Theoretically Efficient Parallel Density-Peak Clustering

Yihao Huang
Phillips Academy
yhuang23@andover.edu

Shangdi Yu
MIT CSAIL
shangdiy@mit.edu

ABSTRACT

Clustering multidimensional points is a fundamental data mining task, with applications in many fields, such as astronomy, neuroscience, bioinformatics, and computer vision. The goal of clustering algorithms is to group similar objects together. Density-based clustering is a clustering approach that defines clusters as dense regions of points. It has the advantage of being able to detect clusters of arbitrary shapes, rendering it useful in many applications.

In this paper, we propose fast and theoretically efficient parallel algorithms for Density-Peaks Clustering (DPC), a method for density-based clustering. DPC is effective in detecting clusters of arbitrary shapes, and allows hyperparameter selection in a user-friendly fashion, unlike standard methods such as DBSCAN. However, existing exact DPC algorithms suffer from high computational cost both theoretically and in practice, which limits DPC's application to large-scale datasets. To remedy the performance issue, we propose three theoretically efficient exact DPC algorithms. Our most performant algorithm achieves lower work complexity (sequential runtime complexity) than the state-of-the-art DPC algorithm; it attains $O(\log(n))$ span complexity (parallel runtime complexity), a dramatic improvement from the $O(n^2)$ span complexity achieved by the previous best DPC algorithm. Our most performant DPC algorithm utilizes a novel data structure which we call a priority search kd-tree. We present the priority search kd-tree and provide complexity analysis for performing queries on this data structure.

We provide optimized implementations of our algorithms and evaluate their performances via extensive experiments. Running on a 30-core machine with two-way hyperthreading, we find that our best algorithm achieves a 8.3–4666.3x speedup over the previous best exact DPC algorithm. Compared to the state-of-the-art approximate DPC algorithm, our best algorithm achieves competitive results and is able to achieve a geometric mean speedup of 8.2x. Our DPC algorithms are scalable, attaining a 8.8–13.2x self-relative speedup.

KEYWORDS

parallel computing, clustering, density-peak, unsupervised learning

1 INTRODUCTION

Clustering multidimensional points is a fundamental task in data analysis and unsupervised machine learning. Algorithms that perform clustering have wide applications spanning many fields. They can be used to identify different types of tissues in medical imaging [66], analyze social networks [51], identify weather regimes in climatology [14], and analyze dwarf galaxies in astrophysics [8]. Clustering algorithms are also widely used as a data processing subroutine in other machine learning tasks [15, 40, 42, 64].

One popular paradigm of clustering algorithms is density-based clustering, which defines clusters as dense regions of points in the coordinate space. When compared with traditional clustering algorithms, density-based clustering has two main advantages. First, they are computationally more tractable. Second, they can discover clusters of arbitrary shapes while algorithms such as k-means clustering can only recover clusters with spherical shapes. For these reasons, density-based clustering has received a tremendous amount of attention, with a number of proposed algorithms [1, 3, 21, 30, 31, 33, 49, 52, 59].

In this paper, we focus on Density-Peaks Clustering (DPC), a density-based clustering algorithm proposed by Rodriguez and Laio [49], which has many advantages. A lot of density-based clustering algorithms, such as DBSCAN [21], are very sensitive towards the choice of a density-noise cutoff hyper-parameter (points with density lower than the cutoff are deemed as irrelevant noise) [21]. DPC, in comparison, has been shown to perform well consistently over different hyper-parameter choices [49]. It is also very easy to set the hyper-parameters of DPC because DPC can generate a decision graph [49] that visually aids the determination of the hyper-parameters. Due to its advantages, DPC has been applied in the analysis of pathogenesis of COVID-19 [71], cancer study [29], neuroscience study [45], market analysis [60], computer vision tasks [39], and natural language processing [58].

DPC, however, suffers from a relatively high runtime complexity of $O(n^2)$ and has low parallelism, which limits its application in performing cluster analysis on large datasets. We aim to improve the running time of DPC in this work. DPC has three main steps.

(1) Compute the density of each point X, which is defined as the number of points in X's neighborhood.
(2) For each point X, connect X to its dependent point, which is defined as the closest neighbor of X that has a higher density than X. The resulting graph is a tree.
(3) Cut all connections with a distance higher than a certain threshold value. Each resulting connected component is a separate cluster. This final step is equivalent to performing single linkage clustering [50] on the tree.

Let n be the number of points in a dataset, a naive implementation of DPC that computes all pair-wise point distances takes $O(n^2)$ time to compute the density of all points and another $O(n^2)$ time to connect each point to its dependent point [49]. Multiple works have attempted to optimize the computational cost of DPC [4, 26, 65, 69], but they cannot break the quadratic computational complexity barrier. Amagata and Hara [2] recently proposed an exact DPC algorithm that leverages a kd-tree to improve the efficiency of density computation and dependent point searching; it is currently the state-of-the-art DPC algorithm. Amagata and Hara [2]'s algorithm takes $O(n^{2-\frac{1}{d}} + n[\rho])$ time to compute the density of every point, where d is the dimensionality of the coordinate space and $[\rho]$ represents the average density of all points. Their algorithm takes $O(n^2)$ time to connect every point to its closest neighbor with higher density (Note that Amagata and Hara claimed a lower complexity for this task, but they did not provide a proof and we cannot find evidence for that claim) [2].They parallelize the density computation for points. However, they do not parallelize finding the dependent point for different points. As a result, their algorithm suffers from high sequential dependency and Step 2 constitutes the computational bottleneck of their algorithm [2].

In this paper, we develop fast parallel algorithms for DPC. As the sizes of modern day datasets grow [36], leveraging parallelism to speed up clustering becomes crucial. In particular, to take advantage of the exponentially increasing number of cores in a commercially available CPU, shared-memory multi-core parallelism becomes indispensable for high performance algorithms [54, 56]. As such, the primary focus of our work is on tackling the computational and parallelism bottleneck of Amagata and Hara [2]'s algorithm—the dependent point finding task. We present three optimized algorithms for solving the dependent point finding task. We use the classic *work-span* model to analyze the theoretical complexity of our parallel algorithm, where briefly, the *work* is the total number of operations performed by the algorithm, and the *span* (or the *depth*) is the length of the longest chain of sequential dependencies in the algorithm.

(1) ***parallel fenwick tree based algorithm*** The first algorithm stores points in multiple kd-trees nested inside a fenwick tree. The fenwick tree partitions points along increasing density values such that each kd-tree stores points within a particular range of density values. To query the dependent point of a point X, we consider the range of density values higher than X's density. This range is partitioned by the fenwick tree into $O(\log(n))$ sub-ranges that each correspond to a kd-tree. We perform queries on these $O(\log(n))$ kd-trees and aggregate the results. The algorithm is highly parallel since each dependent point query can be performed independently. The work of the algorithm is bounded by $O(n\log(n)^2)$ in expectation. The span is bounded by $O(\log(n)\log\log(n))$.
(2) ***iterative incomplete kd-tree based algorithm*** The second algorithm is obtained by optimizing Amagata and Hara [2]'s dependent point finding algorithm using a lazy strategy for the kd-tree. Compared to the $O(n^2)$ work complexity achieved by Amagata and Hara [2], this algorithm has an expected work complexity bounded by $O(n\log(n))$ and a span complexity of $O(n\log(n))$.
(3) ***parallel priority search kd-tree based algorithm*** The third algorithm utilizes a ***priority search kd-tree***, an optimization of a max kd-tree [19, 28]. Priority search kd-tree can be used to query the dependent point of a point X in $O(\log(n))$ time in expectation. Because each point can perform dependent point query independently, this algorithm is highly parallel. It has an expected work of $O(n\log(n))$ and a span of $O(\log(n)\log\log(n))$.

In addition to the three dependent point finding algorithms we present, we also introduce an optimization technique for density computation. Finally, we solve Step 3's single linkage clustering

problem using a parallel union-find data structure [34], which has $O(n\alpha(n))$ expected work and $O(\log(n))$ span with high probability, where α represents the inverse Ackermann function.

We implement our algorithms using the ParGeo library [62] and evaluate them on both synthetic and real-world datasets. We compare our runtime results to Amagata and Hara [2]'s state-of-the-art exact DPC algorithm. Experiments are performed on a 30-core machine with two-way hyper-threading. Our optimized density computation algorithm outperforms Amagata and Hara's density computation by up to 18586.3x. For dependent point finding, our parallel fenwick tree based algorithm achieves up to a 1551.7x speedup over Amagata and Hara's algorithm; our iterative incomplete kd-tree based approach attains up to a 675.9x speedup; our parallel priority search kd-tree based approach attains up to a 4666.3x speedup.

Our contributions are threefold.

(1) We propose an optimization of the max kd-tree, called a priority search kd-tree data structure. We prove that this data structure can perform queries of a point's closest neighbor with higher priority/density in $O(\log(n))$ time in expectation. We also show that it can perform queries of points inside an axis-parallel range with priority value higher than some threshold in $O(n^{1-\frac{1}{d}} + |Q|)$ time, where Q is the set of points satisfying the query constraint.
(2) We introduce three new algorithms for solving the dependent point finding task in a DPC algorithm, and introduce techniques for tackling the density computation and single linkage clustering tasks in a DPC algorithm.
(3) We provide fast implementations of our algorithms and perform extensive experimental evaluations of these algorithms. We show that our DPC algorithms vastly outperform the state-of-the-art.

Our source code is publicly available at https://github.com/michaelyhuang23/ParCluster.

2 RELATED WORK

2.1 Density Peaks Clustering

Density Peaks Clustering (DPC) was invented by Rodriguez and Laio [49] and has received a lot of attention. Many variants of DPC have been developed [13, 17, 35, 44, 58, 65, 68]. Due to DPC's high computational cost, there has also been a line of work focused on improving the computational efficiency of the standard DPC algorithm. Bai et al. [4] utilized k-means clustering as a preprocessing step of DPC to prune the number of points needed to be traversed to find a point's density and dependent point. Gong and Zhang [26] parallelized DPC in a distributed setting and employed Voronoi diagrams to improved its efficiency. Amagata and Hara [2] leveraged a kd-tree to improve the density computation and dependent point finding runtime. Some works also relaxed the definition of DPC and arrived at efficient algorithms for approximate DPC. Zhang et al. [69] proposed LSH-DDP, a parallel algorithm on distributed system that first hashes points into buckets, with spatially-close points being hashed into the same bucket. It then approximates the density and dependent point query of a point X by only considering points from the same bucket as X. Finally, it applies corrections to the approximations as deemed necessary. Amagata and Hara [2] also proposed a parallel approximate DPC that constructs a spatial grid on top of the points. Leveraging the grid structure, the algorithm shares density and dependent point computations between all points inside the same grid cell, thus reducing the computational cost. It should be noted, however, that none of the exact DPC algorithms achieve strong work complexity or span complexity guarantees, a gap that our work tries to bridge. Experimentally, our best algorithm outperforms all existing exact DPC algorithms and is competitive against the state-of-the-art approximate DPC algorithm in terms of practical efficiency.

2.2 Density-based Clustering Algorithms

DPC falls under the category of density-based clustering algorithms, which has been extensively studied. In this subsection, we give a brief overview of density-based clustering algorithms. Some density-based clustering algorithms define density of a point based on the number of points in its vicinity [1, 3, 21, 33, 49]. Others leverage a grid-based definition [30, 31, 52, 59]. Some algorithms define density based on a probabilistic density function [37, 53, 59]. The most famous and standard density-based clustering algorithm is DBSCAN [21], which has many derivatives [3, 7, 12, 20, 27, 55]. DBSCAN first computes each point's density to be the number of points in its neighborhood. It then classifies points into core points and noise points based on a density cutoff and perform single linkage clustering [50] on the core points with a manually chosen distance cutoff. Though DBSCAN is popular, it has some drawbacks. DBSCAN not only suffers from a high sensitivity towards the hyperparameter choices but also cannot effectively cluster data distributions where the regions between clusters have relatively high density—a problem that DPC can evade [2, 4, 69].

2.3 kd-tree and k-Nearest Neighbor Query

A key technique of our DPC algorithms is performing nearest neighbor queries on a data structure structurally similar to a standard kd-tree. kd-trees are binary space partitioning data structures proposed by Bentley [5] to store multi-dimensional points by organizing them into cells, which are partitions of space. The key aspect of a kd-tree is the space partitioning scheme (or the splitting scheme) used. kd-trees can answer two types of queries efficiently: finding points inside a range and finding the k-nearest neighbors of some chosen point. We call the first type ***range query*** and the second ***k-nearest neighbor query***. Bentley [5] showed that a kd-tree can perform range query with complexity $O(n^{1-\frac{1}{d}} + |Q|)$, where Q is the set of points satisfying the query condition. Friedman et al. [23] showed that a kd-tree that always splits the widest dimension attains an expected runtime of $O(k\log(n))$ for k-nearest neighbor query because it only visits $O(k)$ number of cells in expectation. Maneewongvatana and Mount [41] proved that a kd-tree that adopts a sliding mid-point space partitioning scheme only visits $O(k)$ cells in the worst case; however, their kd-tree does not have a bounded height and therefore does not have a $O(k\log(n))$ expected query complexity.

There have also been works that propose variants of kd-trees that specialize in other tasks. Wald et al. [57] proposed implicit kd-tree, which defines the partitioning of space using a recursive splitting-function and is applied in ray tracing. Robinson [48] proposed K-D-B-tree, which is used to organize large point sets stored

on secondary memory. Groß et al. [28] proposed min-max kd-tree, which is designed for storing points with an extra attribute value. Each node of the min-max kd-tree records the minimum and maximum attribute value amongst all points stored under the subtree of that node [28]. Our proposed priority search kd-tree is an optimized variant of a max kd-tree (it can also be perceived as a generalization of the priority search tree data structure [43] to higher dimensions).

3 PRELIMINARIES

In this section, we provide definitions for the notations used in this paper. Then, we introduce the work-span model used in this work to analyze the runtime complexity of our parallel algorithms. Finally, we provide background on the fenwick tree, kd-tree data structures and other relevant parallel primitives.

3.1 Notations

Let $M = \{X_1, X_2, \cdots, X_n\}$ represent a size n set of points we need to perform clustering on. Each point is in d dimensional coordinate space. We use X to denote a generic point in $\mathbb{R}^d$ and X_i to represent a point in our point set M. Let $D(X_i, X_j)$ denote the distance between point X_i and point X_j. For the complexity results of our work to hold, D can be a range of metric distance measurements, as long as they are subject to the constraints detailed in Friedman et al. [23]. For instance, D can be any p-norm [23].

DEFINITION 1. *Given a point $X_i \in M$ and a cutoff value d_{cut}, we define the* **density** *of X_i to be $\rho(X_i) = |\{X \mid X \in M \text{ and } D(X_i, X) \leq d_{cut}\}|$.*

The density of X_i is the number of points inside a hyperball centered at X_i with radius d_{cut}. Given a point X_i, we define its dependent point set M_i as the set of points with density value higher than $\rho(X_i)$. Mathematically, $M_i = \{X_j \mid X_j \in M \text{ and } \rho(X_j) > \rho(X_i)\}$. When $\rho(X_i) = \rho(X_j)$ for some points X_i and X_j, the tie can be broken arbitrarily [2].

DEFINITION 2. *For a point $X_i \in M$, the* **dependent point** *of X_i is a point $\lambda(X_i) \in M_i$ such that,*

$$D(X_i, \lambda(X_i)) \leq D(X_i, X_j) \; \forall \; X_j \in M_i$$

We let $\delta(X_i)$ represent the distance between X_i and its dependent point, which we call the ***dependent distance*** of X_i. If X_i is the point with highest density in M, then it does not have a well-defined dependent point. In that case, we let $\delta(X_i) = \infty$.

Now, we define ***noise points*** and ***cluster centers***. A point $X_i \in M$ is considered a noise point if $\rho(X_i) < \rho_{\text{min}}$ for some density cutoff ρ_{min}. X_i is considered a cluster center if $\delta(X_i) \geq \delta_{\text{min}}$ and it is not a noise point. Each cluster center corresponds to a separate cluster. Each point that is not a cluster center is labeled with the same cluster as its dependent point. d_{cut}, ρ_{min}, and δ_{min} are the three hyperparameters of DPC. They can be set manually using the visual aid of an intuitive decision graph that plots each point X_i's density value $\rho(X_i)$ against its dependent point distance $\delta(X_i)$.

3.2 Model of Computation

Before proceeding with the discussion of our parallel DPC algorithms, we first provide background on how we analyze the runtime complexity of a parallel algorithm. The model we adopt for our parallel runtime complexity analysis is the shared-memory work-span model. The ***work*** of an algorithm is the total number of operations executed by the algorithm, and the ***span*** is the length of the longest dependency path of the algorithm [16]. Given an algorithm's work T_1 and span T_∞, we can bound the running time of the algorithm on P processors T_P using *Brent's Theorem* [9],

$$T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty$$

In our analysis, we assume arbitrary forking, which means that forking n processes takes a span of $O(1)$.

3.3 Relevant Techniques

Our algorithms make heavy use of the fenwick tree [22] and kd-tree [5] data structures. We now provide a brief background on the data structures and define the notations with which we will be using these data structures.

fenwick tree.

Fenwick tree decomposes a range $[1, n]$ into n sub-ranges such that the i^{th} sub-range, represented by $B[i]$, corresponds to the range $[i - \text{LSB}(i) + 1, i]$. Here $\text{LSB}(i)$ represents the least significant bit of integer i. Note that $\sum_{i=0}^{n} |B[i]| = O(n \log(n))$ [22]. The key property of a fenwick tree is that each prefix range $[1, i]$ can be decomposed into $O(\log(n))$ disjoint sub-ranges; we represent the set of these sub-ranges by $S[i]$. In other words,

$$\bigcup_{j \in S[i]} B[j] = [1, i].$$

$S[i]$ can be built in an iterative process. Let $i_0 = i$, $i_1 = i_0 - \text{LSB}(i_0)$, $i_2 = i_1 - \text{LSB}(i_1)$, $\cdots$, then $S[i] = \{i_0, i_1, i_2, \cdots\}$. Given the decomposition, we can access a partition of the range $[1, i]$ in $O(\log(n))$ time using the indices stored in $S[i]$.

kd-tree.

kd-tree is a binary space partitioning data structure. Each node of the kd-tree corresponds to a hyper-rectangular region of space called ***cell*** that contains a set of d dimensional points. Each node of a kd-tree partitions its point set into two equally sized subsets along a hyperplane that is perpendicular to the longest side of that node's cell. All points are stored at the leaf cells of a kd-tree. A kd-tree supports range search operation (searching through all points inside a particular range) and k-nearest neighbor query. It can be constructed with $O(n \log(n))$ work and $O(\log(n) \log\log(n))$ span [67]. A kd-tree T can be dynamic, in which case we can insert a point X into T. Note that a dynamic kd-tree can be unbalanced and not satisfy complexity results of a normal kd-tree. We use BUILD-kD-TREE(M) to represent initializing a kd-tree from the set of points M. Similarly, let BUILD-DYN-kD-TREE(M) denote initializing a dynamic kd-tree from M.

range query with kd-tree.

A kd-tree can be used to efficiently traverse points within a certain range R. When traversing down the kd-tree, we only need to visit a node if its cell intersects with R. If not, it can be pruned from the search. A range search takes $O(n^{1-\frac{1}{d}} + \text{size}(R))$ work [1], where $\text{size}(R)$ denotes the number of points in R [38]. It takes $O(\log(n))$

[1] The work complexity that arises here utilizes a slightly different splitting rule [38].

span. If T represents a kd-tree, we use T.QUERY-RANGE(X, r) to denote a range search on T, in a spherical region with radius r centered at a generic point X. QUERY-RANGE returns the number of points inside the region.

k-nearest neighbor query with kd-tree.

A kd-tree can also be used to find the k-nearest neighbors of a generic point X (note that the variable k in "kd-tree" represents the dimensionality of the tree while the variable k in k-nearest neighbor represents the number of nearest neighbors of X). In the first step of the search, we traverse down the kd-tree to find the leaf that contains the point X. Then, in the backtracking process, we only search the neighboring sibling subtrees. Let X's distance to the current k^{th} nearest neighbor of X be represented by L^k, then we can prune the search of any subtree whose cell is farther than L^k away from X. Friedman et al. [23] proved that the expected runtime of a k-nearest neighbor search can be bounded by $O(k \log(n))$, or more roughly $O(\log(n))$. When applying kd-tree nearest neighbor searching to DPC, we only need to search for the first nearest neighbor. We use T.QUERY-NN(X) to represent performing a nearest neighbor search on T for the point X. QUERY-NN returns the closest neighbor of X.

other parallel primitives.

Besides the fenwick tree and kd-tree data structures we use. We also utilize the parallel primitives defined as follows.

ATOMIC-WRITE(a, b, COND) takes as input a variable a, a value b, and a function COND. COND(a, b) is a function that takes in a, b and outputs a boolean result. ATOMIC-WRITE atomically reads a, and if COND(a, b) = True, it then updates a's value to b. If the update is performed successfully, the function returns true, and otherwise, it returns false. We assume that this takes $O(1)$ work.

RADIX-SORT(A) takes as input a collection of elements of size n with an ordering key defined for each element. It sorts them in parallel according to the natural ordering of the elements' keys. The sort takes $O(n)$ work and $O(\log(n))$ span w.h.p. given that the range of the keys is bounded by $O(n)$ [46].

4 BASELINE PARALLEL DPC ALGORITHM AND MINOR OPTIMIZATIONS

4.1 Amagata and Hara [2]'s Algorithm

Using the preliminary background and notations provided in Section 3, we now provide a brief description of how Amagata and Hara [2]'s proposed DPC algorithm solves the density computation task, the dependent point finding task, and the single linkage clustering task.

(1) *density computation:* Amagata and Hara [2]'s algorithm constructs a kd-tree T from all points. It then computes the density for every point X_i in parallel by $\rho(X_i) \leftarrow T$.QUERY-RANGE(X_i, d_{cut}).

(2) *dependent point finding:* Amagata and Hara [2]'s algorithm uses a dynamic kd-tree T' to find each point's dependent point. It first sorts all points by descending order of density and then sequentially iterate through the sorted points. Point X_i's dependent point is found by $\lambda(X_i) \leftarrow T'$.QUERY-NN$(X_i)$. Then, X_i is inserted into dynamic kd-tree T'.

Algorithm 1 Single linkage clustering with parallel union-find

```
1: procedure SINGLE-LINKAGE-CLUSTER(M, λ, δ, δ_min)
2:     initialize F to be an empty parallel union-find data structure
3:     parfor all X_i in M do
4:         if δ(X_i) < δ_min then    ▷ check if X_i's dependent distance is < threshold
5:             F.UNIONIZE(X_i, λ(X_i))                ▷ unionize X_i with λ(X_i)
6:     return F.cluster-labels
```

(3) *single linkage clustering:* To solve single linkage clustering, Amagata and Hara [2]'s algorithm simply performs a depth first search starting from each cluster center.

The primary computational bottleneck both theoretically and experimentally is the dependent point finding step; this is true especially in low dimensional datasets. Its sequential for loop incurs a high span complexity. Moreover, the dynamic kd-tree point insertions could unbalance the kd-tree, which does not have trivial re-balancing strategies [5]. The unbalanced kd-tree's height is not bounded by $O(\log(n))$, causing its k-nearest neighbor search work complexity and span complexity to explode from an expected complexity of $O(\log(n))$ to $O(n)$. As a result, the overall dependent point finding routine has a work and span complexity of $O(n^2)$. The primary focus of our work is proposing faster algorithms to solve the dependent point finding task.

4.2 Optimizing Density Computation

Before introducing our algorithms for solving the dependent point finding task, we first discuss a simple optimization we use to speedup the density computation operation QUERY-RANGE(X_{center}, r). Let S denote the spherical region with radius r and centered at X_{center}, which is a generic point. In a standard QUERY-RANGE operation used by Amagata and Hara [2], we go down the kd-tree, visiting the points in all leaf cells that intersect with S. We note that since we only seek to count the number of points in S, we do not have to visit every point. If a cell corresponding to a subtree is contained inside S completely, then we can simply add the number of points inside that cell to the count and prune the subtree from the rest of the traversal. It is possible to check whether a hyper-rectangular region R in coordinate space is contained inside a sphere S by finding a point $X_{\text{far}} \in R$ that is farthest from the center of S and checking if X_{far} is enclosed in S. Let X_{min} represent the vertex of R with minimal coordinate values in all dimensions and let X_{max} represent the vertex with maximal coordinate values. Dimension i of the farthest point, X^i_{far}, can be found by:

if $X^i_{\text{center}} < (X^i_{\text{min}} + X^i_{\text{max}})/2$ **then**
  $X^i_{\text{far}} \leftarrow X^i_{\text{max}}$
else
  $X^i_{\text{far}} \leftarrow X^i_{\text{min}}$

4.3 Optimizing Single Linkage Clustering

The depth first search based single linkage clustering can also be improved. We opt to use a lock-free parallel union-find data structure [34] to solve single linkage clustering, thus cutting down the span complexity from Amagata and Hara [2]'s $O(n)$ to $O(\log(n))$. Our approach is inspired by Wang et al. [61]'s success in using a parallel union-find data structure to parallelize the DBSCAN algorithm. The procedure is simple and is shown in Algorithm 1.

Algorithm 2 Parallel dependent point finding with fenwick tree

```
1:  procedure FENWICK-QUERY(T, i, X_{i+1})
2:      λ' ← ∅
3:      build S[i]   ▷ build a list of indices whose corresponding sub-ranges span the range [1, i]
4:      parfor all j in S[i] do
5:          Y ← T[j].QUERY-NN(X_{i+1})
6:          ATOMIC-WRITE(λ', Y, dist(X_{i+1}, Y) < dist(X_{i+1}, λ'))
7:      return λ'
8:  procedure FENWICK-DEPENDENT-POINT(M, ρ)
9:      M̄ ← RADIX-SORT(M)   ▷ let M̄ be a one-based array of all points in descending order of their densities
10:     initialize T as a one-based array of length n
11:     parfor i = 1 to n do
12:         T[i] ← BUILD-k-TREE(range B[i] in M̄)   ▷ construct all n k-trees
13:     initialize λ as a one-based array of length n ▷ λ(X_i) denotes the i^th entry of λ
14:     parfor all X_i in M̄ do
15:         λ(X_i) ← FENWICK-QUERY(T, i − 1, X_i)   ▷ compute each point X_i's dependent point in parallel
16:         if λ(X_i) ≠ ∅ then
17:             δ(X_i) ← dist(X_i, λ(X_i))   ▷ compute dependent distance
18:     return λ
```

Analysis.

The initialization on Line 2 takes $O(n)$ work and $O(1)$ span. Jayanti and Tarjan [34] proved that performing m unionizations on a union-find data structure with n elements takes $O(m\left(\log(\frac{n}{m}+1)+\alpha(n)\right))$ work, where α denotes the inverse Ackermann function. In our case, $m = n$. Therefore, the overall work complexity is $O(n\alpha(n))$. The unionization operation on Line 5 takse span $O(\log(n))$. Thus, the overall span of the algorithm is $O(\log(n))$.

5 FENWICK TREE BASED PARALLEL DEPENDENT POINT FINDING

In this section, we introduce our first algorithm for solving the dependent point finding task: a parallel fenwick tree based algorithm. Our parallel algorithm reduces the worst-case span complexity from Amagata and Hara [2]'s $O(n^2)$ to $O(\log(n))$.

This algorithm can be summarized as follows. We first construct a one-based array $\overline{M}$ of points in M sorted by descending order of their density values. Recall that $n = |\overline{M}|$ is the length of the array. Then, we construct a fenwick tree decomposition of the range $[1, n]$. $B[i]$ corresponds to the range of points in $\overline{M}$ with indices $[i-\text{LSB}(i)+1, i]$. For each range $B[i]$, we construct a kd-tree $T[i]$ containing $B[i]$'s range of points. Recall that $S[i]$ represents a decomposition of the range $[1, i]$ into sub-ranges that are inside B. To perform dependent point query for the i^{th} point in array $\overline{M}$, we simply need to search through every kd-tree that corresponds to a sub-range in $S[i-1]$. These queries can be computed in parallel, thus achieving a low span complexity.

We provide the dependent point search algorithm's pseudocode in Algorithm 2 and will now dissect it in greater details. The main procedure is FENWICK-DEPENDENT-POINT(M, ρ), which takes as input a set of points M and the computed densities of the points ρ (ρ can be stored as an array or hash table). To find the dependent points λ for all points, we first construct a one-based array of points $\overline{M}$ sorted in descending order of their density values, as done on Line 9. We then initialize a one-based array T to store the n kd-trees in the algorithm. On Line 11–12, we construct the n kd-trees; the i^{th} kd-tree $T[i]$ is constructed from the range of points $B[i] = [i-\text{LSB}(i)+1, i]$ in one-based array $\overline{M}$. Finally, on Line 15, we perform FENWICK-QUERY for all points in parallel to find the dependent point for all points.

Now, we will explain procedure FENWICK-QUERY, which takes as input an array of kd-trees T, an index i, and the $(i+1)^{\text{th}}$ point in $\overline{M}$, X_{i+1}; FENWICK-QUERY performs nearest neighbor query for point X_{i+1} on all points $X_1, X_2, \cdots, X_i$. On Line 3, we construct a set $S[i]$ for the inputed index i; set $S[i]$ contains the indices of the fenwick tree sub-ranges that form a partition of $[1, i]$, as described in Section 3.3. Each of those sub-ranges correspond to a kd-tree; we perform nearest neighbor query QUERY-NN on all those kd-trees on Line 5. Let λ' signify the current dependent point of point X_{i+1}. On Line 6, the current λ' is replaced by a newly found nearest neighbor if the newly found nearest neighbor is closer to X_{i+1} than λ' is.

Analysis.

We first analyze the time complexity of subroutine FENWICK-QUERY. We show that it takes $O(\log(n)^2)$ expected work and $O(n)$ worst-case work. The construction of S on Line 3 takes $O(\log(n))$ work [22]. On the other hand, in expectation, each call of QUERY-NN on Line 5 takes $O(\log(n))$ work [23], which accumulates to $O(\log(n)^2)$ expected work over all iterations of the parallel for loop on Line 4. In the worst-case however, each kd-tree nearest neighbor query takes time linear to the number of points in the kd-tree [23]. Thus, Line 5 takes $O(|B[j]|)$ for the j^{th} kd-tree, T_j. Over all iterations of the parallel for loop, the worst-case work complexity is $O(\sum_{j\in S[i]} |B[j]|) = O(i) = O(n)$.

In terms of span, FENWICK-QUERY attains an expected and worst-case span of $O(\log(n))$. Again, the construction of $S[i]$ only takes $O(\log(n))$ span. The ATOMIC-WRITE operation on Line 6 also only incur an additional span of $O(\log(n))$ in the worst case. There can be at most $\log(n)$ atomic updates conflicting with each other because $|S[i]| \le \log(n)$. The nearest neighbor query on Line 5 takes an expected and worst-case span of $O(\log(n))$. The expected span of kd-tree's nearest neighbor query is bounded by its expected work complexity of $O(\log(n))$, which is proven by Friedman et al. [23]. The worst-case nearest neighbor query span complexity turns out also to be $O(\log(n))$ since each branch of the kd-tree can be searched in parallel. Details of the worst-case nearest neighbor query complexity is addressed in Section 7. Since all nearest neighbor queries are executed in parallel, $O(\log(n))$ is also the span complexity for the entire FENWICK-QUERY subroutine.

Now, we examine the main process FENWICK-DEPENDENT-POINT. We show its expected work complexity to be $O(n\log(n)^2)$ and its worst-case work complexity to be $O(n^2)$. Line 9 takes $O(n)$ work since the keys of the sort–the ρ values–are bounded in size by $O(n)$. On Line 12, constructing the i^{th} kd-tree takes time $O(|B_i|\log(|B_i|))$. Therefore, constructing all kd-trees takes $O(\sum_{i=1}^{n} |B_i|\log(|B_i|)) = O(n\log(n)^2)$ work. Finally, all FENWICK-QUERY operations performed in the parallel for loop on Line 14 takes $O(n\log(n)^2)$ work in expectation and $O(n^2)$ work in the worst case. Thus, the overall work complexity of FENWICK-DEPENDENT-POINT is $O(n\log(n)^2)$ in expectation and $O(n^2)$ in the worst case.

Next, we analyze the span bounds of FENWICK-DEPENDENT-POINT. The radix sort on Line 9 takes $O(\log(n))$ span w.h.p.[2] Each

[2]We say $O(f(n))$ *with high probability (w.h.p.)* to indicate $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \ge 1$, where n is the input size.

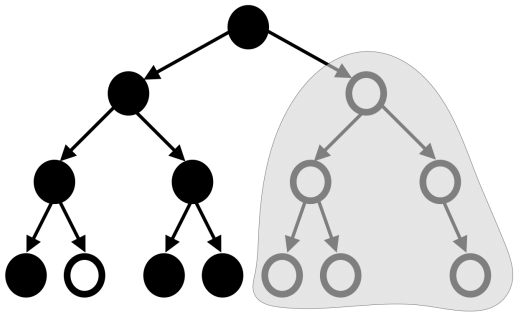


Figure 1: An example graph for an incomplete kd-tree. A node is unfilled if its subtree does not contain any active point; otherwise it is filled. During a k-nearest neighbor search, the entire grayed out subtree can be pruned because it contains no active point.

BUILD-kD-TREE operation on Line 12 has span $O(\log(n)\log\log(n))$. Finally, each call to subroutine FENWICK-QUERY on Line 15 takes $O(\log(n))$ span in the worst case. Thus, the overall span of FENWICK-DEPENDENT-POINT is $O(\log(n)\log\log(n))$ in the worst case.

Finally, we consider the space usage of our algorithm. The i^{th} kd-tree, T_i, takes space $O(|B_i|)$. Thus, the overall space usage is $O(\sum_{i=1}^{n}|B_i|) = O(n\log(n))$.

6 PRIORITY SEARCH kD-TREE BASED DEPENDENT POINT FINDING

In this section, we present two more algorithms for solving the dependent point finding task: an iterative incomplete kd-tree based algorithm and an algorithm based on a parallelization of the incomplete kd-tree data structure, which we call priority search kd-tree. Both of these algorithms attain better expected work complexity than the fenwick tree based algorithm described in Section 5 because they make use of only one kd-tree structure. The parallel priority search kd-tree based algorithm matches the fenwick tree based algorithm in terms of span complexity.

6.1 Iterative Dependent Point Finding with Incomplete kd-tree

First, we introduce an iterative incomplete kd-tree based algorithm for finding dependent points. We note that the computational bottleneck of Amagata and Hara [2]'s dependent point finding algorithm, as described in Section 4, originates from the use of a dynamic kd-tree that is not necessarily balanced, which makes querying slower than a balanced kd-tree. In this subsection, we propose to use a balanced incomplete kd-tree in place of a dynamic kd-tree. Instead of inserting points into the dynamic kd-tree, we utilize a lazy insertion strategy: the kd-tree is constructed with all points in M, but all points are marked as inactive initially. We use a variable isActive_i to track if the i^{th} subtree contains an active point. When we insert a point into the kd-tree, we simply activate the point and set $\text{isActive}_i \leftarrow \text{True}$ for each node i that is an ancestor of the leaf node at which the point is inserted. When traversing the kd-tree to query for k-nearest neighbors, we can prune a subtree i if its isActive_i value is False. An example of incomplete kd-tree is given in Figure 1.

Analysis.

Because the incomplete kd-tree is constructed in the same way as a normal kd-tree, its construction work is $O(n\log(n))$ and construction span is $O(\log(n)\log\log(n))$. An incomplete kd-tree can perform nearest neighbor query in $O(\log(n))$ expected work and $O(n)$ worst-case work, but always take only $O(\log(n))$ span. We reserve the proof of this fact for Section 7. As a result of the complexity bounds for an incomplete kd-tree's nearest neighbor query, the overall dependent point finding algorithm takes $O(n\log(n))$ expected work, $O(n^2)$ worst-case work, and $O(n\log(n))$ worst-case span.

6.2 Priority Search kd-tree

To parallelize the dependent point finding routine described in Section 6.1, we first introduce a parallel analogue of the incomplete kd-tree–a priority search kd-tree–and describe its general properties. A priority search kd-tree is intuitively a generalization of the 1 dimensional priority search tree data structure to higher dimensions. A priority search kd-tree is designed to store a set of points $M = \{X_1, X_2, \cdots, X_i, \cdots, X_n\}$ such that each point $X_i \in \mathbb{R}^d$ is associated with a priority value γ_i. Similar to a normal kd-tree, each node of the priority search kd-tree corresponds to a set of points and a partition of space—called a cell. We store at each node the point with the highest γ value amongst all points in that node's point set; this γ value is referred to as the γ value of the node. The rest of points are split evenly between the children of the node along a hyperplane perpendicular to the longest side of the cell of that node. An example of a priority search kd-tree is represented by Figure 2.

A priority search kd-tree is structurally similar to a max kd-tree [28], which records only the maximum priority value at each node. The actual point with that priority value is stored at a leaf in either the left or right subtree of that node.

Priority search kd-trees can be constructed similar to a normal kd-tree; the only extra step is finding the point with highest priority value at each node. Construction takes $O(n\log(n))$ work and $O(\log(n)\log\log(n))$ span. The data structure takes $O(n)$ memory like a normal kd-tree, because only $O(1)$ extra information is stored at each node compared to a normal kd-tree.

The main application of both priority search kd-tree and max kd-tree are answering ***priority range queries*** and ***priority k-nearest neighbor queries***. A priority search kd-tree is advantageous in that a meaningful priority range query complexity bound can be established for it but not for a max kd-tree.

6.2.1 Priority Range Query. A priority search kd-tree can be used to efficiently answer a priority range query, which can be defined as follows.

DEFINITION 3. *Given a query range $R_q \subseteq \mathbb{R}^d$, a priority threshold γ_q, and a point set M, a priority range search asks for the set of points $Q \subseteq M$ such that for each point $X_i \in Q$, $X_i \in R_q$ and $\gamma_i > \gamma_q$.*

A priority range query can be solved by a normal kd-tree by first querying the set of points inside R_q and then finding the subset of points that satisfies the priority constraint.

This can be optimized by a priority search kd-tree. When performing a priority range query on a priority search kd-tree T, we only visit nodes with a cell that intersect the query region R_q and a γ value higher than the cutoff value γ_q. Let T_q represent the subset

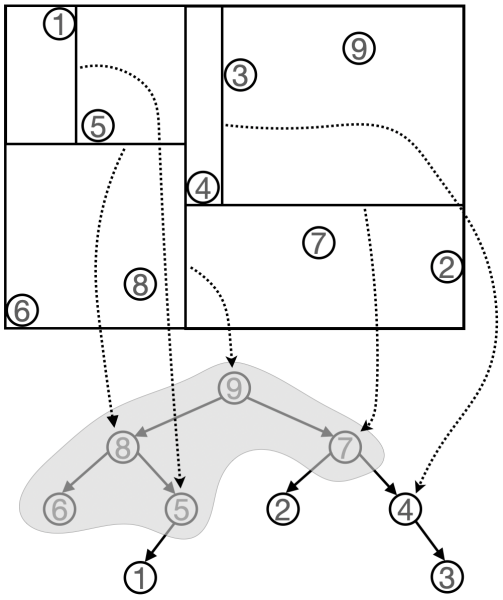


Figure 2: An example graph for a priority search kd-tree. Each point is labeled with its priority value γ, which is an integer from 1 to 9 in this example. Each node of the priority search kd-tree stores the point with the highest γ within the region of the cell of the node; the number inside the circle of the node represents the node's γ value. The dotted lines on the graph connects each node with the splitting hyperplane of that node. The grayed area represent a subtree T_q comprising all nodes with $\gamma > 4$. Because the γ values of a priority kd-tree satisfies the heap property, T_q is always an upper portion of the priority search kd-tree.

of nodes with γ value higher than γ_q. T_q is necessarily a connected subtree that forms an upper portion of T, as illustrated by Figure 2. Therefore, performing a priority range search on T is equivalent to performing a normal range search on the unbalanced kd-tree T_q.

Analysis.

If R_q is an axis-parallel hyper-rectangular region, then a meaningful complexity bound can be established for priority range query on a priority search kd-tree T. All cells visited during the query operation must be in T_q. There are two types of such cells.

(1) A cell that intersects R_q but is not completely inside R_q: It is well known that the number of such cells in T is bounded by $O(n^{1-\frac{1}{d}})$ [16].
(2) A cell that is completely inside R_q: Each such cell must contain a unique point X_i satisfying $X_i \in R_q$ and $\gamma_i > \gamma_q$. There are only a total of $|Q|$ such points. Therefore, the number of cells of this type is $\leq |Q|$.

In sum, the total number of cells visited by a priority range query operation is bounded by $O(n^{1-\frac{1}{n}} + |Q|)$.

Note that this proof cannot be applied to a max kd-tree because each cell in a max kd-tree is not uniquely associated with a point, as assumed in the proof in order to bound the number of Type 2 cells traversed.

6.2.2 *Priority K-Nearest Neighbor Query.*

Another application of a priority search kd-tree is to answer priority k-nearest neighbor queries, which is the usecase for priority search kd-tree in this paper. We define a priority k-nearest neighbor query by,

DEFINITION 4. *Given a generic query point $X_q \in \mathbb{R}^d$, a point set $M \subseteq \mathbb{R}^d$, and a distance measure D, find, among the points in M, k points $\{X_1, X_2, \cdots, X_i, \cdots, X_k\}$ such that X_i is the i^{th} closest point to X_q as measured by D and $\lambda_i > \lambda_q$ for all $i \in [1, k]$.*

This problem can be solved by querying on a priority search kd-tree following a similar procedure as a normal k-nearest neighbor query, with the exception that all subtrees with priority value $\leq \lambda_q$ may be pruned from the search. It can also be solved by querying on a max kd-tree (again, all subtrees with priority value $\leq \lambda_q$ are pruned from the search). The two data structures attain the same complexity for this task. This is because performing priority k-nearest neighbor query on a priority search kd-tree or a max kd-tree can be equated to performing a normal k-nearest neighbor query on an incomplete kd-tree, the complexity of which is bounded in Section 7. Consider a particular query, with a threshold priority value of λ_q. Let T denote a priority search kd-tree or a max kd-tree. Let $T_q \subseteq T$ represent the set of nodes with priority value $> \gamma_q$. Because of the structure of both priority search kd-tree and max kd-tree, T_q must be a connected subtree of T in both cases. A priority k-nearest neighbor search on T is thus equivalent to a normal k-nearest neighbor search on an incomplete kd-tree T with T_q forming its active portion. Thus, similar to the complexity result on an incomplete kd-tree, performing priority k-nearest neighbor query on a priority search kd-tree or max kd-tree takes only $O(\log(n))$ expected work, $O(n)$ worst-case work, and $O(\log(n))$ worst-case span.

It is important to note that a priority nearest neighbor query problem can be equated to the problem of finding dependent points if we set the priority value γ_i for a point X_i to be the density value $\rho(X_i)$.

6.3 Parallel Dependent Point Finding with Priority Search Kd-Tree

We now apply the priority search kd-tree data structure to solve the dependent point finding task. The algorithm introduced in this subsection achieves the same work complexity as the incomplete kd-tree based approach while maintaining a $O(\log(n) \log\log(n))$ worst-case span. The recipe for finding dependent points using a priority search kd-tree is given in Algorithm 3

Algorithm 3 Parallel dependent point finding with priority search kd-tree

```
1: procedure PRIORITY-SEARCH-KD-TREE-DEPENDENT-POINT(M, ρ)
2:     M̄ ← RADIX-SORT(M)  ▷ let M̄ be an array of all points in descending order of their densities
3:     T ← BUILD-PRIORITY-SEARCH-KD-TREE(M, ρ) ▷ construct a priority search kd-tree from the points and their density values
4:     initialize λ as an array of length n  ▷ λ(X_i) denotes the i^th entry of λ
5:     parfor all X_i in M do
6:         λ(X_i) ← T.QUERY-PRIORITY-NN(X_i)  ▷ compute each point X_i's dependent point in parallel
7:         if λ(X_i) ≠ ∅ then
8:             δ(X_i) ← dist(X_i, λ(X_i))  ▷ compute dependent distance
9:     return λ
```

Analysis.

Similar to the analysis for Algorithm 2, the RADIX-SORT operation on Line 2 takes $O(n)$ work and $O(\log(n))$ span w.h.p. The priority search kd-tree construction on Line 3 has a work complexity of $O(n\log(n))$ and span complexity of $O(\log(n)\log\log(n))$. Finally, each QUERY-PRIORITY-NN invocation on Line 6 takes $O(\log(n))$ work in expectation and $O(n)$ work in the worst case. The worst-case span complexity of QUERY-PRIORITY-NN is $O(\log(n))$. Therefore, the overall work complexity of Algorithm 3 is $O(n\log(n))$ in expectation and $O(n^2)$ in the worst case. The overall span complexity is $O(\log(n)\log\log(n))$.

7 K-NEAREST NEIGHBOR QUERY COMPLEXITY

In this section, we provide an analysis of the expected k-nearest neighbor query complexity on an incomplete kd-tree. We define an incomplete kd-tree to be a kd-tree constructed from a set of n points $M = \{X_1, X_2, \cdots, X_n\}$ in d-dimensional space such that X_i is only active if $X_i \in M_q$, where $M_q \subseteq M$.

Our analysis follows in a similar spirit as Friedman et al. [23]'s proof of the expected $O(k\log(n))$ complexity for k-nearest neighbor query on a normal kd-tree. We show that finding the k-nearest neighbors of some query point X_q on an incomplete kd-tree takes $O(k\log(n))$ time in expectation.

We separate the analysis based on the size of M_q.

Case 1: $|M_q| = O(k\log(n))$**.**

In this case, we can simply find the k-nearest neighbors of a point X_q by searching through all active points in the incomplete kd-tree without breaking the $O(k\log(n))$ complexity.

Case 2: $|M_q| = \Omega(k\log(n))$**.**

To analyze this case, we make some similar assumptions as Friedman et al. [23]. First, we assume that all $X_i \in M$ are sampled from $\mathbb{R}^d$ according to some probability density function μ. Similarly, all $X_i \in M_q$ are sampled from some probability density function μ_q. We use $\mu(X)$ and $\mu_q(X)$ to denote the probability density of functions μ and μ_q at a generic point X. Now, we assume that $|M|$ and $|M_q|$ are sufficiently large such that both μ and μ_q can be considered locally uniform. This means μ and μ_q can be taken to be constant within any compact hypercubical region R containing $\sim k$ points in expectation over all samplings of M and M_q according to μ and μ_q.

A result of the assumption that μ is locally uniform is that an incomplete kd-tree's ***compact cells***, or cells with $\sim k$ points, are near hypercubical in shape. This is because each node of the incomplete kd-tree always partitions the point set into two equally sized subsets along a hyperplane perpendicular to the longest side of that node's cell. The assumption that μ is locally uniform ensures that a cell containing $<< n$ points is partitioned into two cells with approximately equal volumes. Thus, under the assumption of local uniformity, the longest side of each cell is halved in each split of the kd-tree's node. As a results, compact cells should have a longest side that is not significantly longer than twice the length of its shortest side. We assume μ_q is also locally uniform. Because each compact cell is halved at each split of a node, we expect the set of active points to be also split approximately evenly given the local uniformity of μ_q. Thus, adjacent compact cells should contain approximately the same number of active points.

Let N_q represents a leaf node with cell R_q such that our query point X_q is contained within R_q. We define N_q^k to be the smallest subtree that contains N_q and contains $\geq k$ active points; let the cell of N_q^k be represented by R_q^k. The children cells of R_q^k are adjacent compact cells. Since one of them (the one containing X_q) has $< k$ active points, the other cell contains $\lessapprox k$ active points because of the local uniformity of μ_q. Thus, the expected number of active points in N_q^k, $[\text{size}(N_q^k)] \lessapprox 2k$, where the square bracket $[\cdot]$ indicates taking the expected value over all samplings of points. Since R_q^k contains $\lessapprox 2k$ active points in expectation, we can also consider μ_q and μ to be constant within R_q^k. We represent their values of constancy by $\mu_q(X_q)$ and $\mu(X_q)$.

We define $V(R_q^k)$ to be the volume of R_q^k and define its probability content to be,

$$u(R_q^k) = \int_{X \in R_q^k} \mu_q dX$$

The probability distribution of $u(R_q^k)$ follows a beta distribution [24]. The expected value of $u(R_q^k)$ is directly related to the expected number of active points in R_q^k and satisfies,

$$[u(R_q^k)] \lessapprox \frac{2k}{|M_q|+1},$$

because $[\text{size}(N_q^k)] \lessapprox 2k$. Since $u(R_q^k) = \mu_q(X_q)V(R_q^k)$, we have,

$$\mu_q(X_q)[V(R_q^k)] \lessapprox \frac{2k}{|M_q|+1}$$
$$[V(R_q^k)] \lessapprox \frac{2k}{(|M_q|+1)\mu_q(X_q)}$$

Consider the kd-tree query procedure for finding the k-nearest neighbors of X_q. In the first step, we traverse down the kd-tree to find N_q, the leaf node containing X_q. Then, we backtrack up the incomplete kd-tree, visiting neighboring sibling subtrees that contain active points. Notice that when we backtrack to a node N_i, we are guaranteed to have searched through all active points within the subtree of N_i. Therefore, once we backtrack to node N_q^k, we are guaranteed to have searched through at least k active points. The maximum possible distance between X_q and its k^{th} nearest neighbor is bounded by the diagonal of R_q^k. Let this diagonal length be denoted by $d(R_q^k)$ and let S_q^k represent a hypercube with side length $2d(R_q^k)$, centered at X_q. When we continue to traverse up the incomplete kd-tree, we only need to search through cells that intersect S_q^k. Because R_q^k is a compact cell with a near hypercubical shape, we have

$$[V(S_q^k)] \approx G(d)[V(R_q^k)]$$
$$\lessapprox \frac{2kG(d)}{(|M_q|+1)\mu_q(X_q)},$$

where $G(d)$ is a constant dependent only on the number of dimensions. This relation can be established because a hypercube's volume is directly proportional to the volume of a hyperball with a radius equal to the hypercube's diameter. Using the expected volume, the

expected side length $e(S_q^k)$ can be estimated,

$$[e(S_q^k)] \sim \sqrt[d]{\frac{2kG(d)}{(|M_q|+1)\mu_q(X_q)}}.$$

Now that we have created a bound on the expected side length of the hypercube region that our k-nearest neighbor query algorithm searches through, we use this expected side length to bound the number of leaf cells the query algorithm inspects.

We denote a generic leaf cell (active or inactive) intersecting S_q^k by R_b. Following the same argument made for R_q^k, the probability content of R_b follows a beta distribution [24]. Thus,

$$[u(R_b)] = \frac{1}{|M|+1}$$
$$[V(R_b)] = \frac{1}{(|M|+1)\mu(X_q)}$$
$$[e(R_b)] \sim \sqrt[d]{\frac{1}{(|M|+1)\mu(X_q)}},$$

where the assumption that μ is constant within S_q^k is used. The expected number of leaf cells traversed can now be bounded by,

$$[C(S_q^k)] \sim \left(\frac{[e(S_q^k)]}{[e(R_b)]}+1\right)^d$$
$$[C(S_q^k)] \sim \left(\sqrt[d]{\frac{2kG(d)(|M|+1)\mu(X_q)}{(|M_q|+1)\mu_q(X_q)}}+1\right)^d.$$

Consider now the probability $P(X \in M_q \mid X \in M)$. Define $\epsilon(X)$ to be a hyperball centered at X with a limiting volume $dV(\epsilon(X))$, then

$$P(X \in M_q \mid X \in M) = \lim_{dV(\epsilon(X))\to 0} \frac{P\big((\epsilon(X)\cap M_q)\neq\emptyset\big)}{P\big((\epsilon(X)\cap M)\neq\emptyset\big)}.$$

Since the probability of sampling a point in M inside the region $\epsilon(X)$ can be computed by $\mu(X)dV(\epsilon(X))$, $P\big((\epsilon(X)\cap M)\neq\emptyset\big) = \sum_{i=0}^{|M|}\mu(X)dV(\epsilon(X)) = |M|\mu(X)dV(\epsilon(X))$. Thus, the equation can be simplified as follows.

$$P(X \in M_q \mid X \in M) = \lim_{dV(\epsilon(X))\to 0} \frac{|M_q|\mu_q(X)dV(\epsilon(X))}{|M|\mu(X)dV(\epsilon(X))} \leq \frac{(|M_q|+1)\mu_q(X)}{(|M|+1)\mu(X)},$$

,

where the approximation at the last step is valid because $|M_q| >> 1$ and $|M| >> 1$.

For points inside the region S_q^k, μ_q and μ are constant. Thus, the value of $P(X \in M_q \mid X \in M)$ is also constant within the region. We denote this value of constancy by $p(X_q)$. The expression for $p(X_q)$ can be substituted back into the approximate expression for $[C(S_q^k)]$, resulting in,

$$[C(S_q^k)] \sim \left(\sqrt[d]{\frac{2kG(d)(|M|+1)\mu(X_q)}{(|M_q|+1)\mu_q(X_q)}}+1\right)^d \leq \left(\sqrt[d]{\frac{2kG(d)}{p(X_q)}}+1\right)^d.$$

This is the expected number of leaf cells intersecting S_q^k. If we let $[C_q(S_q^k)]$ represent the expected number of active leaf cells. $[C_q(S_q^k)]$ can be computed by,

$$[C_q(S_q^k)] = p(X_q)[C(S_q^k)] \sim \left(\sqrt[d]{2kG(d)} + \sqrt[d]{p(X_q)}\right)^d \leq \left(\sqrt[d]{2kG(d)}+1\right)^d.$$

We assume d to be constant. Because only the active leaf cells are visited, the k-nearest neighbor query algorithm traverses $O(k)$ expected number of leaf cells. Visiting each leaf cell takes $O(\log(n))$ time, thus giving a total expected query time complexity of $O(k\log(n))$.

The k-nearest neighbor query can also be performed in parallel, in which case we first traverse down the kd-tree to find the leaf node N_q containing the query point X_q, and then backtracks up the kd-tree to find the smallest subtree containing at least k active points, N_q^k. After this, we only need to inspect kd-tree cells that intersect the spherical region S_q^k. These cells can be inspected in parallel. As a result, the span complexity of a parallel k-nearest neighbor query can be bounded by $O(\log(n))$ in the worst case. The work complexity remains $O(k\log(n))$ in expectation and $O(n)$ in the worst case.

This analysis not only bounds the k-nearest neighbor query runtime for the incomplete kd-tree used in Section 6.1, but also for the priority search kd-tree and max kd-tree [28] described in Section 6.2, since performing nearest neighbor on them is equivalent to performing nearest neighbor query on a normal incomplete kd-tree; the equivalence has been discussed in Section 6.2.2.

It should also be noted that our analysis differs from Friedman et al. [23]'s analysis for a normal kd-tree. Friedman et al. [23] bounded the expected volume of a hyperball containing X_q's k-nearest neighbors and produced a bound on the expected number of cells intersecting the hyperball. Such an approach is deficient in that the number of cells the normal kd-tree query algorithm actually visits is not directly bounded by the number of cells intersecting the hyperball. We evade this deficiency by choosing a different approach.

8 EXPERIMENTS

Finally, in this section, we perform experimental evaluations on the efficiency of our dependent point finding algorithms as well as our proposed optimizations to density computation.

8.1 Experiment Setup

Datasets.

We run experiments on both real world and synthetic datasets. The real world datasets we use include *GeoLife* [70], *PAMAP2* [47], *Sensor* [10, 11], and *HT* [32]. The synthetic datasets we use are produced by the *simden* and *varden* random walking based generators proposed by Gan and Tao [25]. *Simden* generates multiple clusters of points with similar density while *varden* produces multiple clusters with varying density. We also use synthetic datasets generated by a *uniform* sampler. Details of these datasets are listed in Table 1 along with the hyperparamters we chose for each dataset. The d_{cut} hyperparameter is selected such that the computed density values based on the chosen d_{cut} value is nonzero but significantly smaller than the size of the dataset. The $\rho_{\min}$ and $\delta_{\min}$ values are selected such

Name	n	d	selected d_{cut}	selected ρ_{min}	selected δ_{min}
uniform	10^3–10^7	2	30	0	100
simden	10^3–10^7	2	30	0	100
varden	10^3–10^7	2	30	0	100
GeoLife	24876978	3	1	1000	10
PAMAP2	259803	4	0.02	20	0.2
Sensor	3843160	5	0.2	5	2
HT	928991	8	0.5	30	10

Table 1: The real world datasets used in our experiments, along with their sizes (n), their dimensionality (d), and the clustering hyperparameters we select for them. Similar to Amagata and Hara [2], we trim down the dimensionality of *PAMAP2* and *Sensor* in order to obtain a collection of real world datasets with different number of dimensions.

that the total number of clusters produced by the DPC algorithm is relatively small.

Computational Environment.

We use *c2-standard-60* instances on Google Cloud for our experiments. These are 30-core machines with two way hyper-threading and are equipped with Intel 3.1 GHz Cascade Lake processors that can reach a max turbo clock-speed of 3.8 GHz. For all algorithms, 30 threads are used, and hyperthreading is enabled if it further improves the runtime of the algorithm.

Algorithms.

We study the effectiveness of our proposed optimizations for density computation and analyze the performances of our dependent point finding algorithms. The baseline algorithms used for comparisons are the state-of-the-art exact DPC method proposed by Amagata and Hara [2] and the state-of-the-art approximate DPC method[3] proposed by Amagata and Hara [2].

The algorithms studied are detailed in the list below.

(1) DPC-EXACT-BASELINE: Amagata and Hara [2]'s state-of-the-art exact DPC algorithm.
(2) DPC-APPROX-BASELINE: Amagata and Hara [2]'s state-of-the-art approximate DPC algorithm.
(3) DPC-FENWICK: a DPC algorithm that uses the fenwick tree based dependent point finding algorithm in Section 5 along with the density computation and single linkage clustering optimizations introduced in Section 4.
(4) DPC-INCOMPLETE: a DPC algorithm that uses the incomplete kd-tree based dependent point finding algorithm in Section 6.1 along with the density computation and single linkage clustering optimizations introduced in Section 4.
(5) DPC-PRIORITY: a DPC algorithm that uses the priority search kd-tree based dependent point finding algorithm in Section 6.3 along with the density computation and single linkage clustering optimizations introduced in Section 4.

Besides studying the DPC algorithms' overall performance, we also analyze the runtimes of the density computation task separately from the dependent point finding task in order to study the effectiveness of our proposed optimizations for those tasks. The single linkage clustering task is not studied separately as it takes up a negligible percentage of the overall runtime; we refer to Wang et al. [61] for an experimental demonstration of the performance the parallel union-find based single linkage clustering method we adopt.

We implement our algorithms using the libraries ParlayLib [6] and ParGeo [63]. We use C++ to implement our code and the gcc compiler with -O3 optimization level to compile the code.

[3] Amagata and Hara [2] proposed two approximate DPC algorithms. We compare our algorithms with their fastest approximate DPC algorithm.

8.2 Algorithm Runtime Comparison

We first perform cross-comparison between the runtimes of all 5 of our algorithms. Figure 3 and Table 2 shows the runtime comparison across the five DPC algorithms studied.

Comparison with exact DPC baseline.

First, all of our proposed algorithms consistently outperform DPC-EXACT-BASELINE on all datasets, both in terms of density computation and dependent point finding. Our optimized density computation method outperform the baseline exact density computation by 1.4–18586.3x, with a geometric mean of 61.2x. For dependent point finding, our fenwick tree based method outperform the baseline by 12.9–1551.7x, with a geometric mean of 131.2x. Our incomplete kd-tree based method achieves a speedup of 0.9–675.9x, with a geometric mean of 31.9x (a < 1 speedup signifies a slowdown). Our priority search kd-tree based method attains a speedup of 8.3–4666.3x, with a geometric mean of 233.3x.

We also note that Figure 3 shows that our DPC algorithms are able to scale to high dimensional datasets far better than Amagata and Hara [2]'s baseline exact DPC algorithm. For this reason, we also provide the speedup numbers for low dimensional datasets (2D synthetic datasets). Our density computation optimization is still able to achieve a 1.4–4.0x speedup, with a geometric mean of 2.5x. On low dimensional datasets, our DPC-FENWICK's dependent point finding algorithm outperforms the baseline by 12.9–64.7x, with a geometric mean of 39.8x; DPC-INCOMPLETE's dependent point finder outperforms the baseline by 5.0–28.0x, with a geometric mean of 15.3x; DPC-PRIORITY's dependent point finder attains a speedup of 54.2–209.1x, with a geometric mean of 128.0x.

Comparison with approximate DPC baseline.

Our best exact DPC algorithm, DPC-PRIORITY, is able to achieve runtimes that are superior to the approximate DPC baseline on most datasets. DPC-FENWICK and DPC-INCOMPLETE can also achieve competitive results when compared to DPC-APPROX-BASELINE. Over all datasets, our optimized density computation method attains a 1.7–6828.5x speedup over the baseline density approximation algorithm used by Amagata and Hara [2] for DPC; the geometric mean speedup is 23.1x. DPC-FENWICK's dependent point finder outperform the approximate dependent point finder by 0.2–536.2x, with a geometric mean of 4.6x; DPC-INCOMPLETE's dependent point finder is able to outperform the baseline by 0.005–232.2x, with a geometric mean of 1.3x; DPC-PRIORITY's dependent point finder can outperform it by 0.04–1534.1x, with a geometric mean of 8.2x. The range of speedups achieved varies significantly across datasets primarily because DPC-APPROX-BASELINE's performance is highly dependent on the different distribution of points in each dataset. It should be noted that DPC-PRIORITY's dependent point finder is only slower than the baseline approximate dependent point finder on one dataset, and achieves considerable speedup on all

Algorithm	DPC-EXACT-BASELINE		DPC-APPROX-BASELINE		DPC-FENWICK		DPC-INCOMPLETE		DPC-PRIORITY	
Datasets	density.	dep.	density.	dep.	density.	dep.	density.	dep.	density.	dep.
uniform2	30.70	91.30	NaN	NaN	7.65	7.07	7.58	18.26	7.59	1.69
simden2	3.39	290.30	2.23	6.36	1.29	3.86	1.31	11.27	1.27	1.39
varden2	1.82	250.23	5.25	2072.96	1.28	3.87	1.26	8.93	1.28	1.35
GeoLife	NaN	NaN	21.08	5.19	10.20	12.25	10.04	14.95	10.18	2.59
PAMAP2	1.76	4.65	0.83	0.026	0.037	0.11	0.052	5.13	0.037	0.56
Sensor	11850.20	2000.41	202.95	115.50	2.97	1.77	2.94	4.33	2.95	0.98
HT	5836.56	814.50	2, 144.31	0.61	0.31	0.52	0.46	1.21	0.32	0.17

Table 2: The runtime of the 5 DPC algorithms tested on real world and synthetic datasets, decomposed into the density computation component (density.) and the dependent point finding component (dep.). NaN means that the algorithm does not terminate within 48 hours. Our proposed DPC-FENWICK **and** DPC-PRIORITY **consistently outperform the state-of-the-art exact and approximate DPC algorithms.** DPC-PRIORITY **is the most performant algorithm in most scenarios.**

others. Since DPC-APPROX-BASELINE is fairly scalable with respect to the dimensionality of the datasets, we do not provide separate comparisons on low dimensional datasets.

Finally, it should be noted that Table 2 shows that density computation sometimes take up a larger portion of time than the dependent point finding task. The runtime ratio between these two tasks is dependent on the parameter choice for d_{cut}. For our performance analysis, d_{cut} is chosen crudely such that the computed density values are nonzero but are $<< n$. For carefully selected d_{cut} values, the dependent point finding task is more likely to occupy the major portion of time [2].

8.3 Scalability Analysis

We analyze the scalability of our algorithms by performing experiments on synthetic datasets of varying sizes and running the algorithms on different numbers of threads. We use datasets generated by *simden* for scalability analysis because DPC-APPROX-BASELINE, when running on a single thread, does not terminate for the largest *uniform* and *varden* datasets within 48 hours.

Scalability over the size of the dataset.

We analyze the scalability of all 5 DPC algorithms over 2D *simden* datasets of varying sizes (from 10^3 points to 10^7 points); we keep the dimensionality of the datasets tested low because DPC-EXACT-BASELINE does not scale to high dimensional datasets well. Figure 4a shows the runtime of all five DPC algorithms over *simden* datasets of different sizes. It should be noted that our DPC-PRIORITY is able to outperform both the exact DPC baseline and the approximate DPC baseline for *simden* datasets of most sizes. Further, we note that the runtime of our proposed algorithms increase much slower than DPC-EXACT-BASELINE. From *simden* with 10^3 points to *simden* with 10^7 points, the runtime of DPC-EXACT-BASELINE increases by about $1.5 \cdot 10^5$x. In comparison, DPC-FENWICK's runtime increases by only 10^4x; DPC-INCOMPLETE's runtime increases by $1.4 \cdot 10^4$x, and DPC-PRIORITY's runtime increases by $4.5 \cdot 10^3$x. This demonstrates that our algorithm has superior scalability over different graph sizes, which is expected since our algorithms are able to attain stronger complexity results than Amagata and Hara [2]'s algorithms.

Parallel scalability.

Finally, we investigate the parallel scalability of our algorithms. Figure 4b clearly shows that all of our proposed DPC algorithms attain better parallel scallability than the exact DPC baseline. This is expected, because of the fact that we reduced the span complexity from the baseline's $O(n^2)$ to $O(n \log(n))$ for DPC-INCOMPLETE and $O(\log(n) \log\log(n))$ for both DPC-PRIORITY and DPC-FENWICK.

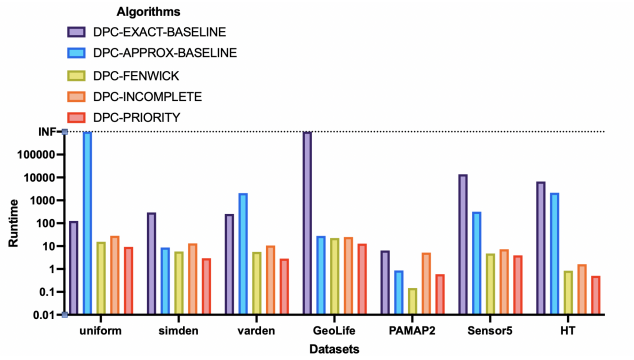
DPC-FENWICK is able to achieve a 8.8x self-relative speedup when running on 60 threads. DPC-PRIORITY, in comparison, achieves a 13.2x self-relative speedup. Both are superior to the 1.3x self-relative speedup attained by DPC-EXACT-BASELINE and are competitive against the 14.4x self-relative speedup achieved by DPC-APPROX-BASELINE.

9 CONCLUSION

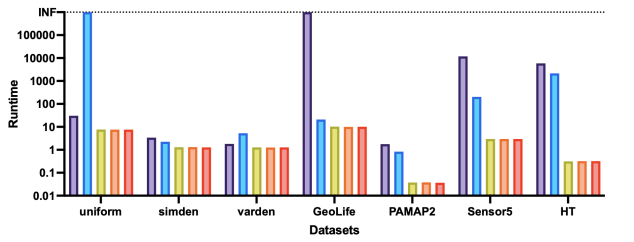
In this paper, we develop theoretically efficient parallel algorithms for performing Density-Peaks Clustering (DPC), an established density-based clustering method with wide applications. We introduce optimizations for the density computation and the single linkage clustering subtasks of DPC. We further propose 3 algorithms for solving the dependent point finding subtask of DPC. Our best proposed algorithm is able to dramatically improve upon the work complexity and span complexity of the state-of-the-art solution, cutting the work complexity from $O(n^2)$ to $O(n \log(n))$ in expectation, and reducing the span complexity from $O(n^2)$ to $O(\log(n) \log\log(n))$. We also introduce priority search kd-tree, a data structure used in our DPC algorithm, and provide proof for the runtime complexity of performing queries on a priority search kd-tree. Finally, we conduct extensive experimental analysis of the performances of our proposed algorithms in comparison to the state-of-the-art exact and approximate DPC algorithms. Our best algorithm is able to achieve a geometric mean speedup of 233.3x over the state-of-the-art exact DPC solution. It is also able to outperform the best approximate DPC algorithm on almost all datasets, attaining a geometric mean speedup of 8.2x. We further show that our algorithms are scalable to datasets of different sizes and parallel computing machines with different number of cores. When running on a 30-core machine with two-way hyperthreading, our fastest algorithm attains a self-relative speedup of 13.2x.

10 FUTURE WORK

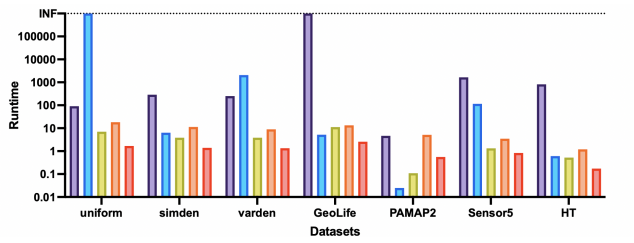
In the future, we intend to further improve the runtimes of our algorithms via performance engineering. We also wish to explore the possibility of applying a Balanced-Aspect-Ratio tree [18] to solving the DPC dependent point finding task, which can potentially result in an algorithm with a tighter, non-probabilistic work bound.



(a) Total runtime of DPC algorithms

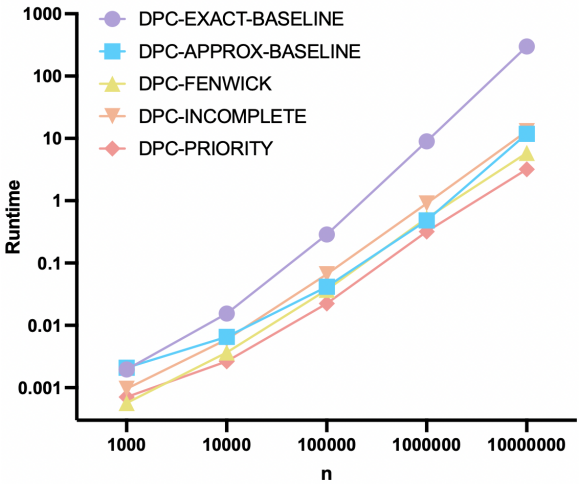


(b) Density computation component

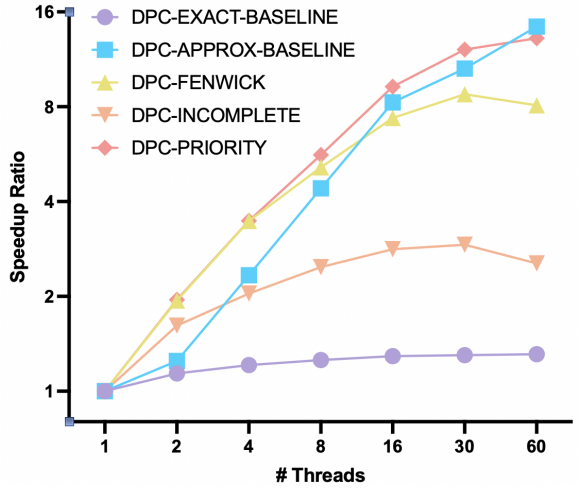


(c) Dependent point finding component

Figure 3: Plot of the running time of all DPC algorithms studied on four real world datasets. All algorithms are run on a 30-core machine. For each algorithm, two-way hyperthreading is enabled if it further improves the algorithm's performance. All units are in seconds and the y-axis is in logarithmic scale. Some algorithms do not have a runtime for a dataset because they do not terminate within 48 hours. It is clear from these plots that our proposed dependent point finding algorithms and density computation optimizations achieve considerable improvement in comparison to the baseline methods.



(a) Scalability analysis over datasets of different sizes



(b) Scalability analysis over different number of threads used

Figure 4: Runtime of all 5 DPC algorithms on *simden* datasets over different sizes and the speedup ratios of the DPC algorithms over different number of threads (note that "60 threads" means a 30-core environment with two-way hyperthreading). The runtime units are in seconds and all axis use logarithmic scale. Our proposed algorithms clearly scale better than the exact DPC baseline algorithm.

Beyond DPC, we wish to explore the application value of the priority search *k*d-tree data structure in solving other computational challenges.

ACKNOWLEDGEMENTS

We would like to thank Shangdi Yu and Professor Julian Shun (MIT CSAIL) for their amazing support and guidance throughout the year. In addition, we are grateful for the MIT PRIMES organization for this opportunity.

REFERENCES

[1] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. 1998. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications *(SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/276304.276314

[2] Daichi Amagata and Takahiro Hara. 2021. *Fast Density-Peaks Clustering: Multicore-Based Parallelization Approach.* Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/3448016.3452781

[3] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/304182.304187

[4] 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. https://doi.org/10.1016/j.patcog.2017.06.023

[5] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (sep 1975), 509–517. https://doi.org/10.1145/361002.361007

[6] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. *ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines*. Association for Computing Machinery, New York, NY, USA, 507–509. https://doi.org/10.1145/3350755.3400254

[7] 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, 2004. proceedings of*. IEEE, 92–96.

[8] Kaley Brauer, Hillary D Andales, Alexander P Ji, Anna Frebel, Mohammad K Mardini, Facundo A Gomez, and Brian W O'Shea. 2022. Possibilities and Limitations of Kinematically Identifying Stars from Accreted Ultra-Faint Dwarf Galaxies. *arXiv preprint arXiv:2206.07057* (2022).

[9] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (April 1974), 201–206.

[10] Javier Burgués, Juan Manuel Jiménez-Soto, and Santiago Marco. 2018. Estimation of the limit of detection in semiconductor gas sensors through linearized calibration models. *Analytica chimica acta* 1013 (2018), 13–25.

[11] Javier Burgués and Santiago Marco. 2018. Multivariate estimation of the limit of detection by orthogonal partial least squares in temperature-modulated MOX sensors. *Analytica chimica acta* 1019 (2018), 49–64.

[12] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. 2013. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*. Springer, 160–172.

[13] Yewang Chen, Xiaoliang Hu, Wentao Fan, Lianlian Shen, Zheng Zhang, Xin Liu, Jixiang Du, Haibo Li, Yi Chen, and Hailin Li. 2019. Fast density peak clustering for large scale data based on kNN. *Knowledge-Based Systems* 187 (07 2019). https://doi.org/10.1016/j.knosys.2019.06.032

[14] 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. https://doi.org/10.1175/JCLI-D-20-0243.1

[15] G.B. Coleman and H.C. Andrews. 1979. Image segmentation by clustering. *Proc. IEEE* 67, 5 (1979), 773–785. https://doi.org/10.1109/PROC.1979.11327

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3. ed.)*. MIT Press.

[17] Hui Du, Yanting Hao, and Zhihe Wang. 2021. An improved density peaks clustering algorithm by automatic determination of cluster centres. *Connection Science* (12 2021), 1–17. https://doi.org/10.1080/09540091.2021.2012422

[18] Christian A Duncan, Michael T Goodrich, and Stephen Kobourov. 2001. Balanced aspect ratio trees: Combining the advantages of kd trees and octrees. *Journal of Algorithms* 38, 1 (2001), 303–333.

[19] Bernardt Duvenhage. 2009. Using an implicit min/max kd-tree for doing efficient terrain line of sight calculations. In *Proceedings of the 6th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*. 81–90.

[20] 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 2003 SIAM international conference on data mining*. SIAM, 47–58.

[21] 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 Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*. AAAI Press, 226–231.

[22] Peter M. Fenwick. 1994. A new data structure for cumulative frequency tables. *Software: Practice and Experience* 24 (1994).

[23] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. 1977. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Softw.* 3, 3 (sep 1977), 209–226. https://doi.org/10.1145/355744.355745

[24] K. Fukunaga and L. Hostetler. 1973. Optimization of k nearest neighbor density estimates. *IEEE Transactions on Information Theory* 19, 3 (1973), 320–326. https://doi.org/10.1109/TIT.1973.1055003

[25] Junhao Gan and Yufei Tao. 2017. On the Hardness and Approximation of Euclidean DBSCAN. *ACM Trans. Database Syst.* 42, 3, Article 14 (jul 2017), 45 pages. https://doi.org/10.1145/3083897

[26] S. Gong and Yanfeng Zhang. 2016. EDDPC: An efficient distributed density peaks clustering algorithm. 53 (06 2016), 1400–1409. https://doi.org/10.7544/issn1000-1239.2016.20150616

[27] 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.

[28] Matthias Groß, Carsten Lojewski, Martin Bertram, and Hans Hagen. 2007. Fast implicit kd-trees: Accelerated isosurface ray tracing and maximum intensity projection for large scalar fields. In *Proc. Computer Graphics and Imaging*. Citeseer, 67–74.

[29] Xinyi Guo, Yuanyuan Zhang, Liangtao Zheng, Chunhong Zheng, Jintao Song, Qiming Zhang, Boxi Kang, Zhouzerui Liu, Liang Jin, Rui Xing, Ranran Gao, Lei Zhang, Minghui Dong, Xueda Hu, Xianwen Ren, Dennis Kirchhoff, Helge Gottfried Roider, Tiansheng Yan, and Zemin Zhang. 2018. Global characterization of T cells in non-small-cell lung cancer by single-cell sequencing. *Nature medicine* 24, 7 (July 2018), 978—985. https://doi.org/10.1038/s41591-018-0045-3

[30] 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. https://doi.org/10.5120/ijca2018917041

[31] 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 (KDD'98)*. AAAI Press, 58–65.

[32] Ramon Huerta, Thiago Mosqueiro, Jordi Fonollosa, Nikolai F Rulkov, and Irene Rodriguez-Lujan. 2016. Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring. *Chemometrics and Intelligent Laboratory Systems* 157 (2016), 169–176.

[33] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. 2004. DBDC: Density Based Distributed Clustering, Vol. 2992. 88–105. https://doi.org/10.1007/978-3-540-24741-8_7

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

[35] Zhenni Jiang, Xiyu Liu, and Minghe Sun. 2019. A Density Peak Clustering Algorithm Based on the K-Nearest Shannon Entropy and Tissue-Like P System. *Mathematical Problems in Engineering* 2019 (07 2019), 1–13. https://doi.org/10.1155/2019/1713801

[36] Nawsher Khan, Ibrar Yaqoob, Ibrahim Hashem, Zakira Inayat, Waleed Kamaleldin, Muhammad Alam, Muhammad Shiraz, and Abdullah Gani. 2014. Big Data: Survey, Technologies, Opportunities, and Challenges. *The Scientific World Journal* 2014 (07 2014), 18. https://doi.org/10.1155/2014/712826

[37] H.-P. Kriegel and M. Pfeifle. 2005. Hierarchical density-based clustering of uncertain data. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*. 4 pp.–. https://doi.org/10.1109/ICDM.2005.75

[38] Der-Tsai Lee and Chak-Kuen Wong. 1977. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica* 9, 1 (1977), 23–29.

[39] Fengfu Li, Hong Qiao, and Bo Zhang. 2018. Discriminatively boosted image clustering with fully convolutional auto-encoders. *Pattern Recognition* 83 (2018), 161–173. https://doi.org/10.1016/j.patcog.2018.05.019

[40] 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. https://doi.org/10.1109/TEVC.2018.2866927

[41] Songrit Maneewongvatana and David M Mount. 1999. It's okay to be skinny, if your friends are fat. In *Center for geometric computing 4th annual workshop on computational geometry*, Vol. 2. 1–8.

[42] 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. https://doi.org/10.1162/COLI_a_00148

[43] Edward M. McCreight. 1985. Priority Search Trees. *SIAM J. Comput.* 14, 2 (1985), 257–276. https://doi.org/10.1137/0214021 arXiv:https://doi.org/10.1137/0214021

[44] Rashid Mehmood, Saeed El-Ashram, Rongfang Bie, Hussain Dawood, and Anton Kos. 2017. Clustering by fast search and merge of local density peaks for gene expression microarray data. *Scientific reports* 7, 1 (2017), 1–7.

[45] Robert J. Peters and Laurent Itti. 2007. Beyond bottom-up: Incorporating task-dependent influences into a computational model of spatial attention. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*. 1–8. https://doi.org/10.1109/CVPR.2007.383337

[46] Sanguthevar Rajasekaran and John H. Reif. 1989. Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms. *SIAM J. Comput.* 18, 3 (1989), 594–607. https://doi.org/10.1137/0218041 arXiv:https://doi.org/10.1137/0218041

[47] Attila Reiss and Didier Stricker. 2012. Introducing a new benchmarked dataset for activity monitoring. In *2012 16th international symposium on wearable computers*. IEEE, 108–109.

[48] John T Robinson. 1981. The KDB-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 10–18.

[49] Alex Rodriguez and Alessandro Laio. 2014. Clustering by fast search and find of density peaks. *Science* 344, 6191 (2014), 1492–1496. https://doi.org/10.1126/science.1242072 arXiv:https://www.science.org/doi/pdf/10.1126/science.1242072

[50] F James Rohlf. 1982. 12 Single-link clustering algorithms. *Handbook of statistics* 2 (1982), 267–284.

[51] Sanjiv Sharma and Rajendra Gupta. 2010. Improved BSP Clustering Algorithm for Social Network Analysis. *International Journal of Grid and Distributed*

Computing 3 (01 2010).

[52] 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 (feb 2000), 289–304. https://doi.org/10.1007/s007780050009

[53] Abir Smiti and Zied Eloudi. 2013. Soft DBSCAN: Improving DBSCAN clustering method using fuzzy set theory. In *2013 6th International Conference on Human System Interactions (HSI)*. 380–385. https://doi.org/10.1109/HSI.2013.6577851

[54] Yifan Sun, Nicolas Agostini, Shi Dong, and David Kaeli. 2019. Summarizing CPU and GPU Design Trends with Product Data.

[55] Apinya Tepwankul and Songrit Maneewongwattana. 2010. U-DBSCAN: A density-based clustering algorithm for uncertain objects. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. IEEE, 136–143.

[56] Balaji Venu. 2011. Multi-core processors - An overview. (10 2011).

[57] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek, and H-P Seidel. 2005. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics* 11, 5 (2005), 562–572.

[58] Peng Wang, Bo Xu, Jiaming Xu, Guanhua Tian, Cheng-Lin Liu, and Hongwei Hao. 2016. Semantic expansion using word embedding clustering and convolutional neural network for improving short text classification. *Neurocomputing* 174 (2016), 806–814. https://doi.org/10.1016/j.neucom.2015.09.096

[59] Wei Wang, Jiong Yang, and Richard R. Muntz. 1997. STING: A Statistical Information Grid Approach to Spatial Data Mining. In *VLDB*.

[60] Yi Wang, Qixin Chen, Chongqing Kang, and Qing Xia. 2016. Clustering of Electricity Consumption Behavior Dynamics Toward Big Data Applications. *IEEE Transactions on Smart Grid* 7, 5 (2016), 2437–2447. https://doi.org/10.1109/TSG.2016.2548565

[61] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-efficient and practical parallel DBSCAN. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2555–2571.

[62] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: a library for parallel computational geometry. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 450–452.

[63] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: A Library for Parallel Computational Geometry. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 450–452. https://doi.org/10.1145/3503221.3508429

[64] 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. *J. Data and Information Quality* 14, 3, Article 15 (may 2022), 23 pages. https://doi.org/10.1145/3491232

[65] Xiao Xu, Shifei Ding, Yanru Wang, Lijuan Wang, and Weikuan Jia. 2021. A fast density peaks clustering algorithm with sparse search. *Information Sciences* 554 (2021), 61–83. https://doi.org/10.1016/j.ins.2020.11.050

[66] 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. https://doi.org/10.1016/S0730-725X(02)00477-0

[67] Rahul Yesantharao. 2022. *Parallel Batch-Dynamic kd-trees*. Master's thesis. Massachusetts Institute of Technology.

[68] Xiaoning Yuan, Hang Yu, Jun Liang, and Xu Bing. 2020. A Novel Density Peaks Clustering Algorithm Based on K Nearest Neighbors With Adaptive Merging Strategy. https://doi.org/10.21203/rs.3.rs-95747/v1

[69] 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. https://doi.org/10.1109/TKDE.2016.2609423

[70] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning transportation mode from raw gps data for geographic applications on the web. In *Proceedings of the 17th international conference on World Wide Web*. 247–256.

[71] Carly G.K. Ziegler, Samuel J. Allon, Sarah K. Nyquist, Ian M. Mbano, Vincent N. Miao, Constantine N. Tzouanas, Yuming Cao, Ashraf S. Yousif, Julia Bals, Blake M. Hauser, Jared Feldman, Christoph Muus, Marc H. Wadsworth, Samuel W. Kazer, Travis K. Hughes, Benjamin Doran, G. James Gatter, Marko Vukovic, Faith Taliaferro, Benjamin E. Mead, Zhiru Guo, Jennifer P. Wang, Delphine Gras, Magali Plaisant, Meshal Ansari, Ilias Angelidis, Heiko Adler, Jennifer M.S. Sucre, Chase J. Taylor, Brian Lin, Avinash Waghray, Vanessa Mitsialis, Daniel F. Dwyer, Kathleen M. Buchheit, Joshua A. Boyce, Nora A. Barrett, Tanya M. Laidlaw, Shaina L. Carroll, Lucrezia Colonna, Victor Tkachev, Christopher W. Peterson, Alison Yu, Hengqi Betty Zheng, Hannah P. Gideon, Caylin G. Winchell, Philana Ling Lin, Colin D. Bingle, Scott B. Snapper, Jonathan A. Kropski, Fabian J. Theis, Herbert B. Schiller, Laure-Emmanuelle Zaragosi, Pascal Barbry, Alasdair Leslie, Hans-Peter Kiem, JoAnne L. Flynn, Sarah M. Fortune, Bonnie Berger, Robert W. Finberg, Leslie S. Kean, Manuel Garber, Aaron G. Schmidt, Daniel Lingwood, Alex K. Shalek, Jose Ordovas-Montanes, Nicholas Banovich, Pascal Barbry, Alvis Brazma, Tushar Desai, Thu Elizabeth Duong, Oliver Eickelberg, Christine Falk, Michael Farzan, Ian Glass, Muzlifah Haniffa, Peter Horvath, Deborah Hung, Naftali Kaminski, Mark Krasnow, Jonathan A. Kropski, Malte Kuhnemund, Robert Lafyatis, Haeock Lee, Sylvie Leroy, Sten Linnarson, Joakim Lundeberg, Kerstin Meyer, Alexander Misharin, Martijn Nawijn, Marko Z. Nikolic, Jose Ordovas-Montanes, Dana Pe'er, Joseph Powell, Stephen Quake, Jay Rajagopal, Purushothama Rao Tata, Emma L. Rawlins, Aviv Regev, Paul A. Reyfman, Mauricio Rojas, Orit Rosen, Kourosh Saeb-Parsy, Christos Samakovlis, Herbert Schiller, Joachim L. Schultze, Max A. Seibold, Alex K. Shalek, Douglas Shepherd, Jason Spence, Avrum Spira, Xin Sun, Sarah Teichmann, Fabian Theis, Alexander Tsankov, Maarten van den Berge, Michael von Papen, Jeffrey Whitsett, Ramnik Xavier, Yan Xu, Laure-Emmanuelle Zaragosi, and Kun Zhang. 2020. SARS-CoV-2 Receptor ACE2 Is an Interferon-Stimulated Gene in Human Airway Epithelial Cells and Is Detected in Specific Cell Subsets across Tissues. *Cell* 181, 5 (2020), 1016–1035.e19. https://doi.org/10.1016/j.cell.2020.04.035