Faster Parallel Exact Density Peaks Clustering

Yihao Huang[*] [†] Shangdi Yu[‡] [†] Julian Shun[‡]

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 parallel algorithms for Density Peaks Clustering (DPC), a popular version of density-based clustering. Existing exact DPC algorithms suffer from low parallelism both in theory and in practice, which limits their application to large-scale data sets. Our most performant algorithm, which is based on priority search kd-trees, achieves $O(\log n \log\log n)$ span (parallel time complexity) for a data set of n points. Our algorithm is also work-efficient, achieving a work complexity matching the best existing sequential exact DPC algorithm. In addition, we present another DPC algorithm based on a Fenwick tree that makes fewer assumptions for its average-case complexity to hold.

We provide optimized implementations of our algorithms and evaluate their performance via extensive experiments. On a 30-core machine with two-way hyperthreading, we find that our best algorithm achieves a 10.8–13169x speedup over the previous best parallel exact DPC algorithm. Compared to the state-of-the-art parallel approximate DPC algorithm, our best algorithm achieves a 1.5–4206x speedup, while being exact.

1 Introduction

Clustering is the task of grouping similar objects into clusters and it is a fundamental task in data analysis and unsupervised machine learning. Clustering algorithms can be used to identify different types of tissues in medical imaging [69], analyze social networks, and identify weather regimes in climatology [17]. They are also widely used as a data processing subroutine in other machine learning tasks [18, 67, 44, 47]. One popular type of clustering is density-based clustering, which defines clusters as dense regions of points in the coordinate space. Density-based clustering algorithms have received a lot of attention [23, 2, 5, 38, 55, 64, 34, 33, 57], because they can discover clusters of arbitrary shapes, while many other popular algorithms, such as k-means, can only recover separable clusters with spherical shapes.

Density peak clustering (DPC) [55] is a popular version of density-based clustering. In this paper, we present fast parallel exact algorithms for DPC that outperform existing state-of-the-art implementations. Many density-based clustering algorithms, such as DBSCAN [23], are sensitive towards the choice of a density-noise cutoff hyper-parameter (points with density lower than the cutoff are deemed as irrelevant noise) [23]. DPC, in comparison, has been shown to perform well consistently over different hyper-parameter choices [55]. It is also very easy to set the hyper-parameters of DPC because DPC can generate a decision graph [55] that visually aids the determination of the hyper-parameters. Due to its advantages, DPC has been applied in many situations, such as the analysis of pathogenesis of COVID-19 [75], cancer studies [32], neuroscience studies [50], market analysis [65], computer vision tasks [43], and natural language processing [63]. DPC has three main steps:

1. Compute the density of each point x, which is the number of points in a ball centered at x with a user-input parameter radius, d_{cut}.
2. For each point x, connect x to its dependent point, which is the closest neighbor of x that has a higher density than x. The resulting graph is a tree.
3. Remove 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 [56] on the tree.

For a data set of n points, a naive implementation of DPC that computes all pairwise point distances takes $\Theta(n^2)$ work to compute the density of all points and to connect each point to its dependent point [55]. This is expensive when the data set is large. Hence, multiple works have attempted to optimize the computational cost of DPC [6, 68, 29, 73, 52, 3]. Rasool et al. [52] propose a sequential algorithm with average-case work complexity of $O(n \log n)$. Amagata and Hara [3] proposed a parallel algorithm that uses a kd-tree to compute density values and find dependent points; it is currently the state-of-the-art parallel DPC algorithm for exact DPC clustering. Their algorithm is able to achieve the same average-case work complexity as Rasool et al. [52]'s algorithm, and their worst-case span[1] complexity is $O(n \log n)$.[2] We will describe more about related work in Section 2.

As the sizes of modern data sets increase, it is important for clustering algorithms to have high parallelism, and

[*]Phillips Academy

[†]The first two authors contributed equally to this work.

[‡]MIT CSAIL

[1]The *span* is the length of the longest chain of sequential dependencies in the algorithm.

[2]Amagata and Hara [3]'s implementation has a $O(n^2)$ span complexity, but it can be trivially reduced to a $O(n \log n)$ span by parallelizing the kd-tree nearest neighbor search.

ideally have polylogarithmic span. In this work, we develop parallel DPC algorithms to improve the span complexity of existing DPC algorithms. Our algorithms are able to achieve $O(\log n \log\log n)$ worst case span complexity. We present new parallel algorithms for Step 2—the parallelism bottleneck for Amagata and Hara [3]'s algorithm. Our algorithms significantly reduce Step 2's span complexity. We also optimize existing parallel algorithms for Steps 1 and 3.

Our first new algorithm for solving Step 2's dependent point finding task utilizes a ***priority search kd-tree***. A priority search kd-tree is an optimization of a max kd-tree [31, 21]. The priority search kd-tree can be constructed from a data set of n points similar to a regular kd-tree [66] in $O(n \log n)$ work and $O(\log n \log\log n)$ span. A priority search kd-tree can be used to directly find the dependent point of a point x. It is designed to process queries for the nearest neighbor of x with a higher density value than x itself. Hence, to compute the dependent point for every point in the data set, we only need to perform priority search kd-tree queries for every point in the data set in parallel. The parallelism across different priority search kd-tree queries allows our algorithm to avoid sequentiality in Amagata and Hara [3]'s algorithm, achieving $O(\log n)$ span complexity for finding dependent points while maintaining $O(n \log n)$ average-case work. Since the priority search kd-tree construction has a span complexity of $O(\log n \log\log n)$, the overall span complexity of the algorithm is $O(\log n \log\log n)$.

We also present a parallel Fenwick tree-based algorithm for finding dependent points. This 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 algorithm takes $O(n^2)$ work in the worst case, but its average-case work is $O(n \log^2 n)$. The span is again bounded by $O(\log n \log\log n)$. Although this algorithm has a higher average-case work bound than our priority-search tree algorithm, its average-case complexity result requires fewer assumptions and it can sometimes be faster in practice.

In addition to our two dependent point finding algorithms, we also introduce an optimization technique for density computation: while counting the number of points within the neighborhood of a particular query point, we prune the searches through kd-tree subtrees completely contained within that neighborhood by storing the number of points each subtree contains inside the kd-tree and directly adding that number to the total number of points. Finally, we solve the single-linkage clustering step of the algorithm (Step 3) by using a parallel union-find data structure [39], which has $O(n\alpha(n, n))$ expected work and $O(\log n)$ span with high probability, where α represents the inverse Ackermann's function.

We implement our algorithms and evaluate them on both synthetic and real-world data sets. We compare our runtime results to state-of-the-art exact and approximate DPC algorithms [3, 52] on a 30-core machine with two-way hyper-threading. Our optimized density computation algorithm outperforms state-of-the-art [3] parallel exact density computation by 1.4–18586.3x. For dependent point finding, our parallel Fenwick tree based algorithm achieves 12.9–1551.7x speedup over state-of-the-art [3] algorithm and our parallel priority search kd-tree based approach attains 8.3–4666.3x speedup. Considering the overall runtime, our best algorithm achieves a 10.8–13169x speedup over the previous best parallel exact DPC algorithm. Since our algorithms are exact, they give the same clustering quality as the original DPC algorithm. Compared to the state-of-the-art parallel approximate DPC algorithm, our best algorithm achieves a 1.5–4206x speedup, while being exact.

Our contributions are threefold:

1. We introduce two novel algorithms for solving the dependent point finding task in DPC, and introduce techniques for speeding up the density computation and single-linkage clustering tasks in DPC.
2. We provide theoretical bounds of our algorithms as well as the priority search kd-tree data structure.
3. We provide fast implementations of our algorithms and perform extensive experimental evaluations showing that our implementations outperform state-of-the-art implementations by up to orders of magnitude.

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

2 Related Work

2.1 kd-trees and K-nearest neighbor queries In this work, we use kd-trees and a variant of it for K-nearest neighbor query and range search. There are also variants of kd-trees that are specialized for other tasks. Maneewongvatana and Mount [46] proved that a kd-tree that adopts a sliding midpoint 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 $O(\log n)$ average-case query complexity. Wald et al. [62] proposed implicit kd-trees, which define the partitioning of space using a recursive splitting-function and is applied in ray tracing. Robinson [54] proposed the K-D-B-tree, which is used to organize large point sets stored in secondary memory. Groß et al. [31] proposed the 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

	Algorithm	worst-case work	average-case work	worst-case span
Density Computation	*Original DPC* [55]*	$\Theta(n^2)$	$\Theta(n^2)$	$O(\log n)$
	Exact Baseline DPC [3]*	$O(n(n^{1-\frac{1}{d}} + \varrho_{\text{avg}}))$	$O(\min(n(n^{1-\frac{1}{d}} + \varrho_{\text{avg}}), n\varrho_{\text{avg}} \log n))$	$O(\log n \log\log n)$
	R-tree DPC [52]*	$O(n^2)$	–	$O(\log^2 n)$
	Fenwick DPC (Ours)	$O(n(n^{1-\frac{1}{d}} + \varrho_{\text{avg}}))$	$O(\min(n(n^{1-\frac{1}{d}} + \varrho_{\text{avg}}), n\varrho_{\text{avg}} \log n))$	$O(\log n \log\log n)$
	Priority DPC (Ours)	$O(n(n^{1-\frac{1}{d}} + \varrho_{\text{avg}}))$	$O(\min(n(n^{1-\frac{1}{d}} + \varrho_{\text{avg}}), n\varrho_{\text{avg}} \log n))$	$O(\log n \log\log n)$
Dependent Point Finding	*Original DPC* [55]*	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$
	Exact Baseline DPC [3]*	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
	R-tree DPC [52]*	$O(n^2)$	–	$O(\log^2 n)$
	Fenwick DPC (Ours)	$O(n^2)$	$O(n \log^2 n)$	$O(\log n \log\log n)$
	Priority DPC (Ours)	$O(n^2)$	$O(n \log n)$	$O(\log n \log\log n)$

Table 1: Average-case work and worst-case work and span of DPC algorithms. The span for the algorithms marked with * is the span of a trivial parallelization of the algorithm. ϱ_{avg} is the average density, where the density value of each point equals the number of points around it within a hyper-cubical region with a side length of $2d_{\text{cut}}$. This is different from the ρ density parameter for DPC, because it is computed as the number of points around it within a hyperball with radius d_{cut}. "–" indicates that we were unable to find a bound with a proof.

value amongst all points stored under the subtree of that node, which can be used to prune searches [62, 31]. Our proposed priority search kd-tree is an optimized variant of a max kd-tree. It can also be viewed as a generalization of the priority search tree data structure [48] to higher dimensions.

2.2 Density Peaks Clustering (DPC) Many variants of the standard DPC [55] have been developed [15, 49, 63, 72, 20, 40, 68], and there has also been a line of work focused on improving the computational efficiency of the standard DPC algorithm. The naive DPC algorithm takes $\Theta(n^2)$ work, and can be implemented to take $O(\log n)$ span for density computation and $O(1)$ span for depending point finding.[3] Bai et al. [6] 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 et al. [29] parallelized DPC in a distributed setting and employed Voronoi diagrams to improve its efficiency. Liu et al. [45] proposed a DPC algorithm for GPUs. Amagata and Hara [3] leveraged the kd-tree data structure to improve the efficiency of density computation and dependent point finding. Rasool et al. [52] used an R-tree to optimize the density computation and dependent point searching efficiency. Their algorithm is sequential, but in theory, it could be parallelized by using a parallel version of R-trees [35] and performing queries in parallel. We summarize the complexity of DPC algorithms in Table 1. The average-case work bounds of our algorithms are proved in the full version of our paper.

Some works have relaxed the definition of DPC to develop efficient algorithms for approximate DPC. Zhang et al. [73] proposed LSH-DDP, a parallel DPC algorithm for distributed memory 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 necessary. Sieranoja and Fränti [59] developed an algorithm that first constructs a K-nearest neighbor graph, and then computes approximate density values and dependent points based on the K-nearest neighbor graph. Amagata and Hara [3] also proposed a parallel approximate DPC algorithm that constructs a spatial grid on top of the points. Leveraging the grid structure, the algorithm shares density and dependent point computations across all points inside the same grid cell, thus reducing the computational cost.

2.3 Density-based Clustering Algorithms DPC falls under the broad category of density-based clustering algorithms. Density-based clustering algorithms come in different varieties. Some density-based clustering algorithms define the density of a point based on the number of points in its vicinity [23, 2, 5, 38, 55, 25, 15]. Others leverage a grid-based definition [64, 34, 33, 57]. Some algorithms define density based on a probabilistic density function [64, 41, 60]. One popular density-based clustering algorithm is DBSCAN [23], which has many derivatives as well [5, 61, 30, 11, 22, 14].

3 Preliminaries

In this section, we provide the definitions and notations used in this paper. We assume that arrays are indexed from 1 to n.

Let $P = \{x_1, x_2, \ldots, x_n\}$ represent a set of n points that we need to cluster. Each point is given in d-dimensional coordinate space. We use x to denote a generic point in $\mathbb{R}^d$ and x_i to represent the i^{th} point in our point set P. 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 should be a metric distance [26].

DPC requires three parameters: d_{cut}, ρ_{min}, and δ_{min}. Intuitively, d_{cut} controls how the density is computed; ρ_{min} controls the noise level; and δ_{min} controls the granularity of clusters. Below, we formally explain how they are used.

DEFINITION 1. *Given a point $x_i \in P$ and a cutoff value d_{cut}, we define the* **density** *of x_i to be $\rho(x_i) = |\{x_j \mid x_j \in P \text{ and } D(x_i, x_j) \leq d_{cut}\}|$, i.e., the number of points inside a*

[3] Note that although the naive DPC algorithm achieves better span complexity than subsequent DPC algorithms, its work complexity is larger.

hyperball centered at x_i with radius d_{cut}.

DEFINITION 2. *Let $P_i = \{x_j \mid x_j \in P$ and $\rho(x_j) > \rho(x_i)\}$. For a point $x_i \in P$, the* **dependent point** *of x_i is a point $\lambda(x_i) \in P_i$ such that, $D(x_i, \lambda(x_i)) \leq D(x_i, x_j) \; \forall \; x_j \in P_i$.*

Given a point x_i, we define its ***dependent point set*** P_i as the set of points with density value higher than $\rho(x_i)$. When $\rho(x_i) = \rho(x_j)$ for some points x_i and x_j, the tie is broken lexicographically. The dependent point $\lambda(x_i)$ of point x_i is thus x_i's nearest neighbor in P_i.

DEFINITION 3. *Let $\delta(x_i) = D(x_i, \lambda(x_i))$ be the* **dependent distance** *of x_i. If x_i is the point with highest density in P, then it does not have a well-defined dependent point. In that case, we let $\delta(x_i) = \infty$.*

DEFINITION 4. *A point $x_i \in P$ is considered a* **noise point** *if $\rho(x_i) < \rho_{min}$ for some density cutoff ρ_{min}.*

DEFINITION 5. *x_i is considered a* **cluster center** *if $\delta(x_i) \geq \delta_{min}$ and it is not a noise point.*

Each cluster center corresponds to a separate cluster. Each non-noise point that is not a cluster center is assigned to be in the same cluster as its dependent point. Noise points do not belong to any cluster.

Thus, d_{cut}, ρ_{min}, and δ_{min} are the three hyper-parameters 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 $\delta(x_i)$ [55]. Rodriguez and Laio [55] suggest that d_{cut} can chosen such that the average number of neighbors is between $0.01n$–$0.02n$. There are also automatic parameter tuning methods [28, 71].

3.1 Model of Computation We use the ***work-span model*** [37, 19], a standard model for analyzing shared-memory parallel algorithms. The ***work*** T_1 of an algorithm is the total number of operations executed by the algorithm, and the ***span*** T_∞ is the length of the longest sequential dependency chain of the algorithm (it is also the parallel time complexity when there are an infinite number of processors). We can bound the expected running time of an algorithm on $\mathcal{P}$ processors by $T_1/\mathcal{P} + O(T_\infty)$ using a randomized work-stealing scheduler [10].

3.2 Relevant Techniques Our algorithms make use of the kd-tree [7] and Fenwick tree [24] data structures.

kd-trees. A ***kd-tree*** [7] is a binary space partitioning tree, where each internal node contains a splitting hyperplane that partitions the points contained in the node between its two children. Let the smallest bounding box containing all points in a node be the node's ***cell***. The root node contains all of the points, and the kd-tree is constructed by recursing on each of its two children after splitting, until a leaf node is reached. Each node stores the coordinates for its cell, which can be used for pruning searches. The kd-tree can be constructed with $O(n \log n)$ work and $O(\log n \log\log n)$ span [66, 70]. kd-trees can answer two types of queries efficiently: finding points inside a radius and finding the nearest neighbors of some chosen point. We call the first type ***range query*** and the second ***nearest neighbor query***.

A kd-tree T can be incremental, in which case we can insert points into T. Note that an incremental kd-tree can be imbalanced and not satisfy complexity results of a normal kd-tree. We use BUILD-kD-TREE(P) to represent initializing a kd-tree from the set of points P.

Range query with kd-trees. Let T.QUERY-RANGE(x, r) denote a range search on T, in a spherical region R with radius r centered at a point x, and returns the number of points inside the region. In a range query, when traversing down the kd-tree, we only need to visit a node if its cell intersects with R; otherwise the node can be pruned from the search. A range query takes $O(n^{1-\frac{1}{d}} + |Q|)$ work on a balanced kd-tree with splitting dimension chosen cyclically, where Q is the set of points returned and d is the dimension of the data set [8]. The query takes $O(\log n)$ span by visiting children in parallel.

Nearest neighbor query with kd-trees. We use T.QUERY-NN(x) to represent performing a nearest neighbor search on T for the point x, which returns the closest neighbor of x. To compute the nearest neighbor of a point x, we first traverse down the kd-tree to find the leaf node that contains the point x. Then, in the backtracking process, we search the sibling subtrees. Let x's distance to the current nearest neighbor candidate of x be represented by L. We prune the search of any subtree whose cell is farther than L away from x. Friedman et al. [26] proved that the average-case work complexity of a nearest neighbor search can be bounded by $O(\log n)$ under the assumptions that the density of points in space is locally uniform and the kd-tree is split at the widest dimension's median per level.

Fenwick tree. The 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 $[i - \text{LSB}(i) + 1, i]$, where $\text{LSB}(i)$ represents the least significant bit of integer i and $\sum_{i=0}^{n} |B[i]| = O(n \log n)$ [24]. 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, $\bigcup_{j \in S[i]} B[j] = [1, i]$. Each $S[i]$ can be built iteratively in $O(\log n)$ work, and we can access a partition of the range $[1, i]$ in $O(\log n)$ work using the indices stored in $S[i]$.

Union-find. The union-find data structure maintains the set membership of elements, and allows for merging of these sets. Initially, each element is in its own singleton set. A UNION(a, b) operation merges a and b into the same set. We

use a lock-free concurrent union-find [39], where performing m unions on a union-find data structure with n elements takes $O(m\left(\log(\frac{n}{m}+1)+\alpha(n,n)\right))$ work and $O(\log n)$ span (α denotes the inverse Ackermann function).

Other parallel primitives. WRITE-MIN(loc, val) is a priority concurrent write that takes as input two arguments, where the first argument is the location to write to and the second argument is the value to write; on concurrent writes, the smallest value is written [58]. We assume that WRITE-MIN takes $O(1)$ work and span. RADIX-SORT(A) takes a sequence of elements A of size n, with an ordering key defined for each element. It sorts them in parallel according to the ordering of the elements' keys. Radix sort takes $O(n)$ work and $O(\log n)$ span with high probability (w.h.p.)[4] given that the range of the keys is bounded by $O(n\log^{O(1)} n)$ [51].

4 Priority Search kd-tree-based Dependent Point Finding

We present our first algorithm for solving the dependent point finding task using our new variant of kd-trees.

4.1 Sequential Dependent Point Finding To warm up, we first introduce a sequential incomplete kd-tree based algorithm for finding dependent points. This algorithm is an improvement over Amagata and Hara [3]'s dependent point finding algorithm. Their algorithm uses an incremental kd-tree, which incurs an expensive cost for inserting points.[5] In Amagata and Hara [3]'s algorithm, points are sorted in reverse order of density, and inserted to the tree one by one in order via top-down traversals of the tree. Each point queries its nearest neighbor in the tree, before being inserted into the tree itself. Since points are inserted in reverse order of density, the nearest neighbor of a point x as returned by the incremental kd-tree must have a higher density value than x and be its dependent point [3].

We propose to use an incomplete kd-tree in place of an incremental kd-tree, and take advantage of the fact that we know all the points to insert. Instead of inserting points into the incremental kd-tree, we utilize a lazy insertion strategy: a balanced kd-tree is constructed with all points in P, but all points are marked as inactive initially. We use a boolean variable isActive$_i$ (initialized to *false*) 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 isActive to *true* for all of its ancestors in the tree by a bottom-up traversal from the leaf node containing the inserted point. When traversing the kd-tree to query for the nearest neighbor, we can prune a subtree i if its isActive$_i$ value is *false*. An example of incomplete kd-tree is given in Figure 1.

Our new method has two advantages. First, the incremental kd-tree can be imbalanced and make querying slower while our kd-tree is always balanced since its structure is not modified after construction. This is especially important when we perform queries in parallel, because the span of the query is the same as the depth of the tree. Second, traversing down the kd-tree to insert a point requires computing which child the point belongs to, but our method only needs to follow the parent pointers starting at a leaf node to traverse up the tree without requiring any other computation.

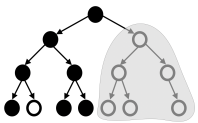


Figure 1: An example of an incomplete kd-tree. A node is unfilled if its subtree does not contain any active point; otherwise it is filled. During a nearest neighbor search, the entire grayed out subtree can be pruned because it contains no active point.

However, this method still has high span, because the points are inserted one by one. The span of each nearest neighbor search is $O(\log n)$, and there are $O(n)$ nearest neighbor queries, so the span is $O(n\log n)$. Amagata and Hara [3]'s algorithm can also be modified to achieve $O(n\log n)$ span by replacing their sequential nearest neighbor queries with parallel nearest neighbor queries. In the rest of the section, we will see that we can use a priority search kd-tree to find the dependent points of all points in parallel.

4.2 Priority Search kd-tree

4.2.1 Priority Search kd-tree Definition To parallelize the dependent point finding routine described in Section 4.1, 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 designed to store a set of points $P=\{x_1, x_2, \ldots, x_i, \ldots, x_n\}$, such that each point $x_i \in \mathbb{R}^d$ is associated with a priority value γ_i. In our case, γ_i is the density. Similar to a normal kd-tree, each node in the priority search kd-tree corresponds to a set of points and a partition of space. Additionally, each node in a priority search kd-tree stores the point with the highest γ value among all points in its point set; this γ value is referred to as the γ value of the node. The remaining 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 shown in Figure 2. A priority search kd-tree is structurally similar to a max kd-tree [31], which records only the maximum priority value at each node. However, in a max kd-tree, the point with the maximum priority value is stored at a leaf in either the left or the right subtree of that node, instead of directly at that node like our priority search kd-tree. 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 because each cell in a max kd-tree is not uniquely associated with a point. We give more details in the full version of our paper.

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

[5] Although Amagata and Hara [3]'s exact DPC algorithm has parallel components, their dependent point finding step is sequential.

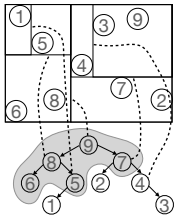


Figure 2: An example of a priority search kd-tree. Each point is labeled with its priority value γ, which is an integer from 1 to 9 in this example. Each node of the priority search kd-tree stores the point with the highest γ within the region of the cell of the node; the number inside the circle of 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 subgraph T_q comprising all nodes with $\gamma > 4$. Because the γ values of a priority kd-tree satisfy the heap property, T_q is always an upper portion of the priority search kd-tree.

Priority search kd-trees can be constructed similarly to a normal kd-tree; the only extra step is finding the point with highest priority value at each node by scanning all of its descendant nodes. The cost of this extra step is subsumed by the cost of splitting at each node. Construction takes $O(n \log n)$ work and $O(\log n \log\log n)$ span. The data structure takes $O(n)$ space like a normal kd-tree, because only $O(1)$ extra information is stored at each node.

4.2.2 Priority Nearest Neighbor Query

Let the ***priority nearest neighbor*** be the nearest neighbor of a point x_i with higher priority. It is easy to see that the priority nearest neighbor query problem is equivalent 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 $\rho(x_i)$. A more formal definition is given below.

DEFINITION 6. *Given a generic query point $x_q \in \mathbb{R}^d$, a distance measure D, and a point set $P \subseteq \mathbb{R}^d$, we define $P_q = \{x_i \mid x_i \in P \text{ and } \lambda_i > \lambda_q\}$. Then, the priority nearest neighbor of x_q is the point in P_q that is closest to x_q as measured by D.*

Priority nearest neighbor queries can be solved by querying a priority search kd-tree following a similar procedure as a normal nearest neighbor query, with the exception that all subtrees with priority value $\leq \lambda_q$ are pruned from the search. Consider a particular query, with a threshold priority value of λ_q. Let T denote a priority search kd-tree. Let $T_q \subseteq T$ represent the set of nodes with priority value $> \gamma_q$. Because of the structure of the priority search kd-tree, T_q must be a connected subgraph of T.[6] A priority nearest neighbor search on T is thus equivalent to a normal 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, a priority nearest neighbor query on a priority search kd-tree takes $O(\log n)$ average-case work with some assumptions, $O(n)$ worst-case work, and $O(\log n)$ worst-case span. Similar assumptions are made in the analysis of the original kd-tree [26]. For average-case analysis, we assume that the points in P_q are sampled from $\mathbb{R}^d$ according to some probability density function μ_q, and that the number of points is sufficiently large such that μ_q can be considered locally uniform. We provide a proof of the bounds in the full version of our paper.

[6] If two tree nodes a and b are both in T_q but not connected, then some ancestor of either a or b is not in T_q and has a priority value less than γ_q. However, this is not possible because the ancestors can only have higher priority values.

Algorithm 1 Parallel dependent point finding with a priority search kd-tree

```
1: procedure PRIORITY-DEPENDENT-POINT(P, ρ)
2:     T ← BUILD-PRIORITY-SEARCH-KD-TREE(P, ρ)
3:     parfor all x_i in P with ρ(x_i) ≥ ρ_min do
4:         λ(x_i) ← T.QUERY-PRIORITY-NN(x_i)
5:     return λ
```

4.3 Parallel Dependent Point Finding with Priority Search kd-tree

We now apply the priority search kd-tree to solve the dependent point finding task. The recipe for finding dependent points using a priority search kd-tree is given in Algorithm 1, where P is the input data set with size n and ρ is an array containing the density of points in P. On Line 2, we construct the priority search kd-tree in $O(n \log n)$ work and $O(\log n \log\log n)$ span. On Lines 3–4, we compute the dependent point for each non-noise point in parallel. Let the work and span of each dependent point search operation be $\mathcal{W}$ and $\mathcal{S}$, respectively, on the priority search kd-tree. The work and span of this step is then $O(n\mathcal{W} + n \log n)$ and $O(\mathcal{S} + \log n \log\log n)$, respectively. In the worst case, $\mathcal{W} = O(n)$ and $\mathcal{S} = O(\log n)$. Under the assumptions stated earlier, $\mathcal{W} = O(\log n)$ in the average case.

THEOREM 4.1. *Constructing a priority search kd-tree and finding all dependent points can be done in $O(n \log n)$ average-case work, $O(n^2)$ worst-case work, and $O(\log n \log\log n)$ worst-case span. The space usage is $O(n)$.*

5 Fenwick Tree-based Parallel Dependent Point Finding

We introduce another parallel algorithm for solving the dependent point finding task, which is based on Fenwick trees. This algorithm has the same overall span as the priority search kd-tree algorithm, but has a higher work complexity. However, the average-case work complexity analysis does not make assumptions about the density of P_q. It only assumes that P's underlying probability density function is locally uniform (the standard assumption for a kd-tree's $O(\log n)$ average-case nearest neighbor query complexity to hold).

This algorithm can be summarized as follows. We first construct an array $\overline{P}$ of points in P sorted by descending order of their density values. Assume that $\overline{P}$ is indexed from 1 to n. Then, we construct a Fenwick tree decomposition of

Algorithm 2 Parallel dependent point finding with a Fenwick tree

```
1: procedure FENWICK-QUERY(B, i, x)
2:     (δ, 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 ← B[j].QUERY-NN(x)
6:         WRITE-MIN((δ, λ'), (dist(x, y), y))
7:     return λ'
8: procedure FENWICK-DEPENDENT-POINT(P, ρ)
9:     P̄ ← RADIX-SORT(P)      ▷ sorting in descending order of density
10:    Initialize B as an array of length n
11:    Initialize λ as an array of length n, with values all being ∞
12:    parfor i = 1 to n do
13:        B[i] ← BUILD-kD-TREE(P̄[i − LSB(i) + 1, i])
14:    parfor x_i in P̄[2 : n] with ρ(x_i) ≥ ρ_min do
15:        λ(x_i) ← FENWICK-QUERY(B, i − 1, x_i)
16:    return λ
```

the range $[1, n]$. $B[i]$ contains the kd-tree that has points in $\overline{P}$ with indices $[i - \text{LSB}(i) + 1, i]$. Recall that $S[i]$ represents a decomposition of the range $[1, i]$ into sub-ranges that are inside B. To perform the dependent point query for the i^{th} point in array $\overline{P}$, we simply need to search through every kd-tree that corresponds to a sub-range in $S[i-1]$. These queries can be done in parallel, thus giving a low span complexity.

We provide the pseudocode for the algorithm in Algorithm 2. The main procedure is FENWICK-DEPENDENT-POINT(P, ρ), which takes as input an array of points P and an array ρ containing the computed densities of the points. On Lines 9–11, we first initialize the $\overline{P}$ array containing points sorted in descending order of their density values, an array B to store the n kd-trees in the algorithm, and an array λ to store the dependent points. On Lines 12–13, we construct the n kd-trees; the i^{th} kd-tree $B[i]$ is constructed from the range of points $\overline{P}[i - \text{LSB}(i) + 1, i]$. Finally, on Lines 14–15, we perform FENWICK-QUERY for non-noise points in parallel to find the dependent point for all points. We do not need to find the dependent point for x_1 as it is the point with highest density.

Now, we will explain procedure FENWICK-QUERY, which takes as input an array of kd-trees B, an index i, and a point x; FENWICK-QUERY performs a nearest neighbor query for point x among the points $x_1, x_2, \ldots, x_i$. On Line 3, we construct a set $S[i]$ for the input index i, which contains the indices of the Fenwick tree sub-ranges that form a partition of $[1, i]$, as described in Section 3.2. Each of these sub-ranges corresponds to a kd-tree; we perform nearest neighbor queries QUERY-NN on all of these kd-trees on Line 5. Let λ' signify the current dependent point of point x and δ the distance between x and λ'. On Line 6, the dependent point with the smallest distance to x (breaking ties using the point ID) is computed using the concurrent WRITE-MIN function. The current λ' is replaced by a newly found nearest neighbor if the newly found nearest neighbor is closer to x than λ' is.

Analysis. We first analyze the complexity of the FENWICK-QUERY subroutine. We show that it takes $O(\log^2 n)$ average-case work and $O(n)$ worst-case work. The construction of S on Line 3 takes $O(\log n)$ work [24]. Each call of QUERY-NN on Line 5 takes $O(\log n)$ average-case work [26], which sums to $O(\log^2 n)$ work across all iterations of the parallel for-loop on Line 4. In the worst case, however, each kd-tree nearest neighbor query takes time linear in the number of points in the kd-tree [26], meaning that Line 5 takes $O(|B[j]|)$ work for the j^{th} kd-tree, T_j. Across 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 has a worst-case span of $O(\log n)$. The construction of $S[i]$ takes $O(\log n)$ span. The WRITE-MIN operation on Line 6 takes $O(1)$ span. The nearest neighbor query on Line 5 takes a worst-case span of $O(\log n)$ since each branch of the kd-tree can be searched in parallel and kd-trees have $O(\log n)$ depth [26]. Since all nearest neighbor queries are executed in parallel, the span for the entire FENWICK-QUERY subroutine is $O(\log n)$.

Now, we examine the main process FENWICK-DEPENDENT-POINT. We show that its average-case work complexity is $O(n \log^2 n)$ and its worst-case work complexity is $O(n^2)$. Line 9 takes $O(n)$ work since the keys of the sort—the ρ values—are bounded in value by $O(n)$. On Lines 12–13, constructing the i^{th} kd-tree takes $O(|B_i| \log |B_i|)$ work. Therefore, constructing all kd-trees takes $O(\sum_{i=1}^{n} |B_i| \log |B_i|) = O(n \log^2 n)$ work. Finally, the FENWICK-QUERY operations performed in the parallel for-loop on Lines 14–15 take $O(n \log^2 n)$ average-case work and $O(n^2)$ worst-case work. Overall, FENWICK-DEPENDENT-POINT takes $O(n \log^2 n)$ average-case work and $O(n^2)$ worst-case work.

Next, we analyze the span of FENWICK-DEPENDENT-POINT. The radix sort on Line 9 takes $O(\log n)$ span w.h.p. [51]. Each BUILD-kD-TREE operation on Line 13 takes span $O(\log n \log \log n)$. Finally, each call to the subroutine FENWICK-QUERY on Line 15 takes $O(\log n)$ worst-case span. 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 $O(\sum_{i=1}^{n} |B_i|) = O(n \log n)$.

THEOREM 5.1. *Constructing a Fenwick tree and finding all dependent points can be done in $O(n \log^2 n)$ average-case work, $O(n^2)$ worst-case work, and $O(\log n \log \log n)$ worst-case span. The space usage is $O(n \log n)$.*

6 Optimization of Other Steps

6.1 Optimizing Density Computation (Step 1) In our full paper, we show that the DPC algorithm's density computation takes $O(\min(O(n(n^{1-\frac{1}{d}} + \varrho_{\text{avg}})), n\rho_{\text{avg}} \log n))$ average-case work in theory. We now discuss an optimization that improves

Algorithm 3 Single-linkage clustering with parallel union-find

```
1: procedure SINGLE-LINKAGE-CLUSTER(P, λ, δ_min, ρ_min)
2:     parfor all x_i in P do
3:         if λ(x_i) ≠ ∞ then δ(x_i) ← dist(x_i, λ(x_i))
4:     Initialize F to be an empty parallel union-find data structure
5:     parfor all x_i in P do
6:         if δ(x_i) < δ_min or ρ(x_i) < ρ_min then        ▷ check if x_i's dependent distance is < threshold
7:             F.UNION(x_i, λ(x_i))
8:     return F.cluster-labels
```

the performance of density computation in practice. Let R denote the spherical region with radius r and centered at x^{center}. Since we only want the count of points in R, we do not have to visit every point. If a cell corresponding to a subtree is contained completely inside R, 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, instead of visiting every point. We can check whether a hyper-rectangular region in coordinate space is contained inside a sphere R by finding the corner of the region x_{far} that is farthest from the center of R and checking if x_{far} is enclosed in R.

6.2 Optimizing Single-Linkage Clustering (Step 3) In this subsection, we optimize Step 3 of DPC. We use a lock-free parallel union-find data structure [39] to solve single-linkage clustering, thus cutting down the $O(n)$ span complexity from Amagata and Hara [3]'s algorithm to $O(\log n)$. Our algorithm is shown in Algorithm 3. It takes an array of points P, an array of their dependent points λ, and the parameters $\delta_{\min}$ and $\rho_{\min}$. On Lines 2–3, we compute the dependent distance of all points in parallel, which takes $O(n)$ work and $O(1)$ span. On Lines 4–7, we use union-find to cluster points with their dependent points if their dependent distance is less than $\delta_{\min}$ or if their density is less than $\rho_{\min}$. The initialization on Line 4 takes $O(n)$ work and $O(1)$ span. On Line 7, performing $O(n)$ unions on a union-find data structure with n elements takes $O(n\alpha(n, n))$ work [39] and $O(\log n)$ span, and this is also the overall work and span.

7 Experiments

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

7.1 Experiment Setup We run experiments on both real-world and synthetic data sets. The real-world data sets that we use are *GeoLife* [74], *PAMAP2* [53], *Sensor* [13, 12], *HT* [36], and *Gowalla* [16, 42]. The synthetic data sets that we use are produced by the *simden* and *varden* random walk based generators by Gan and Tao [27]. *Simden* generates multiple clusters of points with similar density while *varden* produces multiple clusters with varying density. We also use synthetic data sets generated by a *uniform* sampler. In addition, we use a synthetic data set *Query* [1, 4]. Details of these data sets are listed in Table 2 along with the hyper-parameters that we use for each data set. The d_{cut} hyper-parameter 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 data set. The $\rho_{\min}$ and $\delta_{\min}$ values are selected such that the total number of clusters produced by the DPC algorithm is relatively small. We use Euclidean distances in our experiments.

Name	n	d	d_{cut}	$\rho_{\min}$	$\delta_{\min}$
uniform	10^3 to 10^7	2	30	0	100
simden	10^3 to 10^7	2	30	0	100
varden	10^3 to 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
Query	50000	3	0.01	0	0.05
Gowalla	1256248	2	0.03	0	40

Table 2: The real world data sets used in our experiments, along with their sizes (n), their dimensionality (d), and the clustering hyper-parameters that we select for them. The numbers for Query and Gowalla are after de-duplication.

Computational Environment. We use *c2-standard-60* instances on the Google Cloud Platform. These are 30-core machines with two-way hyper-threading with Intel 3.1 GHz Cascade Lake processors that can reach a max turbo clock-speed of 3.8 GHz.

Algorithms. We implement our algorithms using the ParlayLib [9] and ParGeo [66] libraries. We use C++ for all implementations, and the gcc compiler with the -O3 flag to compile the code. We evaluate the following algorithms.

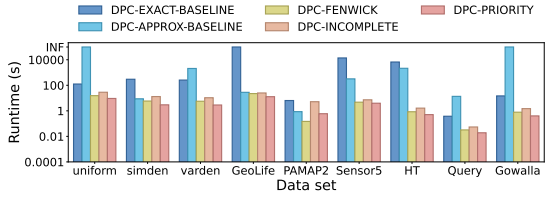
- DPC-EXACT-BASELINE: Amagata and Hara [3]'s state-of-the-art implementation of the partially parallel exact DPC algorithm, which computes the densities in parallel.
- DPC-APPROX-BASELINE: Amagata and Hara [3]'s fastest parallel approximate DPC algorithm.
- DPC-INCOMPLETE: our partially parallel DPC algorithm that uses the incomplete kd-tree based dependent point finding algorithm in Section 4.1 along with the optimizations in Section 6.
- DPC-PRIORITY: our parallel DPC algorithm that uses the priority search kd-tree based dependent point finding algorithm in Section 4.3 along with the optimizations in Section 6.
- DPC-FENWICK: our parallel DPC algorithm that uses the Fenwick tree based dependent point finding algorithm in Section 5 along with the optimizations in Section 6.

We also compare with Rasool et al. [52]'s sequential DPC algorithm. We could not obtain their source code, so we compare with the numbers they reported in their paper on the same data sets and similar machines.

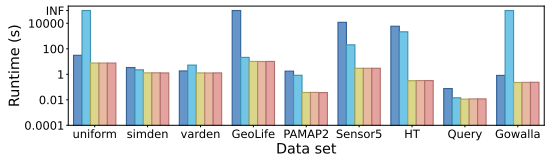
7.2 Runtime Comparison In this subsection, we compare the DPC algorithms' overall performance, and the runtimes of

Algorithm	DPC-EXACT-BASELINE			DPC-APPROX-BASELINE			DPC-FENWICK			DPC-INCOMPLETE			DPC-PRIORITY		
Datasets	density	dep.	total	density	dep.	total	density	dep.	total	density	dep.	total	density	dep.	total
uniform2	30.70	91.30	125.44	–	–	–	7.65	7.07	15.43	**7.58**	18.26	28.50	7.59	**1.69**	**9.30**
simden2	3.39	290.30	296.81	2.23	6.36	8.74	1.29	3.86	5.85	1.31	11.27	13.12	**1.27**	**1.39**	**2.94**
varden2	1.82	250.23	256.28	5.25	2072.96	2078.41	1.28	3.87	5.63	**1.26**	8.93	10.60	1.28	**1.35**	**2.86**
GeoLife	–	–	–	21.08	5.19	28.12	10.20	12.25	22.48	**10.04**	14.95	25.03	10.18	**2.59**	**12.80**
PAMAP2	1.76	4.65	6.41	0.83	**0.026**	0.86	**0.037**	0.11	**0.15**	0.052	5.13	5.18	0.037	0.56	0.59
Sensor	11850.20	2000.41	13852.72	202.95	115.50	318.60	2.97	1.77	4.75	**2.94**	4.33	7.29	2.95	**0.98**	**3.94**
HT	5836.56	814.50	6652.43	2144.31	0.61	2144.93	**0.31**	0.52	0.85	0.46	1.21	1.63	0.32	**0.17**	**0.51**
Query	0.08	0.30	0.38	0.014	13.31	13.58	**0.01**	0.019	0.03	0.01	0.039	0.05	0.01	**0.007**	**0.02**
Gowalla	0.82	13.57	14.72	–	–	–	**0.23**	0.49	0.78	0.23	1.09	1.48	0.24	**0.16**	**0.40**

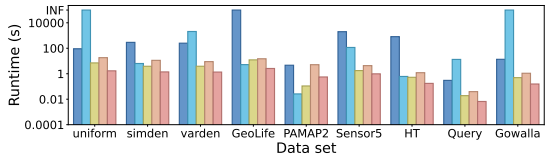
Table 3: The running times (seconds) of the 5 DPC algorithms on real-world and synthetic data sets, decomposed into the density computation step (density) and the dependent point finding step (dep.). "–" means that the algorithm did not terminate within 48 hours.



(a) Total running time of DPC algorithms



(b) Running time of density computation



(c) Running time of dependent point finding

Figure 3: Running times (seconds) of DPC algorithms in log-scale. All algorithms are run on a 30-core machine with hyper-threading. Some algorithms time out and do not terminate within 48 hours. These algorithms have time "INF". Our proposed dependent point finding algorithms and density computation optimizations achieve significant improvement in comparison to other methods. For uniform, simden, and varden, we used $n = 10^7$.

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 in all algorithms. Figure 3 and Table 3 show the runtime comparison across the DPC algorithms.

As shown in Table 3, the portion of time that density computation and dependent point finding take is highly dependent on the data set and d_{cut} parameters we use. For our experiments, d_{cut} is chosen such that the computed density values are nonzero but are much less than n.

Comparison across exact DPC algorithms. From Figure 3, we can see that all of our proposed algorithms consistently outperform DPC-EXACT-BASELINE on all data sets. Only DPC-INCOMPLETE is slightly slower than DPC-EXACT-BASELINE for PAMAP2 in the dependent point finding step. Overall, DPC-PRIORITY is the fastest on almost all data sets, and achieves a 10.8–13169x speedup over DPC-EXACT-BASELINE (Figure 3a).

On a single thread, DPC-PRIORITY also outperforms Rasool et al. [52]'s state-of-the-art sequential exact DPC algorithm. Rasool et al. [52] reported their sequential R-tree based algorithm's running time for Query and Gowalla. To compare with their results, we performed experiments on Query and Gowalla using a machine with the same processor specifications as the one they used, and found that our algorithms are 1.1–1.4x faster.

Our optimized density computation method from Section 6 (which is the same for all three of our algorithms) outperforms DPC-EXACT-BASELINE's density computation step by 1.4–18586.3x, with a geometric mean of 31.5x (Figure 3b). Our method is faster than DPC-EXACT-BASELINE because we do not need to iterate over all points in the range. Moreover, we pre-allocate memory for all nodes in our kd-tree, while the nodes in DPC-EXACT-BASELINE's kd-tree are allocated dynamically, which can lead to more cache misses.

For dependent point finding (Figure 3c), DPC-FENWICK outperforms DPC-EXACT-BASELINE by 12.9–1551.7x, with a geometric mean of 81.9x. DPC-INCOMPLETE achieves a speedup of 0.9–675.9x, with a geometric mean of 23.6x. DPC-PRIORITY attains a speedup of 8.3–4666.3x, with a geometric mean of 168.7x. Our fully parallel algorithms DPC-FENWICK and DPC-PRIORITY are faster than DPC-EXACT-BASELINE mainly because they can find dependent points for all points in parallel. Our DPC-INCOMPLETE algorithm, which inserts points iteratively like DPC-EXACT-BASELINE, is still faster because our kd-tree is more balanced, does not need to insert points, and has a more cache-friendly layout.

Among our new algorithms, DPC-FENWICK and DPC-PRIORITY are faster than DPC-INCOMPLETE because the former two are fully parallel. DPC-PRIORITY is faster than DPC-FENWICK on most data sets, due to its lower average-case work bound, but DPC-FENWICK can sometimes be faster

(e.g., on PAMAP2) depending on the data set distribution.

Comparison with approximate DPC baseline. DPC-PRIORITY is able to achieve running times that are superior to DPC-APPROX-BASELINE on all data sets. DPC-FENWICK and DPC-INCOMPLETE can also achieve competitive results when compared to DPC-APPROX-BASELINE. Across all data sets, our optimized density computation method attains a 1.7–6828.5x speedup over DPC-APPROX-BASELINE; the geometric mean speedup is 17.6x (Figure 3b).

Considering just the dependent point finding step (Figure 3c), DPC-FENWICK outperforms DPC-APPROX-BASELINE by 0.2–536.2x, with a geometric mean of 3.4x; DPC-INCOMPLETE outperforms DPC-APPROX-BASELINE by 0.005–232.2x, with a geometric mean of 1.0x; and DPC-PRIORITY outperforms DPC-APPROX-BASELINE by 0.05–1534.1x, with a geometric mean of 6.7x. The range of speedups varies significantly across data sets primarily because DPC-APPROX-BASELINE's performance is highly dependent on the distribution of points on each data set. DPC-PRIORITY's dependent point finding step is only slower than that of DPC-APPROX-BASELINE on one data set (PAMAP2), and achieves considerable speedup on all others.

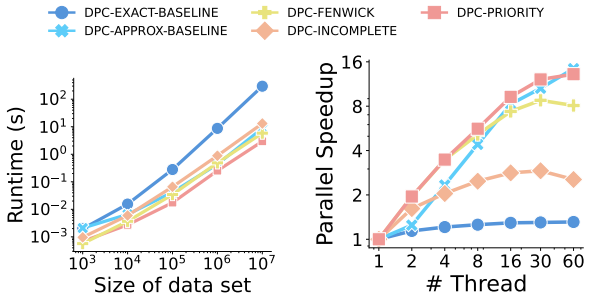
In total (Figure 3a), DPC-PRIORITY achieves a 1.5–4206x speedup against DPC-APPROX-BASELINE, with a geometric mean speedup of 55.4x. DPC-FENWICK achieves a 1.3-2523x speedup over DPC-APPROX-BASELINE, with a geometric mean speedup of 43.7x. DPC-INCOMPLETE obtains a 0.7–1316x speedup against DPC-APPROX-BASELINE, with a geometric mean speedup of 16.8x.

Effect of parameters on running time. $\delta_{\min}$ is only used in Step 3 of DPC. Since Step 3 takes negligible percentage of the overall runtime, using different $\delta_{\min}$ values has little effect on the total runtime. Increasing $\rho_{\min}$ increases the number of noise points, and hence the overall running time decreases because noise points are skipped in Steps 2 and 3. For d_{cut}, the effect on total running time depends on the distribution of data sets. We show the effect of d_{cut} in our full paper. In general, a higher d_{cut} leads to increased running time.

7.3 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. Figure 4a shows the runtime of all DPC algorithms over *simden* datasets of different sizes (from 10^3 points to 10^7 points). Our DPC-PRIORITY outperforms both DPC-EXACT-BASELINE and DPC-APPROX-BASELINE for *simden* data sets of all sizes tested. Furthermore, we see that the running time of our proposed algorithms increases much more slowly than DPC-EXACT-BASELINE as the data size increases. We use a linear fit on the logarithm of runntime and $\log n$ to get the slopes of the lines in Figure 4a. The slope for DPC-EXACT-BASELINE is 1.31, for DPC-APPROX-BASELINE is 0.94, for DPC-FENWICK is 1.02, for DPC-INCOMPLETE is 1.05, and for DPC-PRIORITY is 0.94. This demonstrates that our algorithm has superior scalability across different graph sizes.



(a) Running time (seconds) across data sets of different sizes.

(b) Speedup across different numbers of threads.

Figure 4: Running time of DPC algorithms on *simden* data sets of different sizes and the parallel speedup of the DPC algorithms across different number of threads ("60 threads" means 30 cores with two-way hyper-threading) on the *simden* data set of size 10^7. All axes use logarithmic scale.

Parallel scalability. Finally, we investigate the parallel scalability of our algorithms. Figure 4b shows that all of our proposed DPC algorithms obtain better parallel scallability than DPC-EXACT-BASELINE. DPC-FENWICK is able to achieve a 8.8x self-relative speedup and DPC-PRIORITY 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. DPC-FENWICK and DPC-INCOMPLETE have smaller speedups on 60 hyper-threads than on 30 threads due to the extra overhead of hyper-threading.

8 Conclusion

In this paper, we developed efficient parallel algorithms for density peaks clustering, and proved strong work and span bounds for them. We performed experiments of our algorithms, showing that they outperform previous state-of-the-art DPC algorithms and achieve good parallel scalability on large data sets. For future work, we are interested in exploring distributed versions of DPC.

Acknowledgements. This research is supported by MIT PRIMES, Siebel Scholars program, DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, Google Research Scholar Award, cloud computing credits from Google-MIT, FinTech@CSAIL Initiative, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] Query analytics workloads dataset data set, note = https://archive.ics.uci.edu/ml/datasets/query+analytics+workloads+dataset.

[2] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. page 94–105, 1998.

[3] Daichi Amagata and Takahiro Hara. Fast density-peaks clustering: Multicore-based parallelization approach. In *Proceedings of the International Conference on Management of Data*, page 49–61, 2021.

[4] Christos Anagnostopoulos, Fotis Savva, and Peter Triantafillou. Scalable aggregation predictive analytics. *Applied Intelligence*, 48(9):2546–2567, 2018.

[5] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: Ordering points to identify the clustering structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 49–60, 1999.

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

[7] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9): 509–517, sep 1975.

[8] Jon Louis Bentley and Jerome H Friedman. Data structures for range searching. *ACM Computing Surveys (CSUR)*, 11(4):397–409, 1979.

[9] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. ParlayLib - a toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, page 507–509, 2020.

[10] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

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

[12] Javier Burgués and Santiago Marco. Multivariate estimation of the limit of detection by orthogonal partial least squares in temperature-modulated mox sensors. *Analytica Chimica Acta*, 1019:49–64, 2018.

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

[14] Ricardo Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. Hierarchical density estimates for data clustering, visualization, and outlier detection. *TKDD*, pages 5:1–5:51, 2015.

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

[16] Eunjoon Cho, Seth A Myers, and Jure Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1082–1090, 2011.

[17] David Coe, Mathew Barlow, Laurie Agel, Frank Colby, Christopher Skinner, and Jian-Hua Qian. Clustering analysis of autumn weather regimes in the northeast United States. *Journal of Climate*, 34(18):7587 – 7605, 2021.

[18] G.B. Coleman and H.C. Andrews. Image segmentation by clustering. *Proceedings of the IEEE*, 67(5):773–785, 1979.

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

[20] Hui Du, Yanting Hao, and Zhihe Wang. An improved density peaks clustering algorithm by automatic determination of cluster centres. *Connection Science*, pages 1–17, 12 2021.

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

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

[23] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, page 226–231, 1996.

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

[25] Dimitris Floros, Tiancheng Liu, Nikos Pitsianis, and Xiaobai Sun. Sparse dual of the density peaks algorithm for cluster analysis of high-dimensional data. In *IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–14, 2018.

[26] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 09 1977.

[27] Junhao Gan and Yufei Tao. On the hardness and approximation of Euclidean DBSCAN. *ACM Trans. Database Syst.*, 42(3), jul 2017.

[28] José Carlos García-García and Ricardo García-Ródenas. A methodology for automatic parameter-tuning and center selection in density-peak clustering methods. *Soft Computing*, 25:1543–1561, 2021.

[29] S. Gong and Yanfeng Zhang. EDDPC: An efficient distributed density peaks clustering algorithm. *Journal of Computer Research and Development*, 53:1400–1409, 06 2016.

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

[31] Matthias Groß, Carsten Lojewski, Martin Bertram, and Hans Hagen. Fast implicit kd-trees: Accelerated isosurface ray tracing and maximum intensity projection for large scalar fields. In *Proceedings of the IASTED International Conference on Computer Graphics and Imaging*, pages 67–74, 2007.

[32] 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. Global characterization of T cells in non-small-cell lung cancer by single-cell sequencing. *Nature Medicine*, 24(7):978—985, July 2018.

[33] B. Hanmanthu, R. Rajesh, and Priyanshul Niranjan. Parallel optimal grid-clustering algorithm exploration on MapReduce framework. *International Journal of Computer Applications*, 180:35–39, 05 2018.

[34] Alexander Hinneburg and Daniel A. Keim. An efficient approach to clustering in large multimedia databases with noise. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, page 58–65, 1998.

[35] Erik G. Hoel and Hanan Samet. Data-parallel R-tree algorithms. In *International Conference on Parallel Processing*, volume 3, pages 47–50, 1993.

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

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

[38] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. DBDC: Density based distributed clustering. volume 2992, pages 88–105, 03 2004.

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

[40] Janu Jiang, Xiyu Liu, and Minghe Sun. A density peak clustering algorithm based on the k-nearest shannon entropy and tissue-like p system. *Mathematical Problems in Engineering*, 2019:1–13, 07 2019.

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

[42] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[43] Fengfu Li, Hong Qiao, and Bo Zhang. Discriminatively boosted image clustering with fully convolutional auto-encoders. *Pattern Recognition*, 83:161–173, 2018.

[44] Qiuzhen Lin, Songbai Liu, Ka-Chun Wong, Maoguo Gong, Carlos A. Coello Coello, Jianyong Chen, and Jun Zhang. A clustering-based evolutionary algorithm for many-objective optimization problems. *IEEE Transactions on Evolutionary Computation*, 23(3):391–405, 2019.

[45] Zhuojin Liu, Shufeng Gong, Yuxuan Su, Changyi Wan, Yanfeng Zhang, and Ge Yu. Improving density peaks clustering through GPU acceleration. *Future Generation Computer Systems*, 141:399–413, 2023.

[46] Songrit Maneewongvatana and David M Mount. It's okay to be skinny, if your friends are fat. In *Center for Geometric Computing Workshop on Computational Geometry*, volume 2, pages 1–8, 1999.

[47] Antonio Marco and Roberto Navigli. Clustering and diversifying web search results with graph-based word sense induction. *Computational Linguistics*, 39:709–754, 09 2013.

[48] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

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

[50] Robert J. Peters and Laurent Itti. Beyond bottom-up: Incorporating task-dependent influences into a computational model of spatial attention. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007.

[51] Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989.

[52] Zafaryab Rasool, Rui Zhou, Lu Chen, Chengfei Liu, and Jiajie Xu. Index-based solutions for efficient density peak clustering. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[53] Attila Reiss and Didier Stricker. Introducing a new benchmarked dataset for activity monitoring. In *International Symposium on Wearable Computers*, pages 108–109, 2012.

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

[55] Alex Rodriguez and Alessandro Laio. Clustering by fast search and find of density peaks. *Science*, 344(6191): 1492–1496, 2014.

[56] F James Rohlf. 12 single-link clustering algorithms. *Handbook of Statistics*, 2:267–284, 1982.

[57] Gholamhosein Sheikholeslami, Surojit Chatterjee, and Aidong Zhang. WaveCluster: A wavelet-based clustering approach for spatial data in very large databases. *The VLDB Journal*, 8(3–4):289–304, 02 2000.

[58] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Reducing contention through priority updates. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 152–163, 2013.

[59] Sami Sieranoja and Pasi Fränti. Fast and general density peaks clustering. *Pattern Recognition Letters*, 128:551–558, 2019.

[60] Abir Smiti and Zied Eloudi. Wave DBSCAN: Improving DBSCAN clustering method using fuzzy set theory. In *International Conference on Human System Interactions (HSI)*, pages 380–385, 2013.

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

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

[63] Peng Wang, Bo Xu, Jiaming Xu, Guanhua Tian, Cheng-Lin Liu, and Hongwei Hao. Semantic expansion using word embedding clustering and convolutional neural network for improving short text classification. *Neurocomputing*, 174:806–814, 2016.

[64] Wei Wang, Jiong Yang, and Richard R. Muntz. STING: A statistical information grid approach to spatial data mining. In *Proceedings of the International Conference on Very Large Data Bases*, page 186–195, 1997.

[65] Yi Wang, Qixin Chen, Chongqing Kang, and Qing Xia. Clustering of electricity consumption behavior dynamics toward big data applications. *IEEE Transactions on Smart Grid*, 7(5):2437–2447, 2016.

[66] Yiqiu Wang, Rahul Yesantharao, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. ParGeo: A library for parallel computational geometry. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 88:1–88:19, 2022.

[67] Renzhi Wu, Nilaksh Das, Sanya Chaba, Sakshi Gandhi, Duen Horng Chau, and Xu Chu. A cluster-then-label approach for few-shot learning with application to automatic image data labeling. *ACM Journal of Data and Information Quality (JDIQ)*, 14(3):1–23, 2022.

[68] Xiao Xu, Shifei Ding, Yanru Wang, Lijuan Wang, and Weikuan Jia. A fast density peaks clustering algorithm with sparse search. *Information Sciences*, 554:61–83, 2021.

[69] Miin-Shen Yang, Yu-Jen Hu, Karen Chia-Ren Lin, and Charles Chia-Lee Lin. Segmentation techniques for tissue differentiation in MRI of ophthalmology using fuzzy clustering algorithms. *Magnetic Resonance Imaging*, 20(2):173–179, 2002.

[70] Rahul Yesantharao, Yiqiu Wang, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic kd-trees. *CoRR*, abs/2112.06188, 2021.

[71] Lifeng Yin, Yingfeng Wang, Huayue Chen, and Wu Deng. An improved density peak clustering algorithm for multi-density data. *Sensors*, 22(22):8814, 2022.

[72] Xiaoning Yuan, Hang Yu, Jun Liang, and Bing Xu. A novel density peaks clustering algorithm based on k nearest neighbors with adaptive merging strategy. *International Journal of Machine Learning and Cybernetics*, 12(10):2825–2841, 2021.

[73] Yanfeng Zhang, Shimin Chen, and Ge Yu. Efficient distributed density peaks for clustering large data sets in MapReduce. *IEEE Transactions on Knowledge and Data Engineering*, 28(12):3218–3230, 2016.

[74] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. Learning transportation mode from raw GPS data for geographic applications on the web. In *International Conference on World Wide Web*, pages 247–256, 2008.

[75] 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. 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):1016–1035.e19, 2020.