Streamline Density Peak Clustering for Practical Adoptions

Shuai Yang  
North Carolina State University  
Raleigh, North Carolina  
syang16@ncsu.edu

Xipeng Shen  
North Carolina State University  
Raleigh, North Carolina  
xshen5@ncsu.edu

Min Chi  
North Carolina State University  
Raleigh, North Carolina  
mchi@ncsu.edu

ABSTRACT
Since Density Peak Clustering (DPC) algorithm was proposed in 2014, it has drawn lots of interest in various domains. As a clustering method, DPC features superior generality, robustness, flexibility and simplicity. There are however two main roadblocks for its practical adoptions, both centered around the selection of cutoff distance, the single critical hyperparameter of DPC. This work proposes an improved algorithm named Streamlined Density Peak Clustering (SDPC). SDPC speeds up DPC executions on a sequence of cutoff distances by 2-8.8X while at the same time reducing memory usage by a magnitude. As an algorithm preserving the original semantic of DPC, SDPC offers an efficient and scalable drop-in replacement of DPC for data clustering.

CCS CONCEPTS
• Information systems → Clustering; • Theory of computation → Unsupervised learning and clustering.

KEYWORDS
density clustering, algorithm optimization, hyperparameter tuning

ACM Reference Format:

1 INTRODUCTION
Density Peak Clustering (DPC) is a general density-based clustering algorithm proposed in 2014 [23]. Compared to traditional clustering methods, DPC excels at multiple aspects: (1) Generality: Unlike some other clustering methods (e.g., K-Means), DPC applies to arbitrary data distributions and cluster shapes; (2) Robustness: it carries a built-in mechanism that detects and removes outliers; (3) Flexibility: no prior knowledge on the number of clusters is needed.

For these attractive properties, since its proposal, DPC has drawn interest in various domains, from facial images clustering [23], to image segmentation [5], text clustering [18] and so on. It has shown better clustering quality over other clustering algorithms [24], such as K-means [20], DBSCAN [11], Affinity Propagation [12].

Two main roadblocks however have been limiting practical adoptions of DPC: both center around cutoff distance, the single but critical hyperparameter in DPC. Cutoff distance, denoted as \(d_c\), defines the range of neighborhood when the density of a point is computed. Data points within the \(d_c\) radius of the concerned data point are called its neighbors. The number of neighbors is the density of that data point. The value of \(d_c\) thus determines density computation—the key measure DPC relies on—and hence the quality of the final clusters. The influence is dramatic. As Figure 1 shows, a tiny change in \(d_c\) can cause large changes in the clustering results, which echoes the prior reported observations on the unstableness of clustering quality due to \(d_c\) values [25].

Roadblock-I: Temporal Delays. For the sensitivity of DPC on \(d_c\), when applying DPC, users need to carefully select the appropriate values for \(d_c\). As there are no well established methods for computing the best \(d_c\) value, iterative methods are typically used, in which, the practitioner runs DPC many times with a different \(d_c\) value used each time, and adopts the best among all the trials (as per Adjusted Rand Index(ARI), Davies-Bouldin index or other cluster quality measures) as the final clustering results. The delay in producing the clustering results is the first roadblock for practical DPC adoptions.

Roadblock-II: Spatial scalability. There can be many possible values of \(d_c\), arbitrarily selecting the values to try may waste much time. The original DPC design [23], based on empirical observations, suggests to focus \(d_c\) choices to a range such that the average number of neighbors of a data point is 1–2% of the total number of data points. To determine that range, the design requires the use of \(O(N^2)\) memory space to store all the pairwise distances (\(N\) is the number of data points), and keep them in memory throughout the whole process, because these distances could be requested by the subsequent stages of DPC. The large space overhead makes DPC hard to apply to large datasets, forming the other main roadblock for practical adoptions of DPC.

There have been some prior efforts trying to alleviate the issues. They fall into two main categories. (1) The first category skips the selection of \(d_c\): Wang and others [25] propose to directly estimate the density via nonparametric multivariate kernel density estimation. Mehmood and others [21] further refine the kernel density estimation with heat diffusion method. Other work uses KNN instead of \(d_c\) to identify neighbors [10]. All of these approaches, however, change the definition of density in the original DPC and hence fall short in maintaining the semantic of DPC as well as the clustering results. (2) The second category directly estimates the optimal \(d_c\): Field Theory in physics is introduced to model the density as potential in data field [26]. It estimates the optimal \(d_c\) through minimizing potential entropy of datasets. Gao and others [13] take...
into account the degree of dispersion of a dataset when determining the optimal \( d_c \) and derive a formula for \( d_c \) based on the standard deviations of each dimension. Zhang and others [30] propose to use distributed systems for DPC and estimate \( d_c \) via Reservoir Sampling. Although these efforts are valuable attempts, none of them can offer the guarantee that the resulting quality of the density and hence clustering results are not compromised compared to those by the original DPC.

The goal of this work is to provide a drop-in replacement of the original DPC with significantly enhanced speed and scalability, especially when the best cutoff distance needs to be empirically determined. We achieve the goal by distinctively focusing on computation reuse and stream-style data processing. The drop-in replacement is named Streamlined DPC (SDPC). SDPC reduces the memory cost of DPC by a magnitude through an incremental algorithm named double-buffer incremental introselect, and meanwhile, speeds up DPC executions on a sequence of \( d_c \) values by 2.2X-8.8X by harnessing information latent in previous iterations.

Inspired by studies for other algorithms [28, 29], unlike prior methods, our approach keeps DPC semantics unchanged — that is, keeping its clustering results the same as the original algorithm gives. This property guarantees no quality loss in whatever problem settings, and hence preserves all the appealing properties of the original DPC and all the confidence that users have built through the many applications of DPC. Although designed primarily to replace the original DPC, the techniques in SDPC can also benefit other variants of DPC by, for instance, speeding up the search for \( d_c \) in a range around the value estimated with previously proposed methods. To our knowledge, SDPC is the first drop-in replacement of the original DPC that preserves its semantic with substantially improved efficiency and scalability. By removing the two major roadblocks, it helps better materialize the full potential of DPC in practical usage.

2 BACKGROUND AND NOTATIONS

For reference, we put all notations used in this paper into Table 1.

![Table 1: Notations](image)

<table>
<thead>
<tr>
<th>Notations</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_{c,i} )</td>
<td>The ( i )th smallest cut-off distance</td>
</tr>
<tr>
<td>( x_i )</td>
<td>The ( i )th data point</td>
</tr>
<tr>
<td>( \rho_i )</td>
<td>Density of ( x_i )</td>
</tr>
<tr>
<td>( \delta_i )</td>
<td>Distance from ( x_i ) to its nearest data point of higher ( \rho )</td>
</tr>
<tr>
<td>( d_{i,j} )</td>
<td>Distance between ( x_i ) and ( x_j )</td>
</tr>
<tr>
<td>( N )</td>
<td>Total number of data points</td>
</tr>
<tr>
<td>( n )</td>
<td>Total number of pairwise distances between data points</td>
</tr>
<tr>
<td>( q_i )</td>
<td>Index of data point with the ( i )th highest ( \rho )</td>
</tr>
<tr>
<td>( q_i^c )</td>
<td>Index of data point with the ( i )th currently highest ( \rho )</td>
</tr>
<tr>
<td>( n_i )</td>
<td>Index of ( x_i )'s nearest data point of higher ( \rho )</td>
</tr>
<tr>
<td>( C_i )</td>
<td>The ( i )th cluster</td>
</tr>
<tr>
<td>( \rho_i^b )</td>
<td>Density threshold for the border region of ( C_i )</td>
</tr>
<tr>
<td>( c_l )</td>
<td>Cluster label of ( x_i )</td>
</tr>
<tr>
<td>( d_i^c )</td>
<td>The ( i )th smallest distance among all pairwise distances between data points</td>
</tr>
<tr>
<td>( p_i )</td>
<td>Index of ( d_i^c ) in the sorted distance list</td>
</tr>
<tr>
<td>( n_{b,i} )</td>
<td>Neighbor list of ( x_i )</td>
</tr>
<tr>
<td>( nbS_i )</td>
<td>List of neighbor sizes of ( x_i ) under different ( d_c )</td>
</tr>
<tr>
<td>( \rho_{i,j} )</td>
<td>Average density of ( p_i ) and ( p_j )</td>
</tr>
</tbody>
</table>

![Figure 2: The density of a data point is the number of data points that are closer than \( d_c \) to it. The number on each data point shows the density of it.](image)

DPC is based on the observation that every cluster center is surrounded by neighbors of lower density, while it has a relatively long distance to any other data points of higher density. There are two key concepts in DPC: (1) \( \rho \): density and (2) \( \delta \): distance to the nearest data point of higher density.

2.1 Density (\( \rho \))

**Definition 2.1.** The Density \( \rho_i \) is the number of data points that are closer than \( d_c \) to the data point \( x_i \). As shown in Figure 2:

\[
\rho_i = \sum_{j \in I_S(i)} \chi(d_{i,j} - d_c)
\]

where if \( x < 0 \), \( \chi(x) = 1 \), otherwise, \( \chi(x) = 0 \). \( d_c \) is the cutoff distance and \( d_{i,j} \) is the distance between data points \( x_i \) and \( x_j \). \( I_S \) is the index set of all data points in the dataset.
2.2 Distance to the Nearest Data Point of Higher Density ($\delta$)

Definition 2.2. $\delta_i$ is the distance from data point $x_i$ to the nearest data point with a higher density:

$$\delta_i = \begin{cases} 
\min_{j < i} \{d_{q_j - q_i}\}, & i \geq 2 \\
\max_{j > i} \{\delta_{q_j}\}, & i = 1 
\end{cases}$$

(2)

where $q_i$ is the index of the data point with the $i$th highest $\rho$. If two data points have the same $\rho$, they can be ordered by their original order in dataset. For the highest density point $x_{q_1}$, $\delta_{q_1}$ is defined as the maximum of all other $\delta$’s. That is to guarantee that $x_{q_1}$ is one of the cluster centers, which we will elaborate later.

Since cluster centers are supposed to have both large $\rho$ and $\delta$, as the example shows in Figure 2, if we draw a graph using $\rho$ as $x$-axis and $\delta$ as $y$-axis, the cluster centers are outliers located at the upper right of the graph. This graph is called decision graph which serves to identify cluster centers. Figure 3 is the decision graph for the example shown in Figure 2. It’s easy to identify two cluster centers at upper right which correspond to the two data points with density 6 in Figure 2. Decision graph also helps unravel noise. Data points at the upper left corner are likely to be noise due to their low density and large distance to other high $\rho$ data points. The cluster centers are hence picked as the $K$ data points that have the largest products ($\rho \cdot \delta$). $K$ can be determined automatically or specified by users[23].

2.3 Cluster Halo and Cluster Core

Besides the clustering results, DPC further divides all the data points into two categories: cluster core (robust assignment) and cluster halo (noise). For each cluster, DPC defines a border region, which includes data points in that cluster that are meanwhile less than $d_c$ away from data points in other clusters. To tell cores from halos, a density threshold is computed for each cluster as follows:

$$\rho_i^k = \max_{x_i \in C_i} \{ \max_{x_j \in C_i \land d_{x_i, x_j} < d_c} \left( \frac{\rho_i + \rho_j}{2} \right) \}$$

(3)

Data points with density larger than the threshold form the cluster core, and other data points in the cluster form its halo.

3 IMPLEMENTATION OF DPC

Rodriguez and Laio have provided an implementation of DPC in the supplementary materials of their original DPC paper [23]. This section briefly describes this implementation as well as its speed and space bottlenecks. At a high level, the algorithm consists of three steps as outlined by the left side chart in Figure 4.

Stage I: Getting the value of $d_c$. The first stage is to determine the value of $d_c$. As a rule of thumb, $d_c$ is chosen so that the average number of neighbors (i.e. data points with distance less than $d_c$ to the concerned data point) is around 1-2% of the total number of data points in the dataset. To get such a $d_c$, DPC sorts all the pairwise distances of all data points in ascending order and picks the value at the position of $\rho \cdot n$, where $\rho$ is specified by user and usually $\in [1\%, 2\%]$.

This implementation is straightforward, but it is both time and memory consuming. In this paper, we use $N$ to represent the total number of data points in a dataset and $n$ for the total number of pairwise distances. Hence, $n = (N(N-1))/2 \in O(N^2)$. Therefore, sorting all the distance takes $O(N^2 \log N^2)$ time and the algorithm requires $O(N^2)$ memory space. When $N$ is large, this stage becomes the bottleneck in both time and space. The limitation is severe. A machine with 40GB memory cannot run DPC to cluster a dataset with only 58000 data points (the Shuttle dataset [8]), as just storing the 1.68 billion pairwise distances would take 25GB memory. It seriously hampers the scalability of DPC.

Stage II: Computing $\rho$ and $\delta$ of each data point. With the $d_c$ value picked in Stage I, for each data point $x_i$, Stage II of DPC iterates all other data points and counts the number of data points with distance less than $d_c$ to $x_i$, which takes $O(N^2)$ time. To compute $\delta$, DPC sorts all the data points in terms of their density in descending order. Then, the process of computing $\delta$ is to locate the nearest data point in front of the concerned data point in that sorted list. The sorting takes $O(N \log N)$ and getting all $\delta$ in the sorted list costs $O(N^2)$. After that, computing $\rho \cdot \delta$ and pinpointing cluster centers with the $k$ largest values take linear time.

Stage III: Cluster assignment and identifying cluster halo. Every data point is assigned to the same cluster as its nearest data point of higher density, which takes $O(N)$ time. Let $n_i$ $(i > 1)$ be the index of $x_i$’s nearest data point with a higher density. Thus $x_{n_i}$ is $x_i$’s nearest data point of higher density. The process follows descending order of density. Because only in that order, we can guarantee the cluster label of $x_{n_i}$ has been already determined before $x_i$.

DPC adopts a straightforward approach to identify the cluster halo. It iterates all pairs of data points. As long as two data points come from different clusters and their distance is less than $d_c$, the average density of two data points are used to update the density threshold for border regions of both clusters. When the density threshold is determined, all data points with density less than the threshold of the cluster they belong to are marked as cluster halo. The time complexity of this stage is also $O(N^2)$.

To summarize, based on the preceding analysis of DPC, sorting all distances to get the value of $d_c$ is the most computationally expensive. Since all distances remain unchanged during the tuning process, it’s possible to reuse sorted distances so that sorting...
4 REUSE-CENTERED OPTIMIZATIONS

The speed enhancement by SDPC centers around computation reuse and avoidance of unnecessary computations in the tuning process. It consists of six optimizations, which streamline all the three stages of DPC, as outlined on the right side of Figure 4. We next explain the design of each of the optimizations in SDPC.

4.1 Calculate $d_c$ via Introselect and ordering

In the previous section, we have noticed the process of getting the value of $d_c$ entails large overhead. It involves sorting all distances. Our target is to find the distance value at a specific position (e.g. 2% · $n$) of the sorted distance list. This problem is equivalent to "find the kth smallest element in an unsorted list". It is not necessary to sort the entire dataset to get that value. There are some well-known algorithms that can solve this problem in linear time, such as Quickselect [15], Median of Medians [4] and Introselect [22].

Introselect is a hybrid of Quickselect and Median of Medians, combining their advantages. Quickselect has fast average performance, so Introselect starts with Quickselect: Every time it chooses a pivot and partitions elements into two parts, depending on whether their values are less than or greater than the pivot, then it works recursively on the side that has the element it looks for, as shown in Figure 5. But when it does not make sufficient progress, Introselect switches to Median of Medians which divides elements into sublists and gets the median of each sublist. Then it uses the median of these medians as the pivot to partition elements. Median of Medians guarantees linear worst-case performance.

SDPC adopts the Introselect to attain the value of $d_c$ in linear time. If $d_c$ corresponds to the $(p \cdot n)$th smallest value of all pairwise distances, after applying Introselect, $d_c$ is at the position $p \cdot n$ in the list of all pairwise distances. Besides that, distances are partitioned into two parts by Introselect. Distances less than $d_c$ are in front of $d_c$ while distances larger than $d_c$ are behind it, but distances in a partition are not sorted.

In the process of tuning, suppose that the range of interest for $d_c$ is 1-2% of the total number of data points. If the step size is 0.1%, the choices of $d_c$ are those where the average number of neighbors is 1%, 1.1%, 1.2%,...,2% of all data points. In another word, the values of $d_c$ are $d^{[1\%\cdot n]}_i$, $d^{[1.1\%\cdot n]}_i$, $d^{[1.2\%\cdot n]}_i$, ..., $d^{[2\%\cdot n]}_i$, respectively, where $d^{[x\%\cdot n]}_i$ is the $x$th smallest value of all pairwise distances. The challenge is how to determine these values efficiently. One straightforward way is to sort all distances and then get the value of each $d_c$. However, sorting all the distances easily outweighs the benefits it may bring. As aforementioned, Introselect is applied to get the value of $d_c$ fast. SDPC further optimizes the order in exploring $d_c$ choices to reduce computations during repeated runs of Introselect for each $d_c$, as described in the following optimization.

**Optimization Strategy 1:** Assume there are m choices for $d_c$: $d_{c_1} < d_{c_2} < d_{c_3} < ... < d_{c_m}$, the execution order of Introselect takes the descending order of $d_c$.

Let’s assume the corresponding position of $d_c$ in the sorted distance list is $p_i$; based on the definition of $d_{c_i}$, it is clear that
When SDPC finds \( d_{c_i} \) through Introselect, we know \( d_{c_1}, d_{c_2}, \ldots, d_{c_{m-1}} \) are all located in front of \( d_{c_m} \) that distance list. Therefore, SDPC capitalizes on this observation to get speedups. Optimization Strategy 2: After the exploration moves from \( d_{c_{k-1}} \) to \( d_{c_k} \), when updating \( \rho \), it needs to consider only the distances in the interval between \( p_k \) and \( p_{k+1} \) in the distance list. The operations are to increase the corresponding two end points' \( \rho \) value by one for each distance falling into that interval. Optimization Strategy 3 proposes a new way to update \( \delta \) during the process of restoring the order of \( \rho \), which could largely reduce the number of recomputations of \( \delta \); Optimization Strategies 4 and 5 accelerate remaining recomputations.

1) Updating \( \rho \): When a new \( d_c \) is applied, \( \rho \) of each data point could be changed accordingly. SDPC leverages the results that have already been computed in the trials of some earlier choices of \( d_c \) to avoid unnecessary computations. For two consecutive trials of SDPC on \( d_{c_{k-1}} \) and \( d_{c_{k+1}} \), where \( d_{c_k} < d_{c_{k+1}} \), densities of data point \( x_i \) under two trials are \( \rho_i^{k} \) and \( \rho_i^{k+1} \), respectively. We can regard \( \rho_i^{k+1} \) as being composed of two parts of data points: data points whose distances from \( x_i \) are less than \( d_{c_{k}} \), and those between \( d_{c_{k}} \) and \( d_{c_{k+1}} \). Note that data points in the first part have already been counted in \( \rho_i^{k} \), so only those with distances between \( d_{c_{k}} \) and \( d_{c_{k+1}} \) are needed to be checked.

The crux of the problem becomes how to get the number of data points in the second part efficiently. Thanks to the use of Introselect, SDPC has divided all distances into a number of ordered partitions. Distances less than \( d_{c_{k}} \) and \( d_{c_{k+1}} \) are all in front of \( p_k \) and \( p_{k+1} \), respectively, where \( p_k \) and \( p_{k+1} \) are corresponding positions of \( d_{c_{k}} \) and \( d_{c_{k+1}} \), in the sorted distance list. The distances belong to \( \{d_{c_k}, d_{c_{k+1}}\} \) are all located in the interval between \( p_k \) and \( p_{k+1} \). The increment of \( \rho \) relates with only the distances belonging to that interval. SDPC capitalizes on this observation to get speedsup.

Optimization Strategy 1 helps SDPC attain the values of all \( d_c \) fast. After getting all \( d_c \) values, we can then test each \( d_c \) in ascending order.

4.2 Update \( \rho \) and \( \delta \) via computation reuse

This section describes optimizations involved in updating densities \( \rho \) and distances to the nearest neighbor with a higher density \( \delta \). When \( d_c \) changes, \( \rho \) changes accordingly. The sorted list of data points (ordered on \( \rho \)) from the previous trial becomes out of order. Restoring the order is needed for updating \( \delta \). Therefore, this stage can be subdivided into two parts: (1) updating \( \rho \); (2) restoring the order of \( \rho \) and updating \( \delta \). Optimization Strategy 2 is for the first part while Optimization Strategies 3 to 5 are all for the second part, among which, Optimization Strategy 3 proposes a new way to update \( \delta \) during the process of restoring the order of \( \rho \), which could largely reduce the number of recomputations of \( \delta \); Optimization Strategies 4 and 5 accelerate remaining recomputations.
Specifically, SDPC goes from the beginning of the data points list (ordered on \( \rho \) in the previous trial of \( d_c \)). When it encounters an out-of-order data point, it swaps the involved data points to fix the order and at the same time updates \( \delta \). For instance, suppose that the data point it encounters is \( x_j \), the data point before it is \( x_i \), and \( \rho_i < \rho_j \). SDPC swaps \( x_j \) and \( x_i \) to fix the order problem. After swapping, it uses Optimization Strategy 3 to update \( \delta_i \) and \( \delta_j \). SDPC then continues to work on the rest of the list in this manner. Figure 7 illustrates how Optimization Strategy 3 works.

**Optimization Strategy 3:** Let \( x_i \) and \( x_j \) be two adjacent data points in the sorted list of data points with \( \rho \) mis-ordered (\( \rho_i < \rho_j \)). The updated \( \delta_i \) and \( n_j \) (the index of \( x_j \)'s nearest data point of higher \( \rho \)) are as follows:

\[
\delta_i = \min(\delta_i, d_{i,j}) \tag{5}
\]

\[
n_j = \begin{cases} n_i, & d_{i,j} \geq \delta_i \\ j, & \text{otherwise} \end{cases} \tag{6}
\]

and the updated \( \delta_j \) and \( n_j \) are as follows:

\[
\delta_j = \begin{cases} 
\min_{i < s} (d_{q_i^j, q_i^s}), & n_j = i \text{ and } s \geq 2 \\
 n_j, & n_j \neq i \text{ and } s \geq 2 
\end{cases} \tag{7}
\]

\[
n_j = \arg\min_{i < s} (d_{q_i^j, q_i^s}), \quad n_j = i \text{ and } s \geq 2 \tag{8}
\]

where \( s \) is the determined position of \( x_j \) which we will explain later, \( q_i^j \) is the index of data point with the currently \( i \)th highest \( \rho \). We say 'currently' because the update process is still in progress. Note that \( s = 1 \) is a special case, we do not update it right now. Because there are two possibilities: (1) the final position of \( x_j \) is 1, \( \delta_j \) is then assigned with the maximum value of all \( \delta \) after all \( \delta \) are updated; (2) the final position is not 1, it will be swapped with other data points and then Formula 5 and 6 are applied to update \( x_j \).

This optimization strategy is easy to understand. For \( x_i \), after the swapping, it now has one more data point of higher density. Therefore, comparing the current minimum with the distance to the new point of higher density leads to the new minimum. As for \( x_j \), after swapping, it has one fewer point of higher density. There could be two different cases. (1) \( x_j \) is not \( x_n \). In this case, we do not need to update \( \delta_j \) and \( n_j \). Because we know \( x_n \) is still in front of it. (2) \( x_j \) is \( x_n \). In this case, because \( x_j \) is no longer a data point of higher density of \( x_j \), recomputing the \( \delta_j \) and \( n_j \) is unavoidable. But the recomputation is not executed immediately, as the position of \( x_j \) has not been determined and it could be further swapped with elements in front. Once the position of \( x_j \) is determined, we recompute \( \delta_j \) and \( n_j \).

Through Optimization Strategy 3, most of recomputations in DPC can be avoided, which brings significant speedups. But as we have mentioned, there are still remaining cases in which recomputations have to be done. For these cases, SDPC uses an early-stop scheme to accelerate recomputations, as explained next.

For every data point \( x_i \), SDPC maintains a list of all its neighbors \( nb_i \). Suppose that we have \( m \) choices of \( d_c \), and \( d_{c_1} < d_{c_2} < \ldots < d_{c_m} \). Firstly, SDPC executes with \( d_{c_1} \), \( nb_i \) adds all neighbors within \( d_{c_1} \) and forms the first partition. When applying \( d_{c_2} \), neighbors with distances between \( [d_{c_1}, d_{c_2}] \) are added to \( nb_i \) and form the second partition. The same goes with the other \( d_c \) choices. Hence, \( nb_i \) is divided into multiple partitions. These partitions are in order, even though the data points inside each partition are not. Based on this property, we propose the fourth Optimization Strategy.

![Figure 7: Example of the process of updating \( \delta \). (a) Go from the beginning to find the first out-of-order element, which is \( x_4 \) in this example; (b) Swap \( x_4 \) with \( x_3 \), assume \( d_{4,5} = 210 < \delta_5 \), \( \delta_5 \) and \( n_5 \) are hence updated. On the other side, as \( n_4 \neq 5 \), no need to change \( n_4 \) or \( \delta_4 \); (c) Continue to check \( x_4 \) with the element in front of it, since \( \rho_3 < \rho_4 \), they are out of order; (d) Swap \( x_3 \) with \( x_3 \), assume \( d_{3,4} = 365 > \delta_3 \), hence no update to \( \delta_3 \) or \( n_3 \). However, as \( n_4 = 3 \), \( n_4 \) and \( \delta_4 \) need to be recomputed. But the recomputation is not executed immediately, since the position of \( x_4 \) has not been determined. It's just marked (in red) to indicate that recomputation is needed; (e) Continue to check \( x_4 \) with \( x_7 \). They are in correct order; the position of \( x_4 \) is now determined. (f) Recomputation of \( n_4 \) and \( \delta_4 \) is executed.](image-url)
Optimization Strategy 4: For data point $x_i$, if one data point of a higher density is found in the $j$th partition of $nb_i$, then SDPC skips the subsequent partitions $k(k > j)$ in updating $\delta_i$ and $n_i$.

Proof. Let $P_j$ and $P_k$ be the set of indices of data points within the $j$th and $k$th partition of $nb_i$, respectively, where $k > j$.

$\forall y \in P_j, d_{x_i, x_y} \leq \min_{z \in P_k} d_{x_i, x_z}$.

$\exists y' \in \{y | \rho_{y'} > \rho_i, y \in P_j\}$,

$x_{y'}$ is a data point of higher density of $x_i$.

$\therefore d_{x_i, x_{y'}} \leq \min_{z \in P_k} d_{x_i, x_z}$.

$\therefore n_i$ must not be in $P_k$.

Note that SDPC can not stop immediately when it finds a data point of a higher density in partition $j$ of $nb_i$, because data points in a partition are not ordered; some other data points in the partition may also meet the condition and are even closer to $x_i$.

Furthermore, it is not necessary to start searching from the very beginning of $nb_i$. The trial of a previous $d_c$ value already provides some information that can help skip partitions that are impossible to contain new $x_{n_i}$, as the following optimization strategy exploits.

Optimization Strategy 5: For data point $x_i$, assume that $x_{n_i}$, in the previous trial of $d_c$, is in the $j$th partition of $nb_i$; in updating $\delta_i$ and $n_i$, SDPC skips data points in partition $h$, for all $h < j$, as they cannot be the new $x_{n_i}$.

(Note: when updating $\delta_i$ and $n_i$, data points after $x_i$ have not been processed. This new $x_{n_i}$ may not be the final one, it could be further updated later)

Proof. Let $P_j$ be the set of indices of data points within the $j$th partition of $nb_i$.

$\therefore$ In the previous trial of $d_c$, $n_i \in P_j$.

Assume $y \in P_h(h < j)$, we have $\rho_y < \rho_i$, otherwise $n_i = y \in P_k$.

$\therefore x_y$ must be positioned after $x_i$ in the list ordered by $\rho$.

Recall the Optimization Strategy 3 and Figure 7, the updating process starts from the beginning of that list.

When updating $x_i$, $x_y$ is still untouched. New $x_{n_i}$ comes from the data points currently in front of $x_i$.

$\therefore x_y$ will not be the new $x_{n_i}$.

Optimization Strategies 4 & 5 together can help accelerate the computations in updating $\rho$ and $\delta$. It’s worth noting that if we keep $nb_i$ sorted by their distance to $x_i$, it will be easier to identify the new nearest data point of higher $\rho$. We have explored this approach, but the overhead introduced by maintaining a sorted neighbor list outweighs the benefit.

4.3 Identify halos via computation skipping

Cluster core is the part with high confidence while cluster halo is the border region which DPC is less certain about. The separation provides users with more information about the clustering results.

The key step in identifying cluster halo is to determine the density threshold which is defined in Formula 3. This step could be time-consuming since it involves enumerating all pairs of data points. SDPC manages to accelerate this step which is the performance bottleneck, thereby speeds up the whole process.

According to the definition of density threshold, the computation only happens between data points and their neighbors (data points within $d_c$ distance to the concerned point). Since SDPC maintains a list of neighbors for each data point, it only needs to go through that list, reducing the number of computations from $\frac{N(N-1)}{2}$ to $cN$, where $c$ is the average number of neighbors, typically falling in the range of [1% · $N$, 2% · $N$].

If $x_i$ and $x_j$ are neighbors, their average density $\bar{\rho}_{i,j}$ is used to update the density thresholds of clusters that each of them belongs to. $\bar{\rho}_{i,j}$ will be computed when checking $x_i$ with $nb_i$, as well as checking $x_j$ with $nb_j$. To avoid recomputing $\bar{\rho}_{i,j}$, SDPC employs a rule that $\bar{\rho}_{i,j}$ is computed only once when this current data point’s density is less than or equal to the other one’s. Based on that, SDPC employs optimization strategy 6 to further reduce computations.

Optimization Strategy 6: During updating density thresholds, for a data point $x_i$, if $\delta_i > d_c$, then SDPC skips $x_i$ and continues to the next data point.

The rationale is straightforward. Condition $\delta_i > d_c$ means the distance to the nearest data point with higher density is larger than $d_c$. That is to say $x_i$ has no neighbor whose density is higher than its own. Since computations of updating density thresholds happen only between a data point and its neighbors of a higher density, SDPC can hence skip $x_i$, avoiding examining its neighbors.

5 ENABLE SPATIAL SCALABILITY

Recall that the original DPC uses $O(N^2)$ to store all pairwise distances and sorts them in its calculation of a $d_c$ value, which forms the roadblock for DPC scalability. In our optimized algorithm described in Section 4.1, by using Introselect, it circumvents sorting all pairwise distances. But it still needs to store all pairwise distances stored in memory. This section describes our solution to the problem. Our solution is a double-buffer incremental introselect. It employs two buffers. Suppose that we are looking for the $k$th smallest distance. The first buffer is of size $k$, and the second buffer is of size $\alpha k, (\alpha > 0)$. In our experiments, it is set to 1. The first buffer keeps the current $k$ smallest distances; the second buffer stores new data.

Data elements are read into the buffers one after one. Every time when the buffer is full, Introselect is applied to the concatenation of the two buffers to find the current $k$th smallest distance. As a consequence of Introselect, all distances less than the current $k$th distance must reside in front of it, and hence the first buffer now has the current $k$ smallest distances even though they are not in order. Distances in the second buffer are discarded, since none of them are possible to be part of the global $k$ smallest distances. New distances are loaded into the second buffer to overwrite those discarded distances. When the buffer gets full, we run Introselect again. This procedure repeats until all distances have been processed. The distances in the first buffer are now the global $k$ smallest distances.

The memory consumption reduces from $\frac{N(N-1)}{2}$ units (one distance per unit) to $(1 + \alpha k)k$ is the position of the largest $d_c$ in the sorted distance list, which equals to $\frac{2\alpha N(N-1)}{2}$ as the original DPC work [23] suggests. When $\alpha = 1$ (as used in our experiments), the memory consumption equals $2\% N(N - 1)$. While it involves Introselect $O(\frac{N}{k})$ times, every time the Introselect works on a much smaller buffer of size $(1 + \alpha k)k$. The total time still remains linear.

One extra optimization is that each time when a new distance is loaded into the second buffer, SDPC compares the new distance $d_{new}$ with the current $k$th smallest distance. If $d_{new}$ is larger, it is
immediately discarded. Because it is impossible to be one of the global $k$ smallest distances. As the algorithm proceeds, the current $k$th smallest distance becomes closer and closer to the global $k$th smallest distance, the optimization becomes increasingly effective.

Note that in the other stages of SDPC, likewise, rather than keeping all pairwise distances in memory like what DPC does, SDPC keeps only distances less than $d_c$ in memory. They cover most of distances required in the following stages. The only exception is that when SDPC updates $\delta$ for a data point $x$, its nearest data point with a higher density could be more than $d_c$ away from $x$. In that case, SDPC computes their distance on-the-fly, which are uncommon in our experiments, on average only 7% distances fall into this category.

6 EXPERIMENTS

We evaluate the efficiency of SDPC on a variety of benchmark datasets as shown in Table 2. Dataset Elly_2d10c13a [1], is an artificial dataset with 10 highly overlapping clusters. Fig2_PanelB is a dataset used in the DPC paper with artificially added noise from an uniform distribution. The following 5 real world datasets are collected from UCI Machine Learning Repository [8]. Wine Quality [7] aims to utilize physicochemical properties (e.g. citric acid and pH) to model the quality of wine. Anuran Calls [6] is a dataset of audio records for different frog calls. HTRU2 [19] collects emission signals of pulsars which are a type of star, of considerable scientific interest. Shuttle [8] dataset is used for extracting rules determining whether autolanding of a spacecraft is preferable. Skin Segmentation [2] contains RGB values of skin textures from images of people from diverse groups. The datasets are chosen to be diverse in both properties and the intended usage.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Dataset Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>num. of points</td>
<td>num. of distances</td>
</tr>
<tr>
<td>Elly_2d10c13a</td>
<td>2796</td>
</tr>
<tr>
<td>Fig2_PanelB</td>
<td>4000</td>
</tr>
<tr>
<td>Wine Quality</td>
<td>4898</td>
</tr>
<tr>
<td>Anuran Calls</td>
<td>7195</td>
</tr>
<tr>
<td>HTRU2</td>
<td>17898</td>
</tr>
<tr>
<td>Shuttle</td>
<td>58000</td>
</tr>
<tr>
<td>Skin Seg.</td>
<td>245057</td>
</tr>
</tbody>
</table>

Figure 8: Running time of DPC and SDPC on Skin Segmentation dataset and its subsets with 64 $d_c$. Subsets are composed of 60000, 80000, 100000, 120000, 140000 and 160000 data points

In the original implementation of DPC, it sorts all the distances at the very beginning. Since distances are not changed during the process of tuning, there is no need to re-sort them every time. We hence revised the original DPC implementation so that sorted distances are reused. We take this revised implementation as our baseline which is faster than directly running original DPC. We next report the comparisons between the baseline and SDPC, in terms of both end-to-end time, the time of each of the major stages as well as the memory cost.

End-to-End Time. Table 3 reports the running times on each dataset in various settings. The numbers in the parentheses show the speedups (X) over the DPC. Note that the speedups are calculated with original full precision time so it may be a little different from the results calculated from the running time in Table 3 (only one decimal digital kept). It’s clear to see the pattern that more $d_c$ corresponds to larger speedups. This is intuitive: More times of executions leave larger space for optimization strategies to take effect. As shown in the 4th column, the best end-to-end performance we have observed is on the Shuttle dataset, which attains 8.8 times speedups. For the Skin Segmentation dataset, the original DPC algorithm runs out of the memory, so it is not listed in Table 3. By exploring the executions on different subsets of the Skin Segmentation dataset, we probe the upper bound of dataset size the original DPC can handle. Subsets of different numbers of data points are collected whose size are: 60000, 80000, 100000, 120000, 140000 and 160000. Figure 8 shows that the largest number of data points DPC can deal with on the machine is 100000 while SDPC can process the whole dataset (245057 data points) with even less running time ($3.02 \times 10^3$ s vs $2.99 \times 10^3$ s).

In our experiments, the largest number of $d_c$ choices is 64, but in practice, with more $d_c$ being included, even larger speedups will be achieved. Small datasets tend to show smaller speedups, as there are less room for improvement. Overall, the results show consistent superior performance of SDPC, across datasets and across the numbers of $d_c$ choices. Next, we provide a detailed analysis of the results of each major stage and discuss some attained insights.
Table 3: Running Time Comparison of Different Algorithms

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$d_c$ #</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DPC</td>
<td>SDPC</td>
<td>DPC</td>
<td>SDPC</td>
<td>DPC</td>
<td>SDPC</td>
<td>DPC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Getting $d_c$</td>
<td>Computing $\rho$ and $\delta$</td>
<td>Identifying cluster halo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elly_2d10c13a</td>
<td>8</td>
<td>6.3</td>
<td>2.8</td>
<td>2.5</td>
<td>1.2</td>
<td>0.23</td>
<td>0.66</td>
<td>0.42</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>8.0</td>
<td>2.8</td>
<td>2.4</td>
<td>1.7</td>
<td>0.30</td>
<td>1.02</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>12.1</td>
<td>3.0</td>
<td>2.4</td>
<td>1.2</td>
<td>0.41</td>
<td>1.52</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>19.9</td>
<td>3.4</td>
<td>2.5</td>
<td>1.1</td>
<td>0.55</td>
<td>2.18</td>
<td>3.4</td>
</tr>
<tr>
<td>Fig2_PanelB</td>
<td>8</td>
<td>12.8</td>
<td>5.5</td>
<td>5.1</td>
<td>2.1</td>
<td>0.35</td>
<td>8.2</td>
<td>0.86</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16.2</td>
<td>5.7</td>
<td>5.1</td>
<td>4.7</td>
<td>4.7</td>
<td>12.3</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>23.8</td>
<td>6.1</td>
<td>5.3</td>
<td>11.5</td>
<td>0.58</td>
<td>19.7</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>41.0</td>
<td>6.7</td>
<td>5.3</td>
<td>24.1</td>
<td>0.87</td>
<td>27.5</td>
<td>7.2</td>
</tr>
<tr>
<td>Wine Quality</td>
<td>8</td>
<td>31.5</td>
<td>8.6</td>
<td>7.2</td>
<td>5.2</td>
<td>1.2</td>
<td>4.4</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>37.4</td>
<td>9.4</td>
<td>7.5</td>
<td>10.0</td>
<td>1.6</td>
<td>6.4</td>
<td>2.9</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>52.2</td>
<td>10.2</td>
<td>7.6</td>
<td>20.5</td>
<td>2.1</td>
<td>9.8</td>
<td>5.9</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>77.2</td>
<td>11.6</td>
<td>7.7</td>
<td>40.4</td>
<td>2.8</td>
<td>14.3</td>
<td>11.7</td>
</tr>
<tr>
<td>Anuran Calls</td>
<td>8</td>
<td>49.3</td>
<td>22.3</td>
<td>16.3</td>
<td>10.1</td>
<td>5.9</td>
<td>1.7</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>59.1</td>
<td>23.7</td>
<td>16.4</td>
<td>21.3</td>
<td>6.7</td>
<td>3.2</td>
<td>5.5</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>83.4</td>
<td>25.2</td>
<td>16.6</td>
<td>40.2</td>
<td>7.4</td>
<td>5.4</td>
<td>10.4</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>132.7</td>
<td>27.6</td>
<td>17.0</td>
<td>80.8</td>
<td>8.2</td>
<td>9.8</td>
<td>20.9</td>
</tr>
<tr>
<td>HTRU2</td>
<td>8</td>
<td>305.2</td>
<td>108.7</td>
<td>98.4</td>
<td>70.6</td>
<td>9.1</td>
<td>7.7</td>
<td>19.1</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>381.6</td>
<td>113.7</td>
<td>99.6</td>
<td>135.4</td>
<td>11.3</td>
<td>12.0</td>
<td>38.1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>573.4</td>
<td>119.8</td>
<td>100.5</td>
<td>280.9</td>
<td>14.1</td>
<td>19.9</td>
<td>80.4</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>948.8</td>
<td>132.2</td>
<td>103.9</td>
<td>574.1</td>
<td>18.0</td>
<td>31.9</td>
<td>162.8</td>
</tr>
<tr>
<td>Shuttle</td>
<td>8</td>
<td>3597.3</td>
<td>1186.9</td>
<td>107.1</td>
<td>913.1</td>
<td>95.1</td>
<td>9.6</td>
<td>224.9</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>5262.6</td>
<td>1203.7</td>
<td>1056.2</td>
<td>1828.8</td>
<td>104.7</td>
<td>17.5</td>
<td>451.7</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>7439.7</td>
<td>1278.1</td>
<td>1075.4</td>
<td>3487.4</td>
<td>116.1</td>
<td>30.1</td>
<td>892.4</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>12309.5</td>
<td>1404.5</td>
<td>1094.7</td>
<td>7309.2</td>
<td>134.9</td>
<td>54.2</td>
<td>1795.7</td>
</tr>
</tbody>
</table>

Table 4: Recomputations Comparison for Updating $\delta$.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of Recomputations of $\delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DPC</td>
</tr>
<tr>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>Elly_2d10c13a</td>
<td>176148</td>
</tr>
<tr>
<td>Fig2_PanelB</td>
<td>252000</td>
</tr>
<tr>
<td>Wine Quality</td>
<td>308574</td>
</tr>
<tr>
<td>Anuran Calls</td>
<td>453285</td>
</tr>
<tr>
<td>HTRU2</td>
<td>1127574</td>
</tr>
<tr>
<td>Shuttle</td>
<td>3654000</td>
</tr>
</tbody>
</table>

Speedups on getting $d_c$. As Table 3 shows, more $d_c$'s show slightly more speedups in this stage for a dataset, but the increment is not a lot. The reason is that this part includes the time spends on I/O operations. Even though SDPC does not need to keep all distances in the memory, it still needs to read all distances from the hard drive. The time spent on the I/O operations is a major part of the time. It is the same for the same dataset.

Avoided computations in updating $\rho$ and $\delta$. This stage updates $\rho$, the density, and $\delta$, the distance to the nearest data point of higher $\rho$. As Table 3 shows, in this stage DPC achieves significant speedups, especially when the number of $d_c$ gets larger. SDPC has proposed four Optimization Strategies for this process. Besides the running time speedups, Table 4 presents the number of recomputations of $\delta$ executed in DPC and SDPC with 64 $d_c$ choices. Recomputing $\delta$ is the most computationally intensive in this stage, because for each data point, it needs to iterate all data points with higher density. Optimization Strategy 3 manages to avoid this recomputation. The second and third columns in Table 4 show the total number of recomputations needed by DPC and SDPC, respectively. The number of recomputations in SDPC is on average 5.7% of that in DPC; 94.3% recomputations are avoided. For the remaining recomputations, Optimization Strategies 4 & 5 take effects to accelerate recomputations. Column four shows the number of recomputations done with Optimization Strategies 4 & 5. When Optimization 4 & 5 are not suitable, the original approach is used for updating $\delta$; the numbers of such recomputations are listed in column five. They are very small portions of the numbers in column 1. The synergy of all of these Optimization Strategies brings SDPC as much as 54.2X speedups (on Shuttle dataset) for this stage.

Speedups in identifying cluster halo. Speedups in this stage are only marginally affected by the change of the number of $d_c$. Our exploration shows that the speedups strongly correlate with the degree of overlap between clusters in the dataset (-0.83 Pearson correlation coefficient), which is intuitive given that this step mainly considers the overlapping areas.

Memory cost. Recall that our algorithm reduces the memory for storing distances from $N(N-1)/2$ units (one distance per unit) to $\frac{2N(N-1)}{d_c}$, independent of $d_c$. We hence report the memory cost in one $d_c$ setting ($d_c=64$). Table 5 reports the measured maximum resident set size in memory. As the dataset increases, the savings increase, until it reaches about 90%. The savings corresponding to subsets of Skin Segmentation dataset with 60000, 80000 and 100000 data points are 88.6%, 89.5% and 89.1%. This trend meets our
Table 5: Memory Cost Comparison
(The percentages in parentheses are the memory cost of SDPC compared with that in DPC.)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>DPC Memory Usage (MB)</th>
<th>SDPC Memory Usage (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elly_2d10c13a</td>
<td>96.4</td>
<td>17.1 (17.7%)</td>
</tr>
<tr>
<td>Fig2_PanelB</td>
<td>189.5</td>
<td>28.6 (15.1%)</td>
</tr>
<tr>
<td>Wine Quality</td>
<td>314.4</td>
<td>41.2 (13.1%)</td>
</tr>
<tr>
<td>Anuran Calls</td>
<td>653.4</td>
<td>82.5 (12.6%)</td>
</tr>
<tr>
<td>HTRU2</td>
<td>4500.7</td>
<td>666.7 (10.2%)</td>
</tr>
<tr>
<td>Shuttle</td>
<td>42217.7</td>
<td>4514.5 (10.7%)</td>
</tr>
</tbody>
</table>

expectation. In theory, the improved Introsellet Algorithm reduces stored distances from \(\frac{N(N-1)}{2}\) to \(2\%N(N-1)\) which is 4% of original method. Also, for every data point, SDPC maintains a list of all its neighbors, it takes another 4%. There are some other data and intermediate memory usage which weight more for small dataset.

7 RELATED WORK
Section 1 has mentioned a set of work closely related with the selection of cutoff distances in DPC. Many efforts are made to improve DPC from other aspects. Xie and others [27] design a new cluster assignment strategy to improve the robustness of DPC. An adaptive cluster center selection approach is proposed for automatical center tree structure and improve the decision graph by constructing a \(\Delta\)-tree.

Accelerating clustering algorithms have been a popular research topic. Strategies proposed for other clustering algorithms have inspired our work. Bound-based strategies are designed to speedup the GMM training [29]. Ideas of computation-reuse has been applied to K-Means [9, 14]. Delaunay diagram is used to accelerate 2-D spatial clustering [16].

8 CONCLUSION
This work proposes a set of techniques to harness computation reuse opportunities and to reduce memory usage of DPC. The new algorithm, SDPC, gives significant speedups and large memory savings across all datasets of various sizes and dimensions, without compromising the clustering quality. By removing the two major roadblocks for practical adoptions, SDPC offers an efficient and scalable drop-in replacement of DPC. (The code and datasets are shared at https://github.com/PICTureRG/SDPC.)

ACKNOWLEDGMENTS
This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-1525609 and CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES