as follows: Section 2 gives an overview of DBSCAN and related work; Section 3 outlines leveraged previous work; Section 4 presents our method for avoiding distance calculations; Section 5 proposes the data partitioning and reconciliation strategy for processing large datasets; Section 6 presents the performance evaluation; and finally, Section 7 concludes the paper.

[1] See [26] for a comprehensive discussion of practical ranges of the ϵ parameter.

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330349

2 BACKGROUND

2.1 The DBSCAN Clustering Algorithm

We give a brief overview of DBSCAN, but refer the reader to the original paper [9], or Algorithm 1 in [23] for more information.

The DBSCAN [9] algorithm clusters data using density — points co-located in a dense region are clustered together, and points in underdense regions are considered outliers or noise. The ϵ parameter specifies a search distance, and the *minpts* parameter specifies a density threshold. A point is considered a *core point* if it finds at least *minpts* neighbors within its ϵ-neighborhood (a multi-dimensional range query with a distance ϵ). Given a core point, and its neighbors within ϵ, the algorithm then searches the ϵ-neighborhoods of these neighbors, and points having at least *minpts* neighbors are added to a list of points to search. Therefore, if a point q is reachable by a core point, p, then q is added to the cluster, and q may expand the cluster if it has at least *minpts* within its ϵ-neighborhood. Thus, the algorithm chains points together. *Border points* are defined as those that do not have at least *minpts* points within their ϵ-neighborhood and do not expand the cluster, but are reachable by a core point. After a cluster has been computed through searching the ϵ-neighborhoods of all points in the cluster, the algorithm then selects the next non-visited point, and the next cluster is started if the point is a core point. Points without sufficient neighbors that are not reachable by a core point in a cluster are assigned to noise. For each point, the algorithm outputs an assignment to a cluster or noise. DBSCAN performs ϵ-neighborhood searches on all points in the dataset, which makes the algorithm computationally expensive. However, indexing schemes reduce the complexity of DBSCAN [26].

2.2 Overview of Related Work

We give an overview of related work. However, in Section 2.3 we focus on three key advances in the literature. There have been many optimizations to DBSCAN including distributed-memory algorithms [23, 30] including MapReduce [7, 16], Spark [18, 27], approximate distributed-memory solutions [24], parallel multi-core CPU approaches [11], and algorithms for the GPU [1, 4, 13]. It is well-known that DBSCAN executes in a sequential nature; clusters are expanded through ϵ-neighborhood searches and points in the neighborhood may be assigned to the cluster. Therefore, a major challenge to parallelization is to break the sequential nature of the algorithm to exploit opportunities for concurrent tasks/operations. In GPU algorithms, such as the CUDA-DClust [4], and distributed-memory algorithms [23, 24, 30] the overarching approach is to generate subclusters with multiple threads or processes, and then merge the subclusters to generate the final clusters. This allows for parallel clustering across processing elements, which is needed to scale DBSCAN on a single computer using multi-core CPUs or GPUs, or across multiple distributed-memory compute-nodes.

2.3 Key Advances in the Literature

Below, we highlight three key DBSCAN advances that we leverage.

Dense Box Algorithm– A distributed-memory GPU-accelerated DBSCAN algorithm was proposed by Welton et al. [30]. The authors cluster billions of points in distributed-memory using GPU-equipped nodes, and make several optimizations to the CUDA-DClust [4] algorithm. Of particular interest to our work in this paper is their "dense box algorithm" that will assign points to a cluster if they exist in a dense region. If there are at least $\geq$*minpts* points within a 2-D region of $\epsilon/(2\sqrt{2}) \times \epsilon/(2\sqrt{2})$, the points in this "dense box" are assigned to a cluster, as the points are guaranteed to be within ϵ of each other. This avoids all distance calculations between the points within this dense box region. Welton et al. [30] points out that their detection of dense boxes adds little additional complexity, as the dense boxes are already generated by their modified kd-tree index [3] that they use for ϵ-neighborhood searches. We use a similar dense box algorithm approach as [30].

Disjoint Set Data Structure– A distributed-memory multi-core CPU DBSCAN algorithm was proposed by Patwary et al. [23]. The authors observe the inherent sequential nature of the DBSCAN algorithm, where the cluster expansion phase yields few opportunities for parallelism. Consequently, a major contribution in Patwary et al. [23] is to concurrently cluster the dataset in parallel by having multiple processes cluster their own local points. After all processes have clustered their local data points, the clusters are merged into the final clusters. Their approach relies on using the disjoint set data structure [10], where each point is initially considered to be within its own cluster. During the cluster expansion phase, when a point is to be added to a cluster (e.g., it is within the ϵ-neighborhood of a core point), only pointers need to be updated to represent a point and the cluster it belongs to. This approach is far more efficient than updating cluster assignments each time a point's cluster id may change as a function of the points encountered during the cluster expansion phase, or when merging subclusters. In this paper, we use the disjoint set data structure to update cluster assignments.

Hybrid CPU/GPU Algorithm– Gowanlock et al. [13, 14] proposed a hybrid CPU/GPU approach. In contrast to CUDA-DClust [4], the ϵ-neighborhood of each point in D is computed using a grid-based index on the GPU, which is then used as input to a modified DBSCAN algorithm on the CPU; thus, the ϵ-neighborhood of each point has been precomputed. Patwary et al. [23] and Welton et al. [30] ameliorate the sequential nature of the algorithm by clustering in parallel and then merge to create the final clusters, and Gowanlock et al. [13] performs all of the ϵ-neighborhood searches in parallel before assigning the points to clusters. An additional benefit of HYBRID-DBSCAN in Gowanlock et al. [13, 14] is that they address large result set sizes that exceed the GPU's global memory capacity. We employ HYBRID-DBSCAN in this work.

3 LEVERAGING HYBRID-DBSCAN

HYBRID-DBSCAN [13, 14] computes the ϵ-neighborhood of all points on the GPU, and the data is used by DBSCAN for clustering on the CPU, thus eliminating CPU index searches (Section 2.3). Compared to HYBRID-DBSCAN, BPS-HDBSCAN supports datasets that exceed GPU global memory by partitioning the data, concurrently clustering the data partitions, and avoiding distance calculations in dense regions. Thus, HYBRID-DBSCAN is limited to smaller datasets, and BPS-HDBSCAN is needed to cluster on the billion-point scale.

We leverage HYBRID-DBSCAN for computing the ϵ-neighborhood of points on the GPU, and clustering on the CPU. We present the high-level details of the index and batching scheme, but refer the reader to [13, 14] for more detail.

Indexing Scheme – HYBRID-DBSCAN uses a grid-based indexing scheme for the GPU, with cells of length ϵ. Therefore, to perform an ϵ-neighborhood search around a query point (the point being searched) in a given cell, points that may be within ϵ are guaranteed to be within adjacent cells, and the cell containing the point (9 total cells in 2-D). Having all points search the same number of cells reduces thread divergence in the kernel, in comparison to having execution pathways that may diverge (such as having threads perform parallel tree traversals in tree-based indexes [13]). While we use this grid-based indexing structure, we make a minor modification proposed by Gowanlock & Karsin [12], which is to only index non-empty grid cells. Thus, the size of the index is a function of the space occupied by the data, and not the entire bounding volume, which was not the case in [13]. We denote the index as I.

Since we only store non-empty cells, and use a lookup array for the grid cells and data points, the space complexity is $O(|D|)$; including constant factors, the worst case is $4|D|$. This small indexing structure allows us to use more of the GPU's global memory for other purposes, such as processing larger datasets or partitions.

Batching Scheme – HYBRID-DBSCAN also uses a batching scheme such that the total result set size can exceed the GPU's global memory capacity. A key novelty of the batching scheme is that it samples the dataset to determine the number of batches to be executed, n_b, by estimating the total result set size. This has the advantage of not requiring result set buffer overflow mitigation.

The GPU kernel counts the number of points within ϵ of a sample of $p_i \in D$, and then estimates the total result set size. We use n_s CUDA streams for concurrent data transfers to/from the GPU and computation on the GPU. At minimum, we use $n_s = n_b = 3$. We use asynchronous data transfers of the result set for each batch into pinned memory buffers as a staging area, where the aggregate size is $n_s \cdot b_s$. On one GPU, we use $b_s = 10^8$; thus, at minimum, the total size of pinned memory buffers on the host is 3×10^8. This size is doubled when we use two GPUs. We set the overestimation factor to 0.25 (denoted as α in [13]) to accommodate reference points (detailed in Section 4.4) that have a larger search distance.

3.1 HYBRID-DBSCAN Algorithm Overview

We give a brief algorithmic overview of HYBRID-DBSCAN (see [13, 14] for additional details). Note that the only difference between the description of HYBRID-DBSCAN in Gowanlock et al. [13, 14] and that used in this paper is that we only index non-empty grid cells. Furthermore, in our description of HYBRID-DBSCAN in Algorithm 1, the algorithm takes as input the index, I, as we assume it has already been computed. For illustrative purposes, we have separated the computation of the neighbortable, N, from HYBRID-DBSCAN, as we refer to it when we outline BPS-HDBSCAN in Section 5.1.

HYBRID-DBSCAN is outlined in Algorithm 1. The *neighbortable*, N, is computed on the GPU and is defined as the neighbors within ϵ of all query points, Q (line 2). A modified DBSCAN algorithm (line 3) is executed on the CPU that uses N, Q, and *minpts* to compute the assignment of points to clusters, which are returned on line 4.

Algorithm 1 The HYBRID-DBSCAN Algorithm.

```
1:  procedure HYBRID-DBSCAN(D, I, ε, minpts, Q)
2:      N ← neighborTableGPU(D, I, ε, Q)
3:      C^Dbscan ← DBSCAN(N, Q, minpts)
4:      return C^Dbscan

5:  procedure NEIGHBORTABLEGPU(D, I, ε, Q)
6:      N ← ∅;
7:      n_b ← estimateBatches(D, I, ε, b_s)
8:      for k ∈ {1, 2, ..., n_b} do
9:          R ← ∅; gpuResultSet ← ∅
10:         gpuResultSet ← GPUCALCGLOBAL(D, I, ε, Q, k)
11:         R ← SortByKey(gpuResultSet)
12:         N ← N ∪ ConstructNeighborTable(R)
13:     return N
```

For space, we omit presenting the modified DBSCAN algorithm. In Gowanlock et al. [13], Q contains all point ids, i, in $p_i \in D$. However, BPS-HDBSCAN uses a query set Q, that may not contain all $p_i \in D$.

We outline the construction of the neighbortable, N, as follows. The algorithm initializes N on line 6. Next, the number of batches, n_b, from the batch estimator are computed, and a loop is entered which computes all batches (lines 7–8). Result set buffers are initialized on line 9. The GPU kernel is executed on line 10 which takes as input D, the index, I, ϵ, the query ids, Q, and the batch number, k, and generates gpuResultSet, which is a list of key/value pairs: the key is a point $p_i \in D$, and the value is an id of a point within ϵ of the key. Thus, gpuResultSet contains key/value pairs (a, b) where $dist(p_a, p_b) \leq \epsilon$, and $dist$ is the Euclidean distance function between p_a and p_b. The gpuResultSet is intermediate data stored in global memory on the GPU. Next, gpuResultSet is sorted by key (line 11), such that each point's neighbors within ϵ are stored contiguously in memory. The sorted array is transferred and stored in R on the host. Using R, the host (CPU) then constructs the neighbortable, N, by storing the neighbors within ϵ of each point (line 12). We use $n_s = 3$ streams to overlap data transfers and computation; therefore, the loop on line 8 is executed in parallel by 3 CPU threads.

4 DISTANCE CALCULATION AVOIDANCE

We describe a similar approach to Welton et al. [30] that eliminates computing the distances between points if they exist in very dense regions, and exploits the disjoint set data structure utilized by Patwary et al. [23]. Our dense box algorithm is denoted as DENSEBOX.

The dense box algorithm in Welton et al. [30], uses spatial divisions in their index to detect dense boxes, and thus, eliminating dense boxes has little cost. In contrast, since we use HYBRID-DBSCAN which employs an ϵ-length grid cell index, we need to generate a separate grid for the DENSEBOX computation. Consequently, DENSEBOX can degrade performance if we employ the algorithm on a dataset that has few dense regions. We propose an approach that dynamically enables or disables DENSEBOX as a function of the data distribution and DBSCAN parameters, such that DENSEBOX does not degrade performance. DENSEBOX is outlined as follows.

4.1 Merging Adjacent Dense Boxes

We create a grid with cells of length $\epsilon/(2\sqrt{2})$. Next, we mark all cells that have at least *minpts* points contained within as dense boxes. The points contained within these boxes are by definition core

points of a cluster. Using cells of length $\epsilon/(2\sqrt{2})$ guarantees that adjacent dense boxes contain points that are within ϵ of each other. Figure 1 shows an example where four dense boxes are detected (cells with dashed outlines). In our algorithm, we merge dense boxes that are adjacent to each other (the three dense boxes on the left with dashed outlines are merged in Figure 1). The points in the single dense box on the right with the blue dashed outline are not merged with the other dense boxes because they are not adjacent, and are not guaranteed to contain points within ϵ of each other.

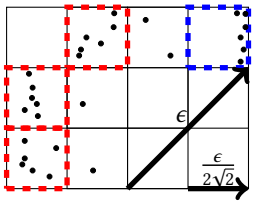


Figure 1: Example of dense boxes detected having *minpts* $\geq$ 5 (dashed outlines). The three adjacent dense boxes on the left cannot be merged with the densebox on the right, because they are separated by a non-dense box.

4.2 Merging Non-Adjacent Dense Boxes

While dense boxes can be detected and merged if they are adjacent to each other, another case is where two dense boxes may be separated by a non-dense box cell of length $\epsilon/(2\sqrt{2})$ and may need to be merged because they contain points that are within ϵ of each other. However, since these dense boxes are not adjacent, they are not guaranteed to contain any points within ϵ of each other, and cannot be simply merged, as in Section 4.1. Figure 2(a) shows two dense boxes (dashed blue outline) that are separated by a non-dense box, and none of the points are within ϵ of each other. These two dense boxes should not be merged. Figure 2(b) shows the case where two dense boxes are separated by a non-dense box, and some of the points are within ϵ of each other. In this case, the two dense boxes should be merged. To detect a merge, only one point within a dense box needs to be within ϵ of a point in the other dense box. This is because dense boxes are guaranteed to contain $\geq$ *minpts* points. We merge non-adjacent dense boxes with each other when we detect that they should be merged (as in Figure 2(b)).

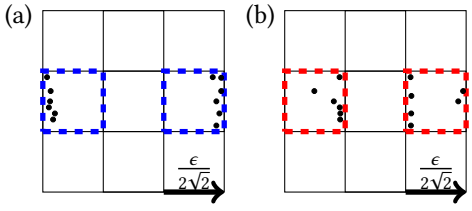
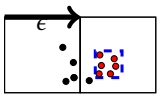


Figure 2: Dense boxes separated by a non-dense box. (a) Dense boxes do not merge (no pair of points between the two dense boxes are within ϵ). (b) Dense boxes merge (at least one pair of points between the two dense boxes are within ϵ).

4.3 Merging Dense Boxes & DBSCAN Clusters

Points that are not assigned to a dense box must be clustered with HYBRID-DBSCAN [13, 14] (Section 3). These clusters may merge with dense boxes. Figure 3 shows an example with cells of length ϵ, where a cluster generated by HYBRID-DBSCAN (points in black) need to merge with a dense box (red points within the blue dashed outline box). Points within the dense box are within ϵ of the points in the DBSCAN cluster, and the two clusters are merged.

Figure 3: Example of merging a dense box and HYBRID-DBSCAN cluster.



4.4 Detection of Dense Box Merges

Merges between non-adjacent dense boxes (Section 4.2) and between dense boxes and DBSCAN clusters (Section 4.3) must be detected. To detect these merges, we require finding whether points within the dense boxes and/or DBSCAN clusters are within ϵ of each other. However, since we explicitly do not perform ϵ-neighborhood searches around dense box points, and do not consider the dense box points when generating the DBSCAN clusters, there is no expansion of ϵ-neighborhoods between points that merge the two clusters in the same manner as the original DBSCAN algorithm.

We propose to use *reference points* to detect merges. Given our grid with cells of length ϵ, we place a reference point in each non-empty cell. Then, when we compute the ϵ-neighborhood of all points not found within a dense box using HYBRID-DBSCAN, we also compute neighborhood searches around the reference points. The points in the neighborhood around each reference point may contain points that are part of one or more dense boxes or HYBRID-DBSCAN clusters. Given the distance between these points, we can then determine whether a merge should occur.

Figure 4: Example reference point with a 1.5ϵ-neighborhood search. The reference point detects a merge between the HYBRID-DBSCAN cluster and the dense box. Other reference points are shown in the center of non-empty cells.

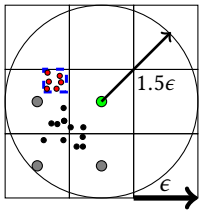


Figure 4 shows an example of merging using reference points. A reference point is in the center of the grid. We perform a 1.5ϵ-neighborhood search around each reference point (a search distance of ϵ would miss potential merges). The 1.5ϵ-neighborhood of the reference point will contain all of the points in the DBSCAN cluster (black points), and the dense box cluster (points within the dashed outline). Then, we compare the points between the two clusters. If any single point within a dense box is within ϵ of a point in the DBSCAN cluster, then we merge the two clusters. Likewise, we merge two dense boxes if a point in each are within ϵ of each other (Figure 2(b)). Thus, we detect merges between: (*i*) two non-adjacent dense boxes; and (*ii*) a dense box and a HYBRID-DBSCAN cluster.

This is similar to the eight "representative points" that represent a dense box in Welton et al. [30], such that it can be merged when encountered by the point expansion phase. We find that we can use a single reference point to capture the clusters to merge.

We compute the 1.5ϵ-neighborhoods of the reference points when we compute the ϵ-neighborhoods of the points not found

within dense boxes using Hybrid-Dbscan, and thus, we capitalize on amortizing GPU overheads during these kernel invocations. We adapt the batching scheme of Hybrid-Dbscan (Section 3) to BPS-HDbscan, by also including in our calculations of the total result set size the 1.5ϵ-neighborhood of each reference point. Thus, we modify the expected total result set size, to reflect this larger search area, and the number of reference points.

4.5 Merging using Disjoint Sets

There can be a large number of merges between dense boxes and other dense boxes or Hybrid-Dbscan clusters. Thus, relabeling a point's cluster assignment at every merge would be very expensive. To avoid excessive updates to cluster assignments, we use the disjoint set data structure [10] which records merges between clusters. After all clusters have been merged, we then update the cluster assignment list once using a single scan for each $p_i \in D$.

We first label all dense boxes as clusters, where each dense box is given an enumerated cluster id. The disjoint set is initialized using the total number of cluster ids, where each dense box is represented by a single element. Then, we find all dense boxes that are adjacent to each other (Section 4.1), and *union* those dense box ids. The $union(x, y)$ operation in the disjoint set data structure [10] finds the two sets containing the dense boxes, x, and y, and combines or merges the sets. Likewise, we obtain the list of clusters from Hybrid-Dbscan, and use the neighborhoods yielded by the reference points to detect merges between two non-adjacent dense boxes (Section 4.2), or between Hybrid-Dbscan clusters and dense boxes (Section 4.3). Thus, the three types of merges employ the disjoint set data structure.

The disjoint set data structure [10] is highly efficient. The amortized search (and union) operations occur in nearly $O(1)$ time in practice, as the search procedure has been shown to increase with the inverse of Ackermann's function [28]. After all of the merges have occurred, we use the mapping of elements in the disjoint set to update the list of cluster ids.

The disjoint set data structure has been used in the multi-core distributed-memory Dbscan approach of Patwary et al. [23], and is used in their work to allow multiple threads (or processes) to cluster subclusters of the data in parallel and then merge the subclusters. This approach is very similar to our merging of dense boxes with each other or with Hybrid-Dbscan clusters.

4.6 Selectively Using the Dense Box Algorithm

As discussed in Welton et al. [30], the dense box algorithm is not useful in all contexts. First, if ϵ is sufficiently large and the value of *minpts* is low, then there are likely to be many dense boxes. However if ϵ is small, then the probability of at least *minpts* points being contained within a dense box is very low. Furthermore, if the value of *minpts* is high, then this also decreases the probability of at least *minpts* points being contained within a dense box. Note that increasing the search radius, ϵ, typically increases the computational complexity of Dbscan, as more points need to be checked to determine if they are within ϵ. However, increasing ϵ increases the probability that points are found within a dense box. Therefore, DenseBox can dramatically reduce the complexity of clustering with high ϵ, and may even make clustering with high ϵ more efficient than a lower value of ϵ, which is an unintuitive consequence of the algorithm.

DenseBox has a cost. We must first partition the dataset into cells of length $\epsilon/(2\sqrt{2})$ and compute which cells are dense boxes. We also need to reconcile merges between DenseBox clusters and/or Hybrid-Dbscan clusters using reference points. Therefore, if the fraction of data points in D that can be detected using DenseBox is low, then it is not advantageous to use the DenseBox algorithm, and we can simply use Hybrid-Dbscan from Gowanlock et al. [13, 14]. However, DenseBox should be employed if it can significantly reduce the number of ϵ-neighborhood searches.

We propose to selectively use DenseBox when the data distribution, ϵ, and *minpts* values are conducive to eliminating many distance calculations (i.e., a substantial fraction of points within the dataset are found in dense boxes). We use information from our grid-based index to determine whether we should employ the DenseBox algorithm. From our indexing procedure, we compute the number of ϵ-length non-empty cells, denoted as $|G|$. Therefore, we can compute the mean number of points per cell in the index.

The ratio of the area of the ϵ to $\epsilon/(2\sqrt{2})$ length grid cells is as follows: $(\epsilon^2)/\frac{\epsilon^2}{(2\sqrt{2})^2} = 8$. Therefore, the mean number of points per cell of length $\epsilon/(2\sqrt{2})$ is $|D|/8|G|$. However, the data is not guaranteed to be uniformly distributed within these cells, and therefore some cells will have more or less than $|D|/8|G|$ points. Thus, some of these $\epsilon/(2\sqrt{2})$ length cells are likely to contain $\geq$*minpts* points (meeting the dense box criterion), and others will not. Additionally, it may be beneficial to employ DenseBox when we expect only a fraction of D to be found in dense boxes.

We employ a heuristic to determine whether we should utilize the DenseBox algorithm. If the following evaluates to true, we use DenseBox: $|D|/8|G| \geq \mu \cdot minpts$, where the μ parameter controls the threshold estimated number of points needed within a cell of length $\epsilon/(2\sqrt{2})$. We note the following observations:

•If $\mu = 1$, then on average, we expect that at least *minpts* points are found within each dense box length cell.

•If $\mu = 0$, then dense box is enabled, regardless of the average number of points in each cell.

•Low μ increases the probability that DenseBox is employed.

•$\mu > 1$ is likely to rarely enable DenseBox, except when the dataset has a few very overdense regions, or *minpts* is very small.

Selectively using DenseBox uses information from the ϵ-length grid cell index regardless of whether DenseBox is utilized, which is why it is not formulated in terms of the $\epsilon/(2\sqrt{2})$ length grid cells. We refer to using DenseBox dynamically as DBox-Dynamic.

5 CLUSTERING LARGE DATASETS

A drawback of the GPU is its limited global memory capacity. Therefore, in cases where the dataset and all algorithm components exceed global memory, the entire dataset must be divided into n_p partitions. Clustering occurs on all partitions, and then the partitions are merged. Likewise, data partitioning is needed in distributed-memory implementations [23, 30] such that each node can concurrently cluster separate regions of a dataset.

We sample D and bin this sampled data in the first dimension. Then we simply partition D into n_p partitions, where each partition has roughly the same number of points.

After all n_p partitions have been clustered, we then merge the partitions by detecting when two clusters from neighboring partitions share an overlapping core point and must be merged. We compute this using the ϵ-neighborhood of each point within a *shadow region*, and reference points, similarly to detecting dense box merges (Section 4.4). When a reference point detects points in its 1.5ϵ-neighborhood belonging to two (or more) different clusters, we check to see if any two points belonging to two different clusters have at least *minpts* neighbors. If they do, and one of the points within its ϵ-neighborhood belongs to a different cluster, then we have detected that these two clusters should be merged. Figure 5 shows an illustrative example with two clusters detected in two separate partitions, where the shadow region is denoted by the shaded gray cells. The ϵ-neighborhood of a point in the cluster in partition 1 contains points in the cluster in partition 2. Using the ϵ-neighborhood of points within the shadow region, we detect that these two clusters must be merged.

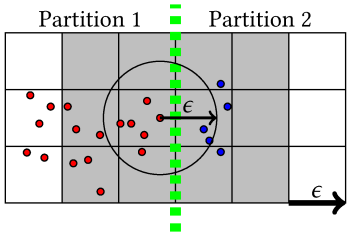


Figure 5: Using points in the shadow region (shaded cells), we detect that the clusters in partitions 1 and 2 should be merged (vertical dashed line demarcates partitions).

Points assigned to noise in two separate partitions that are in the shadow region may form a cluster. For all points detected as noise in each partition's shadow region(s), we check to see if they should form a cluster. If so, we assign them cluster ids, as appropriate.

Figure 6 shows an example of partitioning the *Gaia* catalog, where red regions denote the greatest density (galactic midplane and center), and blue regions denote the lowest density. The figure shows that while each partition contains roughly the same number of points, the density in each partition may vary considerably. Figure 6 shows a total of $n_p = 10$ partitions, where 5 partitions are being concurrently clustered, denoted as $n_c = 5$, two partitions have been clustered (leftmost partitions), and clustering has not yet started on 3 partitions (rightmost partitions). Of the 5 partitions currently being clustered (dashed outlines), DBox-Dynamic disables DenseBox on 2 partitions and enables DenseBox on 3 partitions.

5.1 BPS-HDbscan Algorithm Overview

The pseudocode of BPS-HDbscan is outlined in Algorithm 2, which takes as input the dataset (D), ϵ, *minpts*, and the number of partitions (n_p). The final cluster assignments, C, and the cluster assignments for each partition, C, are initialized to empty sets on lines 2–3. Next, the dataset is partitioned into n_p partitions on line 4. A loop clusters each partition on line 5, where there are n_p total partitions/iterations. Next, clusters computed by the DenseBox algorithm and by Dbscan are initialized on lines 6–7. The index is constructed on the partitioned data on line 8. Next, if the DenseBox algorithm is to be executed, then the if statement is entered on line 9. The DenseBox algorithm is executed on line 10, which takes as input the data in the partition, $\mathcal{D}[j]$, ϵ, and *minpts*, and returns a set of

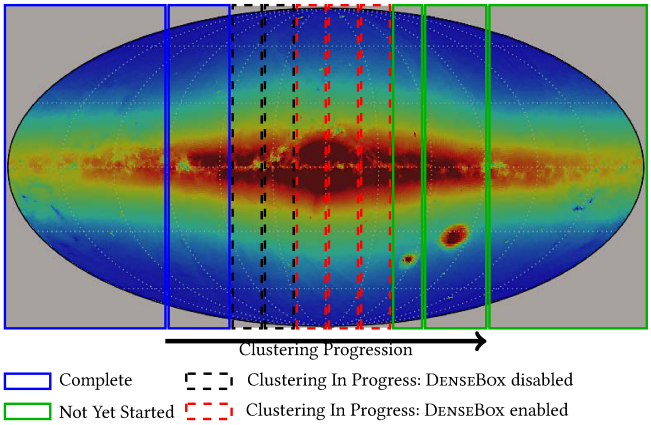


Figure 6: Example of concurrently clustering $n_c = 5$ partitions (dashed outlines), where there are $n_p = 10$ total partitions. The data shown is a density map [8] from the *Gaia* catalog [5], where red (blue) denotes high (low) density regions. The partition sizes are illustrated to contain roughly the same number of points within each partition, but span different spatial sizes. DBox-Dynamic is enabled, and some partitions are clustered without DenseBox (black dashed outline) or with DenseBox (red dashed outline).

clusters, C^{DBox}, and the point ids of those points that could not be clustered with DenseBox, denoted as L. The reference points, M, are computed as a function of the non-empty cells in the index, I, and ϵ on line 11. A query set, Q, containing L and M is computed on line 12. Next, the neighbors in Q are computed as follows: those within ϵ of each point in L (not computed by DenseBox on line 10), and the neighbors within 1.5ϵ of each reference point in M are computed on the GPU on line 13, and the neighbors of each point, N, are returned (see Algorithm 1). The neighbors, N, L, and *minpts* are used as input to Dbscan, which clusters all of the points that were not clustered with DenseBox (line 14). Next, the clusters generated by DenseBox, C^{DBox}, and by Dbscan, C^{Dbscan}, are merged as necessary, using $\mathcal{D}[j]$, M, and N, and the final clusters for the partition are generated (line 15).

If the DenseBox algorithm was not employed on line 9, then Hybrid-Dbscan is executed on line 18 (Algorithm 1), and the final clusters for the partition are returned.

Lastly, if there is more than one partition ($n_p > 1$), the partitions must be reconciled, where clusters may be merged, and noise points detected in individual partitions may form clusters, which yields the final set of clusters, C, on line 20. If there is only a single partition ($n_p = 1$), then the final cluster assignments are simply the cluster assignments from the first (only) partition (line 22).

In Algorithm 2, we execute a number of partitions, n_c, in parallel (loop iterations on line 5), as they are independent of each other. Furthermore, parts of: index construction (line 8), the DenseBox procedure (line 10), reference point generation (line 11), computing the neighbortable on the GPU (line 13), and merging clusters (line 15) are computed in parallel. The details of parallelization

Algorithm 2 BPS-HDBSCAN

```
1:  procedure BPS-HDBSCAN(D, ϵ, minpts, n_p)
2:      C ← ∅
3:      C ← initClusterSets(n_p)
4:      𝒟 ← partitionDataset(D, n_p)
5:      for j ∈ {1, 2, ..., n_p} do
6:          C^DBox ← ∅
7:          C^Dbscan ← ∅
8:          I ← constructIndex(𝒟[j], ϵ)
9:          if EnableDenseBox then
10:             C^DBox, L ← denseBox(𝒟[j], ϵ, minpts)
11:             M ← generateReferencePoints(ϵ, I)
12:             Q ← L ∪ M
13:             N ← neighborTableGPU(𝒟[j], I, ϵ, Q)
14:             C^Dbscan ← DBSCAN(N, L, minpts)
15:             C[j] ← mergeClusters(𝒟[j], M, N, C^DBox, C^Dbscan)
16:         else
17:             Q ← D[j]
18:             C[j] ← HYBRID-DBSCAN(𝒟[j], I, ϵ, minpts, Q)
19:     if n_p > 1 then
20:         C ← reconcilePartitions(C, D)
21:     else
22:         C ← C[1]
    return
```

are not shown, but are not particularly worthy of elaboration (e.g., concurrent execution of data-parallel operations in for loops).

Note that when we use the GPU when computing the neighbortable (line 13 or line 18), we only allow a single thread computing a partition to use the GPU at a given time. This is because we allocate pinned memory buffers for the result set that are reused across partitions for constructing the neighbortable (Section 3). Thus, if multiple threads computing their respective partitions attempt to use the GPU at the same time, then they must wait for each other to obtain the resource. In the evaluation, we test using 1 or 2 GPUs.

DENSEBOX is executed based on the condition on line 9 in Algorithm 2. In the evaluation, we test three DENSEBOX configurations: off (thus using HYBRID-DBSCAN), on, and dynamic.

6 EXPERIMENTAL EVALUATION

6.1 Datasets

We cluster the coordinates (ra, dec) of objects in the *Gaia* catalog (data release 2), which contains 1.69 billion points [5], and refer to this dataset as *Gaia*. We also cluster OpenStreetMap (*OSM*) GPS track data [22], and removed duplicate points in the dataset. The *OSM* dataset consists of 473.7 million points after duplicate point removal. We also select the first 50 million points in each dataset for quantifying the correctness of BPS-HDBSCAN, and comparing to a sequential implementation, denoted as *Gaia50M* and *OSM50M*. These datasets have contrasting point distributions, allowing us to compare performance under different workloads.

6.2 Experimental Methodology

Experimental Setup – The GPU code is written in CUDA 9 [21], and all C/C++ host code is compiled with the GNU compiler with the O3 optimization flag, and parallelized using OpenMP. We compare the performance of BPS-HDBSCAN with HYBRID-DBSCAN and R-TREE (described below). We store the dataset as 32-bit floats. The following parameters are fixed in the experimental evaluation: $n_s = 3$, and $b_s = 10^8$. See Table 1 for parameter descriptions.

We execute the CUDA kernel, GPUCALCGLOBAL, in Algorithm 1 with 256 threads per block. Experiments are averaged over 3 trials, and executed on a platform with 2× Intel Xeon E5-2683 v4 (32 physical cores), clocked at 2.1 GHz, 256 GiB of RAM, and 2× Nvidia Titan X GPUs (each with 12 GiB of global memory).

As discussed in Section 5.1, many of the BPS-HDBSCAN operations occur concurrently within a single partition. We use nested parallelism in OpenMP for performing these tasks in parallel. We allow multiple partitions to be computed concurrently, as specified by the n_c parameter. However, the majority of the host-side (CPU) operations are memory-bound, so n_c does not need to be large to saturate memory bandwidth. Recall that the compute-intensive ϵ-neighborhood searches occur on the GPU.

We quantify BPS-HDBSCAN correctness in Section 6.4. The quality metric is expensive and requires several weeks to compute. Thus, in Section 6.4 (only in this section) we use a different platform, but note that the output of BPS-HDBSCAN is identical across platforms.

Table 1: Summary of notation used in the evaluation.

Symbol	Description
ϵ	Input distance parameter to the algorithm.
minpts	Input density parameter to the algorithm.
n_p	The number of partitions generated from the input dataset.
n_c	The maximum number of concurrently executing partitions.
DBOX-OFF	Disabling using DENSEBOX on all partitions.
DBOX-ON	Enabling using DENSEBOX on all partitions.
DBOX-DYNAMIC	Selectively using DENSEBOX based on each partition's data distribution and DBSCAN parameters.
μ	Parameter in the DBOX-DYNAMIC heuristic controlling the likelihood that DENSEBOX is enabled on a given partition.
n_{GPU}	The number of GPUs used (1 or 2).
b_s	The result set batch size used to compute the neighbortable (N). Each CUDA stream has a buffer of this size.
n_s	The number of CUDA streams used for each GPU. Used to overlap data transfers and computation.
N	The neighbortable that stores the ϵ-neighborhood of points.

Selection of Parameters – As discussed in related work [13, 23, 26], the ϵ and *minpts* DBSCAN parameters must be carefully selected to not obtain too few or too many clusters. As such, we carefully select ϵ and *minpts*. In the literature, the *minpts* value has been selected to be fairly low, such as ≤ 25 [23], 4 [9], and 2 times the dataset dimensionality [25]. Thus, we employ similar *minpts* values.

6.3 Algorithms & Configurations

We execute BPS-HDBSCAN with three DENSEBOX algorithm options:
DBOX-OFF– Disabling DENSEBOX is equivalent to using the HYBRID-DBSCAN algorithm of Gowanlock et al. [13, 14] to cluster the dataset. Thus, when we do not use DENSEBOX, we simply use HYBRID-DBSCAN, and reconcile the partitions (merge clusters, etc.) as needed.
DBOX-ON– We use DENSEBOX on all partitions, regardless of the data characteristics and input parameters. Thus, if DENSEBOX has low utility on a partition (few points found in dense boxes), then the algorithm may degrade performance compared to DBOX-OFF.
DBOX-DYNAMIC– We selectively use DENSEBOX on a partition as a function of the data characteristics (controlled by the parameter μ). Depending on the parameters and data characteristics, some partitions may employ DENSEBOX, and some may not.

We compare performance to a sequential CPU-only algorithm.

R-Tree– This implementation is the standard CPU-only Dbscan algorithm that performs ϵ-neighborhood searches using an index, and we employ an R-tree [15]. We index each $p_i \in D$ in a minimum bounding box and perform ϵ-neighborhood searches on each point to find its neighbors. The minimum bounding box of a query point is defined by the minimum and maximum corners of the box, and are constructed as follows: $MBB^{min} = (p_i^x - \epsilon, p_i^y - \epsilon)$, and $MBB^{max} = (p_i^x + \epsilon, p_i^y + \epsilon)$, where p_i^x and p_i^y are the 2-D coordinates of p_i.

Regarding BPS-HDbscan, we include the time to index the dataset in the grid. However, we exclude the time to index using R-Tree, as we do not optimize R-tree index construction. As is customary, we exclude the time to load the dataset from disk.

6.4 Correctness of DBSCAN Clustering Output

The cluster assignments can differ depending on the order in which the points are visited in Dbscan. For instance, a point can be visited that exists in a sparse region and be assigned to noise, whereas the same point, if encountered during the cluster expansion phase, can be added to a cluster as a border point. While Dbscan assigns each $p_i \in D$ to a single cluster, it is possible that a border point could be assigned to potentially one of two (or more) clusters. Therefore, Dbscan has slightly non-deterministic behavior depending on the order in which the data points are encountered.

Gowanlock et al. [11] and Welton et al. [30] use a metric in Januzaj et al. [17] to evaluate the correctness of parallel clustering results. We use the same metric to compare the cluster assignments of Dbscan executed on a single core with R-Tree vs. BPS-HDbscan. In particular, the metric in [17] assigns a quality score between 0 and 1 to each point. If a point in both executions is assigned to noise, then the point receives a quality score of 1; however, if the point is assigned to a cluster in one of the executions and noise in the other, then the point receives a quality score of 0. If a point is assigned to a cluster in both executions then the score is $\frac{|E \cap F|}{|E \cup F|}$, where E is the cluster assigned to the point using sequential Dbscan (R-Tree) and F is the cluster assigned to the point using BPS-HDbscan. Thus, if the points in E and F are identical, then the point receives a quality score of 1. The average score of all points yields the total quality score of BPS-HDbscan clustering output.

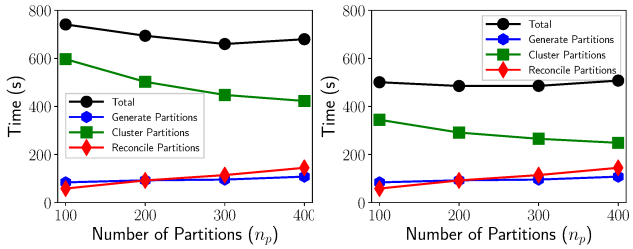
Table 2: Quality score of the clustering output on *Gaia50M* with $\epsilon = 0.03$, and *OSM50M* with $\epsilon = 0.0006$ and *minpts* = 4. On *Gaia50M* (*OSM50M*) 6% (2.54%) of the data points are assigned to noise.

Num. Partitions (n_p)	DBox-Off *Gaia50M*	DBox-On *Gaia50M*	DBox-Off *OSM50M*	DBox-On *OSM50M*
2	0.999988	0.999950	0.999968	0.999967
4	0.999953	0.999937	0.999890	0.999973
8	0.999930	0.999914	0.999825	0.999947

Table 2 shows the accuracy of BPS-HDbscan on the *Gaia50M* and *OSM50M* datasets. The datasets are limited to 50 million points because computing the quality of the result is computationally expensive. BPS-HDbscan achieves nearly identical output compared to the sequential implementation, R-Tree, when disabling or enabling DenseBox across a range of n_p.

6.5 Varying the Number of Partitions

Figure 7 plots the response time of various components of BPS-HDbscan with DBox-Off vs. the number of partitions (n_p) on the entire *Gaia* dataset ($|D| = 1.69$ billion), for $\epsilon = 0.01$, and *minpts* = 4 on (a) 1 and (b) 2 GPUs. We disable DenseBox because we want to observe performance when each partition must compute the entire ϵ-neighborhood of each point. The time to generate and reconcile the partitions are identical in Figure 7(a) and (b).



(a) $n_{GPU} = 1$ (b) $n_{GPU} = 2$

Figure 7: Response time vs. the number of partitions (n_p) for (a) $n_{GPU} = 1$ and (b) $n_{GPU} = 2$. BPS-HDbscan is executed with DBox-Off, $\epsilon = 0.01$, *minpts* = 4, and $n_c = 16$.

In Figure 7(a), observe that the time to generate the partitions increases with n_p. This is expected, and we see weak linear growth. Likewise, we find that reconciling the partitions by merging points in the shadow region also grows linearly with n_p; however, the growth is more substantial, because there is significant work involved in processing points in the shadow regions. Interestingly, the time to cluster the partitions when DBox-Off decreases as n_p increases. This is because with larger n_p, the pipeline can be filled faster, as the partitions are clustered independently, where up to $n_c = 16$ threads/partitions are being computed concurrently. The threads must contend for the single GPU, as only a single thread computing a partition may access the GPU at a time. The first 16 partitions to be computed can require many of the 16 threads to wait for each other to finish computing the ϵ-neighborhoods of the points in their respective partitions. However, the GPU wait times decrease after the first 16 partitions are computed, as the threads contend for the GPU to a smaller extent. This occurs because the probability that more than two threads need the GPU at any given time decreases, as other components of the clustering phase are being computed (e.g., indexing, Dbscan point assignment). Therefore, having smaller partitions (larger n_p), allows for less waiting for GPU resources overall. We see from the figure that increasing n_p has diminishing returns, as the difference in response time for clustering the partitions between $n_p = 300$ and $n_p = 400$ does not vary considerably.

Figure 7(b) shows the same trends for $n_{GPU} = 2$. At $n_p = 200$ the time to cluster for $n_{GPU} = 2$ is 291.44 s whereas it requires 502.51 s for $n_{GPU} = 1$. This demonstrates that while multiple partitions are concurrently clustered, threads need to wait to use a GPU, and the addition of a GPU substantially improves the performance of the clustering phase (the longest executing phase of the algorithm).

Combining the three time components in Figure 7 for either one or two GPU cases, we find that since the time to generate and reconcile partitions increases with n_p, and the time to cluster the partitions decreases with n_p, the total time is fairly consistent across $n_p = [100, 400]$. This demonstrates that n_p can be selected in a large range and achieve good performance.

Increasing n_p decreases the memory footprint of BPS-HDbscan, as small n_p has large partitions, which must store the ϵ-neighborhoods of each point in the neighbortable, N. However, once the points in N are clustered, then this working memory is no longer used. Therefore, increasing n_p can also be used to decrease the working memory required by BPS-HDbscan without significant performance loss.

6.6 Efficacy of DenseBox

Figure 8 plots the response time for each DenseBox configuration on the *Gaia* dataset: DBox-On, DBox-Dynamic ($\mu = 0.25, 0.5, 0.75, 1$), and DBox-Off for $\epsilon = 0.005, 0.01, 0.015$. In Figure 8, the configurations are ordered as a spectrum. Using DBox-Dynamic, as μ increases, it approaches the same behavior as DBox-Off. Thus, DBox-On is similar to DBox-Dynamic with $\mu = 0.25$, and DBox-Off is similar to DBox-Dynamic with $\mu = 1$.

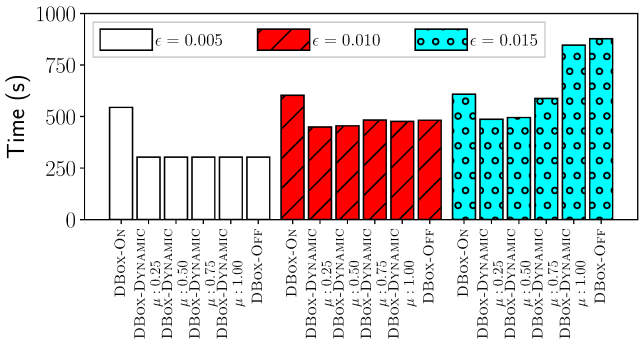


Figure 8: Response time of BPS-HDbscan for several DenseBox configurations: DBox-On, DBox-Dynamic ($\mu = 0.25, 0.5, 0.75, 1$), and DBox-Off on the *Gaia* dataset with *minpts* = 4, $n_p = 200$, $n_c = 16$, and $n_{GPU} = 2$.

At $\epsilon = 0.005$ (the smallest ϵ value), we find that DBox-Dynamic does not enable DenseBox on any partition across all values of μ, as the search radius is too small, and thus DBox-Dynamic is equivalent to DBox-Off. Performance degrades using DBox-On compared to DBox-Off, as BPS-HDbscan is forced to use the DenseBox algorithm and few points can be eliminated from performing ϵ-neighborhood searches. This demonstrates that DenseBox should be used judiciously, otherwise, it can degrade performance.

At $\epsilon = 0.01$ (the intermediate ϵ value), we find that DBox-On leads to the worst performance, and the best performance is obtained when we use DBox-Dynamic with $\mu = 0.25$. However, the difference between DBox-Dynamic with $\mu = 0.25$ and DBox-Off is similar, which suggests at this value of ϵ, DenseBox can only marginally improve performance.

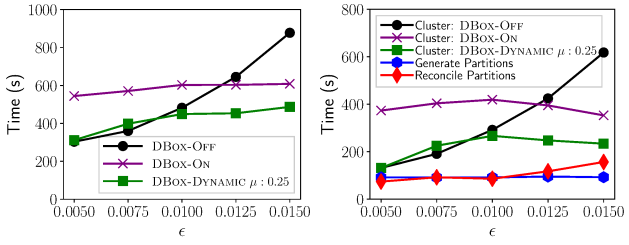
At $\epsilon = 0.015$ (the largest ϵ value), we find that DBox-Off significantly degrades performance compared to DBox-On. Furthermore, we find that of the DBox-Dynamic configurations, using $\mu = 0.25$ achieves the best performance. We clearly see that the DenseBox algorithm is superior to not using the algorithm when ϵ (and *minpts*) are conducive to eliminating many GPU ϵ-neighborhood searches.

To optimize BPS-HDbscan, we find that across the three values of ϵ in Figure 8, which span a large range of of noise (2.44–19.35%) we do not need to select DBox-Off or DBox-On; rather, we can use DBox-Dynamic which will enable DenseBox when the data in a given partition is likely to be conducive to eliminating several ϵ-neighborhood calculations. Furthermore, we observe that we can simply select $\mu = [0.25, 0.5]$ to achieve good performance.

6.7 Performance vs. Search Distance

We examine the performance of BPS-HDbscan on *Gaia* and *OSM* as a function of ϵ. As ϵ increases, the number of neighbors increase, which increases the workload of the original Dbscan algorithm. We compare DBox-Off, DBox-On, and DBox-Dynamic with $\mu = 0.25$.

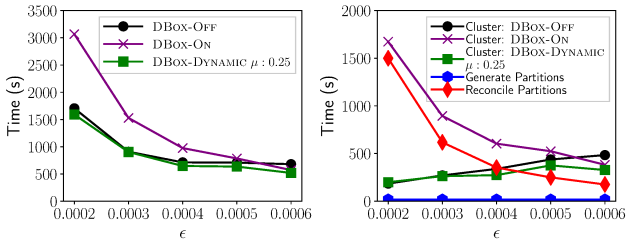
6.7.1 Comparison of BPS-HDbscan Configurations. Figure 9(a) plots the total response time vs. ϵ on the *Gaia* dataset. We find that performance degrades with increasing ϵ when DBox-Off. Since the ϵ-neighborhood of all points needs to be computed, then increasing ϵ increases the workload. However, we find that DBox-Off outperforms DBox-On when $\epsilon \lesssim 0.0125$, illustrating that employing DenseBox across all partitions can degrade performance; however, when $\epsilon \geq 0.0125$, DBox-On outperforms DBox-Off as more points can be eliminated from executing ϵ-neighborhood searches using the DenseBox algorithm. We find that DBox-Dynamic (with $\mu = 0.25$) generally outperforms both DBox-Off and DBox-On, by selectively employing DenseBox (with the exception at $\epsilon = 0.0075$, where DBox-Off slightly outperforms DBox-Dynamic). Thus, DBox-Dynamic decreases the chances of degrading performance on low ϵ values by disabling DenseBox, and ensures that it is more likely to be enabled as ϵ increases.



(a) *Gaia*: Total Response Time (b) *Gaia*: Time Breakdown

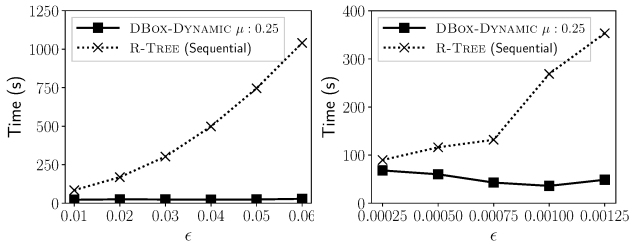
Figure 9: Response time vs. ϵ on *Gaia*, where BPS-HDbscan is executed with $n_{GPU} = 2$, *minpts* = 4, $n_p = 200$, and $n_c = 16$. (a) Total response time; and, (b) Time breakdown.

Figure 9(b) shows the time to cluster using the DenseBox configurations and the time to generate and reconcile the partitions (these components are independent of DenseBox configuration). Comparing the time to cluster, we observe that the time decreases on DBox-On and DBox-Dynamic when $\epsilon > 0.01$, illustrating that larger ϵ values eliminate more points using DenseBox.



(a) *OSM*: Total Response Time (b) *OSM*: Time Breakdown

Figure 10: Response time vs. ϵ on *OSM*, where BPS-HDbscan is executed with $n_{GPU} = 2$, *minpts* $= 4$, $n_p = 32$, and $n_c = 4$. (a) Total response time; and, (b) Time breakdown.



(a) *Gaia50M*: Total Response Time (b) *OSM50M*: Total Response Time

Figure 11: Total response time vs. ϵ (*minpts* $= 4$) on (a) *Gaia50M*; and (b) *OSM50M*. BPS-HDbscan is executed with $n_{GPU} = 2$, $n_p = 4$, and $n_c = 4$.

Figure 10(a) plots the total response time on the *OSM* dataset. We find that on this dataset, lower ϵ values require more time to compute, even with DBox-Off. This is an unintuitive result, as increasing ϵ increases the workload (more distance comparisons are required). However, in Figure 10(b) we see that a large amount of time is spent reconciling the partitions. This is because with a small ϵ value, there are more non-empty cells, and thus, more reference points are needed to merge adjacent partitions. Consequently, small ϵ degrades performance. We note that on the *OSM* dataset, ϵ is very small to obtain a reasonable number of clusters and fraction of noise points. This is because the dataset has overdense regions with many co-located points (GPS positions in large cities). Overall, we find that DBox-Dynamic outperforms DBox-Off and DBox-On across all values of ϵ on this dataset.

6.7.2 Comparison to the Sequential Implementation. We compare to a standard sequential Dbscan algorithm, R-Tree, that searches an R-tree index to compute the ϵ-neighborhood of points. We note that R-Tree runs out of memory on the full datasets, so we compare performance on the smaller datasets. Figure 11(a) and (b) plot the total response time of R-Tree and BPS-HDbscan using DBox-Dynamic with $\mu = 0.25$ on *Gaia50M* and *OSM50M*. Recall that we include in the response time measurements index construction for BPS-HDbscan, but not R-Tree, thus giving an advantage to R-Tree. We find that on the *Gaia50M* dataset, the performance of R-Tree degrades significantly with ϵ, whereas it is nearly independent of ϵ on BPS-HDbscan. On the *OSM50M* dataset, we observe the same idiosyncratic behavior in Figure 10(a), where the response time decreases with ϵ, due to the overhead associated with reconciling the partitions on small ϵ values. The speedup of BPS-HDbscan over the sequential implementation, R-Tree, is up to 36.69× on *Gaia50M*, and 7.47× on *OSM50M*.

7 DISCUSSION & CONCLUSIONS

Previous work that clusters using Dbscan on the billion point scale utilizes distributed-memory, such as Welton et al. [30], which is the first billion-point scale paper [29], and He et al. [16]. Patwary et al. [23] clustered hundreds of millions of data points, but at higher dimensionality than the abovementioned works [16, 30].

BPS-HDbscan clusters on the scale of a billion points using a hybrid CPU/GPU approach with up to two GPUs on a single computer. To our knowledge, this is the first paper to cluster with Dbscan on the billion-point scale on a single machine, which requires fewer computational resources than the distributed-memory approaches. While we cannot make a direct comparison to other approaches due to varying datasets and hardware platforms, we find that BPS-HDbscan achieves similar performance to the distributed-memory Mr. Scan implementation of Welton et al. [30]. For instance, on their Twitter experiment with 1.6 billion points with *minpts* = 4, Mr. Scan performs its clustering and merging phase in ~140 s on over 2,000 GPU nodes. This performance is comparable to the clustering and partition reconciliation phases of BPS-HDbscan on *Gaia* (1.69 billion points), which ranges between 206 s ($\epsilon = 0.005$) and 391 s ($\epsilon = 0.015$). We reiterate that this is an order of magnitude, not a direct, comparison of approaches.

Motivated by Welton et al. [30], BPS-HDbscan selectively uses DenseBox depending on the data distribution and whether the Dbscan parameters are conducive to generating many dense boxes which can eliminate searching the ϵ-neighborhoods of a large fraction of points in a dataset. Similarly to Patwary et al. [23], BPS-HDbscan uses the disjoint set data structure to decrease the expensive cost of relabeling points. This is important for the DenseBox procedure and reconciling the partitions. BPS-HDbscan uses the Hybrid-Dbscan approach of Gowanlock et al. [13], which performs ϵ-neighborhood searches on the GPU and point assignments on the CPU. By computing multiple partitions concurrently, BPS-HDbscan effectively saturates both the CPU and GPU with work.

Future work includes performance modeling, evaluating different partitioning strategies, and a distributed-memory approach.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant OAC-1849559. I would like to thank two groups of anonymous reviewers: (*i*) the reviewers of this ICS'19 paper that provided constructive feedback on the manuscript; and (*ii*) a TPDS reviewer of the article [14] that suggested combining the dense box algorithm with Hybrid-Dbscan. I thank Ben Welton for clarifying details of Mr. Scan [30], and Frédéric Loulergue for the use of his computer.

REFERENCES

[1] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. 2013. G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering. *Procedia Computer Science* 18 (2013), 369 – 378.

[2] Giacomo Beccari, Henri M J Boffin, Tereza Jerabkova, Nicholas J Wright, Venu M Kalari, Giovanni Carraro, Guido De Marchi, and Willem-Jan de Wit. 2018. A sextet of clusters in the Vela OB2 region revealed by Gaia. *Monthly Notices of the Royal Astronomical Society: Letters* 481, 1 (2018), L11–L15.

[3] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *CACM* 18, 9 (1975), 509–517.

[4] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. 2009. Density-based Clustering Using Graphics Processors. In *Proc. of the 18th ACM Conf. on Information and Knowledge Management (CIKM '09)*. 661–670.

[5] AGA Brown, A Vallenari, T Prusti, JHJ de Bruijne, C Babusiaux, CAL Bailer-Jones, et al. 2018. Gaia Data Release 2. Summary of the contents and survey properties. *Astronomy & Astrophysics, to appear* (2018).

[6] Castro-Ginard, A., Jordi, C., Luri, X., Julbe, F., Morvan, M., Balaguer-Núñez, L., and Cantat-Gaudin, T. 2018. A new method for unveiling open clusters in Gaia - New nearby open clusters confirmed by DR2. *A&A* 618 (2018), A59.

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *CACM* 51, 1 (2008), 107–113.

[8] ESA. 2019. Milky Way Density Map. https://gea.esac.esa.int/archive/documentation/GDR2/Catalogue_consolidation/chap_cu9gat/sec_cu9gat_skydensity/

[9] Martin Ester, Hans Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of the 2nd KDD*. 226–231.

[10] Bernard A Galler and Michael J Fisher. 1964. An improved equivalence algorithm. *Commun. ACM* 7, 5 (1964), 301–303.

[11] Michael Gowanlock, David M. Blair, and Victor Pankratius. 2016. Exploiting Variant-based Parallelism for Data Mining of Space Weather Phenomena. In *Proc. of the 30th IEEE Intl. Parallel & Distributed Processing Symposium*. 760–769.

[12] M. Gowanlock and B. Karsin. 2018. GPU Accelerated Self-Join for the Distance Similarity Metric. In *Proc. of the 2018 IEEE Intl. Parallel and Distributed Processing Symposium Workshops*. 477–486.

[13] Michael Gowanlock, Cody M Rude, David M Blair, Justin D Li, and Victor Pankratius. 2017. Clustering Throughput Optimization on the GPU. In *Proc. of the IEEE Intl. Parallel and Distributed Processing Symposium*. 832–841.

[14] Michael Gowanlock, Cody M Rude, David M Blair, Justin D Li, and Victor Pankratius. 2019. A Hybrid Approach for Optimizing Parallel Clustering Throughput using the GPU. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (2019), 766–777.

[15] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proc. of ACM Intl. Conf. on Management of Data*. 47–57.

[16] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. 2014. MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. *Frontiers of Computer Science* 8, 1 (2014), 83–99.

[17] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. 2004. DBDC: Density based distributed clustering. In *International Conference on Extending Database Technology*. Springer, 88–105.

[18] G. Luo, X. Luo, T. F. Gooch, L. Tian, and K. Qin. 2016. A Parallel DBSCAN Algorithm Based on Spark. In *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*. 548–553.

[19] F. Massaro, N. Álvarez-Crespo, A. Capetti, R. D. Baldi, I. Pillitteri, R. Campana, and A. Paggi. 2019. Deciphering the Large-scale Environment of Radio Galaxies in the Local Universe: Where Are They Born? Where Do They Grow? Where Do They Die? *The Astrophysical Journal Supplement Series* 240, 2, Article 20 (2019), 20 pages.

[20] H Meusinger, J Brünecke, P Schalldach, et al. 2017. A large sample of Kohonen selected E+ A (post-starburst) galaxies from the Sloan Digital Sky Survey. *Astronomy & Astrophysics* 597 (2017), A134.

[21] NVIDIA. 2017. CUDA Programming Guide 9.0. http://docs.nvidia.com/cuda

[22] OpenStreetMap. [n. d.]. https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/. Accessed 05-20-2019.

[23] Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Weikeng Liao, Fredrik Manne, and Alok Choudhary. 2012. 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*. Article 62, 11 pages.

[24] Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Fredrik Manne, Salman Habib, and Pradeep Dubey. 2014. Pardicle: Parallel Approximate Density-based Clustering. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*. 560–571.

[25] Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 1998. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. *Data Mining and Knowledge Discovery* 2, 2 (1998), 169–194.

[26] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.* 42, 3, Article 19 (2017), 21 pages.

[27] Abdul Ghaffar Shoro and Tariq Rahim Soomro. 2015. Big data analysis: Apache spark perspective. *Global Journal of Computer Science and Technology* (2015).

[28] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (1975), 215–225.

[29] Benjamin Welton and Barton P Miller. 2014. The Anatomy of Mr. Scan: A Dissection of Performance of an Extreme Scale GPU-Based Clustering Algorithm. In *Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. IEEE Press, 54–60.

[30] Benjamin Welton, Evan Samanas, and Barton P Miller. 2013. Mr. Scan: Extreme Scale Density-Based Clustering using a Tree-Based Network of GPGPU Nodes. In *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.