PECANN: Parallel Efficient Clustering with Graph-Based Approximate Nearest Neighbor Search

Anonymous Author(s)

Abstract

This paper studies density-based clustering of points. These methods use dense regions of points to detect clusters of arbitrary shapes. In particular, we study variants of density peaks clustering, a popular type of algorithm that has been shown to work well in practice. Our goal is to cluster *large high-dimensional* datasets, which are prevalent in practice. Prior solutions are either sequential and cannot scale to large data, or are specialized for low-dimensional data.

This paper unifies the different variants of density peaks clustering into a single framework, PECANN (**P**arallel **E**fficient **C**lustering with **A**pproximate **N**earest **N**eighbors), by abstracting out several key steps common to this class of algorithms. One such key step is to find nearest neighbors that satisfy a predicate function, and one of the main contributions of this paper is an efficient way to do this predicate search using graph-based approximate nearest neighbor search (ANNS). To provide ample parallelism, we propose a doubling search technique that enables points to find an approximate nearest neighbor satisfying the predicate in a small number of rounds. Our technique can be applied to many existing graph-based ANNS algorithms, which can all be plugged into PECANN.

We implement five clustering algorithms with PECANN and evaluate them on synthetic and real-world datasets with up to 1.28 million points and up to 1024 dimensions on a 30-core machine with two-way hyper-threading. Compared to the state-of-the-art FASTDP algorithm for high-dimensional density peaks clustering, which is sequential, our best algorithm is 45x–734x faster while achieving competitive ARI scores. Compared to the state-of-the-art parallel DPC-based algorithm, which is optimized for low dimensions, we show that PECANN is two orders of magnitude faster. As far as we know, our work is the first to evaluate DPC variants on large high-dimensional real-world image and text embedding datasets.

ACM Reference Format:
Anonymous Author(s). 2024. PECANN: Parallel Efficient Clustering with Graph-Based Approximate Nearest Neighbor Search. In . ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Clustering is the task of grouping similar objects into clusters and is a fundamental task in data analysis and unsupervised machine learning [2, 9, 51]. For example, clustering algorithms can be used to identify different types of tissues in medical imaging [105], analyze social networks [71], and identify weather regimes in climatology [19]. They are also widely used as a data processing subroutine in other machine learning tasks [21, 63, 69, 102]. One popular type of clustering is density-based clustering, where clusters are defined as dense regions of points in space. Recently, density-based clustering algorithms have received a lot of attention [3, 7, 34, 45, 46, 53, 80, 85, 97] because they can discover clusters of arbitrary shapes and detect outliers (unlike popular algorithms such as k-means, which can only detect spherical clusters).

Density peaks clustering (DPC) [80] is a popular density-based clustering technique for spatial data (i.e., point sets) that has proven very effective at clustering challenging datasets with non-spherical clusters. Due to DPC's success, many DPC variants have been proposed in the literature (e.g., [16, 28, 35, 38, 47, 87, 90, 91, 103, 107, 108]). However, existing DPC variants are sequential and/or tailored to low-dimensional data, and so cannot scale to the large, high-dimensional datasets that are common in practice.

This paper addresses this gap by proposing a novel framework called PECANN: **P**arallel **E**fficient **C**lustering with **A**pproximate **N**earest **N**eighbors. PECANN contains implementations for a variety of different DPC density techniques that both scale to large datasets (via efficient parallel implementations) and run on high dimensional data (via approximate nearest neighbor search). Designing a unifying framework for DPC variants is non-trivial, as DPC variants can differ significantly. Developing a modular and extensible framework that can seamlessly incorporate various DPC variants and allow for easy comparison and experimentation requires careful abstraction and encapsulation of the key algorithmic components. Furthermore, extending DPC to high dimensions is challenging as there are no efficient parallel solutions for constrained nearest neighbor search in high dimensions, which is needed for DPC. Before going into more details on our contributions, we review the main steps of DPC variants and discuss existing bottlenecks.

The three key steps of DPC variants are as follows:

(**1**) Compute the density of each point x.
(**2**) Construct a tree by connecting each point x to its closest neighbor with higher density than x.
(**3**) Remove edges in the tree according to a pruning heuristic. Each resulting connected component is a separate cluster.

Step (**1**) is computed differently based on the variant, but all variants use a function that depends on either the k-nearest neighbors of x or the points within a given distance from x. Efficient implementations of this step rely on nearest neighbor queries or range queries. In low dimensions, these queries can be answered efficiently using spatial trees, such as kd-trees. However, kd-trees are inefficient in high dimensions due to the curse of dimensionality [101]. Step (**2**) again requires finding nearest neighbors, but with the constraint that only neighbors with higher density are considered. Step (**3**) can easily be computed using any connected components algorithm. Steps (**1**) and (**2**) form the bottleneck of

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the computation, and take quadratic work in the worst case, while Step (**3**) can be done in (near) linear work. Note that different clusterings can be generated by reusing the tree from Step (**2**) and simply re-running Step (**3**) using different pruning heuristics. The tree from Step (**2**) can be viewed as a cluster hierarchy (or dendrogram) that contains clusterings at different resolutions.

Existing papers on DPC variants mainly focus on their own proposed variant, and as far as we know, there is no unified framework for implementing and comparing DPC variants and evaluating them on the same datasets. Furthermore, most DPC papers focus on clustering low-dimensional data, but many datasets in practice are high dimensional ($d > 100$). The PECANN framework unifies a broad class of DPC variants by abstracting out these three steps and providing efficient parallel implementations for different variants of each step. For Step (**1**), we leverage graph-based approximate ANNS algorithms, which are fast and accurate in high dimensions [68, 96]. For Step (**2**), we adapt graph-based ANNS algorithms to find higher density neighbors by iteratively doubling the number of nearest neighbors returned until finding one that has higher density. Our doubling search guarantees that the algorithm finishes in a logarithmic number of rounds, making it highly parallel. For Steps (**1**) and (**2**), PECANN supports the following graph-based ANNS algorithms: Vamana [55], pyNNDescent [70], and HCNNG [73]. For Step (**3**), we use a concurrent union-find algorithm [54] to achieve high parallelism. Prior work [87] has explored using graph-based ANNS for high-dimensional clustering, but their algorithm is not parallel and they only consider one DPC variant and one underlying ANNS algorithm. In addition, we provide theoretical work and span bounds[1] of PECANN that depend on the complexity of the underlying ANNS algorithm. PECANN is implemented in C++, using the ParlayLib [10] and ParlayANN [68] libraries, and provides Python bindings as well.

We use PECANN to implement five DPC variants and evaluate them on a variety of synthetic and real-world data sets with up to 1.28 million points and up to 1024 dimensions. We find that using a density function that is the inverse of the distance to the k^{th} nearest neighbor, combined with the Vamana algorithm for ANNS, gives the best overall performance. On a 30-core machine with two-way hyper-threading, this best algorithm in PECANN achieves 37.7–854.3x speedup over a parallel brute force approach, and 45–734x speedup over fastdp [87], the state-of-the-art DPC-based algorithm for high dimensions, while achieving similar accuracy in terms of ARI score. fastdp is sequential, but even if we assume that it achieves a perfect speedup of 60x, PECANN still achieves a speedup of 0.76–12.24x. Compared to the state-of-the-art parallel density peaks clustering algorithm by Huang et al. [48], which is optimized for low dimensions, our best algorithm achieves 320x speedup while achieving a higher ARI score on the MNIST dataset (in high dimensions, this algorithm is slower than brute force and failed to run on larger datasets).

Our contributions are summarized below.

1. We introduce the PECANN framework that unifies existing k-nearest neighbor-based DPC variants and supports parallel implementations of them that scale to large high-dimensional datasets. We provide fast parallel implementations for five DPC variants.
2. We extend graph-based ANNS algorithms with a parallel doubling-search method for finding higher density neighbors.
3. We perform comprehensive experiments on a 30-core machine with two-way hyper-threading showing that PECANN outperforms the state-of-the-art DPC-based algorithm for high dimensions by 45–734x. As far as we know, we are the first to compare different variants of DPC on large high-dimensional real-world image and text embedding datasets. We also empirically compared the performance of different graph-based ANNS approaches in the context of DPC.

Our code and the full version of the paper are available at https://anonymous.4open.science/r/PECANN-DPC-807C/.

[1]The *work* is the number of operations and the *span* (or parallel time) is the length of the longest sequential dependence of a computation.

Notation	Meaning
P	input set of points
n, d	size and dimensionality of P
x_i	i^{th} point in P
G	a similarity search index
$D(x_i, x_j)$	distance (dissimilarity) between x_i and x_j
ρ_i, λ_i	density and dependent point of x_i
δ_i	dependent distance of x_i (i.e., $D(x_i, \lambda_i)$)
k	the number of neighbors used for computing densities
$\mathcal{N}_i$	(approximate) k-nearest neighbors of x_i
$\mathcal{W}_c, \mathcal{S}_c$	the work and span of constructing G
$\mathcal{W}_{nn}, \mathcal{S}_{nn}$	the work and span of finding nearest neighbors using G

Table 1: Notation

2 Preliminaries

2.1 Definitions and Notation

A summary of the notation is provided in Table 1. Let $P = \{x_1, \ldots, x_n\}$ represent a set of n points in d-dimensional coordinate space to be clustered . We use x_i to represent the i^{th} point in P. Let G be a search index that supports searching for the exact or approximate nearest neighbors of a query point. Let $D(x_i, x_j)$ denote the distance (dissimilarity) between points x_i and x_j, where a larger distance value means the two points are less similar. D can be any distance measure the search index G supports.

Let the ***neighbors*** ($\mathcal{N}_i$) of a point x_i be either its exact or approximate k-nearest neighbors. Let ρ_i be the ***density*** of point x_i, representing how dense the local region around x_i is. A larger ρ_i value indicates a denser local region. For example, in the original DPC algorithm [80], the density of a point x is the number of points within a given radius of x, and in the SD-DP (sparse dual of density peaks) algorithm [35], the density of a point is the inverse of its distance to its k^{th} nearest neighbor. In this paper, we consider the densities that can be computed from the k-nearest neighbors of x.

Definition 1. *Let $P_i = \{x_j \mid x_j \in P \wedge \rho_j > \rho_i\}$. For x_i, its the exact **dependent point** is a point $\lambda_i \in P_i$ such that, $D(x_i, \lambda_i) \leq D(x_i, x_j)\ \forall\ x_j \in P_i$ (i.e., it is the closest point with higher density than x_i). The **dependent distance** (δ_i) of x_i is $D(x_i, \lambda_i)$, i.e., the distance to its dependent point (or ∞ if it does not have one).*

Definition 1 defines the dependent point to be the *closest* point with higher density, which is expensive to compute in high dimensions. For high-dimensional data, we relax the constraint to allow

reporting an *approximate* nearest neighbor with higher density (i.e., considering just the points with higher density, choose approximately the closest one). Roughly speaking, an ***approximate nearest neighbor*** of a point x is one whose distance from x is not too far from the distance of the true nearest neighbor from x. In our experiments, we use the Euclidean distance function, which is one of the most commonly used distance functions for clustering.

Points that are outliers and do not belong to any cluster are classified as ***noise points***. A noise point is in its own singleton cluster. For example, some algorithms require a density cutoff parameter $\rho_{\min}$, and points that have $\rho_i < \rho_{\min}$ are considered noise points. A ***cluster center*** is a point whose density is a local maximum within a cluster. Each cluster center corresponds to a separate cluster. One way to pick cluster centers is using a parameter $\delta_{\min}$, where a point x_i is considered a cluster center if $\delta_i > \delta_{\min}$.

We use the ***work-span model*** [22, 52], a standard model of computation for analyzing shared-memory parallel algorithms. The ***work*** $\mathcal{W}$ of an algorithm is the total number of operations executed by the algorithm, and the ***span*** $\mathcal{S}$ is the length of the longest sequential dependence of the algorithm (it is also the parallel time complexity when there are an infinite number of processors). We can bound the expected running time of an algorithm on $\mathcal{P}$ processors by $\mathcal{W}/\mathcal{P} + O(\mathcal{S})$ using a randomized work-stealing scheduler [11].

2.2 Relevant Techniques

Graph-based Approximate Nearest Neighbor Search. We use approximate nearest neighbor search (ANNS) algorithms in PECANN. Graph-based ANNS algorithms can find approximate nearest neighbors in high dimensions efficiently and accurately compared to alternatives such as locality-sensitive hashing, inverted indices, and tree-based indices [68, 96, 100]. These algorithms first construct a graph index on the input points, and later answer nearest neighbor queries by traversing the graph using a greedy search. Some popular methods include Vamana [55], HNSW [67], HCNNG [73], and PyNNDescent [70]. Manohar et al. [68] provides parallel implementations for constructing these indices using a prefix-doubling approach, as well as a sequential implementation for answering a single query. Multiple queries can be processed in parallel. We describe more graph-based ANNS methods in Section 7. Graph-based indices usually support any distance measure, while some indices [55, 70] use heuristics that assume the triangle inequality holds.

ANNS on a Graph Index. We use the function G.FIND-KNN(x, k) to perform an ANNS on a graph G for the point x, which returns the approximate k-nearest neighbors of x. Most graph-based ANNS methods use a variant of a *greedy (beam) search* (Algorithm 1) to answer a k-nearest neighbor query [68]. For a query point x, the algorithm maintains a ***beam*** $\mathcal{L}$ with size at most L (the ***width*** of the beam) as a set of candidates for the nearest neighbors of x.

Let $G.E_{\text{out}}(x)$ be the vertices incident to the edges going out from x in G. We call these the ***out-neighbors*** of x. On each step, the algorithm pops the closest vertex to x from $\mathcal{L}$ (Line 4), and processes it by adding all its of out-neighbors to the beam (Line 5). A set $\mathcal{V}$ is used to maintain all points that have been processed (Line 6). If $|\mathcal{L}|$ exceeds L, only the L closest points to x will be kept (Line 7). The algorithm stops when all vertices in the beam have been visited, as no new vertices can be explored (Line 3). The algorithm returns the k closest points to x from $\mathcal{L}$ and the visited point set $\mathcal{V}$ (Line 8).

Algorithm 1 Greedy Beam Search, modified from [55]

Input: Query point x, starting point set S, graph index G, beam width L, dissimilarity measure D, and integer k.

1: $\mathcal{V} \leftarrow \emptyset$ ▷ visited points
2: $\mathcal{L} \leftarrow S$ ▷ points in the beam
3: **while** $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ **do**
4: $p^* \leftarrow \text{argmin}_{(q \in \mathcal{L} \setminus \mathcal{V})} D(x, q)$
5: $\mathcal{L} \leftarrow \mathcal{L} \cup G.E_{\text{out}}(p^*)$
6: $\mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$
7: **if** $|\mathcal{L}| > L$ **then** keep only the L closest points to x in $\mathcal{L}$
8: **return** k closest points to x in $\mathcal{L} \cup \mathcal{V}$

In some circumstances (e.g. when parts of the dataset contain many near duplicates), it is possible that the algorithm will traverse fewer than k points for a query, and thus return fewer than k points. To solve this problem, options include resorting to a brute force search or repeating the search from other starting points.

Parallel Primitives. PAR-FILTER(A, f) takes as input a sequence of elements A and a predicate f, and returns all elements $a \in A$ such that $f(a)$ is true. PAR-ARGMIN(A, f) takes as input a sequence of elements A and a function $f : A \to R$, and returns the element $a \in A$ that has the minimum $f(a)$. PAR-SUM(A) takes as input a sequence of numbers A, and returns the sum of the numbers in A. PAR-FILTER, PAR-ARGMIN, and PAR-SUM all take $O(n)$ work and $O(\log n)$ span. PAR-SELECT(A, k) takes as input a sequence of elements A and an integer $0 < k \leq |A|$, and returns the k^{th} largest element in A. It takes $O(n)$ work and $O(\log n \log\log n)$ span [52]. We use the implementations of these primitives from ParlayLib [10].

A ***union-find*** data structure maintains the set membership of elements and allows the sets to merge. Initially, each element is in its own singleton set. A UNION(a, b) operation merges the sets containing a and b into the same set. A FIND(a) operation returns the membership of element a. We use a concurrent union-find data structure [54], which supports unions and finds operations in parallel. Performing m unions on a union-find data structure with n elements takes $O(m(\log(\frac{n}{m} + 1) + \alpha(n, n)))$ work and $O(\log n)$ span (α denotes the inverse Ackermann function) [54].

3 PECANN Framework

We present the PECANN framework in Algorithm 2. To make our description of the framework more concrete, we will give an example of instantiating the framework in this section. Section 4 will provide more examples and Section 5 will provide the work and span analysis of PECANN.

The input to PECANN is a point set P, a positive integer k, a distance measure D, and three functions F_{density}, F_{noise}, and F_{center} that indicate how the density, noise points, and center points are computed, respectively. In the pseudocode, ρ is an array of densities of all points in P and $\mathcal{N}$ is an array containing k-nearest neighbors for all points. λ is an array containing dependent points. c is an array containing the cluster IDs of all points and c_i is the cluster ID of x_i. The framework has the following six steps.

1. Construct Index. On Line 1, we construct an index G, which can be any index that supports k-nearest neighbor search. For example,

Algorithm 2 PECANN Framework

Input: Point set P, integer $k > 0$, distance measure D, F_{density}, F_{noise}, F_{center}

1: G = BuildIndex(P)
2: **parfor** $i \in 1 \dots n$ **do**
3: $\mathcal{N}_i \leftarrow G$.Find-Knn($x_i$, k) ▷ find k-nearest neighbors
4: **parfor** $i \in 1 \dots n$ **do**
5: $\rho_i \leftarrow F_{\text{density}}(x_i, \mathcal{N}_i)$ ▷ compute densities
6: $\lambda \leftarrow$ ComputeDepPts(G, P, ρ, $\mathcal{N}$, D)
7: $P_{\text{noise}} \leftarrow F_{\text{noise}}(P, \rho, \lambda, \mathcal{N})$ ▷ compute noise points
8: $P_{\text{center}} \leftarrow F_{\text{center}}(P \setminus P_{\text{noise}}, \rho, \lambda, \mathcal{N})$ ▷ compute center points
9: Initialize a union-find data structure UF with size $n = |P|$
10: **parfor** $x_i \in P \setminus (P_{\text{noise}} \cup P_{\text{center}})$ **do**
11: UF.Union(i, λ_i)
12: **parfor** $i \in 1 \dots n$ **do**
13: $c_i \leftarrow UF$.Find(i)
14: Return c

Algorithm 3 Dependent Point Computation

1: **function** DPBruteForce(x_i, $\mathcal{N}_{\text{candidates}}$, ρ, D)
2: $\mathcal{N}_{\text{candidates}} \leftarrow$ par-filter($\mathcal{N}_{\text{candidates}}$, $j : \rho_j > \rho_i$)
3: **if** $\mathcal{N}_{\text{candidates}} = \emptyset$ **then return** $\emptyset$
4: $\lambda_i \leftarrow$ par-argmin($\mathcal{N}_{\text{candidates}}$, $j : D(x_i, x_j)$)
5: **return** λ_i
6: **function** ComputeDepPts(G, P, ρ, $\mathcal{N}$, D)
7: **parfor** $x_i \in P$ **do**
8: $\lambda_i \leftarrow$ DPBruteForce(x_i, $\mathcal{N}_i$, ρ, D)
9: $P_{\text{unfinished}} \leftarrow$ par-filter(P, $x_i : \lambda_i = \emptyset$)
10: $k^{\text{dep}} \leftarrow L_d$ ▷ L_d is an integer parameter $> k$
11: **while** $|P_{\text{unfinished}}| >$ threshold **do**
12: **parfor** $x_i \in P_{\text{unfinished}}$ **do**
13: $\mathcal{N}_{\text{candidates}} \leftarrow G$.FindKnn($i$, k^{dep})
14: $\lambda_i \leftarrow$ DPBruteForce(x_i, $\mathcal{N}_{\text{candidates}}$, ρ, D)
15: $k^{\text{dep}} \leftarrow 2 \cdot k^{\text{dep}}$
16: $P_{\text{unfinished}} \leftarrow$ par-filter($P_{\text{unfinished}}$, $x_i : \lambda_i = \emptyset$)
17: **parfor** $x_i \in P_{\text{unfinished}}$ **do**
18: $\lambda_i \leftarrow$ DPBruteForce(x_i, P, ρ, D)
19: **return** λ

it can be a kd-tree, which is suitable for low-dimensional exact k-nearest neighbor search [37], or a graph-based index for ANNS on high-dimensional data [55, 67, 68, 70, 73]. It can also be an empty data structure, which would lead to doing brute force searches to find the exact k-nearest neighbors. An example of a graph index corresponding to a point set is shown in Figure 1.

2. Compute k-nearest Neighbors. On Lines 2–3, we compute the k-nearest neighbors of all points in parallel, using the index G. If we run the greedy search (Algorithm 1) on the example in Figure 1 with $k = 1, L = 1$, and S containing only the query point, we would find that the nearest neighbors of a, b, c, d, e, and f are c, c, b, f, d, and d, respectively (in this example, we assume that the graph index returns exact nearest neighbors).

3. Compute Densities. On Lines 4–5, we compute the density for each point in parallel using F_{density}. An example density function is $\frac{1}{D(x_i, x_j)}$, where x_j is the furthest neighbor from x_i in $\mathcal{N}_i$ [35]. For this density function, the densities of the points in Figure 1 are $\rho_a = \frac{1}{\sqrt{2}}$, $\rho_b = 1$, $\rho_c = 1$, $\rho_d = \frac{1}{\sqrt{2}}$, $\rho_e = \frac{1}{2}$, and $\rho_f = \frac{1}{\sqrt{2}}$. The ranking of the densities from high to low (breaking ties by node ID) is b, c, a, d, f, e.

4. Compute Dependent Points. On Line 6, we compute the dependent point of all points in parallel. The dependent points in our example are shown in Figure 2. We explain the details of how we compute the dependent points in Section 3.1. As mentioned in Section 1, the resulting tree from this step is a hierarchy of clusters (dendrogram), which can be returned if desired. To compute a specific clustering, the following two steps are needed.

5. Compute Noise and Center Points. On Lines 7–8, we compute the noise points and center points using the input functions F_{noise} and F_{center}. An example of F_{noise} is `par-filter`$(P, x_i : \rho_i > \rho_{\min})$, where $\rho_{\min}$ is a user-defined parameter. Points whose densities are at most $\rho_{\min}$ are classified as noise points. An example of F_{center} is `par-filter`$(P, x_i : D(x_i, \lambda_i) \geq \delta_{\min})$, where $\delta_{\min}$ is a user-defined parameter. Non-noise points whose distance are at least $\delta_{\min}$ from their dependent point are classified as center points. In our example (Figure 3), if we let $\rho_{\min} = \frac{1}{\sqrt{2}}$, then e is a noise point. If we let $\delta_{\min} = 2.5$, then b and d are center points.

6. Compute Clusters. On Lines 9–13, we compute the clusters using a concurrent union-find data structure [54]. In parallel, for all points that are not noise points or center points, we merge them into the same cluster as their dependent point. This ensures that points (except noise points and center points) are in the same cluster as their dependent point. Figure 3 shows the clustering obtained on our example. Since e is a noise point and b and d are center points, we skip processing their outgoing edge during the union step (Line 11 of Algorithm 2).

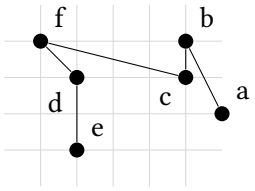

Figure 1: Example point set and a corresponding graph index.

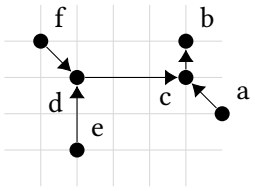

Figure 2: Each point has an edge pointing to its dependent point.

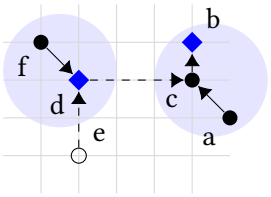

Figure 3: Clustering result with e as a noise point (white circle), and b and d as center points (blue diamond shaped). The dashed edges are ignored during the union step (Line 11 of Algorithm 2). The two blue circles are the two clusters found.

3.1 Dependent Point Computation

Our parallel algorithm for computing the dependent points (Algorithm 3) takes as input the index G, the point set P, the array of

densities ρ, the array of (approximate) k-nearest neighbors $\mathcal{N}$, and the distance measure D.

DPBruteForce is a helper function (Lines 1–5) that searches for the nearest neighbor of x_i with density higher than ρ_i among $\mathcal{N}_{\text{candidates}}$ using brute force. It returns $\emptyset$ if no points in $\mathcal{N}_{\text{candidates}}$ have a higher density than ρ_i.

On Lines 7–8, we first search within the k-nearest neighbors of each point to find its dependent point. This optimization is also used in several other works [16, 35, 91]. On Line 9, we obtain the set of points $P_{\text{unfinished}}$ that have not found their dependent points. Line 10 initializes k^{dep} to L_d.

L_d and `threshold` are parameters used for our performance optimizations. We defer a discussion of these parameters to Section 3.2, and ignore their effect here by setting L_d to be $2k$ and `threshold` to be 0 (this causes Lines 17–18 to have no effect, since $P_{\text{unfinished}}$ will be empty at that point).

The while-loop on Line 11 terminates when all points have found their dependent point. On Lines 12–14, we compute the dependent point for points in $P_{\text{unfinished}}$. If the index is designed for approximate k-nearest neighbor search, we guarantee that the dependent point has a higher density, but it might not be the closest among points with higher densities. Note that on Line 12, we can skip the point with maximum density, since we know that it does not have a dependent point. On Lines 13–14, for each point, we find k^{dep} neighbors of x_i on each round, and if any of the neighbors have a higher density than x_i, we can return the closest such neighbor as the dependent point. We then double k_i^{dep} for the next round (Line 15). A similar doubling optimization is used in Chen et al. [16], but with a cover tree. Furthermore, their algorithm is sequential. On Line 16, we compute the set of points $P_{\text{unfinished}}$ that have not found their dependent point.

Example. On the example dataset from Figure 1, points a, c, e, and f would find their dependent point within their k-nearest neighbor ($k = 1$) on Lines 7–8 because their nearest neighbor has higher density than themselves. b is the point with maximum density and is skipped. For the remaining point d, on the first round we have $k^{\text{dep}} = 2$, and so $\mathcal{N}_{\text{candidates}} = \{e, f\}$. This does not contain any point with a higher density than d, and so we double $K^{\text{dep}} = 2$ and try again. On the second round, $k^{\text{dep}} = 4$, and so $\mathcal{N}_{\text{candidates}} = \{b, c, e, f\}$, which contains d's dependent point c.

3.2 Performance Optimizations

Dependent Point Finding. Now we explain the two integer parameters L_d and `threshold`. The while-loop on Line 11 checks if $|P_{\text{unfinished}}| >$ `threshold`, and when that is no longer true, we do a brute force k-nearest neighbor computation for the remaining points in $P_{\text{unfinished}}$ on Lines 17–18. This optimization is useful because for the points with relatively high density, it can be challenging for the index to find a dependent point (as most neighbors have lower density than them), and for these last few points it is faster to just brute force search than continue to double k^{dep}. Furthermore, when few points are remaining, there is less parallelism available when calling FindKNN, each of which is sequential, compared to the brute force search, which is highly parallel. In our experiments, we set `threshold` to 300, which we found to work well in practice.

L_d is a tunable parameter that is $> k$ (Line 10) and indicates the initial number of nearest neighbors to search for to find a dependent point (Line 13). A larger value of L_d leads to fewer iterations. However, points that require fewer than L_d nearest neighbors to find a dependent point will do some extra work (as they search for more nearest neighbors than necessary). On the other hand, points that require at least L_d nearest neighbors to find a dependent point will do less work overall (they do not need to waste work on the initial rounds where they would not find a dependent point anyway).

Vamana Graph Construction. Vamana [55, 68] is one of the graph-based indices that we use for ANNS. Its parallel construction algorithm [68] builds the graph by running greedy search (from Algorithm 1) on each point x_i (in batches), and then adds edges from x_i to points visited during the search ($\mathcal{V}$). It requires a degree bound parameter R, such that in the constructed graph each vertex has at most R out-neighbors. If adding edges between x_i and $\mathcal{V}$ causes a vertex's degree to exceed R, a pruning procedure is called to iteratively select at most R out-neighbors. The pruning algorithm also has a parameter $\alpha \geq 1$ that controls how aggressive the pruning is; a higher α corresponds to more aggressive pruning, which can lead to less than R neighbors being selected. Intuitively, this heuristic prunes the long edge of a triangle, with a slack of α. The details of the pruning algorithm can be found in [55].

The original Vamana graph construction algorithm [55, 68] starts the greedy search from a single point, which is the medoid of P. Starting from a single point can make the algorithm require a high degree bound and beam width to achieve good results on clustered data because a search can be trapped within the cluster that the medoid is in. To improve performance without needing a large degree bound and beam width (which degrades performance), we use an optimization where we randomly sample a set of starting points for the Vamana graph construction algorithm instead of starting from the medoid alone. This random sampling heuristic is also explored in [62].

4 Usage of PECANN

PECANN allows users to plug in functions that can be combined to obtain new clustering algorithms. In this section, we describe several functions and provide their work and span bounds. Section 5 gives the overall bounds of PECANN using the combination of functions that gives the best performance in practice.

4.1 Indices

Here we describe several approaches for building indices for k-nearest neighbor search. Let the work and span of constructing G be $\mathcal{W}_c$ and $\mathcal{S}_c$, respectively.

Brute Force. The brute force approach corresponds to not having an index at all. When searching for the exact k-nearest neighbors of x_i, it uses a par-select to find the k^{th} smallest distance to x_i, and a par-filter to filter for the points with smaller distances to x_i. In this case, $\mathcal{W}_c$ and $\mathcal{S}_c$ are $O(1)$, while $\mathcal{W}_{nn}$ and $\mathcal{S}_{nn}$ are $O(n)$ and $O(\log n \log\log n)$, respectively.

Tree Indices. Another option is to use a tree index, such as a kd-tree or a cover tree [16]. For a parallel kd-tree, $\mathcal{W}_c = O(n \log n)$

and $\mathcal{S}_c = O(\log n \log\log n)$ [99]. A parallel cover tree can be constructed in $O(n \log n)$ expected work and $O(\log^3 n \log\log n)$ span with high probability [43]. A k-nearest neighbor search in a kd-tree takes $O(n)$ work and $O(\log n)$ span. A k-nearest neighbor search in a cover tree takes $O(c^7(k + c^3)\log k \log \Delta)$ expected work and span [31, 32, 43], where c is the expansion constant of P and Δ is the aspect ratio of P. However, note that these tree indices usually suffer from the curse of dimensionality and do not perform well on high-dimensional datasets.

Graph Indices. Graph-based ANNS algorithms have been shown to be efficient and accurate in finding approximate nearest neighbors in high dimensions [68, 96, 100]. Our framework includes three parallel graph indices from the ParlayANN library [68]: Vamana [55], HCNNG [73], and PyNNDescent [70]. Similar to Vamana, HCNNG also uses the parameter α to prune edges. HCNNG and PyNNDescent also accept a *num_repeats* argument, which represents how many times they will independently repeat the construction process before merging the results together.

When the number of returned neighbors is less than k, we use the brute force method to find the exact k-nearest neighbors. Although these graph indices have been shown to work well in practice, there are only a few works that theoretically analyze their query complexity [50, 59, 74, 77, 86]. Indyk and Xu [50] show that Vamana construction takes $\mathcal{W}_c = O(n^2(R + \log n))$ work. In practice, we find that the work is usually much lower. Using the batch insertion method [68], which inserts points in batches of doubling size, Vamana construction takes $\mathcal{S}_c = O(n \log n(R + \log n))$ span.

4.2 Density, Center, and Noise Functions

In this section, we describe a subset of the density, center, and noise functions (F_{density}, F_{center}, and F_{noise}) that we implement in PECANN. We describe other functions we implement in Section 9.

`kth` Density Function. The density of x_i is $\rho_i = \frac{1}{D(x_i, x_j)}$ where x_j is the furthest neighbor from x_i in $\mathcal{N}_i$, i.e., the distance to the exact or approximate k^{th} nearest neighbor of x_i [16, 35]. Each density computation is $O(k)$ work and $O(\log k)$ span to find the furthest neighbor in $\mathcal{N}_i$.

The density can also be normalized [47]. The normalized density (**`normalized`**) is $\rho_i' = \frac{\rho_i k}{\sum_{j \in \mathcal{N}_i} \rho_j}$. Intuitively, this function normalizes a point's density with an average of the densities of its neighbors. Each normalization takes an extra $O(k)$ work and $O(\log k)$ span.

Threshold Center Function. Recall from Section 2 that $\delta_i = D(x_i, \lambda_i)$ is the dependent distance of x_i. F_{center} obtains the center points by selecting the points whose distance to their dependent point is greater than $\delta_{\min}$, a user-defined parameter. This can be implemented with a `par-filter`, whose work and span are $O(n)$ and $O(\log n)$, respectively. This method is used in [5, 6, 107].

Product Center Function. This method takes as input n_c, a user-defined parameter that specifies how many clusters to output. We compute the product $\delta_i \times \rho_i$ for all points x_i. The n_c points with the largest products are the center points. This function can be implemented with a PAR-SELECT to find the n_c^{th} largest product t, and then a PAR-FILTER to filter out the points with product less than t. The work and span are $O(n)$ and $O(\log n \log\log n)$, respectively. This method is used in [47, 56, 64, 80].

Noise Function. We implement a noise function F_{noise}, which returns the points x_i with density $\rho_i < \rho_{\min}$. These points are then ignored in the remainder of the algorithm. This can be implemented using a parallel filter with $O(n)$ work and $O(\log n)$ span. A noise function of this form is used by [5, 6, 80].

5 Analysis of PECANN

5.1 Work and Span Analysis

The work and span of PECANN (Algorithm 2) depend on the specific index construction algorithm and the specific implementations of the functions F_{density}, F_{noise}, and F_{center}. Here, we will choose the functions that give the best performance in our experiments (`kth` density, product center, and default noise functions).

We first analyze the work and span of computing dependent points as shown in Algorithm 3 (this is called on Line 6 of Algorithm 2). Let $n_{\text{can}} = |\mathcal{N}_{\text{candidates}}|$. Lines 1–5 take $O(n_{\text{can}})$ work and $O(\log n_{\text{can}})$ span. Thus, Lines 7–8 take $O(nk)$ work and $O(\log k)$ span, because $|\mathcal{N}_i| = k$ and $|P| = n$. Line 9 takes $O(n)$ work and $O(\log n)$ span.

On Lines 11–16, for each point, we call G.FindKnn $O(\log n)$ times since we double k^{dep} after each round. Let the work and span of finding the k nearest neighbors using G be $\mathcal{W}_{nn}(k)$ and $\mathcal{S}_{nn}(k)$, respectively. Let $\mathcal{W}_{nn} = \sum_{j=0}^{O(\log n)} \mathcal{W}_{nn}(2^j)$ and $\mathcal{S}_{nn} = \sum_{j=0}^{O(\log n)} \mathcal{S}_{nn}(2^j)$. The filter on Line 16 takes $O(n \log n)$ work and $O(\log^2 n)$ span across $O(\log n)$ rounds. The total work and span across all rounds is $O(n\mathcal{W}_{nn})$ and $O(\mathcal{S}_{nn} + \log^2 n)$. The brute force computation on Lines 17–18 takes $O(n)$ work and $O(\log n)$ span, as only $O(1)$ points remain after the loop on Lines 11–16.

Thus, the work and span of Algorithm 3 are $O(n\mathcal{W}_{nn})$ and $O(\mathcal{S}_{nn} + \log^2 n)$, respectively.

We now analyze the remaining steps of Algorithm 2. Lines 2–3 compute the k-nearest neighbors of all points, which takes $O(n\mathcal{W}_{nn}(k))$ work and $O(n\mathcal{S}_{nn}(k))$ span. Lines 4–5 compute the densities of all points. Using the `kth` density function, this takes $O(nk)$ work and $O(\log k)$ span. Lines 7–8 using the product center and default noise functions take $O(n)$ work and $O(\log n \log\log n)$ span. The union-find operations on Lines 9–13 take $O(n\alpha(n, n))$ work and $O(\log n)$ span.

The following theorem gives the overall work and span of PECANN.

Theorem 1. *The work and span of PECANN using the k^{th} density, product center, and the default noise functions are $O(\mathcal{W}_c + n\mathcal{W}_{nn})$ and $O(\mathcal{S}_c + \mathcal{S}_{nn} + \log^2 n)$, respectively.*

5.2 Approximation Analysis

In this section, we give a brief analysis of the approximation guarantees of PECANN. Proofs and more detailed analyses can be found in Section 10. Our analysis of the density approximation is based on the `kth` density function described above. Our analysis of the approximate dependent point computation is based on the threshold center function described above.

5.2.1 Density Estimation Assuming some guarantee in approximate k-nearest neighbor search, we can show that the density peaks of the exact algorithm that do not *conflict* with other points will remain density peaks. A conflict occurs when the density ranges

Name	n	d	Description	# Clusters
gaussian	10^5 to 10^8	128	Standard benchmark	10 to 10000
MNIST	70,000	784	Raw images	10
ImageNet	1,281,167	1024	Image embeddings	1000
birds	84,635	1024	Image embeddings	525
reddit	420,464	1024	Text embeddings	50
arxiv	732,723	1024	Text embeddings	180

Table 2: The datasets used in our experiments, along with their sizes (n), their dimensionality (d), and the number of ground truth clusters.

of two points overlap. The density range of a point bounds the approximate density value of the point.

Lemma 2. *Consider the threshold center function, which obtains the center points by selecting the points whose distance to their dependent point is greater than $\delta_{\min}$. If the density interval of a point does not conflict with any other interval and it is a true density peak, then it is still a density peak in PECANN given the same threshold $\delta_{\min}$.*

Note that there may be additional density peaks returned by the approximate algorithm, but the true density peaks in the exact algorithm are guaranteed to still be density peaks.

5.2.2 Dependent Point Estimation Now we analyze the approximate dependent point found by Algorithm 3. The following lemma guarantees that the approximate dependent points returned by our algorithm are not too much further than the true dependent points. Let d_j be the distance to the true j^{th} nearest neighbor from query point q. As far as we know, other approximate DPC methods [4, 5, 41] do not provide approximation bound on approximate dependent point search.

Lemma 3. *Suppose we find the approximate dependent point among the βk-approximate nearest neighbor, for $\beta \geq 1$. The approximate dependent point is at most $c^2 \frac{d_{\beta k}}{d_k}$ further from the exact dependent point given the same densities for some constant $c \geq 1$.*

In Algorithm 3, we use $\beta = 2$ for Lemma 4, since we double the number of nearest neighbors to find until we have found a dependent point.

6 Experiments

6.1 Experimental Setup

Computational Environment. We use *c2-standard-60* instances on the Google Cloud Platform. These are 30-core machines with two-way hyper-threading with Intel 3.1 GHz Cascade Lake processors that can reach a max turbo clock-speed of 3.8 GHz. The instances have two non-uniform memory access (NUMA) nodes, each with 15 cores. Except for the experiments specifically investigating how performance scales with the number of threads, we use all 60 hyper-threads for our experiments.

Datasets. We use a variety of real-world and artificial datasets, summarized in Table 2 and described below.

- gaussian is a synthetic mixture of datasets generated from a Gaussian distribution. To generate a gaussian dataset of dimension $d = 128$ with size n and c clusters, we first sample c centers x_i uniformly from $[0, 1]^d$, and then we sample n/c points from a Gaussian centered at each x_i with variance 0.05.

Dataset	L	L_d	R	k
MNIST	32	32	32	16
ImageNet	128	128	128	16
reddit, arxiv	64	64	64	16
gaussian, birds	32	32	32	16

Table 3: Default parameters used for datasets.

- MNIST [24] is a standard machine learning dataset that consists of 28×28 dimensional images of grayscale digits between 0 and 9. The i^{th} cluster corresponds to all occurences of digit i.
- ImageNet [23] is a standard image classification benchmark with more than one million images, each of size 224×224×3. The images are from 1000 classes of everyday objects. Unlike for MNIST, we do not cluster the raw ImageNet images, but instead first pass each image through ConvNet [65] to get an embedding. Each ground truth cluster contains the embeddings corresponding to a single image class from the original ImageNet dataset.
- birds [39] is a dataset that contains images of 525 species of birds. The images have the same number of dimensions as ImageNet, and we pass it through the same ConvNet model to obtain an embedding dataset. The ground truth clusters are the 525 species of birds. This dataset is interesting because it is out of distribution for the original ConvNet model.
- reddit and arxiv are text embedding datasets studied in the recent Massive Text Embedding Benchmark (MTEB) work [72]. We restrict our attention to embeddings from the best model on the current MTEB leaderboard, GTE-large [61]. We also restrict our attention to the two largest datasets from MTEB, reddit, where the goal is to cluster embeddings corresponding to post titles into subreddits, and arxiv, where the goal is to cluster embeddings corresponding to paper titles into topic categories.

Algorithms. We implement our algorithms using the ParlayLib [10] and ParlayANN [68] libraries. We use C++ for all implementations, and the gcc compiler with the -O3 flag to compile the code. We also provide Python bindings for PECANN. We evaluate the following algorithms.

- PECANN: Our framework described in Section 3 with the different density functions described in Section 4. Unless specified otherwise, we use the kth density function without normalization with $k = 16$, the Vamana graph index with $\alpha = 1.1$, and the product center function with n_c set to the number of ground truth clusters, and the default noise function. In Table 3, we give the rest of the default parameters that we used for each dataset.
- fastdp [87]: A single-threaded approximate DPC algorithm that also uses graph-based ANNS to estimate densities. As a conservative estimate, we assume that fastdp can be modified to have ideal parallel speedup, and denote this with fastdp (scaled). In other words, we divide all fastdp runtimes by 60 in plots to get fastdp (scaled).
- k-means: The FAISS [57] implementation of k-means, an extremely efficient k-means implementation. It is parallelized by using parallel k-nearest neighbor search. The k-means algorithm takes in k, the number of clusters, *niter*, the number of iterations, and *nredo*, the number of times to retry and choose the best clustering. Unless specified otherwise, the number of clusters used in k-means is the number of clusters in the ground truth clustering.
- BruteForce: An instantiation of PECANN, where we use a naive parallel brute force approach for every step. This method takes

$O(n^2)$ work. It also first searches within the k-nearest neighbors to find the dependent point. We refer to the result of BruteForce as the "exact DPC" result.

- DBSCAN: A popular density-based clustering algorithm for low-dimensional data [34, 83]. We use the DBSCAN implementation in scikit-learn [75] for high-dimensional datasets, which is parallelized with parallel nearest neighbor search. We also tried Wang et al. [98]'s parallel implementation, which is optimized for low-dimensional data, and found it slower than scikit-learn on high-dimensional data. DBSCAN has two parameters ϵ and min_pts: ϵ defines the maximum distance between two points to be considered neighbors. min_pts specifies the minimum number of points required to form a dense region (core point) and thereby trigger the formation of a cluster.

We also tried a parallel exact DPC algorithm that uses a priority search kd-tree-based dependent point finding algorithm that was designed for low dimensions [48]. We changed the first step of Huang et al. [48] from a range search to a k-nearest neighbor search to match our framework. On `MNIST`, their algorithm takes 280s on our 30-core machine, which is 320 times slower than PECANN. This method is prohibitively slow because kd-trees suffer from the curse of dimensionality, where performance in high dimensions degrades to no better than a linear search [101]. We thus do not further compare against this method.

Evaluation. We evaluate clustering quality using the Adjusted Rand Index (ARI) [49], homogeneity, and completeness [81].

Consider our clustering C and the ground-truth or exact clustering $\mathcal{T}$. Intuitively, ARI evaluates how similar C and $\mathcal{T}$ are. Homogeneity measures if each cluster in C contains members from the same class in $\mathcal{T}$. Completeness measures whether all members in $\mathcal{T}$ of a given class are in the same cluster in C.

Let n_{ij} be the number of objects in the ground truth cluster i and the cluster j generated by the algorithm, n_{i*} be $\sum_j n_{ij}$, n_{*j} be $\sum_i n_{ij}$, and n be $\sum_i n_{i*}$. The ARI is computed as

$$\frac{\sum_{i,j}\binom{n_{ij}}{2} - [\sum_i \binom{n_{i*}}{2}\sum_j \binom{n_{*j}}{2}]/\binom{n}{2}}{\frac{1}{2}[\sum_i \binom{n_{i*}}{2} + \sum_j \binom{n_{*j}}{2}] - [\sum_i \binom{n_{i*}}{2}\sum_j \binom{n_{*j}}{2}]/\binom{n}{2}}.$$

The ARI score is 1 for a perfect match, and its expected value is 0 for random assignments.

The formulas for homogeneity and completeness of clusters are defined as follows: homogeneity $= 1 - \frac{H(C|\mathcal{T})}{H(C)}$; completeness $= 1 - \frac{H(\mathcal{T}|C)}{H(\mathcal{T})}$. $H(C|\mathcal{T})$ is the conditional entropy of the class distribution given the cluster assignment, $H(C)$ is the entropy of the class distribution, $H(\mathcal{T}|C)$ is the conditional entropy of the cluster distribution given the class, and $H(\mathcal{T})$ is the entropy of the cluster distribution. For example, consider a ground-truth clustering $\mathcal{T}$ where all classes have the same number of points. If C assigns every point to its own cluster of size 1, it has homogeneity score 1 and a low completeness score when $n_c \ll n$. On the other hand, if C assigns all points to a single cluster, it has completeness score 1 and homogeneity score 0.

6.2 Scalability

Figure 4 shows the parallel scalability of PECANN on our larger datasets. PECANN achieves an average of 14.36x self-relative speedup on one NUMA node with 30 hyper-threads and an average of 16.57x self-relative speedup on two NUMA nodes with 60 hyper-threads.

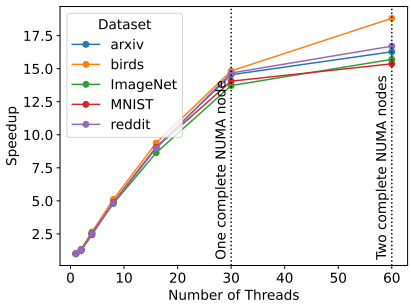

Figure 4: Self-relative parallel speedup across different numbers of hyper-threads.

Dataset	k	Index	Density	Dependent Point	Union-Find
`arxiv`	8	**60.9**	35.0	4.1	0.0
`arxiv`	32	**52.5**	45.6	1.9	0.0
`MNIST`	8	**53.4**	43.7	2.6	0.3
`MNIST`	32	34.8	**63.7**	1.4	0.1
`reddit`	8	**54.6**	36.9	8.4	0.1
`reddit`	32	**58.6**	40.1	1.2	0.1
`birds`	8	**72.0**	20.8	6.0	1.2
`birds`	32	**55.2**	38.8	5.1	0.9
`ImageNet`	8	**70.0**	27.3	2.6	0.1
`ImageNet`	32	**59.0**	39.0	2.0	0.0

Table 4: Runtime percentage breakdown with the `kth` density method with $k = 8, 32$ on large datasets.

We also study the runtime of PECANN as we increase the size of the synthetic `gaussian` dataset and vary the number of clusters between 10 to 10,000 (Figure **??**). We use a linear fit on the logarithm of runtime and $\log n$ to obtain the slopes of the lines in **??**. The slope s reflects the exponent in the growth of runtime with respect to data size. We find that the slope ranges from 1.12–1.2 depending on the number of output clusters, and thus experimentally the clustering grows approximately as $O(n^{1.2})$ for this dataset. This shows that PECANN has good scalability with respect to n.

6.3 Runtime Decomposition

We present the runtime decomposition of PECANN on each dataset with all density methods and all values of k in Figure 5. As a summary, Table 4 presents the runtime breakdown for the `kth` density method for $k = 8, 32$. The bottleneck of the running time is the index construction time and the k-nearest neighbor time when computing densities. When k is larger, the k-nearest neighbor search time for density computation is longer, as expected. Computing clusters with union-find is fast because this step has low work, as discussed in Section 4. The dependent point computation time is significantly shorter than the density computation because the dependent point for some points can be obtained from the k-nearest neighbors (Lines 7–8 in Algorithm 3), so we do not need to run nearest neighbor searches for these points. Additionally, even when the dependent point is not in the k-nearest neighbors, our doubling

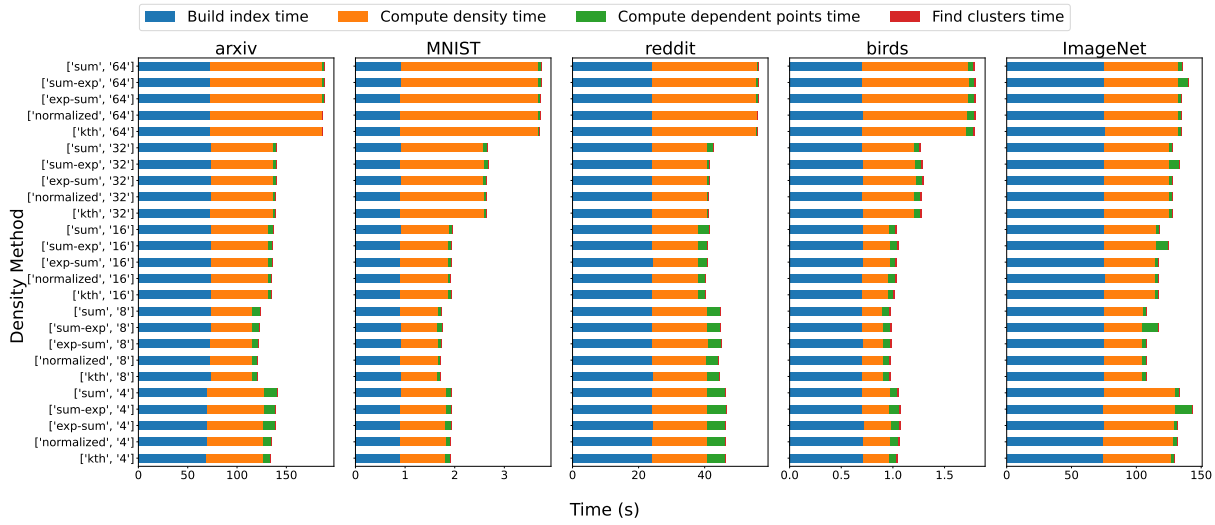

Figure 5: Runtime decomposition of PECANN with different density functions and values of k.

technique finds a dependent point in the first few rounds for most points, thereby usually avoiding an expensive exhaustive search.

6.4 Comparison of Different Density Functions, Values of k, and Different Graph Indices

In Figure 6, we show the runtime vs. ARI of different density functions and values of k. We see that the `kth` density function is the most robust and achieves the highest ARI score on most datasets. We also observe that using $k = 16$ provides a good trade-off between quality and time. `exp-sum`, `sum`, and `sum-exp` are other density functions in PECANN that are described in our full paper. These density functions are combinations of the distances to the k-nearest neighbors.

We can easily swap in different graph indices into our framework and compare the results. In Figure 7, we show a Pareto frontier of the clustering quality vs. runtime on `ImageNet` for each of the following different graph indices: Vamana [55], pyNNDescent [70], and HCNNG [73]. The Pareto frontier comprises points that are non-dominated, meaning no point on the frontier can be improved in quality without worsening time and vice versa. In other words, the curve we plot represents the optimal trade-off in the parameter space between clustering time and quality.

To create the Pareto frontier, we do a grid search for each method over different choices of maximum degree R and the beam sizes for construction, k-nearest neighbor search, and dependent point finding. We choose all combinations of these four parameters from $[8, 16, 32, 64, 128, 256]^4$. We set the density method to be `kth` without normalization and $k = 16$. We set $\alpha = 1.1$ for Vamana and PyNNDescent. HCNNG and PyNNDescent additionally accept a *num_repeats* argument, which represents how many times we should independently repeat the construction process before merging the results together; we set this parameter equal to 3. We see that all graph indices are able to achieve similar maximum ARI with respect to the ground truth: Vamana, HCNNG, and PyNNDescent achieve maximum ARIs of 0.709, 0.715, and 0.713, respectively. HCNNG attains this maximum slightly faster than the other two indices, but when compared to the exact DPC result, `HCNNG` has a smaller maximum ARI, which means its clustering deviates more from the exact solution. Indeed, HCNNG has a maximum ARI compared to exact DPC of 0.918, while PyNNDescent and Vamana attain a maximum ARI of 0.995 compared to exact DPC.

Hyperparameter Regression Analysis. We also run a linear regression for each of our five main datasets to predict the clustering time and the ARI from the four Vamana hyperparameters: the maximum degree R for graph construction, and the three beam size hyperparameters for graph construction, k-nearest neighbor search, and dependent point finding. We use the log of each of the hyperparameters for the ARI regression. Averaging across the five regressions, the ARI regressions have an average R^2 of 0.714 and the hyperparameters have average linear regression weights 0.125, 0.139, 7.69e−3, and 1.25e−3, respectively, while the clustering time regressions have an average R^2 of 0.783 and average weights of 0.181, 0.360, 0.0429, and 1.37e−4, respectively. In summary, the maximum degree of the graph and construction beam size have both the largest contribution to the ARI and the largest impact on the clustering time.

6.5 Clustering Quality-Time Trade-off

In Figure 8, we plot the Pareto frontier of clustering quality (ARI with respect to the exact DPC clustering) vs. runtime of PECANN. To obtain the Pareto frontiers, we use the same parameter values as in the last experiment, except that for the smaller datasets with $n < 250,000$ we use a smaller range $[8, 16, 32, 64]^4$ for the parameter search space. We see that PECANN can achieve results very close to the exact DPC clustering. On all datasets except `arxiv`, PECANN achieves at least 0.995 ARI with respect to exact DPC, and on `arxiv`, PECANN achieves 0.989 ARI with respect to exact DPC.

6.6 Comparison of Different Methods

In Figure 9, we plot the Pareto frontier of clustering quality (ARI with respect to the ground truth clustering) vs. runtime for different methods on the larger datasets. To obtain the Pareto frontiers,

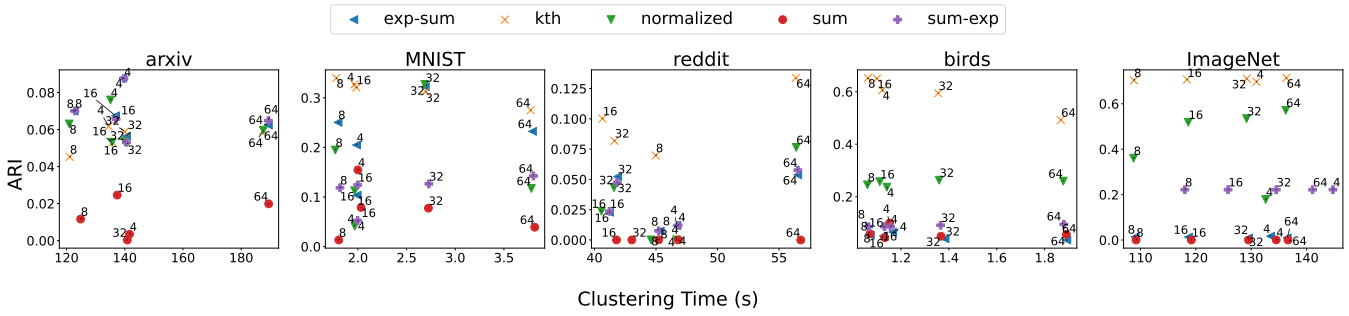

Figure 6: Clustering quality (ARI) vs. runtime of PECANN when using different density functions and values of k. The y-axis shows the ARI scores computed with respect to the ground truth. The x-axis shows the runtime in seconds. Each color represents a density function, and the number next to each data point is the value of k used.

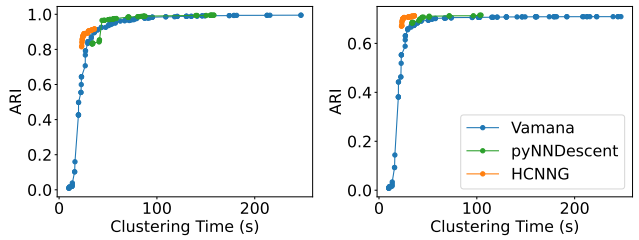

Figure 7: (Left) Pareto frontier of clustering quality with respect to *exact DPC* vs. runtime on `ImageNet`. (Right) Pareto frontier of clustering quality with respect to the *ground truth clustering* vs. runtime on `ImageNet`.

we use the same parameters for Vamana as in the previous experiment. For k-means, we use $nredo \in [1, 2, 3, 4]$ and $niter \in [1, 2, 3, \ldots, 9, 10, 15, 20, 25, \ldots, 40, 45]$, for all combinations where $niter{*}nredo < 100$. For fastdp, we use $window\_size \in [20, 40, 80, 160, 320]$ for all datasets (controlling query quality) and $max\_iterations \in [1, 2, 4, 8, 16, 32, 64]$ (controlling graph construction quality). For DBSCAN, we chose different parameters for each dataset. Sander et al. [82] suggest setting min_pts to $2d-1$. For high dimensional datasets, it may improve results to increase min_pts [83]. ϵ is chosen based on the distribution of the min_pts-nearest neighbor distances [78]. We follow these guidelines when choosing the parameters. The parameters can be found in Section 12.

We observe that DBSCAN has lower quality and higher runtime than all other baselines. As the original authors of DBSCAN state, it is difficult to use DBSCAN for high-dimensional data [83].

We observe that fastdp, even when given ideal parallel speedup, is slower that PECANN on all datasets except `MNIST`, where the two methods have similar performance. In terms of accuracy, PECANN has better maximum ARI on `birds` and `arxiv`, while fastdp has better maximum ARI on `reddit` (although as we discuss below, `reddit` is not well suited to DPC).

Compared with k-means, PECANN obtains better quality and is faster on `ImageNet` and `birds`, where the number of ground truth clusters is large, and performs about equal with k-means on `arxiv`. However, PECANN has worse quality for a given time limit on `reddit` and `mnist`. Although PECANN has worse quality than k-means on two datasets when k-means uses the correct number of clusters, k-means's quality is sensitive to the number clusters. As shown in Section 6.7, k-means can have lower quality than PECANN on these two datasets when k does not match the number of ground truth clusters.

We summarize the best ground truth ARI and the corresponding parallel running time that all these methods achieve, as well as BruteForce, in Table 5. Compared to density-based methods, PECANN achieves 37.7–854.3x speedup over BruteForce, 45-734x speedup over fastdp, and 0.76–12.24x speedup over fastdp (scaled), while achieving comparable ARI. PECANN also achieves up to 0.7 higher ARI than DBSCAN, and is up to orders-of-magnitude faster.

For some more intuition on the runtime differences between PECANN and k-means, we note that the work of each iteration of k-means is linear in the number of clusters multiplied by n, and so k-means is very fast on datasets like `MNIST` with a small number of ground truth clusters, while it is comparatively slower on datasets like `birds` and `imagenet` that have many clusters.

In terms of an explanation for the quality difference between PECANN and k-means, PECANN gets better maximum accuracy on `imagenet` and `birds`, which may be because the ground truth clusters in these datasets form shapes that our density-based method PECANN can find, but that the geometrically constrained k-means cannot. On the other hand, for `reddit`, PECANN has lower quality than k-means. Since we still obtain cluster quality very close to the exact DPC on `reddit` (see Figure 8), this dataset is a case where the density based DPC method is worse than the simpler k-means heuristic.

6.7 Varying Number of Clusters

In Figure 10, we show the ARI (with respect to ground truth) of PECANN using Vamana and k-means when we pass a number of clusters to the algorithm different than the ground truth (we plot the ratio between the number of clusters used and the number of ground truth clusters). Not knowing the number of ground truth clusters is common in real-world settings, so algorithm performance in this regime is important. We see that PECANN is better than k-means when the number of clusters used is larger than the true number of clusters (except on `reddit`, where we have argued above that DPC is not suitable). When the number of clusters used is larger, the quality of k-means decays quickly while the quality of PECANN is more robust. When the true number of clusters used is smaller than the ground truth, the quality of the two methods is similar

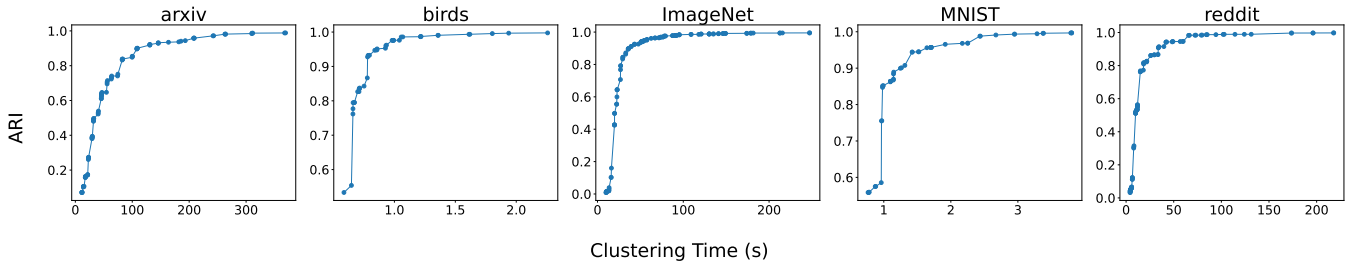

Figure 8: Pareto frontier of clustering quality of PECANN with respect to exact DPC.

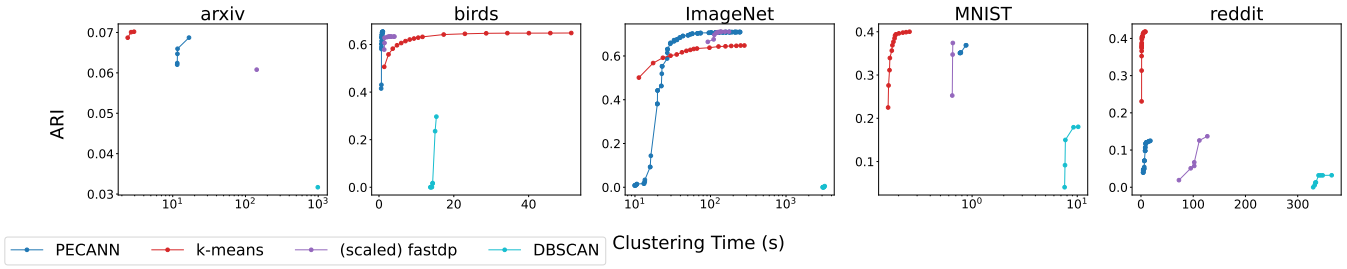

Figure 9: Pareto frontier of clustering quality of all methods with respect to ground truth vs. runtime. Up and to the left is better. PECANN is the best method on `ImageNet` and `birds`, has similar performance to the best method (k-MEANS) on `arxiv`, and is slower or has worse quality than the best method (k-means) on `mnist` and `reddit`. FASTDP is sequential, and in this plot we assume it achieves a perfect speedup of 60 to obtain FASTDP (SCALED). The x-axis on `arxiv`, `ImageNet`, and `MNIST` are in log-scale.

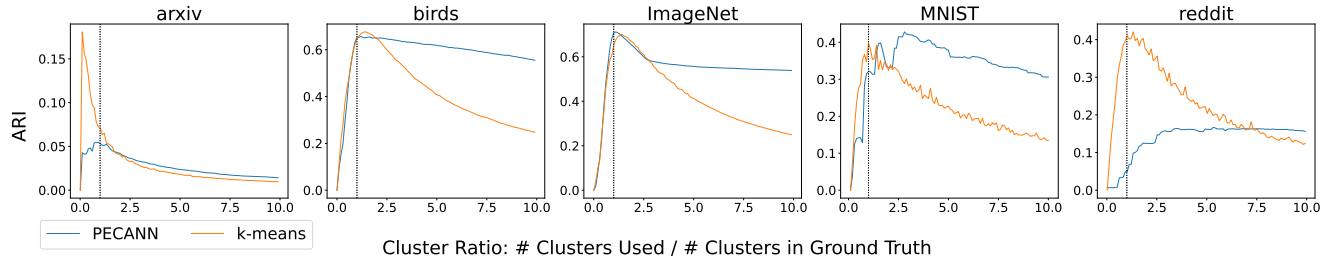

Figure 10: The ARI of PECANN using the VAMANA graph index vs. that of k-MEANS, when clustering with different numbers of clusters than the ground truth. y-axis is the ARI with respect to the ground truth. x-axis is the ratio between the number of clusters used and the number of clusters in the ground truth. The vertical dotted line is $x = 1$, where the correct number of clusters is used.

on `birds`, `ImageNet`, and `MNIST`, while k-MEANS is better on `arxiv` and `reddit`.

Moreover, as mentioned in Section 1, DPC variants can produce a hierarchy of clusters, which contains more information than k-means. Each run of k-means produces only a single cluster. Thus, to perform the experiment in Figure 10, in PECANN we can just redo the postprocessing step (Lines 7–14 of Algorithm 2), whereas for k-means we must rerun the algorithm from scratch for each choice of the number of clusters. For example, it takes about 4 hours to generate Figure 10 for `arxiv` and and about 90 hours for `imagenet`, whereas all datasets with PECANN take less than a few minutes.

In Figure 11, we show a Pareto frontier of the completeness and homogeneity scores (with respect to the ground truth) of PECANN and k-MEANS on different datasets. We generate this Pareto frontier using the same experiment setup as just described, except that now we record homogeneity and completeness instead of ARI as we vary the number of clusters given to each method. Thus, points along the Pareto frontier in Figure 11 are optimal tradeoffs between homogeneity and completeness as we vary the cluster granularity. We see that PECANN strictly dominates k-MEANS on `birds` and `MNIST`, and k-MEANS is better on `arxiv` and `reddit`. On `ImageNet`, PECANN achieves higher completeness and k-MEANS achieves higher homogeneity.

7 Related Work

In this section, we give an overview of the different DPC variations that have been proposed since the original DPC algorithm [80], particularly the ones that are based on k-nearest neighbors. We also briefly introduce other density-based clustering algorithms. Finally, we discuss recent advances on graph-based ANNS.

Variants of DPC. The original DPC algorithm [80] uses a range search to compute the density of a point x, where the density is defined as the number of points in a ball of fixed radius centered at x. In contrast, while PECANN supports any density metric, our

Algorithm	Dataset	Time (s)	Maximum ARI
PECANN	arxiv	11.65	0.07
FASTDP (SCALED)	arxiv	142.63	0.06
BRUTEFORCE	arxiv	9953.15	0.07
KMEANS	arxiv	2.41	0.07
DBSCAN	arxiv	986.58	0.03
PECANN	birds	0.86	0.65
FASTDP (SCALED)	birds	2.15	0.63
BRUTEFORCE	birds	66.04	0.66
KMEANS	birds	28.66	0.65
DBSCAN	birds	15.34	0.30
PECANN	ImageNet	101.58	0.71
FASTDP (SCALED)	ImageNet	127.60	0.71
BRUTEFORCE	ImageNet	31979.98	0.71
KMEANS	ImageNet	188.17	0.65
DBSCAN	ImageNet	3254.29	0.01
PECANN	MNIST	0.87	0.37
FASTDP (SCALED)	MNIST	0.66	0.37
BRUTEFORCE	MNIST	32.80	0.34
KMEANS	MNIST	0.22	0.40
DBSCAN	MNIST	9.21	0.18
PECANN	reddit	14.90	0.12
FASTDP (SCALED)	reddit	127.03	0.14
BRUTEFORCE	reddit	2888.20	0.10
KMEANS	reddit	5.36	0.42
DBSCAN	reddit	339.92	0.03

Table 5: The maximum ARI score with respect to the ground truth achieved by different clustering algorithms across different datasets, and their corresponding parallel running time. The sequential FASTDP runtime can be computed by multiplying FASTDP (SCALED) by 60.

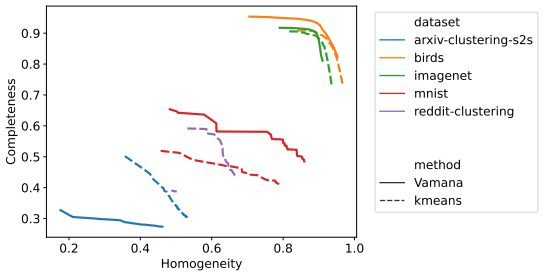

Figure 11: Pareto frontiers of completeness vs homogeneity of PECANN and k-MEANS on different datasets. Up and to the right is better.

paper focuses specifically on k-nearest neighbor-based DPC variants, which do not require a range search. These methods are less sensitive to noise and outliers [35] and are more computationally efficient to compute in high dimensions. Some of these methods (e.g., [35, 91, 103, 107]) also have a refinement step after obtaining the initial DPC clustering. For these methods, PECANN can be used to efficiently obtain the first DPC clustering before the refinement step.

Floros et al. [35] and Chen et al. [16] use the inverse of the distance to the k^{th} nearest neighbor as the density measure. Sieranoja and Fränti [87] propose FASTDP, which uses the inverse of the average distance to all k-nearest neighbors as the density measure, and finds the k-nearest neighbors by constructing an approximate k-nearest neighbor graph. Their motivation for not using the original DPC density function is that they are considering non-metric distance measures. Specifically, they use string similarity measures such as the Levenshtein distance and the Dice coefficient. In our experiments, we used the Euclidean distance measure for FASTDP to be consistent. d'Errico et al. [30] propose a variant of DPC for high-dimensional data. It combines DPC with a non-parametric density estimator called PAk, but their algorithm is sequential. Du et al. [28] propose a density function that depends on the shortest path distance in the k-nearest neighbor graph. Some works propose to normalize the density of each point by the density of its neighbors [38, 47, 90], as the normalization helps to reduce the influence of large density differences across clusters and is better for detecting clusters with different densities [47]. Yin et al. [108] use k-nearest neighbor searches to partition the data, which they show works well when the average density between clusters is very different. There are also works that propose to use density measures based on the mutual neighborhood [14, 60, 91], natural neighborhood [25, 112], order similarity [106], and the exponential of the sum of distances to neighbors [27, 60, 103, 107].

There are also algorithms that perform dimensionality reduction on the dataset before running DPC [14, 27], such as using principal components analysis.

Parallel, Approximate, and Dynamic DPC. Zhang et al. [109] propose an approximate DPC algorithm in the distributed (MapReduce) setting using locality-sensitive hashing. It partitions the data set into buckets, and searches within relevant buckets to find approximate dependent points. It resorts to scanning the whole dataset when the approximate dependent point does not seem accurate. Amagata and Hara [5] propose a partially parallel exact DPC algorithm and two parallel grid-based approximate DPC algorithms. They show that their algorithms are faster than previous solutions, including LSH-DDP [109], CFSFDP-A [8], FastDPeak [16], and DPCG [104]. They also propose parallel static and dynamic DPC algorithms for data in Euclidean space [4, 6]. Amagata and Hara [6] show that their dynamic algorithm outperforms previous dynamic DPC algorithms [41, 93]. Huang et al. [48] propose a fully parallel exact DPC algorithm based on priority kd-trees and show their algorithm outperforms previous tree-index approaches [5, 79]. Lu et al. [66] propose speeding up DPC using space-filling curves. Unlike PECANN, these algorithms [5, 6, 8, 48] are only efficient on low-dimensional datasets and must be used with Euclidean distance.

Amagata [4] proposes an approximate dynamic DPC algorithm for metric data, but their algorithm is sequential, and only tested on datasets with up to 115 dimensions. In comparison, our framework is parallel and we experimented on datasets in up to 1024 dimensions. There are also dynamic algorithms for k-nearest neighbor-based DPC variants [26, 84].

Density-based Clustering Algorithms. DPC falls under the broad category of density-based clustering algorithms, which have the advantage of being able to detect clusters of arbitrary shapes. Some density-based clustering algorithms define the density of a point based on the number of points in its vicinity [3, 7, 17, 34, 35, 53, 80]. Others use a grid-based definition, which first quantizes the space into cells and then does clustering on the cells [45, 46, 85, 97]. Still others use a probabilistic density function [58, 89, 97]. One popular density-based clustering algorithm is DBSCAN [34], which has many derivatives as well [7, 12, 13, 18, 33, 42, 92]. However,

the original authors of DBSCAN state that it is difficult to use for high-dimensional data [83].

Graph-based Approximate Nearest Neighbor Search (ANNS). Finally, we give an overview of graph-based ANNS methods, which are empirically effective [68, 96, 100]. Existinggraph-based indices include Hierarchical Navigable Small World Graph (HNSW) [67], DiskANN (also called Vamana) [55], HCNNG [73], PyNNDescent [70], τ-MNG [76], and many others (e.g., [15, 20, 111]). We refer the readers to Manohar et al. [68] and Wang et al. [96] for comprehensive overviews of these methods and their comparisons with non-graph-based methods, such as locality-sensitive hashing, inverted indices, and tree-based indices.

The dependent point search in DPC can also be viewed as a filtered search, where the points' labels are their density, and we filter for points with densities larger than the query point's density. Various graph-based similarity search algorithms have been adapted recently to support filtering [40, 44, 95, 110]. Gollapudi et al. [40] propose the Filtered DiskANN algorithm, which supports filtered ANNS queries, where nearest neighbors returned must match the query's labels. Gupta et al. [44] developed the CAPS index for filtered ANNS via space partitions, which supports conjunctive constraints while DiskANN does not. Both DiskANN [88] and CAPS can be made dynamic. However, these solutions use categorical labels, and a point can have multiple labels. Using this approach for dependent point finding requires quadratic memory just to specify the labels (the i^{th} least dense point would need $i-1$ labels, which are the $i-1$ smaller density values than its density), which is prohibitive. Indeed, we tried running the Filtered DiskANN code on our datasets but it ran out of space on our machine.

There are also works that explore the theoretical aspects of graph-based ANNS [50, 59, 74, 77, 86]. It is known that if we want to be able to find the exact nearest neighbor for any possible query via a greedy search, then the graph has to contain the Delaunay graph as a subgraph. Unfortunately, Delaunay graphs have high degrees in high dimensions and cannot be constructed efficiently [74, 77]. Laarhoven [59] provides bounds for nearest neighbor search on datasets uniformly distributed on a d-dimensional sphere with $d \gg \log n$ and provides time–space trade-offs for ANNS. Prokhorenkova and Shekhovtsov [77] extend Laarhoven [59]'s work and analyze the performance of graph-based ANNS algorithms in the low-dimensional ($d \ll \log n$) regime. Peng et al. [76] propose a new graph index and prove that if the distance between a query and its nearest neighbor is less than a constant, the search on their graph is guaranteed to find the exact nearest neighbor and the time complexity of the search is small. Indyk and Xu [50] study the worst-case performance of graph-based ANNS algorithms, including DiskANN, HNSW, and NSG. They show non-trivial bounds on accuracy and query time for a "slow preprocessing" version of DiskANN, and provide examples of poor worst-case behavior for the regular version of DiskANN, HNSW, and NSG.

There has also been work that uses approximate nearest neighbor oracles for clustering problems other than DPC [94].

8 Conclusion

We have presented the PECANN framework for density peaks clustering variants in high dimensions. We adapt graph-based approximate nearest neighbor search techniques to support (filtered) proximity searches in density peaks clustering variants. PECANN is highly parallel and scales to large datasets. We show several example clustering algorithms that can be implemented using PECANN, and evaluate them on large high-dimensional datasets. We show significant improvements in running time and clustering quality over other clustering algorithms.

References

[1] Mohamed Abbas, Adel El-Zoghabi, and Amin Shoukry. 2021. DenMune: Density peak based clustering using mutual nearest neighbors. *Pattern Recognition* 109 (2021), 107589.

[2] Charu C. Aggarwal and Chandan K. Reddy. 2013. *Data Clustering: Algorithms and Applications* (1st ed.). Chapman & Hall/CRC.

[3] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. 1998. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. 94–105.

[4] Daichi Amagata. 2022. Scalable and Accurate Density-Peaks Clustering on Fully Dynamic Data. In *IEEE International Conference on Big Data*. 445–454.

[5] Daichi Amagata and Takahiro Hara. 2021. Fast Density-Peaks Clustering: Multicore-Based Parallelization Approach. In *Proceedings of the International Conference on Management of Data*. 49–61.

[6] Daichi Amagata and Takahiro Hara. 2023. Efficient Density-peaks Clustering Algorithms on Static and Dynamic Data in Euclidean Space. *ACM Transactions on Knowledge Discovery from Data* 18, 1 (2023), 1–27.

[7] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 49–60.

[8] Liang Bai, Xueqi Cheng, Jiye Liang, Huawei Shen, and Yike Guo. 2017. Fast density clustering strategies based on the k-means algorithm. *Pattern Recognition* 71 (2017), 375–386.

[9] Pavel Berkhin. 2006. A survey of clustering data mining techniques. In *Grouping Multidimensional Data*. Springer, 25–71.

[10] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 507–509.

[11] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.

[12] Bhogeswar Borah and Dhruba K. Bhattacharyya. 2004. An improved sampling-based DBSCAN for large spatial databases. In *International Conference on Intelligent Sensing and Information Processing*. 92–96.

[13] Ricardo Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. 2015. Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection. *TKDD*, Article 5 (2015), 5:1–5:51 pages.

[14] Liangchen Chen, Shu Gao, and Baoxu Liu. 2022. An improved density peaks clustering algorithm based on grid screening and mutual neighborhood degree for network anomaly detection. *Scientific Reports* 12, 1 (2022), 1409.

[15] Patrick Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. 2023. FINGER: Fast Inference for Graph-based Approximate Nearest Neighbor Search. In *Proceedings of the ACM Web Conference 2023*. 3225–3235.

[16] Yewang Chen, Xiaoliang Hu, Wentao Fan, Lianlian Shen, Zheng Zhang, Xin Liu, Jixiang Du, Haibo Li, Yi Chen, and Hailin Li. 2020. Fast density peak clustering for large scale data based on kNN. *Knowledge-Based Systems* 187 (2020), 104824.

[17] Yewang Chen, Xiaoliang Hu, Wentao Fan, Lianlian Shen, Zheng Zhang, Xin Liu, Jixiang Du, Haibo Li, Yi Chen, and Hailin Li. 2020. Fast density peak clustering for large scale data based on kNN. *Knowledge-Based Systems* 187 (07 2020).

[18] Youguang Chen, William Ruys, and George Biros. 2020. KNN-DBSCAN: a DBSCAN in high dimensions. *arXiv preprint arXiv:2009.04552* (2020).

[19] David Coe, Mathew Barlow, Laurie Agel, Frank Colby, Christopher Skinner, and Jian-Hua Qian. 2021. Clustering Analysis of Autumn Weather Regimes in the Northeast United States. *Journal of Climate* 34, 18 (2021), 7587 – 7605.

[20] Benjamin Coleman, Santiago Segarra, Alexander J Smola, and Anshumali Shrivastava. 2022. Graph Reordering for Cache-Efficient Near Neighbor Search. *Advances in Neural Information Processing Systems* 35 (2022), 38488–38500.

[21] G.B. Coleman and H.C. Andrews. 1979. Image segmentation by clustering. *Proc. IEEE* 67, 5 (1979), 773–785.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms (4. ed.).* MIT Press.

[23] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition*. 248–255.

[24] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[25] Shifei Ding, Wei Du, Xiao Xu, Tianhao Shi, Yanru Wang, and Chao Li. 2023. An improved density peaks clustering algorithm based on natural neighbor with a merging strategy. *Information Sciences* 624 (2023), 252–276.

[26] Hui Du, Qiaofeng Zhai, Zhihe Wang, Yongbiao Li, and Manjie Zhang. 2022. A Dynamic Density Peak Clustering Algorithm Based on K-Nearest Neighbor. *Security and Communication Networks* 2022 (2022).

[27] Mingjing Du, Shifei Ding, and Hongjie Jia. 2016. Study on density peaks clustering based on k-nearest neighbors and principal component analysis. *Knowledge-Based Systems* 99 (2016), 135–145.

[28] Mingjing Du, Shifei Ding, Xiao Xu, and Yu Xue. 2018. Density peaks clustering using geodesic distances. *International Journal of Machine Learning and Cybernetics* 9 (2018), 1335–1349.

[29] Mingjing Du, Shifei Ding, and Yu Xue. 2018. A robust density peaks clustering algorithm using fuzzy neighborhood. *International Journal of Machine Learning and Cybernetics* 9 (2018), 1131–1140.

[30] Maria d'Errico, Elena Facco, Alessandro Laio, and Alex Rodriguez. 2021. Automatic topography of high-dimensional data sets by non-parametric density peak clustering. *Information Sciences* 560 (2021), 476–492.

[31] Yury Elkin and Vitaliy Kurlin. 2022. Counterexamples expose gaps in the proof of time complexity for cover trees introduced in 2006. In *Topological Data Analysis and Visualization (TopoInVis)*. 9–17.

[32] Yury Elkin and Vitaliy Kurlin. 2023. A new near-linear time algorithm for k-nearest neighbor search using a compressed cover tree. In *International Conference on Machine Learning (ICML)*. 9267–9311.

[33] Levent Ertöz, Michael Steinbach, and Vipin Kumar. 2003. Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data. In *Proceedings of the SIAM International Conference on Data Mining*. 47–58.

[34] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*. 226–231.

[35] Dimitris Floros, Tiancheng Liu, Nikos Pitsianis, and Xiaobai Sun. 2018. Sparse Dual of the Density Peaks Algorithm for Cluster Analysis of High-dimensional Data. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–14.

[36] Pasi Fränti and Sami Sieranoja. 2018. K-means properties on six clustering benchmark datasets. *Applied Intelligence* 48 (2018), 4743–4759.

[37] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. 1977. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Software* 3, 3 (09 1977), 209–226.

[38] Yangli-ao Geng, Qingyong Li, Rong Zheng, Fuzhen Zhuang, Ruisi He, and Naixue Xiong. 2018. RECOME: A new density-based clustering algorithm using relative KNN kernel density. *Information Sciences* 436 (2018), 13–30.

[39] Gerry. 2023. *Birds 525 Species - Image Classification*. https://www.kaggle.com/datasets/gpiosenka/100-bird-species

[40] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *Proceedings of the ACM Web Conference 2023*. 3406–3416.

[41] Shufeng Gong, Yanfeng Zhang, and Ge Yu. 2017. Clustering stream data by exploring the evolution of density mountain. *Proceedings of the VLDB Endowment* 11, 4 (2017), 393–405.

[42] Markus Götz, Christian Bodenstein, and Morris Riedel. 2015. HPDBSCAN: highly parallel DBSCAN. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. 1–10.

[43] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. 2022. Parallel cover trees and their applications. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 259–272.

[44] Gaurav Gupta, Jonah Yi, Benjamin Coleman, Chen Luo, Vihan Lakshman, and Anshumali Shrivastava. 2023. CAPS: A Practical Partition Index for Filtered Similarity Search. *arXiv preprint arXiv:2308.15014* (2023).

[45] B. Hanmanthu, R. Rajesh, and Priyanshul Niranjan. 2018. Parallel Optimal Grid-Clustering algorithm exploration on MapReduce Framework. *International Journal of Computer Applications* 180 (05 2018), 35–39.

[46] Alexander Hinneburg and Daniel A. Keim. 1998. An Efficient Approach to Clustering in Large Multimedia Databases with Noise. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*. 58–65.

[47] Jian Hou and Aihua Zhang. 2019. Enhancing density peak clustering via density normalization. *IEEE Transactions on Industrial Informatics* 16, 4 (2019), 2477–2485.

[48] Yihao Huang, Shangdi Yu, and Julian Shun. 2023. Faster Parallel Exact Density Peaks Clustering. In *Proceedings of the SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*. 49–62.

[49] Lawrence Hubert and Phipps Arabie. 1985. Comparing partitions. *Journal of Classification* 2, 1 (1985), 193–218.

[50] Piotr Indyk and Haike Xu. 2023. Worst-case Performance of Popular Approximate Nearest Neighbor Search Implementations: Guarantees and Limitations. In *NeurIPS*.

[51] Anil Kumar Jain, Musti Narasimha Murty, and Patrick J. Flynn. 1999. Data Clustering: A Review. 31, 3 (Sept. 1999), 264–323.

[52] J. Jaja. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

[53] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. 2004. DBDC: Density Based Distributed Clustering. In *International Conference on Extending Database Technology*, Vol. 2992. 88–105.

[54] Siddhartha V. Jayanti and Robert E. Tarjan. 2021. Concurrent disjoint set union. *Distributed Computing* 34, 6 (2021), 413–436.

[55] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems* 32 (2019).

[56] Dong Jiang, Wenke Zang, Rui Sun, Zehua Wang, and Xiyu Liu. 2020. Adaptive density peaks clustering based on K-nearest neighbor and Gini coefficient. *Ieee Access* 8 (2020), 113900–113917.

[57] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[58] H.-P. Kriegel and M. Pfeifle. 2005. Hierarchical density-based clustering of uncertain data. In *IEEE International Conference on Data Mining*.

[59] Thijs Laarhoven. 2018. Graph-Based Time-Space Trade-Offs for Approximate Near Neighbors. In *34th International Symposium on Computational Geometry*, Vol. 99. 57:1–57:14.

[60] Zejian Li and Yongchuan Tang. 2018. Comparative density peaks clustering. *Expert Systems with Applications* 95 (2018), 236–247.

[61] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281* (2023).

[62] Peng-Cheng Lin and Wan-Lei Zhao. 2019. A comparative study on hierarchical navigable small world graphs. *Computing Research Repository (CoRR) abs/1904.02077* (2019).

[63] Qiuzhen Lin, Songbai Liu, Ka-Chun Wong, Maoguo Gong, Carlos A. Coello Coello, Jianyong Chen, and Jun Zhang. 2019. A Clustering-Based Evolutionary Algorithm for Many-Objective Optimization Problems. *IEEE Transactions on Evolutionary Computation* 23, 3 (2019), 391–405.

[64] Rui Liu, Hong Wang, and Xiaomei Yu. 2018. Shared-nearest-neighbor-based clustering by fast search and find of density peaks. *information sciences* 450 (2018), 200–226.

[65] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. 2022. A ConvNet for the 2020s. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022).

[66] Jing Lu, Yuhai Zhao, Kian-Lee Tan, and Zhengkui Wang. 2020. Distributed density peaks clustering revisited. *IEEE Transactions on Knowledge and Data Engineering* 34, 8 (2020), 3714–3726.

[67] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836.

[68] Magdalen Dobson Manohar, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 270–285.

[69] Antonio Marco and Roberto Navigli. 2013. Clustering and Diversifying Web Search Results with Graph-Based Word Sense Induction. *Computational Linguistics* 39 (09 2013), 709–754.

[70] Leland McInnes. 2020. *PyNNDescent for Fast Approximate Nearest Neighbors*. https://pynndescent.readthedocs.io/en/latest/

[71] Nina Mishra, Robert Schreiber, Isabelle Stanton, and Robert Endre Tarjan. 2007. Clustering Social Networks. In *International Workshop on Algorithms and Models for the Web-Graph*, Vol. 4863. 56–67.

[72] Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. 2023. MTEB: Massive Text Embedding Benchmark. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*. 2014–2037.

[73] Javier Vargas Munoz, Marcos A Gonçalves, Zanoni Dias, and Ricardo da S Torres. 2019. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition* 96 (2019), 106970.

[74] Gonzalo Navarro. 2002. Searching in metric spaces by spatial approximation. *The VLDB Journal* 11 (2002), 28–46.

[75] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[76] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.
[77] Liudmila Prokhorenkova and Aleksandr Shekhovtsov. 2020. Graph-based nearest neighbor search: From practice to theory. In *International Conference on Machine Learning*. 7803–7813.
[78] Nadia Rahmah and Imas Sukaesih Sitanggang. 2016. Determination of optimal epsilon (eps) value on dbscan algorithm to clustering data on peatland hotspots in sumatra. In *IOP conference series: earth and environmental science*, Vol. 31. IoP Publishing, 012012.
[79] Zafaryab Rasool, Rui Zhou, Lu Chen, Chengfei Liu, and Jiajie Xu. 2020. Index-based solutions for efficient density peak clustering. *IEEE Transactions on Knowledge and Data Engineering* (2020).
[80] Alex Rodriguez and Alessandro Laio. 2014. Clustering by fast search and find of density peaks. *Science* 344, 6191 (2014), 1492–1496.
[81] Andrew Rosenberg and Julia Hirschberg. 2007. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. 410–420.
[82] 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 (1998), 169–194.
[83] 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 Transactions on Database Systems (TODS)* 42, 3 (2017), 1–21.
[84] Seyed Amjad Seyedi, Abdulrahman Lotfi, Parham Moradi, and Nooruldeen Nasih Qader. 2019. Dynamic graph-based label propagation for density peaks clustering. *Expert Systems with Applications* 115 (2019), 314–328.
[85] Gholamhosein Sheikholeslami, Surojit Chatterjee, and Aidong Zhang. 2000. WaveCluster: A Wavelet-Based Clustering Approach for Spatial Data in Very Large Databases. *The VLDB Journal* 8, 3–4 (02 2000), 289–304.
[86] Anshumali Shrivastava, Zhao Song, and Zhaozhuo Xu. 2023. A Theoretical Analysis Of Nearest Neighbor Search On Approximate Near Neighbor Graph. *arXiv preprint arXiv:2303.06210* (2023).
[87] Sami Sieranoja and Pasi Fränti. 2019. Fast and general density peaks clustering. *Pattern Recognition Letters* 128 (2019), 551–558.
[88] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *arXiv preprint arXiv:2105.09613* (2021).
[89] Abir Smiti and Zied Eloudi. 2013. Wave DBSCAN: Improving DBSCAN clustering method using fuzzy set theory. In *International Conference on Human System Interactions (HSI)*. 380–385.
[90] Zhi-gang Su and Thierry Denoeux. 2018. BPEC: Belief-peaks evidential clustering. *IEEE Transactions on Fuzzy Systems* 27, 1 (2018), 111–123.
[91] Lin Sun, Xiaoying Qin, Weiping Ding, Jiucheng Xu, and Shiguang Zhang. 2021. Density peaks clustering based on k-nearest neighbors and self-recommendation. *International Journal of Machine Learning and Cybernetics* 12 (2021), 1913–1938.
[92] Apinya Tepwankul and Songrit Maneewongwattana. 2010. U-DBSCAN: A density-based clustering algorithm for uncertain objects. In *IEEE International Conference on Data Engineering Workshops*. 136–143.
[93] Liudmila Ulanova, Nurjahan Begum, Mohammad Shokoohi-Yekta, and Eamonn Keogh. 2016. Clustering in the face of fast changing streams. In *Proceedings of the SIAM International Conference on Data Mining*. 1–9.
[94] Enayat Ullah, Harry Lang, Raman Arora, and Vladimir Braverman. 2022. Clustering using Approximate Nearest Neighbour Oracles. *Transactions on Machine Learning Research* (2022).
[95] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2022. Navigable proximity graph-driven native hybrid queries with structured and unstructured constraints. *arXiv preprint arXiv:2203.13601* (2022).
[96] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 14, 11 (jul 2021), 1964–1978.
[97] Wei Wang, Jiong Yang, and Richard R. Muntz. 1997. STING: A Statistical Information Grid Approach to Spatial Data Mining. In *Proceedings of the International Conference on Very Large Data Bases*. 186–195.
[98] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-Efficient and Practical Parallel DBSCAN. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2555–2571.
[99] Yiqiu Wang, Rahul Yesantharao, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: A Library for Parallel Computational Geometry. In *Proceedings of the European Symposium on Algorithms (ESA)*. 88:1–88:19.
[100] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Graph-and Tree-based Indexes for High-dimensional Vector Similarity Search: Analyses, Comparisons, and Future Directions. *Data Engineering* (2023), 3–21.
[101] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, Vol. 98. 194–205.
[102] Renzhi Wu, Nilaksh Das, Sanya Chaba, Sakshi Gandhi, Duen Horng Chau, and Xu Chu. 2022. A Cluster-then-label Approach for Few-shot Learning with Application to Automatic Image Data Labeling. *ACM Journal of Data and Information Quality (JDIQ)* 14, 3 (2022), 1–23.
[103] Juanying Xie, Hongchao Gao, Weixin Xie, Xiaohui Liu, and Philip W Grant. 2016. Robust clustering by detecting density peaks and assigning points based on fuzzy weighted K-nearest neighbors. *Information Sciences* 354 (2016), 19–40.
[104] Xiao Xu, Shifei Ding, Mingjing Du, and Yu Xue. 2018. DPCG: an efficient density peaks clustering algorithm based on grid. *International Journal of Machine Learning and Cybernetics* 9, 5 (2018), 743–754.
[105] Miin-Shen Yang, Yu-Jen Hu, Karen Chia-Ren Lin, and Charles Chia-Lee Lin. 2002. Segmentation techniques for tissue differentiation in MRI of Ophthalmology using fuzzy clustering algorithms. *Magnetic Resonance Imaging* 20, 2 (2002), 173–179.
[106] Xiaofei Yang, Zhiling Cai, Ruijia Li, and William Zhu. 2021. GDPC: Generalized density peaks clustering algorithm based on order similarity. *International Journal of Machine Learning and Cybernetics* 12 (2021), 719–731.
[107] Liu Yaohui, Ma Zhengming, and Yu Fang. 2017. Adaptive density peak clustering based on K-nearest neighbors with aggregating strategy. *Knowledge-Based Systems* 133 (2017), 208–220.
[108] Lifeng Yin, Yingfeng Wang, Huayue Chen, and Wu Deng. 2022. An Improved Density Peak Clustering Algorithm for Multi-Density Data. *Sensors* 22, 22 (2022), 8814.
[109] Yanfeng Zhang, Shimin Chen, and Ge Yu. 2016. Efficient Distributed Density Peaks for Clustering Large Data Sets in MapReduce. *IEEE Transactions on Knowledge and Data Engineering* 28, 12 (2016), 3218–3230.
[110] Weijie Zhao, Shulong Tan, and Ping Li. 2022. Constrained Approximate Similarity Search on Proximity Graph. *arXiv preprint arXiv:2210.14958* (2022).
[111] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards Efficient Index Construction and Approximate Nearest Neighbor Search in High-Dimensional Spaces. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1979–1991.
[112] Qingsheng Zhu, Ji Feng, and Jinlong Huang. 2016. Natural neighbor: A self-adaptive neighborhood method without parameter K. *Pattern Recognition Letters* 80 (2016), 30–36.

9 Additional Functions in PECANN

9.1 Density Functions

PECANN provides the following density functions in addition to the ones presented in Section 4.2.

`exp-sum`. The density of x_i is $\rho_i = \exp(-\frac{\sum_{j\in N_i} D(x_i,x_j)^2}{k})$, which is exponential in the negative of the average squared distance between x_i and its k-nearest neighbors [27]. Each density computation takes $O(k)$ work and $O(\log k)$ span by using a parallel sum.

`sum-exp`. The density of x_i is $\rho_i = \frac{\sum_{j\in N_i} \exp(-D(x_i,x_j)^2)}{k}$ [107]. Each density computation is $O(k)$ work and $O(\log k)$ span to compute the summation using a parallel sum. It can also be viewed as a variant of the kernel density of the original DPC algorithm [80].

`sum`. The density of x_i is $\rho_i = -\sum_{j\in N_i} D(x_i, x_j)$, which is the negative sum of distances to the k-nearest neighbors. Each density computation takes $O(k)$ work and $O(\log k)$ span to compute the summation using a parallel sum. We include this as a simple baseline.

9.2 Center Functions

We describe the additional center functions F_{center} that we implement in PECANN. Recall from Section 2 that $\delta_i = D(x_i, \lambda_i)$ is the dependent distance of x_i.

Local Center. For this method, a point is a center point if it has the highest density among its k-nearest neighbors. This can be implemented using a parallel filter with $O(nk)$ work and $O(\log n)$ span. This density finder is usually accompanied by further steps to merge and refine the initial DPC clusters [35, 91].

There are also methods that plot a decision graph for visualization and pick the cluster centers manually [28, 29]. Finally, some iterative methods have been proposed (e.g., [1, 64, 103]), but they have been infrequently used.

9.3 Noise Function

This noise function described in Section 4.2 is the only one that PECANN currently implements because it is the most commonly used, but alternative noise function definitions can be supported. For example, Abbas et al. [1] define noise points based on the number of neighborhoods a point belongs in, and Xie et al. [103] compute noise points by using a threshold on the dependent distance.

10 Approximation Analysis

10.1 Density Estimation

We analyze the density approximated by the `kth` density function assuming that we can find c-approximate k-nearest neighbors.

Definition 2 (c-approximate k-nearest neighbors)**.** *Let p_j be the true j^{th} nearest neighbor of query point q. Let $\mathcal{N}$ be the returned set of approximate k-nearest neighbors of q. Let $\hat{p}_j$ be the point in $\mathcal{N}$ that is j^{th} furthest from q,*

$\mathcal{N}$ is c-approximate $c \geq 1$ if (1) for all $j \leq k, D(q, p_j) \leq D(q, \hat{p}_j) \leq c \cdot D(q, p_j)$, and (2) $\{p' : D(p', q) \leq \frac{D(p_k, q)}{c}\} \subseteq \mathcal{N}$, i.e., the set of points within distance $\frac{D(p_k, q)}{c}$ to q is a subset of $\mathcal{N}$.

The first condition guarantees that the furthest point in the approximate k-nearest neighbors are not too far from the true k-nearest neighbors. Note that for some density functions, the first condition can be weaker. For example, the `kth` density function only requires this condition when $j = k$. The second condition guarantees that the points that are sufficiently close to the query point are returned among the approximate k-nearest neighbors.

Definition 3 (Density Interval)**.** *The density interval of a point q is a range that gives the lower and upper bounds of the approximate density of q.*

Let r_q be the distance between q and its k-nearest neighbor. If we use an algorithm that guarantees c-approximate k-nearest neighbors, point q has density interval $[\frac{1}{cr_q}, \frac{1}{r_q}]$. Consider all points $[1, \ldots, n]$, ordered from having the highest true density to having the lowest true density. Consider the list of intervals $[[\frac{1}{cr_1}, \frac{1}{r_1}], \ldots, [\frac{1}{cr_n}, \frac{1}{r_n}]]$ in the same order (note that we are only using this order for analysis, and our algorithm does not need to compute this order).

Definition 4 (Conflict)**.** *A point q_i's density range $[a_i, b_i]$ has a conflict with another point q_j's density range $[a_j, b_j]$ if $[a_i, b_i]$ and $[a_j, b_j]$ has any overlap. For $i < j$, conflict happens when $a_i < b_j$.*

If the list of intervals does not conflict, our density estimation does not affect the correctness of subsequent steps, as only the relative ranking of densities is used when identifying dependent points. Moreover, if a contiguous chunk of points $[x_i, \ldots, x_j]$ have conflicts, these overlaps only affect the dependent point search for the points $[x_i, \ldots, x_j]$, and not points before i and after j in the ordering.

The following lemma guarantees that the density peaks of the exact algorithm that do not conflict with other points will remain density peaks.

Lemma 2. *Consider the threshold center function, which obtains the center points by selecting the points whose distance to their dependent point is greater than $\delta_{\min}$. If the density interval of a point does not conflict with any other interval and it is a true density peak, then it is still a density peak in PECANN given the same threshold $\delta_{\min}$.*

Proof. A point q is a density peak with threshold $\delta_{\min}$ if q's distance to its dependent point is greater than $\delta_{\min}$. Since there is no conflict with q's interval, the set of points with higher density than q is the same as in the exact algorithm. As a result, q can only find an approximate nearest neighbor that is either the same distance from or further away from its exact dependent point. Therefore, its distance to the approximate dependent point must be at least as large by Definition 2 and it stays a density peak. □

Note that there may be additional density peaks returned by the approximate algorithm, but the true density peaks in the exact algorithm are guaranteed to still be density peaks.

10.2 Dependent Point Estimation

Now we analyze the approximate dependent point found by Algorithm 3. The following lemma guarantees that the approximate dependent points returned by our algorithm are not too much further than the true dependent points. Let d_j be the distance to the true j^{th} nearest neighbor from query point q.

Lemma 4. *Suppose we find the approximate dependent point among the βk-approximate nearest neighbor, for $\beta \geq 1$. The approximate dependent point is at most $c^2 \frac{d_{\beta k}}{d_k}$ further from the exact dependent point given the same densities.*

Proof. When we find an approximate k-nearest neighbor, everything within $\frac{d_k}{c}$ has been found by Definition 2, so if we have not found a dependent point of query point q, it must be at least $\frac{d_k}{c}$-away from q. Suppose we found our approximate parent within the approximate βk-nearest neighbor which has a distance at most $cd_{\beta k}$ to p. Then, the approximate dependent point is at most $c^2 \frac{d_{\beta k}}{d_k}$ times further from q than the exact dependent point. □

In Algorithm 3, we use $\beta = 2$ for Lemma 4, since we double the number of nearest neighbors to find until we have found a dependent point.

11 Additional Experiments

`S2` and `Unbalanced` [36] are small 2-dimensional baseline datasets used in prior clustering papers. We summarize the datasets in Table 6 and describe the parameters we used for them in Table 7. For DBSCAN, we used Wang et al. [98]'s parallel C++ implementation, which is optimized for low-dimensional data sets, instead of scikit-learn. The other algorithms are the same as described in Section 6.1.

We show in Table 8 the runtime and ARI score with respect to the ground truth of all methods run using a single thread on the small datasets `S2` and `Unbalanced`. Compared to the density-based

Name	n	d	Description	# Clusters
S2	5,936	2	Standard benchmark	15
Unbalanced	6,500	2	Standard benchmark	8

Table 6: Small datasets used in our exeriments.

Dataset	L	L_d	R	k
S2, Unbalanced	12	4	16	6

Table 7: Default parameters used for the small datasets.

Dataset	Algorithm	Details	Time	ARI
S2	FASTDP	N/A	0.172	0.933
S2	k-MEANS	$nredo = 1$	0.005	0.860
S2	k-MEANS	$nredo = 50$	0.237	0.940
S2	PECANN	product center finder	0.486	0.925
S2	PECANN	threshold center finder	0.473	0.925
S2	BRUTEFORCE	threshold center finder	0.499	0.925
S2	DBSCAN	$\epsilon = 52000, min\_pts = 128$	0.006	0.877
Unbalanced	FASTDP	N/A	0.246	1.000
Unbalanced	k-MEANS	$nredo = 1$	0.003	0.691
Unbalanced	k-MEANS	$nredo = 50$	0.098	0.832
Unbalanced	PECANN	product center finder	0.737	0.843
Unbalanced	PECANN	threshold center finder	0.651	1.000
Unbalanced	BRUTEFORCE	threshold center finder	0.623	1.000
Unbalanced	DBSCAN	$\epsilon = 16000, min\_pts = 3$	0.005	0.999989

Table 8: Runtime and ARI score with respect to ground truth for all methods using a single thread on the small low-dimensional synthetic datasets S2 and Unbalanced. When using the threshold center finder, we set $\delta_{\min} = 102873$ for S2 and $\delta_{\min} = 30000$ for Unbalanced. We set $\rho_{\min} = 0$ for the noise function. For k-means, we used $niter = 20$.

methods, k-MEANS has a slightly higher ARI on S2, but significantly worse ARI on Unbalanced. This shows that the relative performance of k-MEANS and density-based methods depends on the dataset, which we also observed on large high-dimensional real-world datasets.

12 Details on DBSCAN Parameters

In this subsection, we present the parameters we used for the DBSCAN algorithm. Let $range(start, stop, step)$ represent the set of numbers from $start$ to $stop$ with increment $step$. We put all noise points into a single cluster when evaluating the ARI.

We follow the guidelines for choosing parameters as described in Section 6.6. We also explored other parameters by trial and error to try our best to obtain high ARI scores using DBSCAN. However, we find that on high-dimensional data, it is difficult for DBSCAN achieve high ARI. This is consistent with the observation of the original authors of DBSCAN [83].

For Unbalanced, we used $\epsilon \in range(5000, 20000, 1000)$ and $min\_pts \in range(1, 50, 2)$. We find that the highest ARI is achieved when when $\epsilon = 16000, min\_pts = 3$, which gives an almost perfect clustering. There are 9 clusters with 1 noise point.

For S2, we used $\epsilon \in range(40000, 70000, 2000)$ and $min\_pts \in range(100, 150, 2) \cup range(1, 50, 2)$. We find that the highest ARI is achieved when $\epsilon = 52000, min\_pts = 128$. There are 16 clusters with 275 noise points.

For MNIST, we used $\epsilon \in range(0.5, 9, 0.5)$ and $min\_pts \in range(1, 5, 1) \cup range(100, 1000, 200) \cup range(1500, 1700, 100) \cup range(5000, 9000, 1000)$. We find that the highest ARI is achieved when $\epsilon = 3, min\_pts = 1$. There are 60074 clusters and no noise points.

For birds, we used $\epsilon \in range(20, 40, 5) \cup range(6, 14, 1)$ and $min\_pts \in range(2000, 2200, 100) \cup range(1, 5, 1) \cup range(120, 270, 30)$. We find that the highest ARI is achieved when $\epsilon = 12, min\_pts = 1$. There are 44663 clusters and no noise points.

On arxiv, DBSCAN with $\epsilon \geq 0.64$ runs out of memory. We used $\epsilon \in range(0.32, 0.62, 0.02)$ and $min\_pts \in range(2000, 2200, 100) \cup range(1, 5, 1) \cup [10, 50, 100, 500, 1000, 5000]$. We find that the highest ARI is achieved when $\epsilon = 0.4, min\_pts = 1$. There are 447198 clusters and no noise points.

On reddit, we used $\epsilon \in range(0.4, 0.72, 0.02)$ and $min\_pts \in range(2000, 2200, 100) \cup range(1, 5, 1) \cup range(3000, 13000, 1000)$. We find that the highest ARI is achieved when $\epsilon = 0.46, min\_pts = 4$. There are 46132 clusters and 8089 noise points.

On ImageNet, DBSCAN with $\epsilon \geq 36$ runs out of memory. We used $\epsilon \in range(20, 34, 1)$ and $min\_pts \in range(2000, 2200, 100) \cup range(1, 5, 1) \cup range(700, 1300, 200)$. We find that the highest ARI is achieved when $\epsilon = 20, min\_pts = 700$. There are 393 clusters and 606651 noise points.