Efficient Density-peaks Clustering Algorithms on Static and Dynamic Data in Euclidean Space

DAICHI AMAGATA and TAKAHIRO HARA, Osaka University, Japan

2

Clustering multi-dimensional points is a fundamental task in many fields, and density-based clustering supports many applications because it can discover clusters of arbitrary shapes. This article addresses the problem of Density-Peaks Clustering (DPC) in Euclidean space. DPC already has many applications, but its straightforward implementation incurs $O(n^2)$ time, where n is the number of points, thereby does not scale to large datasets. To enable DPC on large datasets, we first propose empirically efficient exact DPC algorithm, Ex-DPC. Although this algorithm is much faster than the straightforward implementation, it still suffers from $O(n^2)$ time theoretically. We hence propose a new exact algorithm, Ex-DPC++, that runs in $o(n^2)$ time. We accelerate their efficiencies by leveraging multi-threading. Moreover, real-world datasets may have arbitrary updates (point insertions and deletions). It is hence important to support efficient cluster updates. To this end, we propose D-DPC for fully dynamic DPC. We conduct extensive experiments using real datasets, and our experimental results demonstrate that our algorithms are efficient and scalable.

CCS Concepts: • **Information systems** → **Clustering**; • **Theory of computation** → **Shared memory algorithms;**

Additional Key Words and Phrases: Density-peaks clustering, parallel algorithms, multi-dimensional points

ACM Reference format:
Daichi Amagata and Takahiro Hara. 2023. Efficient Density-peaks Clustering Algorithms on Static and Dynamic Data in Euclidean Space. *ACM Trans. Knowl. Discov. Data.* 18, 1, Article 2 (August 2023), 27 pages.
https://doi.org/10.1145/3607873

1 INTRODUCTION

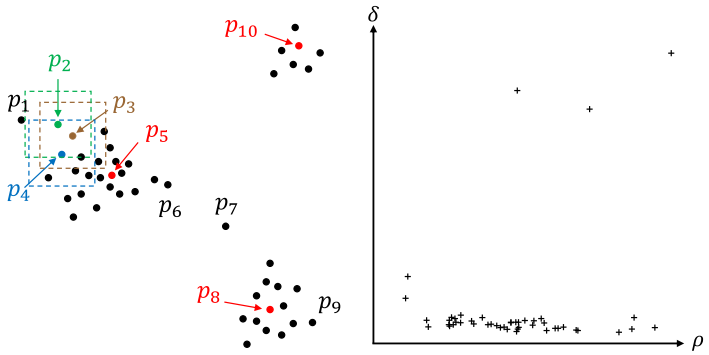
Given a set P of n points in a d-dimensional space, clustering them aims at dividing P into some subsets, i.e., clusters. This multi-dimensional point clustering is a fundamental task for many data mining applications. Density-based clustering particularly supports them well, because it (i) can discover clusters of arbitrary shapes and (ii) does not need the number of clusters as an (initial) input.

This article considers *Density-Peaks Clustering* (DPC) [38]. DPC computes, for each point $p_i \in P$,

- *local density* ρ_i: the number of points p_j inside a specified region centered at p_i and
- δ_i: the distance from p_i to its nearest neighbor point in P with higher local density than ρ_i. (This point is denoted by q_i.)

This work partially supported by AIP Acceleration Research JPMJCR23U2 and JST CREST Grant Number JPMJCR21F2.
Authors' address: D. Amagata and T. Hara, Osaka University, Japan; emails: {amagata.daichi, hara}@ist.osaka-u.ac.jp.

1556-4681/2023/08-ART2
https://doi.org/10.1145/3607873



(a) Illustration of Example 1 (b) Illustration of Example 2

Fig. 1. Illustration of Examples 1 and 2.

Then DPC identifies

- *noise*: points with less local density than ρ_{min}, and
- *cluster centers*: each cluster center p is not a noise and has $\delta \geq \delta_{min}$. (Each cluster center should have a comparatively long distance to its nearest neighbor with higher local density than its one, because its local density is *peak* at its area.)

After that, each of the remaining points is assigned to the same cluster as its q.

Example 1. Figure 1(a) illustrates a set of 2-dimensional points (best viewed in color). In Figure 1(a), each dashed rectangle is a region for computing local density. For example, $\rho_2 = 4$, $\rho_3 = 5$, and $\rho_4 = 7$. Although the nearest neighbor of p_3 is p_2, we have $\rho_2 < \rho_3$, and q_3 (the nearest neighbor of p_3 with higher local density than ρ_3) is p_4. Therefore, δ_3 is the distance between p_3 and p_4. When $\rho_{min} = 3$, p_1, p_6, p_7, and p_9 are regarded as noise. Furthermore, the three red points (i.e., p_5, p_8, and p_{10}) are cluster centers, because δ_5, δ_8, and δ_{10} are much larger than the others.

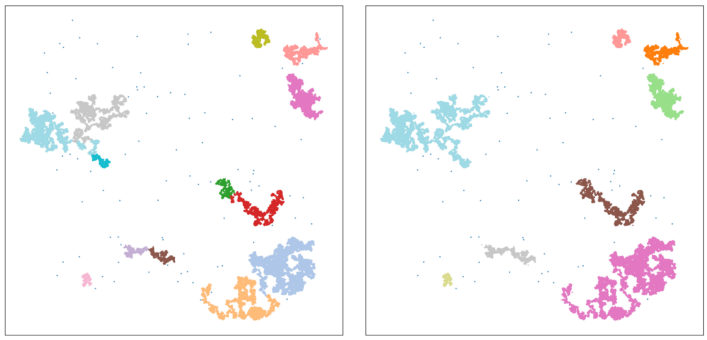
In addition to the inherent advantages of density-based clustering mentioned before, DPC has two advantages. One of them is that, even if users are not domain experts, they can intuitively select cluster centers and noise from a *decision graph*, which visualizes $\langle \rho, \delta \rangle$ into a 2-dimensional space.

Example 2. Figure 1(b) illustrates the decision graph obtained from a set of points in Figure 1(a). Removing noise is an easy task, as we can specify ρ_{min} so points with small local density are ignored. We see that three points have much larger δ than the others, suggesting that these points are *density-peaks*. (The point set in Figure 1(a) clearly has three clusters, so having three density-peaks certainly follows this observation.) Therefore, we can select these points as cluster centers.

Another advantage is that it can divide a dense space into sub-spaces if the space has density-peaks. However, the famous density-based clustering DBSCAN [19] cannot deal with this case well if each cluster has large differences in densities [33].

Example 3. Figure 2 compares the clustering results of DPC (Figure 2(a)) and DBSCAN[1] (Figure 2(b)) on a synthetic dataset [2]. At a glance, DBSCAN functions well, but this dataset has more than eight clusters, i.e., DBSCAN merges some clusters that have independent density-peaks, whereas DPC does not.

[1]We tried many parameter values, and this result is stably obtained. (When ϵ is small, the number of clusters is larger but the clusters are crude.)



(a) DPC (b) DBSCAN

Fig. 2. Difference of the clustering results between DPC and DBSCAN. Each cluster is formed by points with the same color.

This example shows that DPC is robust to clusters having different density distributions. In addition, even if some clusters exist near each other, they are clearly partitioned.

1.1 Motivation

Real-life applications generate datasets with arbitrary shaped clusters that may not be clearly separated [18] (e.g., they may have points existing between close clusters), and DPC deals with them well even if they have such clusters. In addition to this, DPC supports easy selection of cluster centers and noise (see Example 2). DPC, therefore, has been already employed in many fields (e.g., market analysis [14], neuroscience [35], document summarization [51], graphics [30], and computer vision [43]) and data (e.g., entomology, cardiology, and biological audio processing [40]). This fact demonstrates the importance of DPC.

DPC is obviously an important data mining, database, and data science operation for the above applications, and they need to deal with large datasets. This fact requires an efficient DPC algorithm. The main concern of DPC is scalablity to large datasets, because its straightforward implementation incurs $O(n^2)$ time. To alleviate this issue, existing works [8, 34, 37] devised pruning techniques. CFSFDP-A [8] employs the triangle inequality to reduce the distance computations of ρ and q. However, it can fail to prune any points, incurring brute-force accesses in the worst case. FDDP [34] proposes a pruning technique based on space-filling curve to reduce the search space for computing ρ and q. Our empirical results (e.g., Table 6) show that this approach still incurs many unnecessary point accesses. IB-DPC [37] employs a tree-based index to prune unnecessary sub-trees effectively, but it is a heuristic approach having the same drawback as CFSFDP-A. It can be seen that these existing algorithms improve only *practical* running time, and they do not solve the $O(n^2)$ time issue.

Furthermore, supporting cluster updates is also important, because real-world datasets are subjective to updates, e.g., insertions of new data and removal of (nearly) duplicated data [15, 22]. Some works [25, 40] considered dynamic DPC and reported the effectiveness of DPC on dynamic data. However, they do not consider deletions [25] or do not consider efficiency [40]. DPC on fully dynamic data is also worth being addressed to cover applications that need to deal with both point insertions and deletions.

1.2 Contribution

Motivated by the above facts, for both static and dynamic data, we devise efficient DPC algorithms with parallelizability that are optimized for Euclidean space. We consider Euclidean space due to

Table 1. Time and Space Complexities of Each Exact Algorithm for Static Data (a Single Thread Case)

Algorithm	Time	Space	Reference
Scan	$O(n^2)$	$O(n)$	[38]
CFSFDP-A	$\Omega(n^2)$	$O(n)$	[8]
FDDP	$O(n^2)$	$O(n^2)$	[34]
IB-DPC	$O(n^2)$	$O(n)$	[37]
Ex-DPC	$O(n^2)$	$O(n)$	This article
Ex-DPC++	$o(n^2)$	$O(n)$	This article

the fact that it is one of the most commonly used distance functions. We summarize, in Table 1, the worst time/space complexities of existing and our algorithms for static data. Our main contributions are as follows:

(1) We first propose Ex-DPC, an exact algorithm for static data that exploits a kd-tree [9]. Although its worst time is still $O(n^2)$, it employs efficient pruning approaches to the computations of p and q, so the number of distance computations to obtain them for a point is much less than n in practice.
(2) We propose Ex-DPC++, an improved version of Ex-DPC. Different from the existing algorithms [8, 34, 37] (and Ex-DPC), Ex-DPC++ uses two types of tree indices (i.e., kd-tree and cover-tree) to exploit their theoretical search time bound. In addition, Ex-DPC++ incorporates a dataset partitioning technique based on local density. These approaches derive the main advantage of Ex-DPC++: It runs in time sub-quadratic to n (i.e., $o(n^2)$ time). This is a novel result (recall Table 1, i.e., this is the first algorithm that reduces the time complexity of exact DPC (under the Euclidean space constraint).
(3) To efficiently support cluster updates in the fully dynamic data model, we find when we need to update q of a given point and the search space if we need to update it. This has not been investigated so far. Given a single update, we show that $O(\rho_{avg} n)$ amortized time is required to update q of each point, where ρ_{avg} is the average local density. Our algorithm, D-DPC, exploits this theoretical finding to minimize the update cost. An empirically and theoretically efficient exact DPC algorithm on dynamic data has not been known, and DPC is the first algorithm that achieves this.
(4) Our experiments on real datasets show that (i) Ex-DPC++ is the fastest among all evaluated algorithms in all tests, and (iii) D-DPC significantly outperforms the state-of-the-art dynamic DPC algorithms.

Comparison with our conference version. This article is an extended version of our conference paper [2]. Sections 5, 6, and 7 are new contents.

- Although Reference [2] introduces that Ex-DPC runs in $o(n^2)$ time, it holds under some assumptions [3]. Unfortunately, these assumptions do not necessarily hold, so this article removes them. Section 5 shows that our new algorithm, Ex-DPC++, yields $o(n^2)$ time without any assumptions, which has not been achieved in the existing works.[2] Note that this article focuses on exact algorithms and omits the approximation algorithm in Reference [2] (for conciseness), because (i) Ex-DPC++ is faster than it, and (ii) the approximation algorithm needs $O(n^2)$ time in the worst case. (To our knowledge, no $o(n^2)$ time approximation algorithms with error guarantee are known so far.)

[2]The analysis in Reference [37] is wrong, because R-trees do not have any theoretical properties [36].

- Moreover, Reference [2] did not consider fully dynamic data, whereas this article shows how to efficiently deal with data updates in Section 6.
- Section 7 also provides new results: (i) We compare DPC with state-of-the-art density-based clustering w.r.t. clustering effectiveness (Section 7.1), and (ii) we show the empirical efficiencies of Ex-DPC++ and D-DPC (Sections 7.2 and 7.3).

1.3 Organization

The rest of this article is organized as follows: Section 2 introduces preliminary information, and we review related work in Section 3. Sections 4 and 5 present Ex-DPC and Ex-DPC++, respectively. We propose our dynamic algorithm D-DPC in Section 6. Section 7 reports our experimental results, and Section 8 concludes this article.

2 PRELIMINARY

2.1 Problem Definition

Let P be a set of n points in a d-dimensional space $\mathbb{R}^d$. We assume that d is small (e.g., $d < 10$), as with related works, e.g., References [21, 22, 39]. This is natural for density-based clustering, because density suffers from the curse of dimensionality. Section 5 assumes that P is static, whereas Section 6 assumes that P is fully dynamic, i.e., P can have point insertions and deletions.

Density-Peaks Clustering (DPC) aims at dividing P into some subsets based on density-peaks. To this end, DPC requires two important metrics, *local density* ρ and *distance to nearest neighbor with higher local density* δ for each $p \in P$. (Recall that Example 1 introduces their example.)

Definition 1 (Local Density). Let $R_i \subseteq P$ be a set of points inside the axis-aligned rectangle centered at $p_i \in P$, where each side length is d_{cut}.[3] The local density ρ_i of p_i is $|R_i|$.

Definition 2 (q_i). Given a point $p_i \in P$, q_i satisfies:

$$q_i = \underset{p_j \in P : \rho_i < \rho_j}{\arg\min} \; dist(p_i, p_j), \tag{1}$$

where $dist(p_i, p_j)$ is the Euclidean distance between p_i and p_j.

Definition 3 (δ_i). Given a point p_i, $\delta_i = dist(p_i, q_i)$.

Assume that p_j has the highest local density in P, then it is trivial that p_j cannot have q_j. We hence set $\delta_j = \infty$. Next, we define *noise* and *cluster center*:

Definition 4 (Noise). If a point $p_i \in P$ has $\rho_i < \rho_{min}$, then p_i is a noise.

Definition 5 (Cluster Center). If a non-noise point $p_i \in P$ has $\delta_i \geq \delta_{min}$, where δ_{min} is a user-specified threshold, then p_i is a cluster center.

We can specify ρ_{min} and δ_{min} at the same time when d_{cut} is specified or after a decision graph is viewed. Note that ρ_{min} is specified to remove points with (very) small local densities (e.g., $\rho_{min} = 10$) like DBSCAN (the noise concept follows distance-based outliers [5, 6]). Moreover, δ_{min} is specified so each point p_i with much longer δ_i than the other points (like the ones in Figure 1(b)) is selected as a cluster center.

Once a cluster center, say, p_i, is identified, we set $q_i = p_i$. There are points p_j such that $q_j = p_i$. Also, there are points p_k such that $q_k = p_j$. We hence say that p_k (and also p_i and p_j) is (are) *reachable* from the cluster center p_i. Based on this, we define:

[3] Actually, applications can specify any value as side length of each dimension. For ease of presentation, we assume that d_{cut} is specified for every dimension. (Even if applications prefer to use a radius or hyper-sphere to define local density, R_i is still easy to use, because we can consider R_i as the minimum bounding rectangle of the hyper-sphere).

Table 2. Summary of Notation

Notation	Meaning
p	Point
P	Point set
n	Number of points (i.e., $\|P\|$)
d	Dimensionality of points
$dist(p, p')$	Distance between p and p'
ρ	Local density of p
q	Nearest neighbor of p with higher local density than ρ
δ	$dist(p, q)$
$\mathcal{K}$	A kd-tree
C	A cover-tree

Definition 6 (Cluster). Given P, a cluster C, whose cluster center is p_i, is a non-empty subset of P such that non-noise points $p \in C$ are reachable from p_i.

Each point $p \in P$ has a single q, so p belongs to a single cluster and DPC provides a unique set of clusters. Table 2 summarized the notations frequently used in this article.

This article assumes a single machine with a multicore CPU (or with multicore CPUs). The other parallel computation environments are not the scope of this article. The objective of this article is to devise practically and theoretically fast DPC algorithms that return clusters exactly following Definition 6 and are *easily* parallelizable, e.g., their practical running time can be reduced without operations incurring delays, such as `atomic` and `sync`. Table 1 shows that the existing works suffer from $O(n^2)$ time. As a first attempt, we propose a practically fast algorithm in Section 4. Because this algorithm also needs $O(n^2)$ time in the worst case, we overcome this issue in Section 5 and achieve $o(n^2)$ time.

2.2 Tree-based Indices

We here introduce two tree-based indices, kd-tree [9] and cover-tree [10], because we use them in our algorithms. This section introduces their theoretical time complexities for building, range search, and nearest neighbor search.

2.2.1 kd-Tree. This is a multi-dimensional binary-tree structure that needs $O(n)$ space and enables efficient range search in Euclidean space. This index can be built in $O(n \log n)$ time [9], and a point insertion into (deletion from) a balanced kd-tree needs $O(\log n)$ time. Consider an orthogonal (or a rectangular) range *reporting* query that outputs all points in the axis-aligned query rectangle. This query is efficiently processed on a kd-tree, and its time complexity is $O(n^{1-\frac{1}{d}} + \text{OUT})$ under d-dimensional Euclidean space, where OUT is the number of reported points. Also, kd-tree runs a range *counting* query, which outputs the number of points in the query rectangle, in $O(n^{1-\frac{1}{d}})$ time. Unfortunately, a kd-tree needs $O(n)$ time for a nearest neighbor search in the worst case.

2.2.2 Cover-tree. This data structure is designed for accelerating nearest neighbor search in metric spaces. A cover-tree is built in $O(c^6 n \log n)$ time and needs $O(n)$ space, where $c \geq 2$ is a data-dependent *constant* [10, 31]. The cover-tree supports an $O(c^6 \log n)$ time nearest neighbor search [31], which is from the fact that the maximum number of children in any node is $O(c^4)$ and the height of a cover-tree is $O(c^2 \log n)$.[4] (Cover-trees do not have the worst time for range searches.) Assuming that $c = O(1)$, we hereinafter omit the constant c in our analysis.

[4]In practice, the maximum number of children and the height are small. For example, in our experiments, they are, respectively, at most 8 and 66, suggesting that c is sufficiently small.

3 RELATED WORK

3.1 Existing Density-based Clustering

3.1.1 Static Data. In Section 7.1, we compare DPC with state-of-the-art density-based clustering algorithms to confirm the advantages of DPC. We below introduce each of them.

In DBSCAN [19], each point $p \in P$ is evaluated whether it is a *core point* or not, based on two input parameters ϵ and $minPts$. If at least $minPts$ points exist within ϵ from p, then p is a core point. DBSCAN assumes that, if the distance between two core points is within ϵ, then there is a connection between them. Informally, DBSCAN forms a cluster by connecting core points in the above way. We do not say that DPC can replace DBSCAN, because an appropriate clustering algorithm for a given dataset is dependent on the data distribution. However, DPC is more effective for datasets that have skewed density and points existing between close clusters. This is because DBSCAN may consider multiple dense point groups as a single cluster if there are points existing in the border spaces between different groups, whereas DPC is robust to such a data distribution, as shown in Figure 2. This is the main difference between DPC and DBSCAN.[5]

The clustering results of DBSCAN are dependent on ϵ and $minPts$. To suggest meaningful values of ϵ, OPTICS [7] was devised. OPTICS visualizes possible forms of DBSCAN-based clusters at any ϵ. Similarly, HDBSCAN* [12] also overcomes the concern of specifying ϵ. For a fixed $minPts$, it yields a clustering hierarchy consisting of all possible clusters derived from DBSCAN*, a variant of DBSCAN, that does not allow border points (see Reference [12] for details). Although OPTICS and HDBSCAN* help specify ϵ for DBSCAN(*), they do not remove the above difference between DPC and DBSCAN.

DENCLUE [27, 28] has a similar policy to DPC w.r.t. forming clusters. DENCLUE uses a kernel density estimate (e.g., Gaussian kernel) to measure the density of a given coordinate (e.g., a point). To form a cluster, it employs *hill climbing*, i.e., it computes the density gradient and tries to find the coordinate (or a point) with a local maximum w.r.t. the density. The points sharing the same local maximum belong to the same cluster in DENCLUE. This density-gradient-based cluster forming is similar to DPC, as DPC uses the nearest neighbor with higher local density to catch density gradient. Different from DPC, DENCLUE cannot explicitly control the number of clusters, because DENCLUE tries to find *all* local maxima even when some of them are negligible. Such local maxima can be ignored via the hyper-parameter setting of DENCLUE, but DENCLUE does not have a guide such as a decision graph.

3.1.2 Dynamic Data. Most existing density-based clustering algorithms for dynamic data [13, 17, 22, 26, 32, 41] are based on DBSCAN. (Each of them adapts a minor change against the original DBSCAN definition, and some of them are summarized in a nice survey [52].) Therefore, they inherit the difference between DPC and DBSCAN introduced in Section 3.1.1. In this article, we do not consider them, because (i) some of them consider the insertion-only case, and (ii) literature [25] shows that the state-of-the-art dynamic DPC algorithm (which is introduced in Section 3.2.2) outperforms such DBSCAN-based algorithms.

3.2 Existing DPC Works

This section reviews existing DPC works. Note that some variants of DPC, e.g., References [16, 29, 42, 47], have also been proposed, but this article follows the original concept in Reference [38].

3.2.1 Static Data. We introduce state-of-the-art exact DPC algorithms (i.e., their outputs follow Definition 6), as we use them as baselines in Section 7.2. CFSFDP-A [8] selects *pivot points* and

[5] A further discussion of their difference can be found in Reference [25]. Existing works cited in References [8, 16, 25, 40, 49] also compared DPC and DBSCAN w.r.t. clustering qualities. Interested readers may refer to them.

utilizes triangle inequality to avoid unnecessary distance computation. These pivots are obtained by k-means clustering. Unfortunately, k-means clustering is sensitive to noise, so its pivot selection does not provide good filtering power, meaning that the candidate size is still large.

FDDP [34] is a recently proposed exact DPC algorithm. FDDP was also originally proposed for distributed computing environments, but its approaches to computing ρ and q are still available on multicore CPUs. FDDP utilizes z-ordering to limit the search spaces of computing ρ and q and to partition the dataset equally so each partition can contain similar points. This partitioning approach, however, does not consider density, suffering from bad load balancing. In Reference [34], the authors argue that FDDP runs in $O(n \log n)$ time. However, they assume that, for each point p, the number of points accessed to compute ρ and δ is a constant $\ll n$. This is impossible, because ρ is not a constant.

Similar to our algorithms, Index-based DPC (IB-DPC) [37] employs a tree-based index (i.e., an R-tree) to efficiently compute ρ and δ of each point. It prunes unnecessary sub-trees that are not necessary to access for exactly computing ρ and δ. However, unlike our algorithms, IB-DPC uses the index in a heuristic manner and fails to bound the worst case time complexity.

3.2.2 Dynamic Data. Although the effectiveness of dynamic DPC has been confirmed [25, 40], efficient exact algorithms for fully dynamic DPC have not been addressed. The first dynamic DPC algorithm, oDP, was proposed in Reference [40]. Given a point insertion/removal, it updates local density of each point by a linear scan. Unfortunately, oDP does not provide how to update q.

EDMStream, an approximation algorithm for the DPC problem with the decay model (i.e., an insertion-only environment) was proposed in Reference [25]. This algorithm uses a set of sample points to form clusters. For a new point, if there is a sample point that is sufficiently close, then it is *absorbed* in the sample point. EDMStream runs a linear scan to update the local density of each sampled point. Although EDMStream utilizes local densities to avoid unnecessary computation for updating q,[6] a linear scan is still used to update q of a given sample point p when it is not pruned. (Reference [1] also proposed a dynamic DPC algorithm in high-dimensional metric space, and, as this is a different assumption from that of this article, we do not consider this algorithm.)

These works are not efficient, since they always rely on linear scans to update ρ and q. In addition, how to deal with point deletions has not been considered. We overcome these drawbacks in Section 6. In Section 7.3, we use EDMStream and a variant of oDP as baselines and show that our dynamic DPC algorithm outperforms them.

4 EX-DPC

Ex-DPC, our first exact DPC algorithm, assumes that a point set P is indexed by an in-memory kd-tree $\mathcal{K}$. Its detail is summarized in Algorithm 1. To start with, we present Ex-DPC with a single thread. As Definitions 1 and 2 suggest, (i) ρ computation and q computation require different approaches, and (ii) q is dependent on local density (i.e., it can be obtained after local densities are computed). We present how Ex-DPC computes ρ and q of each point in Sections 4.1 and 4.2, respectively. We then explain how to parallelize Ex-DPC.

4.1 Computing ρ

Definition 1 suggests that computing the local density of a point $p_i \in P$ corresponds to doing an orthogonal range search. We here employ a search-based technique, similar to the state-of-the-art algorithms [8, 34, 49]. Specifically, Ex-DPC runs an orthogonal range search on $\mathcal{K}$ to obtain ρ_i, for

[6] Actually, the definition of local density in EDMStream is different from the original one (Definition 1). EDMStream assumes that, given a new point, the local density of only a single sample point needs to be updated, and its pruning relies on this assumption. However, this does not hold in our problem.

ALGORITHM 1: Ex-DPC

Input: P (a point set), d_{cut}, $\mathcal{K}$ (a kd-tree)

```
/* Computing ρ */
foreach p_i ∈ P do
    ρ_i ← RANGE-SEARCH(p_i, d_cut, K)        ▷ orthogonal range search on kd-tree K with query point p_i
/* Computing q */
K ← ∅                                        ▷ destroy K
Sort P in descending order of local density
foreach p_i ∈ P (assume ρ_i > ρ_{i+1}) do
    q_i ← NN-SEARCH(p_i, K)                  ▷ nearest neighbor search on K with query point p_i
    Insert p_i into K                        ▷ incremental kd-tree update
```

each $p_i \in P$. Recall Section 2.2, and a kd-tree supports an efficient orthogonal range search with $O(n^{1-\frac{1}{d}} + \text{OUT})$ time, where OUT is the number of reported points. For p_i, its search range is R_i, and we have $\text{OUT} = |R_i| = \rho_i$.

4.2 Computing q

Recall the constraint of q: Given $p_i \in P$, we have to retrieve q_i from a set of points with higher local densities than ρ_i. Since the local density depends on d_{cut}, it is difficult to build a data structure for efficiently obtaining q of every point $p \in P$ in a pre-processing phase. Although the kd-tree supports practically efficient nearest neighbor search, it is not guaranteed that the nearest neighbor point of p_i has higher local density than ρ_i. That is, simply running a nearest neighbor search on a kd-tree cannot solve the DPC problem correctly. Hence, it is challenging to compute q efficiently.

We overcome this challenge with an idea: *For p_i, we can build a kd-tree that contains only points p_j having $\rho_i < \rho_j$ incrementally.* Our approach is as follows:

(1) Destroy $\mathcal{K}$ (i.e., $\mathcal{K}$ becomes an empty set).
(2) Sort P in descending order of local density.
(3) Pick the first point in P, say, p_1, set $\delta_1 = \infty$, and insert p into $\mathcal{K}$. Set $i = 2$.
(4) Pick the ith point in P, say p_i, run a nearest neighbor search with query point p_i on $\mathcal{K}$, set the result as q_i, and insert p_i into $\mathcal{K}$.
(5) Increment i by one, and repeat the above operation.

Notice that, for the ith point p_i in P, the kd-tree contains only points with higher local densities than ρ_i. (We assume that all points have different local densities, which is practically possible by adding a random value $\in (0, 1)$ to ρ_i, as done in Reference [38].) Therefore, for p_i, its nearest neighbor search retrieves q_i correctly.

4.3 Analysis of Ex-DPC

4.3.1 Space Complexity. Since Ex-DPC uses only a single kd-tree as an index, Ex-DPC needs $O(n)$ space.

4.3.2 Time Complexity. From the analysis in Section 2.2, Ex-DPC needs $O(n(n^{1-\frac{1}{d}} + \rho_{avg}))$ time to compute the local density of every point in P, where ρ_{avg} is the average local density. However, Ex-DPC needs $O(n^2)$ time to compute q of every point in P, since it needs a one-time sort, n times point insertions into $\mathcal{K}$, and n times nearest neighbor searches on $\mathcal{K}$, each of which, respectively, needs $O(n \log n)$, $O(n^2)$, and $O(n^2)$ time in the worst case. The worst time of Ex-DPC is hence $O(n^2)$ in total (although its practical performance is much better than its theoretical time).

ALGORITHM 2: Ex-DPC++

Input: P (a point set), d_{cut}, $\mathcal{K}$ (a kd-tree)

/* Computing ρ */
foreach $p_i \in P$ **do**
  $\rho_i \leftarrow$ Range-Counting$(p_i, d_{cut}, \mathcal{K})$ ▷ orthogonal range "counting" on kd-tree $\mathcal{K}$ with query point p_i
/* Computing q */
Sort P in ascending order of local density
Divide P into disjoint subsets $P_1, P_2, \ldots, P_s$, where $s = \sqrt{\frac{n}{\log n}}$ ▷ dataset partitioning based on local density
foreach $i \in [1, s]$ **do**
  Build a cover-tree C_i of P_i ▷ prepare the data structure employed in Lemma 1
foreach $p_i \in P$ **do**
  foreach $j \in [1, s]$ **do**
    case $\min_{p \in P_j} \rho > \rho_i$ **do**
      $q_i \leftarrow$ NN-Search(p_i, C_j) ▷ nearest neighbor search on C_j with query point p_i, see case (i) in Section 5.2
      Update q_i and δ_i
    case $(\min_{p \in P_j} \rho < \rho_i) \wedge (\max_{p \in P_i} \rho < \rho_i)$ **do**
      Scan P_j and update q_i and δ_i ▷ linear scan P_j to find nearest neighbor of p_i with higher local density than ρ_i, see case (ii) in Section 5.2

4.4 Parallelization

We can parallelize the local density computation in Ex-DPC, but, unfortunately, it cannot compute q_i and q_j independently, i.e., line 7 of Algorithm 1 cannot be parallelized. This is derived from the fact that Ex-DPC needs to compute q of each point p *one by one*, since the kd-tree is incrementally updated. Hence, we focus on parallelizing its local density computation.

Given P and multiple threads, we parallelize local density computation by assigning each point in P to one of the threads. Then, each thread independently runs a range search for each assigned point. To exploit the parallel processing environment (i.e., to hold a balanced load), each thread should have (almost) the same processing cost. Recall that the range search cost of p_i is $O(n^{1-1/d} + \rho_i)$, indicating that the cost depends on its local density, which cannot be pre-known and differs among points. We therefore employ a heuristic that assigns a point to a thread *dynamically*. Specifically, for each thread, Ex-DPC assigns a point, and when a thread has finished its range search, Ex-DPC assigns another point to the thread. We use OpenMP for multi-threading, and to implement the above approach, "#pragma omp parallel for schedule (dynamic)" is used.

5 EX-DPC++

Ex-DPC has two issues: One is its $O(n^2)$ running time, and the other is the partial parallelizablity. This section proposes Ex-DPC++ that overcomes these concerns. Ex-DPC++ takes different approaches to computing ρ and q from those of Ex-DPC to avoid $O(n^2)$ time, and Algorithm 2 details Ex-DPC++. In the following, we present how to employ the approaches.

5.1 Computing ρ

The main concern of local density computation in Ex-DPC is that its cost is proportional to $n \cdot \rho_{avg}$, where ρ_{avg} is the average local density. Ex-DPC++ removes this drawback by employing *range counting* on a kd-tree $\mathcal{K}$. That is, for each $p \in P$, Ex-DPC++ computes ρ via range counting on $\mathcal{K}$,

which needs $O(n^{1-\frac{1}{d}})$ time (see Section 2.2). Removing the factor of ρ_{avg} held by Ex-DPC yields a significant speedup in theory and practice.

Corollary 1. *Ex-DPC++ needs $O(n^{2-\frac{1}{d}})$ time to compute the local density of every point in P.*

5.2 Computing q

Different from Ex-DPC, Ex-DPC++ exploits cover-trees.[7] A cover-tree can be built in $O(n \log n)$ time and running a nearest neighbor search on it needs only $O(\log n)$ time. However, simply using a cover-tree does not achieve $o(n^2)$ time and almost full parallelizablity. We overcome this and obtain such a non-trivial result by combining the theoretical properties of cover-tree and a dataset partitioning approach.

After Ex-DPC++ computes the local density for each point in P, it sorts P and partitions P into equally sized s subsets, $P_1, \ldots, P_s$, based on this order. Note that, for i and j such that $i < j$, we have $\rho \geq \rho'$ for any $p \in P_i$ and $p' \in P_j$. Then, each point $p \in P$ has three cases (i)–(iii), as elaborated below. Given a point $p_i \in P$ and a subset P_j, we have the following cases:

(i) All points in P_j have higher local density than ρ_i. In this case, we run a nearest neighbor search on the cover-tree of P_j.
(ii) P_j has not only points with higher local density than ρ_i but also points with less local density than ρ_i. In this case, we run a linear scan of P_j and obtain the nearest neighbor point with higher local density than ρ_i in P_j. (The points in P_j are sorted by local density, so we stop the scan whenever we access a point with less local density than ρ_i.) It is important to note that there is at most one subset that has this case for p_i.
(iii) The local densities of all points in P_j are less than (or equal to) ρ_i. In this case, p_i ignores P_j.

We obtain q_i for $p_i \in P$ by evaluating each subset P_j based on the above approach.

In the worst case, for a point p, we need to run a nearest neighbor search $s-1$ times and run a linear scan once. The cost of running $s-1$ times nearest neighbor searches is $O((s-1)\log \frac{n}{s})$, whereas that of running a linear scan once is $O(\frac{n}{s})$. We consider the following equation to easily understand the bottleneck:

$$O\left(\frac{n}{s}\right) = O\left((s-1)\log \frac{n}{s}\right). \tag{2}$$

It is important to notice that, in the above equation, s cannot be $O(1)$ and $O(n)$, i.e., s would be $o(n)$. We approximately solve this equation by ignoring all constants and the $\log s$ factor. Then, we have

$$s \approx \sqrt{\frac{n}{\log n}}. \tag{3}$$

Thanks to this observation, we have the following result:

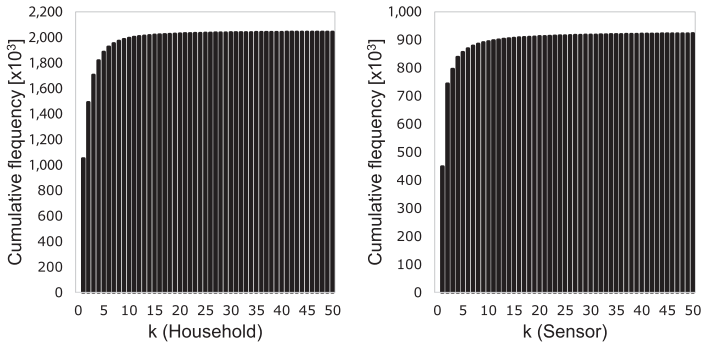
Lemma 1. *Ex-DPC++ needs $O(n^{1.5}\sqrt{\log n})$ time to compute q of every point $p \in P$.*

Proof. Sorting P needs $O(n \log n)$ time, whereas building s cover-trees needs $O(n \log n)$ time in total. By substituting the Formula (3) into Equation (2), we see that computing q needs $O(\sqrt{n \log n})$ time. This is done for n points, thereby this lemma holds. □

5.3 Parallelization

Parallelizing Ex-DPC++ is straightforward, as it evaluates ρ and q of each $p \in P$ *independently*. Specifically, we can parallelize "for loop" of lines 2, 7, and 9. Now it is clear that Ex-DPC++ has two important merits: (i) space and computational efficiencies and (ii) parallelizability.

[7] Ex-DPC++ employs the improved cover-trees proposed in Reference [31].



(a) Household (b) Sensor

Fig. 3. Cumulative frequency with which q exists in the k nearest neighbors of p.

5.4 Time and Space Complexities

From Corollary 1 and Lemma 1, we have the main result of this section.

Theorem 1. *The time complexity of Ex-DPC++ is* $O(n^{2-\frac{1}{d}} + n^{1.5}\sqrt{\log n})$.

We see that the above time is sub-quadratic to n, i.e., $o(n^2)$, and better than that of Ex-DPC and the other existing algorithms. From the space complexity analysis of Ex-DPC and cover-tree, the following is true:

Theorem 2. *The space complexity of Ex-DPC++ is* $O(n)$.

5.5 Optimization

Although Ex-DPC++ avoids $O(n)$ time to compute q for each $p \in P$, this approach needs to run at most $s-1$ nearest neighbor searches and a linear scan of a subset of P. This is slow in practice, thus a practically faster way of computing q is desirable. To this end, we come up with the following heuristic: In real datasets with variations in density, for p, q tends to exist in the k nearest neighbors of p. Figure 3 describes the cumulative frequency with which q of a given $p \in P$ exists in its kth nearest neighbors in two real datasets (that we used in our experiments). This figure demonstrates the above-mentioned observation and shows that a small k is sufficient (Household and Sensor, respectively, have about 2 million and 0.9 million points). Therefore, we incorporate this observation into Ex-DPC++.

Notice that the k nearest neighbors of p can be obtained in a pre-processing phase, as they are not dependent on any DPC parameters. Hence, we compute them offline and maintain the k nearest neighbors in order for each point $\in P$. (A set of these k nearest neighbors is thus an input of Algorithm 2.) When computing q for $p \in P$, the optimized Ex-DPC++ first accesses its k nearest neighbors. Ex-DPC++ accesses them from the nearest to the kth one. Assume that Ex-DPC++ now accesses the ith nearest neighbor of p, and let p^i be this ith nearest neighbor. If ρ is less than the local density of p^i, then it is trivial that $q = p^i$ from Definition 2, thereby Ex-DPC++ can skip evaluating q_i on the cover-trees. Notice that accessing k nearest neighbors needs $O(k) = O(1)$ time (for a constant k). This simple yet effective technique improves the practical time of Ex-DPC++ without losing its theoretical result.

6 D-DPC

This section assumes that P is subjective to updates (i.e., point insertions and deletions). In this case, given an update, the local density of each point is also updated, and this involves updating

q of $p \in P$. It is important to notice that, as long as we monitor q of each point p, cluster labels can be computed *when required*. Labeling when required is a common assumption in clustering literatures [22, 26, 50]. Therefore, we focus on how to monitor ρ and q of each point $p \in P$ and propose D-DPC. We assume that, before we have dataset updates, ρ and q of each point $p \in P$ are initialized by an exact DPC algorithm. D-DPC maintains ρ and q of each $p \in P$ *exactly* when a new or removed point is given.

6.1 Notation

Assume that we have an update (an insertion of a new point or a deletion of an existing point) at time t. We use q_i^t (δ_i^t) to denote q (δ) of p_i at time t. Let P^t be the set of points that have been generated so far and not removed at time t. We re-define n to represent $|P^t|$.

6.2 Baseline

The most straightforward approach to dealing with updates is to run an algorithm for static data whenever we receive an update. If the update frequency is low, then this approach may be tolerable. Otherwise, however, this approach cannot catch up the latest set of points. An algorithm that can *incrementally* update ρ and q is hence required to minimize the update time. Such an algorithm allows users to see the latest clusters when they want.

Given an update (insertion or deletion of p), a linear scan of P can update the local density of each point in P exactly. Notice that insertion or deletion of p updates the local densities of ρ points. These local density updates may incur an update of q_i for p_i.

- Assume that a new point is inserted. For each point p_i such that ρ_i increases, the candidates for q_i exists in the whole space. However, for each point p_j such that ρ_j does not increase, the candidates for q_j exist in the set of points whose local densities are updated.
- Given the deletion of an existing point, for each point p_i such that ρ_i decreases, the candidates for q_i exist in the whole space. In addition, for each point p_j such that ρ_j is larger than that of the previous q_j, the candidates for q_j also exists in the whole space.

A simple algorithm updates q of a given point p that has one of the above cases, through a linear scan of P. This algorithm considers the case where q_i (q_j) has to be updated for p_i (p_j), but this is actually "coarse-grained" and its search space is always P.

6.3 Updating ρ in D-DPC

Assume that we are given an update, an insertion, or a deletion of a point p_i. The local density of each point $p_j \in P^t$, such that $dist(p_i, p_j) < d_{cut}$, is updated. To update ρ_j (and obtain ρ_i in the insertion case), D-DPC employs a kd-tree $\mathcal{K}$.

More specifically, given a new or removed point p_i, D-DPC first inserts p_i into $\mathcal{K}$ or removes it from $\mathcal{K}$. Then, D-DPC runs a range search on $\mathcal{K}$. Let N_i be the set of points included in the search result. For each $p_j \in N_i$, D-DPC updates ρ_j. If the update is insertion, then D-DPC computes $\max_{N_i} \rho_j$ and $\min_{N_i} \rho_j$. From the discussion in Section 2.2, we have:

Corollary 2. *Given a new or removed point p_i, D-DPC needs $O(n^{1-\frac{1}{d}} + \rho_i)$ time to monitor the exact local density of each point in P^t when $\mathcal{K}$ is balanced. Otherwise, it needs $O(n)$ time.*

6.4 Updating q in D-DPC: Insertion Case

We next solve the main challenge of the dynamic DPC problem: how to deal with updating q. We clarify a theoretical upper-bound time to update q of every point $p \in P^t$.

To start with, we theoretically investigate cases in which q of $p \in P^t$ has to be updated when we have a point insertion. Given a new point p_i at time t, it is important to observe that the local

density of each point $\in N_i$ increases at least by one, whereas that of each point $\notin N_i$ does not change. Let $p' = q^{t-1}$. To efficiently update q of each $p \in P^t$, we consider six cases. We below introduce these cases. For ease of explanation, we first assign each point to one of two cases: A given point in P^t exists in N_i or not. Figure 4(a) illustrates an insertion of p_i (blue point) into the point set in Figure 1(a). The local density of each of the green points increases by one.

Case $p \in N_i$. In Figure 4(a), the points inside the rectangle fit into this case. We partition this case into two cases based on local density, since the local density of each point in N_i is updated. This means that we may have $q^t \neq q^{t-1}$ for such a point (e.g., green points in Figure 4(a)).

Each point in this case is assigned to one of an additional two cases through the following question: Does p have $\rho \geq \rho'$? If yes (no), then we assign this point to case 1 (2).

Case $p \notin N_i$. In Figure 4(a), the points outside the rectangle fit into this case. Since the local density of each point $p \notin N_i$ does not change, for p, $q^t = q^{t-1}$ or $q^t = p_j$ where $p_j \in N_i$. We therefore address the following question: Does there exist a point $p_j \in N_i$ such that p_j can be q^t?

We first use $\max_{N_i} \rho_j$ to answer the question. If $\rho \geq \max_{N_i} \rho_j$, then we assign p to case 3. Otherwise, we use $\min_{N_i} \rho_j - 1$ to confirm the change of the inferior-superior relationship in local density. If $\rho < \min_{N_i} \rho_j - 1$, then the relationship does not change, and we assign p to case 4. If we cannot answer the question by using local density, then we use δ^{t-1} and the triangle inequality. That is, for p, we investigate whether there exists a point $p_j \in N_i$ such that $\delta^{t-1} > dist(p, p_j)$ by using $dist(p, p_j) - d_{cut}$. If $\delta^{t-1} \leq dist(p, p_i) - d_{cut}$, then we assign p to case 5. Otherwise, p is assigned to case 6.

Summary of the six cases. We summarize the above cases.

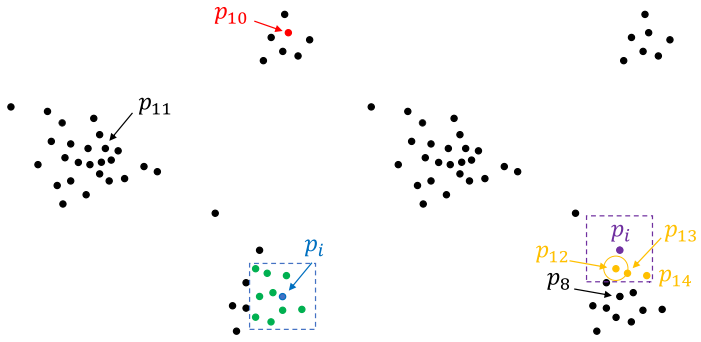
Case 1 $(p \in N_i) \wedge (\rho \geq \rho')$.
Case 2 $(p \in N_i) \wedge (\rho < \rho')$.
Case 3 $(p \notin N_i) \wedge (\rho \geq \max_{N_i} \rho_j)$.
Case 4 $(p \notin N_i) \wedge (\rho < \min_{N_i} \rho_j - 1)$.
Case 5 $(p \notin N_i) \wedge (\delta^{t-1} \leq dist(p, p_i) - d_{cut})$.
Case 6 $p \notin N_i$ and p does not have cases 3, 4, and 5.

We now analyze the cases where q has to be updated, and our result is as follows:

Lemma 2. *Given P^{t-1} and a new point p_i, q of $p \in P^t = P^{t-1} \cup \{p_i\}$ is or can be updated in cases 1, 2, 4, and 6.*

Proof. We present how the insertion of p_i affects q of p.

- *Case* 1: From Definition 2, $q^t \neq q^{t-1}$, so we have to update q of p that has this case. Note that p_i has this case, since it does not have q_i^{t-1}.
- *Case* 2: Because we do not have points p_j $(\neq p') \in P^{t-1}$ within $dist(p, p')$ from p such that $\rho < \rho_j$, it is guaranteed that $q^t = q^{t-1}$ or $q^t = p_i$.
- *Case* 3: In this case, from Definition 2, $p_j \in N_i$ cannot have $\rho < \rho_j$. Therefore, we have $q^t = q^{t-1}$.
- *Case* 4: In this case, for p, P^{t-1} does not have any point p_j that "newly" becomes $\rho < \rho_j$. This means that $q^t = q^{t-1}$ or $q_t = p_i$, as with case 2.
- *Case* 5: From the triangle inequality, all points $p_j \in N_i$ have $dist(p, p') \leq dist(p, p_j)$. We therefore have $q^t = q^{t-1}$.
- *Case* 6: In this case, $p_j \in N_i$ can have $\rho < \rho_j$ and $dist(p, p_j) < dist(p, p')$. We have $q^t = q^{t-1}$ or $q^t = p_j$ where $p_j \in N_i$, so we have to compare them. □



(a) Insertion of p_i (b) Deletion of p_i

Fig. 4. Examples of insertion/deletion from the point set in Figure 1(a).

Example 4. In Figure 4(a), consider the points outside the rectangle. We have $\max_{N_i} \rho_j = 10$ and $\min_{N_i} \rho_j - 1 = 2$. For a point $p \notin N_i$ such that $\rho \geq 10$, N_i does not contain q^t (case 3). Also, if $\rho < 2$, at time $t-1$, then the local density of p is also less than those of the green points. The blue point, i.e., p_i, is the only new point that has a larger local density than ρ (case 4). Last, $\rho_{10} = 7$ and $\delta_{10}^{t-1} = dist(p_{10}, p_{11})$. From the triangle inequality, no points inside the rectangle have a shorter distance than δ_{10}^{t-1} (case 5).

6.4.1 Algorithm. As a consequence of Lemma 2, we focus on only points that have cases 1, 2, 4, or 6. Algorithm 3 details our algorithm when a new point is inserted. First focus on each point $p \in N_i$. If p has case 1, then D-DPC retrieves q by a *progressive* k-NN search. In this search, for $j \in [1, k-1]$, the jth nearest neighbor is outputted whenever it is identified, which is enabled by an approach based on priority queue. We hence do not have to wait for the k points to be returned if the jth nearest neighbor has higher local density than ρ. The rationale is the same as that in the optimization of Ex-DPC++.[8] (If the k points have less local density than ρ, then D-DPC finds q^t by a linear scan of P^t.) If p has case 2, then D-DPC compares q^{t-1} and p_i.

Next, consider a point $p \notin N_i$. D-DPC first checks whether p has $\rho < \max_{N_i} \rho_j$. If not, then D-DPC evaluates whether p has $\rho > \min_{N_i} \rho_j - 1$. If p has case 4, then D-DPC compares q^{t-1} and p_i. Otherwise, D-DPC evaluates whether $\delta^{t-1} > dist(p, p_i) - d_{cut}$. If yes, then D-DPC simply scans N_i to search points p_j such that $\rho < \rho_j$ and $dist(p, p_j) < \delta^{t-1}$. If there exists such a point, then D-DPC updates q^t. Otherwise, $q^t = q^{t-1}$.

6.4.2 Analysis. Let x_1, x_2, x_4, and x_6 be the numbers of points that have cases 1, 2, 4, and 6, respectively. The following theorem yields a theoretical upper-bound time to obtain q of each point $p \in P^t$ when we have a new point:

Theorem 3. *Given a new point p_i at time t, D-DPC needs $O(\rho_i n)$ time to obtain q of each point $p \in P^t$ in the worst case.*

Proof. Focus on $p \in P^t$ that has case 1, 2, 4, or 6. In case 1, it is trivial that D-DPC needs at most $O(n)$ time to update q. In cases 2 and 4, trivially $O(1)$ time is required. In case 6, verifying q needs $O(\rho_i)$ time, since $|N_i| = \rho_i$. Then, D-DPC needs

$$O(x_1 n + x_2 + x_4 + x_6 \rho_i) = O(x_1 n + x_6 \rho_i) \tag{4}$$

[8] D-DPC does not maintain the k nearest neighbors for each point, because this is negatively sensitive to deletions.

ALGORITHM 3: D-DPC (insertion case)

Input: P^{t-1} (a point set), d_{cut}, $\mathcal{K}$ (a kd-tree) of P^{t-1}, p_i (a new point)

```
1  P^t ← P^{t−1} ∪ {p_i}
2  Insert p_i into K                                   ▷ incremental kd-tree update
3  /* Updating ρ */
4  N_i ← Range-Search(p_i, d_cut, K)                   ▷ maintain the result set N_i of the orthogonal range search on K with
                                                        query point p_i
5  foreach p ∈ N_i do
6      ρ ← ρ + 1                                       ▷ update the local density of each point in N_i
7  ρ_i ← |N_i|                                         ▷ set the local density of the new point
8  /* Updating q */
9  foreach p ∈ P^t do
10     p' ← q_j^{t−1}                                  ▷ let p' be q_j at time t − 1 (i.e., the previous q_j)
11     if p_j ∈ N_i then
12         if ρ ≥ ρ' then
13             q^t ← Progressive-kNN-Search(p, k, K)    ▷ case 1 in Lemma 2. k'(≤ k)-th nearest neighbor
                                                          is returned whenever it is identified.
14         else
15             if dist(p, p_i) < δ^{t−1} then
16                 q^t ← p_i                           ▷ case 2 in Lemma 2
17     else
18         if ρ < max_{N_i} ρ_j then
19             if ρ < min_{N_i} ρ_j − 1 then
20                 Run lines 15– 16                    ▷ case 4 in Lemma 2
21             else
22                 if δ^{t−1} > dist(p, p_i) − d_cut then
23                     foreach p_j ∈ N_i s.t. ρ < ρ_j do
24                         if δ^{t−1} > dist(p, p_j) then
25                             q^t ← p_j               ▷ case 6 in Lemma 2
```

time to update q of every $p \in P^t$ having case 1, 2, or 6. Notice that $x_1 \leq \rho_i$ and $x_6 \leq n - \rho_i$. From this and Equation (4), we have $O(\rho_i n)$ time to exactly obtain q of each point in P^t. This result and Corollary 2 derive Theorem 3. □

Remark 1. Recall Section 5.5, and q of each point usually exists near it. The progressive k nearest neighbor search hence quickly identifies q usually. In addition, we have $x_1 \ll \rho_i$ and $x_6 \ll n - \rho_i$ in practice (x_1 is around 2, and x_6 is at most several hundred in our experiments).

6.4.3 Parallelization. The local density update needs a single range search on $\mathcal{K}$, thus, we focus on updating q. Clearly, the above algorithm can evaluate q of each point in P^t independently. We hence can utilize the parallelization approach for local density computation in Ex-DPC to parallelize D-DPC.

6.5 Updating q in D-DPC: Deletion Case

We next assume that a point p_i is removed. The local density of each point in N_i decreases by one, whereas that of each point $\notin N_i$ does not change. To update q of each point in P^t, we consider

three cases: (1) p cannot have $q^t = q^{t-1}$, (2) $p \in N_i$, and (3) $p \notin N_i$. More precisely, we assign each point in P^t to one of the following cases:

Case 7 $(\rho \geq \rho') \vee (q^{t-1} = p_i)$.
Case 8 $p \in N_i$ and p does not have case 7.
Case 9 $p \notin N_i$ and p does not have case 7.

We then identify the cases where we have to update q and its search space.

Lemma 3. *Assume that a point p_i is removed, and q has to be updated in cases 7 and 8.*

Proof. As with Lemma 2, we present how the deletion of p_i affects q of p.

- *Case* 7: This case is essentially the same as case 1.
- *Case* 8: We still have $\rho < \rho'$. However, due to the decrease of ρ, some points $p_j \notin N_i$ may newly have $\rho < \rho_j$. Therefore, q^t exists in $\{p_j \mid \rho < \rho_j, dist(p, p_j) \leq \delta^{t-1}\}$.
- *Case* 9: Each point $p_j \in P^t$ with $dist(p, p_j) < dist(p, p')$ has $\rho_j < \rho$, since the local density of p does not change. Therefore, we have $q^t = q^{t-1}$. □

Example 5. In Figure 4(a), consider p_i is removed. We have $N_i = \{p_{12}, p_{13}, p_{14}\}$. Case 7 is trivial, so this example considers case 8; p_{12} and p_{13} fall into this case. For p_{12}, $q_{12}^{t-1} = p_{13}$. Due to the decrease of ρ_{12}, points $\notin N_i$ may have q_{12}^t. If so, then such a point p_j must have $dist(p_{12}, p_j) < \delta^{t-1} = dist(p_{12}, p_{13})$. We hence need to search the points in the range (smaller orange circle). For p_{13}, $q_{13}^{t-1} = p_8$, and the points inside the larger orange circle need to be evaluated to update q_{13}. From this figure, case 9 is also clear, because the points in the purple rectangle do not affect q^{t-1} of each point $p \notin N_i$.

6.5.1 Algorithm. For each point $p \in P^t$ having case 7, D-DPC retrieves q^t by a progressive k nearest neighbor search on $\mathcal{K}$, since this case essentially faces the same situation as case 1. For each point $p \in P^t$ having case 8, D-DPC retrieves q^t by a circular range search, whose radius is δ^{t-1}, on $\mathcal{K}$, since q^t must exist in this search space. Algorithm 4 summarizes these procedures.

6.5.2 Analysis. Let x_7 and x_8, respectively, be the numbers of points that have cases 7 and 8.

Theorem 4. *For a deletion case, D-DPC needs $O(\rho_{avg} n)$ amortized time to obtain q of each point $p \in P^t$.*

Proof. Consider a deletion of a (random) point p_i such that $\rho_i \approx \rho_{avg}$. Given a point $p \in P^t$ having case 7, D-DPC needs at most $O(n)$ time to update q, so $O(x_7 n)$ time is required. Given a point $p \in P^t$ having case 8, D-DPC needs $O(n)$ time to update q in the worst case, thus $O(x_8 n)$ time is required to deal with points having this case. We have $x_7 \leq n - \rho_{avg}$, because we rarely have points p with $(p \in N_i) \wedge (q^{t-1} = p_i)$. Also, we have $x_8 \leq \rho_i$. The average of x_7 is less than 1, because the amortized number of points p whose q is $p_i \in P^t$ is $n/n = 1$. Hence, $O(x_7 n + x_8 n) = O(\rho_{avg} n)$. □

6.5.3 Parallelization. Each point having case 7 or 8 can be evaluated independently, thus, in deletion cases, parallelizing D-DPC is straightforward.

7 EXPERIMENTS

Our experiments were conducted on a machine equipped with dual 12-core 3.0 GHz Intel Xeon E5-2687W v4 processors and 512 GB RAM.

7.1 Evaluation of Density-based Clustering Algorithms

We first compare the clustering results of density-based clustering algorithms.

ALGORITHM 4: D-DPC (deletion case)

Input: P^{t-1} (a point set), d_{cut}
$\mathcal{K}$ (a kd-tree) of P^{t-1}, p_i (a removed point)
$P^t \leftarrow P^{t-1} \setminus \{p_i\}$
Remove p_i from $\mathcal{K}$ ▷ incremental kd-tree update
/* Updating ρ */
$N_i \leftarrow$ Range-Search$(p_i, d_{cut}, \mathcal{K})$ ▷ maintain the result set N_i of the orthogonal range search on $\mathcal{K}$ with query point p_i
for *each* $p \in N_i$ **do**
  $\rho \leftarrow \rho - 1$ ▷ update the local density of each point in N_i
/* Updating q */
foreach $p \in P^t$ **do**
  $p' \leftarrow q_j^{t-1}$ ▷ let p' be q_j at time $t-1$ (i.e., the previous q_j)
  if $(\rho \geq \rho') \vee (q^{t-1} = p_i)$ **then**
    $q^t \leftarrow$ Progressive-kNN-Search$(p, k, \mathcal{K})$ ▷ case 7 in Lemma 3
  else
    if $(p \in N_i) \wedge (q^{t-1} \neq p_i)$ **then**
      $N \leftarrow$ Range-Search$(p, \delta^{t-1}, \mathcal{K})$ ▷ case 8 in Lemma 3
      $q^t = \underset{p_j \in N : \rho < \rho_j}{\arg\max} \; dist(p, p_j)$

7.1.1 Algorithms. In this experiment, we evaluated OPTICS [7], DBSCAN [19], HDBSCAN* [12], DENCLUE [27], and DPC. We used scikit-learn[9] for OPTICS and DBSCAN. For HDBSCAN*, we used a public code.[10] Note that all DPC algorithms evaluated in Section 7.2 (including ours) return the exact DPC result. That is, they can produce all the DPC results introduced in Section 7.1.

7.1.2 Datasets. We used five synthetic 2-dimensional datasets, *S1*, *S2*, *S3*, *S4* [20], and *Syn*. S1, S2, S3, and S4 have 15 Gaussian clusters and the same cardinality (5,000), whereas the degree of cluster overlapping of Sx increases as x increases (i.e., cluster borders become ambiguous). In addition, they have ground truth labels. Syn was generated based on a random walk model introduced in Reference [21] and consists of 100,000 points. The domain of each dimension in Syn was $[0, 10^5]$.

7.1.3 Result. We measured ARI (Adjusted Rand Index) of each algorithm on S1, S2, S3, and S4 by using their ground truth labels. Table 3 shows the result: DPC is the winner. From this result, we see that DPC is more robust to datasets that have clusters with ambiguous borders.

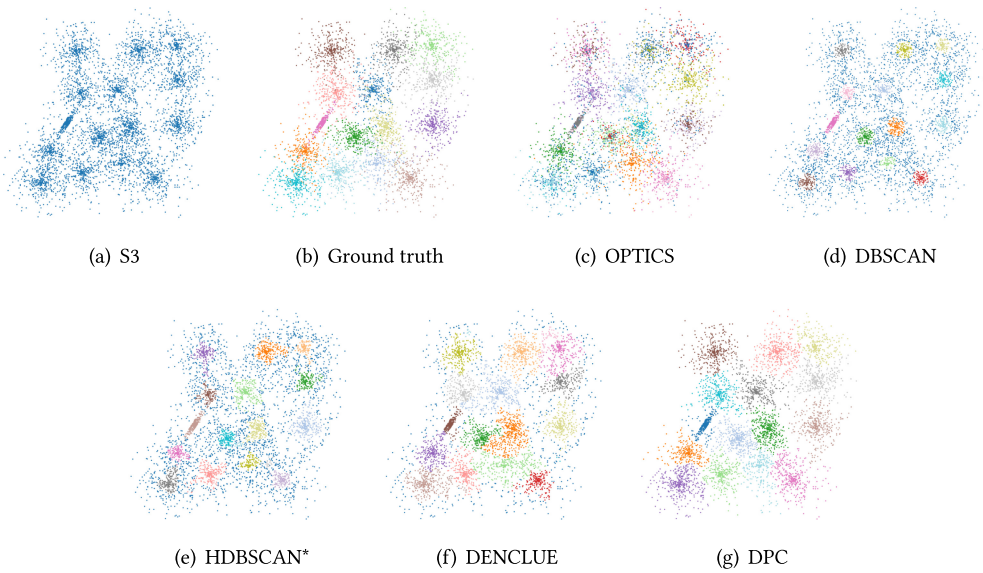
To understand this more intuitively, we show the visualization result of each algorithm on S3 in Figure 5. OPTICS shows a noisy result (although the ground truth also has some noisy labels). DBSCAN and HDBSCAN* return "small" clusters. They have to specify points in comparatively sparse regions as noise, so many points (blue points) do not belong to any clusters. (Otherwise, many clusters would be merged as with those in Figure 2(b).) If applications want to provide each point with the same label as that of its close point, then DBSCAN and HDBSCAN* may fail to do this. DENCLUE yields a visually better result than those of DBSCAN and HDBSCAN*, but some clusters have different shapes from those in the ground truth. Different from the ARI results, DPC seems to return almost the same clusters as those in the ground truth. (There are indeed small

[9] https://scikit-learn.org/stable/.
[10] https://hdbscan.readthedocs.io/en/latest/.

Table 3. ARI of Each Algorithm

Algorithm	S1	S2	S3	S4
OPTICS	0.692	0.650	0.595	0.524
DBSCAN	0.960	0.781	0.360	0.372
HDBSCAN*	0.931	0.689	0.163	0.187
DENCLUE	0.958	0.772	0.560	0.278
DPC	**0.989**	**0.932**	**0.720**	**0.632**

Bold shows the winner.



(a) S3 (b) Ground truth (c) OPTICS (d) DBSCAN

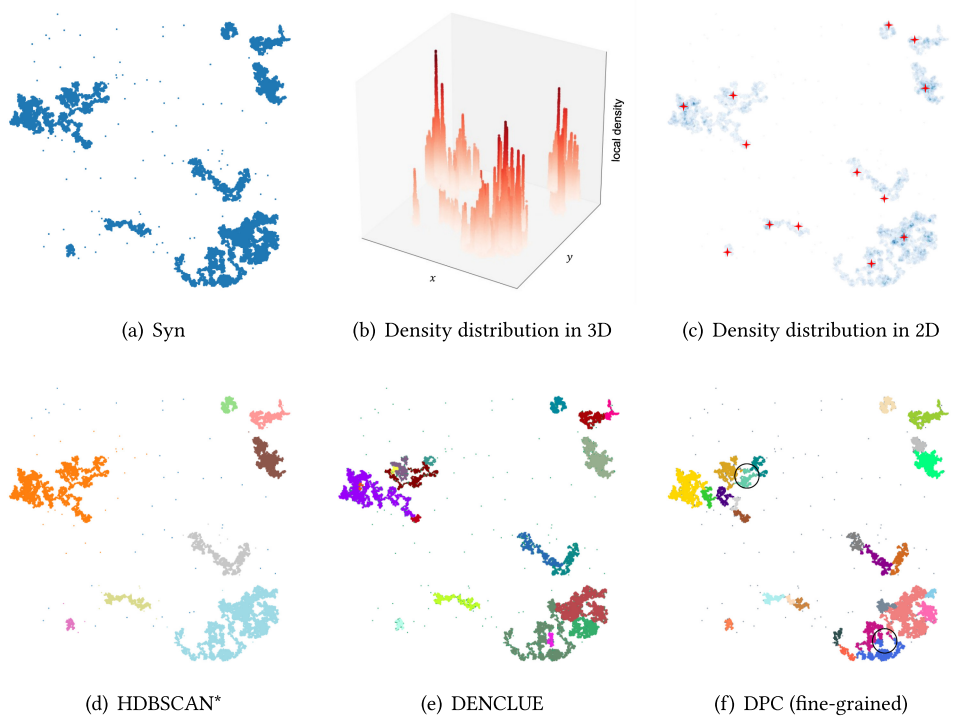
(e) HDBSCAN* (f) DENCLUE (g) DPC

Fig. 5. Clustering results on S3 (best viewed in color).

differences in cluster borders.) This result is obtained from the merit of DPC: its ability to catch density-peaks and density gradient, thanks to δ.

We next show the clustering results on Syn. Syn is illustrated in Figure 6(a) and is a more complex dataset than Sx; see Figure 6(b), which shows its density distribution in 3D, i.e., the x- and y-axes show the coordinate, whereas the z-axis shows the local density (darker means higher). Figure 6(c) shows its density distribution in 2D (darker means denser), and the red stars represent the main density-peaks in the corresponding regions. As Syn does not have ground truth, we show only the visualization results. Recall that the results of DPC and DBSCAN appear in Figure 2. In addition, we omit the result of OPTICS, because OPTICS could not form meaningful clusters.

Figure 6(d) describes that HDBSCAN* returns the same clusters as those in DBSCAN (see Figure 2(b)). DENCLUE tries to catch all local maxima w.r.t. density, but having all of them is not guaranteed. Therefore, as shown in Figure 6(e), DENCLUE sometimes fails to find local maxima (i.e., density-peaks); see left-top and right-bottom regions. DPC successfully finds the density-peaks (the red stars), so Syn is well partitioned into subsets, each of which is formed from the density-peak. In addition, thanks to δ_{min}, DPC can control the granularity of clusters, different from DENCLUE. If users want finer-grained clusters than those in Figure 2(a), then they can provide a smaller δ_{min}. Figure 6(f) shows this case (the clusters in Figure 2(a) are further partitioned if they have some sub-density-peaks). Here, in Figure 6(f), we observe a counter-intuitive result: Some points



(a) Syn (b) Density distribution in 3D (c) Density distribution in 2D

(d) HDBSCAN* (e) DENCLUE (f) DPC (fine-grained)

Fig. 6. Clustering results on Syn (best viewed in color).

in the same clusters (specified by circles) are not connected "by a land." This is because of the definition of q. If a given point p does not have close points, which have higher local density than ρ and are accessible by a land, then such a phenomenon happens.[11] If applications want to avoid or fix this (by disobeying the policy of following the nearest neighbor point with larger density), then automatically doing it may not be a trivial task.

Actually, DPC's clustering result quality on some other irregular-shaped datasets (e.g., famous benchmark datasets with irregular shapes, namely, `aggregation`, `curve`, `flame`, and `spiral` [20]) have already been evaluated in existing works [8, 34, 42, 49]. They show that DPC is better than other density-based clustering algorithms (e.g., OPTICS and DBSCAN). For conciseness, this article does not show the results, and interested readers may refer to them.

7.2 Evaluation on Static Data

This section focuses on evaluations of the performances of DPC algorithms. Table 3, Figures 5, and 6 show that the other density-based clustering algorithms return *(totally) different results*, thus comparing their running times is meaningless. They are hence not the scope of this subsection.

7.2.1 Datasets. We used four real datasets shown in Table 4, which were also used in References [21, 23, 24, 45, 46]. For Airline,[12] we removed date and string information and used the remaining attributes. For Household and Sensor,[13] we used global sensor readings and MOX gas sensor values,

[11] This is not specific to DPC, as DBSCAN also can have this case (if ϵ is large) [21, 23].
[12] http://kt.ijs.si/elena_ikonomovska/datasets/airline/2008_14col.data.bz2.
[13] https://archive.ics.uci.edu/ml/index.php.

Table 4. Statistics of Datasets

Dataset	n	d	Domain
Airline	5,810,462	3	$[0, 10^6]$
Household	2,049,280	4	$[0, 10^5]$
Sensor	928,991	8	$[0, 10^5]$
T-Drive [48]	15,503,523	2	$[0, 10^5]$

Table 5. Ex-DPC++ vs. Ex-DPC++ w/o Optimization w.r.t. Time [Sec]

	Ex-DPC++	Ex-DPC++ w/o optimization
Airline	43.38	245.24
Household	13.97	660.07
Sensor	9.16	94.89
T-Drive	73.93	> 1800

respectively. For T-Drive, we used its records of x-y coordinates. To be fair for the competitors, we normalized them, and their domains in each dimension are also shown in Table 4. We set d_{cut} according to the guidance of Reference [38].

7.2.2 Algorithms. We evaluated *FDDP* [34], *IB-DPC* [37], *Ex-DPC*, and *Ex-DPC++* (we set $k = 50$).[14] We followed the original paper to set the inner parameters of the state-of-the-art algorithms. We omit the results of the brute-force algorithm [38] and CFSFDP-A [8] in this article, because Reference [2] has already shown that they are significantly outperformed by Ex-DPC.

All evaluated algorithms were implemented in C++, and we used OpenMP for multi-threading. We report the running times of the evaluated algorithms. The default number of threads is 12.

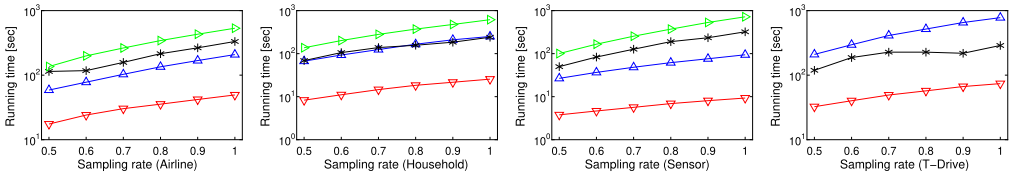
7.2.3 Impact of optimization of Ex-DPC++. First, we study the impact of the optimization presented in Section 5.5. Table 5 shows the result. The optimization provides a clear speedup, because many points can obtain their q in $O(1)$ time. Particularly, on Household, Sensor, and T-Drive, the optimization yields more than 10× speedup.

7.2.4 Impact of cardinality. We investigate the scalability of each algorithm to the number of points in a dataset. We varied the number of points in each dataset via uniform sampling, i.e., by varying the sampling rate. (The other parameters are fixed by their default values.) Figure 7 plots the result. Ex-DPC++ always outperforms the others, and Ex-DPC is better than or competitive with IB-DPC except the case of T-Drive. Note that FDDP could not work on T-Drive due to run out of memory.

Let us look at Table 6 again. Ex-DPC++ improves the time to compute ρ and δ against Ex-DPC. Ex-DPC++ is about 4, 13, 11, and 10 times faster than Ex-DPC on Airline, Household, Sensor, and T-Drive, respectively. This result confirms the merits of range counting and utilizing k nearest neighbors.

7.2.5 Impact of #threads. Table 7 shows the scalability to the number of threads on Airline. We omit the results on the other datasets, because their tendencies are the same. Normally, each algorithm improves its running time with an increase in the number of available threads, and our algorithms are faster than the others. FDDP has bad load balance, as we observe that their running times do not change much with a larger number of threads. For example, FDDP obtains only 5×

[14] https://github.com/amgt-d1/Ex-DPC-plus-plus.



(a) Airline (b) Household (c) Sensor (d) T-Drive

Fig. 7. Impact of cardinality (sampling rate). "▷" shows FDDP, "∗" shows IB-DPC, "△" shows Ex-DPC, and "▽" shows Ex-DPC++.

Table 6. Decomposed Time [Sec] on Airline, Household, Sensor, and T-Drive (Parameters Are Default Ones)

	Airline		Household		Sensor		T-Drive	
Algorithm	ρ comp.	δ comp.	ρ comp.	δ comp.	ρ comp.	δ comp.	ρ comp.	δ comp.
FDDP	527.26	5.67	607.41	7.00	715.68	2.42	-	-
IB-DPC	123.28	208.56	174.82	64.80	304.10	17.09	101.41	182.14
Ex-DPC	80.36	130.21	56.60	169.39	88.43	5.25	60.55	712.97
Ex-DPC++	43.39	5.42	13.97	3.59	7.07	1.88	27.97	41.23

Table 7. Impact of Number of Threads (Time [Sec] and Speedup Ratio) on Airline

#threads	1		12		24		48	
Algorithm	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
FDDP	1,803.31	1.00	533.49	3.38	419.34	4.30	359.13	5.02
IB-DPC	3,256.81	1.00	332.39	9.80	204.45	15.93	139.18	23.40
Ex-DPC	1,044.41	1.00	210.43	4.97	177.62	5.88	159.01	6.57
Ex-DPC++	567.05	1.00	48.80	11.62	34.38	16.49	25.77	22.00

speedup with 48 (hyper-)threads compared with its single thread case. The limitation of Ex-DPC (computing q cannot be done in parallel) is observed from the result. As we have more threads, the main overhead of Ex-DPC becomes computing q, then its running time cannot be reduced much.

Compared with the above algorithms, Ex-DPC++ receives more benefits from available threads. For example, when a single CPU is used, the speedup ratio of Ex-DPC++ is almost the same as the number of threads. When the number of threads is more than 12, two CPUs work, so memory access latency occurs (due to remote memory access), which prevents linear-scale speedup. In addition, when the number of threads is more than 24, hyper-threading also works. Hyper-threads are not physical cores, so it is practically impossible to have linear scalablity to the number of threads when they include hyper-threads. These phenomena are also observed in Reference [11].

7.2.6 Memory Usage. Last, we study the memory usage of the evaluated algorithms by using the default setting. Table 8 shows the result. Recall that FDDP could not work on T-Drive, thus its result is omitted. Ex-DPC consumes the least memory. Ex-DPC++ needs more memory than Ex-DPC, since it maintains the k nearest neighbors of each point, but it still fits into modern RAM easily.

7.3 Evaluation on Dynamic Data

7.3.1 Datasets. We used Airline, Household, Sensor, and T-Drive.

Table 8. Memory Usage [GB]

	Airline	Household	Sensor	T-Drive
FDDP	29.521	71.256	65.185	-
IB-DPC	0.563	0.346	0.133	1.186
Ex-DPC	**0.461**	**0.171**	**0.093**	**1.167**
Ex-DPC++	1.853	0.674	0.326	3.583

7.3.2 Algorithms. We evaluated the following algorithms:

- *oDP+*: an extension of oDP [40]. (Recall that oDP does not provide a specific algorithm for updating q.) As with oDP, oDP+ updates the local density of each point via a linear scan. For updating q, oDP+ uses the algorithm introduced in Section 6.2.
- *EDMStream* [25]: an approximation algorithm for dynamic DPC. As for local density, we used Definition 1. (Recall that EDMStream originally uses a different definition.)
- *D-DPC*: our algorithm introduced in Section 6.[15]
- *D-DPC-S*: a variant of D-DPC. This algorithm uses linear scans for updating ρ and q (if necessary) to observe the effectiveness of Lemmas 2 and 3 by comparing with oDP+.

Note that the above algorithms are parallelizable, and we used 12 threads by default.

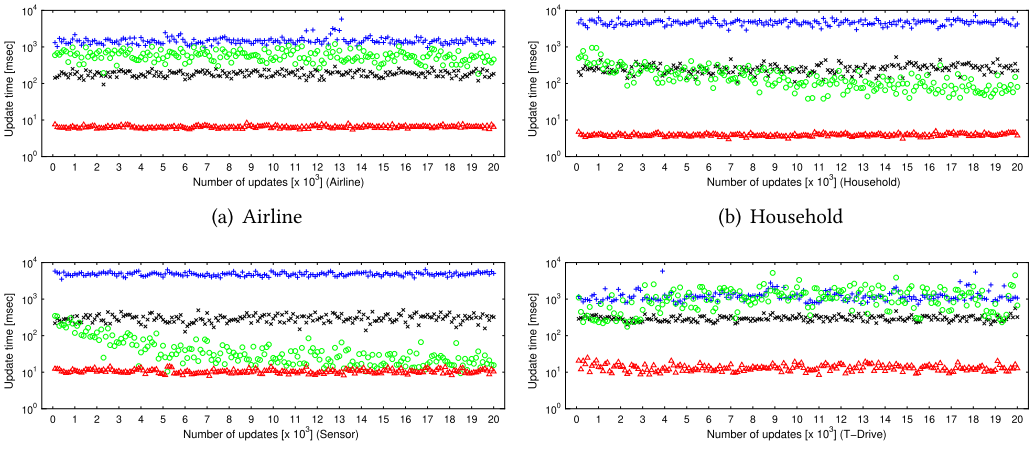
7.3.3 Workload. For each dataset, we used $n - u$ points as the initial dataset, and u updates (consisting of insertion of new points and removal of existing points) were used as a workload. Specifically, a workload contained $u \times \gamma$ deletions and $u \times (1 - \gamma)$ insertions, i.e., an update was insertion with probability $(1 - \gamma)$ and deletion with probability γ. When an update was insertion, we inserted a point $\notin P^{t-1}$ in the original generation order. However, when an update was deletion, we removed the oldest point $\in P^{t-1}$. By default, $\gamma = 0.2$, because deletions occur less often than insertions [4, 22], and $u = 20,000$ similar to Reference [44].

7.3.4 Comparison with state-of-the-art. We study the update efficiencies of the evaluated algorithms. (D-DPC uses a single kd-tree as an index, thus the memory usage of D-DPC is at most that of Ex-DPC shown in Table 8.) Figure 8 plots the average update time of each algorithm every 100 updates, and Table 9 shows the time to complete the workload.

We see that oDP+ is quite slow, and Figure 8 shows that oDP+ needs more than one second to update q of each point when an update is given in most cases. D-DPC-S considers when q^t of each point $p \in P^t$ needs to be updated by using Lemmas 2 and 3. Therefore, compared with oDP+, D-DPC-S generally obtains a significant speedup, which demonstrates the impact of Lemmas 2 and 3. Although EDMStream is an approximation algorithm, it cannot obtain clear speedup, and it is outperformed by D-DPC-S on Airline and T-Drive, as shown in Table 9. EDMStream is actually sensitive to deletions. Recall that EDMStream uses samples and not-sampled points are absorbed by their nearest sample. If a sample is deleted, then EDMStream needs to compute new samples from absorbed points, their local densities, and their nearest points with higher local density. This is quite expensive and incurs non-stable performance, as Figure 8 describes.

D-DPC clearly outperforms both the state-of-the-art exact and approximation algorithms, and its performance is stable and scalable. For example, D-DPC completes the workload about 91 (234), 43 (1,218), 5 (459), and 87 times faster than EDMStream (oDP+) on Airline, Household, Sensor, and T-Drive, respectively. In addition to keeping small x_1, x_6, x_7, and x_8, D-DPC further reduces the

[15] https://github.com/amgt-d1/D-DPC.



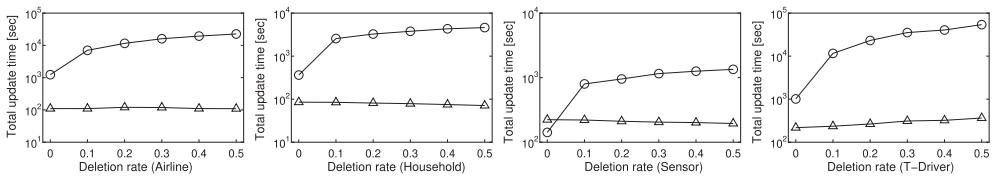
(a) Airline (b) Household

(c) Sensor (d) T-Drive

Fig. 8. Comparison with state-of-the-art. "+" shows oDP+, "○" shows EDMStream, "×" shows D-DPC-S, and "△" shows D-DPC.

Table 9. Total Update Time to Complete the Workload [Sec]

	Airline	Household	Sensor	T-Drive
oDP+	30,221	93,792	96,980	25,184
EDMStream	11,741	3,352	992	23,122
D-DPC-S	3,727	5,191	6,037	5,952
D-DPC	**129**	**77**	**195**	**265**



(a) Airline (b) Household (c) Sensor (d) T-Drive

Fig. 9. Impact of deletion rate. "○" shows EDMStream and "△" shows D-DPC.

search space by exploiting the progressive k nearest neighbor search when we need to update q. Therefore, D-DPC obtains a significant speedup compared with D-DPC-S.

7.3.5 Impact of Deletion Rate. Figure 9 studies the sensitivity to deletion rate γ. EDMStream is efficient only when $\gamma = 0$ (but is still outperformed by our *exact* algorithm D-DPC on Airline, Household, and T-Drive), and it needs longer time as γ increases. EDMStream incurs long time to update the sample points, and the number of this event becomes larger as γ increases.

However, D-DPC does not suffer from a large number of deletions. Figure 9 shows that the time of D-DPC almost does not vary. This robustness of D-DPC against deletions is a clear advantage for fully dynamic data.

7.3.6 Impact of Workload Size. Table 10 studies the scalability of D-DPC by varying workload size u. (The other algorithms are clearly slower than D-DPC, thus were not tested.) We see that the total update time of D-DPC is proportional to u. This means that, even when u is large, the

Table 10. Impact of Workload Size u on the Total Update Time [Sec] of D-DPC

u	Airline	Household	Sensor	T-Drive
20,000	129.01	77.43	194.63	264.94
40,000	222.36	154.44	410.59	528.07
60,000	335.52	225.60	605.74	790.72
80,000	445.10	316.40	801.24	1,112.38
100,000	565.30	383.64	989.74	1,388.18

average computation time per update does not change and D-DPC keeps the similar behavior as those depicted in Figure 8.

8 CONCLUSION

In this article, to efficiently deal with Density-Peaks Clustering (DPC) on static data, we proposed Ex-DPC and Ex-DPC++. We also addressed the fact that real-world applications are often subjective to updates (insertions and deletions) and efficiently supporting cluster updates is also important. We tackled the problem of DPC on dynamic data, and we proposed D-DPC, which efficiently updates the main criteria of DPC and can utilize multi-threading. Our experimental results have confirmed that (i) Ex-DPC++ is much faster than the state-of-the-art exact algorithms, and (ii) our algorithm for dynamic DPC updates the main criteria of DPC significantly faster than the state-of-the-art exact and approximation ones.

REFERENCES

[1] Daichi Amagata. 2022. Scalable and accurate density-peaks clustering on fully dynamic data. In *IEEE Big Data*. 445–454.
[2] Daichi Amagata and Takahiro Hara. 2021. Fast density-peaks clustering: Multicore-based parallelization approach. In *SIGMOD*. 49–61.
[3] Daichi Amagata and Takahiro Hara. 2022. Fast density-peaks clustering: Multicore-based parallelization approach. *arXiv:2207.04649v2* (2022).
[4] Daichi Amagata, Takahiro Hara, and Chuan Xiao. 2019. Dynamic set kNN self-join. In *ICDE*. 818–829.
[5] Daichi Amagata, Makoto Onizuka, and Takahiro Hara. 2021. Fast and exact outlier detection in metric spaces: A proximity graph-based approach. In *SIGMOD*. 36–48.
[6] Daichi Amagata, Makoto Onizuka, and Takahiro Hara. 2022. Fast, exact, and parallel-friendly outlier detection algorithms with proximity graph in metric spaces. *VLDB J* 31, 4 (2022), 1–25.
[7] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering points to identify the clustering structure. *ACM SIGMOD Rec.* 28, 2 (1999), 49–60.
[8] Liang Bai, Xueqi Cheng, Jiye Liang, Huawei Shen, and Yike Guo. 2017. Fast density clustering strategies based on the k-means algorithm. *Pattern Recog.* 71 (2017), 375–386.
[9] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.
[10] Alina Beygelzimer, Sham Kakade, and John Langford. 2006. Cover trees for nearest neighbor. In *ICML*. 97–104.
[11] Panagiotis Bouros, Nikos Mamoulis, Dimitrios Tsitsigkos, and Manolis Terrovitis. 2021. In-memory interval joins. *VLDB J.* 30, 4 (2021), 667–691.
[12] Ricardo J. G. B. Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. 2015. Hierarchical density estimates for data clustering, visualization, and outlier detection. *ACM Trans. Knowl. Discov. Data* 10, 1 (2015), 5.
[13] Feng Cao, Martin Estert, Weining Qian, and Aoying Zhou. 2006. Density-based clustering over an evolving data stream with noise. In *SDM*. 328–339.
[14] Gromit Yeuk-Yin Chan, Fan Du, Ryan A. Rossi, Anup B. Rao, Eunyee Koh, Cláudio T. Silva, and Juliana Freire. 2020. Real-time clustering for large sparse online visitor data. In *WWW*. 1049–1059.
[15] T. H. Hubert Chan, Arnaud Guerqin, and Mauro Sozio. 2018. Fully dynamic K-center clustering. In *WWW*. 579–587.
[16] Bo Chen, Kai Ming Ting, Takashi Washio, and Ye Zhu. 2018. Local contrast as an effective means to robust clustering against varying densities. *Mach. Learn.* 107, 8 (2018), 1621–1645.

[17] Yixin Chen and Li Tu. 2007. Density-based clustering for real-time stream data. In *KDD*. 133–142.
[18] Zengjian Chen, Jiayi Liu, Yihe Deng, Kun He, and John E. Hopcroft. 2019. Adaptive wavelet clustering for highly noisy data. In *ICDE*. 328–337.
[19] 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 *KDD*. 226–231.
[20] Pasi Fränti and Sami Sieranoja. 2018. K-means properties on six clustering benchmark datasets. *Appl. Intell.* 48, 12 (2018), 4743–4759.
[21] Junhao Gan and Yufei Tao. 2015. DBSCAN revisited: Mis-claim, un-fixability, and approximation. In *SIGMOD*. 519–530.
[22] Junhao Gan and Yufei Tao. 2017. Dynamic density based clustering. In *SIGMOD*. 1493–1507.
[23] Junhao Gan and Yufei Tao. 2017. On the hardness and approximation of euclidean DBSCAN. *ACM Trans. Datab. Syst.* 42, 3 (2017), 14.
[24] Junhao Gan and Yufei Tao. 2018. Fast Euclidean optics with bounded precision in low dimensional space. In *SIGMOD*. 1067–1082.
[25] Shufeng Gong, Yanfeng Zhang, and Ge Yu. 2017. Clustering stream data by exploring the evolution of density mountain. *PVLDB* 11, 4 (2017), 393–405.
[26] Michael Hahsler and Matthew Bolaños. 2016. Clustering data streams based on shared density between micro-clusters. *IEEE Trans. Knowl. Data Eng.* 28, 6 (2016), 1449–1461.
[27] Alexander Hinneburg and Hans-Henning Gabriel. 2007. DENCLUE 2.0: Fast clustering based on kernel density estimation. In *IDA*. 70–80.
[28] Alexander Hinneburg and Daniel A. Keim. 2003. A general approach to clustering in large databases with noise. *Knowl. Inf. Syst.* 5, 4 (2003), 387–415.
[29] Jian Hou, Aihua Zhang, and Naiming Qi. 2020. Density peak clustering based on relative density relationship. *Pattern Recog.* 108 (2020), 107554.
[30] Ruizhen Hu, Wenchao Li, Oliver Van Kaick, Hui Huang, Melinos Averkiou, Daniel Cohen-Or, and Hao Zhang. 2017. Co-locating style-defining elements on 3D shapes. *ACM Trans. Graph.* 36, 3 (2017), 33.
[31] Mike Izbicki and Christian Shelton. 2015. Faster cover trees. In *ICML*. 1162–1170.
[32] Bogyeong Kim, Kyoseung Koo, Juhun Kim, and Bongki Moon. 2021. DISC: Density-based incremental clustering by striding over streaming data. In *ICDE*. 828–839.
[33] Hans-Peter Kriegel, Peer Kröger, Jörg Sander, and Arthur Zimek. 2011. Density-based clustering. *Wiley Interdiscip. Rev.: Data Mining Knowl. Discov.* 1, 3 (2011), 231–240.
[34] Jing Lu, Yuhai Zhao, Kian-Lee Tan, and Zhengkui Wang. 2022. Distributed density peaks clustering revisited. *IEEE Trans. Knowl. Data Eng.* 34, 8 (2022), 3714–3726.
[35] 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. *Scient. Rep.* 7 (2017), 45602.
[36] Miao Qiao, Junhao Gan, and Yufei Tao. 2016. Range thresholding on streams. In *SIGMOD*. 571–582.
[37] Zafaryab Rasool, Rui Zhou, Lu Chen, Chengfei Liu, and Jiajie Xu. 2022. Index-based solutions for efficient density peak clustering. *IEEE Trans. Knowl. Data Eng.* 34, 5 (2022), 2212–2226.
[38] Alex Rodriguez and Alessandro Laio. 2014. Clustering by fast search and find of density peaks. *Science* 344, 6191 (2014), 1492–1496.
[39] Hwanjun Song and Jae-Gil Lee. 2018. RP-DBSCAN: A superfast parallel DBSCAN algorithm based on random partitioning. In *SIGMOD*. 1173–1187.
[40] Liudmila Ulanova, Nurjahan Begum, Mohammad Shokoohi-Yekta, and Eamonn Keogh. 2016. Clustering in the face of fast changing streams. In *SDM*. 1–9.
[41] Li Wan, Wee Keong Ng, Xuan Hong Dang, Philip S. Yu, and Kuan Zhang. 2009. Density-based clustering of data streams at multiple resolutions. *ACM Trans. Knowl. Discov. Data* 3, 3 (2009), 1–28.
[42] Guangtao Wang and Qinbao Song. 2016. Automatic clustering via outward statistical testing on density metrics. *IEEE Trans. Knowl. Data Eng.* 28, 8 (2016), 1971–1985.
[43] Wenguan Wang, Jianbing Shen, Fatih Porikli, and Ruigang Yang. 2018. Semi-supervised video object segmentation with super-trajectories. *IEEE Trans. Pattern Anal. Mach. Intell.* 41, 4 (2018), 985–998.
[44] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2019. Leveraging set relations in exact and dynamic set similarity join. *VLDB J.* 28, 2 (2019), 267–292.
[45] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-efficient and practical parallel DBSCAN. In *SIGMOD*. 2555–2571.
[46] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. 2021. Fast parallel algorithms for euclidean minimum spanning tree and hierarchical spatial clustering. In *SIGMOD*. 1982–1995.
[47] Shuai Yang, Xipeng Shen, and Min Chi. 2019. Streamline density peak clustering for practical adoptions. In *CIKM*. 49–58.

[48] Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. 2011. Driving with knowledge from the physical world. In *SIGKDD*. 316–324.
[49] Yanfeng Zhang, Shimin Chen, and Ge Yu. 2016. Efficient distributed density peaks for clustering large data sets in mapreduce. *IEEE Trans. Knowl. Data Eng.* 28, 12 (2016), 3218–3230.
[50] Yu Zhang, Kanat Tangwongsan, and Srikanta Tirthapura. 2017. Streaming k-means clustering with fast queries. In *ICDE*. 449–460.
[51] Yang Zhang, Yunqing Xia, Yi Liu, and Wenmin Wang. 2015. Clustering sentences with density peaks for multi-document summarization. In *NAACL-HLT*. 1262–1267.
[52] Alaettin Zubaroğlu and Volkan Atalay. 2021. Data stream clustering: A review. *Artif. Intell. Rev.* 54, 2 (2021), 1201–1236.

Received 16 January 2023; revised 21 May 2023; accepted 5 July 2023